UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

ASKELADDEN, L.L.C.,

Petitioner

v.

JABAA, L.L.C.

Patent Owner.

IPR2024-00916

U.S. Patent 7,480,637

DECLARATION OF JAMES L. MULLINS, Ph.D.

TABLE OF CONTENTS

I. INTRODUCTION 4

II. QUALIFICATIONS 4

III. LEGAL STANDARDS 7

IV. DOCUMENTS 14

A. Document: Exhibit 1015. Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001, First Printing October 2000. 499 pages ("Rescorla") 14

1. Authenticity 14

2. Public Accessibility 19

V. AVAILABILITY FOR CROSS-EXAMINATION 26

VI. RIGHT TO SUPPLEMENT 26

VII. CONCLUSION 27

TABLE OF EXHIBITS

Attachment 1	Curriculum Vitae of James L. Mullins Ph.D.
Exhibit 1015	Scanned version of Eric Rescorla, *SSL and TLS: Designing and Building Secure Systems* published by Addison-Wesley in 2001
Attachment A-1	Scanned excerpts of Rescorla from the U.S. Library of Congress
Attachment A-2	Scanned excerpts of Rescorla from the College of William and Mary's Swem Library
Attachment A-3	*WorldCat* entry for Rescorla (with Washington, D.C. as the geographical location)
Attachment A-4	*WorldCat* entry for Rescorla (with Virginia as the geographical location)
Attachment A-5	U.S. Library of Congress OPAC record for Rescorla
Attachment A-6	U.S. Library of Congress MARC record for Rescorla
Attachment A-7	Swem Library, College of William and Mary, OPAC record for Rescorla
Attachment A-8	Swem Library, College of William and Mary, MARC record for Rescorla
Attachment A-9	https://www.loc.gov/marc/bibliographic/bd650.html, "650 – Subject Added Entry-Topical Term (R)," last accessed May 13, 2024
Attachment A-10	https://www.oclc.org/bibformats/en/9xx.html, "9xx Fields", last accessed May 13, 2024

I, James L. Mullins, am above the age of majority and hereby declare under penalty of perjury:

I. INTRODUCTION

1. I have been retained by counsel for Petitioner Askeladden, L.L.C.. I am being paid my usual hourly fee of $275/hour. My compensation in no way depends on the content of my testimony or the outcome of the proceedings.

2. In this declaration, I opine on the authenticity of certain documents referenced herein and explain when and how these documents were disseminated or otherwise made available to the extent that persons interested and ordinarily skilled in the subject matter or art, exercising reasonable diligence, could have located the document.

II. QUALIFICATIONS

3. I am the Founder and Owner of the firm Prior Art Documentation Librarian Services, LLC at 205 St. Cuthbert, Williamsburg, VA 23188. Attached as Attachment 1 is a true and correct copy of my Curriculum Vitae describing my background and experience. Further information about my firm, Prior Art Documentation Librarian Services, LLC (PADLS), is available at www.priorartdoclib.com.

4. I am currently Dean Emeritus of Libraries and Esther Ellis Norton Professor Emeritus at Purdue University, 2018 – present. I was previously employed as follows:

- Dean of Libraries and Professor and Esther Ellis Norton Professor, Purdue University, West Lafayette, IN, 2004 – 2017;
- Assistant/Associate Director for Administration, Massachusetts Institute of Technology (MIT) Libraries, Cambridge, MA 2000 – 2004;
- University Librarian and Director, Falvey Memorial Library, Villanova University, Villanova, PA, 1996 – 2000;
- Director of Library Services, Indiana University South Bend, South Bend, IN, 1978 – 1996;
- Part-time Instructor, School of Library and Information Science, Indiana University, Bloomington, IN, 1979 – 1996;
- Associate Law Librarian, and associated titles, Indiana University School of Law, Bloomington, IN, 1974 –1978;
- Catalog Librarian, Assistant Professor, Georgia Southern College (now University), Statesboro, GA, 1973 – 1974.

5. During my more than forty-four years as an academic librarian and library science scholar, I have gained extensive experience with techniques and

systems for cataloging and indexing printed materials in the library setting, including catalog records and online library management systems ("LMS") built using Machine-Readable Cataloging ("MARC") standards.

6. Additionally, throughout my career, I have been an active researcher and writer. I am an author of numerous publications in the field of library science and have given presentations in library sciences at national and international conferences. I am a member of the American Library Association ("ALA"), where I served as the chair of the Research Committee of the Association of College and Research Libraries ("ACRL"). My service to ALA included service on the editorial board of the most prominent library journal, *College and Research Libraries*. I also served on the Standards Committee, College Section of the Association of College and Research Libraries, where I was instrumental in developing a re-issue of the *Standards for College Libraries* in 2000.

7. As an academic library administrator, I ensured that students were educated to identify, locate, assess, and integrate information garnered from research library resources. I have also facilitated the research of faculty colleagues either directly or through the provision of and access to the requisite print and/or digital materials and services at the universities where I worked.

8. For the purpose of this declaration, I have reviewed the documents and appendices referenced herein. These materials are records created in the

ordinary course of business by publishers, libraries, indexing services, and others in my field. From my years of experience, I am familiar with the process for creating many of these records, and I know that these records are created by people with knowledge of the information contained within the record. Further, these records are created with the expectation that researchers and other members of the public will use and rely on them. All materials cited in this declaration and its appendices are of a type that experts in my field would reasonably rely upon and refer to in forming their opinions. Although I have used headings and paragraphs to organize this declaration, my opinions are based on the contents of the entire declaration.

III. LEGAL STANDARDS

9. ***Authentication:*** I am informed by counsel and understand that an item is considered authentic if there is sufficient evidence to support a finding that the item is what it is claimed to be by the proponent of the item. I am also informed that authenticity can be established based on the properties of the item itself, such as the appearance, content, substance, internal patterns, or other distinctive characteristics of the item.

10. ***Public Accessibility:*** I am informed by counsel and understand that a given reference qualifies as "publicly accessible" if it were disseminated or otherwise made available such that persons interested in and ordinarily skilled in

the relevant subject matter could locate it through the exercise of reasonable diligence. I understand that one way that public accessibility can be shown is if a publication is cataloged and indexed by a library such that a person interested in the relevant subject matter could locate it using reasonable diligence. I also understand that there are other ways that a given reference may be deemed publicly accessible, including where that reference is cited by others in the field.

11. ***The Priority Date:*** I have been informed by counsel that for the purposes of this declaration I should consider evidence of public accessibility prior to December 2005, the priority date of U.S. Patent No. 7,480,637 ("the '637 patent"). As described further below, I have concluded that the reference discussed herein was publicly accessible, including to a Person of Ordinary Skill in the Art, prior to December 2005.

12. ***The Person of Ordinary Skill in the Art:*** I have been informed by counsel that a person of ordinary skill in the art at the time of the invention ("POSA") is a hypothetical person who is presumed to be familiar with the relevant field and its literature at the time of the invention. This hypothetical person is also a person of ordinary creativity, capable of understanding the scientific principles applicable to the pertinent field.

13. In addition, I have been informed by counsel that as relevant to these proceedings, the POSA would have the skills of one or more individuals with a

bachelor's degree in electrical engineering, computer engineering, computer science, or a related subject, and one to two years of work experience in electronic device design, secure electronic communications, or a related field. Less experience may be necessary with additional education (e.g., a Master's degree), and likewise, less education may be necessary with additional work experience (e.g., 5-6 years).

14. In my opinion, prior to December 2005, such a person would have been actively engaged in academic research and learning through study and practice relating to electrical engineering, computer engineering, computer science, electronic device design, and/or secure electronic communications.

15. ***Monograph Publications:*** Monograph publications are written on a single topic, presented at length, and distinguished from an article. They include books, dissertations, proceedings of a conference, and technical reports. A library typically creates a catalog record when the monograph is acquired by the library. First, it will search the Online Computer Library Center ("OCLC") to determine if a record has already been created by the Library of Congress or another OCLC institution.

16. If a record is found in OCLC, the record is downloaded into the library's management system that typically includes the online public access catalog ("OPAC"). Once the record is downloaded into the library's management

system, the library adds the library's identifier to the OCLC database so when a search is completed on WorldCat, the library will be indicated as possessing the publication. Once a record is created in a library's management system, the record is searchable and viewable through the library's OPAC, typically by author, title, and subject heading, at that library and from anywhere in the world through the internet by accessing that library's OPAC.

17. ***Indexing:*** A common way to search for a publication is to search for relevant information in an index of periodicals and other publications. Having found relevant material, the researcher will then normally obtain the material online, look for it in libraries, or purchase it from the publisher, a bookstore, a document delivery service, or other provider. Indexing services use a wide variety of controlled vocabularies to provide subject access and other means of discovering the content of documents. The formats in which these access terms are presented vary from service to service. Online indexing services commonly provide bibliographic information, abstracts, and full-text copies of the indexed publications, along with a list of the documents cited in the indexed publication. These services also often provide lists of publications that cite a given document. A citation of a document is evidence that the document was publicly available and in use by researchers no later than the publication date of the citing document.

18. ***MARC:*** The MARC standard is an industry-wide standard method of storing and organizing library catalog information. MARC practices have been consistent since the MARC format was developed by the Library of Congress in the 1960s, and by the early 1970s became the U.S. national standard for disseminating bibliographic data. By the mid-1970s, MARC format became the international standard, and persists through the present. A MARC-compatible library is one that has a catalog consisting of individual MARC records for each of its items. The underlying MARC format underpins the OPAC that is available to library users to locate a particular holding of a library. Today, MARC is the primary communications protocol for the transfer and storage of bibliographic metadata in libraries. The MARC practices discussed below were in place during the time frames relevant to the documents referenced herein.

19. Libraries worldwide use the machine-readable MARC format for catalog records. MARC-formatted records include a variety of subject access points based on the content of the document being cataloged. A MARC record for a particular work comprises several fields, each of which contains specific data about the work. Each field is identified by a standardized, unique, three-digit code corresponding to the type of data that follows.

20. When a MARC record is created by the Library of Congress or an OCLC member institution, the date of creation for that record is automatically

populated in the fixed field (008), with characters 00 through 05 in year, month, day format (YYMMDD).[1] Therefore, the MARC record creation date reflects the date on which the publication associated with the record was first cataloged. Thereafter, the local library's computer system may automatically update the date in field 005 every time the library updates the MARC record.

21. The 9XX MARC fields, which are not part of the standard MARC 21 format, were defined for use by the Library of Congress, processing or holding notes for a local library, and for internal OCLC use. See Attachment A-9. For example, the MARC 955 field is reserved for use by the Library of Congress to track the progress of a new acquisition from the time it is submitted for Cataloging in Publication (CIP) review until it is published and fully cataloged and, therefore, publicly available for use within the Library of Congress. Fields 901-907, 910, and 945-949 have been defined by OCLC for local use and will pass OCLC validation. Fields 905 or 910 are often used by an individual library for internal processing purposes, for example the date of cataloging and/or the initials of the cataloger.

[1] Some of the newer library catalog systems also include hour, minute, second (HHMMSS).

22. ***OCLC:*** OCLC is a not-for-profit worldwide consortium of libraries. Like MARC standards, OCLC's practices have been consistent since the 1970s through the present. Accordingly, the OCLC practices discussed below were in place during the timeframe discussed throughout this declaration. Among other services, OCLC and its members are responsible for maintaining the WorldCat database used by libraries throughout the world.

23. ***WorldCat[2]:*** WorldCat is the world's largest public online catalog, maintained by the OCLC, and built with the records created by the thousands of libraries that are members of OCLC. OCLC provides bibliographic and abstract information to the public based on MARC-compliant records through its OCLC WorldCat database. WorldCat requires no knowledge of MARC tags and codes and does not require a login or password. WorldCat is easily accessible through the World Wide Web to all who wish to search it.

24. ***Wisconsin TechSearch (WTS)[3]:*** WTS offers an array of article delivery and research services to individuals and organizations, including those not affiliated with the University of Wisconsin.

[2] https://www.worldcat.org/.

[3] https://wts.wisc.edu/.

IV. DOCUMENTS

A. Document: Exhibit 1015. Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001, First Printing October 2000. 499 pages ("Rescorla")

25. I have been asked to opine on the authenticity and public accessibility of a book authored by Eric Rescorla titled *SSL and TLS: Designing and Building Secure Systems* published by Addison-Wesley in 2001 comprised of 499 pages. ("Rescorla").

26. I have evaluated Rescorla by (1) assessing Exhibit 1015, provided to me by counsel; (2) obtaining and assessing scans of a copy of Rescorla owned by the Library of Congress provided to me, through several requests, by WTS on December 14, 2023, January 31, 2024, and February 12, 2024; (3) obtaining and assessing scans of a copy of Rescorla owned by the College of William and Mary Library that I obtained on January 31, 2024.

1. Authenticity

27. Attachment A-1 is a scan of portions of Rescorla obtained from the Library of Congress, as described above. It includes a scan of the bound front cover, spine, and back cover, with stamps that read, "Rescorla SSL and TLS" (Figure 1) and "TK5105.59.R47 2001 Copy 1" (Figure 2) as shown below.

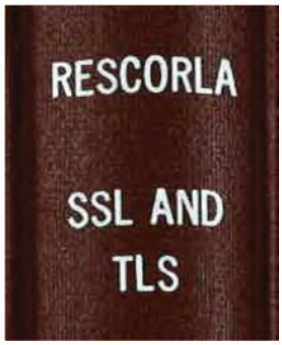


Figure 1: Attachment A-1 (binding spine)



Figure 2: Attachment A-1 (binding spine)

Attachment A-1 also includes a scan of the published cover; title page; and verso of the title page with a handwritten call number that reads: "LIBRARY OF CONGRESS NOV -7, 2000" (Figure 3) and a handwritten call number that reads "TK5105.59 .R47 2001 Copy 1" (Figure 4), as shown below.

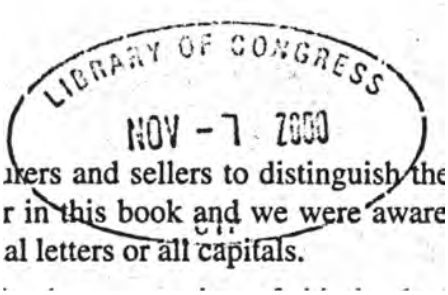


Figure 3: Attachment A-1 (verso of the title page)

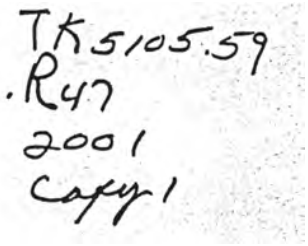


Figure 4: Attachment A-1 (verso of the title page)

Attachment A-1 also includes page xvii; pages 5-15; pages 17-24; pages 51-52; pages 81-88; page 124; pages 146-156; pages 202-204; page 230-235; pages 291-295; pages 301-202; pages 322-327; page 413; and pages 455-457.

28. All identifying characteristics of Attachment A-1, including the stamps and notations, are consistent with library practice and procedure that I have observed during my career as a professional librarian, specifically with those applied by the Library of Congress, and support a finding of authenticity. I have no cause for concern about the authenticity or accuracy of these attributes.

29. Rescorla was found within the custody of a library, the Library of Congress, one of the most likely locations for an authentic publication to be located.

30. **Attachment A-2** is a scanned version of Rescorla obtained from the College of William and Mary. It includes: a scan of the cover of Rescorla; a scan of the spine with a label that reads: "TK5105.59 .R47 2001" (Figure 5), as shown below.



Figure 5: Attachment A-2 (spine)

Attachment A-2 also includes a scan of the fore edge of Rescorla with a stamp that reads: "Library College of William and Mary" (Figure 6), as shown below.



Figure 6: Attachment A-2 (fore edge)

Attachment A-2 also includes a scan of the title page and verso of the title page with a handwritten call number that reads: "TK5105.59 .R47 2001" (Figure 7), as shown below.



Figure 7: Attachment A-2 (verso of the title page)

Attachment A-2 also includes pages 5-15; pages 17-24; pages 51-52; pages 81-88; page 124; pages 146-156; pages 202-204; page 230-235; pages 291-295; pages 301-202; pages 322-327 ; page 413; and pages 455-457.

31. All identifying characteristics of Attachment A-2, including the stamps and notations, are consistent with library practice and procedure that I have observed during my career as a professional librarian. Since I personally retrieved this copy of Rescorla from the Swem Library of the College of William and Mary (of which I serve on the Advisory Board), I have no cause for concern about the authenticity or accuracy of these attributes.

32. Rescorla was found within the custody of a library, the Swem Library, College of William and Mary, one of the most likely locations for an authentic publication to be located.

33. I compared Exhibit 1015 with Attachment A-1 and Attachment A-2 and found no differences. Given the above, I opine that Exhibit 1015 is a true and correct copy of Rescorla.

34. I opine that Rescorla is an authentic document since it was cataloged, indexed and held in research libraries, the Library of Congress and Swem Library, College of William and Mary, where, if authentic, it would logically be cataloged/indexed and publicly accessible. See Attachments A-6 and A-8, which are MARC records for Rescorla in each library.

2. Public Accessibility

35. **Attachment A-3** is a true and correct copy of the *WorldCat* entry for Rescorla. I obtained Attachment A-3 by completing a search on *WorldCat* on December 13, 2023 with Washington, D.C. as the geographical location. As shown in Attachment A-3, Rescorla is the document associated with this *WorldCat* entry, as verified by author: Eric Rescorla; title: *SSL and TLS: Designing and Building Secure Systems;* and ISBN: 0-201-61598-3.

36. Rescorla could have been located by searching for the title: *SSL and TLS: Designing and Building Secure Systems*; and/or by searching the subject headings: Computer network protocols; Computer networks security measures; among others.

37. When I searched *WorldCat* for holdings of Rescorla with the geographical designation of Washington, D.C., the Library of Congress was second on the list among the 274 libraries shown as holding Rescorla worldwide.

38. The search discussed above could have been performed anywhere in the world by anyone who accessed *WorldCat* through an OCLC member library and would have returned the Rescorla reference after its publication in the early 2000s and, as relevant to the '637 patent, well before December 2005.

39. **Attachment A-4** is a true and correct copy of the *WorldCat* entry for Rescorla. I obtained Attachment A-4 by completing a search on *WorldCat* on February 12, 2024 with Virginia as the geographical location. As shown in Attachment A-4, Rescorla is the document associated with this *WorldCat* entry, as verified by author: Eric Rescorla; title: *SSL and TLS: Designing and Building Secure Systems;* and ISBN: 0-201-61598-3.

40. Rescorla could have been located on *WorldCat* by searching for the title: *SSL and TLS: Designing and Building Secure Systems*; and/or by searching the subject headings: Computer network protocols; Computer networks security measures; among others.

41. When I searched *WorldCat* for holdings of Rescorla with the geographical designation of Virginia, William and Mary, was fifth on the list among the 274 libraries shown as holding Rescorla worldwide.

42. The search discussed above could have been performed anywhere in the world by anyone who accessed *WorldCat* through an OCLC member library in the early 2000s through December 2005.

43. **Attachment A-5** is a download I made from the Library of Congress OPAC on December 13, 2023. The document cataloged in this record is Rescorla as verified by the OPAC record fields including Personal Name: Rescorla, Eric, 1972- ; main title: *SSL and TLS: Designing and Building Secure Systems;* publisher: Addison-Wesley; publication date; 2001; and ISBN: 0201615983.

44. I compared the Library of Congress Classification (call number): TK5105.59.R47 2001 with that shown on the spine of the bound book (Figure 2) and handwritten on the verso of the title page (copyright) (Figure 4) in Attachment A-1, and it is the same except for the inclusion of Copy 1 which was a shelf indicator location and is not indicated in the OPAC or MARC records.

45. The date stamped on Attachment A-1 (Figure 3) indicates receipt of Rescorla by the Library of Congress on November 7, 2000, after which it would have been sent to the cataloging department for cataloging and indexing.

46. **Attachment A-6** is the MARC record I downloaded from the Library of Congress OPAC on December 13, 2023. The MARC format provides information about the processing of Rescorla by the Library of Congress. As mentioned above, the 9XX field in the MARC format is allocated to local libraries

to enter information specific to that library. The Library of Congress has reserved the 955 field to indicate date of receipt of the published book and cataloging. The MARC 955 Field in this record reads:

955 __ |a to ASCD pc21 06-27-00; jf03 06-27-00; jf08 06-27-00 to SL; jf12 to Dewey 06-28-00; aa07 06-28-00; CIP ver. pb18 to BCCD 12-28-00; copy 2 added jf16 to BCCD 02-06-01

Figure 8: Attachment A-6

47. The 955 field above indicates that Rescorla was received by the Library of Congress for creation of Cataloging in Publication (CIP) on 06-27-00 (June 27, 2000); after publication Rescorla was reviewed for verification of CIP information and addition of final cataloging information (pagination, etc.) by the Library of Congress on 12-28-00 (December 28, 2000). This is consistent with the date stamp above in Attachment A-1 indicating receipt by the Library of Congress on November 7, 2000.

48. Once Rescorla was fully cataloged/indexed on the later date, December 28, 2000, the record for Rescorla would have been immediately accessible in the Library of Congress OPAC. Rescorla, the book, would have been available for public access after labeling and transfer to the shelf, typically within 10 days, therefore, available by January 8, 2001 or within a few days thereof.

49. Once Rescorla was entered into the general collection of the Library of Congress, members of the public could access the book by having it brought to

either the Jefferson or Adams Reading Rooms. The collections of the Library of Congress are searchable by subject matter, author, or title such that a skilled researcher could find works in which they were interested. For example, a member of the public could have located a copy of Rescorla by searching for author: Eric Rescorla; the title: *SSL and TLS: Designing and Building Secure Systems*: and/or by searching the subject headings: *Computer networks - security measures; World Wide Web – Security measures*; and/or *Computer network protocols.* These subject headings are located in MARC fields 650 of Attachment A-6 and demonstrate how the work was indexed. See Attachment A-10, which describes subject headings.

50. Members of the public could read, study, and make notes about a selected work in the Library of Congress Reading Rooms. Further, members of the public were permitted to make photocopies of portions of the work while in the Reading Rooms. Accordingly, a copy of Rescorla was accessible to the general public when it was available at the Library of Congress on about January 8, 2001.

51. **Attachment A-7** is a download I made from the Swem Library of the College of William and Mary OPAC on February 1, 2024. The document cataloged in this record is Rescorla as verified by the OPAC record fields including Author: Rescorla, Eric, 1972- ; Title: *SSL and TLS: Designing and Building Secure Systems;* publisher: Addison-Wesley; publication date; 2001; and ISBN: 0201615983.

52. I compared the Library of Congress Classification (call number): TK5105.59.R47 2001 (Figure 4 from the Library of Congress) with that shown on the spine of the book (Figure 5 from the Swem Library) and handwritten on the verso of the title page (copyright) (Figure 7 from the Swem Library) and it is the same.

53. **Attachment A-8** is the MARC record I downloaded from the Swem Library of the College of William and Mary OPAC on February 1, 2024. The MARC format provides information about the processing of Rescorla by the Swem Library of the College of William and Mary. As mentioned above, the 9XX field in the MARC format is allocated to local libraries to enter information specific to that library. Based on the information contained in the 999 field of Attachment A-8, it is more likely than not that the Swem Library reserved the 999 field to indicate date of receipt of the published book and cataloging. The MARC 999 Field in this record reads toward the end: *$tBOOK $u4/30/2001 $xOCLC.,* as seen below:

```
999     ##$aTK5105.59 .R47 2001 $wLC $c1 $i11001546875 $d5/3/2008 $e4/30/2008 $lSTACKS $mSWEM $n2 $p$39.95 $rY $sY $tBOOK $u4/30/2001
```

Figure 9: Attachment A-8

54. The 999 field above indicates that Rescorla was received by the Swem Library, more likely than not, on April 30, 2001.

55. Once Rescorla was fully cataloged/indexed along with labeling and transfer to the shelf, typically within 10 days, Rescorla would have been available by May 10, 2001, or within a few days thereof.

56. Once Rescorla was entered into the general collection of the Swem Library OPAC, Rescorla could have been retrieved by searching for author: Eric Rescorla; the title: *SSL and TLS: Designing and Building Secure Systems*: and/or by searching the subject headings: *Computer networks - security measures; World Wide Web – Security measures*; and/or *Computer network protocols.* These subject headers are located in MARC fields 650 of Attachment A-8 and demonstrate how the work was indexed. See Attachment A-10, which describes subject headings.

B. Conclusion regarding Rescorla

57. I opine that Rescorla, Exhibit 1015, is an authentic document. I further conclude that Rescorla would have been publicly accessible through the Library of Congress within approximately 10 days after December 28, 2000 (January 8, 2001 or within a few days thereof) and at the Swem Library, College of William and Mary within approximately 10 days after April 30, 2001 (May 10, 2001 or within a few days thereof).

V. AVAILABILITY FOR CROSS-EXAMINATION

58. In signing this Declaration, I recognize that this Declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the U.S. Patent and Trademark Office. I also recognize that I may be subject to cross-examination in the case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

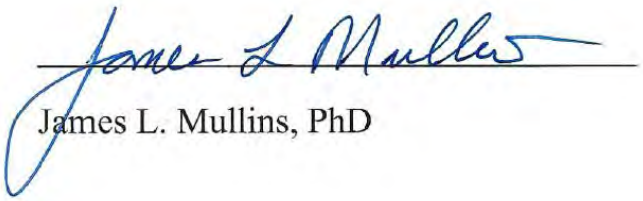
VI. RIGHT TO SUPPLEMENT

59. I reserve the right to supplement my opinions in the future to respond to any arguments that the Patent Owner raises and to take into account new information as it becomes available to me.

VII. CONCLUSION

60. I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the full knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the U.S. Code.

Dated: May 15, 2024



James L. Mullins, PhD

ATTACHMENT 1

JAMES L. MULLINS, PhD

Prior Art Documentation Librarian Services, LLC

205 Saint Cuthbert, Williamsburg, VA 23188

jlmullins@priorartdoclib.com

ph. 765 479 4956

Prior Art Documentation Librarian Services, LLC. (PADLS). Founded January 2018. As of May 2024, 1 have completed 147 declarations for 43 law firms; seven depositions were scheduled, three cancelled a few days prior. One was held in February 2019, a second in June 2023, and a third in March 2024.

Library Experience:

2018-present Dean Emeritus, Libraries & Norton Professor Emeritus, Purdue University

2011 - 2017 Dean of Libraries & Esther Ellis Norton Professor, Purdue University

2004 - 2011 Dean of Libraries & Professor, Purdue University, West Lafayette, IN

2000-2004 Assistant/Associate Director for Administration, MIT Libraries, Massachusetts Institute of Technology, Cambridge, MA.

1996-2000 University Librarian & Director, Falvey Memorial Library. Villanova University, Villanova, PA.

1978-1996 Director of Library Services, Indiana University South Bend.

1974-1978 Associate Librarian, Indiana University Bloomington, School of Law.

1973-1974 Instructor/Catalog Librarian. Georgia Southern College (now University).

Teaching Experience:

1977-1996 Associate Professor (part-time), School of Library and Information Science, Indiana University. Subjects taught: Cataloging, Management, and Academic Librarianship.

Education:

1972 - University of Iowa. Honors Bachelor of Arts, History, Religion & Political Science.

1973 - University of Iowa. Master of Arts in Library Science.

1984 - Indiana University. Doctor of Philosophy. Concentration: Academic Library Administration. Emphasis: Law Librarianship.

Awards and Recognition:

2017 Wilmeth Active Learning Center/Library of Engineering and Science, Grand Reading Room, was announced by President Mitch Daniels, Purdue University, that it would be re-named the James L. Mullins Reading Room to honor his leadership and reputation in the academic library profession. September 2017. Portrait unveiled December 2017.

2017 Distinguished Alumnus Award by the School of Informatics and Computing, Indiana University, Bloomington. Given June 25, 2017.

2016 Hugh C. Atkinson Memorial Award, jointly sponsored by the four divisions of the American Library Association (ALA), June 27, 2016.

2015 ACRL Excellence in University Libraries Award, April 23, 2015.

Named Esther Ellis Norton Professor of Library Science by Purdue Trustees, December 11, 2011.

International Review Panel to evaluate the University of Pretoria Library, February 20 – 24, 2011. Pretoria, South Africa.

Publications: (selected)

Designing the Academic Library, Chapter Contributor. Association of College and Research Libraries, American Library Association, 2022.

A Purdue Icon: Creation, Life, and Legacy, edited by James L. Mullins, Founder's Series, Purdue University Press, 138pp., August 2017.

"The policy and institutional framework." In *Research Data Management, Practical Strategies for Information Professionals,* edited by Ray, J M. Purdue University Press, pp.25-44, 2014.

"DataCite: linking research to data sets and content." In Benson, P and Silver, S. *What Editors Want: An Author's Guide to Scientific Journal Publishing*. University of Chicago Press, pp. 21-23, December 2012.

"Library Publishing Services: Strategies for Success," with R. Crow, O. Ivins, A. Mower, C. Murray-Rust, J. Ogburn, D Nesdill, M. Newton, J. Speer, C. Watkinson. *Scholarly Publishing and Academic Resources Coalition (SPARC),* version 2.0, March 2012.

"The Changing Definition and Role of Collections and Services in the University Research Library." *Indiana Libraries,* Vol 31, Number 1 (2012), pp.18-24.

"Are MLS Graduates Being Prepared for the Changing and Emerging Roles that Librarians must now assume within Research Libraries?" *Journal of Library Administration.* Volume 52, Issue 1, 2012, p. 124-132

Baykoucheva, Svetla. What Do Libraries Have to Do with e-Science?: An Interview with James L. Mullins, Dean of Purdue University Libraries. Chem. Inf. Bull. [Online] 2011, 63 (1), 45-49. http://www.acscinf.org/publications/bulletin/63-1/mullins.php (accessed Mar 16, 2011).

"The Challenges of e-Science Data-set Management and Scholarly Communication for Domain Sciences and Technology: a Role for Academic Libraries and Librarians," chapter in, *The Digital Deluge: Can Libraries Cope with e-Science?"* Deanna B. Marcum and Gerald George, editors, Libraries Unlimited/Teacher Ideas Press, 2009. (a monograph publication of the combined proceedings of the KIT/CLIR proceedings).

"Bringing Librarianship to e-Science," *College and Research Libraries.* vol. 70, no. 3, May 2009, editorial.

"The Librarian's Role in e-Science" *Joho Kanri (Journal on Information Processing and Management),* Japan Science and Technology Agency (formerly Japan Information Center of Science and Technology), Tokyo, Japan. Translated into Japanese by Taeko Kato. March 2008.

The Challenge of e-Science Data-set Management to Domain Sciences and Engineering: a Role for Academic Libraries and Librarians," KIT (Kanazawa Institute of Technology)/CLIR (Council of Library and Information Resources) International Roundtable for Library and Information Science, July 5-6, 2007. Developments in e-science status quo and the challenge, The Japan Foundation, 2007.

"An Administrative Perspective," Chapter 14, *Proven Strategies for Building an Information Literacy Program*, Susan Curzon and Lynn Lampert, editors, Neal-Schuman Publishers, Inc., New York, 2007. pp. 229-237.

Library Management and Marketing in a Multicultural World, proceedings of the IFLA Management and Marketing (M&M) Section, Shanghai, China, August 16-17, 2006, edited. K.G. Saur, Munchen, Germany, June 2007. 390 pp.

Top Ten Assumptions for the Future of Academic Libraries and Librarians: a report from the ACRL Research Committee, with Frank R. Allen and Jon R. Hufford. *College & Research Libraries,* April 2007, vol.68, no.4. pp.240-241, 246.

To Stand the Test of Time: Long-term Stewardship of Digital Data Sets in Science and Engineering. A report to the National Science Foundation from the ARL Workshop on New Collaborative Relationships: the Role of Academic Libraries in the Digital Data Universe. September 26-27, 2006, Arlington, VA. p.141. http://www.arl.org/bm~doc/digdatarpt.pdf

"Enabling Interaction and Quality in a Distributed Data DRIS," *Enabling Interaction and Quality: Beyond the Hanseatic League*. 8th International Conference on Current Research Information Systems, with D. Scott Brandt and Michael Witt. Promoted by euro CRIS. Leuven University Press, 2006. pp.55-62. Editors: Anne Garns Steine Asserson and Eduard J. Simons.

"Standards for College Libraries, the final version approved January 2000," prepared by the ACRL College Libraries Standards Committee (member), *C&RL News*, March 2000, p.175-182.

"Standards for College Libraries: a draft," prepared by the ACRL College Libraries Section, Standards Committee (member), *C&RL News*, May 1999, p. 375-381.

"Statistical Measures of Usage of Web-based Resources," *The Serials Librarian,* vol. 36, no. 1-2 (1999) p. 207-10.

"An Opportunity: Cooperation between the Library and Computer Services," in *Building Partnerships: Computing and Library Professionals.* Edited by Anne G. Lipow and Sheila D. Creth. Berkeley and San Carlos, CA, Library Solutions Press, 1995. p. 69-70.

"Faculty Status of Librarians: A Comparative Study of Two Universities in the United Kingdom and How They Compare to the Association of College and Research Libraries Standards, " in *Academic Librarianship, Past, Present, and Future: a Festschrift in Honor of David Kaser.* Englewood, Colorado; Libraries Unlimited, 1989. p. 67-78. Review in: *College & Research Libraries,* vol. 51, no. 6. November 1990, p. 573-574.

Presentations: (Representative)

"How Long the Odyssey? Transitioning the Library and Librarians to Meet the Needs and Expectations of the 21st Century University," David Kaser Lecture, School of Informatics & Computing, Indiana University, Bloomington, IN, November 16, 2015.

Presentation at University of Cape Town, Cape Town, South Africa, August 20, 2015.

"The Challenge of Discovering Science and Technology Information," Moderator, International Federation of Library Associations (IFLA) Science and Technological Libraries Section Program, Cape Town, South Africa, August 18, 2015.

"An Odyssey in Data Management: Purdue University," International Federation of Library Associations (IFLA) Research Data Management: Finding Our Role – A program of the Research Data Alliance, Cape Town, South Africa, August 17, 2015.

Presentation at University of Pretoria, Pretoria, South Africa, August 11, 2015.

Co-Convener with Sarah Thomas, Harvard University, at the Harvard Purdue Symposium on Data Management, Harvard University, Cambridge, MA, June 15-18, 2015.

"Strategic Communication," panel discussion on the Director's role and perspective on library communications at Committee on Institutional Cooperation (CIC) Center for Library Initiatives (CLI) Annual Conference, University of Illinois Urbana-Champaign, May 20, 2015.

"Issues in Data Management," panel discussion moderated by Catherine Woteki, United States Undersecretary for Research, Education & Economics at 20th Agriculture Network Information Collaborative (AgNIC) Annual Meeting in the National Agricultural Library, Beltsville, MD, May 6, 2015.

"Active learning/IMPACT & the Active Learning Center at Purdue University," Florida Institute of Technology, Melbourne, FL, February 11, 2015.

"Science+art=creativity: libraries and the new collaborative thinking," panel moderator, International Federation of Library Associations (IFLA) 80th General Conference and Assembly, Lyon, France, August 19, 2014.

"Purdue University The Active Learning Center—A new concept for a library," Association of University Architects 59th Annual National Conference, University of Notre Dame, South Bend, IN, June 23, 2014.

"Big Data & Implications for Academic Libraries," keynote speaker, Greater Western Library Alliance (GWLA) Cyber-infrastructure Conference, Kansas City, MO, May 28, 2014.

"Research Infrastructure," panel moderator, Association of Research Libraries (ARL) 164th Membership Meeting, Ohio State University, Columbus, OH, May 7, 2014.

"An Eight Year Odyssey in Data Management: Purdue University," International Association of Scientific and Technological University Libraries (IATUL) 2013 Workshop Research Data Management: Finding Our Role, University of Oxford, UK, December 2013.

"Purdue University Libraries & Press: from collaboration to integration," Ithaka Sustainable Scholarship, The Evolving Digital Landscape: New Roles and Responsibilities in Higher Education, libraries as publishers, New York, New York, October 2013.

"Tsinghua and Purdue: Research Libraries for the 21st Century," Tsinghua University, Tsinghua, China, August 2013.

"Purdue Publishing Experience in the Libraries Publishing Coalition," Association of American University Presses Annual Meeting, Press-Library Coalition Panel, Boston, Massachusetts, June 21, 2013.

"Indiana University Librarians Day: Purdue University Libraries Ready for the 21st Century," Indiana University Purdue University Indianapolis (IUPUI), June 7, 2013.

"Purdue University Libraries and Open Access; CNI Project Update," Coalition for Networked Information, San Antonio, TX, April 5, 2013.

Memorial Resolution, honoring Joseph Brannon, to the Board of the Association of College & Research Libraries, Seattle, WA, January 2013.

"An overview of sustaining e-Science collaboration in an Academic Research Library—the Purdue experience," Duraspace e-Science Institute webcast, October 17, 2012.

"The Role of Libraries in Data Curation, Access, and Preservation: an International Perspective, " Panel Moderator, 78th General Conference and Assembly, International Federation of Library Associations, Helsinki, Finland, August 15, 2012.

"21st Century Libraries," moderator of First Plenary Session, International Association of Technological University Libraries 33rd Annual Conference, Singapore, June 4, 2012.

"Planning for New Buildings on Campus," panel presenter, University of Calgary Building Symposium on Designing Libraries for the 21st Century, Calgary, Alberta, Canada, May 17, 2012.

"Data Management and e-Science, the Purdue Response." Wiley-Blackwell Executive Seminar-2012, Washington, DC, March 23, 2012.

"An overview of Sustaining e-Science Collaboration in Academic Research Libraries and the Purdue Experience." Leadership & Career Development Program Institute, Association of Research Libraries (ARL). Houston, TX, March 21, 2012.

"An overview of Data Activities at Purdue University in response to Data Management Requirements." Coalition for Academic Scientific Computation (CASC). Arlington, VA, September 8, 2011.

"Getting on Track with Tenure," Association of College and Research Libraries (ACRL) Research Program Committee. Washington, DC, June 26, 2011.

"Integration of the Press and Libraries Collaboration to Promote Scholarly Communication," Association of Library Collections & Technical Services (ALCTS) Scholarly Communication Interest Group – American Library Association, New Orleans, Louisiana, June 25, 2011.

"Cooperation for improving access to scholarly communication," with N. Lossau (Germany), C. Mazurek (Poland), J. Stokker (Australia), panel moderator and presenter, Second Plenary Session, International Association of Scientific and Technological University Libraries (IATUL) 32nd Conference 2011, Warsaw, Poland. May 29-June 2, 2011.

"Riding the Wave of Data," STM Annual Spring Conference 2011. Trailblazing & transforming scholarly publishing **2011**. Washington, D.C., April 28, 2011.

"Confronting old assumptions to assume new roles: physical and operational integration of the Press and Libraries at Purdue University," keynote speaker, 2011 BioOne Publishers & Partners Meeting. Washington, D. C., April 22, 2011.

"Are MLS Graduates Being Prepared for the Changing and Emerging Roles that Librarians must now assume within Research Libraries?" University of Oklahoma Libraries Seminar, March 4, 2011, Oklahoma City, Oklahoma.

"The Future Role of University Librarians," the University of Cape Town, South Africa, February 25, 2011.

"New Roles for Librarians: the Application of Library Science to Scientific/Technical Research – Purdue University – a case study. International Council for Science and Technology (ICSTI); Ottawa, Canada. June 9, 2009.

"Reinventing Science Librarianship: Models for the Future," Association of Research Libraries / Coalition for Networked Information. October 16-17th, 2008, Arlington, VA. Moderator and convener of Data Curation: Issues and Challenges.

"Practical Implementation and Opportunities Created at Purdue University," African Digital Curation Conference, Pretoria, South Africa, (live video transmission), February 12, 2008.

Keynote speaker. "*Scholarly Communication & Academe: The Winter of Our Discontent,"* XXVII Charleston Conference on Issues in Book and Serial Acquisition, Charleston, South Carolina. November 8, 2007.

Keynote speaker. "*Enabling Access to Scientific & Technical Data-sets in e-Science: a role for Library and Archival Sciences,"* Greater Western Library Alliance (GWLA), Tucson, Arizona. September 17, 2007. A meeting of library directors and vice presidents for research of member institutions.

"*The Challenge of e-Science Data-set Management to Domain Sciences and Engineering: a Role for Academic Libraries and Librarians,"* KIT (Kanazawa Institute of Technology)/CLIR (Council of

Library and Information Resources) International Roundtable for Library and Information Science, July 5-6, 2007. Invited to participate by the Deputy Librarian of Congress.

International Association of Technological University Libraries (IATUL), Stockholm, Sweden. June 8, 2007**.** Invited paper**,** *Enabling International Access to Scientific Data-sets: creation of the Distributed Data Curation Center (D2C2).*

"A New Collaboration for Librarians: The Principles of Library and Archival Sciences Applied to the Curation of Datasets," Symposium of the Libraries and the College of Engineering, University of Louisville, April 6, 2007.

"Purdue University Libraries: Through Pre-eminent Innovation and Creativity, Meeting the Challenges of the Information Age," Board of Trustees, Purdue University, February 15, 2007.

ARL Workshop on New Collaborative Relationships: The Role of Academic Libraries in the Digital Data Universe, September 26-27, 2006, Arlington, VA. Invited participant.

NARA and SDSC: A partnership. A panel before the National Science Foundation, June 27, 2006. Arlington, VA. Invited participant.

"Kaleidoscope of Scientific Literacy: fusing new connections," with Diane Rein, American Library Association, Association of College and Research Libraries, Science & Technology Section, Annual Conference, New Orleans, June 26th, 2006.

"Leadership for Learning: Building a Culture of Teaching in Academic Libraries – an administrative perspective," American Library Association, Association of College and Research Libraries, Instruction Section, Annual Conference, New Orleans, June 25th, 2006.

"Building an interdisciplinary research program in an academic library:
how the Libraries'associate dean for research makes a difference at Purdue University," International Association of Technological University Libraries (IATUL), Porto, Portugal, May 23rd, 2006.

"Enabling Interaction and Quality in a Distributed Data DRIS," *Enabling Interaction and Quality: Beyond the Hanseatic League*. 8th International Conference on Current Research Information Systems, with D. Scott Brandt and Michael Witt. Promoted by euro CRIS, Bergen, Norway, May 12th, 2006, Brandt, and Witt presented in person

"Interdisciplinary Research," with D. Scott Brandt, Coalition for Networked Information (CNI) Spring Meeting: Project Briefing, Washington, D.C., April 3rd, 2006.

"An Interview with Purdue's James Mullins," a podcast submitted by Matt Pasiewicz, on *Educause Connect,* http://connect.educause.edu/James_L_Mullins_Interview_CNI_2005

"Managing Long-Lived Digital Data-sets and their Curation: Interdisciplinary Policy Issues," Managing Digital Assets Forum, Association of Research Libraries (ARL), Washington, D.C., October 28th, 2005.

"The Odyssey of a Librarian." Indiana Library Federation (ILF), District 2 Meeting, South Bend, Indiana. October 4th, 2005.

"New College Library Standards," Standards Committee Presentation, ALA, Chicago, July 7, 2000.

SUNY Library Directors, Lake George, New York. "*The College Library Standards: a Tool for Assessment."* April 5, 2000.

Tri-State College Library Association, *Finding You Have Talents You Never Knew You Had,* Penn State Great Valley, March 25, 2000.

Using Web Statistics, American Library Association, New Orleans, June 24, 1999.

Keynote speaker at the JSTOR Workshop, January 29, 30, 1999. University of Pennsylvania, Philadelphia, PA.

"The New Standards for Electronic Resources Statistics," Society of Scholarly Publishers, Washington, D.C., September 17, 1998.

"Evaluating Online Resources: Now that you've got them what do you do?," joint presenter with Chuck Hamaker, LSU, at the NASIG Conference, Boulder, Colorado. June 1998.

"What Employers Are Looking for in New Librarians?" Pennsylvania Library Association, Philadelphia. September 26, 1997.

"The Theory of Matrix Management" panel presentation of the Comparative Library Organization Committee of the Library Organization and Management Section of the Library Administration and Management Association, a division of the American Library Association, Annual Meeting, Chicago, June 24, 1990.

Professional Involvement: (summary of recent emphasis)

The focus for my professional involvement and research has moved recently toward managing massive data-sets. This has resulted in working with faculty in the sciences and technology to determine how librarians can collaborate in managing, curating, and preserving data-sets for future access and documentation. This has included various speaking opportunities as well as participation in planning with the National Science Foundation (NSF) on ways in which librarians can be integrated more completely into the funded research process. Participation in the Kanazawa Institute of Technology/Council of Library Resources Roundtable was particularly rewarding and provided new opportunities to share with international colleagues the issues surrounding data-set management. I was

the champion for the creation of the Distributed Data Curation Center (D2C2) at Purdue University (http://d2c2.lib.purdue.edu/)

Throughout my career, beginning with my dissertation, I have been actively involved with assessing and evaluating libraries. In the fall of 1999, I contacted twenty-two academic library directors to determine whether the need was also felt by others. The response was overwhelmingly affirmative. This resulted in a meeting at ALA Midwinter, January 2000. A formal meeting followed at Villanova University in April 2000. As convener, I helped to form the University Libraries Group (ULG), modeled after the Oberlin Group for college libraries. The ULG is made up of university libraries that support diverse wide-ranging programs through doctoral level and have a level of support that places them in the top tier of academic institutions. A few of the member libraries, along with Villanova, are William and Mary, Wake Forest, Lehigh, Carnegie-Mellon, Tufts, Marquette, Miami of Ohio, and Southern Methodist.

In 1994 appointed to the Standards Committee, College Section, Association of College and Research Libraries. During the next six years, the Committee concentrated on changing the focus of the standards from quantitative analysis of input and output factors to emphasis on assessment of the outcome. Culmination of the work was a re-issue of the *Standards for College Libraries* in 2000. The knowledge gained through my work experience enabled me to formulate the changes needed in the standards. This work allowed for close collaboration with accrediting agencies, both professional and regional.

During this same time another focus emerged, the impact of digital resources. Through my work on the JSTOR Statistics Task Force, standards were developed on the collection of use of electronic databases. This Standard was later adopted in 1998 by the International Consortium of Library Consortia (ICOLC).

In 2002, the American Library Association appointed me to serve as the liaison to the Marketing and Management Section of the International Federation of Library Associations (IFLA).

Professional Service: (representative list)

Nominations Committee, Association of Research Libraries (ARL), 2016.

Steering Committee, Scholarly Publishing and Academic Resources Coalition (SPARC), 2016 – 2017.

"Excellence in Library Services," Chair, Review Team, University of Hong Kong, Hong Kong, August 24-27, 2015.

Chair, Management Advisory Board, 2015-2017; Member, Scientific Advisory Board, arXiv, Cornell University, 1/1/2013 – present.

Advisory Board for the Wayne State University School of Library and Information Science, July 2012 – present.

Advisory Board for Microsoft Academic Search, 2012 – 2015. Redmond, WA.

Transforming Research Libraries, a Strategic Direction Steering Committee of the Association of Research Libraries (ARL), 2012-2015.

Science and Technology section, representing ARL, International Federation of Library Associations (IFLA), Chair, 2013 – 2017; Member, 2011 to present.

Member of University of Pretoria, South Africa, Library Review Committee. August 2013.

Co-chair, Local Arrangements Planning Committee for 2013 Conference, Association of College and Research Libraries (ACRL), a division of the American Library Association (ALA).

Association of Research Libraries Leadership & Career Development Program Mentor, 2011-2017.

e-Science Task Force, Association of Research Libraries. July 2006 – present. Chair, October 2011 – October 2012.

Board of Directors, International Association of Technological University Libraries (IATUL). January 2008 – December 2014.

Midwest Collaborative for Library Services (MCLS); Board Member, October 2010 – December 2012.

Chair, Library Directors, Committee on Institutional Cooperation (CIC), July 2010 – June 2012.

Board of Directors, Association of Research Libraries (ARL); October 2008 – October 2011.

Scholarly Communication Steering Committee, Association of Research Libraries (ARL) 2008-2011.

Editorial Board, *College and Research Libraries*, Association of College and Research Libraries, American Library Association. January 2008 – December 2014.

Chair, Organizing Committee for IATUL Conference 2010, June 21-24, 2010, Purdue University, West Lafayette, Indiana/Chicago, Illinois.

Conference Planning Committee for National Conference of the Association of College and Research Libraries, 2009, Seattle, Washington.

Research Committee, Association of College and Research Libraries, ACRL, division of ALA. 2002-2007, chair, 2005-2007.

Association of Research Libraries, Search and Screen Committee, Executive Director. March – January 2008.

Center for Research Libraries, Board of Directors. April 2006 – April 2012.

Academic Libraries of Indiana, Board of Directors, 2004 – present. Vice-president, 2005-2007. President, 2007- 2009.

ALA Representative to the International Federation of Library Associations (IFLA), Marketing and Management (M&M) Section, initial term 2003-2007, re-appointed for second term, 2007-2011.
Invited to represent Research Libraries at the ACRL/3M Wonewok Retreat to assess Marketing of Academic Libraries, October 2002.
Hugh A. Atkinson Award Committee, LAMA Representative, ALA, 2001-2005.
Program Committee, Library Administrators and Management Association (LAMA), a division of ALA. 1996-2001.
ACRL, Standards and Accreditation Committee, a division of ALA. Liaison to RBMS Section of ACRL. 1997-2002.
Elected to the Executive Committee of LAMA, LOMS, a division of the American Library Association, 1998-2000. Nominated as Chair/Elect for 2003 – 2005.
Columbia University Press Advisory Committee. 1996 - 2000.
LITA/LAMA Conference Evaluation Committee, Pittsburgh, Pennsylvania, October 1996.
"New Learning Communities," Coalition for Networked Information, Indianapolis. November 19-21, 1995. Facilitator for invitational, national conference committed to developing collaborative learning and teaching techniques, involving librarians.
Planning Committee-Evaluation. LITA/LAMA 1996 Conference, Pittsburgh. This first conference, to be held jointly between two divisions of ALA, will focus on new technologies within libraries.
Indiana Cooperative Library Services Authority (InCoLSA), elected to Executive Committee, April 1991, served as President in 1993-94. InCoLSA is a statewide network of academic, public, school, and special libraries that supports library cooperation for cataloging, interlibrary loan, collection development and application of new technologies.
Governor’s Conference on Libraries and Information Services. Served on Planning Committee, Academic Libraries Representative, appointed by the Governor to represent academic libraries in Indiana, Chair, Finance Committee, April 1989-July 1991.
Indiana Library Endowment Foundation Board, 1984-92. Charter Member, 1984, President, 1988-1992. 2004-2005.

University Service: (Summary)

Served on search and screen committees for senior positions including chancellor, dean, and directors; most recently I have been asked to serve on the search committee for the provost of Purdue University. At MIT service included the Library Council & appointment to the Administrative Council by President

Vest, 2001-2003 & Member of the Faculty Committee on the Library System. At Purdue appointed by the President to the Search Committee for the Provost, October 2007 to May 2008; member of the Capital Projects Committee, and IT Operational Oversight Committee as senior academic dean, 2008-2014.

Global Council, Global Policy Institute, 2012 – 2016.

Academic Program Excellence and Rankings (APER) project team, 2014.

Representative of the Academic Deans on the Re-engineering Business Operations, Purdue University, 2016 –

Academic Deans Council chaired by Provost – 2004 – 2017.

University Promotion and Tenure Committee – 2006 – 2017.

"Outstanding Team Award, Electronic Reserve Project," served as Chair, recognition awarded by the President of Villanova University to one team who made an outstanding contribution to the operations of the University, selected by a committee of administrators, faculty, and staff. Awarded September 9, 1999.

Nominated for the IUSB Lundquist Award, 1995 & 1996. The Lundquist award is given to faculty who have "exhibited excellence in teaching, scholarly or artistic achievement, and diversified relevant service..."

EXHIBIT 1015

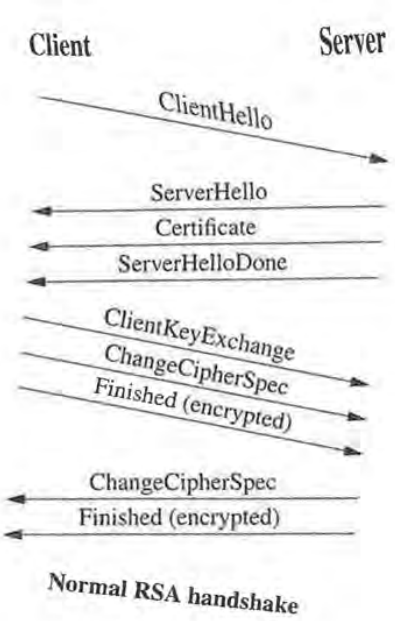
SSL and TLS

Designing and Building Secure Systems

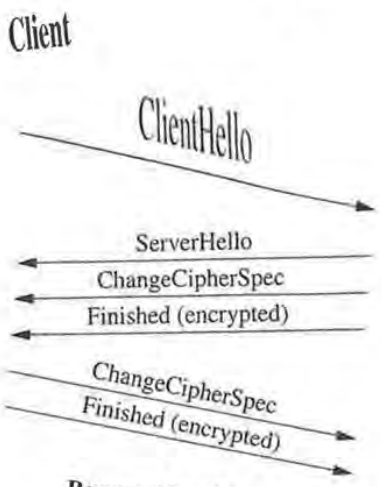


Eric Rescorla

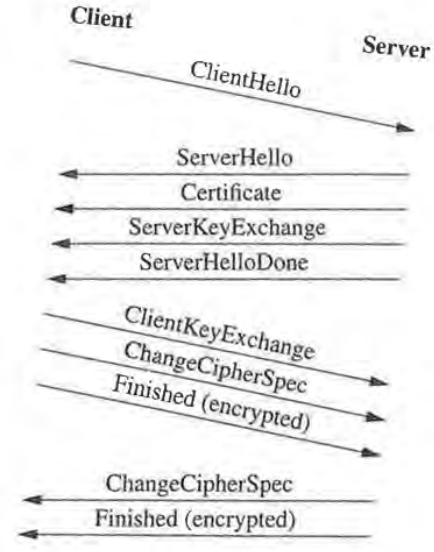
Ex.1024
Askeladden, L.L.C. / Page 43 of 788



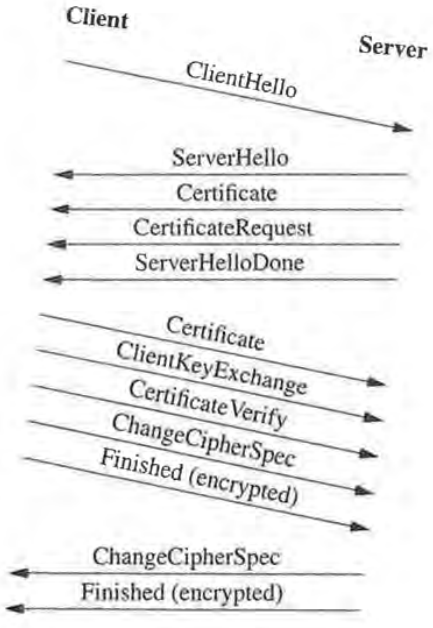
Normal RSA handshake



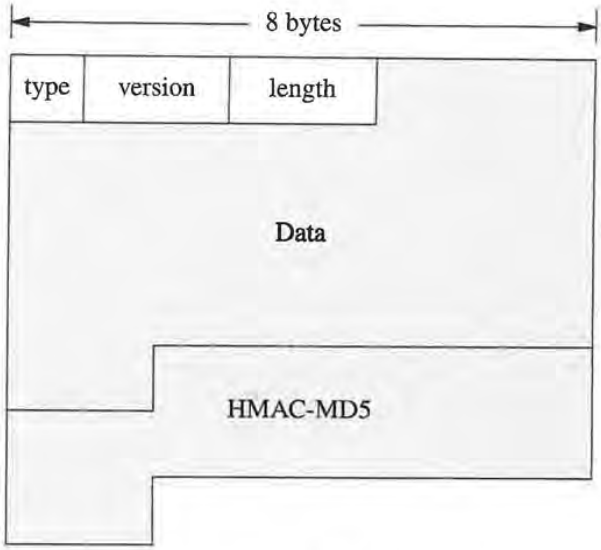
Resumed handshake



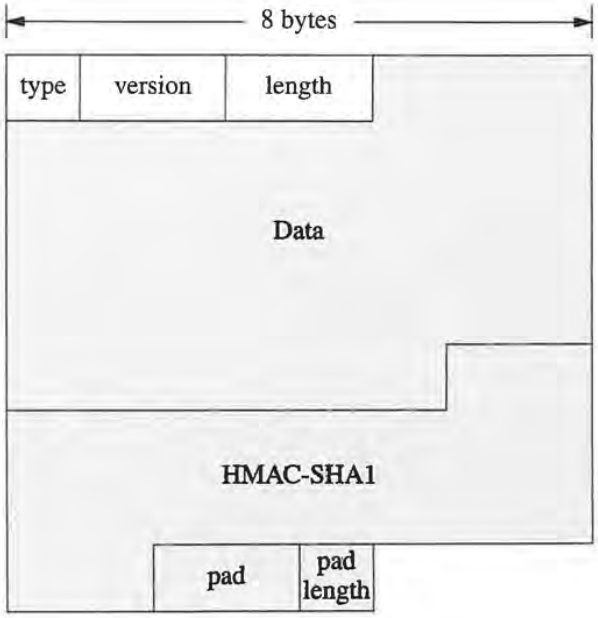
Ephemeral RSA (or DH) handshake



Client Authentication handshake



Record with stream cipher and HMAC-MD5



Record with block cipher and HMAC-SHA1

SSL and TLS

SSL and TLS

Designing and Building Secure Systems

Eric Rescorla



Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where these designations appear in this book and we were aware of a trademark claim, the designations have been printed in initial capital letters or all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained herein.

Copyright © 2001 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit us on the Web at www.awl.com/cseng

Library of Congress Cataloging-in-Publication Data

Rescorla, Eric, 1972-
SSL and TLS: designing and building secure systems / Eric Rescorla
p. cm.
Includes bibliographical references and index.
ISBN 0-201-61598-3 (alk. paper)
1. Computer networks—Security measures. 2. World Wide Web—Security measures
3. Computer network protocols. I. Title.
TK5105.59.R47 2000
05.8—dc21 00-057593

ISBN 0-201-61598-3

Text printed on recycled paper.
1 2 3 4 5 6 7 8 9 10-CRS-04 03 02 01 00
First printing, October 2000

For Lisa

Contents

Preface		**xvii**
Chapter 1.	**Security Concepts**	**1**
1.1	Introduction	1
1.2	The Internet Threat Model	1
1.3	The Players	2
1.4	The Goals of Security	3
1.5	Tools of the Trade	5
1.6	Putting It All Together	15
1.7	A Simple Secure Messaging System	16
1.8	A Simple Secure Channel	17
1.9	The Export Situation	24
1.10	Real Cryptographic Algorithms	25
1.11	Symmetric Encryption: Stream Ciphers	26
1.12	Symmetric Encryption: Block Ciphers	27
1.13	Digest Algorithms	32
1.14	Key Establishment	32
1.15	Digital Signature	36
1.16	MACs	38
1.17	Key Length	39
1.18	Summary	40

Ex.1024
Askeladden, L.L.C. / Page 53 of 788

Chapter 2.	**Introduction to SSL**	**43**
2.1	Introduction	43
2.2	Standards and Standards Bodies	43
2.3	SSL Overview	44
2.4	SSL/TLS Design Goals	44
2.5	SSL and the TCP/IP Suite	46
2.6	SSL History	47
2.7	SSL for the Web	51
2.8	Everything over SSL	52
2.9	Getting SSL	53
2.10	Summary	55
Chapter 3.	**Basic SSL**	**57**
3.1	Introduction	57
3.2	SSL Overview	57
3.3	Handshake	58
3.4	SSL Record Protocol	61
3.5	Putting the Pieces Together	62
3.6	A Real Connection	63
3.7	Some More Connection Details	65
3.8	SSL Specification Language	68
3.9	Handshake Message Structure	70
3.10	Handshake Messages	71
3.11	Key Derivation	82
3.12	Record Protocol	88
3.13	Alerts and Closure	91
3.14	Summary	94
Chapter 4.	**Advanced SSL**	**95**
4.1	Introduction	95
4.2	Session Resumption	96
4.3	Client Authentication	96
4.4	Ephemeral RSA	97
4.5	Rehandshake	99
4.6	Server Gated Cryptography	100
4.7	DSS and DH	102
4.8	Elliptic Curve Cipher Suites	103
4.9	Kerberos	104
4.10	FORTEZZA	104
4.11	The Story So Far	106

4.12	Session Resumption Details	106
4.13	Client Authentication Details	108
4.14	Ephemeral RSA Details	112
4.15	SGC Details	115
4.16	DH/DSS Details	124
4.17	FORTEZZA Details	126
4.18	Error Alerts	128
4.19	SSLv2 Backward Compatibility	135
4.20	Summary	137
Chapter 5.	**SSL Security**	**139**
5.1	Introduction	139
5.2	What SSL Provides	139
5.3	Protect the master_secret	140
5.4	Protect the Server's Private Key	140
5.5	Use Good Randomness	140
5.6	Check the Certificate Chain	141
5.7	Algorithm Selection	142
5.8	The Story So Far	142
5.9	Compromise of the master_secret	142
5.10	Protecting Secrets in Memory	145
5.11	Securing the Server's Private Key	146
5.12	Random Number Generation	154
5.13	Certificate Chain Verification	156
5.14	Partial Compromise	162
5.15	Known Attacks	166
5.16	Timing Cryptanalysis	167
5.17	Million Message Attack	168
5.18	Small-Subgroup Attack	170
5.19	Downgrade to Export	171
5.20	Summary	173
Chapter 6.	**SSL Performance**	**175**
6.1	Introduction	175
6.2	SSL Is Slow	175
6.3	Performance Principles	176
6.4	Cryptography Is Expensive	179
6.5	Session Resumption	181
6.6	Handshake Algorithm and Key Choice	182
6.7	Bulk Data Transfer	183
6.8	Basic SSL Performance Rules	184

6.9	The Story So Far	184
6.10	Handshake Time Allocation	184
6.11	Normal RSA Mode	186
6.12	RSA with Client Authentication	188
6.13	Ephemeral RSA	189
6.14	DSS/DHE	191
6.15	DSS/DHE with Client Authentication	194
6.16	Performance Improvements with DH	195
6.17	Record Processing	197
6.18	Java	199
6.19	SSL Servers under Load	202
6.20	Hardware Acceleration	204
6.21	Inline Hardware Accelerators	206
6.22	Network Latency	209
6.23	The Nagle Algorithm	212
6.24	Handshake Buffering	214
6.25	Advanced SSL Performance Rules	216
6.26	Summary	217
Chapter 7.	**Designing with SSL**	**219**
7.1	Introduction	219
7.2	Know What You Want to Secure	220
7.3	Client Authentication Options	221
7.4	Reference Integrity	222
7.5	Inappropriate Tasks	224
7.6	Protocol Selection	224
7.7	Reducing Handshake Overhead	227
7.8	Design Strategy	227
7.9	The Story So Far	228
7.10	Separate Ports	228
7.11	Upward Negotiation	229
7.12	Downgrade Attacks	230
7.13	Reference Integrity	233
7.14	Username/Password Authentication	235
7.15	SSL Client Authentication	236
7.16	Mutual Username/Password Authentication	238
7.17	Rehandshake	242
7.18	Secondary Channels	244
7.19	Closure	244
7.20	Summary	247

Chapter 8. Coding with SSL 249

8.1 Introduction 249
8.2 SSL Implementations 249
8.3 Sample Programs 250
8.4 Context Initialization 252
8.5 Client Connect 258
8.6 Server Accept 263
8.7 Simple I/O Handling 265
8.8 Multiplexed I/O Using Threads 269
8.9 Multiplexed I/O with `select()` 274
8.10 Closure 282
8.11 Session Resumption 285
8.12 What's Missing? 286
8.13 Summary 288

Chapter 9. HTTP over SSL 291

9.1 Introduction 291
9.2 Securing the Web 291
9.3 HTTP 293
9.4 HTML 295
9.5 URLs 298
9.6 HTTP Connection Behavior 300
9.7 Proxies 301
9.8 Virtual Hosts 302
9.9 Protocol Selection 302
9.10 Client Authentication 303
9.11 Reference Integrity 303
9.12 HTTPS 304
9.13 HTTPS Overview 304
9.14 URLs and Reference Integrity 307
9.15 Connection Closure 314
9.16 Proxies 316
9.17 Virtual Hosts 320
9.18 Client Authentication 322
9.19 `Referrer` 327
9.20 Substitution Attacks 327
9.21 Upgrade 328
9.22 Programming Issues 331
9.23 Proxy `CONNECT` 331
9.24 Handling Multiple Clients 336
9.25 Summary 341

Chapter 10. SMTP over TLS 343

10.1 Introduction 343
10.2 Internet Mail Security 343
10.3 Internet Messaging Overview 345
10.4 SMTP 346
10.5 RFC 822 and MIME 349
10.6 E-Mail Addresses 351
10.7 Mail Relaying 352
10.8 Virtual Hosts 355
10.9 MX Records 356
10.10 Client Mail Access 357
10.11 Protocol Selection 357
10.12 Client Authentication 357
10.13 Reference Integrity 357
10.14 Connection Semantics 358
10.15 STARTTLS 358
10.16 STARTTLS Overview 358
10.17 Connection Closure 363
10.18 Requiring TLS 364
10.19 Virtual Hosts 364
10.20 Security Indicators 365
10.21 Authenticated Relaying 366
10.22 Originator Authentication 367
10.23 Reference Integrity Details 367
10.24 Why Not CONNECT? 371
10.25 What's STARTTLS Good For? 372
10.26 Programming Issues 373
10.27 Implementing STARTTLS 373
10.28 Server Startup 374
10.29 Summary 375

Chapter 11. Contrasting Approaches 377

11.1 Introduction 377
11.2 The End-to-End Argument 378
11.3 The End-to-End Argument and SMTP 378
11.4 Other Protocols 379
11.5 IPsec 380
11.6 Security Associations 381
11.7 ISAKMP and IKE 381
11.8 AH and ESP 384
11.9 Putting It All Together: IPsec 386
11.10 IPsec versus SSL 386

11.11	Secure HTTP	389
11.12	CMS	390
11.13	Message Format	391
11.14	Cryptographic Options	392
11.15	Putting It All Together: S-HTTP	393
11.16	S-HTTP versus HTTPS	397
11.17	S/MIME	400
11.18	Basic S/MIME Formatting	401
11.19	Signing Only	401
11.20	Algorithm Choice	403
11.21	Putting It All Together: S/MIME	405
11.22	Implementation Barriers	406
11.23	S/MIME versus SMTP/TLS	407
11.24	Choosing the Appropriate Solution	408
11.25	Summary	410
Appendix A.	**Example Code**	**411**
A.1	Chapter 8	411
A.1.1	C Examples	411
A.1.2	Java Examples	425
A.2	Chapter 9	431
A.2.1	HTTPS Examples	431
A.2.2	mod_ssl Session Caching	436
Appendix B.	**SSLv2**	**455**
B.1	Introduction	455
B.2	SSLv2 Overview	455
B.3	Missing Features	457
B.4	Security Problems	458
B.5	PCT	461
B.6	What about SSLv1?	463
Bibliography		**465**
Index		**475**

Preface

The *Secure Sockets Layer* (SSL) is by far the most widely deployed security protocol in the world. Essentially every commercial Web browser and server supports secure Web transactions using SSL. When you buy online using "secure" Web pages (an estimated 20 billion dollars' worth of such transactions will occur in 2000), you're almost certainly using SSL.

Although its most common use is for securing Web traffic, SSL is actually quite a general protocol suitable for securing a wide variety of kinds of traffic. File transfer (FTP), remote object access (RMI, CORBA, IIOP), e-mail transmission (SMTP), remote terminal service (Telnet) and directory access (LDAP) are just some of the applications that have already been secured with SSL or its successor, *Transport Layer Security* (TLS).

The effort to secure all these protocols has taught us a number of significant lessons. First, doing a good job of using SSL/TLS to secure a protocol requires having a fairly deep knowledge of how it works. It is not possible to simply treat SSL/TLS as a black box that somehow magically provides security when used.

Second, although each application is slightly different, there seems to be a set of security problems that are common to every application you wish to secure. For instance, we usually need to figure out some way for the insecure and secure versions of an application protocol to coexist. Although there aren't cookie-cutter solutions to these problems, the security community is starting to develop a common set of techniques for solving these problems using SSL/TLS.

These techniques can often be applied to a new application protocol with minimal modification. In essence, we've developed a set of *design patterns* for securing protocols. Much of the work of securing a system is in recognizing which pattern most closely matches the system you're working with and then using the appropriate techniques.

The purpose of this book, then, is to address both of these needs. After reading this book, you should know most if not all of what you need to know in order to design secure systems using SSL/TLS. You'll know enough about SSL/TLS to understand what security features it can deliver and what it can't deliver. Further, you'll be familiar with the common design patterns for using SSL/TLS and be ready to apply them to new situations.

Ex.1024
Askeladden, L.L.C. / Page 61 of 788

What This Book Provides

This book is intended for anyone who wants to understand and use SSL/TLS.

- For designers, it provides information on designing systems that use SSL/TLS as well as a library of the techniques that have already been used.
- For programmers who program with SSL/TLS, it provides information on what your libraries are doing under the covers and what those functions you're calling are really doing. Understanding these details is critical for obtaining acceptable and predictable application performance.
- For SSL/TLS implementors it acts as an adjunct to the standard, explaining obscure sections and describing both common practice and why things are the way they are.

Intended Audience

This book assumes a basic familiarity with how the TCP/IP protocols work. Readers who are unfamiliar with TCP/IP would be best served to consult one of the many fine books describing TCP/IP. *TCP/IP Illustrated, Volume 1* [Stevens1994] is a good choice. RFC 791 [Postel1991a], RFC 792 [Postel1991b], and RFC 793 [Postel1991c] provide the definitive reference for TCP/IP. Although some of this book will be understandable without a deep understanding of TCP/IP, much of the discussion of performance will be difficult to follow without an understanding of TCP behavior.

Because SSL/TLS is a cryptographic protocol, properly understanding it requires at least basic familiarity with cryptographic algorithms, including public key cryptography, symmetric cryptography, and digest algorithms. Chapter 1 provides an introduction to cryptography and communications, but space is too limited to do a complete job. We attempt to cover the requisite cryptographic details for understanding SSL/TLS; however, readers interested in a broader understanding of the cryptographic issues should consult a cryptography text such as [Schneier1996a] or [Kaufman1995].

Organization of the Book

This book is written in two halves, matching the two primary goals described previously: understanding the protocol and understanding how to use it. The first half, Chapters 1–6, is devoted to describing SSL and TLS. We concern ourselves with the technical details of how they work and their security and performance properties in isolation.

The second half of the book, Chapters 7–11, covers the design of application protocols and systems that use SSL/TLS for security. First we describe general guidelines for using SSL/TLS and then we discuss several protocols that have already been secured using SSL/TLS.

Chapter 1 — Security Concepts provides an introduction to cryptography and communications security, with an eye toward its use in SSL/TLS. If you're already familiar with communications security, you may want to skip this chapter. If, on the other hand, you're not familiar with security, you'll want to read this chapter carefully so you don't get lost later.

Chapter 2 — Introduction to SSL is a broad overview of the history of SSL/TLS and what sorts of security features it provides. We also provide a snapshot of the status of SSL/TLS-secured protocols as of the time of this writing.

Chapter 3 — Basic SSL covers the most common SSL/TLS operational mode. We describe an entire SSL/TLS connnection from start to finish. This chapter should give you a very good idea of how SSL/TLS works in practice. All the other operational modes can be easily understood once you understand this chapter.

Chapter 4 — Advanced SSL covers the rest of the major operational modes. We cover session resumption, client authentication, and a number of algorithms that are only now seeing deployment with SSL/TLS, such as DH/DSS and Kerberos.

Chapter 5 — SSL Security describes the security benefits that SSL offers as well as (even more important) those it doesn't offer. Whereas previous chapters mostly focused on describing how things work, this chapter focuses on what you need to do to make a system that uses SSL/TLS secure.

Chapter 6 — SSL Performance describes the performance profile of TLS-based systems. It's been widely observed that security imposes significant performance demands on systems, but it's not widely understood that this impact is limited to certain parts of the protocol. We'll discuss these issues with an eye to getting better performance while preserving good security.

Chapter 7 — Designing with SSL is a guide to using SSL/TLS to secure application layer protocols. We focus on identifying the required security properties and on well-understood design techniques for satisfying these properties.

Chapter 8 — Coding with SSL discusses the common programming idioms required to write software that uses SSL/TLS. We provide complete sample programs in C and Java using the OpenSSL and PureTLS toolkits.

Chapter 9 — HTTP over SSL describes the application that started it all. SSL was originally designed by Netscape to work with HTTP, and we cover both the traditional way of doing things and the replacement that's currently being proposed.

Chapter 10 — SMTP over TLS describes the use of TLS to secure the *Simple Mail Transport Protocol* (SMTP), which is used for transporting e-mail. SMTP is a bad match for TLS, and this chapter illustrates some of the limitations of SSL and TLS.

Chapter 11 — Contrasting Approaches is devoted to describing other alternatives to securing your applications. SSL/TLS isn't always the best solution and part of knowing how to use a protocol is knowing when not to. This chapter tries to give you a perspective on your other choices. We discuss IPSEC, S-HTTP, and S/MIME as alternatives to SSL/TLS.

How to Read This Book

This book is suitable for a number of audiences of different technical abilities and requirements. You should read any section that interests you, but depending on your needs you may want to focus on specific sections.

Protocol Designers

If you're designing a new application-level protocol or securing an existing protocol with SSL/TLS, you should read the first parts of Chapters 1–6 so that you have a good general understanding of how SSL/TLS works. Then carefully read Chapter 7 for a guide to SSL/TLS design principles. You can skip Chapter 8 unless you intend to implement your design, but be sure to read Chapters 9 and 10 so you can see real-world examples of how SSL/TLS should and should not be used in practice. Finally, before you start to design, read Chapter 11 to make sure that SSL/TLS is appropriate for your design and that you wouldn't be better served by using another security protocol.

Application Programmers

If you're writing an application that uses a preexisting SSL/TLS toolkit you can safely read only the first parts of Chapters 1–6. You should also read the summaries at the end of each chapter. These sections discuss SSL/TLS and SSL/TLS implementation techniques in overview form. This will provide enough information to understand what your SSL/TLS toolkit is doing. You should carefully read Chapters 7 and 8, paying special attention to the programming techniques discussed in Chapter 8. If you are implementing HTTP or SMTP over SSL, you should also read the chapters that deal with those protocols.

SSL/TLS Implementors

If you're implementing SSL/TLS from scratch, you should read the entire book. If you're already familiar with cryptography you can skip Chapter 1; however, if you don't have a detailed knowledge of cryptography you should read the entire chapter. You should pay particular attention to Chapters 2–6, which provide a detailed description of SSL/TLS and of the various implementation techniques required to produce a fast and secure implementation.

Just Curious

If you just want to learn about SSL/TLS you can skip around in the book. If you don't already know about cryptography, read all of Chapter 1. Then read Chapters 2–6 so you know how SSL/TLS works. Then you can read as much or as little of the rest of the book as interests you. It's probably worth reading Chapter 11 to get some perspective on how SSL/TLS compares to other security protocols.

SSL/TLS Versions

By now you've no doubt gotten tired of seeing the name SSL/TLS. We've been using it to avoid being specific about exactly which version we mean. There are currently two

versions of SSL in wide deployment: SSL version 2 (SSLv2) and SSL version 3 (SSLv3). TLS, a modification of SSLv3, was standardized by the *Internet Engineering Task Force* (IETF) in 1999. Despite what you might think from the names, SSLv2 and SSLv3 are completely different protocols, and TLS is extremely similar to SSLv3. SSLv2 is essentially obsolete, and TLS isn't really in wide deployment as of this writing. In general, we'll use the term SSL to refer to SSLv3/TLS interchangeably. When we mean one or the other, we'll specify it explicitly. In the few instances when we're talking about SSL version 2, we'll use SSLv2.

Typographical Conventions

This book contains a number of network traces of real SSL or TLS sessions. In the presentation of such traces, we use a constant-width font to show the output of the program (`CONSTRUCTED`), and an italic font for comments inserted afterwards (*Comment*). Where the network trace displays a hexadecimal representation of the protocol data, we render that in a bold constant-width font (**`01 02 03`**). When we display encrypted data in plaintext form, we'll use an italic constant-width font (*`data`*).

Excerpts from standards (e.g., Internet RFCs) and protocol structure definitions in the main text are rendered in a sans serif font: (helvetica). In figures, we use Times for readability. Code snippets are rendered in a constant-width font (`int`). In a few special cases we'll need to wrap long lines, in which case we'll use the symbol ↵ at the end of the wrapped line to indicate that the text continues.

Historical notes and asides will be indented and rendered in this smaller font.

Network Traces

The network traces in this book are all traces of real sessions, mostly captured on the author's home ethernet. A variety of client and server programs were used, including OpenSSL, Netscape Navigator, Internet Explorer, and qmail. The traces were captured using the `tcpdump` program and stored to disk. The author's `ssldump` SSL decoding package was used to generate the traces shown in the text. `tcpdump` can be obtained from `http://www.tcpdump.org/`. `ssldump` can be obtained from `http://www.rtfm.com/ssldump/`.

Source Code

This book contains a number of snippets of source code. The source code in Chapters 8 and 9 were written by the author and are Copyright © 1999–2000 Eric Rescorla. They may be used and reproduced for any purpose and without fee but no warranty of any kind is provided. Machine-readable versions are available from the author's Web site at `http://www.rtfm.com/sslbook/examples`.

The Java source code in Chapter 5 is part of the PureTLS Java SSL/TLS implementation. It may be obtained at `http://www.rtfm.com/puretls`. This source code is covered by the following copyright.


```
Copyright (C) 1999, Claymore Systems, Inc.
All Rights Reserved.

ekr@rtfm.com  Tue May 18 09:43:47 1999

This package is a SSLv3/TLS implementation written by Eric Rescorla
<ekr@rtfm.com> and licensed by Claymore Systems, Inc.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software
   must display the following acknowledgement:
   This product includes software developed by Claymore Systems, Inc.
4. Neither the name of Claymore Systems, Inc. nor the name of Eric
   Rescorla may be used to endorse or promote products derived from this
   software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

The SSL session caching example in Appendix A is from Ralf S. Engelschall's (rse's) Mod_ssl package, available at `http://www.modssl.org/`, The version used is 2.6.1-1.3.12. It is subject to the following copyright.


```
==========================================================================
Copyright (c) 1998-1999 Ralf S. Engelschall. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials
   provided with the distribution.

3. All advertising materials mentioning features or use of this
   software must display the following acknowledgment:
   "This product includes software developed by
    Ralf S. Engelschall <rse@engelschall.com> for use in the
    mod_ssl project (http://www.modssl.org/)."

4. The names "mod_ssl" must not be used to endorse or promote
   products derived from this software without prior written
   permission. For written permission, please contact
   rse@engelschall.com.

5. Products derived from this software may not be called "mod_ssl"
   nor may "mod_ssl" appear in their names without prior
   written permission of Ralf S. Engelschall.

6. Redistributions of any form whatsoever must retain the following
   acknowledgment:
   "This product includes software developed by
    Ralf S. Engelschall <rse@engelschall.com> for use in the
    mod_ssl project (http://www.modssl.org/)."

THIS SOFTWARE IS PROVIDED BY RALF S. ENGELSCHALL ``AS IS'' AND ANY
EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RALF S. ENGELSCHALL OR
HIS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
```

```
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
=====
```

Acknowledgments

It's impossible to write any book, much less a technical book, without the assistance of many people. As is customary, I'd like to use this space to acknowledge some of them.

My technical reviewers not only kept me honest but also ensured that my writing was as clear as I could make it. Joshua Ball, Joe Balsama, Douglas Barnes, Debasish Biswas, Andrew Brown, Robert Bruen, Megan Conklin, Russ Housley, Paul Kocher, Brian Korver, Chris Kostick, Marcus Leech, Robert Lynch, Joerg Meyer, D. Jay Newman, Tim Newsham, Stacey O'Rourke, Radia Perlman, Mark Schertler, Win Treese, Tom Weinstein, and Tom Woo all reviewed sections of the manuscript.

A number of people generously provided answers to technical questions that I had while writing, often without even knowing that they were doing so. In particular, I'd like to thank John Banes, Steve Bellovin, Burt Kaliski, Paul Kocher, Bodo Moeller, Dan Simon, and Robert Zuccherato for helping to fill in gaps in my knowledge. Special thanks goes to Terence Spies who provided valuable comments on the entire manuscript and answered numerous technical questions about Microsoft's SSL implementation.

This book makes extensive use of OpenSSL to generate demonstration SSL traffic. OpenSSL wouldn't exist without the hard work of Eric Young and Tim Hudson in creating SSLeay as well as the OpenSSL team in maintaining and improving OpenSSL after Eric Young's departure for greener pastures.

Brian Korver and Stacey O'Rourke from Network Alchemy/Nokia provided significant assistance in collecting some of the performance data in Chapter 6. Not only did they allow me to use their machines and network, they graciously reconfigured them in obscure ways so that I could produce particular cases.

Although I never met him, the late W. Richard Stevens nevertheless contributed greatly to this book. The idea of showing network traces to illustrate the protocol is taken from Stevens's fine *TCP/IP Illustrated* series. I've striven throughout this book (with only limited success) to emulate his clear and accessible style.

It wouldn't be possible to publish anything without the publisher and I've been quite pleased to be working with Addison-Wesley. I'd particularly like to thank Mary Hart for first suggesting this project to me and then putting up with the endless delays as the book exploded from a trim 200 pages to break through the 400 barrier. I'd also like to thank my production manager, Kathy Glidden, who patiently answered my endless typesetting questions.

Although they didn't contribute directly to the writing of this book, I'd like to thank Allan Schiffman, Marty Tenenbaum, and Jay Weber. Jay and Marty gave me a chance when I had talent and no experience. Over the eight years I've known him, Allan has

taught me an immeasurable amount about computer science. In a similar vein, had my parents not taught me how to think, I wouldn't be able to do anything useful at all.

Jennifer Gates played a major part in keeping me sane during the writing—an unenviable task. In the past few years, she and her husband Lee have many times provided hospitality and friendship above and beyond the call of duty.

The other major factor in my continued sanity was triathlon. Kevin Joyce and Kyle Welch provided valuable advice and motivation in the upkeep of my habit.

Finally, I'd like to thank Lisa Dusseault and Kevin Dick. Lisa and Kevin both read the entire manuscript and helped take my rough drafts and turn them into readable chapters. Without them, I would most likely never have finished at all and the text would certainly be far worse than it is now.

Camera-ready copy was produced by the author using James Clark's Groff package. I welcome electronic mail from readers with comments, suggestions, etc.

Mountain View, CA
September 2000

Eric Rescorla
ekr@rtfm.com

1

Security Concepts

1.1 Introduction

This chapter is intended to provide a basic introduction to communications security and cryptography. Communications security is a complicated topic and many fine books have been written about it. Our intent here is not to provide an exhaustive discussion of the topic but rather to teach you enough to understand the concepts and terminology that will be used throughout the rest of the book. Readers who are already familiar with cryptography and communications security should feel free to skip this chapter entirely.

We start by explaining the sorts of threats we're concerned about and the various sorts of security services we can provide. Next we provide a broad overview of cryptographic algorithms and how to put them together to provide these security services. Finally, we discuss some details of the various algorithms which will be relevant when we discuss their use in SSL/TLS.

1.2 The Internet Threat Model

The first thing that we need to do is define our *threat model*. A threat model describes what resources we expect the attacker to have available and what attacks the attacker can be expected to mount. Nearly every security system is vulnerable to some threat or another. To see this, imagine that you keep your papers in a completely unbreakable safe. That's all well and good, but if someone has planted a video camera in your office they can see your confidential information whenever you take it out to use it, so the safe hasn't bought you that much.

Therefore, when we define a threat model, we're concerned not only with defining what attacks we are going to worry about but also those we're not going to worry about. Failure to take this important step typically leads to complete deadlock as designers try to figure out how to counter every possible threat. What's important is to figure out which threats are realistic and which ones we can hope to counter with the tools available to us.

Designers of Internet security protocols typically share a more or less common threat model. First, it's assumed that the actual end systems that the protocol is being

Ex.1024
Askeladden, L.L.C. / Page 71 of 788

executed on are secure. Protecting against attacks where one of the end systems is under the control of the attacker is extraordinarily difficult, if not impossible. This assumption comes with two caveats. First, compromise of any single end system shouldn't break security for everyone. There should be no *single point of failure*. For instance, if an attacker breaks system A, then all communications between B and A may be compromised, but communications between B and C should be safe. If we must have a single point of failure it must be possible to harden it against attack. Second, attackers may control systems that attempt to pose as legitimate end systems. All we're assuming is that users can expect that their own machines haven't been compromised.

Other than that, we assume that the attacker has more or less complete control of the communications channel between any two machines. He can certainly inject packets into the network with arbitrary address information, both for the sender and the receiver, and can read any packet that is on the network and remove any packet packet he chooses. Any packet you receive must be assumed to potentially come from the attacker and any packet you send might be modified in transit. An attack that depends on the attacker writing data to the network is known as an *active attack*. An attack that merely involves reading data off the network is known as a *passive attack*.

An obvious corollary of the assumption that the attacker can modify traffic is that the attacker can shut down all communication between any pair of machines simply by removing all relevant packets. This is one form of *denial-of-service* attack. Another form would be to force you to use up enormous CPU resources responding to connections. Conventionally, protocol designers don't worry about denial-of-service attacks, not because these attacks aren't important but because they're extraordinarily difficult to prevent.

One of the most important functions of a threat model is to arrange that security doesn't become more expensive than it is worth. Security measures should be employed only up to the point where the cost to implement them doesn't exceed the expected risk. Failure to make this judgment correctly can easily lead to a situation where no risk is judged acceptable and thus no acceptable system can be designed.

Part of the risk calculation is the effort required by the attacker to mount a given attack, and cost generally increases with each attack prevented. No security system is resistant to every attack. The function of a security model is to allow designers to determine which attacks are worthwhile to prevent.

However, accurately estimating how much security you need requires accurately estimating the attacker's capabilities. If an attack that was originally considered impractical is discovered to be simple, then there will be a window of vulnerability while people adjust their security models and implementations to compensate.

1.3 The Players

To make it easier to understand the various examples we'll be discussing in this chapter, we'll use the same names repeatedly for the various parties. By convention, the two communicating parties are referred to as Alice and Bob, after the names used in the original RSA paper [Rivest1979]. The attacker is known merely as "the attacker."

1.4 The Goals of Security

Most people speak of security as if it were a single monolithic property of a protocol, but as the previous discussion shows, that's clearly not true. Depending on precisely what capabilities the attacker has, he poses different risks to the security of our data. Communications security consists of a number of distinct but somewhat related properties. Exactly how these are partitioned depends on whom you're talking to. The partitioning we've found most useful is to divide them into three major categories: *confidentiality*, *message integrity*, and *endpoint authentication*.

Confidentiality

When most people think of security, they think of *confidentiality*. Confidentiality means that your data is kept secret from unintended listeners. Usually, these listeners are eavesdroppers. When the government taps your phone, that poses a risk to your confidentiality. (Incidentally, it's also a passive attack, unless the Fed on the line starts trying to imitate your voice.)

Obviously, if you have secrets, you're concerned that no one else knows them and so at minimum you want confidentiality. When you see spies in the movies go into the bathroom and turn on all the water to foil bugging, the property they're looking for is confidentiality.

Message Integrity

The second primary goal is *message integrity*. The basic idea here is that we want to be sure that the message we receive is the one that the sender sent. In paper-based systems, some message integrity comes automatically. When you receive a letter written in pen you can be fairly certain that no words have been removed by an attacker because pen marks are difficult to remove from paper. However, an attacker could have easily added some marks to the paper and completely changed the meaning of the message.

On the other hand, in the electronic world, since all bits look alike, it's trivial to tamper with messages in transit. You simply remove the message from the wire, copy out the parts you like, add whatever data you want, and generate a new message of your choosing, and the recipient is no wiser. This is the equivalent of the attacker taking a letter you wrote, buying some new paper, and recopying the message with changes. It's just a lot easier to do electronically because all bits look alike.

Endpoint Authentication

The third property we're concerned with is *endpoint authentication*. What we mean by this is that we know that one of the endpoints in the communication (typically the sender) is the one we intended. Without endpoint authentication, it's very difficult to provide either confidentiality or message integrity. For instance, if we receive a message from Alice, the property of message integrity doesn't do us much good unless we know that it was in fact sent by Alice and not the attacker. Similarly, if we want to send a

confidential message to Bob, it's not of much value to us if we're actually sending a confidential message to the attacker.

Note that endpoint authentication can be provided asymmetrically. When you call someone on the phone, you can be fairly certain that you have the right person—or at least that you got a person who's actually at the phone number you called. On the other hand, if they don't have caller ID, then the receivers of a phone call have no idea who's calling them. Calling someone on the phone is an example of recipient authentication, in which you know who the recipient of the call is (it's difficult but not impossible to compromise the phone network), but they don't know anything about the sender.

On the other hand, cash is an example of sender authentication. A dollar bill is like a message signed by the government. The government has no idea who's got any given dollar bill, but you can be fairly certain that any bill was actually printed by the U.S. Mint because currency is difficult to forge.

A Physical Example

To make this all a little clearer, let's examine a physical system that provides all three properties together and see how the various physical security features interact to provide each property. We'll start with a system with no security features and keep making changes until we have all the security features we want.

You receive a postcard in the mail, addressed to you, ostensibly from Alice. What can you know about it and its contents? Not much. Any random eavesdropper could have read it, and they could have changed it as well.

One potential improvement might be for Alice to write a letter instead of a postcard and put the letter in an envelope. Does this improve the situation? Not much, actually. By examining the envelope, you can be fairly confident that it wasn't tampered with (it could have been steamed open, but you can get envelopes that are resistant to this), so you might think that you've got confidentiality and message integrity for the contents. This isn't necessarily true. The attacker could have opened the original envelope, read the message or modified the contents, and then resealed it in a new envelope. So, in fact, you can't have any confidence in the message at all. Both message integrity and confidentiality have collapsed because you don't have sender authentication.

Note that from Alice's perspective, the situation is even worse. As far as she knows, the message was opened as soon as the mail carrier picked it up from her door. We'll show how to fix this problem a little later.

The traditional way to ensure that a letter is not opened in transit is to affix a wax seal across the envelope flap. It's possible to open and reattach such seals, but let's assume for the moment that it wasn't possible. Another more modern approach is for the sender to sign across the flap. Assuming that you recognize either the seal or the sender's signature, you can have some confidence that the envelope arrived unopened, because signatures are difficult to forge.

However, we can have this confidence only if we have some way of knowing who ought to have sent us the message. What if the attacker opened the envelope and then resealed it with his own envelope, and signed his own name across the flap? If we didn't

know that the signature should in fact be Alice's, we wouldn't be able to detect this attack.

To see why this would be useful, imagine that the envelope contains an order form for something, along with Alice's credit card number. The attacker changes the message to have his shipping address but Alice's credit card number and reseals the envelope. You incorrectly ship to the attacker and bill Alice. This attack is very difficult to stop with the sorts of technology we've discussed so far.

The mechanisms we've been talking about so far—envelopes, seals, etc, are *tamper-evident*. The receiver can tell that the contents were tampered with, but can't stop it from happening. The easiest way to stop this new attack is to use packaging that's *tamper-resistant*. (The phrase *tamper-proof* is often used, but that seems overly optimistic.) Imagine that instead of sending a letter, Alice sends us a safe containing the letter. The only two keys to the safe are in Alice's and our possession.

This pretty much fixes the problem entirely. Because you have the only other key and you know that you didn't send the letter yourself, you can be quite confident that Alice sent it, so you have sender authentication. Because no one besides you or Alice can open the safe, you know that it hasn't been opened and read or modified in transit, so you've got confidentiality and message integrity as well. Similarly, Alice knows that no one else but you can open the safe, so she can be assured that she has recipient authentication, confidentiality, and message integrity.

This approach has (at least) one serious problem. You have to exchange one of a pair of keys with everyone with whom you might ever want to communicate. If you think about how many messages the typical business sends in a day, you'll see that this method is inconvenient.

This example is useful for two purposes. First, it illustrates the various security services we might wish to provide. Just as important, it illustrates the concept of the threat model. Because it's fairly expensive to go around mailing safes, this isn't a very practical approach, and people typically settle for lower levels of security and just assume that the postal service is trustworthy. For particularly confidential information, they might use a courier, but that's about the typical limit of their concern. Attackers who are able to open the mail and arbitrarily change it just aren't part of the standard business threat model. The cost of providing high security for every message mailed is simply so high that no company can afford to do so.

1.5 Tools of the Trade

Note that we haven't even mentioned cryptography yet. We're going to get to it now. *Cryptology* is the theory of designing the various algorithms we use to provide security. *Cryptography* is the study of using these algorithms to secure systems and protocols. This section is going to provide a broad overview of the types of cryptographic algorithms available to us. The next section will discuss how to use them to provide security services.

Encryption

Conceptually, the easiest sort of algorithm to understand is encryption. The idea is simple: an encryption algorithm takes some data (called *plaintext*) and converts it to *ciphertext* under control of a *key*. The ciphertext looks like random data and no useful information about the plaintext (except possibly the length) can be gathered without knowing the key. The key is usually just a short random string, usually on the order of 8–24 bytes. The relationship between these elements is shown in Figure 1.1.

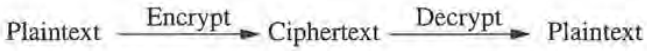


Figure 1.1 Encryption and decryption

An encryption algorithm should be viewed as providing pure confidentiality. The exact effects on the plaintext of tampering with the ciphertext vary depending on the exact encryption algorithm used, but such tampering is often quite difficult to detect. Thus, when we receive a message that has only been encrypted, we don't know that it hasn't been tampered with.

A good encryption algorithm should have its security totally determined by the number of possible keys. The fastest attack should be *exhaustive search*: given a ciphertext, the attacker tries each key one at a time until he finds a key that produces a plausible decryption. The security of the algorithm should depend solely on the secrecy of the key. The algorithm should not need to be secret to be secure.

An attack in which the attacker knows the plaintext corresponding to the ciphertext is called a *known plaintext* attack. An attack in which the attacker doesn't know the plaintext is called a *ciphertext only* attack. Even if the attacker doesn't know the details of the plaintext, he may know something about it, which would allow him to mount a ciphertext only attack. For instance, the attacker may know that the plaintext is ASCII, in which case any decryption that includes non-ASCII characters must be using the wrong key.

Because the sender and the recipient share the same key (which key must be kept secret), what we've just described is sometimes referred to as *secret key cryptography*, as opposed to *public key cryptography*, which we'll talk about in a little bit.

Encryption algorithm design has been a very active field, and there are tens if not hundreds of available algorithms. The most popular algorithms include the *Data Encryption Standard* (DES) [NIST1993a], *Triple-DES* (DES repeated three times) [ANSI1985] , *RC2* [Rivest1998], and *RC4* (no official RC4 specification has ever been published but see [Schneier1996a] for a description).

Message Digest

A message digest is simply a function that takes as an input an arbitrary length message and outputs a fixed-length string that is characteristic of the message. The most important property of a message digest is *irreversibility*. It should be extremely difficult to

compute a message given its digest. It's easy to show via a counting argument that the digest value doesn't give enough data to generate the original message; the number of possible messages (of arbitrary length) is far longer then the number of fixed-length digests; many messages must map to a single digest, making it impossible to reverse the function. However, for a digest to be secure, it must be difficult to generate *any* of the messages that digests to this value. By difficult, we mean that you need to search a message space of proportional size to the size of the digest in order to find a matching message text.

The second important property of a message digest is that it should be difficult to produce two messages M and M' such that they have the same digest. This property is known as *collision-resistance*. It turns out that the strength of any message digest against finding collisions is only half the size of the digest, so a 128-bit digest is only 64 bits strong against collisions; it requires roughly 2^{64} operations to generate a collision. As a consequence, the limiting factor in choosing a digest length is usually strength against collisions rather than strength against reversibility.

The primary uses for message digests are for the computation of digital signatures and *message authentication codes* (MACs), both of which we'll talk about later in this chapter. However, there's at least one use of simple digests that's obvious now: they can be used to prove possession of a secret without revealing the secret. Imagine that you've invented something new and you want to show that you invented it first without actually telling anyone what it is. You write the secret down, compute the digest, and publish the digest in the newspaper classified ads. Then, if anyone challenges your primacy, you can show them the original text and they can independently verify the digest.

The most widely used message digest algorithms are Message Digest 5 (MD5) [Rivest1992] and Secure Hash Algorithm 1 (SHA-1) [NIST1994a]. The term *hash algorithm* is often used synonymously with the term *message digest*, because digest algorithms are superficially similar to normal hash algorithms. I'll be using the terms *hash* and *digest* interchangeably throughout the text.

Message Authentication Codes

Imagine that Alice and Bob share a key and Alice wants to send Bob a message that he will know is from her. If she wanted to encrypt it, that would be straightforward—she'd just use the key they share as an encryption key. But as we said, this doesn't provide any real assurance that the message hasn't been tampered with, and only some assurance that it's from Alice. What we need is a new tool, the *message authentication code* (MAC). A MAC is sort of like a digest algorithm, but it also incorporates a key into the computation, so the MAC is dependent on both the key being used and the message being MACed. Actually, MACs are usually constructed from digest algorithms.

Although there have been a number of attempts to construct MACs based on various digest algorithms, the Internet security community seems to be settling on a construct called HMAC [Krawczyk1997], which describes how to create a MAC with provable security properties based on any digest for which a certain somewhat reasonable set of assumptions holds. A variant of HMAC is used in SSLv3, and HMAC itself is used in TLS.

The Key Management Problem

With the addition of a MAC to our toolbox, we now have an electronic system equivalent to mailing safes around, as we discussed at the beginning of the chapter. Naively, Alice can take our message, encrypt it with our shared key, add a MAC also based on the shared key, and send it to Bob. She knows that only Bob can read it because only Bob has the shared key needed to decrypt it. Similarly, Bob knows that only Alice could have sent the message because only Alice has the shared key needed to create the MAC on the message. Thus, he knows that Alice sent the message and that it hasn't been tampered with.

So, we have everything we need, right? Wrong. Although these messages are lighter than safes and thus cheap to mail around, we still have the problem of sharing a key with everyone we communicate with. This is inconvenient because it means having all those keys floating around. But more important, it means that you must actually meet everyone with whom you want to communicate in order to exchange keys. This puts a damper on your being able to buy anything over the Internet unless you've personally met the vendor. This inconvenience is referred to as the key management problem.

Key Distribution Centers

The most popular solution to the key management problem is *public key cryptography* (PKC), which we'll talk about in the next section, but there is also a fix for the key management problem using only the tools we've talked about so far. The basic idea is that we use a *trusted third party*, which we trust to authenticate other parties to us. The third party is usually implemented as a secure machine somewhere on the network. This machine is called a *key distribution center* (KDC). Each person who wants to communicate securely shares a key with the KDC. When Alice wants to communicate with Bob, she sends a message to the KDC, protected with the key she shares with the KDC, requesting to communicate with Bob. The KDC generates a new encryption key for the Alice-Bob communication and returns it in a message called a *ticket*.

A ticket consists of two messages in one. The first is a message to Alice with the new key. The second is a message to Bob encrypted under Bob's key containing the new key. Alice forwards Bob's half of the ticket to Bob and now Alice and Bob share a key. The basic version of this protocol was invented by Needham and Schroeder [Needham1978], but the most widely deployed variant is Kerberos, which is used extensively for authentication and encryption at MIT, among other places (see [Miller1987]).

This scheme has two primary drawbacks. First, the KDC must be online all the time because no communication can be initiated when it's offline. Second, the KDC can read all the traffic sent by any two parties. It can also forge traffic between two parties. Worse yet, if the KDC is ever compromised, all traffic sent between any two KDC users is compromised. This is not good. Nevertheless, there has been a fair amount of interest in this type of protocol for closed systems.

Public Key Cryptography

In 1976, some very smart people at Stanford figured out a better way to solve the key management problem. In the article "New Directions in Cryptography" [Diffie1976], Whitfield Diffie and Martin Hellman suggested what we now know as public key cryptography. The basic idea is that you have a function in which encryption and decryption use different keys. You publish your encryption key (the *public key*) but keep your decryption key (the *private key*) secret. (Because the public and private keys are different, public key cryptography is sometimes called asymmetric cryptography and secret key cryptography is sometimes called symmetric cryptography). This means that anyone can send you a secret message without ever meeting you. This solves the confidentiality part of the equation while eliminating the inconvenience of preshared keys.

> As it turns out, public key cryptography was also invented by one of the British intelligence agencies, the *Communications-Electronics Security Group* (CESG) during 1970 to 1974. They referred to the practice as *non-secret encryption* (NSE). Interestingly, the techniques invented include both Diffie-Hellman and a variant of RSA, which implies that these methods are somehow fundamental. However, the results were classified and the academic community reinvented both techniques independently. The prior existence of these techniques in the intelligence community was only revealed in 1998. [Ellis1987] details the history of NSE. [Ellis1970] describes a demonstration that NSE was possible. [Cocks1973, Williamson1974, Williamson1976] describe the specific techniques.

It turns out that PKC has a solution to the authentication part of the problem, too. Your private key can be used to create something called a *digital signature*, which bears the same relationship to a MAC that public key encryption bears to secret-key encryption; you use your private key to *sign* a message and the receiver uses your public key to *verify* your signature. Note that a digital signature has one important property that a MAC does not: *nonrepudiation*. Either sender or receiver can generate a MAC, but only the signer can generate a signature. Thus, the recipient can prove that the sender signed the message and the sender cannot deny it.

Certification

Unfortunately, what we have so far doesn't completely solve the key management problem, though it gives us the tools we need to solve it. The easiest way to see what is still wrong is to ask how the parties get each other's public keys. If the keys are published electronically or if they're exhanged by the communicating parties, then an attacker can tamper with them while they are in transit to the receiver. When two parties want to communicate, the attacker intercepts their keys and instead sends his own key to each party. Thus, each party encrypts to him and then he reencrypts to the real recipient, as shown in Figure 1.2. This is called a *man-in-the-middle* attack.

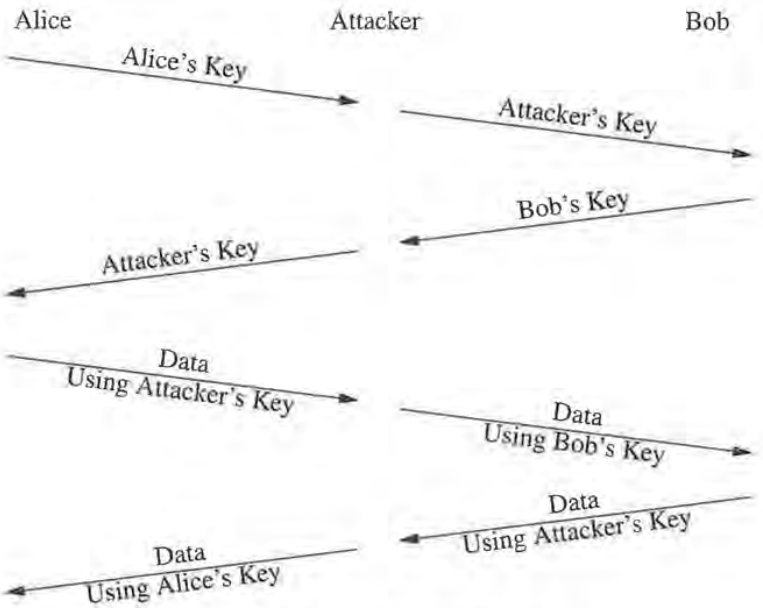


Figure 1.2 The man-in-the-middle attack

However, if keys are published physically, it's a major inconvenience. The solution is (again) to have a trusted third party, called a *Certificate Authority* (CA). The CA publishes a directory signed with the CA's private key. In actual practice, rather than signing a directory, what the CA does is sign individual messages which contain both the name of the key owner and his public key. These messages are generally known as *certificates*, hence the name certificate authority. The primary standard for certificates is X.509 [ITU1988a] profiled for the Internet in RFC 2459 [Housley1999a].

The CA's public key is published in some physical form, but there aren't many CAs and they don't change their keys often, so it's not such a big problem. In actual practice, CA keys are usually compiled into the software that needs them so that they are distributed with that software. This makes a lot of sense when the software is shipped on CD-ROM or floppy, but sometimes that software is just downloaded, in which case we're right back where we started—which just goes to show that people often don't act rationally.

Figure 1.3 shows an expanded view of a public key certificate. The details aren't that important, but notice the basic structure: The certificate contains an *issuer name* (the name of the signer of the certificate, in this case "Secure Server..."), *subject name* (the name of the holder of the key that the certificate vouched for, in this case "www.amazon.com..."), the *subject public key* (the key itself), a bunch of control information such as expiry date and serial number, and a *signature* over the whole data object.

```
version:                              v1
serial number:                        2A 17 EF 73 97 07 74 7B E2 4B FB
    61 95 DB 4D 77
signature
    algorithm:                        md5WithRSAEncryption
issuer:
    C=US
    O=RSA Data Security, Inc.
    OU=Secure Server Certification Authority
validity
    not before:                       Sat Jan 28 02:21:56 1995
    not after:                        Thu Feb 15 02:21:55 1996
subject:
    C=US
    ST=Washington
    L=Seattle
    O=Amazon.com, Inc.
    OU=Software
    CN=www.amazon.com
subject public key
    algorithm:                        rsaEncryption
    modulus
        bit length:                   1024
        value:
            00 C8 1B 8B FA 40 C3 5B E3 46 3F 17 10 56 19 64 C4 F4 F9
            CC AE CA F7 0B 02 1C C3 2D 27 60 91 16 C0 A1 23 8B CA 90
            77 31 25 CA D9 DE B0 87 F5 25 C9 12 7A 95 DF DC 6C E4 1C
            C3 31 9F 77 BE 69 3E 9F BB 35 BF F3 3D BA 7A 72 DA 5D 0C
            60 91 29 F8 89 67 50 5C 32 46 63 F2 FF 42 9D 24 F2 DC 6F
            E5 CA D3 CD 3A AB 9D 5F A9 4D B0 82 91 E3 D3 EA AA EF 78
            8A C1 06 B6 6D EA 56 B8 7E 68 5D AF 4D 85 AF
    public exponent:
        bit length:                   2
        value:                        03
signature
    value:
        03 43 60 4B 5B 4B F1 78 56 BF B4 9B 81 E6 EE 0D 19 1B 4E 43 BD
        D9 C7 62 62 55 32 C7 15 A4 33 3A CA 0E 60 E5 FE D7 53 94 C6 AC
        17 D0 CE 7B 11 27 0C 3B 26 19 6D 35 55 4C D8 26 F4 5F F0 90 0D
        90 7F FC 39 47 FE EE B4 72 92 93 BF 93 7F 5C 56 38 10 F5 E5 58
        B5 6C 3E E0 B4 55 8D 74 BE 84 F1 53 67 49 5B 14 12 E6 A7 59 A9
        97 9E 6C E4 59 A6 8F 4E 7E B5 D9 2D 80 3F 38 3C 4C 11 A7 37
```

Figure 1.3 A public key certificate

Public key solutions involving certification still involve a trusted third party (the CA), but they do fix the primary problems we described with KDC-based systems. Because the same certificate can be used to prove one's public key to anyone, the CA doesn't have to be online in order for Alice and Bob to communicate. And because the CA doesn't have access to anyone's private keys, it can't read any of the messages.

Distinguished Names

The subject name and the issuer name in the certificate are X.500 *distinguished names* (DNs) [ITU1988b]. The idea behind a distinguished name is to provide a unique name for every network entity. In order to do this, a DN has a hierarchical structure. A DN consists of a series of *relative distinguished names* (RDNs). RDNs are like components of DNS names. Each RDN specifies the name of the entity in the namespace of some other entity. Thus, the highest-level RDN must be globally unique, but each RDN under that must be unique only within the scope of the previous RDNs. RDNs have structure as well. Each RDN consists of a series of *attribute-value assertions*. An AVA is basically a key/value pair. Figure 1.4 shows a possible sample DN.



Figure 1.4 A sample DN

Each box in Figure 1.4 represents an RDN. Thus, the top-level RDN contains one AVA, with the *attribute* being *Country* and the *value* being *US*, and so on. Note that although

in theory each RDN may have multiple AVAs, in practice, they almost always have just one. Similarly, although in theory any AVA may appear at any level, in practice they go from least specific (country) to most specific (common name—roughly corresponding to your name—or e-mail address). It's quite common to render distinguished names in text form, in which case the attribute names are abbreviated. The name in Figure 1.4 would be rendered

```
C=US, O=RTFM, OU=Consulting, CN=Eric Rescorla
```

Extensions

Certificates contain a standard set of values, but it's quite common to want to put other information in them. X.509 version 3 provides for an arbitrary series of *extensions*. Extensions are simply type-value pairs. Although X.509 allows for private extensions, a number of standard extensions have been provided. The three important ones for our purposes are

subjectAltName contains other names for this user. These may be other DNs, but may also be other name forms, including dNSName (a DNS hostname) and emailAddress (an e-mail address).

keyUsage contains a bitmask of acceptable uses for this key, including signing, encryption, etc.

extendedKeyUsage allows a list of arbitrary *object identifiers* (OIDs), (see the *ASN.1, BER, and DER* section) listing detailed possible uses for this key.

Certificate Revocation

Consider the case where user's key is compromised by being lost or exposed. We'd like to have some way to communicate to other users that a public key and the certificate verifying it were no longer to be trusted. The most common way to approach this problem is to use something called a *Certificate Revocation List* (CRL, pronounced "krill").

A CRL is a signed, dated list of all the certificates that have been *revoked*, that is, they are no longer considered to be valid. Assuming that the CA in question issues CRLs, verifying a certificate requires obtaining the appropriate (latest) CRL and checking that the certificate doesn't appear on that list.

Unfortunately, CRLs don't work very well, for two reasons. First, they need to be issued periodically, which means that there is a window of vulnerability between the time period when a CRL issues and the next CRL is due. The key might have been compromised but users can't tell because the CRL on which it would have appeared has not been issued. Second, CRL distribution is problematic. CRLs can get rather large and either protocols need to carry them around or users need to get them from the CAs, which requires the CA to be online.

If the CA is going to be online, it can also provide *online certificate status*. This can provide completely up-to-date revocation status. The user can make a request of the CA for the status of a given certificate and the CA provides a signed response. The *Online*

Certificate Status Protocol (OCSP) [Myers1999] can be used to do this job. However, the need to sign these responses creates a high load on the server CPU. Obviously, this is undesirable. Some optimizations that reduce this load [Kocher1996a] have been described but are not in wide deployment.

ASN.1, BER, and DER

ASN.1 falls into the category of "highly unpleasant things that it is sometimes necessary to know." A lot of our basic security tools, in particular X.509 certificates, end up being defined using ASN.1. ASN.1 was designed as part of the International Telecommunications Union's *Open Standards Interconnect* (OSI) effort as a description language for the OSI protocols. The basic idea was to create a system for describing data structures that would allow machine generation of data encoders and decoders (generically, *codecs*).

This seemed like a good idea at the time: you have a language which can be used to describe data formats as structured types, much like that for C structs and Java classes. This language is called *Abstract Syntax Notation 1* (ASN.1). You also describe how to map structures in this language into data encodings on the wire. This allows you to write a compiler that does two things:

1. Generate mappings from ASN.1 structures to whatever language you're using
2. Automatically generate codecs for the structures

Unfortunately, ASN.1 is complicated and counterintuitive, making it hard to write the ASN.1 specifications themselves as well as making compilers fairly complicated. Worse yet, apparently on the theory that more is better, the people who designed ASN.1 defined not one but upwards of four different sets of encoding rules for laying out the data on the wire. Each encoding serves a slightly different set of goals, but the result is a confusing mess.

The two ASN.1 encoding rules we'll be concerned with are *Basic Encoding Rules* (BER) and *Distinguished Encoding Rules* (DER). Whereas BER allows several ways to encode any given piece of data, DER is a subset of BER that picks one and sticks with it so that you can be sure that any given structure will be encoded the same way by any codec. The advantage of BER is that it is often more efficient to BER encode data than to DER encode it. However, if you are digitally signing data, you want semantically equivalent messages to always have the same encoding, and so you need DER.

We're not even going to try to describe how to read ASN.1 in any detail. In order to get the general idea, all you really need to know is that it's like C structures except that the type definitions are backwards: the name comes first and the data type comes second. Figure 1.5 shows a sample ASN.1 structure. What it says is that Foo is a sequence of two elements, bar and mumble. bar is of type INTEGER and mumble is of type BIT STRING.

```
Foo ::= SEQUENCE {
  bar INTEGER,
  mumble BIT STRING
}
```

Figure 1.5 A sample ASN.1 structure

This example should give you enough information to get the general idea of what ASN.1 structures mean. If you're interested in the details, your best bet is to refer to [RSA1993a]. The actual standard is in [ITU1988c] and [ITU1988d].

For the purposes of this book, you only really need to know about one other thing: *object identifiers* (OIDs). OIDs are globally unique byte strings assigned to any kind of object, such as algorithms, keyusages, etc. The OID space is *federated*—ISO has assigned sections of the OID space to various entities who can further divide the space among other entities. Thus, a large number of organizations can assign OIDs and it's possible to get your own OID section (called an *arc*) if you want to assign private but unique OIDs.

1.6 Putting It All Together

We now have enough pieces to build some simple security systems. The system we're about to describe sounds complicated, but it's important to realize that nearly every piece of communcations security technology we'll be discussing is built from one of four simple pieces: *encryption*, *digest*, *public key encryption*, and *digital signature*. These pieces are often called *security primitives*, and primitives can be combined to build more complicated structures.

For instance, public key methods are much slower than secret key methods, so it's convenient to combine the two techniques, using public keys to exchange secret keys. To encrypt a message, Alice generates a random secret key (variously called a *session key*, *message encryption key*, *content encryption key*, or *data encryption key*) and encrypts it under Bob's public key, which she gets from Bob's certificate. She then encrypts the message with a symmetric algorithm using the session key. This combination of public key encryption and symmetric encryption provides fast message encryption with the benefits of certificate-based key management.

Similarly, digital signature algorithms are very slow and can only be used with small messages. But combined with message digests, they can be used to efficiently sign large messages. To sign a message, Alice computes the message digest of the message and signs that digest with her private key. The combination of message digests and digital signatures provides message integrity and sender authentication without shared keys.

1.7 A Simple Secure Messaging System

Let's start by designing a simple secure messaging system that would be suitable for sending secure e-mail messages. A large number of such protocols have been designed, including S/MIME [Dusse1998], PGP [Atkins1996], and PEM [Linn1993, Kent1993, Balenson1993, Kaliski1993]. All of them follow the same basic model, which we will now describe.

The sending process, shown in Figure 1.6, closely follows the description of the previous section.

1. Alice computes the message digest.
2. She signs the message digest and attaches the resulting digital signature plus her certificate to the message.
3. She produces a random session key and uses it to encrypt the signed message, certificate, and signature.
4. Finally, she encrypts the session key under Bob's public key and attaches the wrapped session key to the message. She now has a message she can send to Bob.

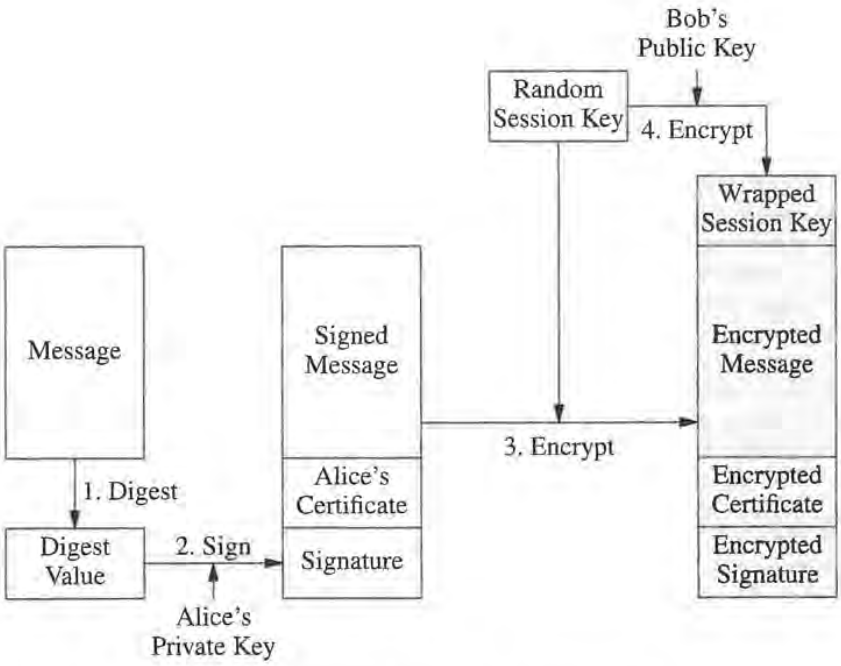


Figure 1.6 The message sending process

The receiving process, shown in Figure 1.7, reverses these steps.

1. Bob uses his private key to decrypt the session key.
2. He uses the session key to decrypt the message, certificate, and digital signature.
3. He computes the message digest of the message himself.
4. He verifies Alice's certificate and extracts Alice's public key.
5. He uses Alice's public key to verify Alice's digital signature.

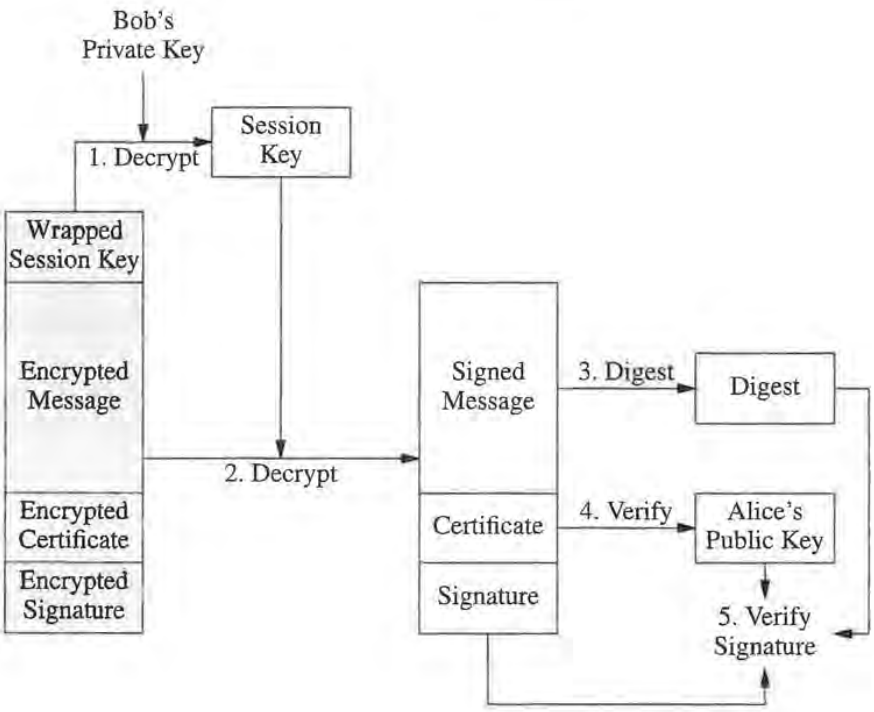


Figure 1.7 The message receiving process

1.8 A Simple Secure Channel

The system we've just described works pretty well if all we're doing is sending single messages around, as with e-mail, but it rapidly becomes inadequate if what we want is to have a communications channel over which we can pass arbitrary messages. We'd like to be able to establish a set of secret keys that we could use for the entire connection. This would let us avoid doing expensive public key operations for every packet,

which is especially important for interactive applications where each keystroke might generate a packet.

A second improvement we'd like to add in an interactive protocol is the ability to do *certificate discovery*. In the messaging protocol we described in the previous secion, Alice needs to be able to get Bob's certificate before she can send him a message. This is done either by consulting a directory or by having Bob send a nonencrypted message containing his certificate. In an interactive protocol, though, Bob can send Alice his certificate as the first thing he does.

A typical design for such an interactive system is to have a handshake phase where Alice and Bob authenticate each other and establish a set of keys. They then move on to a data transfer phase where they use those keys to actually transmit the data they're interested in. The rest of this section builds a simple protocol that does this job. We'll call it *Toy Security Protocol* (TSP).

The basic stages in our protocol will be

Handshake. Alice and Bob use their certificates and private keys to authenticate each other and exchange a shared secret.

Key derivation. Alice and Bob use the agreed upon shared secret to derive a set of cryptographic keys which can be used to protect the traffic.

Data transfer. The data to be transmitted is broken up into a series of *records*, each of which is individually protected. This allows data to be transmitted as soon as it is ready and processed as soon as it is received.

Connection closure. Special protected closure messages are used to securely close the connection. This prevents an attacker from forging closes and truncating the data being transferred.

A Simple Handshake

The first thing that Alice does is to send Bob a message saying that she's ready to communicate. There's no cryptographic content here. It's just a "Hello." Bob responds with his certificate. We've now accomplished certificate discovery, so we could go back to sending around individual messages like we did before, but there's a better way; what we actually want to do is arrange that Alice and Bob share a single secret, which we'll call the *master secret* (MS). Afterwards, Alice should know that only Bob knows the MS and vice versa. Then we'll use the MS to create a set of keys that we can use to encrypt the data.

The first step is for Alice to generate a random number to be the MS. Actually doing this requires having a *cryptographically secure* random number generator, which is trickier than it sounds. There are a number of techniques, but they border on magic and we won't discuss them here. Once Alice has the master secret, she simply encrypts it under Bob's public key to produce the *encrypted master secret* (EMS). We've now done the first half of the job. Alice knows that only she and Bob have the MS. If Bob doesn't care about her identity, she could just send the EMS to Bob and they could start

communicating. This is called *one-way authentication* (see Figure 1.8.) This sort of authentication would be useful (for example) if Bob were running an Internet store and only needs Alice's credit card number.

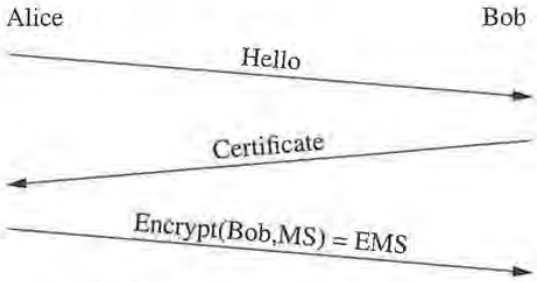


Figure 1.8 A handshake with one-way authentication

On the other hand, if Bob does care about Alice's identity, we need to add one more feature to the protocol to let him have that assurance. If Alice signs the EMS with her private key, then Bob can verify the signature and be sure that Alice vouches for the EMS she just sent.

This new protocol also contains one new subtlety: Bob sends Alice a random number called a *nonce* along with his certificate (see Figure 1.9.) The idea here is to stop an attacker from retransmitting all of Alice's messages to Bob and convincing him that he's engaging in a new conversation. This attack is known as a *replay attack*. We'll use the nonce when we make keys to ensure that the keys produced by this handshake are different from the keys produced by any other handshake.

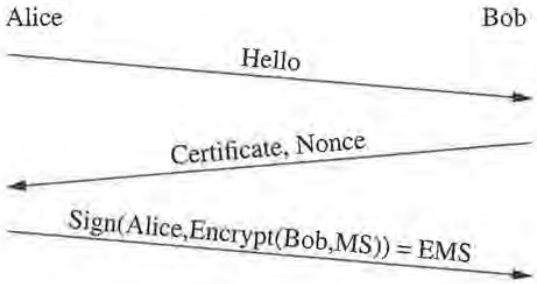


Figure 1.9 A handshake with mutual authentication

A Simple Data Transfer Protocol

At the end of the handshake, Alice and Bob share a master secret, but what good is that? The purpose of the exercise is for Alice and Bob to transfer data, not random numbers. In order to achieve that goal, we need an entirely different protocol whose job it is to arrange that. The rest of this section describes how to build a simplified one.

Making Some Keys

The first thing we need to do is to make some keys. In general, it's considered a bad idea to use the same key for more than one type of cryptographic operation. We'll see one reason for this in the discussion of stream ciphers later on, but you can consider it a form of insurance. Imagine that the attacker figured out how to break one algorithm (say the encryption algorithm). If you use the same key for MAC and encryption and the encryption is broken, then the communication is totally insecure. If you use different keys, then the MAC will still be secure even if the encryption is broken.

In this case we need four keys, one for encryption in each direction and one for MACs in each direction. In order to do this we need a tool we haven't seen yet, which is typically called a *key derivation function* (KDF). A KDF takes the master secret and (usually) some other random data and creates the appropriate keys out of it. In this case, the KDF takes the master secret and the nonce that is transmitted by Bob. KDFs are usually constructed using message digests and we'll skip the details here.

For convenience, we'll label these keys as follows:

E_{cs}—encryption key for data sent from client to server
M_{cs}—MAC key for data sent from client to server
E_{sc}—encryption key for data sent from server to client
M_{sc}—MAC key for data sent from server to client

Data Records

The next task is to describe how we're going to package the data. If we're working over a reliable protocol like TCP, we might imagine that we could just encrypt the data in a constant stream as we write it to the network. But then where would we put the MAC? If we simply transmit it at the end, then we don't have any message integrity until we've processed all the data. This isn't really any good because we can't safely act on received data if we don't know it hasn't been tampered with.

The solution to this problem is to break up the data into a series of records, each of which carries its own MAC. When we've read a record and checked its MAC, then we know that it's ok and we can act on it. What we're about to describe is the minimal record format that does this job.

When we read data off the wire, we need to be able to tell which data bytes are the encrypted data and which are the MAC. If we were willing to commit to a constant amount of data in each record, we could simply know which was which, but this doesn't work very well: in some circumstances, we want to be able to transmit one byte at a time, and having one-byte-long records would mean that we expanded the data

something like twenty times when we transmitted it. This really isn't acceptable if we want to transmit a lot of data in bulk.

Since fixed-length records are so inefficient, we need some way to have variable length records. This means that we have to have a length field to tell us where the record data ends. This field has to come before the record so we know how much data to read. The MAC can come before or after, but it typically comes after to allow it to be appended to the message after it's computed. (This is a matter of programming convenience). This leaves us with the record format shown in Figure 1.10.

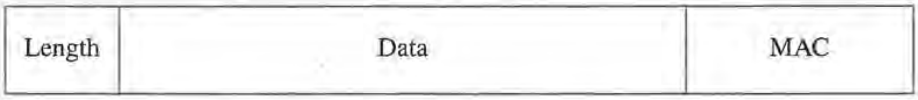


Figure 1.10 A simple record format

So, when the client wants to encrypt a data block D of length L, it goes through the following process:

1. Compute a MAC over the data using M_{cs}. We'll call this MAC M.
2. Encrypt D using E_{cs}. We'll call this C. We're assuming here that encrypting the data doesn't expand it. As we'll see in Section 1.13, this sometimes isn't true.
3. Transmit $L||C||M$ (the || stands for concatenation).

To read the data, the server goes through the reverse process.

1. Read L from the wire. We now know that D is L bytes long.
2. Now read C and M from the wire.
3. Decrypt C using E_{cs} to recover D.
4. Compute M' over D using M_{cs}.
5. If $M' = M$ then everything is ok. Process the record. Otherwise report an error.

In other words, the MAC computation is as shown in Figure 1.11 where x is either *cs* or *sc* depending on which direction we're sending the data.

Sending:
$M = MAC(M_x, D)$

Receiving:
$M' = MAC(M_x, D)$
compare M to M'

Figure 1.11 MAC computation

Sequence Numbers

Unfortunately, this simple protocol has a security hole. Because the records aren't labelled by the order they're transmitted in, an attacker can take a record off the wire and send it again to the receiver. (Remember, this is called a *replay attack*.) To see why a replay attack might be used, consider the case in which the messages are financial transactions and the attacker might want to arrange to be paid twice. Without protection against replay attack, the attacker could simply replay the message requesting payment.

The current protocol also allows the attacker to remove records or reorder them. It's easy to think up situations in which reordering or replay attacks might be useful and we'll leave it as an exercise for the reader. In order to fix this problem, we need to use a *sequence number*. The first record that each side transmits is numbered 1, the second 2, etc. When you receive a record, you check that its sequence number is the one you expect. If it's not, you report an error.

Of course, the sequence number has to be part of the input to the MAC in order to prevent the attacker from changing the sequence number. The easiest way to do this is simply to prepend the sequence number to D before it's encrypted and MACed. However, if we're communicating over a reliable protocol (like TCP), then the sequence number is implicit and we don't even have to transmit it. Messages will always be delivered in order, so we can just maintain a counter on either side and use it as part of the MAC computation:

$$M = MAC(M_x, Sequence \| D)$$

Note that the sequence number doesn't prevent the attacker replaying *all* of Alice's messages. However, because Bob generates a new nonce for each handshake with Alice, if the attacker tries this attack, Bob will generate a different set of keys (because his nonce will be different) and so when he goes to decrypt the data records, he'll get garbage rather than the plaintext, and the MAC check will fail.

Control Information

We have one more security problem to deal with. Remember that it's easy for the attacker to forge packets. Well, TCP connection closes are just another packet, so it's easy for the attacker to forge connection closes. That means that the attacker can mount a *truncation attack*, in which the attacker convinces one side (or both) that there is less data than there actually was. In order to fix this problem, we need to have some way for Alice to tell Bob (or vice versa) that she's done sending data. Then, if the attacker forges a TCP close, the victim can tell that something is wrong because the end of data message never arrives.

There are a number of tricks we could use to prevent this attack. For instance, we could say that a zero-length record means that you're done. However, in general it would be nice if we had some place to transmit *control messages* that aren't part of the data stream. For instance, we might want to have some way to report errors. An easy way to accomplish this is to have each record be *typed* by what kind of data it carried, regular or control. We can do this by adding a *type field* to each record, and the type field has to be MACed in order to protect it, as shown in Figure 1.12.

Ex.1024
Askeladden, L.L.C. / Page 92 of 788



Figure 1.12 Improved record format

The MAC would then be computed using:

$$M = MAC(M_s, Sequence||Type||D)$$

Of course, the type field isn't much good if we don't define some types. We'll go with something simple. If the type field is 0, that means just treat the record as if it were ordinary data. If it's 1, that means that this record is control information. The data portion of the record needs to be scanned by the protocol to figure out what to do next. We'll adopt a simple convention that all control data consists of a simple number. We'll reserve 0 to indicate that the connection is closing, and use nonzero numbers to indicate errors, as shown in Figure 1.13.

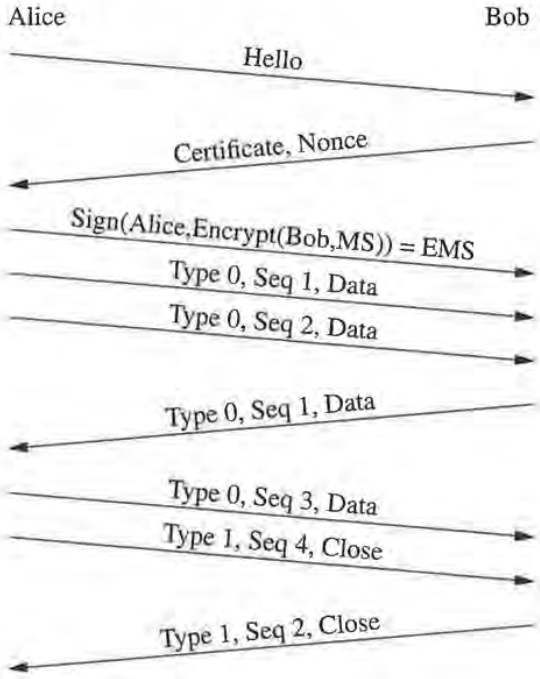


Figure 1.13 Full Toy Security Protocol (TSP) connection

Summary

As the astute reader will probably have guessed, we've now reproduced most of the essential features of SSL. We've got handshaking, key exchange, mutual authentication, and secure data transfer. What's left?

First, TSP isn't complete. We haven't specified enough detail for someone to actually implement it. Some of these details aren't that important. For instance, how long is the length field? Two bytes? Three? Variable? This governs the maximum record length, so it's not unimportant, but mainly it has to be big enough—and we all have to agree. Some of the other details are a little more tricky. Exactly what algorithms are we going to use for key exchange, authentication, etc.? In any case, we'd have to fill in a lot of detail before this protocol would even work.

TSP is skimpy on details, but more important, it is missing some desirable features. The most important thing it's missing is *negotiation*. We'd like to be able to support a number of different algorithm choices for each essential task. That way, if one algorithm is broken, we can switch over easily. We'd also like to be able to choose whether clients are authenticated or anonymous. Nevertheless, it's an interesting and instructive toy and captures most of the idea.

1.9 The Export Situation

Because so much computer software is written in the United States, the U.S. export situation has had a dramatic impact on the design of communications security systems. Export policy in the U.S. has been historically driven by the National Security Agency (NSA). NSA has responsibility for securing the government's communications and snooping on the communications of others. The actual enforcement of export policy is always left to someone else, originally the State Department and now the Bureau of Export Administration (BXA) of the Commerce Department, but all the important decisions are made by NSA.

The Stone Age

Up until September of 1998, the general idea behind export policy was simple. It was legal to export encryption technology for use in authentication, but confidentiality was severely restricted. If your product included any confidentiality at all, it had to be reviewed before export. The rules for what would pass review were never published but it was generally understood that the limits were 40-bit-strong encryption and 512-bit-strong key exchange.

Provided you met these criteria, you could get what was called *commodities jurisdiction*, which basically meant that you could apply to the Department of Commerce for a general license to export your software without having to seek approval. If you wanted to sell your software widely as a mass market product or have it downloaded over the net, this was the approval you needed. Even with this classification, you still couldn't export your software to a few embargoed countries: Cuba, Iran, Iraq, Libya, North

Korea, Sudan, and Syria. Also, some countries such as France heavily restricted cryptography.

The NSA also had a special approval process for a few ciphers (RC2 and RC4). If you used one of them (in 40-bit mode, of course) with key exchange of no stronger than 512 bits, then your review went faster and you were almost certain to get approval. It was possible to get approval for other algorithms and IBM got approval for a reduced-key DES called CDMF [Johnson1993] but in general things went more smoothly if you used RC2 or RC4. It was widely assumed that NSA had done extensive analysis of RC2 and RC4 and perhaps knew how to break them quickly, either via cryptanalytic attack or custom hardware. In any case, NSA made it easy to export RC2 or RC4 and difficult to export other ciphers. This is why you see RC2 and RC4 in so many systems.

There was one notable exception to the general weak cryptography rule. If the system you were exporting was solely for financial transactions (this basically meant it would be used only by banks), it could use stronger cryptography.

Medieval Times

In September of 1998 the encryption rules were amended to allow the export of 56-bit encryption and 1024-bit key exchange. Products were still required to undergo review, but they could at least use semi-strong algorithms. However, at the time DES was known to be weak, so it still wasn't permitted to export really strong algorithms.

The Modern Age

Many times after the 1998 relaxation, rumors were floated that the export controls would be repealed completely. Finally, in January 2000, they were substantially relaxed. Publicly available source code ("open source" software) could be simply posted on the net. "Retail" software still had to undergo a one-time technical review (it's not clear what this was really for), but in practice the BXA has allowed the export of a number of commercial products with strong cryptography. Thus, in practice it is now legal to export strong cryptography from the U.S., except to the embargoed seven countries mentioned earlier.

Despite these liberalizations, the impact of export controls lives on. Many protocols contain features designed to let them work in an exportable environment. SSL is no exception. First, it contains some weak algorithms as well as strong ones so that SSL implementations could be exported. Second, it contains a number of hacks such as ephemeral RSA and Server Gated Cryptography (both discussed in Chapter 4), which are designed to preserve as much security as possible while remaining exportable.

1.10 Real Cryptographic Algorithms

So far, we've been acting as if all cryptographic algorithms of a given type were more or less the same. Unfortunately, that's not true. For instance, each digital signature algorithm is slightly different, as is each key exchange algorithm and each symmetric

encryption algorithm. These details turn out to be security-relevant, as we'll see in Chapter 5. In the rest of this chapter we'll give a brief overview of the most popular algorithms so that when they're discussed later in the text you'll be familiar with them.

If you're not interested in the details, you can completely skip this section and read the summary at the end of the chapter. When we talk about security issues in other sections of the text, you should be able to understand the implications of the discussion without understanding the algorithm details.

1.11 Symmetric Encryption: Stream Ciphers

Symmetric ciphers come in two major variants, *stream ciphers* and *block ciphers*. Stream ciphers are the easiest type to understand, so we'll cover them first. The way that a stream cipher works is very simple. You have a function that generates a stream of data one byte at a time. That data is called the *keystream*. The input to the function is the encryption key, which controls exactly what keystream is generated. Without the key, you can't predict the keystream. You take each byte of keystream and combine it with a byte of plaintext to get a byte of ciphertext. Exactly how you combine them turns out not to matter much, but the most popular way is to use exclusive-or (XOR).

$$C[i] = KS[i] \oplus M[i]$$

$$M[i] = KS[i] \oplus C[i]$$

We'll be using this notation throughout the rest of the chapter, so we'll take this opportunity to explain it. $C[i]$ refers to the ith unit of ciphertext. (In this case unit means byte.) $KS[i]$ refers to the ith unit of the keystream and $M[i]$ refers to the ith unit of message. $\oplus$ is the symbol for exclusive-or. So, what this means is take the ith byte of keystream and XOR it with the ith byte of message to get the ith byte of ciphertext. $A \oplus B \oplus A = B$, so we can decrypt the plaintext simply by XORing the ciphertext with the key.

Unfortunately, the symmetry of encryption and decryption causes stream ciphers to have some dangerous security properties. The worst of these is that you can never *ever* reuse the same section of keystream to encrypt two different messages. Imagine that we've encrypted two messages M and M' with the same key. If the attacker learns M, he can easily compute KS simply by computing $M \oplus C$. Once the attacker knows KS, he can compute M' given C':

$$M' = (M \oplus C) \oplus C'$$

This situation occurs surprisingly often because many communications formats contain large amounts of predictable and repetitive data which allows the attacker to guess a large amount of a given message. If another message is encrypted with the same key, the attacker can then leverage this knowledge to recover that message as well.

Worse yet, the attacker can mount an attack even if he doesn't know plaintext of either message. XORing the two ciphertexts together eliminates the effects of the key entirely:

$$C \oplus C' = M \oplus M'$$

Because there are well-known methods for decrypting two plaintexts given their XOR, having two such ciphertexts is disastrous. If you want to encrypt multiple texts with a stream cipher, you must either use separate keys or different sections of keystream.

Even if you use a keystream only once, it's still trivially easy to tamper with data encrypted with a stream cipher. Imagine that you, the attacker, know the text of a message M and the ciphertext C, and you want to change the message so that it decrypts to a new message M', it's trivial to compute the ciphertext C' that achieves this.

$$C' = C \oplus (M \oplus M')$$

Even if you don't know the plaintext, you can make predictable changes in it because each ciphertext bit maps one-to-one with a plaintext bit. Consequently, any plaintext bit can be flipped by changing the corresponding bit in the ciphertext. As a consequence, if you use a stream cipher, you absolutely must use a strong MAC along with it.

RC4

The only stream cipher to have received widespread attention and use is RC4. RC4 was designed by Ron Rivest and was used for quite some time as a proprietary cipher in RSA Data Security's (RSADSI) products. In 1994 someone anonymously posted a cipher that he claimed to be RC4 to the Cypherpunks mailing list. Subsequent testing has demonstrated compatibility between the cipher and RSA's RC4 implementation, and it is widely assumed that the cipher was either reverse engineered from a disassembled version of RC4 code or that someone leaked RSA's source. Because RSADSI has a trademark on the term RC4, the cipher is generally referred to as Alleged RC4 or Arcfour.

RC4 is a variable-key-length cipher, with a key that can be anywhere between 8 and 2048 bits long. Whatever the key length, the key is expanded into an internal state table of constant size, and so the algorithm runs equally quickly with any key. SSL and TLS always use RC4 with a 128-bit (16 byte) key length. RC4 is extremely fast; a Pentium II/400 can achieve speeds on the order of 45 MB/s.

1.12 Symmetric Encryption: Block Ciphers

The other major type of symmetric cipher is a *block cipher*. A block cipher can be thought of as a huge lookup table. The data to be encrypted is processed in blocks of bytes (typically 8 or 16) and each possible plaintext block corresponds to a row in the table. The key is used to select a column in the table. So, to encrypt a block, you find the column corresponding to the key, run down the table to find the row corresponding to the block you want to encrypt, and the entry at that point is the ciphertext. Of course, such a table would be unmanageably huge, so in practice you've got a function that does the computation for you, but the idea is to have a function that simulates a randomly permuted table.

Another way to look at this is that a block cipher is a function of two variables, the key and the input block. You have two functions E (for encryption) and D (for decryption). M is the plaintext and C is the ciphertext.

$$C = E(K, M)$$
$$M = D(K, C)$$

What we've got so far gives us the ability to encrypt only eight bytes or so. Usually the message we want to encrypt is far longer than that, so we've got to find a way to do that. The obvious way is what's called *Electronic Codebook* (ECB) mode. Simply speaking, we break the message up into block-sized chunks and individually encrypt it using our encryption algorithm:

$$C[i] = E(K, M[i])$$
$$M[i] = D(K, C[i])$$

This is straightforward, but it has an obvious disadvantage. Imagine that we have two blocks $M[j]$ and $M[k]$ that are the same. In that case, $C[j]$ and $C[k]$ will also be the same. If we've got a pattern that appears frequently, the attacker can detect this and learn something about our plaintext. We'd rather not have this property.

Cipher Block Chaining (CBC) mode fixes this problem. In CBC mode, the encryption of each plaintext block depends on the ciphertext for the previous block of ciphertext. This is accomplished by XORing the previous ciphertext block with the plaintext before encryption:

$$C[i] = E(K, M[i] \oplus C[i-1])$$
$$M[i] = D(K, C[i]) \oplus C[i-1]$$

In CBC mode, even if we have two plaintext blocks that are the same, they most likely won't encrypt to the same result because the previous ciphertext blocks will be different. Consider the case where $M[j]$ and $M[k]$ are equal, but $M[j-1]$ and $M[k-1]$ are not. In that case, $C[j-1] \neq C[k-1]$ and so $C[j] \neq C[k]$.

The only remaining question is what to do with the first ciphertext block. Since there is no preceding block of ciphertext to XOR in, we can't follow our usual procedure. We could just use a block of all zeros, but then attackers would be able to tell if the first cipher blocks of two messages were identical. To solve this problem, we generate a random block, called an *initialization vector* (IV). The IV is XORed with the first cipher block of the message before encryption. Typically the IV is just sent with the message, but it can also be generated from some value shared by both sides. IVs do not have to be secret, but they should be *fresh*, which is to say different for each message.

There are a large number of other modes for operating block ciphers in, including *Output Feedback* (OFB) and *Cipher Feedback* (CFB), but CBC is by far the most popular one and is the only one relevant for SSL.

Block ciphers in CBC mode always expand the data a little bit, whereas stream ciphers don't. The reason for this is that the data input must be a multiple of the block size. Since most data isn't a multiple of the block size, you need to *pad* the final block

out until it ends on a block boundary. This is slightly tricky, because when you decrypt you want to be able to remove all the padding, but nothing else. The way this is typically done is to pad the message with a byte value equal to the number of pad bytes added. If the message actually ends on a block boundary, you have to pad with a full block. The data expansion isn't a problem when encrypting large messages, but if you encrypt one byte at a time then you'll see a lot of data expansion as each byte in produces an encryption block out. This can waste network bandwidth.

Block ciphers in CBC mode are easier to use safely than stream ciphers. It's perfectly safe to use the same key for different data as long as you change the IV. Even if you don't change the IV, pretty much all the attacker learns is that two messages are the same, whereas reusing a key would totally compromise a stream cipher.

Block ciphers perform better under integrity attacks too. The attacker can't make predictable changes to the plaintext by changing the ciphertext. Any change damages both the block that was changed and the next block. An important exception to this rule is the first block. If the IV is in the message and isn't integrity protected, then the attacker can make predictable changes to the first block by changing the IV. Because the IV is XORed in, the attacker can flip any bit of his choosing by flipping the same bit in the IV. In general, it's still important to use a MAC when encrypting with block ciphers. However, the attacker can't mount as precise an attack on block cipher-encrypted data as on stream cipher-encrypted data.

When using CBC mode, you need to worry about *CBC rollover*. Imagine that two data blocks $M[i]$ and $M[j]$ encrypt to the same value C. If $M[i+1] = M[j+1]$, then $C[i+1] = C[j+1]$. Thus, the attacker will be able to know that $M[i+1] = M[j+1]$. On average, a cipher with an X-bit blocksize will have two ciphertext values collide every $2^{X/2}$ (2^{32} for DES) blocks. Thus, it's important not to encrypt more than this amount of data with a given key.

DES

By far the most widely used symmetric cipher is the *Data Encryption Standard* (DES). DES was designed by IBM in the 1970s in response to a request for proposals by the National Bureau of Standards (NBS), now the National Institute of Standards and Technology (NIST). Although IBM had patented DES, they agreed to make it freely available and it was standardized in [NIST1993a].

DES is a 64-bit block cipher with a 56-bit key. This means that data is encrypted in blocks of 8 bytes and the key space is 56 bits. DES keys are actually 64 bits long, however, with the low order bit of each byte used as a parity check to attempt to detect transmission and key wrapping/unwrapping errors. In practice, because DES keys are usually wrapped with public key techniques, the parity check is redundant and the parity bits are usually ignored.

DES is certainly the most widely analyzed public cipher. When it was proposed, it was known that NSA had provided design assistance and some of the design decisions were both unobvious and unexplained. The IBM cryptographers who developed DES were silent on what attacks they knew about, and it was clear that they knew about potential weaknesses that the rest of the world didn't. Naturally, there was concern that DES had been intentionally weakened.

Early attention focused on a particular part of the algorithm called the S-boxes, and for a long time there was concern that the S-box structure had been designed to provide the NSA a *back door* for cryptanalysis. Finally, in 1990–1993, Eli Biham and Adi Shamir used their *differential cryptanalyis* technique on DES [Biham1991a, Biham1991b, Biham1993a, Biham1993b] and it became clear that the DES S-boxes had been optimized to resist differential attack. Only extremely paranoid people still worry about the design of the S-boxes.

No really good analytic attack has ever been found against DES. However, DES is now quite weak, basically as a consequence of having too short a key (56 bits) and of computers getting faster. In 1997 a distributed effort (dividing up the search space between a large number of machines on the Internet) recovered a single DES key using exhaustive search and in 1998 Deep Crack, a custom-built DES search machine, recovered a DES key in 56 hours. As of this writing, the record is below 24 hours. DES is now useful only for low-value or short-lived information.

3DES

Because DES has withstood such aggressive analysis, when the key length became too short a very attractive prospect was just to run the data through more than once, a process called *superencryption*. Unfortunately, just using DES twice (2DES) turns out not to be that much more secure than DES. There's an attack called the *meet-in-the-middle* attack that allows you to break 2DES in the same time as DES if you have 2^{56} blocks of memory to work with. As a consequence, people were forced to encrypt the data three times (3DES). 3DES has an effective strength of 112 bits, the strength that you would naively expect 2DES to have.

> Actually, there's a time-memory tradeoff for the meet-in-the-middle attack. If you're willing to use more CPU time, you can reduce the memory requirement. The time-memory product remains at 2^{112}, however (see [Menezes1996]).

What we've just said doesn't completely specify how 3DES works: not all three operations have to be encryption. The most popular version of 3DES uses what's called *Encrypt-Decrypt-Encrypt* (EDE) mode. What this means is that you encrypt with Key 1, Decrypt with Key 2, and Encrypt with Key 3. This turns out to be just as safe as *Encrypt-Encrypt-Encrypt* (EEE) and has one major advantage: If you set all three keys equal, this is the same as single DES, which means that 3DES hardware can be made to interoperate with DES hardware. This variant is usually referred to as 3DES-EDE.

3DES is (not surprisingly) about three times slower than DES. DES wasn't very fast to start with, and so people are often wary of using 3DES in performance-critical applications. Also, using 192 bits of keying material (168 bits of key and 24 bits of parity) to get 112 bits of security seems unaesthetic to many people.

It's possible to use 3DES with two keys (the two encrypts are done with one key and the decrypt with the second) but there is a known weakness on two-key 3DES that requires $O(t)$ space and $2^{120-\lg(t)}$ operations. This attack isn't really practical, but it can't be mounted at all against three-key 3DES. Thus, conservative practice is to use three-key 3DES.

RC2

RC2 is a block cipher invented by Ron Rivest, who also invented RC4. Like RC4, RC2 was a trade secret of RSADSI, and like RC4 it was eventually published by persons unknown. Subsequently, Ron Rivest published an RFC describing RC2. However, because RSADSI holds a trademark on the name RC2, the RFC refers to it as RC2(r).

RC2 is a variable-length cipher with a twist. It also has a variable *effective key length*. So, you can use it with a 64-bit key but it can be broken with 2^{40} operations. This makes it convenient for export reasons. Like DES, it has a 64 bit block size. In SSL, RC2 is always used with a 128-bit key and a 128-bit effective key length.

AES

As mentioned previously, although DES has no known severe cryptographic weaknesses, the key length is too short, and it's long past time to replace it. In 1997 NIST put out a call for submissions for an *Advanced Encryption Standard* (AES). The AES is to have a minimum block size of 128 bits and three key lengths of 128, 192, and 256 bits, making it strong enough for the foreseeable future. Moreover, AES algorithms are supposed to be fast in software and nearly all the submissions are faster than DES.

As of this writing, NIST has narrowed the field to five finalists, MARS [Burwick1999], Serpent [Anderson1999], Twofish [Schneier1998], Rijndael [Daemen1999], and RC6 [Rivest1995]. NIST is expected to determine a final standard in 2000. One of the requirements for submission is that the algorithm must be offered for use on a royalty-free basis if chosen, so use of the AES will be free.

Cipher	Key Length	Speed(MB/s))
DES-CBC	56	9
3DES-CBC	168	3
RC2-CBC	variable	.9
RC2-CBC	variable	3*
RC4	variable	45

* The faster RC2 benchmark is on Celeron 450 running Windows 2000[Dai2000]. The rest of the benchmarks were generated with OpenSSL.

Figure 1.14 Some common cryptographic algorithms (OpenSSL, FreeBSD, Pentium II 400)

Summary

Figure 1.14 summarizes the symmetric algorithms we've discussed. As you can see, RC4 is by far the fastest algorithm and 3DES and RC2 are really quite slow. Although RC2 and RC4 have variable key sizes, they run at the same speed no matter what key size you use. By contrast, 3DES is much slower than DES, but correspondingly more secure. Now that 128-bit ciphers have been cleared for export, there really isn't any reason to use RC2, except for compatibility with 40-bit systems, because it has neither security nor performance advantages over 3DES.

1.13 Digest Algorithms

As far as protocol design is concerned, all digest algorithms are very similar. The only difference is the size of the output. The two most popular algorithms are MD5, which was designed by (guess who) Ron Rivest, and SHA-1, designed by NIST (presumably with help from NSA).

MD5 and SHA-1 both derive from a common ancestor, MD4, also designed by Rivest. MD5 is basically a stronger version of MD4. It has a 128-bit output, as does MD4. SHA-1 is based on MD4, but has a 160-bit output. This makes it somewhat stronger. Because of what's called the *Birthday Paradox*, the difficulty of finding two messages that digest to the same value (a collision) is approximately the square root of the key space. Consequently, it requires about 2^{64} operations to find a collision with MD5 and 2^{80} operations with SHA-1. Recently, there have been some indications that there may be an easier way than brute force to find collisions in MD5 [Dobbertin1996]. Conservative designs all use SHA-1.

> The name *Birthday Paradox* comes from a mathematical party trick. If you have more than 23 randomly chosen people in a room, the probability that some two will have the same birthday is over 50%. This result surprises most people, who expect the required number of people to be around 128. The key insight is that we're computing the probability that *some* two people will have the same birthday, not that someone will have a given birthday, and that the space of birthdays that have already been assigned to someone gets depleted each time you add a person. A similar effect on a larger scale applies with hash collisions, where we're interested in the probabiliity that *some* two messages will have the same digest.

An older MD4-like algorithm called MD2 is sometimes used to sign very old certificates. No modern systems use it. There also used to be an older version of SHA, but it was revised to correct a weakness, creating SHA-1. Figure 1.15 compares MD5 and SHA-1.

Digest	Output size (bits)	Speed (MB/sec)
MD5	128	65
SHA-1	160	31

Figure 1.15 Digest algorithm comparison (OpenSSL, FreeBSD, Pentium II 400)

1.14 Key Establishment

The two primary uses of public key cryptography are key establishment and digital signatures. This section talks about key establishment and the next section talks about digital signatures.

There are really two flavors of key establishment. In *key exchange* or *key transport*, one side generates a symmetric key and encrypts it using the public key of the other

side. In *key agreement*, both sides cooperate to generate a shared key. We'll be talking about two algorithms, RSA and Diffie-Hellman (DH). RSA can be used as a key transport algorithm. DH is a key agreement algorithm.

RSA

RSA is the public key algorithm most people think of when they think of public key cryptography. It was invented in 1977 by Ron Rivest, Adi Shamir, and Len Adelman (hence RSA) [Rivest1979]. From a high level, RSA is very simple. Each user has a public key and a private key. The public key can be distributed freely and the private key must be kept secret.

RSA public keys actually consist of two numbers, the *modulus* (usually represented by the letter n) and the *public exponent* (usually represented by the letter e). The modulus is the product of two very large prime numbers, conventionally represented by the letters p and q. p and q also need to be kept secret. The security of RSA is based on the difficulty of factoring n to get p and q.

The private key is another number, usually called d, which can be computed only if you know p, q, and e. When we talk about the key length of an RSA key, we're talking about the length of the modulus. The RSA public exponent (e) must be relatively prime to e and $(p-1)(q-1)$. For convenience, e is usually chosen to be one of a few small prime numbers (usually 3, 17, or 65537). Having a small e makes operations performed with the public key faster. With e chosen, d is computed as

$$d = e^{-1} mod((p-1)(q-1))$$

To encrypt a message M using RSA and the public key (e,n), you compute $C = M^e mod\ n$. What the *mod n* means is that the computation is done *modulo n*, which means that you take M to the eth power and then divide by n and take the remainder. Decrypting the message requires the corresponding private key. For example, $2^5 mod\ 10 = 2$ and $2^5 mod\ 7 = 4$. To decrypt the message, you compute $M = C^e mod\ n$. Just take our word for it that this works.

One thing that this explanation glosses over is that it assumes that the message is a number, when actually it's just a string of bytes. You need some rules to convert the message into a number. This number should be more or less the same size as (but not greater than) n. The reason for this requirement is that we want to ensure that $M^e > n$. If this isn't the case, then $C = M^e mod\ n = M^e$ (i.e., the *mod n* doesn't do anything). It's easy for the attacker to recover M by taking the integer e^{th} root of C. On the other hand, if $M^e > n$, then this attack isn't possible. Making M approximately the same size as n ensures that $M^e > n$.

The standard procedure for how to do this conversion is specified in PKCS #1 [RSA1993b]. We'll summarize it here, because it turns out that there are security problems with this procedure when applied to SSL. PKCS #1 version 2 [Kaliski1998a] fixes these security problems but isn't currently in wide use.

The first thing to do is to compute how large the formatted encryption block needs to be. If n is an L-bit number, then the encryption block is $L/8$ bytes long (rounded up). The first (high order) byte is always 0. The second byte is set to the *block type*. For

Ex.1024
Askeladden, L.L.C. / Page 103 of 788

encryption, this is 2 and for signature it's 1. This ensures that the number formed is just slightly smaller than n.

The message goes in the low-order bytes of the encryption block and is prepended with a zero byte. (Figure 1.16 shows this padding in pictorial form.) The bytes in between are filled with *pad bytes*. For block type 1, the pad bytes have the value 255. For block type 2, the pad data consists of random nonzero bytes. The idea here is that when a message is signed repeatedly, the signature should be the same, but if you encrypt the same message repeatedly, the encryption block should be different. When you recover the encryption block, you can find where the data begins by starting at the third byte and moving to the right until you find a zero byte. This is the last byte of padding.

There must always be at least 8 bytes of padding for encryption to prevent plaintext guessing attacks. Without the padding, the attacker would try encrypting a plaintext guess with the public key to see if it produced the given ciphertext. With the padding, the attacker must try 2^{64} different padding combinations for each plaintext, which makes this attack much harder.

Once you have the encryption block formatted, you convert it into an integer by taking the first byte as the most significant and the last byte as the least significant. This is the same as *big-endian* byte order.

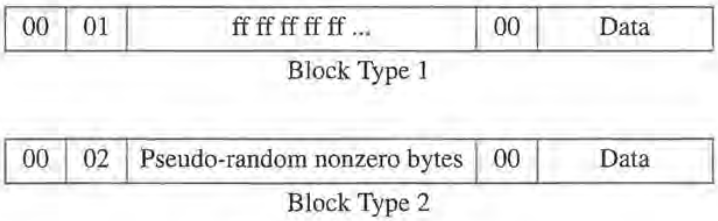


Figure 1.16 PKCS-1 padding

RSA was patented in the United States; however, the patent expired on September 20, 2000 [Rivest1983]. The patent was held by RSADSI. Despite this patent, RSA became the de facto standard for public key cryptography and has been licensed by nearly every company that does public key.

To use RSA for key transport, generate a random session key and encrypt that (appropriately padded) under the recipient's public key. The recipient then decrypts the message and removes the padding with the result that you now share a session key.

Diffie-Hellman

Diffie-Hellman (DH) is the first public key algorithm ever published [Diffie1976]. DH is a key agreement algorithm rather than a key exchange algorithm. Instead of the sender

generating a key and encrypting it for the receiver, the sender and receiver collectively generate a key that is private to them. The sender and the receiver each have key pairs. To compute the agreed key, the sender combines his private key with the receiver's public key. The recipient combines his private key with the sender's public key. A DH public key is often called a *share*, because it's one side's share of the key agreement.

DH also uses modular exponentiation, but the modulus is a large prime (usually called p). The modulus is publicly known and has to be shared between the sender and the receiver. There is also another number called the generator (g) which must be shared between sender and receiver. g is chosen such that for any value $Z < p$ there exists a value W so that $g^W \bmod p = Z$. Thus, it "generates" all the numbers from 1 to $p - 1$. To generate a key, you generate a random number (X) smaller than p and compute $Y = g^X \bmod p$. X is the private key and Y is the public key. We'll use Xs and Ys to refer to the sender's keys and Xr and Yr to refer to the receiver's keys.

To compute the shared key (ZZ), the sender computes

$$ZZ = Yr^{Xs} \bmod p = (g^{Xr})^{Xs} \bmod p = g^{Xr \cdot Xs} \bmod p$$

The recipient computes

$$ZZ = Ys^{Xr} \bmod p = (g^{Xs})^{Xr} \bmod p = g^{Xs \cdot Xr} \bmod p$$

It should be easy to convince yourself that these numbers are the same and that the sender and the recipient thus share the same number. It's more difficult to show that an attacker can't compute ZZ. In fact, it's not known that this is the case. However, it is widely believed that to do so would require having either Xs or Xr, which would require computing X given Y, a problem known as *discrete logarithm*. Currently, there are no known efficient ways of computing discrete logarithms.

The most important operational difference between RSA and DH is that in DH all the communicating parties must share g and p (collectively known as the *group parameters* or just *group*.) Often this means that a community of people will all use the same group. However, it is possible for the recipient to simply randomly generate his own group, in which case he must arrange to transmit g and p to the sender, often in his certificate. The sender generates a temporary key in the recipient's group, which he uses to encrypt to the recipient only. This is called *ephemeral-static* DH. If the recipient also makes up a key just for this transaction, it's called *ephemeral-ephemeral* DH.

Ephemeral-ephemeral mode has the advantage that if the parties delete their keys when they are done communicating, then even if their machines are subsequently broken into, the attacker cannot read any old traffic, since there won't be any private keys on the machine. If one of the keys is static, the attacker might be able to break into the machine with that key and recover it. This property is called *Perfect Forward Secrecy* (PFS). Note that with DH, unlike in RSA, the recipient must know the sender's public key to decrypt the message.

The strength of DH keys depends on two values, the size of p and the size of X. X is often chosen to be the same general size as p, and so only the size of p is relevant. However, X can be chosen to be much smaller than p for performance reasons. In that case, the length of X must be chosen to be about twice the length of the symmetric key you want to generate from ZZ. For instance, if you're agreeing on DES keys, X must be

at least 112 bits long. Note that this doesn't mean that X must be greater than 2^{112}, but merely that it must be randomly chosen over an interval at least as large as 2^{112}. It's also desirable that X be chosen so that its size makes it as strong against attack as p (see [Menezes1996]). Because attacking X is harder than attacking p, X can still be much smaller than p, however. For a 1024 bit p, an X of 160 bits would be quite appropriate. Note that this statement implies that a 1024-bit DH key is really too short to provide full security for a 3DES key. The DH key will be the weakest link in the chain. Similar comments apply to using 1024-bit RSA keys to secure 3DES keys.

1.15 Digital Signature

RSA

Using RSA for digital signature is almost exactly the same as using it for key transport, except with the roles of the public and private keys reversed. To sign, one computes a message digest and "encrypts" it using one's private key. To verify, the receiver "decrypts" the digest and compares it to the message digest he has independently computed on the message. If they match, the signature is valid.

The only part of RSA signature that's a bit obscure is the block formatting. Rather than just encrypting the message digest, you encrypt a DER encoded DigestInfo structure [RSA1993b]. The DigestInfo structure is a combination of the algorithm (e.g., MD5 or SHA-1) rendered as an *algorithm identifier*, which is just a specific series of bytes and the digest value itself (see Figure 1.17).

```
DigestInfo ::= SEQUENCE {
  digestAlgorithm DigestAlgorithmIdentifier,
  digest Digest }

DigestAlgorithmIdentifier ::= AlgorithmIdentifier

Digest ::= OCTET STRING
```

Figure 1.17 DigestInfo structure

To see why things are done this way, consider what would happen if we signed the digest directly. Imagine that there exists digest algorithm $H1$ that was widely used for signing. Now, suppose that $H1$ is catastrophically broken—that is, it became possible to generate random messages that digested to a chosen value. Now, suppose further that we had signed a message M with a different digest algorithm $H2$, which was still strong. The attacker could generate a new message M' with the property that $H1(M') = H2(M)$. If he attached our signature to his new message M' (labelled as being signed with the broken hash $H1$) the signature would still verify! Thus, the

attacker would have succeeded in generating a message of his choice that apparently was signed by us. Including the identifier of the digest in the signature itself stops this attack.

DSS

The *Digital Signature Algorithm* (DSA) was invented by NSA and proposed by NIST in 1991. The idea seems to have been to have an algorithm which could be used for digital signature but not for key establishment. This algorithm was intended to replace RSA, which could be used for both. This would have satisfied the NSA's goal of having widely available authentication technology but restricted encryption technology. The DSA was standardized as a *Federal Information Processing Standard* (FIPS-186) called the *Digital Signature Standard* (DSS), and DSA is now widely known as DSS.

DSS is based on the same cryptomath as DH: modular exponentiation in a prime field. We won't bother to describe it, because it's fairly complicated and the details aren't that relevant. The important thing to know is that the keys are more or less the same as those for DH with one notable difference: p (the large prime) is constructed such that $p-1$ is divisible by another (smaller) prime called q. This allows the algorithm to be made faster.

DSS signatures consist of two large (160 bit) numbers called r and s. Unlike RSA, which can be used with any digest, DSS can be used only with SHA-1. (Actually, it could be used with any 160-bit digest, but the standard requires SHA-1). DSA signatures do not include a digest identifier, so allowing multiple digests would open DSA up to the substitution attack described previously.)

It is important to know that DSS differs from RSA in the general way that the signature is verified. With RSA, you recover the message digest from the signature and compare it to the computed message digest. DSS requires you to perform a computation based on the message digest and the signature and returns a yes or no answer. There is no way to recover the sender's calculated message digest using DSS. This has led to confusion in converting RSA-based protocols and implementations to DSS.

The patent status of DSS is unclear. NIST claims that it's free, but RSADSI has claimed that it's covered by Claus Schnorr's patent [Schnorr1991], to which RSADSI happens to have exclusive license rights.

Summary

Figure 1.18 summarizes the performance of RSA, DH, and DSS using Wei Dai's Crypto++ [Dai2000] on a Celeron 450 running Windows 2000 Beta 3. The first thing to notice is that RSA public key operations (encryption and signature verification) are much faster than private key operations (decryption and signature). This isn't an inherent property of RSA, but it might as well be. It's actually a property of the way people choose RSA keys, but because it makes public key operations much faster, pretty much everyone chooses keys the same way.

DSS performance is much more symmetric, with the performance for private and public operations being worse than RSA private operations for small key sizes but better

Ex.1024
Askeladden, L.L.C. / Page 107 of 788

Algorithm	Key length	ms/operation
RSA encryption	512	.2
RSA decryption	512	4
RSA encryption	1024	1
RSA decryption	1024	27
DH agreement	512	5
DH agreement	1024	19
RSA signature	512	4
RSA verification	512	.3
RSA signature	1024	27
RSA verification	1024	.7
DSS signature	512	4
DSS signature (w/ precomputation)	512	2
DSS verification	512	5
DSS signature	1024	15
DSS signature (w/precomputation)	1024	5
DSS verification	1024	18

Figure 1.18 Public key algorithm comparison (Celeron 450)

for large key sizes. Because DH is the same on both sides, performance is usually symmetric (though see a special case in Chapter 6 where it is not). In general, if you're worried about performance, RSA is a better choice. However, if you're worried about performance, you may well want to purchase cryptographic hardware, in which case the speed difference isn't relevant.

1.16 MACs

The only MAC algorithm we'll be using is HMAC [Krawczyk1997], which uses digest algorithms to construct a MAC with provable security properties. Before HMAC was introduced in 1996, protocol designers used a variety of ad hoc MAC constructions, but were never very confident of their security. The sole exception to this was an algorithm based on DES called DES-MAC [ANSI1986], which was widely analyzed but quite slow. HMAC is quickly replacing all those other algorithms in most applicatons.

HMAC uses a nested keyed digest. What this means is that we compute a digest whose input is both the key and the data and then we use that digest value as the input to another keyed digest. The actual algorithm is

$HMAC(K, M) = H(K \oplus opad || H(K \oplus ipad || M))$
H = the digest algorithm we're using
ipad = a string consisting of the byte 0x36
opad = a string consisting of the byte 0x5c

K is 64 bytes long for all common message digests. If your K is shorter than 64 bytes, you pad it to the right with zero bytes.

1.17 Key Length

The most common way to talk about the strength of an algorithm is by talking about its key length. This turns out to be fairly deceptive because not all key lengths mean the same thing. Nevertheless, it's a useful metric and it's the one that's most commonly used. What we're trying to achieve here is to have an idea of the computational resources needed to attack a cipher. The key length provides an upper bound on how hard that is. It measures the difficulty of breaking it by *brute force*, which simply means throwing computational power at breaking the system.

It is important to remember that computers are getting faster, so algorithms gradually get weaker as the computers used to attack them get better. It is commonly estimated that computer speed doubles every 18 months. This observation is referred to as *Moore's Law* in honor of Gordon Moore, who first noticed the phenomenon. It's widely expected that eventually the rule of performance growth will start to slow down, but a number of predictions as to when have been wrong and it's certainly not safe to predict that speeds will stop increasing any time soon.

> Actually, what Moore observed was that the number of components on a chip doubled every 18 months, but Moore's Law has grown to refer to a set of rules of thumb about performance trends in computers.

Symmetric Algorithms

The only way to attack a well-designed symmetric cipher is to try each key until we find one that works. In this case, the length of the key directly controls how strong the cipher is. A key length of 40 bits allows 2^{40} different keys and therefore requires 2^{40} cryptographic operations to search through all possible keys. A 56-bit key requires 2^{56} operations, and so on.

The most important thing to realize here is that the strength of a cipher increases exponentially with key length, not linearly. A 56-bit cipher is roughly 65000 (2^{16}) times stronger than a 40-bit cipher: it takes 2^{16} times more resources to attack. So, if a 40-bit cipher can be cracked with X machines in time Y, and a 56 bit cipher can be cracked with aX machines in bY time, then the ab product must equal 2^{16}.

In 1996, an expert panel of cryptographers known as the "Group of Seven" (Matt Blaze, Whitfield Diffie, Ron Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Weiner) concluded that 90 bits should provide adequate security for 20 years [Blaze1996]. For information with a shorter lifetime, 75 or 80 bits should be enough. Anything shorter than 70 bits is questionable and anything shorter than 56 bits is known to be vulnerable to commercial attackers. Several widely used ciphers have 128-bit keys, which are certainly plenty long, and the new Advanced Encryption Standard will have a 128-bit key.

It's very important to remember that key length is not the only determinant of cipher strength. Many ciphers with long keys have been broken by new analytic techniques. For instance, LUCIFER, the predecessor to DES, had a 128-bit key but was vulnerable to differential cryptanalysis. Nevertheless, if you're using a cipher with too short a key, then you're definitely in trouble.

Asymmetric Algorithms

Key lengths for asymmetric ciphers are not directly comparable to key lengths for symmetric ciphers. Because of the math required to make asymmetric ciphers work at all, the keys tend to be quite long. For ciphers such as RSA and Diffie-Hellman, key lengths will typically be between 512 and 2048 bits. This definitely does not mean that 512-bit RSA is stronger than 128-bit RC4. On the contrary, it is likely far weaker.

Unfortunately, we're not sure how much weaker. With symmetric ciphers, it's fairly clear what a brute force attack looks like: You search every key until you find the right one. With asymmetric ciphers, this isn't how it's done. With RSA, for instance, the best known attack is to factor the RSA modulus. The development of a new factoring algorithm or just an improvement on the current algorithms automatically makes all existing keys weaker.

That said, we've got some rough idea of RSA's strength. RSADSI has a factoring challenge where they publish a large number of "challenge numbers" and award cash prizes for the factorization. To date, the largest RSA modulus number that is publicly known to have been factored was RSA-155, a 155-digit (512 bit) number. At Eurocrypt 1999, Adi Shamir described a design for an RSA key breaker called "Twinkle" capable of factoring 512-bit numbers [Shamir1999]. These results show that 512 bit RSA keys are at the outer edge of breakability using current technology. In general, 768 or 1024 bits are the minimal acceptable key lengths. DH and DSA keys are about as strong as RSA keys of equivalent length. Current information suggests that 1024-bit asymmetric keys are about as strong as 80-bit symmetric keys.

1.18 Summary

This chapter has provided a basic introduction to communications security. At least some comprehension of this material is required to understand the rest of this book.

Communications security has three primary objectives: confidentiality, message integrity, and endpoint authentication. Confidentiality means that the data you're sending is kept secret. Message integrity means that you can detect when a message was tampered with in transit. Endpoint authentication means that you can be confident you are speaking to the right person.

The network cannot be trusted. A basic assumption of Internet security is that the attacker may have control of the network. In order to retain security in such an environment, we use cryptographic techniques.

We deal in four fundamental types of algorithms: symmetric ciphers, message digests, public key ciphers, and digital signatures. All of our more complicated constructs are built from these four primitives. There are many different algorithms of each type.

Public key systems are usually used in tandem with symmetric systems. Public key cryptography provides capabilities that cannot be attained with symmetric cryptography but it is too inefficient to be used alone.

For noninteractive (messaging) applications, we generate self-contained messages. We use public key cryptography to wrap the symmetric keys we use to encrypt the message. We use message digests in concert with digital signature functions to provide message integrity.

For interactive applications, we establish keys with a handshake and then use symmetric algorithms. We use public key cryptography in a handshake to authenticate both sides and exchange keys. We then use those keys to protect individual data records using symmetric techniques.

2

Introduction to SSL

2.1 Introduction

This chapter provides an introduction to SSL and TLS as well as background for the rest of the book. We first provide a general overview of how SSL works and then we describe the history of SSL and its variants, culminating in TLS. We also cover the implementation and deployment status of the various versions.

We then provide a brief overview of the use of SSL with the *HyperText Transfer Protocol* (HTTP), the native protocol of the World Wide Web and the protocol to which SSL is best matched. Finally, we describe the movement to deploy other non-Web protocols over SSL.

2.2 Standards and Standards Bodies

Most of the standards we'll be discussing in this book were produced by the *Internet Engineering Task Force* (IETF). IETF documents are easily recognizable because they have names of the form RFC #### (RFC 2246 is the TLS document). Note that not all IETF RFCs are standards. Some are merely informational documents which are published in archival form. This allows vendors to document their protocols in a stable and public way.

IETF Standards Track documents actually go through three stages, *Proposed Standard*, *Draft Standard*, and *Standard*, and a document can go through changes at any point in the cycle. However, the process of progressing a document has started to take so long that anything at Proposed level or above is generally treated as a standard by implementors, and we'll mostly do the same here.

IETF documents have two useful features that documents produced by other standards bodies don't have. First, they're typically written either by implementors or by their close colleagues, so they usually address immediately relevant problems. Second, IETF documents are free and can be downloaded from Web and FTP sites all over the world. The main site is at `http://www.ietf.org/`.

Ex.1024
Askeladden, L.L.C. / Page 113 of 788

The two other sorts of standards that we'll be dealing with are those produced by the *American National Standards Institute* (ANSI) and those produced by the *International Telecommunications Union* (ITU). ANSI standards typically have names like X9.42 (the standard for Diffie-Hellman key exchange). ITU standards have names that look like X.509 (the standard for certificates). Being able to identify which is which isn't that important unless you want to buy one of the documents; neither organization produces free documents.

Soapbox: It's difficult to overstate the importance of having free documents. It's quite common for implementors of IETF protocols to be unwilling or unable to pay for the non-free protocol documents that the IETF protocols depend on. As a consequence, bootleg copies and summaries of the non-free standards are passed around from hand to hand, with the result that implementations of these (often very complex) standards tend to be non-standard.

For this and other cultural reasons, IETFers tend to look down on ANSI and the ITU (especially the ITU), but a number of their documents cover important space and have been incorporated by reference into IETF documents, so it's important to be aware of them.

2.3 SSL Overview

Secure Sockets Layer (SSL) is a protocol that provides a secure channel between two machines. It has facilities for protecting data in transit and identifying the machine with which you are communicating. The secure channel is *transparent*, which means that it passes the data through unchanged. The data is encrypted between client and server, but the data that one end writes is exactly what the other end reads. Transparency allows nearly any protocol that can be run over TCP to be run over SSL with only minimal modification. This is very convenient.

SSL has gone through a number of revisions, beginning with version 1 and culminating in its adoption by the IETF as the *Transport Layer Security* (TLS) standard. All the pre-IETF versions of SSL were designed by engineers at Netscape Communications. Version 1 was never widely deployed, and so we'll begin our discussion with version 2 (SSLv2). Appendix B contains a brief discussion of version 1.

2.4 SSL/TLS Design Goals

SSLv2 Design Goals

SSL was originally designed for the World Wide Web environment. Netscape's intent was to provide a single solution for all their communications security problems, including Web, mail, and news traffic. However, their immediate problem was Web traffic, and SSL was designed to fit those needs. This meant that SSL had to work well with the

main protocol used by the Web, *Hypertext Transfer Protocol* (HTTP) [Fielding1999].

When SSLv2 was first designed in 1994, the Web security problem that people were most worried about was how to pass information from the client to the server without disclosing it to attacking third parties. The paradigmatic usage scenario is providing a credit card number for online shopping. Consequently, the first major design goal was to provide *confidentiality* for traffic between client and server.

The credit card case also motivates the second and third design goals. Because credit card numbers are inherently valuable, it's important that they be revealed only to someone with whom you intend to do business—and it had better be the right person. This requires that the customer (the Web browser) be sure that he's sending his credit card to the correct merchant (the Web server). On the other hand, there was no need for the server to know who the user is because the credit card number is all the identity information required to bill him. Thus, the second major design goal: *server authentication*. Optionally, SSL also provided *client authentication*.

Because customers will often want to do business with merchants with whom they have never done business before, it was very important that the protocol be able to handle this situation automatically, with a minimum of hassle to the user. This design goal is best referred to as *spontaneity*.

Because Netscape wanted to use SSL as a unified security solution, it also had to work correctly for other protocols besides HTTP. The designers observed that most popular Internet protocols ran over TCP connections, so a protocol that provided a secure, transparent channel could be used to provide security for all such protocols. This property, *transparency*, is probably the single largest reason for SSL's success, because it made it relatively simple to provide some security for almost any protocol merely by running it over SSL. Indeed, immediately after introducing SSL, Netscape started using it to protect not only HTTP traffic but also Usenet news traffic over *Network News Transfer Protocol* (NNTP).

Unfortunately, not all protocols require the same security properties, and so the results of "just" running a protocol over SSL are often undesirable, either from a security or a performance perspective. Much of the purpose of this book is to give you enough information to design faster and more secure systems than a naive use of SSL would allow.

SSLv3 Design Goals

The wide popularity of SSLv2 more or less assured that all of the original SSLv2 design goals would be retained in SSLv3 [Freier1996]. The primary job of the SSLv3 designers was to fix a number of security problems in SSLv2 (see Appendix B for more on this). This meant designing a system that was both stronger and easier to analyze. In particular, the SSLv3 designers explicitly wanted to design a mechanism for securely negotiating multiple cryptographic algorithms. The result is that SSLv3 supported a far greater number of algorithms than SSLv2. In the end, these desires led to SSLv3 being completely different from SSLv2, retaining only the basic protocol flavor.

2.5 SSL and the TCP/IP Suite

All versions of SSL and TLS share the same basic approach: they provide a secure channel between two communicating programs over which arbitrary application data can be sent. In theory, an SSL connection acts very like a "secured" TCP connection. As the name *Secure Sockets Layer* makes clear, SSL connections are intended to act just like sockets connected by TCP connections.

The primary purpose of having SSL semantics mimic TCP semantics is to make the application programmer's life easier. In recognition of this goal, most SSL implementations provide an *application programming interface* (API) consciously modeled after the most popular networking API, *Berkeley sockets*. Figure 2.1 provides an example of some calls from a typical SSL API (OpenSSL) and the corresponding Unix API calls.

Sockets API	OpenSSL
`int socket(int, int, int)`	`SSL *SSL_new(SSL_CTX *)`
`int connect(int, const struct sockaddr *, int)`	`int SSL_connect(SSL *)`
`ssize_t write(int, const void *, size_t)`	`int SSL_write(SSL *, char *, int)`
`ssize_t read(int, void *, size_t)`	`int SSL_read(SSL *, char *, int)`

Figure 2.1 Sockets API compared to SSL API

From the application programmer's perspective, in a perfect world, he would be able to replace all sockets calls with SSL calls and get security more or less automatically. It would even be possible to secure an application merely by relinking with a library that provided secure (but otherwise indistinguishable) versions of the ordinary sockets call. Unfortunately, although this is possible to some extent, SSL semantics do not *precisely* match TCP semantics, which can lead to problems and confusion, as we'll see in Chapter 8. Figure 2.2 shows SSL's place in the protocol stack, just below the application and just above TCP.

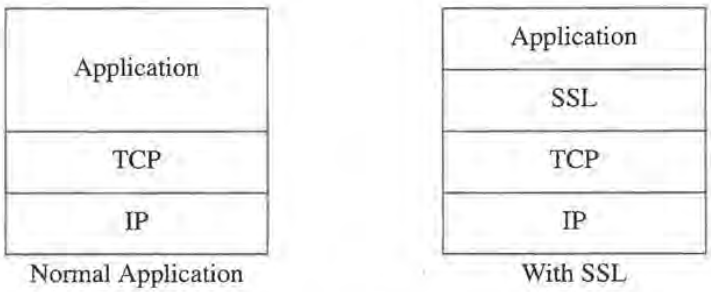


Figure 2.2 SSL's place in the protocol stack

SSL assumes that the underlying packet delivery mechanism is *reliable*: data written to the network will be delivered to the program on the other end in order, with no data being dropped or delivered twice. In theory, there are any number of transport protocols that could provide this service, but in practice SSL nearly always runs on top of TCP. SSL cannot run over UDP or directly over IP.

SSL's dependence on a reliable transport protocol to deliver its data has been a point of contention in the past and at least two attempts have been made to remove the dependency. In general, what this involves is arranging for acknowledgments and retry timeouts so that lost messages can be resent. Microsoft's STLP and the Wireless Applications Forum's WTLS [WAP1999a] are both SSL variants that are intended to work correctly over a datagram transport such as UDP.

2.6 SSL History

Figure 2.3 shows the family tree of various SSL variants, with the earliest (SSLv2) at the top and the latest (WTLS) at the bottom The original SSLv2 specification was first publicly released in November 1994, and deployed in Netscape Navigator 1.1 shortly thereafter in March 1995. The primary design was done by Kipp Hickman, a Netscape employee, with a minimum of public comment.

> The original Netscape Navigator SSLv2 implementation was badly broken. It seeded its pseudo-random number generator from the time of day and a few other sources. Wagner and Goldberg [Goldberg1996] showed how to break an SSL connection from Navigator 1.1 in under an hour.

Because SSL had been developed with a minimum of vendor input, as other vendors started implementing SSL, a number of somewhat incompatible independent variants were developed to fix problems with the protocol. The most important of these was Microsoft's *Private Communications Technology* (PCT) [Benaloh1995], published in October of 1995.

PCT

PCT's authors (Josh Benaloh, Butler Lampson, Daniel Simon, Terence Spies, and Bennet Yee) had significant cryptographic experience and they applied it to the protocol. In general, PCT had better-thought-out security properties than SSL, though it was backward compatible with SSL.

Although SSLv2 had version numbers, Netscape had control of the version number space, and behavior of a server upon seeing a newer version number than it supported was never well specified. As a consequence, an implementation indicated that it spoke PCT by sending a special PCT_SSL_COMPAT cipher in its cipher list. SSLv3's semantics for version upgrading are better defined; however, there is still no way to indicate that one supported specific new features, meaning that there was no good way to

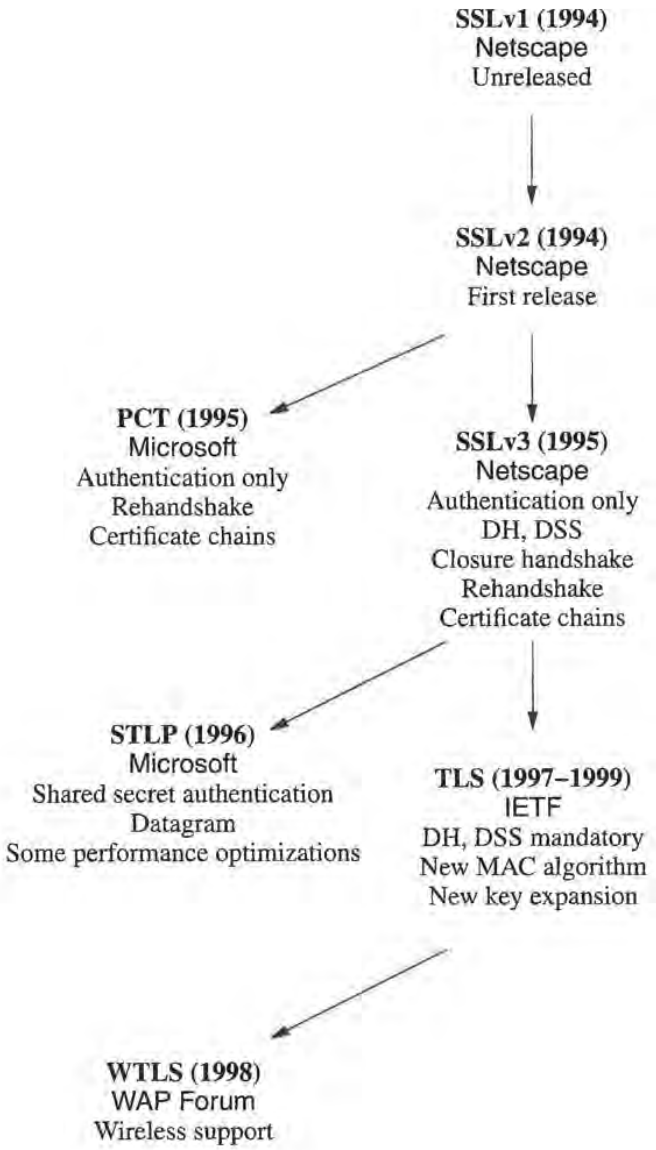


Figure 2.3 Family tree of SSL variants

add experimental features. As a result, this hack is occasionally used by TLS and SSLv3 implementations as well.

Three significant changes were made. First, PCT included a non-encrypted operational mode, providing only data authentication. Second, it tightened up the key expansion transform. Due to United States export restrictions, SSLv2 had an *export* mode in which the encryption security was limited to 40 bits. Unfortunately, SSLv2 used the same key for encryption and authentication, and thus authentication security was also limited to 40 bits. PCT allowed weak encryption to coexist with strong authentication. Third, PCT improved performance by decreasing the number of required round trips. For more on SSLv2 and PCT, see Appendix B.

SSLv3

Although an abortive first draft of "SSLv2.1" as a modification of SSLv2 circulated around Netscape, Netscape eventually decided to burn everything and start over again. They hired Paul Kocher, a noted security consultant, to work with Allan Freier and Phil Karlton to develop a new version of SSL, to be called SSL version 3 (SSLv3). SSLv3 was released in late 1995.

PCT had preserved the general style of SSLv2 to the extent that it used the same specification language and some of the same messages, but SSLv3 invented a completely new specification language and an all-new record type and data encoding. It also addressed some of the same issues as PCT, including adding an authentication-only mode and completely redoing the key expansion transform.

SSLv3 also added a number of new features not in PCT, including a number of new ciphers (*Digital Signature Standard* (DSS), *Diffie-Hellman* (DH), and the National Security Agency's *FORTEZZA* cryptographic token), and support for a *closure* handshake to prevent *truncation attacks* on the data stream. With SSLv2, it was possible for an attacker to forge a TCP connection closure and make it appear that less data was transmitted than in fact was. SSLv3's closure handshake allows this attack to be detected. As with PCT, SSLv3 was backward compatible with SSLv2.

> Backward compatibility in SSL has traditionally been fairly limited. The sense in which it's usually meant is that an SSL client/server pair can safely interact even if they are different versions, provided that they share a common version. It does not mean that valid messages in a previous version are always valid in a newer version, and in fact that has traditionally not been the case.

TLS

In May of 1996 the *Internet Engineering Task Force* (IETF) chartered the *Transport Layer Security* (TLS) working group to try to standardize an SSL-like protocol. Although the official working group charter said otherwise, it was widely understood that this would be an attempt to harmonize the Microsoft and Netscape approaches, and although a number of other approaches were proposed, they received essentially no support. The intention was to be finished by the end of 1996.

Microsoft produced a submission called *Secure Transport Layer Protocol* which was a modification of SSLv3, adding a number of features that Microsoft considered critical. Primary among them were support for datagrams (e.g., UDP) and support for client authentication using shared secrets. Although it was always possible to send the shared secret over an encrypted SSL connection, this meant that in the exportable case, the secret was encrypted with only a 40-bit key. STLP incorporated a mode that allowed a much stronger shared-secret-based authentication for clients. STLP also incorporated some modest performance improvements and improved cipher extensibility as well as allowing the TCP "client" to be the STLP "server."

In late 1996 the major players (and a number of minor ones) met in Palo Alto at a rump group meeting chaired by cryptographer Bruce Schneier. Typically, IETF working groups meet face to face only three times a year at the IETF meetings, but sometimes they'll meet informally when the job is particularly urgent. This was such a meeting.

It rapidly became clear that there was very little support for changing any of SSLv3, except for a few obvious (and minor) bugs. In particular, there was a lot of resistance—largely in the name of backward compatibility—to the sort of changes Microsoft had proposed. The primary open issue seemed to be what the name of the new protocol could be, since using either SSL or PCT seemed too much like saying that Microsoft or Netscape had won. Whatever the name, however, the new protocol would just be a minor cleanup of SSLv3.

Eventually, after several more working group meetings, the TLS working group decided to name the protocol TLS, which all parties were equally unhappy with. A number of minor changes were made to the document, in the name of security, with the effect that the key expansion and the message authentication computations are totally incompatible with SSLv3, destroying most backward compatibility.

By far the most contentious change made in TLS was the decision to require implementations to support DH, DSS, and *Triple-DES* (3DES). This was problematic for two reasons. First, Netscape had only implemented RSA for authentication and key exchange. Because Netscape constituted the vast majority of browsers in the field, interoperability with Netscape's SSL implementation was the de facto minimal standard. Because interoperation with older clients and servers was required by the market, this meant that implementors would have to implement both RSA and DH/DSS.

Far more inconvenient at the time, however, was the requirement to add 3DES. At the time, United States export regulations forbade general export of cryptography stronger than 40 bits. Thus, it was illegal to export a TLS implementation containing 3DES. The result was that an implementation could either be TLS compliant or U.S. exportable, but not both.

To understand why these controversial changes were made, you have to understand how the IETF works. The IETF is organized into a number of working groups working on specific tasks (e.g., to develop TLS). Each working group is part of one or more general *areas*. Currently, there are eight areas: Applications, General, Internet, Operations and Management, Routing, Security, Transport, and General Services. Each area has one or more *area directors*, who oversee work in that area. The area directors collectively form the *Internet Engineering Steering Group* (IESG), which must approve any document before it becomes an Internet archival document, a *Request For Comment* (RFC).

Ex.1024
Askeladden, L.L.C. / Page 120 of 788

At the April 1995 IETF meeting in Danvers, Massachusetts, the IESG adopted what came to be known as the *Danvers Doctrine*, which stated that the IETF would design protocols that embodied good engineering principles, regardless of exportability. At the time, this doctrine implied support for DES at a minimum and over time it came to mean 3DES. Moreover, the IETF had a longstanding preference for unencumbered (i.e., free) algorithms when possible. When the Merkle-Hellman patent (covering all of public key cryptography) expired in 1998, but RSA was still patented, the IESG began pressuring working groups to adopt free public key algorithms. Finally, many IETF members felt it was good practice for protocols to have a mandatory set of options to ensure that any two implementations could talk to each other.

When the TLS working group had finished its work, in late 1997, it sent the document off to the IESG, and the IESG sent it back with instructions to add a mandatory cipher—DSS for authentication, DH for key agreement, and 3DES for encryption, thus solving the above three issues. Much discussion on the mailing list ensued, with Netscape in particular resisting mandatory algorithms in general and 3DES in particular. After a prolonged standoff between the IESG and the working group, grudging consensus was reached and the document was sent back with the appropriate changes.

Unfortunately, in the meantime, another obstacle appeared. The IETF *Public Key Infrastructure* (PKIX) working group, which was standardizing a profile for X.509 certificates, was winding up its work. TLS depended on certificates and hence on PKIX, and IETF rules forbid protocols advancing ahead of other protocols on which they depend. PKIX finalization took rather longer than expected and added an additional delay. Finally, in January of 1999, over two years late, TLS was published as RFC 2246 [Dierks1999]. As of this writing, deployment has been less than spectacular. Internet Explorer supports TLS but Netscape Navigator does not. Most toolkits do support TLS.

2.7 SSL for the Web

The primary use of SSL is to protect Web traffic using HTTP. We discuss this topic extensively in Chapter 9. Here we do so to provide a brief overview of a typical application in order to explain some of the choices that were made in SSL design.

The process is simple. In HTTP, a TCP connection is created and the client sends a request. The server responds with a document. When SSL is used, the client creates a TCP connection, establishes an SSL channel on top, and then sends the same HTTP request over the SSL channel. The server responds in a similar fashion over the SSL connection.

Because the SSL handshake looks like garbage to an ordinary HTTP server, in order for the process to work correctly the client needs some way to know that the server is prepared to accept SSL connections, as not all servers do so. A Web address (technically known as a *Uniform Resource Locator* (URL)) beginning with `https` rather than `http` is used to indicate that SSL should be used, for example

```
https://secure.example.com
```

As a consequence, the combination of HTTP running over SSL is often referred to as *HTTPS*.

If all goes well and the server and client can communicate, the user is typically rewarded with some sort of user interface indication that security is in use. In older versions of Netscape, a blue bar was displayed below the toolbar and a key appeared in the lower left-hand corner of the browser. In newer versions of Netscape, the open lock in the lower left-hand corner closes. In Internet Explorer, a lock appears in the lower right-hand corner of the screen. These are all indications that the current page was fetched using SSL.

Although nearly all modern commercial Web servers implement SSL, initially Netscape charged a significantly different price for their SSL-enabled server (*Netscape Commerce Server*) than for their ordinary server. Moreover, due to patent reasons, none of the free servers in the United States support SSL with RSA. (And since RSA is the de facto standard, this might as well mean not at all.) Even if you have a server that implements SSL, you might choose not to enable security. HTTPS puts a much higher load on the server machine than does pure HTTP, and obtaining the required certificate is fairly inconvenient. Moreover, certificates aren't cheap. Verisign, the biggest CA as of this writing, charges over $300 U.S. for a single certificate.

In January 2000, Netcraft's survey [Netcraft2000] found about 1.5 million servers running SSL, but only about 60,000 of them were certified by a well-known third party. The rest were self-signed or issued by a private CA. A client would require special configuration to connect to these servers.

2.8 Everything over SSL

Because so many protocols run over TCP, and SSL connections act so much like TCP connections, securing a preexisting protocol by layering it over SSL is a very attractive design decision. In addition to HTTP over SSL and NNTP over SSL (often referred to as *SNEWS*), proposals soon appeared for using SSL to secure all the major Internet protocols, including SMTP [Hoffman1999a], Telnet [Boe1999], and FTP [Ford-Hutchinson2000]. Many vendors also use SSL to secure their proprietary protocols.

In order to accommodate connections from clients who do not use SSL, servers must typically be prepared to accept both secured and non-secured versions of the application protocol. All of the protocols mentioned in the previous paragraph use one of two basic strategies: *separate ports* and *upward negotiation*. Both of these strategies have advantages and disadvantages, which we'll discuss in Chapter 7.

In a separate-ports strategy, the protocol designer simply assigns a different *well-known port* to the protocol and the server implementor has the server listen both on the original port and on the new secure port. Any connections that arrive on the secure port are automatically SSL negotiated before application protocol traffic begins. HTTPS uses this strategy, as we'll discuss in Chapter 9. Figure 2.4 shows the port assignments for some commonly used protocols.

Name	Port	Use
ftps-data	989/tcp	ftp protocol, data, over TLS/SSL
ftps-data	989/udp	ftp protocol, data, over TLS/SSL
ftps	990/tcp	ftp protocol, control, over TLS/SSL
ftps	990/udp	ftp protocol, control, over TLS/SSL
nntps	563/tcp	nntp protocol over TLS/SSL
nntps	563/udp	nntp protocol over TLS/SSL

Figure 2.4 Port number assignments for SSL-ized protocols

When using an upward negotiation strategy, the protocol designer modifies the application protocol to support a message indicating that one side would like to upgrade to SSL. If the other side agrees, an SSL handshake starts. Once the handshake is completed, application messages resume over the new SSL channel. SMTP over TLS uses this strategy, as we'll discuss in Chapter 10.

2.9 Getting SSL

By far the easiest way to get an SSL implementation is to download it. The OpenSSL Project at `http://www.openssl.org/` provides a high-quality, free, source implementation of SSLv2, SSLv3, and TLS available for worldwide downloading. OpenSSL is based on Eric Young's SSLeay library, which was first released in 1995. The code is written in more or less ANSI C and has been widely ported. OpenSSL is licensed under a BSD-style license and is therefore free for commercial and noncommercial use. Until very recently, due to patent reasons, OpenSSL could not be used inside the United States in RSA mode. However, since RSA's patent expired in September 2000, it is now legal in the U.S. as well.

The following vendors sell C/C++ SSL/TLS toolkits.

- Certicom (`http://www.certicom.com/`)
- Netscape Communications (`http://home.netscape.com/`)
- RSA Security (`http://www.rsasecurity.com/`)
- SPYRUS/Terisa Systems (`http://www.spyrus.com/`)

Netscape had the first SSL implementation. RSA has hired Eric Young and their toolkit is based on SSLeay. Consensus and SPYRUS/Terisa's implementations were both independently developed.

Full disclosure: The author worked at Terisa for three years and was the primary author of Terisa's SSL product.

Similarly, the easiest way to get a Java implementation is to download it. The author's free Java SSLv3/TLS implementation, PureTLS, is available at `http://www.rtfm.com/puretls`. The following vendors also have Java implementations.

- Baltimore (`http://www.baltimore.com/`)
- Certicom (`http://www.certicom.com/`)
- Phaos Technology Corporation (`http://www.phaos.com/`)
- Sun (`http://www.javasoft.com/`)

As of this writing, Sun's is royalty-free for "non-commercial" use but is available in binary form only.

Both Netscape and Internet Explorer implement SSL as do Netscape's Web servers and Microsoft's IIS. Microsoft also allows programmers to call their SSL implementation (SChannel), but that's only an option for Windows programmers. Several free source implementations of SSL (based on OpenSSL) for the free HTTP server Apache are available.

- ApacheSSL (`http://www.apachessl.com/`)
- mod_ssl (`http://www.modssl.org/`)

A number of commercial Apache-based SSL implementations are available.

- Raven (`http://www.covalent.net/`)
- Stronghold (`http://www.c2.net/`)

What's Implemented

Nearly every currently available SSL implementation supports SSLv3 with RSA. A notable exception is some installed Web servers (typically old versions of Netscape Commerce Server) that support only SSLv2. Most implementations support SSLv2 for backward compatibility, though some newer implementations omit SSLv2 support. Most SSL toolkits support the TLS protocol changes, but support for DSS and DH isn't always available. Microsoft's implementations also support PCT and TLS, and the versions of IIS and IE shipping with Windows 2000 also support DSS and DH. As of this writing, Netscape's products still support only SSLv3 and do not support DSS and DH.

For maximal compatibility, it's important to support SSLv2 and SSLv3. There are essentially no implementations that support only TLS, so support for TLS is not necessary, although it is desirable because it's an IETF standard.

2.10 Summary

This chapter has provided an overview of SSL and its place in the networking universe, providing the context into which the material we'll be discussing for the rest of the book fits.

SSL and TLS provide a generic channel security mechanism on top of TCP. Any protocol that can be carried over TCP can be secured using SSL or TLS.

SSL was originally designed by Netscape. SSLv2 and SSLv3 were basically Netscape-only productions, but TLS is the product of an IETF working group starting with SSLv3.

SSL provides server authentication, encryption, and message integrity. Optionally, there is support for cryptographic client authentication.

SSL APIs are typically designed to mimic normal networking APIs. In C, SSL APIs typically emulate Berkeley sockets, and in Java they emulate the Java socket abstraction. This makes it very easy to convert an insecure application to SSL.

SSL is very widely used for HTTP traffic. However, more and more protocols are now being secured with SSL/TLS. TLS is slowly replacing SSL.

A large number of SSL and TLS implementations are available. There are free implementations in both C and Java and a number of commercial implementations are available.

3

Basic SSL

3.1 Introduction

This chapter provides a detailed account of SSL operation for the most common case, server-only authentication using RSA. By titling this chapter Basic SSL, we mean that this is the basic mode of SSL, not that we're not going to go into depth about the protocol (we are). This chapter covers only the simplest mode of SSL. The next chapter will discuss several more-complicated modes.

This chapter is organized into two primary sections. The first is an overview of how SSL works and how the various pieces fit together. We'll broadly describe each protocol message and its general purpose. The second section dissects a sample SSL connection in detail. We use an automated tool to generate a session trace and then examine each protocol message in detail, seeing how the individual fields fit together to accomplish the job of the protocol. If you're reading this as an implementation guide or want a really detailed understanding of what's happening, you should read both sections. If your primary interest is simply in using SSL, you can probably safely skip the second half.

3.2 SSL Overview

The basic design of SSL should look fairly familiar from the Toy Security Protocol we designed in Chapter 1. A connection is divided into two phases, the *handshake* and *data transfer* phases. The handshake phase authenticates the server and establishes the cryptographic keys which are used to protect the data to be transmitted. The handshake must be completed before any application data can be transmitted. Once this has been done, the data is broken up and transmitted as a series of protected records.

Ex.1024
Askeladden, L.L.C. / Page 127 of 788

3.3 Handshake

The purpose of the SSL handshake is threefold. First, the client and the server need to agree on a set of algorithms which will be used to protect the data. Second, they need to establish a set of cryptographic keys which will be used by those algorithms. Third, the handshake may optionally authenticate the client, as we'll see in Chapter 4. First, we'll discuss the handshake process in general terms without discussing specific protocol messages and then we'll show how it breaks down into messages.

The overall process works something like this (see Figure 3.1 for a pictorial representation).

1. The client sends the server a list of the algorithms it's willing to support, along with a random number used as input to the key generation process.
2. The server chooses a cipher out of that list and sends it back along with a certificate containing the server's public key. The certificate also provides the server's identity for authentication purposes and the server supplies a random number which is used as part of the key generation process.
3. The client verifies the server's certificate and extracts the server's public key. The client then generates a random secret string called the pre_master_secret and encrypts it using the server's public key. It sends the encrypted public key to the server.
4. The client and server independently compute the encryption and MAC keys from the pre_master_secret and the client and server's random values.
5. The client sends a MAC of all the handshake messages to the server.
6. The server sends a MAC of all the handshake messages to the client.

So, what has this process accomplished? Remember that our two goals were, first, to agree on a set of algorithms and, second, to establish a set of cryptographic keys. Steps 1 and 2 accomplish the first goal. The client gets an opportunity to tell the server which algorithms it supports and the server chooses an algorithm. When the client receives the message the server sent in step 2, it also knows the algorithm, so both sides now know what algorithm to use.

The second goal, establishing a set of cryptographic keys, is accomplished by steps 2 and 3. In step 2 the server provides the client with its certificate, which allows the client to transmit a secret to the server. After step 3, the client and the server both share the pre_master_secret. The client has the pre_master_secret because it generated it, and the server has the pre_master_secret because it decrypted it.

Note that step 3 is the key step in the whole handshake. All the data that is going to be protected depends on the security of the pre_master_secret. What's going on is very simple: the client uses the server's public key (extracted from the certificate) to encrypt a shared key, and the server uses its private key to decrypt the shared key. The rest of the handshake is mainly devoted to ensuring that this exchange can happen safely. In step 4 the client and the server then separately use the same *key derivation function* (KDF) (see Section 3.11) to generate the master_secret. The master_secret is used to generate the cryptographic keys, again using the KDF.

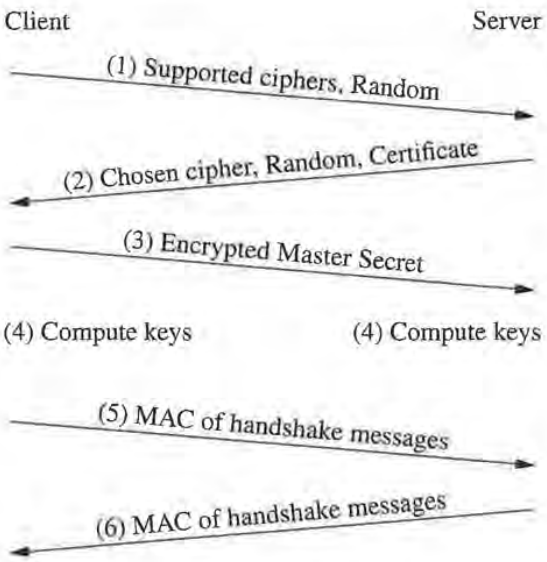


Figure 3.1 Overview of the SSL handshake

Steps 5 and 6 serve to protect the handshake itself from tampering. Imagine an attacker who wished to control the algorithms used by the client and the server. It's quite common for the client to offer a range of algorithms, some weak and some strong, so that it can communicate with servers that only support weak algorithms. The attacker could delete all the strong algorithms from the client's offer in step 1 and thus force the server to choose a weak algorithm. The MAC exchange in steps 5 and 6 stops this, because the client's MAC will be computed over the original messages and the server's MAC will be computed over the messages the attacker modified, and they won't match when checked. Because the client- and server-provided random numbers are inputs to the key generation process, the handshake is secure from replay attack. These messages are the first messages encrypted under the newly encrypted algorithms and keys.

So, at the end of this process, the client and the server have agreed on the cryptographic algorithms to use and have a set of keys to use with those algorithms. More important, they can be sure that an attacker hasn't interfered with the handshake and that therefore the negotiation reflects their true preferences.

Handshake Messages

Each step of the description we've just given is accomplished by one or more handshake messages. We'll briefly describe here which messages map to which steps and then

cover each individual message in detail later in this chapter. Figure 3.2 shows the individual messages.

- Step 1 corresponds to a single SSL handshake message, ClientHello.
- Step 2 corresponds to a series of SSL handshake messages. The first message the server sends is the ServerHello, which contains its algorithm preferences. Next it sends its certificate in the Certificate message. Finally, it sends the ServerHelloDone message, which indicates that this phase of the handshake is done. The reason that the ServerHelloDone is needed is that some of the more complicated handshake variants involve other messages being sent after the Certificate message. When the client receives the ServerHelloDone message, it knows that no such other messages will be arriving and so it can proceed with its part of the handshake.
- Step 3 corresponds to the ClientKeyExchange message.
- Steps 5 and 6 correspond to the Finished message. The Finished message is the first message that's protected using the just-negotiated algorithms. To protect the handshake from tampering, the content of the message is a MAC of *all* the previous handshake messages. However, since the Finished message is protected using the negotiated algorithms, the message itself is also MACed with the newly negotiated MAC keys.

  Note: This diagram omits two messages, the ChangeCipherSpecs. We'll cover them later, but they're not important at this stage in the discussion.

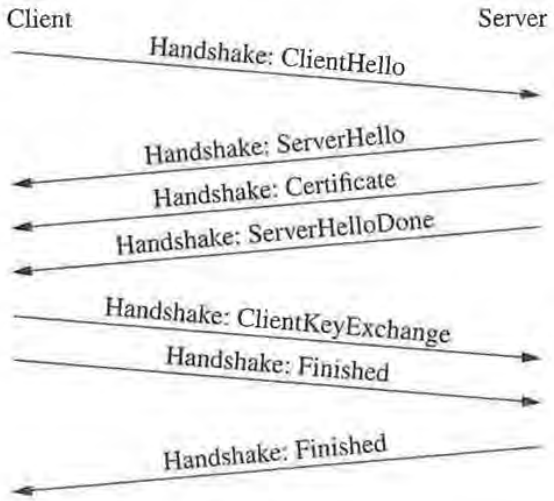


Figure 3.2 SSL handshake messages

3.4 SSL Record Protocol

All we've managed to do so far is to authenticate the server and share some keying material. Remember that the point of having SSL in the first place is to be able to exchange encrypted and authenticated *data*. The purpose of the handshake is to set up the shared state required to make sending and receiving protected data possible. In SSL the actual data transfer is accomplished using the *SSL Record Protocol*.

The SSL Record Protocol works by breaking up the data stream to be transmitted into a series of *fragments*, each of which is independently protected and transmitted. On the receiving end, each record may be independently decrypted and verified. This approach allows data to be transferred from one side of the connection to the other as soon as it is ready and processed as soon as it is received.

Before a fragment can be transmitted, it must be protected against attack. To provide integrity protection, a MAC is computed over the data. The MAC will be transmitted along with the fragment and must be verified by the receiving implementation. The MAC is then appended to the fragment and the concatenated data and MAC are encrypted to form the *encrypted payload*. Finally, a header is attached to the payload. The concatenated header and encrypted payload are referred to as a *record*. Records are what is actually transmitted. Figure 3.3 shows the transmission process.

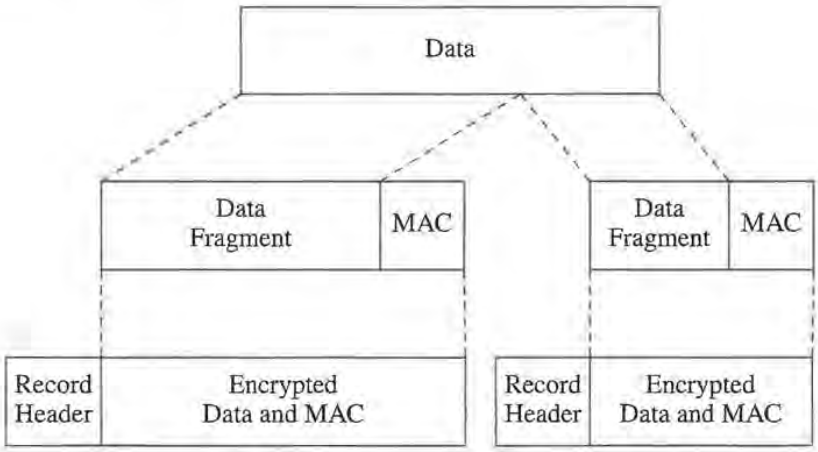


Figure 3.3 SSL data fragmentation and protection

Record Header

The job of the record header is to provide information that is necessary for the receiving implementation to interpret the record. In practice, this means three pieces of information: the *content type*, the length, and the SSL version. The length field allows the receiver to know how many more bytes to read off the wire before it can process the

message. The version number is simply a redundant check to ensure that each side agrees on the version.

The content type field identifies the type of the message. As we discussed in Chapter 1, it's convenient to be able to send other kinds of traffic over the same protected channel. In particular, we'd like to be able to send protected messages to indicate errors and connection closure termination. The content type field allows implementations to distinguish this sort of management traffic from data that is intended for the higher-level application.

Content Types

SSL supports four content types: application_data, alert, handshake, and change_cipher_spec. All data sent and received by software that uses SSL is sent as application_data. The other three content types are used for management traffic, such as performing the handshake and signaling errors.

The alert content type is primarily used for signalling various types of error. Most alerts signal things that went wrong in the handshake, but some indicate errors that occurred when trying to decrypt or authenticate records. The other use of alert messages is to signal that the connection is about to close. This is necessary to prevent the truncation attack discussed in Section 1.8.

The handshake content type is used (unsurprisingly) to carry handshake messages. Even the initial handshake messages that form the connection are carried by the record layer as records of type handshake. Because no cryptographic keys have been established, these initial messages aren't encrypted or authenticated, but otherwise the processing is the same. It is possible to initiate a new handshake over a preexisting connection (see Section 4.5) in which case the new handshake records are encrypted and authenticated just as any other data would be.

The change_cipher_spec message has a special purpose. It indicates a change in the encryption and authentication of records. Once a handshake has negotiated a new set of keys, the change_cipher_spec record is sent to indicate that those new keys will now be used. This is covered in more detail in Section 3.10.

3.5 Putting the Pieces Together

As we've seen, SSL is a *layered* protocol consisting of a record layer and a number of different message types which are carried on top of the record layer. In turn, the record layer is carried on top of some reliable transport protocol such as TCP. Figure 3.4 shows the structure of the protocol.

The rest of this chapter is a detailed look at a particular SSL connection. It covers similar territory to the first part of the chapter, but at a much deeper level of detail. The material covered here is primarily the contents of the specific handshake messages. Thus, if you're interested in implementing SSL, analyzing the protocol, or designing an SSL variant, you should read on. Otherwise, you can safely skip to Chapter 4 where we cover the rest of the handshake modes. First, we discuss the general features of the

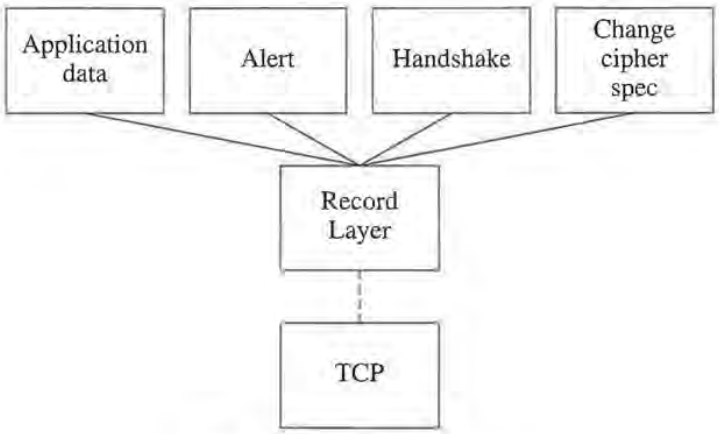


Figure 3.4 SSL protocol structure

connection and then move on to discussing the individual messages in detail.

As before, we'll cover the handshake first, describing how algorithms are negotiated by means of cipher suites and how the server's key is used to establish a shared pre_master_secret which can be used to create the cryptographic keys for the connection. Then we'll cover the cryptographic transformations that the SSL Record Protocol uses to protect the data being communicated.

3.6 A Real Connection

Figure 3.5 shows a ladder diagram for the connection we'll be studying. As with previous such diagrams, messages from the client (`speedy`) to the server (`romeo`) are shown as going from left to right and messages from server to client are shown as going from right to left.

The first message sent is the ClientHello, which contains the client's proposed cryptographic parameters, including the ciphers it's prepared to use. It also contains a random value to be used in key generation. The server responds with three messages: first, it sends the ServerHello which selects a cipher and a compression algorithm. This message contains a random value from the server.

The reason why SSL includes compression is that encrypted data is random for all practical purposes and thus incompressible. Thus, link-level compression such as is found in most modems doesn't work on SSL data. If you want to compress the data before transmission you must do so before encryption. Unfortunately, due to intellectual property considerations, neither SSLv3 nor TLS defined any compression algorithms.

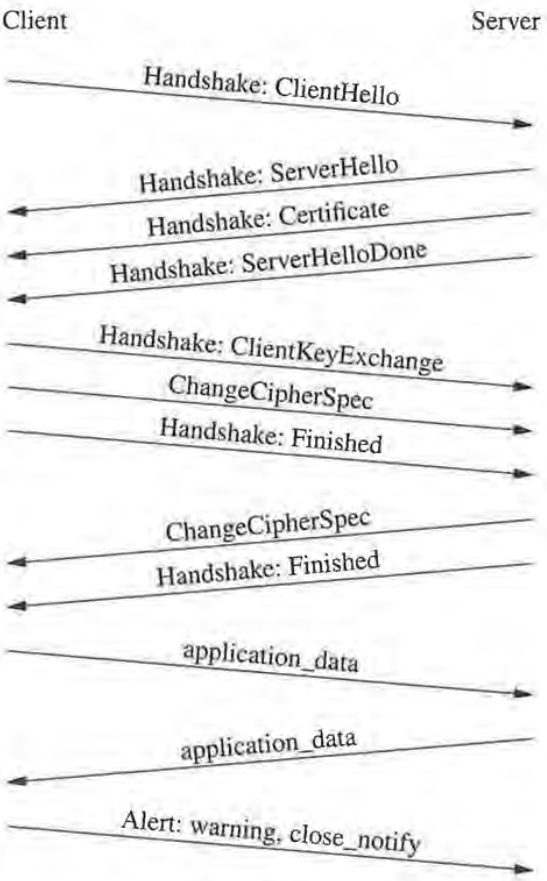


Figure 3.5 Time sequence of SSL messages

Thus, compression is almost never used with SSL. As an exception, OpenSSL does support compression, so if both sides are based on OpenSSL and suitably configured, compression can be used.

Next, the server sends the Certificate message, which contains the server's public key, in this case an RSA key. Finally, it sends the ServerHelloDone, which indicates that no more messages will be sent in this phase of the handshake.

The client now sends a ClientKeyExchange message, which contains a randomly generated key encrypted under the server's RSA key. This is followed by a ChangeCipherSpec message, which indicates that all messages that the client sends afterwards will be encrypted using the just-negotiated cipher. The Finished message contains a

check value for the entire connection so that the server can tell that the ciphers were securely negotiated.

Once the server has received the client's Finished message, it sends its own ChangeCipherSpec and Finished messages and the connection is ready to begin sending application data.

> The SSLv3 specification allows a party to start sending application data as soon as that party's Finished message is *sent*. However, this creates a security risk for client data if the connection has been attacked. If the attacker has changed the handshake to negotiate a weaker cipher suite, the client can't detect this until it receives the server's Finished message. To fix this; in TLS the client must wait until it has received the server's Finished message before it can send its first byte of data. Note that this isn't a security problem if the server chose the most secure of the client's ciphers, but rather than trying to detect this case, it's safest to simply wait for the other server's Finished message.

The next two messages are application data sent by the client and server, respectively. In this case, these were the result of typing data at the client and server programs, but in an actual transaction, this would be when the application protocol data was sent.

Finally, the client shuts down the connection, but first it sends a close_notify alert to indicate that the connection is about to close. The close_notify is followed by a TCP FIN (not shown here). The server responds with its own TCP FIN. The FINs indicate that the host's TCP implementation has closed its side of the connection and will not send any more data.

> Note that the server did not send a close_notify. The SSL specification is a little unclear on exactly whether close_notifys are required, but under perfect circumstances, both the server and client would send close_notifys.

3.7 Some More Connection Details

In order to understand the protocol messages we'll be examining, it's convenient to have a piece of software that can break apart the various messages and display their contents. We've used the `ssldump` program for this purpose.

Figure 3.6 shows an `ssldump` trace of all the messages in a simple SSL connection. We'll be talking about each message in detail in a little bit, but here we present a trace of the whole connection so you can get a feel for the flavor.

Since this is the first `ssldump` trace presented in the book, we'll take a moment to describe the output. The first line simply indicates the setup of the underlying TCP connection that the SSL session will use. In this case it's a connection between `romeo` and `speedy`, two machines on the test network. The name of the host (`romeo`) that initiated the TCP connection (i.e., the host which called `connect()`) is printed first, and the host (`speedy`) that accepted the TCP connection (i.e., the host which called `accept()`) is printed second. In normal SSL operation, the initiating host takes the client role and the accepting host takes the server role.

Each numbered line represents an SSL record. Thus, this connection consists of 12 different records. The first number is the number of the record in the sequence that was processed. This is followed by two timestamps, the first being the time since the creation of the TCP connection and the second being the time since the last record. Both times are represented in seconds and to four decimal places. Next is an indication of which direction the record was sent: C>S indicates client to server (in this case romeo to speedy) and S>C indicates server to client (speedy to romeo). Next is the record type, which corresponds to one of the layers described in the previous section.

```
New TCP connection: speedy(3266) <-> romeo(4433)
1 0.0456 (0.0456) C>S   Handshake
      ClientHello
        Version 3.1
        cipher suites
                TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
                TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
                TLS_RSA_WITH_3DES_EDE_CBC_SHA
                TLS_RSA_WITH_IDEA_CBC_SHA
                TLS_RSA_WITH_RC4_128_SHA
                TLS_RSA_WITH_RC4_128_MD5
                TLS_DHE_RSA_WITH_DES_CBC_SHA
                TLS_DHE_DSS_WITH_DES_CBC_SHA
                TLS_RSA_WITH_DES_CBC_SHA
                TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
                TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
                TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
                TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
                TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compression methods
                NULL
2 0.0461 (0.0004) S>C   Handshake
      ServerHello
        session_id[32]=
          74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
          27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
          c4 b3 a4 f6 61 ef f5 cd
        cipherSuite             TLS_RSA_WITH_DES_CBC_SHA
        compressionMethod               NULL
3 0.0461 (0.0000) S>C   Handshake
      Certificate
4 0.0461 (0.0000) S>C   Handshake
      ServerHelloDone
```

```
5 0.2766 (0.2304) C>S   Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[64]=
          17 4d 00 32 bc b2 af 95 09 0a 45 24
          97 d8 34 dc 73 20 4d 00 91 a5 0d ed
          c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
          bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3
          00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
          0b bc db a2
6 0.2766 (0.0000) C>S   ChangeCipherSpec
7 0.2766 (0.0000) C>S   Handshake
      Finished
8 0.2810 (0.0044) S>C   ChangeCipherSpec
9 0.2810 (0.0000) S>C   Handshake
      Finished
10 1.0560 (0.7749) C>S   application_data
11 6.3681 (5.3121) S>C   application_data
12 7.3495 (0.9813) C>S   Alert
        level       warning
        value       close_notify
Client FIN
Server FIN
```

Figure 3.6 A simple SSL connection

For some record types, `ssldump` performs further decoding, and this is shown on subsequent indented lines. In particular, if the record type is handshake, `ssldump` does a fair amount of decoding. The next line is the handshake type, and subsequent lines contain decodings of some relevant fields. Exactly how much information is conveyed is controlled by command line flags. This particular trace was generated with settings that show only a moderate amount of detail. The intent in this session is to show message flow, so some fields are elided from the trace. Later in the chapter we discuss each message in detail and show all fields in each message.

For example, the first record is a handshake message. The next line shows that it's a ClientHello, sent from `romeo` to `speedy`. It offers a number of cipher suites, ranging from ephemeral Diffie-Hellman with 3DES to RSA with RC4-40, but only one compression method, NULL.

Because `ssldump` was written partially in order to generate traces for this book, the traces are more or less direct program output, with the occasional addition of comments in *italics*. The message trace displayed here was generated by using OpenSSL's `s_client` and `s_server` programs, which are simple command-line client and server programs. Encrypted data is rendered in *`fixed-width italic`*.

Ex.1024
Askeladden, L.L.C. / Page 137 of 788

3.8 SSL Specification Language

It's fairly common practice for protocols to be described in terms of a specification language which describes the various fields of the protocol messages and how they appear on the wire. This is intended to provide an unambiguous and concise description of the protocol. One example of such a language is ASN.1.

Both TLS and SSLv3 use the same specification language to describe their messages. Although the syntax is vaguely reminiscent of C type definitions, the language was invented for SSLv3 and the only definition is in the specifications.

> The TLS specification describes the syntax as "somewhat casually defined," and although most of the specification is obvious, new constructs are introduced several times without much introduction. At one point, the author attempted to write a grammar-driven parser (using YACC) for the language, with the aim of mechanically generating a decoder, but abandoned the project in frustration.

Basic types

The specification uses five basic types, opaque, uint8, uint16, uint24, and uint32. opaque is simply a single uninterpreted byte. uint8, uint16, uint24, and uint32 are unsigned 8-, 16-, 24-, and 32-bit integers respectively and are represented on the wire by sequences of 1, 2, 3, or 4 bytes. All numbers are represented in "network byte order"—high byte first—so the number 1 is represented as a uint32 by the hex bytes 00 00 00 01. The constructs /* and */ bracket comments just as in C.

Vectors

A vector is simply a sequence of elements of a given type. Vectors come in two flavors, fixed-length or variable-length. Fixed-length vectors are represented using [] and variable-length vectors are represented using <>. It is important to note that all lengths are in bytes, not in number of elements. As a consequence, the declaration uint16 foo[4] refers to two 16-bit integers, not four. This allows the decoder to be layered; it can treat structures as opaque strings (since it knows their length) and pass them on to another layer for parsing.

Variable-length vectors can be represented either by specifying both an upper and lower length bound or just an upper length bound. For example:

```
opaque stuff<1..20>  /* A string of between 1 and 20 bytes */
uint32 numbers<16>   /* Up to 4 32 bit integers */
```

Fixed-length vectors are encoded on the wire simply by concatenating their elements end-to-end with the first element coming first. Variable-length vectors are encoded in the same way, except that a length field is prepended. A length field is an integer type just large enough to encode the upper bound length of the vector. For instance, opaque vec<300> would require a uint16 length field.

Enumerateds

An enumerated type is simply a field that can assume only a specific series of values. Each value is named. For example:

```
enum { red(1), blue(2), green(3) } colors;
```

specifies a type called colors which can take on the values red, blue, and green, which are represented on the wire by the integers 1, 2, and 3. When an enum is encoded on the wire, it is represented by an integral type just large enough to hold its largest value. So, colors would be represented by a uint8, one byte.

It is possible to specify an explicit maximum size for an enum, by including an unnamed value:

```
enum { warning(1), fatal(2), (255) } AlertLevel;
```

As a special case, it is possible to define an enum with no values at all. This is used occasionally in the specification to refer to state values that are internal to an implementation. Obviously, such an enum can never be encoded on the wire, because there are no definitions for how to encode the values.

```
enum { high, low } security;
```

Structures

Constructed types may be made by using the struct definition, which is very similar to C `struct` syntax:

```
struct {
  type1 field1,
  type2 field2,
  type3 field3,
  ...
} name;
```

structs are encoded on the wire simply by concatenating the encodings of the various fields, in the order that they appear in the struct definition. struct definitions can also be nested. The name field of a struct may be omitted if it is directly included in another struct.

Variants

Structures may be defined with variant contents which change depending upon some external information. This is done with a construct that looks superficially like C's

`switch` statement, but is really more like a C `union` or an ASN.1 `CHOICE`.

```
select (type) {
    case value1: Type1
    case value2: Type2
    ...
} name;
```

The type that controls the contents of a select is always an enum. Here's an example:

```
struct {
   select (KeyExchangeAlgorithm) {
     case rsa: EncryptedPreMasterSecret;
     case diffie_hellman: DiffieHellmanClientPublicValue;
   } exchange_keys;
} ClientKeyExchange;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;
```

This switches the definition of the ClientKeyExchange struct based on the value of KeyExchangeAlgorithm. If KeyExchangeAlgorithm is rsa, then the value is of type EncryptedPreMasterSecret. If it is of diffie_hellman then the value is of type DiffieHellmanClientPublicValue. Note that the value of KeyExchangeAlgorithm is not represented anywhere in the message, but is merely part of the system context. Some variants, however, refer to other fields in the message.

3.9 Handshake Message Structure

Each SSL handshake message consists of a simple header followed by a body which is dependent on the message type. Figure 3.7 shows the definition for the message:

The header is 4 bytes long and consists of a 1-byte type field followed by 3 length bytes. The length field refers to the length of the rest of the handshake message (not including the type and length fields). The contents of the rest of the message are completely dependent on the msg_type field. The length field is not strictly necessary because the end of the handshake message can be determined from the components. However, having an explicit length field makes it much easier to implement the protocol. The implementation can break apart sequential handshake messages in one layer and then process them in another. Note that the mapping of handshake messages to SSL records is not fixed. Multiple handshake messages may appear in a single record. Theoretically, a single handshake message might span multiple records, but in practice this does not occur.

```
struct {
   HandshakeType msg_type;
   uint24 length;
   select (HandshakeType) {
      case hello_request:      HelloRequest;
      case client_hello:       ClientHello;
      case server_hello:       ServerHello;
      case certificate:        Certificate;
      case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done:   ServerHelloDone;
      case certificate_verify:  CertificateVerify;
      case client_key_exchange: ClientKeyExchange;
      case finished:           Finished;
   } body;
} Handshake;

enum {
   hello_request(0), client_hello(1), server_hello(2),
   certificate(11), server_key_exchange (12),
   certificate_request(13), server_hello_done(14),
   certificate_verify(15), client_key_exchange(16),
   finished(20), (255)
} HandshakeType;
```

Figure 3.7 SSL Handshake Message

3.10 Handshake Messages

The first step in creating an SSL connection is the handshake, which establishes the algorithms and keying material for the rest of the connection. This section covers the SSL handshake we showed earlier in detail. For each handshake message, we show the relevant piece of specification language and the expanded trace of that handshake message. Then, we explain the meaning and values of each field.

There are two ways to read this section. If you want to really understand what's going on, then read each piece of specification and match it up with the `ssldump` trace, using the accompanying text as a guide. If all you want to do is get the general idea, then just read the text, referring to the structure definitions and the protocol traces as a guide.

ClientHello

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-1>;
  CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;

struct {
  uint8 major;
  uint8 minor;
} ProtocolVersion;

struct {
  uint32 gmt_unix_time;
  opaque random_bytes[28];
} Random;

opaque SessionID<0..32>;
uint8 CipherSuite[2];
enum { null(0), (255) } CompressionMethod;
```

```
ClientHello
  Version 3.1
  random[32]=
    38 f3 cb de 80 4c b4 79 0a 07 9f b3
    51 ba b8 62 69 e3 8f bf ce c7 ff 25
    3c 3b 84 16 38 b2 5e f7
  cipher suites
            TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_IDEA_CBC_SHA
            TLS_RSA_WITH_RC4_128_SHA
            TLS_RSA_WITH_RC4_128_MD5
            TLS_DHE_RSA_WITH_DES_CBC_SHA
            TLS_DHE_DSS_WITH_DES_CBC_SHA
            TLS_RSA_WITH_DES_CBC_SHA
            TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
            TLS_RSA_EXPORT_WITH_RC4_40_MD5
  compression methods
            NULL
```

The first handshake message sent is always the ClientHello message. The main purpose of this message is for the client to communicate its preferences for connection parameters. After sending the ClientHello, the client waits for the server to send its hello message. In SSLv3 and TLS, the client presents its acceptable parameters and the server chooses the final set. The three negotiable parameters are the version, represented by the client_version field, the cryptographic algorithms, represented by cipher_suites, and the compression algorithm, represented by compression_methods. Figure 3.8 lists the main TLS cipher suites.

The client_version field should contain the highest SSL version number that the client is prepared to speak. For SSLv3, this means major=3 and minor=0. For TLS, major=3 and minor=1. In this trace, the client has indicated that it speaks TLS. In general, it is expected that an implementation speaks all lesser versions. SSL provides no facility for indicating that you do not speak previous versions. This means that if a client knows that (say) version 2.0 is insecure, there's no way to tell the server: use version 3.0 or above. You simply have to try out the server and terminate the connection if the server chooses version 2.

All cryptographic selections for a connection are bundled together into CipherSuites, which are represented by arbitrarily chosen 2-byte constants. The CipherSuite specifies the server authentication algorithm, the key exchange algorithm, the bulk encryption algorithm, and the digest (message integrity) algorithm. The suites are listed in the order of descending client preference. Figure 3.8 shows a list of the TLS cipher suites.

The cipher suites marked with a † are not part of the TLS standard. They were introduced in [Banes1999] in response to the liberalization of U.S. export policy that allowed 1024-bit key agreement and 56-bit symmetric encryption. These cipher suites allow 1024-bit RSA and DH combined with either DES or 56-bit RC4. [Banes1999] also adds support for DSS/DH with RC4-128, which had been omitted in the TLS standard. Although these algorithms are widely implemented, the full liberalization of U.S. export policy has rendered them unnecessary, so it's unclear if they will ever be standardized. Note that because these algorithms are labelled export, they go through the export key derivation procedure described in Section 3.11.

The final negotiable parameter is the compression algorithm. Compression algorithms are represented by 1-byte constants. In actual practice, due mainly to patent considerations, no compression algorithms have ever been defined for SSL, and so the only supported algorithm is the mandatory NULL algorithm, which does not change the data. Some proprietary implementations implement private compression algorithms, as does OpenSSL.

The second parameter in the ClientHello is a 32-byte value, (called random) which is used as one of the inputs to the key generation procedure. The first 4 bytes of the value are the time of day when the message was generated (in seconds since the Unix epoch, 12:00 midnight, January 1, 1970 GMT) and the other 28 bytes are supposed to be randomly generated. Note that the protocol will operate correctly even if the first 4 bytes are not time. Client and server clocks do not have to be synchronized. The purpose of the random data is to ensure that even if the same pre_master_secret is used the generated encryption and MAC keys will be different. This prevents replay attack.

Cipher Suite	Auth	Key Exchange	Encryption	Digest	Number
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5	0x0001
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA	0x0002
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RSA_EXPORT	RC4_40	MD5	0x0003
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5	0x0004
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	RC4_128	SHA	0x0005
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	RSA_EXPORT	RC2_40_CBC	MD5	0x0006
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA_CBC	SHA	0x0007
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	RSA_EXPORT	DES40_CBC	SHA	0x0008
TLS_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES_CBC	SHA	0x0009
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES_EDE_CBC	SHA	0x000A
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_DSS_EXPORT	DES_40_CBC	SHA	0x000B
TLS_DH_DSS_WITH_DES_CBC_SHA	DSS	DH	DES_CBC	SHA	0x000C
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DH	3DES_EDE_CBC	SHA	0x000D
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_EXPORT	DES_40_CBC	SHA	0x000E
TLS_DH_RSA_WITH_DES_CBC_SHA	RSA	DH	DES_CBC	SHA	0x000F
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DH	3DES_EDE_CBC	SHA	0x0010
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DSS	DHE_EXPORT	DES_40_CBC	SHA	0x0011
TLS_DHE_DSS_WITH_DES_CBC_SHA	DSS	DHE	DES_CBC	SHA	0x0012
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DHE	3DES_EDE_CBC	SHA	0x0013
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DHE_EXPORT	DES_40_CBC	SHA	0x0014
TLS_DHE_RSA_WITH_DES_CBC_SHA	RSA	DHE	DES_CBC	SHA	0x0015
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DHE	3DES_EDE_CBC	SHA	0x0016
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	-	DH_EXPORT	RC4_40	MD5	0x0017
TLS_DH_anon_WITH_RC4_128_MD5	-	DH	RC4_128	MD5	0x0018
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	-	DH	DES_40_CBC	SHA	0x0019
TLS_DH_anon_WITH_DES_CBC_SHA	-	DH	DES_CBC	SHA	0x001A
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	-	DH	3DES_EDE_CBC	SHA	0x001B
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0062
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA †	RSA	RSA	DES_CBC	SHA	0x0063
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0064
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA †	RSA	RSA	RC4_56	SHA	0x0065
TLS_DHE_DSS_WITH_RC4_128_SHA †	RSA	RSA	RC4_56	SHA	0x0066

Figure 3.8 Available TLS cipher suites

The ClientHello also contains a session_id parameter. The session ID is a byte string of up to 32 bytes which can be used by the client to indicate that it wishes to reuse the cryptographic keying material from a previous connection rather than generate new

material. This is typically much faster because public key operations are computationally expensive. In this trace, the client and server had no previous session to resume and so the session_id field was zero length. We'll see an example of a resumed session in Chapter 4.

ServerHello

```
struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
} ServerHello;
```

```
ServerHello
  SSL version 3.1
  random[32]=
    38 f3 cb d1 33 63 1c c7 2e 8c 56 43
    9e fb 20 70 cc 4b 16 06 4d 5a 8b 15
    e3 9f 0d 47 39 16 5f 5c
  session_id[32]=
    74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
    27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
    c4 b3 a4 f6 61 ef f5 cd
  cipherSuite           TLS_RSA_WITH_DES_CBC_SHA
  compressionMethod           NULL
```

The ServerHello message is used by the server to choose from the various options offered to it by the client. The server_version, cipher_suite, and compression_method fields in this message are the version, ciphers, and compression algorithms that will be used for this SSL connection.

In this trace, the client and the server both speak TLS, so the chosen version is 3.1. The server has chosen the cipher suite TLS_RSA_WITH_DES_CBC_SHA: RSA key exchange with DES-CBC for message encryption and SHA for message digest. Note that the server is under no obligation to choose the client's preferred cipher suite, even if it is supported. The SSL specification is silent on how the choice is made among the client's supported ciphers. The server is free to defer to the client's preferences or impose its own preferences. Common practice, however, is for the server to choose the client-supported cipher that it prefers most. This is somewhat inconvenient for client implementors because they cannot predict which cipher the server will choose. Among other things, this means that they can't give the user feedback about the likely consequences of making a given connection until the connection process has started.

Finally, the server has chosen the only available compression algorithm, NULL. The server also provides a random value. Together with the client's random value and the pre_master_secret (to be discussed later), it is used to generate the keying material to be used for this connection. This ensures that even if the client is broken and uses the same random value for two handshakes, the final cryptographic keys will be different. This also serves to protect against attempts to replay the handshake, as discussed in Section 1.8.

Typically, the server provides a session_id which the client could later use to resume the session. If a server doesn't want to resume sessions, a zero-length session ID can be provided. In this case, the server has provided one, indicating that it's prepared to resume this session. Exactly how session IDs are generated is left up to the implementation. Although they're often generated purely randomly, it is possible for them to have some structure that makes it easier for the server to look up the session in its cache.

Certificate

```
struct {
  ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;

opaque ASN.1Cert<2^24-1>;
```

```
Certificate
  Subject
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Server test cert (512 bit)
  Issuer
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Test CA (1024 bit)
  Serial         04
  certificate[493]=
    30 82 01 e9 30 82 01 52 02 01 04 30
    0d 06 09 2a 86 48 86 f7 0d 01 01 04
    05 00 30 5b 31 0b 30 09 06 03 55 04
    06 13 02 41 55 31 13 30 11 06 03 55
    04 08 13 0a 51 75 65 65 6e 73 6c 61
    6e 64 31 1a 30 18 06 03 55 04 0a 13
    11 43 72 79 70 74 53 6f 66 74 20 50
    74 79 20 4c 74 64 31 1b 30 19 06 03
    55 04 03 13 12 54 65 73 74 20 43 41
    20 28 31 30 32 34 20 62 69 74 29 30
```

```
1e 17 0d 39 38 30 36 32 39 32 33 35
32 34 30 5a 17 0d 30 30 30 36 32 38
32 33 35 32 34 30 5a 30 63 31 0b 30
09 06 03 55 04 06 13 02 41 55 31 13
30 11 06 03 55 04 08 13 0a 51 75 65
65 6e 73 6c 61 6e 64 31 1a 30 18 06
03 55 04 0a 13 11 43 72 79 70 74 53
6f 66 74 20 50 74 79 20 4c 74 64 31
23 30 21 06 03 55 04 03 13 1a 53 65
72 76 65 72 20 74 65 73 74 20 63 65
72 74 20 28 35 31 32 20 62 69 74 29
30 5c 30 0d 06 09 2a 86 48 86 f7 0d
01 01 01 05 00 03 4b 00 30 48 02 41
00 9f b3 c3 84 27 95 ff 12 31 52 0f
15 ef 46 11 c4 ad 80 e6 36 5b 0f dd
80 d7 61 8d e0 fc 72 45 09 34 fe 55
66 45 43 4c 68 97 6a fe a8 a0 a5 df
5f 78 ff ee d7 64 b8 3f 04 cb 6f ff
2a fe fe b9 ed 02 03 01 00 01 30 0d
06 09 2a 86 48 86 f7 0d 01 01 04 05
00 03 81 81 00 95 be f7 e4 19 27 b6
18 78 03 15 f9 8e b9 ae 08 b2 36 fd
25 58 48 99 63 00 4a 23 82 96 46 65
30 44 83 26 3b 2c ce 0f fa f9 df d6
fb c4 eb 6c e6 e1 6b 3a 65 f7 91 62
bd 70 55 b9 c6 e3 f5 db 9d 87 b0 0e
21 9b b5 87 53 00 3e 5c a4 9d cf 54
77 cd 7a bf 3d c5 7a 30 78 aa a5 28
69 78 e7 96 4a c8 80 46 eb fe e9 fb
8d 24 bc e9 63 9e d2 14 61 c0 79 09
15 41 9c 3d 97 fd 34 3d b6 12 d7 3e
01
```

The Certificate message is simply a sequence of X.509 certificates. The certificates are presented in order, with the first certificate being the certificate belonging to the server and the next certificate (if there is one) containing the key that certifies the server's certificate, and so on. In all common SSL and TLS cipher suites, the server must send its certificate (there are several rarely used cipher suites where the server remains anonymous). The key in the certificate is then either used to encrypt the pre_master_secret or to verify the ServerKeyExchange, depending on which cipher suite was chosen.

In this case, the server has sent only one certificate, for *Server Test Cert (512 bit)*. It's assumed that the client already has the CA certificate for *Test CA (1024 Bit)*. What's presented here is a summary of the most important certificate information. More detail about the structure of X.509 certificates can be found in Chapter 1.

ServerHelloDone

```
struct { } ServerHelloDone;
```

The ServerHelloDone is an empty message that indicates that the server has sent all the handshake messages it will send in this phase. This is required because there are some optional messages that can appear after the Certificate message. We'll see some examples of that in Chapter 4.

ClientKeyExchange

```
struct {
   select (KeyExchangeAlgorithm) {
      case rsa: EncryptedPreMasterSecret;
      case diffie_hellman: DiffieHellmanClientPublicValue;
   } exchange_keys;
} ClientKeyExchange;

struct {
   ProtocolVersion client_version;
   opaque random[46];
} PreMasterSecret;

struct {
   public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct {
   select (PublicValueEncoding) {
      case implicit: struct {};
      case explicit: opaque DH_Yc<1..2^16-1>;
   } dh_public;
} DiffieHellmanClientPublicValue;
```

```
ClientKeyExchange
   EncryptedPreMasterSecret[64]=
      17 4d 00 32 bc b2 af 95 09 0a 45 24
      97 d8 34 dc 73 20 4d 00 91 a5 0d ed
      c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
      bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3
      00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
      0b bc db a2
```

Ex.1024
Askeladden, L.L.C. / Page 148 of 788

The ClientKeyExchange message provides the client's contribution to the creation of the pre_master_secret. When RSA key exchange is being used, this simply means that the client generates a PreMasterSecret structure and encrypts it under the server's RSA key. The pre_master_secret is a 48-byte quantity consisting of two version number bytes followed by 46 randomly generated bytes. It is critical that these bytes be chosen using a cryptographically secure RNG (see Chapter 5 for information on RNGs.) RFC 1750 [Eastlake1994] provides guidance on random number generation. Wagner and Goldberg's successful attack on Netscape's SSLv2 implementation [Goldberg1996] was based on flaws in Netscape's RNG. We provide some guidance on generating strong random numbers in Chapter 5.

> Wagner and Goldberg disassembled the Netscape RNG and observed that the RNG was seeded with the time of day and the process ID. Since both of these numbers are fairly limited in range, it was fairly easy to search the entire space and recover the SSL session key. This process took about an hour at the time and would be faster now.

The public-key-encrypted operator in the EncryptedPreMasterSecret struct means that the pre_master_secret field is to be encrypted using the recipient's public key. The encrypted key is then represented as a variable length vector:

```
opaque encrypted_data<0..2^16-1>
```

The ClientKeyExchange message is one of only two messages that are different between SSLv3 and TLS. This difference is due to a bug in Netscape's SSLv3 implementation that was mimicked by other SSLv3 implementors. When RSA is being used for encryption, the length of the encrypted data is redundant because the length of the EncryptedPreMasterSecret is unambiguously determined both by the length of the ClientKeyExchange message and by the length of the server's RSA key. As a consequence, Netscape's SSLv3 implementation omitted the length bytes when encoding the EncryptedPreMasterSecret, and most other SSLv3 implementations followed their (incorrect) lead. That is to say that most SSLv3 implementations violate the SSLv3 specification. An implementation note in the TLS specifcation points out that most SSLv3 implementations are broken and requires that TLS implementations correctly include the length bytes. The trace we are looking at is from an implementation in TLS mode, and therefore the length bytes are included.

ChangeCipherSpec

```
struct {
    ChangeCipherSpecType type;
} ChangeCipherSpec;

enum { change_cipher_spec(1), (255) } ChangeCipherSpecType;
```

The ChangeCipherSpec message indicates that the sending implementation has switched to the newly negotiated algorithms and keying material and that future messages will be protected using those algorithms. The ChangeCipherSpec message is unique in that it is not properly part of the handshake but rather has its own content type. This is described as a performance improvement in the specification, which says:

> Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent TLS Protocol content type, and is not actually a TLS handshake message.

However, only one implementation known to the author (Netscape's) makes use of this feature in any interesting way. Netscape uses this feature to make sure that the ChangeCipherSpec message is transmitted separately from the Finished message. Because the ChangeCipherSpec message must not be encrypted and the Finished message must be encrypted, they cannot be transmitted in the same record. Having them in different layers ensures this.

This trick is useful only if your implementation attempts to send multiple handshake messages in the same record. As we'll see in Chapter 6, it's sometimes a performance win to send multiple handshake messages in the same TCP segment, and sending them in the same record is one way of doing this. However, many implementations instead opt to transmit multiple records in the same TCP segment, which has much the same effect. For such implementations, this protocol feature is an inconvenience which complicates the handshake state machine.

A typical implementation is to have the handshake state machine read only handshake type traffic. However, it also needs to be aware that the ChangeCipherSpec has been received, and this information must be carried out of band from the record layer processing. If the ChangeCipherSpec message were a handshake message, the handshake layer could simply read it off the network. Instead, the record layer code needs to have special support for recognizing the ChangeCipherSpec message and either telling the handshake layer or throwing an error depending on whether the handshake is in the right state.

The ChangeCipherSpec message consists only of a single type byte, which is mainly a placeholder since it can currently have only a single value of 1. No expansion of the use of this message has been publicly proposed.

Finished

```
SSLv3:
struct {
  opaque md5_hash[16];
  opaque sha_hash[20];
} Finished;
```

```
TLS:
struct {
   opaque verify_data[12];
} Finished;

Finished
   verify_data[12]=
      03 3b 69 b7 26 e3 0e 23 fc 03 79 27
```

The Finished message is the first message encrypted with the new cryptographic parameters. It allows an implementation to verify that none of the handshake messages have been tampered with by an attacker.

Although the details of the SSLv3 and TLS Finished messages are different, the general concept is similar. Each side sends a digest of the negotiated master secret and the concatenated handshake messages to the other, which compares it to the same result computed locally. If they differ, the handshake has been tampered with.

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

md5_hash = MD5(master_secret + pad2 +
          MD5(handshake_messages +
             Sender + master_secret + pad1));
sha_hash = SHA-1(master_secret +  pad2 +
          SHA-1(handshake_messages +
             Sender + master_secret + pad1));
```

Figure 3.9 SSLv3 Finished computations

The SSLv3 Finished message is modeled after the popular cryptographic message authentication function HMAC [Krawczyk1997], which uses nested hashes to create a construction which provably has certain security properties. Figure 3.9 shows the SSLv3 computation. The plus sign indicates concatenation, so the input to the first invocation of MD5 (the inner one) is the string created by concatenating all the handshake messages, the Sender constant, the master secret, and some pad bytes (pad1 is a string of the byte 0x36 repeated 48 times). The input to the second invocation of MD5 is the string created by concatenating the master secret, some pad bytes (pad2 is the a string of the byte 0x5c repeated 48 times), and the output of the first MD5 computation. The SHA-1 computation is identical except that the pad bytes are repeated 40 times instead of 48. The purpose of the Sender constant is to ensure that the client and server produce different Finished messages. This prevents an attacker from reflecting a Finished message back to its sender.

Although the SSLv3 Finished message is based on HMAC, it was modified to be usable for SSL. In particular, because the master secret is available only midway through the handshake, the handshake messages are the first input to the inner digest, whereas in HMAC, the key is always the first input. Consequently, some of the security

properties of HMAC may not apply. Comments from the HMAC designers were the primary reason why the computation was changed in TLS.

```
verify_data =
  PRF(master_secret, finished_label, MD5(handshake_messages) +
    SHA-1(handshake_messages)) [0..11];
```

Figure 3.10 TLS Finished computation

TLS makes wide use of an HMAC-based *pseudo-random function* (PRF) for a number of cryptographic computations, including the Finished message (see Figure 3.10). (We'll see more of the PRF in the section on key derivation.) This PRF is used to create the verify_data value. The inputs are the master secret, an ASCII string for the finished label ("`client`" or "`server`") and an MD5 and SHA-1 digest of the handshake messages. The [0..11] indicates that the first 12 bytes of PRF output are used to create the verify_data.

Note that both MD5 and SHA-1 are used in the creation of the Finished message. The rationale for this is to provide additional security against weaknesses in the digest algorithm. This way, an attacker must break both MD5 and SHA-1 in order to forge a Finished message.

Programming Note: Processing Finished

Note that although the Finished message must include the digest of all the handshake messages so far, it does not include its own digest. However, the second Finished message must include a digest of the first. This creates an interesting programming problem for implementors. A naive implementation might write its handshake handler something like the code in Figure 3.11.

Unfortunately, when the recipient goes to compute the Finished value, the digest will include the just-received Finished message, and so the values won't match. However, the server can't just stop digesting because it needs to include the digest of the first Finished in its response to the client's Finished. Thus, it must either maintain two sets of digests or make a copy of the digest objects before processing the Finished message, as shown in Figure 3.12.

3.11 Key Derivation

Once the pre_master_secret has been exchanged, each implementation needs to expand it into the individual cryptographic keys that will be used for encryption, authentication, etc. This expansion is accomplished using a key derivation function. The SSLv3 and TLS key derivation functions are similar, but differ in the precise cryptographic transformations they use. We'll describe the TLS key derivation function with an aside at the

```
receive_handshake_message(context,message) {
  digest_handshake_message(context,message);
  switch(message.type){
    case ClientHello:
      handle_client_hello(context,message);
      break;
      ...
    case Finished:
      handle_finished(context,message);
      break;
  }
```

Figure 3.11 Naive Finished handling

```
receive_handshake_message(context,message) {
  if(message.type==Finished){
    digests=copy_digests(context);
  }
  digest_handshake_message(context,message);

  switch(message.type){
    case ClientHello:
      handle_client_hello(context,message);
      break;
      ...
    case Finished:
      handle_finished(context,message,digests);
      break;
  }
```

Figure 3.12 Correct Finished handling

end describing the differences from the SSLv3 process. You can refer to Figure 3.13 for a graphical representation of the process.

The PRF

The entire TLS key derivation process uses PRF as the key derivation function. The point of this is to expand the pre_master_secret into another secret called master_secret, and then expand the master_secret into the cryptographic keys that we

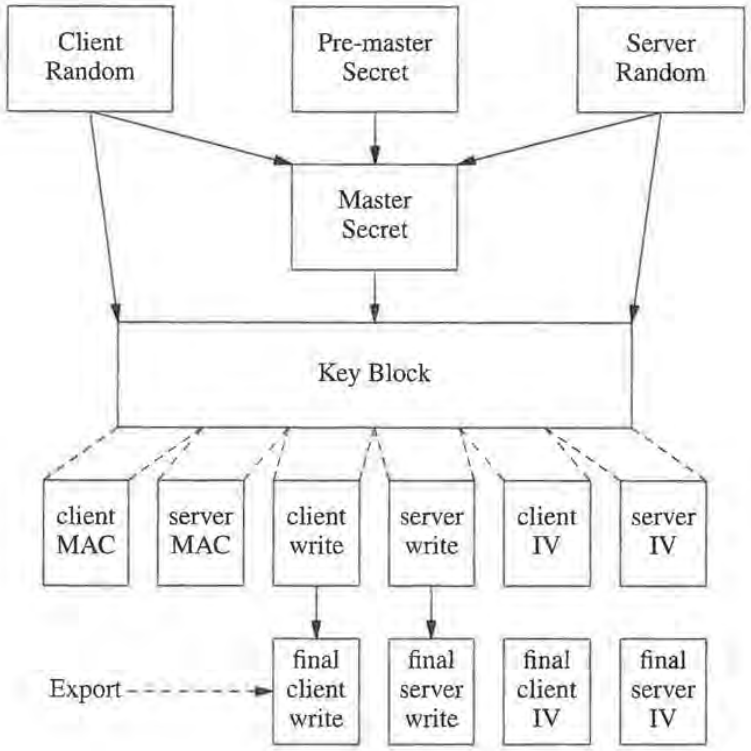


Figure 3.13 Key derivation

need for the various encryption and MAC algorithms. As with the key derivation function in Section 1.8, the client and the server each independently use the same key derivation function to generate the keys from the shared master_secret. The PRF takes three arguments, a secret (presumably random), a fixed ASCII string (the label), and a seed (presumably random, but public), as in PRF(secret, label, seed). The purpose of the label is to allow you to use the PRF to generate different keys from the same secret material simply by using different labels for each key. The label is simply digested as the corresponding ASCII string.

The PRF generates an arbitrary length string of pseudo-random bytes, which are referred to by using the TLS vector notation. So, PRF()[0..9] refers to the first 10 bytes of the PRF output. The PRF computation is fairly complicated and not that interesting except to implementors. We cover it here only for completeness, so feel free to skip over this stuff. It won't be on the exam.

First, the secret part is split into two halves, S1 and S2. S1 is the first half of the secret and S2 is the second half. If the original secret is odd-sized, then the last byte of

S1 and the first byte of S2 are the same.

Each half is used as the secret component in an expansion function P_hash, which is based on HMAC. P_hash generates an arbitrary length byte string by the process shown in Figure 3.14.

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

A() is defined as:
    A(0) = seed
    A(i) = HMAC_hash(secret, A(i-1))

HMAC_hash means HMAC using a given hash algorithm, e.g.,
HMAC_MD5 means HMAC using MD5.
```

Figure 3.14 P_hash function

Now that we've got these functions defined, we construct PRF by XORing a P_MD5 and a P_SHA-1:

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) ⊕
                           P_SHA-1(S2, label + seed);
```

Because MD5 digests are 16 bytes long and SHA-1 digests are 20 bytes long, P_MD5 and P_SHA-1 don't line up. So, if you want to produce an 80-byte output, you have to run P_MD5 five times and P_SHA-1 four times.

The three non-fixed inputs to the key derivation function are the pre_master_secret, the client_random, and the server_random. The pre_master_secret is simply a secret byte string. When RSA key exchange is being used, it's a 48-byte string generated by the client. It's encrypted under the server's public key. The pre_master_secret is the only source of secret information in this process. The other two inputs, the client_random and server_random, are the random values from the ClientHello and ServerHello messages.

The first step in the key derivation is to transform the pre_master_secret into the master_secret. This is done by applying the pseudo-random function PRF to the pre_master_secret, client_random, and server_random:

```
master_secret = PRF(pre_master_secret, "master secret",
                client_random + server_random) [0..47];
```

We now use the PRF with the master_secret to give us enough keying material for all the algorithms we'll need. Depending on the cipher suite being used, we might need up to six values, an encryption key, a MAC key, and an initializaton vector (IV) for both

client and server. First, we add up the lengths of all the values we need and produce that many bytes using the PRF. For instance, if we're using DES and MD5, we need 8 bytes for each encryption key, 16 bytes for each HMAC-MD5 key, and 8 bytes for each IV, for a total of 48 bytes:

```
key_block = PRF(master_secret, "key expansion",
          server_random + client_random)
```

Note that the order of the client_random and server_random is different here from the master_secret computation.

Once we have the key block, we simply slice and dice it one key at a time, in the following order: client_write_MAC_secret, server_write_MAC_secret, client_write_key, server_write_key, client_write_IV, server_write_IV.

Exportable Algorithms

Due to U.S. export restrictions at the time of the SSLv3 and TLS design, exportable ciphers needed to have their encryption keys shortened so that they could be broken with no more than 2^{40} operations. In order to do this, we first generate a short intermediate and then reexpand it using the PRF. This approach prevents a precomputation attack in which the attacker simply builds a big table for all 2^{40} keys. The server and client random values act as a *salt* preventing this attack. The attacker can still attack the key by trying all of the intermediate keys but the time/memory tradeoff won't work.

> The export situation has changed since the design of SSL/TLS. 40-bit keys are essentially worthless for data security. Attacking a 40-bit key is easily within the range of the most modest commercial attacker. In 1998 a dedicated DES keysearch machine was built [Gilmore1998], showing that 56-bit keys could be compromised by a dedicated attacker. In response to this, the U.S. government decided that cryptographic systems with 56-bit keys would now be exportable. Changes have been proposed to the TLS spec to include 56-bit exportable ciphers. These ciphers also allow 1024-bit RSA keys instead of 512-bit RSA keys. In January 2000 the U.S. government essentially removed all export restrictions. However, there is still a lot of old software that has only exportable modes so it's important to know about them. The Bureau of Export Administration Web site describes them and can be reached at
> `http://www.bxa.doc.gov/Encryption/Default.htm/`.

Also, we're required (again for export reasons) to use a non-secret IV (normally the SSL IV is generated partially from secret data and is therefore secret) so we derive it purely from the public client_randomand server_random values. The first half of the iv_block is used to generate the IV that the client uses to encrypt data, and the second half for the IV that the server uses to encrypt data (see Figure 3.15).

```
final_client_write_key =
    PRF(client_write_key, "client write key", client_random + server_random)
final_server_write_key =
    PRF(server_write_key, "server write key", client_random + server_random)

iv_block = PRF(0, "IV block", client_random + server_random)
```

Figure 3.15 Export keying material derivation

SSLv3 Key Derivation

SLv3 key derivation is similar to TLS key derivation, except that the PRF is replaced with a series of expansion functions based on a combination of MD5 and SHA-1. The use of the constants "A","BB", etc. ensures that the output of each digest is different, even though the secret data is the same. This process is shown in Figure 3.16.

```
master_secret =
  MD5(pre_master_secret + SHA-1("A" + pre_master_secret +
    client_random + server_random)) +
  MD5(pre_master_secret + SHA-1("BB" + pre_master_secret +
    client_random + server_random)) +
  MD5(pre_master_secret + SHA-1("CCC" + pre_master_secret +
    client_random + server_random))

key_block =
  MD5(master_secret + SHA-1("A" + master_secret +
    server_random + client_random)) +
  MD5(master_secret + SHA-1("BB" + master_secret +
    server_random + client_random)) +
  MD5(master_secret + SHA-1("CCC" + master_secret +
    server_random + client_random)) +
  ...
```

Figure 3.16 SSLv3 key derivation

Note that the order of the client_random and server_random is opposite for these two computations.

As with TLS, when being used with exportable ciphers, SSLv3 postprocesses the keying material to weaken it, as shown in Figure 3.17.

```
final_client_write_key =
    MD5(client_write_key + client_random + server_random);
final_server_write_key =
    MD5(server_write_key + server_random + client_random);
```

```
client_write_IV =
  MD5(client_random + server_random);
server_write_IV =
  MD5(server_random + client_random);
```

Figure 3.17 SSLv3 export key derivation

Note that even though the SSLv3 key derivation was changed for TLS, no known weaknesses exist in the SSLv3 key derivation.

3.12 Record Protocol

The SSL record protocol is a fairly simple encapsulation protocol. It consists of a series of encrypted, integrity-protected records. Each record consists of a small header block and an encrypted data block which itself contains the content and a MAC. Figure 3.18 shows an example record. The header is shown in white and the encrypted payload is shaded. We've drawn the various pieces of the encrypted payload as they would appear if the encryption were removed.

The record shown has been encrypted with a block cipher such as DES using MD5 as a MAC. Therefore, the record needs to be padded out to fit the DES block length. Note that although the encrypted data must be a multiple of the block length, the record itself is not aligned on block boundaries.

In this trace, we have the client and the server writing one record each. The cipher suite, as we know from the ServerHello message, is TLS_RSA_WITH_DES_CBC_SHA.

SSL data is broken up into a series of *fragments*. The protocol definition provides a maximum fragment size but otherwise doesn't specify how to determine what fragment boundaries are, but usually they're one-to-one with application data write calls. However, boundaries of calls to `SSL_write()` (or equivalent) need not be preserved. Consequently, the record layer may break up any given write into multiple records or conversely aggregate the input from multiple calls into a single record. The maximum fragment data size is 2^{14} bytes, so larger writes must be fragmented into pieces of this size or smaller. Note that the length field is long enough to indicate records up to $2^{16} - 1$ bytes. Indeed, older versions of Internet Explorer incorrectly generated records of greater than 2^{14} bytes.

Once the data has been fragmented, it's compressed—or it would be if any compression algorithms were defined. Because none are, no compression is done. A MAC is computed over the compressed data fragment and concatenated with the data. The MAC covers the data and the headers, as well as including a sequence number to prevent replay and reordering attacks. Figure 3.19 shows the MAC algorithms.

```
TLS MAC = HMAC_hash (MAC_write_secret, seq_num + type + version +
                     length + content)
```

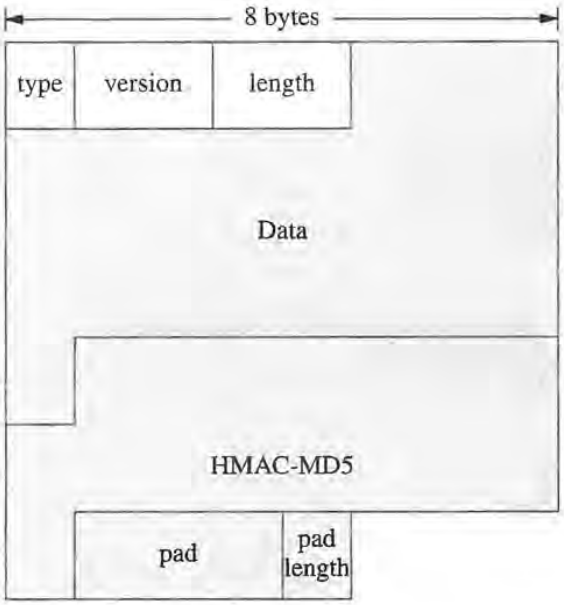


Figure 3.18 Diagram of an SSL record

```
SSLv3 MAC=hash(MAC_write_secret + pad_2 +
        hash(MAC_write_secret + pad_1 + seq_num + content_type +
            length + content));
```

Figure 3.19 MAC computations

seq_num is a 64-bit sequence number, which is independent for the client and server (i.e. the first message that both the client and server send has sequence number zero). The content type here is one of the content types listed in Figure 3.20. The version is simply the SSL/TLS version number. The length is the length of the content (i.e., the data in the record); it is *not* the same as the length in the record header, because that length includes the MAC and possibly padding.

The TLS MAC is standard HMAC. The SSLv3 MAC is based on an early draft of HMAC. pad_1 is the byte 0x36 repeated 48 times for MD5 and 40 times for SHA-1. pad_2 is the byte 0x5c repeated the same number of times. The intention of the padding is to ensure that the MAC key plus pad fills the entire first block of the message digest. Because MD5 and SHA-1 blocks are 64 bytes long, a 16-byte key means a 48-byte pad. Note that this implies that the padding for SHA-1 (a 20-byte key) should be 44 bytes,

but it's 40 in the specification. This is an error in the SSLv3 specification, but it became part of Netscape's implementation and thus the specification was never changed. Once again, we can see that Netscape's implementation constituted the real baseline for SSLv3.

```
struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
} RecordHeader;

enum {
   change_cipher_spec(20), alert(21), handshake(22),
   application_data(23), (255)
} ContentType;

struct {
   uint8 major;
   uint8 minor;
} ProtocolVersion;
```

Figure 3.20 The record header

The sequence number serves to prevent various attacks on the message stream, including *replay* attacks, where the attacker repeats a record, and *reordering* attacks, where the attacker exchanges the order of one or more records. Note, however, that although the sequence number allows these attacks to be detected, because the sequence number isn't in the record, it does not allow them to be corrected. Thus, SSL must be run over a transport protocol that provides *reliable* transport (i.e., it must deliver data to the receiving application in the order it was transmitted without duplicates or omissions). TCP meets these requirements (the data may arrive in any order, but the kernel ensures that the application sees it in transmission order). However, UDP does not, and so SSL cannot be successfully run over UDP, because records might be arbitrarily lost or reordered, which would look like an active attack to the SSL implementation.

Note that the TLS specification describes records somewhat differently, without breaking them up into header and body. This presentation is intended to be easier to follow, and is more typical of how network protocols are usually presented.

The different type values correspond to the different types of data that can be carried over the TLS record layer, as shown in Figure 3.4. The length and version fields we've seen before. The length in the header is the total length of all the data following in the record, including the MAC and padding.

Stream Ciphers

Stream ciphers don't need any padding, so the MAC is simply appended to the data and the entire data block is encrypted. Because the stream cipher keystream can never be reused, the cipher encryption state must be retained from record to record. Logically, it is as if all the data to be encrypted were encrypted in a single pass by the cipher.

Block Ciphers

The length of the input to a block cipher must be a multiple of the block size (typically 8 bytes). Since the input data can be of arbitrary size, the data usually does not line up on a block boundary, and it is necessary to add a number of padding bytes. The most padding bytes that will ever be needed is 7, but up to 255 bytes can be added to hide the plaintext length from sniffers. The number of pad bytes used is recorded in the pad length byte. The value of the pad bytes must be equal to the value in the pad length byte.

For example, if the block length is 8 bytes, the length of the data is 16 bytes, and the MAC length is 16 bytes, then the length before padding is 33 bytes, because the length byte must always be included at the end. At least 7 bytes of padding will be needed, but 15, 23, etc., may also be used. Assuming that the minimum pad length of 7 bytes is used, the last 8 bytes would be 07 07 07 07 07 07 07 07 (7 pad bytes plus the pad length byte) for a total data length of 40 bytes. Note that this padding construct is incompatible with standard block cipher padding (see [RSA1993c]) which does not use a pad length byte, but merely records the total number of pad bytes in the padding. In the example here, the standard algorithm would give the pad 08 08 08 08 08 08 08 08.

All the current block ciphers use *cipher-block chaining* (CBC) mode. The chaining must be continued from record to record: The last cipher-block of record X is used as the IV for record X+1.

Null Cipher

As a special case, the Null cipher simply passes the data untransformed from input to output, with no padding. The starting cipher suite TLS_NULL_WITH_NULL_NULL doesn't even add a MAC, but simply puts the plaintext directly in the record. Note that TLS_NULL_WITH_NULL_NULL is available only during the handshake at the start of a connection. It is illegal to negotiate to it.

3.13 Alerts and Closure

The alert protocol is designed to allow one side to alert the other to exceptional conditions. Alert messages are very simple, conveying a severity level and a description. On

receiving an alert message of level fatal, an implementation must terminate the connection and the session cannot be resumed. A warning message may be ignored; however, an implementation may choose to treat warnings as fatal. Figure 3.21 shows the TLS alert message structure. The SSL alert message structure is identical, but fewer alerts are defined. See Chapter 4 for detailed coverage of all the alerts.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
   close_notify(0),
   unexpected_message(10),
   bad_record_mac(20),
   decryption_failed(21),
   record_overflow(22),
   decompression_failure(30),
   handshake_failure(40),
   no_certificate(41), SSLv3 only
   bad_certificate(42),
   unsupported_certificate(43),
   certificate_revoked(44),
   certificate_expired(45),
   certificate_unknown(46),
   illegal_parameter(47),
   unknown_ca(48),
   access_denied(49),
   decode_error(50),
   decrypt_error(51),
   export_restriction(60),
   protocol_version(70),
   insufficient_security(71),
   internal_error(80),
   user_cancelled(90),
   no_renegotiation(100),
   (255)
} AlertDescription;

struct {
   AlertLevel level;
   AlertDescription description;
} Alert;
```

Figure 3.21 TLS alert message

Closure

The close_notify alert is used to signal that the sender has sent all the data that it is going to send on this connection. The purpose of this alert is to prevent a truncation attack in which the attacker inserts a TCP FIN before the sender is finished sending data, forcing the receiver to think that all data has been received. As a consequence, unless the receiver receives a close_notify it can never be sure that more data wasn't forthcoming. If an implementation receives a TCP FIN without first receiving a close_notify it must mark the session as not resumable.

In practice, many protocols contain their own end-of-data markers, and it's fairly common practice for an implementation receiving one to simply send a close_notify and close the connection immediately without waiting for a close_notify. This behavior is implicitly allowed by the SSL and TLS specifications. This is what has happened in the trace we provided at the beginning of the chapter, where the client provided a close_notify but the server did not. Normally, the closure alert is sent with a level of warning.

Other Alerts

All the other alerts signal errors of various kinds. Although in theory implementations can continue past a non-fatal alert, in practice even such alerts indicate that one side requested something that the other side couldn't fulfill and so they result in connection closure. The TLS specification provides a complete list of all the available alerts and we'll discuss each alert in detail in Chapter 4.

Figure 3.22 shows an example trace terminating in an alert. The client has been configured to require DSS/DH, and the server only supports RSA. Thus, the server rejects the connection with a handshake_failure alert. Note that the server closes the connection immediately after sending the alert.

```
New TCP connection: speedy(1085) <-> romeo(4433)
1 0.0168 (0.0168) C>S   Handshake
      ClientHello
        Version 3.0
        cipher suites
                  TLS_DHE_DSS_WITH_DES_CBC_SHA
        compression methods
                  NULL
2 0.0171 (0.0002) S>C   Alert
        level       fatal
        value       handshake_failure
Server FIN
Client FIN
```

Figure 3.22 A connection ending in an alert

3.14 Summary

This chapter has provided a detailed discussion of the most basic mode of SSL, RSA server-only authentication. This provides the background for our discussion of future modes, all of which share many features with the basic mode.

The SSL handshake negotiates cryptographic algorithms and establishes keying material between client and server. The client offers a choice of cryptographic algorithms and the server selects the algorithms to be used. The server provides its public key using a Certificate message.

The pre_master_secret *is encrypted under the server's public key.* A *key derivation function* is used to convert the pre_master_secret into the cryptographic keys used for the connection.

A Finished *message is used to provide integrity for the entire handshake.* The Finished message contains a message digest of all the handshake messages, preventing an attacker from tampering with the handshake.

The data stream is segmented into records. Each record is individually encrypted and MACed, providing both confidentiality and message integrity. The MAC is computed over a sequence number to prevent replay and reordering attacks.

Errors are signalled via alerts. Alerts are a special type of record distinct from the main data stream, and they are used to signal handshake errors or problems with MACing or decryption.

A close_notify *alert is used to signal the end of the connection.* This prevents truncation attacks where the attacker forges a close on the underlying transport layer.

4

Advanced SSL

4.1 Introduction

The previous chapter covered the simplest type of SSL connection, server-only authentication using RSA. This chapter presents the rest of the major operational modes. As with the previous chapter, this chapter is divided into two sections. The first section covers the modes in overview form and the second describes each message in gory detail. Most readers will want to read the entire first section and skim the second section. Those who wish to challenge themselves should read the entire second section.

Both sections follow the same order of presentation. First, we discuss session resumption, which allows a new SSL connection to be formed without performing a full handshake. Next, we cover client authentication, which allows the client to authenticate itself using its public key and digital certificate.

Because United States export regulations restricted the size of keys that could be used by applications exported from the U.S., SSL has two modes designed to provide maximum security while still remaining exportable, Ephemeral RSA and Server Gated Cryptography. Although these modes are no longer necessary, they're still in widespread use. We show how these work and describe their benefits. We also show the TLS standard DH/DSS key exchange and less-used cipher suites that use Elliptic Curves, Kerberos, and FORTEZZA. We finish with a detailed discussion of all the SSL alerts and of SSLv2 backward compatibility.

A note on presentation: although the timelines we show will include all the messages in the handshake, the detailed presentation will only show the messages which are new or different from the basic handshake.

Ex.1024
Askeladden, L.L.C. / Page 165 of 788

4.2 Session Resumption

As we'll see in Chapter 6, the full SSL handshake can be very expensive, both in terms of CPU time and in terms of the number of round trips required to execute. In order to reduce this performance cost, SSL incorporates a session resumption mechanism. If the client and server have already communicated once, they can short-circuit the full handshake and proceed directly to data transfer.

The most expensive part of the handshake is the establishment of the pre_master_secret, which usually (except in the case of Kerberos) requires public key cryptography. A resumed handshake allows a new connection to use a master_secret established in a previous handshake. This avoids the computationally expensive operations required by public key cryptography.

Sessions versus Connections

SSL makes a distinction between a *connection* and a *session*. A *connection* represents one specific communications channel (typically mapped to a TCP connection), along with its keys, cipher choices, sequence number state, etc. A *session* is a virtual construct representing the negotiated algorithms and the master_secret. A new session is created every time a given client and server go through a full key exchange and establish a new master_secret.

Multiple connections can be associated with a given session. Although all connections in a given session share the same master_secret, each has its own encryption keys, MAC keys, and IVs. This is absolutely necessary for security reasons, because reuse of symmetric keying material can be extremely dangerous. Resumption allows the generation of a new set of symmetric keys and IVs from a common master_secret because the keys depend on the random values which are fresh for each connection. The new random values are combined with the old master_secret to produce new keys.

How It Works

The first time a client and server interact, they create both a new connection and a new session. If the server is prepared to resume the session, it gives the client a session_id in its ServerHello message and caches the master_secret for later reference. When the client initiates a new connection with the server, it uses that session_id in its ClientHello message. The server agrees to resume the session by using the same session_id in its ServerHello. At this point, the rest of the handshake is skipped and the stored master_secret is used to generate all the cryptographic keys. Figure 4.1 shows the timeline for this process.

4.3 Client Authentication

So far, all the handshakes we've discussed authenticate only the server. SSL also provides a mechanism for cryptographically authenticating the client. This would be useful

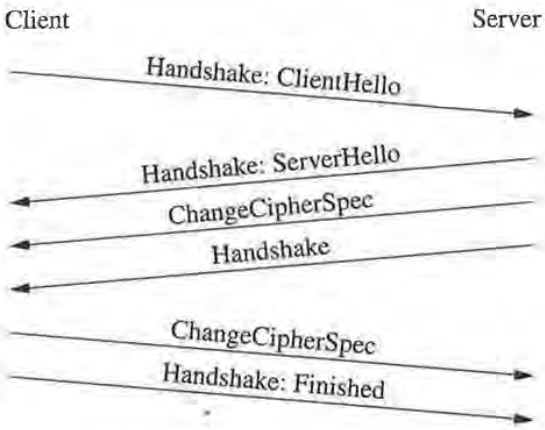


Figure 4.1 Resumed SSL session timeline

if a server wanted to restrict access to some services to only certain authorized clients and could use client authentication to do so. The idea here is that the client uses its private key to sign something, thus proving that it has possession of the private key corresponding to the certificate. Client authentication is initiated by the server sending a CertificateRequest message to the client. The client responds by sending a Certificate message (the same message that the server uses to transmit its certificate) and a CertificateVerify message. The CertificateVerify message is a string signed with the private key associated with the certificate that it transmitted. Client authentication is *always* initiated by the server. There is no mechanism for the client to unilaterally offer client authentication. Figure 4.2 shows the timeline for this process.

4.4 Ephemeral RSA

When SSL was designed, United States export regulations limited RSA encryption key lengths to 512 bits for exportable applications. Unfortunately, a 512-bit permanent RSA key presents an attractive target for attack. Thus, a server that wished to communicate with both domestic and exportable clients would have liked to have two key exchange keys, one 1024-bit and one 512-bit, assuming that it wanted to have strong cryptography when possible.

Although SSLv2 simply forced such a server to have two certified keys if it wanted to communicate at the highest possible security with both export and domestic clients, SSLv3 and TLS incorporate a feature called *ephemeral RSA* which allows

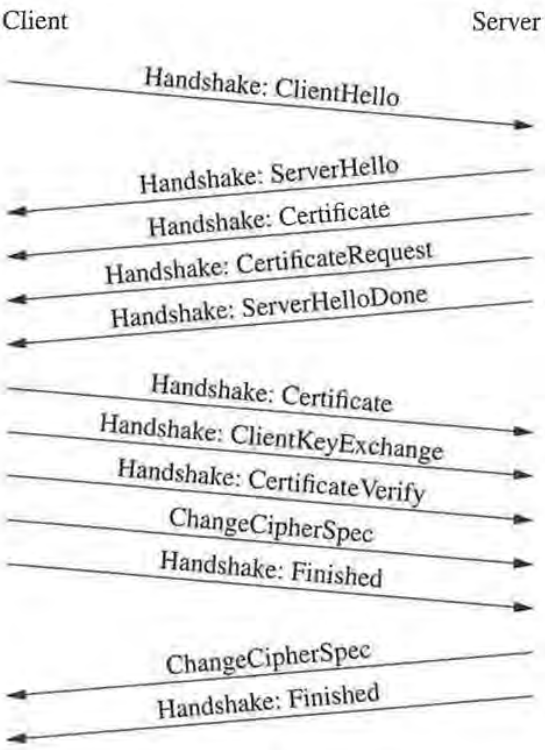


Figure 4.2 Client authentication timeline

communication between an exportable client and a domestic server with a permanent strong key. The way this works is that the server generates a temporary 512 bit key which it signs with its strong key. When communicating with domestic clients, it simply uses its strong key as discussed in Chapter 3.

The only difference in the handshake between ephemeral RSA mode and normal RSA mode is the presence of a single new message: ServerKeyExchange. The server uses the ServerKeyExchange message to transmit the signed RSA key. When the client receives the ServerKeyExchange message, it verifies the server's signature over the ephemeral key and then uses it to wrap the pre_master_secret exactly as described in Chapter 3. Figure 4.3 shows the timeline for this process.

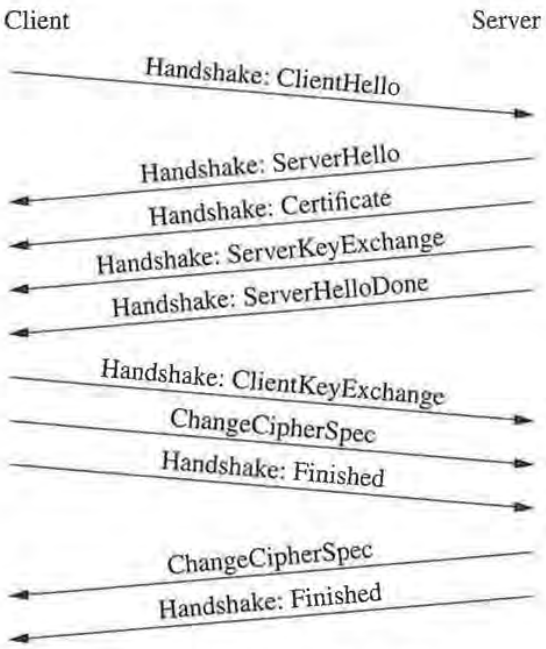


Figure 4.3 Ephemeral RSA timeline

4.5 Rehandshake

Once an SSL connection has been established, it's possible to perform a *rehandshake*. A rehandshake is simply a new SSL handshake performed over the current protected connection. Thus, the handshake messages are encrypted in transit. Once the new handshake has finished, data will be protected using the new session state.

```
struct { } HelloRequest;
```

The client can initiate a new handshake simply by sending a new ClientHello message. If the server wishes to initiate a rehandshake, it must send the empty HelloRequest

handshake message. The client responds to a HelloRequest with a ClientHello, after which the handshake proceeds normally. There's no requirement that either client or server agree to rehandshake. An implementation which does not wish to rehandshake can simply ignore the message or send a no_renegotiation alert. The next section shows an example of a client-initiated rehandshake. Chapter 8 shows an example of a server-initated rehandshake.

4.6 Server Gated Cryptography

Even before they were liberalized, the United States export regulations provided an exception allowing for the use of strong cryptography for certain financial transactions. In order to accommodate this, many SSL implementations include a feature that allows otherwise weak clients to detect that they're talking to a special server and upgrade to strong cryptography. This feature is variously referred to as *Server Gated Cryptography* (SGC) by Microsoft and *Step-Up* by Netscape. Actually, SGC and Step-Up are different variations of SSL. However, the difference isn't relevant at the moment and we'll simply refer to them collectively as SGC for the rest of this section.

SGC relies on the server having a special certificate that indicates that it is allowed to engage in strong cryptography with exportable clients. This certificate is tagged in two ways. First, it must be issued by one of a very short list of trustworthy CAs. Currently, Verisign and Thawte are the only permitted issuers. In the non-SGC situation, CAs merely need to be trusted by the user. However, in this case the CAs need to be trusted by the U.S. government not to issue SGC-capable certificates to non-financial servers. Second, the certificate contains an extension indicating that the certificate-holder is SGC-capable.

Note that this means that SGC-capable clients must actually contain all the code to do strong cryptography. Whether it is used or not actually depends on a simple test in the code. Actually, because the key reduction stage happens in the handshake, even export-only clients must contain strong versions of the symmetric encryption algorithms. In practice, then, browsers contain all the strong crypto. In Netscape, a compiled-in table controls which algorithms are available. In 1998 a program called *Fortify* was released, which takes advantage of this fact to turn on the strong cryptography in an exportable Netscape by patching this table in the binary.

SGC works by using the rehandshake feature of SSL. In the first handshake, the client offers only weak ciphers. Once the client has seen the server's certificate and can verify that SGC is appropriate, it initiates a second handshake, offering strong ciphers in its ClientHello. The messages for the second handshake are transmitted over the current protected session. Figure 4.4 shows a timeline of a Netscape client doing Step-Up.

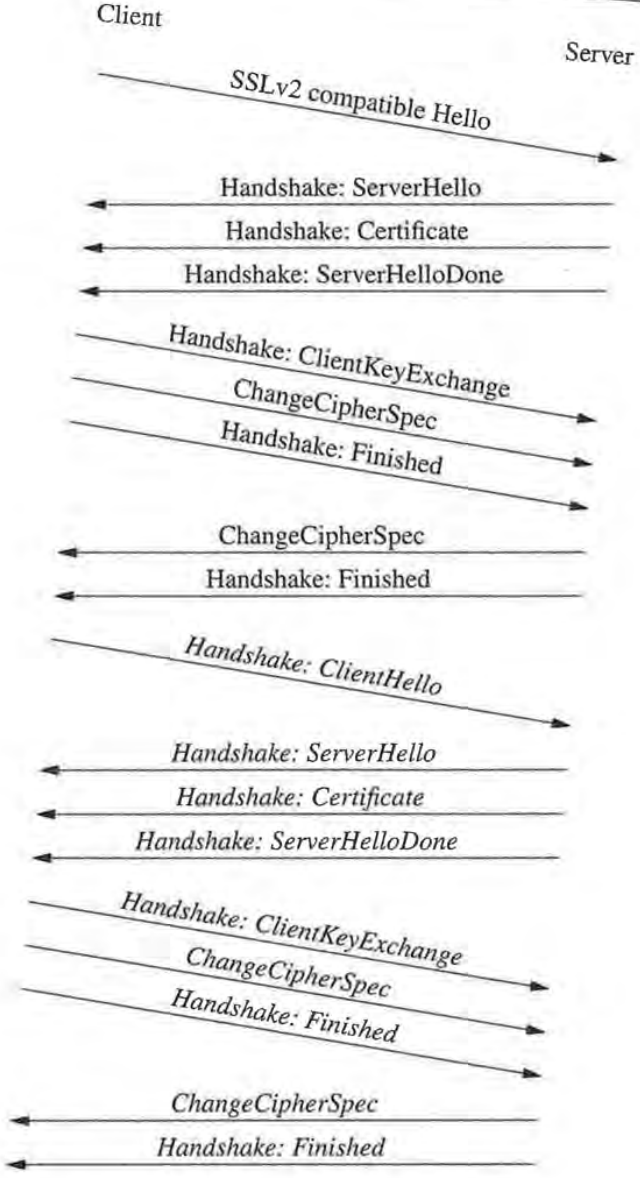


Figure 4.4 Step-Up timeline

4.7 DSS and DH

Although RSA is by far the dominant public key algorithm supported by SSLv2 and SSLv3 implementations, SSLv3 supports a number of cipher suites based on other algorithms, notably Digital Signature Standard (DSS) and Diffie-Hellman (DH). As discussed in Chapter 2, support for these ciphers was made mandatory in TLS. The main idea here was to avoid patented algorithms—in particular RSA. The patent on Diffie-Hellman expired in 1997. The patent situation of DSS is unclear in that RSA Data Security has claimed that the Schnorr patent (U.S. Patent 4995082) [Schnorr1991] covers DSS, but the U.S. National Institute of Standards and Technology claims otherwise. With all that in mind, the IETF decided to go with the devil it *didn't* know and mandate DH/DSS. Since the RSA patent has since expired, this motivation is no longer compelling, but support for DH/DSS lingers on.

Unlike RSA, which can be used for either key exchange or signature, DH can only be used for key agreement and DSS can only be used for digital signature. As a consequence, in order to get a complete solution, DH and DSS are typically used together. In such a system, the certificates are also usually signed with DSS.

There are two primary ways to use DH and DSS in SSL. The most common is to use *ephemeral DH* keys, which are analogous to ephemeral RSA keys. In this case, however, the server generates a temporary DH key and signs it with its DSS key, transmitting the signed key in the ServerKeyExchange message. The client then uses that DH key for key agreement. It is also possible to have a long-term DH key. In this case, the server will have a certificate (signed with DSS) containing its DH key. The client then uses the key in the certificate for key agreement.

The process of key agreement is the same no matter how the client gets the server's DH key. The client needs a DH key in the same group. Normally it won't have one, in which case it will have to generate one. It then transmits the DH key in the ClientKeyExchange message. At this point, the client and the server each have their own private key and the other side's public key, so they can independently compute the DH shared secret. This shared secret is used as the pre_master_secret. From there, the rest of the connection proceeds exactly as with RSA. Figure 4.5 shows the timeline for a handshake using ephemeral DH/DSS.

Note that the cipher suite numbers for DH/DSS are different from those for RSA, even if the symmetric algorithms are identical. Furthermore, the numbers for static DH/DSS are different from those for ephemeral DH/DSS. This highlights an unfortunate consequence of specifying all the algorithms together as a single cipher suite: there is no guarantee that symmetric algorithm support is orthogonal to public key algorithm support. For instance, RFC 2246 specifies five symmetric ciphers that can be used with RSA key exchange (DES, 3DES, RC2, RC4 and IDEA) but only two (DES and 3DES) for use with DH/DSS.

This situation is partially a result of resistance on the part of the working group to using proprietary algorithms such as RC4 and IDEA and partly a result of simple omission. Because positive action had to be taken to provide a full set of symmetric ciphers for DH/DSS, the WG only bothered to standardize the ciphers for which there was significant support and minimal resistance.

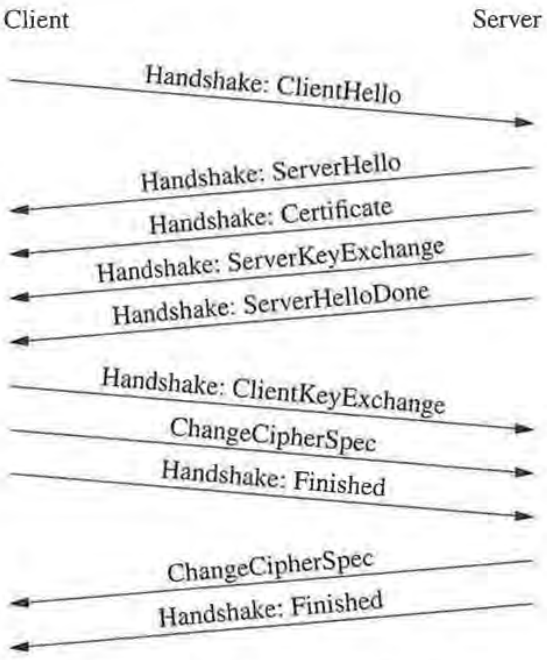


Figure 4.5 DSS/ephemeral DH timeline

4.8 Elliptic Curve Cipher Suites

Recently there has been a fair amount of interest in Elliptic Curve (EC) ciphers. To grossly oversimplify, EC ciphers replace the prime integer field of DH and DSS with a field composed of points on an elliptic curve. The discrete logarithm problem may be harder in such a group, so keys can be shorter and the public key operations comparatively faster. This is especially useful for smartcards where both memory and processing power are extremely limited, but even on fast servers the cost of the modular exponentiations in conventional public key crypto systems is quite high. The idea is that an EC system can be dropped in as a replacement for DH and DSS.

As of this writing, an Internet-Draft has been published for adding several EC-based cipher suites, but no such cipher suites have been standardized. One of the primary obstacles to standardizing such ciphers is that the intellectual property situation is rather vague. At the very least, Certicom and Apple both hold important patents on efficient implementation of EC ciphers, and there is some concern that the ciphers themselves may be patent encumbered.

4.9 Kerberos

Kerberos [Miller1987] is a popular symmetric-key-based authentication system developed at MIT. The general idea is that there is a central server (called a *Ticket Granting Server* (TGS)) that is trusted by every entity in the system. Each entity shares a symmetric key with the TGS. When a client wishes to communicate with a server, it requests a *ticket* from the TGS. The ticket is encrypted using the target's shared key and contains authenticating information for the requestor and a session key to be used for further communication between the two entities. The requestor then sends the ticket to the target.

RFC 2712 [Medvinsky1999] describes how to use Kerberos with TLS. The ClientKeyExchange contains both the ticket and an encrypted pre_master_secret. The shared key found in the ticket is used to encrypt the pre_master_secret. The server extracts the shared key from the ticket and decrypts the pre_master_secret. From that point on the handshake proceeds as usual. The Kerberos cipher suites have not seen wide deployment. Figure 4.6 shows the Kerberos cipher suites defined in RFC 2712.

Cipher Suite	Authentication	Key Exchange	Encryption	Digest	Number
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x001e
TLS_KRB5_WITH_3DES_EDE_CBC_SHA	Kerberos	Kerberos	3DES_EDE_CBC	SHA	0x001f
TLS_KRB5_WITH_RC4_128_SHA	Kerberos	Kerberos	RC4_128	SHA	0x0020
TLS_KRB5_WITH_IDEA_CBC_SHA	Kerberos	Kerberos	IDEA_CBC	SHA	0x0021
TLS_KRB5_WITH_3DES_EDE_CBC_MD5	Kerberos	Kerberos	3DES_EDE_CBC	MD5	0x0022
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x0023
TLS_KRB5_WITH_RC4_128_MD5	Kerberos	Kerberos	RC4_128	MD5	0x0024
TLS_KRB5_WITH_IDEA_CBC_MD5	Kerberos	Kerberos	IDEA_CBC	MD5	0x0025
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA	Kerberos	Kerberos	DES_40_CBC	SHA	0x0026
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA	Kerberos	Kerberos	RC2_40_CBC	SHA	0x0027
TLS_KRB5_EXPORT_WITH_RC4_40_SHA	Kerberos	Kerberos	RC4_40	SHA	0x0028
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5	Kerberos	Kerberos	DES_40_CBC	MD5	0x0029
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5	Kerberos	Kerberos	RC2_40_CBC	MD5	0x002A
TLS_KRB5_EXPORT_WITH_RC4_40_MD5	Kerberos	Kerberos	RC4_40	MD5	0x002B

Figure 4.6 The Kerberos cipher suites

4.10 FORTEZZA

The FORTEZZA card is a U.S. government–designed cryptographic token in a PCMCIA form factor. It uses DSA and SHA for signature and message digests but NSA-designed key agreement and encryption algorithms. The key agreement algorithm is a variant of Diffie-Hellman called *Key Exchange Algorithm* (KEA) and the encryption algorithm is a block cipher called SKIPJACK.

FORTEZZA was originally designed by the NSA to provide strong cryptogaphy while allowing the NSA to intercept communications. The conflict between these goals was resolved by incorporating a *key escrow* feature into the device. Each card had its own key, which was escrowed with the NSA. Thus, when the NSA wished to decrypt a communication, it could recover the key for the card used to encrypt it, but the encryption would be secure against all other attackers.

The FORTEZZA cipher suites require that the client be authenticated. This can happen in one of two ways. The client can either have a KEA certificate or it can have a DSA certificate and use its DSA key to sign a KEA share. In either case, the client will have a certificate to the server. Figure 4.7 shows a FORTEZZA handshake.

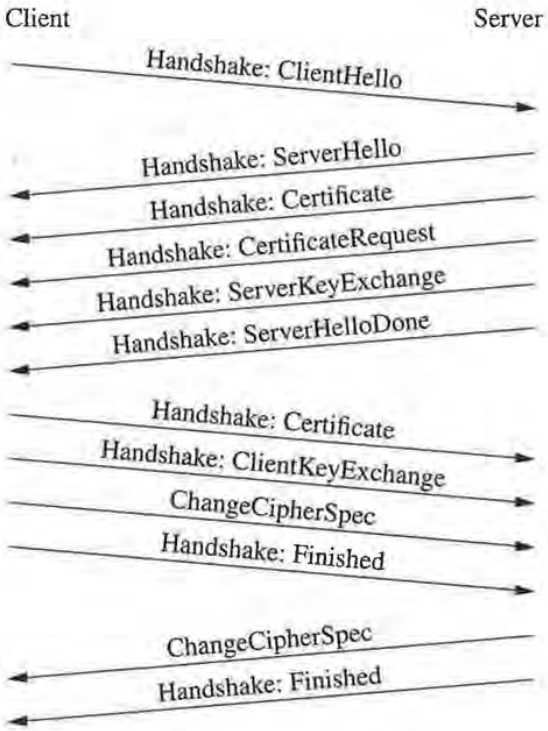


Figure 4.7 FORTEZZA handshake

Even though the server's KEA key is in its certificate, the server still sends a ServerKeyExchange message. This is used to transmit a random value which is used as part of the KEA key agreement process. Also, because the client must be authenticated, the server sends a CertificateRequest message. However, since client authentication happens via key agreement, no CertificateVerify is sent.

When FORTEZZA cards were originally released, SKIPJACK and KEA were classified. FORTEZZA support was specified in SSLv3 but removed for TLS because IETF won't standardize algorithms that aren't fully specified. Netscape did deliver FORTEZZA-capable versions of Navigator and Netscape Server, but they never saw wide use outside the U.S. government. Nevertheless, using FORTEZZA with SSL presents some interesting technical challenges so we'll discuss the details in Section 4.17.

4.11 The Story So Far

As we've seen, in addition to the basic RSA server-only authentication mode, SSL supports a number of other modes with special properties. All of these modes require a different handshake than that required for server-only RSA. The next half of the chapter covers these modes in detail, showing the contents of each new message and how they work together to provide new handshake functionality. The important thing to remember through all this is that, with the exception of FORTEZZA, these variations are simply changes to the handshake. Key derivation from the master_secret and data protection at the record layer are identical no matter which handshake variant you use.

4.12 Session Resumption Details

Session resumption is accomplished using the session_id field of the ClientHello and ServerHello messages. On the initial handshake between any client and server pair, the client's session_id field is absent because the client has no session cached. If the server is willing to allow resumption—it almost always is—it generates a session_id and returns it in the session_id field of the ServerHello. When the handshake is finished, both the client and the server store the session information (the keying material and negotiated ciphers) somewhere that they can get to them later if the session is resumed.

When a client wishes to resume a session, it uses the appropriate session_id in the ClientHello. If the server is willing to resume the session, it responds with the same session_id in the ServerHello. The server then skips the rest of the handshake and immediately sends a ChangeCipherSpec message. The key_block is recomputed using the stored master_secret and the new client and server random values, so all the keys are different.

Figure 4.8 shows the detailed handshake messages for a resumed TLS session. Note that the session_id field in the ClientHello matches the session_id in the ServerHello. If the server did not want to resume the session, it would simply generate a new session_id and continue with the handshake as if the client had not attempted to resume. Typically this happens because the server no longer remembers the session that the client wants to resume, either because it maintains a limited size cache or because the session has timed out.

When resuming a session, the client is required to offer at least the cipher suite that was originally negotiated. If the server agrees to resume, it must choose that cipher suite. In this case, the client has offered a full set of cipher suites. If the server were to choose not to resume the session, it could choose any of the offered cipher suites.

```
New TCP connection: romeo(1300) <-> speedy(4433)
1 0.0022 (0.0022) C>S   Handshake
      ClientHello
        resume [32]=
          6d f3 02 8a 44 a7 42 94 14 7b 59 ad
          8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
          50 fa 4f 9f 27 cf 45 c4
        cipher suites
                  TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
                  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
                  TLS_RSA_WITH_3DES_EDE_CBC_SHA
                  TLS_RSA_WITH_IDEA_CBC_SHA
                  TLS_RSA_WITH_RC4_128_SHA
                  TLS_RSA_WITH_RC4_128_MD5
                  TLS_DHE_RSA_WITH_DES_CBC_SHA
                  TLS_DHE_DSS_WITH_DES_CBC_SHA
                  TLS_RSA_WITH_DES_CBC_SHA
                  TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
                  TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
                  TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
                  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
                  TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compression methods
                  NULL
2 0.0288 (0.0266) S>C   Handshake
      ServerHello
        session_id[32]=
          6d f3 02 8a 44 a7 42 94 14 7b 59 ad
          8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
          50 fa 4f 9f 27 cf 45 c4
        cipherSuite             TLS_RSA_WITH_RC4_128_SHA
        compressionMethod               NULL
3 0.0288 (0.0000) S>C   ChangeCipherSpec
4 0.0288 (0.0000) S>C   Handshake: Finished
5 0.0293 (0.0005) C>S   ChangeCipherSpec
6 0.0293 (0.0000) C>S   Handshake: Finished
```

Figure 4.8 Resumed TLS session handshake messages

Session ID Lookup

Servers and clients use different information to retrieve sessions. When a client initiates an SSL connection, the only available information is the server's hostname (or IP address if no hostname was used) and port, so these must be used to look up the session ID. However, when a server receives a request for a resumed session for the client, the session_id is available and can be used to look up the session. Since the lookup is based on the session_id, session resumption will work even when the client is connecting from a different IP address and port than the one used for the initial handshake.

Because initiating TCP ports are sequentially chosen, the client's port will almost always be different from the port used for the original connection. Usually the client's IP address will be the same for every connection. However, because many clients have their IP addresses dynamically assigned via DHCP or similar mechanisms it is quite possible for a client to attempt to resume a session using a different IP address than the one it used to create it. Servers should support this.

Although permitting resumption from a client with a different IP address may seem dangerous, it is actually secure: first, it's trivial for an attacker to forge messages from a given IP address, so checking the IP address offers little value. More important, the server checks the client's Finished message at the end of the handshake. In order to correctly generate the Finished message, the client must have access not only to the new encryption and MAC keys (in order to encrypt and MAC the message) but to the master_secret in order to compute the Finished hashes. Since an attacker with access to the master_secret could easily take over an existing connection, this is equally secure. Note that to reduce the effect of compromise of any given master_secret, servers typically expire sessions after some period of time (a day is common.)

4.13 Client Authentication Details

As we've seen, SSL client authentication uses two new messages that we haven't seen before, as well as a new usage for an old message. The first new message is the CertificateRequest message which is used by the server to request client authentication. The client responds by sending the Certificate message (which we previously saw being used by the server) to send its certificate and the CertificateVerify message to authenticate itself.

CertificateRequest

```
struct {
   ClientCertificateType certificate_types<1..2^8-1>;
   DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

```
enum {
  rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
  (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

CertificateRequest
    certificate_types           rsa_sign
    certificate_types           dss_sign
  certificate_authority
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Server test cert (512 bit)
  certificate_authority
    C=US
    O=AT&T Bell Laboratories
    OU=Prototype Research CA
  certificate_authority
    C=US
    O=RSA Data Security, Inc.
    OU=Commercial Certification Authority
  certificate_authority
    C=US
    O=RSA Data Security, Inc.
    OU=Secure Server Certification Authority
  certificate_authority
    C=ZA
    ST=Western Cape
    L=Cape Town
    O=Thawte Consulting cc
    OU=Certification Services Division
    CN=Thawte Server CA
    Email=server-certs@thawte.com
  certificate_authority
    C=ZA
    ST=Western Cape
    L=Cape Town
    O=Thawte Consulting cc
    OU=Certification Services Division
    CN=Thawte Premium Server CA
    Email=premium-server@thawte.com
```

The CertificateRequest message indicates to the client that the client should perform client authentication and provides guidance as to what sorts of certificates the server is willing to accept. The server is allowed to specify both the cryptographic algorithm that should be used and a list of the certificate authorities which it trusts to sign client certificates.

The certificate_types field is an array of single byte values, each of which describes a permitted signature algorithm. In this case, the server has specified rsa_sign and dss_sign, indicating that it is prepared to accept a response signed with either RSA or DSS. Note that this value also specifies the algorithm which was used to sign the certificate.

The other two possible values, rsa_fixed_dh and dss_fixed_dh actually do not involve a signature at all. Rather, they indicate that the client and server will use fixed Diffie-Hellman key pairs to agree on the pre_master_secret. If these methods are chosen, the client does not send a CertificateVerify because the connection is implicitly authenticated by the ability to generate the appropriate keys from the pre_master_secret. However, see Section 4.16 for an explanation of fixed DH.

The certificate_authorities field specifies a list of certificate authorities (CAs) from which the server is willing to accept certificates. These CAs are specified by using a BER encoded version of the CA's distinguished name. See Chapter 1 for more information on distinguished names and certificates. It is important to note that if certificate chains are being used, then the CA name specified in the CertificateRequest message need not refer to the CA that signed the client's certificate, but may instead refer to one of the parent CAs.

Certificate

```
struct {
  ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;

opaque ASN.1Cert<2^24-1>;
```

We saw the Certificate message before when the server used it to provide its certificate to the client. Here it is used by the client to send its certificate for client authentication. This message has exactly the same structure as the server's Certificate message, although the certificates themselves are different.

CertificateVerify

```
select (SignatureAlgorithm) {
  case anonymous: struct { };
```

```
    case rsa:
      digitally-signed struct {
        opaque md5_hash[16];
        opaque sha_hash[20];
      };
    case dsa:
      digitally-signed struct {
        opaque sha_hash[20];
      };
} Signature;

struct {
   Signature signature;
} CertificateVerify;
```

```
CertificateVerify
  Signature[64]=
    5a a7 6a 48 54 17 c9 a3 09 87 90 37
    42 17 45 c7 bf de 24 0f 4d 73 9b a8
    69 a6 c7 f3 a0 41 c5 26 fc 6a 48 0d
    e1 02 4f 9d 1b d2 17 55 dc 4e 55 11
    6f 89 7c ed 68 59 70 db b4 e7 fe 11
    d7 2f f6 83
```

The CertificateVerify does all the actual work of client authentication. It contains a digest of the handshake messages signed by the client's private key (the one that corresponds to the certificate in the client's Certificatemessage). This is what the digitally-signed operator means. Because only the holder of the private key corresponding to the certificate could sign this message, the server can conclude that the entity named in the certificate is connecting.

In SSLv3, we compute the hashes of the handshake messages as shown:

```
CertificateVerify.signature.md5_hash =
  MD5(master_secret + pad2 +
    MD5(handshake_messages + master_secret +
      pad1));

CertificateVerify.signature.sha_hash =
  SHA-1(master_secret + pad2 +
    SHA-1(handshake_messages + master_secret +
      pad1));
```

Including the master_secret in the messages actually doesn't add much security value because it's implied by the other handshake messages. TLS uses a simple digest of the handshake messages.

```
CertificateVerify.signature.md5_hash = MD5(handshake_messages);
CertificateVerify.signature.sha_hash = SHA-1(handshake_messages);
```

The details of the signature computation depend on which signature algorithm is being used. In this handshake, the client is signing using RSA, so the signed object is the concatenation of MD5 and SHA-1 digests of the handshake messages up to this point in the handshake. Because DSS requires a 160-bit input, when DSS is being used, only the SHA-1 digest of the messages is used. Because the digest covers the server's random value, the server can be sure that this signature is *fresh* (that is, not replayed) as long as it used a fresh random value in its ServerHello.

It is important to note that this form of RSA signature is different from the usual RSA signature described in [RSA1993b], which only includes one digest. In PKCS #1, the signature is typically computed over a DER encoded DigestInfo structure which also contains the algorithm identifier for the digest. Because the MD5 and SHA-1 digests are mandated as part of the SSL protocol, the digest algorithm identifier is unnecessary (and because there is no identifier for the MD5 and SHA-1 combination, it is also problematic). As a result, the concatenated hash value is signed directly without being first encoded in a DigestInfo. This can cause problems with hardware implementations or all-in-one PKI toolkits, which often want to perform the DigestInfo encoding as part of the signature.

Failure Conditions

If the client has no suitable certificate, a Certificate message containing no certificates should be sent. At this point, the server can either proceed with the connection, using some other means of authentication (for instance, prompting for a password in the application layer) or no authentication at all. If the server is unwilling to continue, it sends a fatal handshake_failure alert.

4.14 Ephemeral RSA Details

```
struct {
  select (KeyExchangeAlgorithm) {
    case diffie_hellman:
      ServerDHParams params;
      Signature signed_params;
    case rsa:
      ServerRSAParams params;
      Signature signed_params;
  };
} ServerKeyExchange;
```

```
struct {
   opaque RSA_modulus<1..2^16-1>;
   opaque RSA_exponent<1..2^16-1>;
} ServerRSAParams;

select (SignatureAlgorithm) {
   case anonymous: struct { };
   case rsa:
      digitally-signed struct {
         opaque md5_hash[16];
         opaque sha_hash[20];
      };
   case dsa:
      digitally-signed struct {
         opaque sha_hash[20];
      };
} Signature;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;
enum { anonymous, rsa, dsa } SignatureAlgorithm;
```

```
ServerKeyExchange
  params
    RSA_modulus[64] =
      c9 61 5e 58 08 6a 5c f0 76 a4 5d 6e
      ae 99 51 f9 ae c6 2c 8a e5 34 1e 9d
      a6 cb 68 b4 d6 47 b2 01 d4 5d 9b 32
      3c 49 2a 90 ac c2 41 cd b4 20 4f 54
      c2 a0 26 e4 b8 ee 69 61 04 23 64 72
      f2 40 e3 9f
    RSA_exponent[3] =
      01 00 01
  signature[128] =
    09 6d 34 ee 8c 2f 14 e5 52 05 26 25
    4c 34 72 35 ef 79 1f 07 c1 71 82 cd
    3d f6 7c 67 05 ff eb 02 bc da bf 8e
    3a e7 c8 37 14 5c ca 05 4a 69 5f 09
    f3 b0 fa 53 33 50 ab 41 e8 63 6b f5
    5a 5d c1 5d f0 09 6d aa bd d8 0b e2
    bb 13 0d e6 09 f8 91 11 b1 ec 25 58
    1f 7e b3 56 81 4e c0 03 71 a4 95 e4
    be 1f 2e 10 75 80 75 1a 65 ed c4 70
    09 0b 19 64 3f dd f0 42 4a 32 73 36
    82 b1 2b 24 2c a4 6b 3a
```

The ServerKeyExchange message contains a signed version of the server's ephemeral public key. Because this message is also used for ephemeral DH, the specification permits both RSA and DH keys. We'll see an example of ephemeral DH in Section 4.16. The message is signed using the server's long-term key, in this case an RSA key. The permitted algorithms are RSA signature, DSS, or anonymous, which simply means that the message is unsigned. Note that the anonymous mode offers no security at all against active attack. The attacker can trivially pose as the server to the client using a man-in-the-middle attack.

Although the specification permits cipher suites that mix-and-match key exchange and signature algorithms, the common combinations are RSA signature with RSA key exchange (shown at the beginning of this section) and DSS signature with DH key exchange (TLS implementations must implement DH/DSS). Cipher suites are defined for RSA signature with DH key exchange, but they are not commonly used. No cipher suites are even defined for DSS signature with RSA key exchange.

The RSA key is presented in the message as a pair of byte strings representing the RSA modulus and exponent. Following Internet convention, the large integers are mapped to byte strings in "network" byte order, with the first byte representing the most significant byte of the integer. The size of the RSA modulus is the important number, and it's 64 bytes long, consistent with a 512-bit key. As we saw in Chapter 1, standard practice is to use a very small public exponent for performance reasons, and here 65537 has been chosen. See Chapter 1 for more information on RSA keys.

The RSA signature shown in the message is computed using a pair of hashes as in the CertificateVerify message, but the input data to the hashes are different. The ServerKeyExchange hash incorporates only the parameters to be signed and the client and server random values. The random values serve to prevent replay of a previous ServerKeyExchange message in a new handshake:

```
md5_hash =
  MD5(client_random + server_random + ServerParams);

sha_hash =
  SHA(client_random + server_random + ServerParams);
```

Because generating RSA keys is computationally expensive, SSL server implementations typically use the same ephemeral key to service a large number of clients. This isn't as secure as generating a fresh key for each client, because a large number of transactions (often as much as a day's worth) are protected under the same 512-bit key. However, the cost of generating a fresh key for each transaction would almost always be prohibitive.

There are two conditions under which ephemeral RSA can be used: first, the cipher suite used must be one of the export cipher suites (i.e., the key exchange algorithm is RSA_EXPORT). Second, the signing key must be longer than 512 bits. Otherwise, normal RSA must be used.

Before we get too excited about ephemeral RSA, it's worthwhile to do a back-of-the envelope computation as to how much security it adds. Typical key lifetimes are on the order of a year or two and servers usually generate a new ephemeral RSA key at most

Ex.1024
Askeladden, L.L.C. / Page 184 of 788

once a day (though on a modern CPU it's possible to do it far more frequently, say every few minutes). Often, they'll generate it at server startup only, and servers can stay up for weeks. So, if you wanted to read all the traffic to a given server, ephemeral RSA would increase your workload by less than a thousand times given typical use patterns. On the other hand, if you want to read a specific message, and you know when that message was sent, ephemeral RSA doesn't make your job any more difficult at all.

4.15 SGC Details

The difficulty faced by SGC/Step-Up is that the client must send its ClientHello before it knows whether the server is a normal SSL server or an SGC/Step-Up capable server. Thus, it must offer only export cipher suites. Otherwise, a normal server might select a strong cipher suite that the client couldn't use. Only when the client receives the server's Certificate message can it detect that the server is SGC/Step-Up capable. At this point, SGC and Step-Up differ. We'll discuss Step-Up first.

Step-Up

The server signals to the client that it's willing to do Step-Up by providing a special value in the extendedKeyUsage extension. When the client sees this value, it knows that it can Step-Up. The browser completes the SSL handshake that it's currently doing (which negotiates an export cipher suite) and then immediately rehandshakes to negotiate a strong cipher suite. Figure 4.9 shows a Step-Up connection in action.

```
New TCP connection: romeo(4290) <-> romeo(443)
1 0.0004 (0.0004) C>S SSLv2 compatible client hello
  Version 3.0
  cipher suites
      SSL2_CK_RC4_EXPORT40
      SSL2_CK_RC2_EXPORT40
      TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
      TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
      TLS_RSA_EXPORT_WITH_RC4_40_MD5
      TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
2 0.0058 (0.0053) S>C   Handshake
      ServerHello
        session_id[32]=
          5e ae 37 80 c5 30 ff 35 59 65 2a 8c
          9b a1 8a 73 ad 2d 88 30 59 9a f9 58
          45 ac 8d a3 85 5f ca e9
        cipherSuite           TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        compressionMethod             NULL
```

```
3 0.0058 (0.0000) S>C   Handshake
      Certificate
        Subject
          C=US
          O=RTFM, Inc.
          CN=romeo.rtfm.com
        Issuer
          C=US
          ST=California
          O=RTFM, Inc.
          CN=Step-Up CA
        Serial         03
        Extensions
          Extension: X509v3 Subject Alternative Name
            <EMPTY>
          Extension: X509v3 Basic Constraints
            CA:FALSE, pathlen:0
          Extension: Netscape Comment
            mod_ssl generated custom server certificate
          Extension: Netscape Cert Type
            SSL Server
          Extension: X509v3 Extended Key Usage
            Netscape Server Gated Crypto
4 0.0058 (0.0000) S>C   Handshake
      ServerHelloDone
5 0.0248 (0.0189) C>S   Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[128]=
          a1 40 7c 99 2f 40 4d 01 dd b0 0a 7b
          f8 8e ee e3 1d f1 ed 35 04 ea 56 5f
          1a 3f 62 75 cf 6e bc b9 58 cf ad 33
          ba be 30 3c 63 d0 85 ea a7 a4 24 e2
          b5 dd d1 21 03 e1 87 bd cb cf 56 54
          8d ed 02 f9 67 d7 bc 9a 73 cb 51 a6
          c2 d6 c2 12 b6 96 06 45 db a3 ed d8
          40 0c ea 11 22 09 61 7c 98 85 8f ed
          ca 4a 51 52 bd e9 14 0a b3 5d 04 be
          58 79 1e 51 cd fd dc ae 66 23 b8 4f
          1b ea 68 10 eb 8a e7 87
6 0.2199 (0.1951) C>S   ChangeCipherSpec
7 0.2199 (0.0000) C>S   Handshake
      Finished
8 0.2223 (0.0023) S>C   ChangeCipherSpec
9 0.2223 (0.0000) S>C   Handshake
      Finished
First handshake completed.
```

Client initiates a new handshake offering strong cipher suites.

```
10 0.2231 (0.0008) C>S   Handshake
      ClientHello
        Version 3.0
        cipher suites
                  TLS_RSA_WITH_RC4_128_MD5
                  value unknown: 0xffe0 Netscape proprietary cipher suite
                  TLS_RSA_WITH_3DES_EDE_CBC_SHA
                  TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
                  TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
                  TLS_RSA_EXPORT_WITH_RC4_40_MD5
                  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        compression methods
                  NULL
11 0.2271 (0.0039) S>C   Handshake
      ServerHello
        session_id[32]=
          de 59 fc ef 7f b5 59 e0 92 f9 31 20
          da a3 f1 82 bf 78 ba 53 8b b6 8a a9
          d6 69 82 af 55 8b 99 27
        cipherSuite           TLS_RSA_WITH_RC4_128_MD5
        compressionMethod           NULL
12 0.2271 (0.0000) S>C   Handshake
      Certificate
        Subject
          C=US
          O=RTFM, Inc.
          CN=romeo.rtfm.com
        Issuer
          C=US
          ST=California
          O=RTFM, Inc.
          CN=Step-Up CA
        Serial         03
        Extensions
          Extension: X509v3 Subject Alternative Name
            <EMPTY>
          Extension: X509v3 Basic Constraints
            CA:FALSE, pathlen:0
          Extension: Netscape Comment
            mod_ssl generated custom server certificate
          Extension: Netscape Cert Type
            SSL Server
          Extension: X509v3 Extended Key Usage
            Netscape Server Gated Crypto
```

(continued)

Figure 4.9 *(continued)*

```
13 0.2271 (0.0000) S>C   Handshake
      ServerHelloDone
14 0.2381 (0.0110) C>S   Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[128]=
           98 29 0c 5f 72 b0 46 03 fd 3d ed 62
           c6 fc ec d3 e3 73 d3 f5 c8 a0 60 0a
           f7 94 de 18 0a 9c 3c db db 7c f4 9c
           34 65 90 dd 11 fd a2 d3 ae 50 f1 d6
           e8 22 79 72 fe 99 b8 e1 03 a4 4a 64
           22 4d e8 a5 58 d5 80 22 d5 da d4 dc
           fc 42 a0 f3 f3 50 a6 56 4f f4 ae 69
           37 7b 9c 8c 53 c8 d7 7a ac 91 2d 7b
           48 8b 10 e5 b0 55 65 45 99 3e ca 69
           f7 c5 9e ae 0b fe f1 36 7b 4c 1c 6d
           7e 4d 42 96 6b 21 55 d4
15 0.4199 (0.1818) C>S   ChangeCipherSpec
16 0.4199 (0.0000) C>S   Handshake
      Finished
17 0.4215 (0.0015) S>C   ChangeCipherSpec
18 0.4215 (0.0000) S>C   Handshake
      Finished
```

Figure 4.9 A connection using Step-Up

Figure 4.9 shows a Step-Up connection between a Netscape client and an Apache server with mod_ssl. Note that in the initial message, the client sends an SSLv2-compatible hello offering only export cipher suites. It must do this on the initial connection because it doesn't yet know that the server is Step-Up capable. On future connections, of course, it could offer strong ciphers immediately if it remembered that the server had a Step-Up certificate.

The server's certificate, shown in record 3, contains the Step-Up extension (shown here as `Netscape Server Gated Crypto`), so the client knows it can do Step-Up. However, the client and the server have already agreed to a set of cipher suites, so this information isn't immediately useful because there's no way in SSL for the client to change its mind at this point. The client could drop the connection and reconnect, but that's not the way that Step-Up works. Instead, the client completes the handshake.

Once the handshake is completed, the client initiates a rehandshake over the newly established encrypted channel. The rest of Figure 4.9 shows that rehandshake. In record 10 the client sends a new ClientHello. This time it offers strong cipher suites as well as

weak ones and naturally the server accepts one. From there, the handshake completes as usual.

Note that the client does not attempt to resume the session. Most of the export cipher suites use ephemeral RSA, in which case resuming is impossible, because we need to establish a new master_secret using the server's 1024-bit RSA key. However, the new (and still unstandardized) EXPORT1024 cipher suites use the long RSA key for key exchange, so this session could technically be resumed. However, the client doesn't offer this. As a consequence, an entirely new RSA key exchange must be performed. Thus, Step-Up handshakes are (at least) twice as expensive computationally as ordinary handshakes.

This trace also shows the SSL feature of *rehandshake*. At any time during a connection, SSL implementations can initiate a new handshake. The new handshake is carried over the currently protected channel. Thus, the second handshake is entirely encrypted. The reason we're able to read it is that `ssldump` has been given the server's private key and is automatically decrypting the data for us. All the encrypted records are rendered in a *`fixed-width italic`* font to show that we've decrypted them. At the end of the handshake, all traffic switches over to the newly negotiated connection.

Careful inspection of Figure 4.9 reveals that the server's certificate was signed by the author's local CA. However, we said earlier that Step-Up certificates must be signed by specific trusted CAs. Surely the author's CA is not one of them! We've taken advantage of a weakness in Netscape's implementation. The CAs are stored in a database and being able to issue Step-Up certificates is simply a database attribute. We've changed that attribute in our local database, thus allowing ourselves to issue Step-Up certificates. We could have just as well connected to a legitimate Step-Up server, but then we wouldn't have the server's private key and be able to decrypt the traffic.

SGC

In the previous section, we saw that Step-Up requires two key exchange stages. This is expensive and the first stage isn't really adding much security value. Microsoft noticed this and SGC used only one key exchange stage. Figure 4.10 shows an SGC connection between IE and Internet Information Server (IIS).

```
New TCP connection: swagger(1897) <-> 151(443)
1 0.3679 (0.3679) C>S SSLv2 compatible client hello
  Version 3.0
  cipher suites
      TLS_RSA_EXPORT_WITH_RC4_40_MD5
      TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
      TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
      SSL2_CK_RC4_EXPORT40
      SSL2_CK_RC2_EXPORT40
```

(continued)

Figure 4.10 *(continued)*

```
2 0.7508 (0.3829) S>C   Handshake
      ServerHello
        session_id[32]=
          22 00 00 00 4b 66 29 fe 02 a7 f3 cc
          56 38 a6 4e 41 ad d8 fd 55 08 75 5d
          fe e3 41 02 fe 4e 27 62
        cipherSuite           TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compressionMethod           NULL
      Certificate
        Subject
          C=IT
          ST=Milano
          L=Assago
          O=BANCO AMBROSIANO VENETO
          OU=INNOVAZIONE TECNOLOGICA
          OU=Terms of use at www.verisign.com
          RPA (c)99
          CN=HB.AMBRO.IT
        Issuer
          O=VeriSign Trust Network
          OU=VeriSign, Inc.
          OU=VeriSign International Server CA - Class 3
          OU=www.verisign.com
          CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
        Serial         69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
        ce ec 1b 59
        Extensions
          Extension: X509v3 Basic Constraints
            CA:FALSE
          Extension: 2.5.29.3
            value omitted
          Extension: Netscape Cert Type
            SSL Server
          Extension: X509v3 Extended Key Usage
            Netscape Server Gated Crypto, Microsoft Server Gated Crypto
          Extension: 2.16.840.1.113733.1.6.7
            value omitted
        Subject
          O=VeriSign Trust Network
          OU=VeriSign, Inc.
          OU=VeriSign International Server CA - Class 3
          OU=www.verisign.com
```

```
      CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
    Issuer
      C=US
      O=VeriSign, Inc.
      OU=Class 3 Public Primary Certification Authority
    Serial          23 6c 97 1e 2b c6 0d 0b f9 74 60 de
    f1 08 c3 c3
    Extensions
      Extension: X509v3 Basic Constraints
        CA:TRUE, pathlen:0
      Extension: X509v3 Key Usage
        Certificate Sign, CRL Sign
      Extension: Netscape Cert Type
        SSL CA, S/MIME CA
      Extension: X509v3 Extended Key Usage
        2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
      Extension: X509v3 Certificate Policies
        Policy: 2.16.840.1.113733.1.7.1.1
        CPS: https://www.verisign.com/CPS
        User Notice:
        Organization: VeriSign, Inc.
        Number: 1
        Explicit Text: VeriSign's Certification Practice Statement, ↵
        www.verisign.com/CPS, governs this certificate & is ↵
        incorporated by reference herein. SOME WARRANTIES DISCLAIMED ↵
        & LIABILITY LTD. (c)1997 VeriSign
  ServerKeyExchange
  ServerHelloDone
```

Client aborts first handshake and initiates a new one on the same connection, offering strong cipher suites.

```
3 0.7599 (0.0090) C>S   Handshake
    ClientHello
      Version 3.0
      cipher suites
              TLS_RSA_WITH_RC4_128_MD5
              TLS_RSA_WITH_RC4_128_SHA
              value unknown: 0x80   Microsoft proprietary cipher suite
              value unknown: 0x81   Microsoft proprietary cipher suite
              value unknown: 0x80   Microsoft proprietary cipher suite
              TLS_RSA_WITH_DES_CBC_SHA
              TLS_RSA_EXPORT_WITH_RC4_40_MD5
              TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
              TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
      compression methods
              NULL
```

(continued)

Figure 4.10 *(continued)*

```
4 1.1402 (0.3802) S>C   Handshake
      ServerHello
        session_id[32]=
          22 00 00 00 4b 66 29 fe 02 a7 f3 cc
          56 38 a6 4e 41 ad d8 fd 55 08 75 5d
          fe e3 41 02 fe 4e 27 62
        cipherSuite           TLS_RSA_WITH_RC4_128_MD5
        compressionMethod           NULL
      Certificate
        Subject
          C=IT
          ST=Milano
          L=Assago
          O=BANCO AMBROSIANO VENETO
          OU=INNOVAZIONE TECNOLOGICA
          OU=Terms of use at www.verisign.com
          RPA (c)99
          CN=HB.AMBRO.IT
        Issuer
          O=VeriSign Trust Network
          OU=VeriSign, Inc.
          OU=VeriSign International Server CA - Class 3
          OU=www.verisign.com
          CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
        Serial         69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
        ce ec 1b 59
        Extensions
          Extension: X509v3 Basic Constraints
            CA:FALSE
          Extension: 2.5.29.3
            value omitted
          Extension: Netscape Cert Type
            SSL Server
          Extension: X509v3 Extended Key Usage
            Netscape Server Gated Crypto, Microsoft Server Gated Crypto
          Extension: 2.16.840.1.113733.1.6.7
            value omitted
        Subject
          O=VeriSign Trust Network
          OU=VeriSign, Inc.
          OU=VeriSign International Server CA - Class 3
          OU=www.verisign.com
          CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
```

```
        Issuer
          C=US
          O=VeriSign, Inc.
          OU=Class 3 Public Primary Certification Authority
        Serial         23 6c 97 1e 2b c6 0d 0b f9 74 60 de
        f1 08 c3 c3
        Extensions
          Extension: X509v3 Basic Constraints
            CA:TRUE, pathlen:0
          Extension: X509v3 Key Usage
            Certificate Sign, CRL Sign
          Extension: Netscape Cert Type
            SSL CA, S/MIME CA
          Extension: X509v3 Extended Key Usage
            2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
          Extension: X509v3 Certificate Policies
            Policy: 2.16.840.1.113733.1.7.1.1
            CPS: https://www.verisign.com/CPS
            User Notice:
            Organization: VeriSign, Inc.
            Number: 1
            Explicit Text: VeriSign's Certification Practice Statement, ↵
            www.verisign.com/CPS, governs this certificate & is ↵
            incorporated by reference herein. SOME WARRANTIES DISCLAIMED ↵
            & LIABILITY LTD. (c)1997 VeriSign
      ServerHelloDone
5 1.1459 (0.0057) C>S   Handshake
      ClientKeyExchange
        EncryptedPreMasterSecret[128]=
          7d 4f fb ae fb 3e 5c 5b 9f 72 82 86
          04 c7 7e 5e 72 e0 81 d3 8a 1a a0 10
          97 36 d3 ab 0a 8b d8 e3 ba 83 1b 1c
          8e 82 73 34 0a 4e 74 ec 0c 57 08 f7
          ce 61 13 cf 8b c2 7b 5e 8a 14 29 28
          94 5f 8a ff 93 42 59 93 12 ee 6d d4
          d6 15 c2 90 c1 b1 df 2a 73 1e fb 11
          bc e1 83 81 cd c7 4e ec 2c 1c e2 ba
          14 ab fa 7c 80 b1 e5 8e 52 9b 55 bf
          b7 84 b3 60 98 67 58 29 cd 50 ab 4d
          2e 3f 21 09 4a 16 a0 57
6 1.1459 (0.0000) C>S   ChangeCipherSpec
7 1.1459 (0.0000) C>S   Handshake Finished
8 1.4361 (0.2901) S>C   ChangeCipherSpec
9 1.4361 (0.0000) S>C   Handshake Finished
```

Figure 4.10 A connection using SGC

The first two records in the handshake are more or less the same as those in Figure 4.10. The client offers export cipher suites and the server provides its certificate showing that it's SGC-capable. (Although note that IIS transmits all of the handshake messages in a single record rather than in separate records as OpenSSL does. This is a matter of taste. There's no real technical reason to do one or the other.) However, instead of completing the connection, IE immediately responds in record *3* with a new ClientHello offering strong cipher suites. At this point, the old handshake is aborted and an entirely new one takes place. Note that IE is also offering the proprietary cipher suites 0x0080 and 0x0081. These are from a precursor to SGC used for Microsoft Money [Banes2000].

This seems like a clever trick, and it is. Unfortunately, it also violates the SSL specification, which has no provision for sending ClientHellos in the middle of a handshake. Thus, to preserve interoperability, Microsoft had to use a different extension from Netscape to signal that not only was a server allowed to use Step-Up, but that it could also do SGC. Note that the certificate in Figure 4.10 contains both the Netscape and Microsoft SGC extensions. Thus, when talking to this server, Step-Up only clients will do Step-Up but SGC-capable clients will do SGC. OpenSSL only added support for SGC in OpenSSL 0.9.5.

4.16 DH/DSS Details

```
struct {
  opaque DH_p<1..2^16-1>;
  opaque DH_g<1..2^16-1>;
  opaque DH_Ys<1..2^16-1>;
} ServerDHParams;
```

```
ServerKeyExchange
  params
    DH_p[64]=
      da 58 3c 16 d9 85 22 89 d0 e4 af 75
      6f 4c ca 92 dd 4b e5 33 b8 04 fb 0f
      ed 94 ef 9c 8a 44 03 ed 57 46 50 d3
      69 99 db 29 d7 76 27 6b a2 d3 d4 12
      e2 18 f4 dd 1e 08 4c f6 d8 00 3e 7c
      47 74 e8 33
    DH_g[1]=
      02
    DH_Ys[64]=
      98 5a ee fc ce ac cf f1 05 cf 08 07
      63 18 dd 50 53 66 a5 b8 0b 88 4d 7e
      7d ea 11 3e 2a 99 63 e8 92 7a 56 cb
      f1 36 74 97 36 4a f0 3e 4e 29 3e a2
      e2 53 36 d8 9c a0 40 aa 8c fc eb c0
      93 b6 c3 e8
```

```
signature[47] =
  30 2d 02 14 78 bb 87 40 13 e4 8d e9
  73 16 4e 0c dd 1c 9e a8 bd 58 99 a1
  02 15 00 95 10 42 e1 cb b9 1d 26 34
  d4 5f b1 0d b8 66 ba 8c 61 20 c3
```

Ephemeral DH with DSS is conceptually very similar to ephemeral RSA. Only the details of the messages are different.

A DH/DSS ServerKeyExchange message contains Diffie-Hellman parameters signed with a DSS key. The provided parameters are the prime modulus (p), the generator for the group (g) and the server's public key (Ys). Because DSS requires a 160-bit input, the DSS signature is computed over the sha_hash value only. Recall that DSS signatures consist of two large integers, called r and s. The SSL representation of these in the message is the DER encoding of an ASN.1 structure defined in ANSI X9.57 [ANSI1995].

```
Dss-Sig-Value ::= SEQUENCE {
    r     INTEGER,
    s     INTEGER
}
```

Note: The description of how to encode DSS signatures in the SSLv3 specification is insufficiently specific and Netscape interprets it differently from most other vendors. Instead of DER encoding r and s, Netscape merely concatenates them into a single 40-byte field. Thus, although Netscape implements DSS for client authentication, it is not interoperable with other implementations. Despite widespread agreement on the "right thing," Netscape has refused to change their implementation, claiming that it complies with the SSLv3 specification and citing installed base. The TLS specification clears up this issue; all TLS implementations must DER encode DSS signatures.

ClientKeyExchange

```
struct {
  select (KeyExchangeAlgorithm) {
    case rsa: EncryptedPreMasterSecret;
    case diffie_hellman: DiffieHellmanClientPublicValue;
  } exchange_keys;
} ClientKeyExchange;

struct {
  select (PublicValueEncoding) {
    case implicit: struct {};
    case explicit: opaque DH_Yc<1..2^16-1>;
  } dh_public;
} DiffieHellmanClientPublicValue;
```

```
ClientKeyExchange
  DiffieHellmanClientPublicValue[64]=
    14 ea e2 18 c1 69 b3 60 fc ea c7 54
    f7 18 db b9 47 c7 cf 95 80 2a 32 b7
    0c 07 11 ab 7a 9d dc 0a 1c 82 a1 35
    23 1f 90 71 2a 94 6d d8 86 b4 e2 84
    e9 a6 a2 00 5e bb 82 09 a3 8a ba f2
    e8 29 87 61
```

Because Diffie-Hellman requires that the client and the server share a common set of parameters (*g* and *p*, found in the ServerKeyExchange), the client needs to send only its public key (*Yc*) in the ClientKeyExchange message. *Yc* is represented in the usual SSL style for big integers, high byte first.

Fixed DH Keys

Because the client has a DH key, it is possible for the client to have a certificate binding the key to the client's identity. This certificate can be used to perform client authentication, using the rsa_fixed_dh and dss_fixed_dh certificate types. These certificates are certificates containing DH keys signed with RSA and DSS respectively. The client sends its certificate in response to the CertificateRequest but does not send a CertificateVerify. Rather, it proves that it possesses the corresponding private key by being able to produce the pre_master_secret using its private key. Because the client's public key is present in the Certificate message, no public key is present in the ClientKeyExchange message (the implicit choice in the DiffieHellmanClientPublicValue structure).

Similarly, it is also possible for the server to have a static DH key that is contained in its certificate. The server's identity is proven by being able to reproduce the pre_master_secret. Neither of these operational modes is as common as the ephemeral DH modes using DSS.

4.17 FORTEZZA Details

The FORTEZZA support described in [Freier1996] does not work. It was apparently written without ever being tested. The various changes required for successful operation are described in [Relyea1996].

The FORTEZZA command interface has a number of properties that make FORTEZZA difficult to use with SSL. The most important one is that the card will not use arbitrary keys. Thus, keys must either be generated on the card by the card random number generator or received encrypted from another card. As a result, it's not possible to use the standard SSL key generation procedure to generate the FORTEZZA encryption keys.

The second important limitation is that the card must generate the IV. This requirement was imposed by the escrow feature. The escrowed session is carried in something called the *Law Enforcement Access Field* (LEAF). The LEAF contained the session key

used to encrypt the actual traffic. The session key was wrapped in the card key. In order to ensure that the LEAF is carried along with the ciphertext, it was embedded in the IV, resulting in a 24-byte IV. The card automatically extracts the actual 8-byte IV from the IV given to it.

> In 1997, the NSA bowed to pressure over the escrow issue and removed the LEAF. However, for compatibility, the 24-byte IV remained with a dummy LEAF. Finally, on June 23, 1998, SKIPJACK and KEA were declassified.

```
struct {
  opaque r_s [128];
} ServerFortezzaParams;

struct {
  opaque y_c<0..128>;
  opaque r_c[128];
  opaque y_signature[40];
  opaque wrapped_client_write_key[12];
  opaque wrapped_server_write_key[12];
  opaque client_write_iv[24];
  opaque server_write_iv[24];
  opaque master_secret_iv[24];
  block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

The ServerFortezzaParams structure is used to carry the server's random value r_s to the client. The FortezzaKeys structure is carried in the ClientKeyExchange message. It contains all the information that the server will need to derive the same keys as the client.

The y_c value contains the client's KEA public key if it's not in the client's certificate. If it is, this field is empty. The r_c value contains the client's random value. y_signature contains the client's signature over y_c. The specification is unclear on what the value of y_signature should be if the client's KEA public key is in its certificate. In KEA, the client and server public keys are combined with the random values to generate a *Token Encryption Key* (TEK) which can then be used to encrypt other keys.

wrapped_client_write_key and wrapped_server_write_key contain the encryption keys wrapped under the KEA-generated shared key. client_write_iv and server_write_iv contain the IVs for these keys (with the LEAF embedded in them). Finally, the FortezzaKeys structure contains the pre_master_secret encrypted under the shared key with an IV given in master_secret_iv. The pre_master_secret is used only to generate the MAC keys.

The FORTEZZA command interface imposes one other limitation: IVs are intended to be generated by the sender and read by the recipient. Thus, it's possible to load an IV onto the card when decrypting but not encrypting. However, both keys and IVs are

generated by the client. The result is that the first block of data encrypted by the server uses an essentially random IV—which the client has no way of knowing. In order to synchronize the CBC state, the server first transmits a dummy encrypted block that is discarded by the client. From then on, the cipher states will be synchronized.

Note that the encryption of this dummy block means that the server never needs to use the server_write_iv. [Relyea1996] says that the server implementation should load the IV in decryption mode to verify the IV and then change over to encryption mode. This step is not required for interoperability. Its purpose is to verify that IV correctly matches the key, thereby preventing a rogue client from using a fake IV (with a bogus LEAF) and thereby thwarting the escrow provisions.

The inability to use random keys also means that ordinary session resumption will not work, since there is no way to use the newly generated keys. [Relyea1996] describes two options:

- Don't resume sessions at all.
- Resume but continue encrypting and decrypting as if the data were a continuation of the previous session. Note that if this approach is taken it is not possible to have two simultaneous connections with a given session, as it ordinarily would be. This is problematic for systems such as HTTPS which use a number of simultaneous connections.

Only two FORTEZZA cipher suites were defined, one with SKIPJACK and one with NULL encryption. Figure 4.11 shows them.

Cipher Suite	Authentication	Key Exchange	Encryption	Digest	Number
SSL_FORTEZZA_DMS_WITH_NULL_SHA	DSA	KEA	NULL	SHA	0x001c
SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA	DSA	KEA	SKIPJACK	SHA	0x001d

Figure 4.11 FORTEZZA cipher suites

4.18 Error Alerts

In Chapter 3, we discussed alerts in general without discussing the specific alert cases other than close_notify. All the other alerts indicate errors of some kind. This section presents a detailed discussion of those alerts.

Recall that alerts may have a level of warning or fatal. A fatal alert always terminates the connection, whereas implementations may continue past a warning alert. While in theory implementations can continue past a non-fatal alert, in practice even warning alerts indicate that one side requested something that the other side couldn't fulfill and so they usually result in connection closure.

The specification requires some alerts to be fatal whereas others may be either fatal or warning. An implementation which receives an alert which should be fatal but which is incorrectly tagged as warning should nevertheless treat it as fatal. In practice, since

many implementations treat all alerts as fatal, implementations should refrain from sending warning alerts unless they're willing to have the other side terminate the connection.

This section describes each SSL error alert in detail. Although some of the alerts could signal an attack in progress, if you're debugging your SSL implementation, they most likely represent bugs. Where appropriate we'll attempt to identify the common implementation errors that these alerts might indicate. Reading this section may allow you to determine what's happened when your implementation receives or generates one of these alerts. See Figure 4.12 for a list of all the alerts.

Alert Name	Level	SSL Version	Number
close_notify	either	either	0
unexpected_message	fatal	either	10
bad_record_mac	fatal	either	20
decryption_failed	fatal	TLS	21
record_overflow	fatal	TLS	22
decompression_failure	fatal	either	30
handshake_failure	fatal	either	40
no_certificate	either	SSLv3	41
bad_certificate	either	either	42
unsupported_certificate	either	either	43
certificate_revoked	either	either	44
certificate_expired	either	either	45
certificate_unknown	either	either	46
illegal_parameter	fatal	either	47
unknown_ca	fatal	TLS	48
access_denied	fatal	TLS	49
decode_error	fatal	TLS	50
decrypt_error	either	TLS	51
export_restriction	fatal	TLS	60
protocol_version	fatal	TLS	70
insufficient_security	fatal	TLS	71
internal_error	fatal	TLS	80
user_cancelled	fatal	TLS	90
no_renegotiation	warning	TLS	100

Figure 4.12 All SSL/TLS alerts

unexpected_message

The unexpected_message alert signals that an inappropriate message was received. This alert almost always signals that one of the implementations is broken and generating inappropriate messages or sending messages in the incorrect order. You might see this message often when debugging an SSL implementation but should never see it in working systems. As a special case, a non SGC-capable server might send an unexpected_message to a client attempting SGC. This alert must always be sent as fatal.

bad_record_mac

The bad_record_mac alert is transmitted if an implementation receives a record with a bad MAC. This might represent an attack. Thus, this alert must be treated as fatal.

If you're testing your implementation and you receive a bad_record_mac alert, it most likely indicates an error in your MAC code. However, if the alert is sent in response to a Finished message, it could also represent an error in key derivation. Even if the encryption keys are also wrong, the decryption may appear to succeed. However, the MAC will fail. In fact, even if the MAC key is correct but the encryption key is wrong, the error will still most likely appear in the MAC check (see the decryption_error alert for a discussion of decryption errors).

Finally, some SSLv3 implementations use the bad_record_mac message to signal an error in verifying the Finished message itself. In TLS, failure to verify the Finished is signalled with the decrypt_error alert. Thus, when debugging a bad_record_mac error, it's important to consider other culprits besides the MAC computation itself.

decryption_failed

The decryption_failed signals that the ciphertext couldn't be decrypted. This error must be fatal. This error is really only relevant with block ciphers. RC4 decryption always appears to succeed whatever the decryption key. The output is simply wrong.

Block cipher decryption can fail in a number of ways. First, the block cipher input length can be something other than a mutiple of the block length. This is an obvious message formatting error. Also, after the data is decrypted, the padding check may reveal errors resulting from incorrect padding or the use of the wrong encryption key.

If the wrong encryption key is used, the padding bytes will be essentially random. However, it's possible that they will still appear correct, particularly if the implementation chooses to examine only the length byte and then remove the padding without examining it. (This procedure is safe because the data is covered by the MAC anyway.) Note that the padding does not have to actually be the same as the original padding. It merely has to be consistent.

In this case, the receiver will have the wrong plaintext but this will only be discovered when the MAC check and a bad_record_mac alert will be received. Obviously, since RC4 decryption always appears to succeed, errors in RC4 keying always show up in the MAC check.

record_overflow

This alert indicates that the record was larger than allowed by the SSL specification. This message must be fatal. In practice, this message almost always represents a bug in the SSL implementation since an attack would likely be caught in the MAC check. Note that early versions of Internet Explorer generated incorrectly long records and so it's not uncommon to see an unsuspecting implementation generate this alert when talking to IE. This alert is new with TLSv1.

decompression_failure

This alert indicates that a record could not be decompressed. It is always fatal. However, since compression is almost never used, this error will rarely be seen in practice.

handshake_failure

The handshake_failure alert is the all-purpose alert used in SSLv3 to indicate problems handshaking. TLS introduced a number of finer-grained alerts to indicate what actually went wrong. In TLS, the handshake_failure alert is only used when no common cipher can be negotiated.

When debugging a handshake_failure alert, you need to examine which stage of the handshake went wrong and from there determine what happened. If you received a handshake_failure alert after sending a series of messages at once, it's often useful to send each message individually, checking for the alert after each one using a network sniffer. This alert must always be fatal.

no_certificate

The no_certificate alert is an SSLv3-only alert sent to indicate that "no appropriate certificate is available." It has been replaced in TLS with certificate_unknown which serves essentially the same purpose.

bad_certificate

The bad_certificate alert indicates that a certificate was corrupt or the signatures were wrong or some similar error. In theory this alert may be a warning. An implementation might choose to ignore the bad certificate but issue the alert as a warning. However, in practice implementations either silently accept the certificate or send an alert. Thus, this alert is really always fatal. This comment applies to the next four alerts as well. Interestingly, Internet Explorer doesn't use this alert at all. Instead, it simply closes the connection. We'll see an example of this in Chapter 8.

unsupported_certificate

This alert means that the implementation received a certificate which was of an unsupported type. Since SSL only supports X.509 certificates, this alert means that an unsupported algorithm was used. Because the algorithm used to sign certificates is not specified in the cipher suite, it's possible to have (for instance) a certificate for an RSA key signed with a DSA key. An implementation which did not support DSA would send an unsupported_certificate error. However, in practice such mixed certificate regimes nearly never occur and an unsupported_certificate error represents a bug in one of the implementations.

revoked_certificate

This alert indicates that the sender received a certificate that was revoked. Most real life SSL implementations do not currently support revocation so this alert almost never appears.

certificate_expired

This alert indicates that one of the certificates has expired. Implementations shouldn't send expired certificates, but sometimes they are misconfigured. This alert is the result.

certificate_unknown

This alert is a catch-all certificate error introduced in TLSv1. Certificate errors that don't fall into any of the previous classes or the unknown_ca class described below generate a certificate_unknown error. SSLv3 implementations would generate a bad_certificate alert for those conditions.

illegal_parameter

The illegal_parameter alert indicates that one of the handshake fields was out of range or inconsistent with some other fields. The handshake obviously can't complete so this error is fatal.

unknown_ca

The unknown_ca alert is sent when an implementation receives a certificate signed by a CA it doesn't recognize or can't find the certificate for. This alert must be fatal. Note that this treatment is inconsistent with the other certificate errors which can be sent as warnings. I know of no reason for the inconsistency. In my opinion, best practice is to either silently accept certificates or to reject them with an alert. Note that in this context *silent* refers only to SSL messages. An implementation might accept a bad certificate only after prompting the user or sending error messages to a logging facility. The

unknown_ca message is new with TLS. SSLv3 implementations send the no_certificate alert instead.

access_denied

This alert means that although a valid certificate was received the identity in the certificate fails access control checks so the connection is being rejected. This alert is always fatal. I know of no implementation which generates this alert.

decode_error

This alert indicates that a message couldn't be decoded because one of the fields was out of range or the length was incorrect. In most cases this represents some sort of implementation error. This alert is always fatal. decode_error was introduced in TLS and it's not clear what to use with SSLv3. OpenSSL and PureTLS both use handshake_failure, presumably on the theory that most of these errors occur in the handshake.

decrypt_error

The decrypt_error alert indicates that one of the handshake cryptographic operations failed. The likely cases are failure to decrypt a ClientKeyExchange, verify a signature, or check a Finished message. This alert is always fatal. decrypt_error was introduced with TLSv1. SSLv3 implementations usually use handshake_failure instead. Note that if you receive a decrypt_error in response to a Finished, this means that your encryption and MAC keys and code are correct but that your Finished computation is broken.

export_restriction

This alert indicates that one of the implementations tried to violate export restrictions. The example provided in RFC 2246 is trying to use a 1024 bit ephemeral key with RSA_EXPORT key exchange. This is a new alert type for TLSv1 and isn't seen much. Note that this is not the alert that you will see when an implementation that supports only strong ciphers tries to talk to one that supports only weak ones. That will be either insufficient_security or handshake_failure. This alert is always fatal.

protocol_version

The protocol_version alert is sent by the server to indicate that the client used an unrecognized protocol version. Because the client can only supply the maximum version that it supports, the server will only send this alert when the client asked for a version older than it will support. For instance, if the client offered only SSLv3 and the server supported only TLS, the server would generate a protocol_version alert. Since essentially all TLS implementations are also SSLv3 implementations, this alert is only seen if one

of the sides has been explicitly configured to support only TLS or if there is a bug. There is no value in configuring for TLS-only support.

Note that if the server only supports a version older than the client offers, it will *not* send a protocol_version alert. Instead, it will simply respond with its version in the ServerHello. If the client does not want to support the server's version choice, it will have to send an alert (presumably protocol_version but the SSLv3 and TLS specs don't specify). The protocol_version alert is always fatal. The protocol_version alert was added in TLS. SSLv3 implementations usually use handshake_failure for this purpose.

insufficient_security

The insufficient_security alert is used to indicate that the cipher suites offered by the client were weaker than required by the server. This alert is new with TLS. SSLv3 implementations use handshake_failure. Since the insufficient_security alert indicates failure to negotiate a cipher suite, it is always fatal.

internal_error

The internal_error alert is used to indicate that the transmitting implementation encountered some sort of internal error such as a memory allocation or hardware failure. This alert is new with TLS so SSLv3 implementations need to use some other error. The internal_error alert is always fatal.

user_cancelled

The user_cancelled alert is used when the user cancels the handshake for some reason. This alert is new for TLS. RFC 2246 says that it should be used as a warning, which is confusing—if the user cancels the handshake then the connection will be unsuccessful so the alert should be fatal.

no_renegotiation

As we'll see in Chapter 4, SSL allows either client or server to initiate a new handshake at any time other than during a handshake. The client initiates renegotiation by sending a new ClientHello. The server initiates renegotiation by sending a HelloRequest message. RFC 2246 says that the client can silently refuse to renegotiate or send a no_renegotiation alert. RFC 2246 is unclear on how the server should respond to an unwanted ClientHello but presumably it should send a no_renegotiation alert as well.

The no_renegotiation alert is new for TLS. SSLv3 implementations simply ignore unwanted renegotiation requests. Since most existing TLS implementations are actually upgraded SSLv3 implementations, they typically silently ignore renegotiation rather than sending the no_renegotiation alert. In any case, the no_renegotiation alert is always a warning.

4.19 SSLv2 Backward Compatibility

SSLv2 had already been widely deployed when SSLv3 was invented. Thus, one of the requirements for SSLv3 was for SSLv3 clients and servers to automatically talk to SSLv2 clients and servers. However, the message formats were totally different. (See Appendix B for a discussion of SSLv2.) It's fairly straightforward for servers to auto-detect whether they are talking to an SSLv3 or SSLv2 client. The SSLv2 and SSLv3 length bytes are in different locations and SSLv3 records appear to be unreasonably long when interpreted as SSLv2 handshake messages. Thus, the server can wait for the client's first message and respond appropriately.

Clients, however, must speak first and so cannot auto-detect. Thus, the SSLv3 specification contains support for sending the ClientHello in a format that was backward-compatible with SSLv2. In essence, the backward-compatible ClientHello maps all the SSLv3 fields onto SSLv2 fields in the SSLv2 CLIENT-HELLO. Thus, when a server receives a backward-compatible CLIENT-HELLO, it can either interpret it as an SSLv2 message and continue with the SSLv2 handshake or translate the fields to SSLv3 values and perform the SSLv3 handshake. Figure 4.13 shows the SSLv2 CLIENT-HELLO.

```
char MSG-CLIENT-HELLO
char CLIENT-VERSION-MSB
char CLIENT-VERSION-LSB
char CIPHER-SPECS-LENGTH-MSB
char CIPHER-SPECS-LENGTH-LSB
char SESSION-ID-LENGTH-MSB
char SESSION-ID-LENGTH-LSB
char CHALLENGE-LENGTH-MSB
char CHALLENGE-LENGTH-LSB
char CIPHER-SPECS-DATA[(MSB<<8)|LSB]
char SESSION-ID-DATA[(MSB<<8)|LSB]
char CHALLENGE-DATA[(MSB<<8)|LSB]
```

Figure 4.13 SSLv2 CLIENT-HELLO

The CLIENT-HELLO shown is in the SSLv2 specification language of [Hickman1995]. Unlike SSLv3, SSLv2 places all the object lengths in one part of the message and all the objects in another. Also, the high and low bytes of lengths are represented individually. Thus, the length of the CIPHER-SPECS-DATA field is given by computing

```
(CIPHER-SPECS-LENGTH-MSB << 8) | CIPHER-SPECS-LENGTH-LSB
```

All of the SSLv3 handshake parameters can be mapped to SSLv2 values.

Ex.1024
Askeladden, L.L.C. / Page 205 of 788

Version

The SSLv2 version number is 2 (MSB=0, LSB=2). To represent SSLv3 or TLS compatibility, we use the appropriate version numbers from the SSLv3 or TLS Version field: 0x300 and 0x301 for SSLv3 and TLS, respectively.

Random

The SSLv2 CHALLENGE value is used as part of the SSLv2 key generation procedure in much the same way as the ClientRandom is used by SSLv3. It's randomly generated, so it can be used as a ClientRandom value. However, it's allowed to be between 16 and 32 bytes, whereas the SSLv3/TLS ClientRandom must be 32 bytes. To convert a CHALLENGE to a ClientRandom value, it is padded with zeros on the left.

Session ID

Whenever resuming an SSLv3 session, the SSLv3 ClientHello must be used. Thus, the SESSION-ID field should always be empty and the length zero.

Cipher Suites

SSLv2 CIPHER-SPECSs are analogous to SSLv3 CipherSuites. They specify the encryption and digest algorithms. (SSLv2 only supports RSA, so they don't specify key exchange or signature.) Thus, in backward-compatibility mode, the CLIENT-HELLO should contain both the SSLv2 and SSLv3/TLS ciphers that the client supports. SSLv2 CIPHER-SPEC values are 3 bytes, so 2-byte SSLv3 values are represented with a zero first byte.

It's also possible to use SSLv2 ciphers when using the backward compatibility handshake. Some of these ciphers correspond to SSLv3/TLS ciphers, but some do not. For instance, SSLv2 has an RC2-128 mode. SSLv3 does not define such a cipher suite. However, it's possible for two SSLv3 clients to negotiate it if they are using the SSLv2 backward compatible handshake. All these cipher suites use RSA for signature and key exchange. Figure 4.14 lists these ciphers.

Cipher Spec	Encryption	Digest	Number
SSL_CK_RC4_128_WITH_MD5	RC4_128	MD5	0x010080
SSL_CK_RC4_128_EXPORT40_WITH_MD5	RC4_40	MD5	0x020080
SSL_CK_RC2_128_CBC_WITH_MD5	RC2_128_CBC	MD5	0x030080
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	RC2_40_CBC	MD5	0x040080
SSL_CK_IDEA_128_CBC_WITH_MD5	IDEA_CBC	MD5	0x050080
SSL_CK_DES_64_WITH_MD5	DES_CBC	MD5	0x060080
SSL_CK_DES_192_EDE3_CBC_WITH_MD5	DES_EDE3_CBC	MD5	0x070080

Figure 4.14 SSLv2 compatibility cipher suites

Rollback Protection

SSL uses PKCS #1 padding to encrypt data using RSA. However, in order to protect against an active attack in which the attacker forces negotiation back to SSLv2, SSLv3- and TLS-capable client implementations must use the value 03 for the last 8 bytes of the padding when doing SSLv2. The server can check this to make sure that it really is talking to an SSLv2-only client (which would fill these 8 bytes with random data). Note that this unusual padding will not appear and does not need to be checked if doing an SSLv3 handshake. The purpose of all this is to prevent an attacker from forcing a client/server pair back to the (weaker) SSLv2 from SSLv3/TLS.

Compatibility

In order for the backward-compatibility handshake to work correctly, an SSLv2-only server must simply negotiate to SSLv2 if it receives a higher version in the CLIENT-HELLO. Unfortunately, the SSLv2 specification wasn't very clear on how servers should handle CLIENT-HELLO messages with version numbers higher than they support. This problem was made worse by the fact that Netscape's SSLREF reference implementation simply rejected connections with higher version numbers. Thus, it's not guaranteed that all SSLv2 servers will respond correctly to the backward-compatible handshake, although the vast majority will.

4.20 Summary

This chapter concludes our detailed coverage of SSL. Each of these modes offers some benefit over the traditional RSA server-only mode discussed in Chapter 3.

Session resumption allows a client/server pair to reuse keying material between connections. This makes subsequent handshakes between a client/server pair faster.

Client authentication uses the client's public key to prove the client's identity to the server. Traditionally, SSL connections have anonymous clients, but with client authentication, clients as well as servers can authenticate using certificates.

Ephemeral RSA allows a server to have a single long (>512 bits) RSA key but still to communicate with exportable clients which require 512-bit RSA keys, by generating a temporary 512-bit key and signing it with the long-term key.

Server Gated Cryptography allows exportable clients to use strong cryptography with specifically designated servers. These servers must have special certificates that indicate that they are SGC eligible.

DSS and DH cipher suites make it possible to implement SSL in the United States without infringing on the RSA patent. Support for DSS and DH has been mandated for TLS.

Elliptic Curve ciphers may be faster than either DH/DSS or RSA, but they have not yet been standardized by IETF as part of TLS.

Kerberos keys can be used to exchange keying material between client and server, while keeping the rest of SSL the same. This makes it possible to bootstrap a Kerberos infrastructure to authenticate SSL connections.

SSLv3 is backward-compatible with SSLv2. The SSLv3 ClientHello can be cast as an SSLv2 CLIENT-HELLO, thus permitting SSLv3 agents which also speak SSLv2 to communicate with SSLv2-only agents.

5

SSL Security

5.1 Introduction

This chapter discusses the security properties of SSL. This chapter is divided into three sections. The first section of the chapter provides general rules for using SSL securely. Understanding this material requires some understanding of cryptography and an understanding of SSL approximately equal to that provided by the first halves of the previous two chapters. The idea here is to explain the security properties of SSL and how to use it securely.

The second section discusses the security threats discussed in the first part in more detail. It's fairly easy to get an interoperable implementation that nevertheless has hidden vulnerabilities if you're not careful while writing it. Where the first part provides general guidelines, this section covers the attacks that are possible if those guidelines aren't followed and discussion of the various implementation techniques that are in common use.

The final part of the chapter covers some more exotic security threats in detail. Although there have been no good attacks on the SSL protocol itself, a number of SSL implementations have been successfully attacked and some care is required to ensure that one's implementation is not vulnerable to these threats. This section is fairly technical, but you can get a good understanding of the security properties of SSL merely by reading the first two-thirds of the chapter.

5.2 What SSL Provides

SSL provides channel-level security. What this means is that the two ends of the connection know that the data being transmitted is being kept secret and that it has not been tampered with. The server is nearly always authenticated so the client knows who's on the other end of the connection. The client can be required to authenticate using his certificate as well. SSL also provides for secure notification of exceptional conditions, including error alerts and connection closure.

Ex.1024
Askeladden, L.L.C. / Page 209 of 788

All of these protections depend on certain assumptions about the system: we assume that the keying material has been correctly generated and is being kept secure. Failure to follow careful procedures can result in a significant loss of security.

5.3 Protect the master_secret

Nearly all the security of the protocol depends on keeping the master_secret private. If the master_secret for a session is compromised, then the session is completely vulnerable to attack. Remember that the master_secret is used to produce all the cryptographic keys used to protect the data. An attacker who knows it can therefore mount a large number of attacks.

Section 5.9 discusses the detailed consequences of master_secret compromise, but the short description is that if an attacker has the master_secret the data might as well be transmitted in the clear. Such an attacker can read all the message traffic and under most circumstances can undetectably forge traffic from either side. This is not good.

5.4 Protect the Server's Private Key

One obvious way for an attacker to get the master_secret is to acquire the server's private key. In order to prevent this, you have to keep the server's private key (surprise!) private. This sounds obvious, but it's one of the most widely violated rules and it's also surprisingly difficult to accomplish.

In general, an attacker who has the server's private key constitutes a serious threat to the security of the system. If the server is using the common static RSA mode discussed in Chapter 3, an attacker with the server's private key can recover the master_secret and undetectably read any traffic he can capture, as well as pose as the server at will. If the server is using ephemeral modes, the attacker can't passively read traffic but can mount an active attack posing as the server. In either case, compromise of the server private key is very dangerous.

Protecting the private key requires storing it securely. Most implementations encrypt the private key on disk and require a password/passphrase to decrypt it. This requires the user/administrator to generate and remember a strong passphrase, which turns out to be remarkably difficult to do. Other implementations store the key in secured hardware. Both approaches require intervention from the administrator to start the server, making unattended reboots in case of system crash or power failure problematic.

5.5 Use Good Randomness

Nearly all cryptographic protocols require *strong random numbers* to operate securely and SSL is no exception. Random number generators (RNGs) are used primarily to generate keys but also for key steps in other cryptographic operations. If the attacker can predict what numbers your RNG is going to output, then he can guess what your keys are, completely compromising the protocol.

SSL uses random numbers in a number of places. First, the private keys for the server (and optionally the client) need to be randomly generated. Second, the client needs to generate random data to perform the key exchange. If RSA is being used, then the client must generate the pre_master_secret and if using DH, it must generate an ephemeral private key. Third, if DSA is being used for signature, then a random number must be generated for each signature. Finally, the client and server both need to generate their handshake random values, but those do not in fact need to be securely generated because they are public.

Thus, both client and server need to be able to generate strong random numbers. If the server generates weak random numbers, then the attacker can guess the server's private key, leading to compromise of the pre_master_secret. If the client generates weak random numbers, then the attacker can guess either the pre_master_secret directly (if in RSA mode) or the client's ephemeral DH private key and transitively the pre_master_secret (if in DH mode). In either case, the protocol is completely compromised.

Most systems use software based *pseudo-random number generators* (PRNGs). A PRNG is an algorithm that provides a stream of unpredictable (though not random) numbers. These PRNGs must be *seeded* with some random data in order to produce a high-quality random stream. The seed data is usually collected from network traffic sampling, system internal variables, or user input timing. Hardware random number generators based on physical randomness are also available. See Section 5.12 for guidance on generating strong randomness.

5.6 Check the Certificate Chain

Let's assume that your SSL implementation is operating correctly, and you receive a certificate from the server. That certificate is signed by a *Certification Authority* (CA) that you don't recognize. This provides no assurances about the server's identity. Although the server has asserted its identity via its certificate, it's nothing more than a certificate because you can't verify that the assertion is correct. You can still establish a secure connection to the server, but you're vulnerable to an active (man-in-the-middle) attack where the attacker poses as the server to you and you to the server. Knowing the server's identity is absolutely necessary to avoid this attack.

A server without a certificate server is obviously unauthenticated, but there are more subtle ways in which a seemingly good certificate chain can provide less security than you might expect. The SSL implementation can help to protect you against these attacks, but often having complete security requires that the application get involved.

The most important countermeasure is to check that the identity being presented in the certificate matches the identity that the application expects. Imagine that you're trying to connect to a server operated by Bob and the server provides you with a certificate bearing the name Alice. The certificate is valid, but it's the wrong one. Consequently, you need to check that identity in the certificate corresponds to the expected identity of the other end of the connection.

Correctly checking the identity requires interaction with the application-layer

Ex.1024
Askeladden, L.L.C. / Page 211 of 788

protocol and potentially with the user. Only they have all the information as to the expected user's identity. Another case in which the user may need to be notified is expired certificates. In some cases, organizations may have policies forbidding the use of expired certificates, but in others the decision may be up to the user. We'll discuss the detailed interaction of the certificate chain with the application-layer protocol in Chapters 7, 9, and 10.

5.7 Algorithm Selection

SSL supports a variety of cipher suites, specifying the set of algorithms to be used for the connection. These algorithms vary in strength from very weak exportable ciphers such as RC4 in 40-bit mode to (hopefully) very strong ciphers such as 3DES. Moreover, asymmetric key pairs can use keys of a range of lengths. The security of an SSL connection is naturally totally dependent on the security of the ciphers it uses. It's therefore necessary to choose a cipher suite commensurate with the value of your data.

To do this, limit the set of cipher suites that your application is willing to support. Because both client and server have to agree on a common cipher suite in order to communicate, this ensures that you will obtain the appropriate level of security. However, a possible side effect is that the server and client may not have a common algorithm set and not be able to communicate. As a consequence, the best policy is typically to allow all cipher suites that provide security above an acceptable minimum level rather than merely choosing to support the strongest algorithms exclusively.

Our recommendation is to use either RSA or DH/DSS with minimum 768-bit keys. Use 3DES or RC4-128 for encryption, 3DES if you want maximum security, and RC4-128 if you want maximum performance. Use SHA-1 for message authentication. See Chapter 1 for performance numbers for these algorithms.

5.8 The Story So Far

We've covered a number of generally important rules for using SSL securely. This ought to provide some idea of the sorts of measures you need to take in order to achieve decent security. This isn't to say that the list is exhaustive—there are an infinite number of ways to screw up, mostly by not paying attention to what you're doing—but it covers a number of important issues that are easy to get wrong. The author has personally seen at least one implementation break each of the previously described rules.

5.9 Compromise of the master_secret

Compromise of the master_secret is disastrous. Let's reexamine key derivation for a moment to see how master_secret (or pre_master_secret) compromise can lead to compromise of the entire system. Figure 5.1 shows the key derivation process. Secret information is shaded, all other information is public.

Ex.1024
Askeladden, L.L.C. / Page 212 of 788

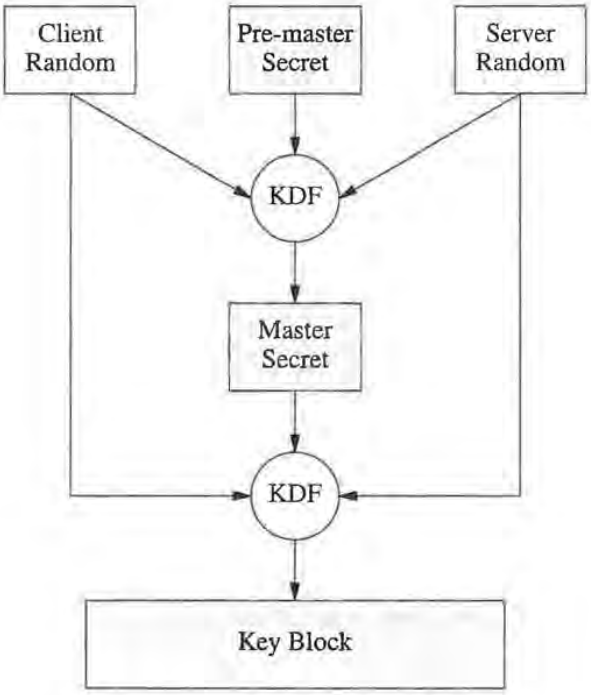


Figure 5.1 SSL key derivation

Recall that key derivation proceeds in two stages. First, the pre_master_secret is converted into the master_secret. The *key derivation function* (KDF) takes three parameters that are different for every handshake: the client_random and server_random values and the pre_master_secret. However, recall that the client_random and server_random values are exchanged before the master_secret is agreed upon, so they must be transmitted in the clear in the handshake. Thus, an attacker who can snoop on traffic already knows them. The only piece of secret information used to generate the master_secret is the pre_master_secret.

The second stage is to convert the master_secret into the individual cryptographic keys. As before, the KDF takes three inputs: the client_random and server_random values and the master_secret. Again, only the master_secret is a secret. An attacker who knows the master_secret can therefore compute all the cryptographic keys that are used to protect the connection. Similarly, an attacker who knows the pre_master_secret can compute the master_secret and transitively the keys.

So, if an attacker does recover the master_secret, the results are very bad. From the perspective of such an attacker, you might as well be transmitting your messages in the clear.

Confidentiality Attacks

As we said earlier, an attacker who knows the master_secret can compute all the cryptographic keys used for the connection. Naturally, this includes the encryption keys used to encrypt the data. Recall that our threat model assumes that attackers can read any data that is transmitted over the network. Of course, when we're using SSL, this data is encrypted, but this poses no problem to an attacker with the master_secret. Such an attacker has the encryption keys and can easily decrypt each record captured.

There is one trivial technical problem: If the attacker fails to capture every record, some work is needed to decrypt subsequent records. Recall that each record is encrypted as if it were part of a continuous stream. For a stream cipher, each record uses a different section of keystream. In order to decrypt a record, the attacker needs to know how many bytes have already been encrypted. However, even if the attacker misses a record, he can usually get a good guess at how many bytes he's missed from the TCP sequence number and do a little searching to find the exact right section of keystream.

With block ciphers, the situation is a little trickier. The IV of each record is the last cipher block of the previous record. If the attacker has missed that record, then he has no way of decrypting the first cipher block of the record, but he can decrypt the rest of the record and any records that come after that.

Note that this attack is completely passive. If the attacker is willing to interfere with network traffic, he can work around the missed records problem more easily by arranging for retransmits. On the other hand, if the attacker is willing to mount an active attack, he can also mount some significant integrity attacks. We'll discuss that next.

Integrity Attacks

Of course, an attacker who has the master_secret can compute the MAC keys. In combination with the encryption keys, such an attacker can mount a wide variety of integrity attacks. We'll describe a few here, but this list is not exhaustive.

The easiest form of attack is simply to take over the connection. The attacker forges a close on the underlying connection to one side (typically using TCP RST) and proceeds to impersonate that side to the other. The attacker can create convincing forgeries of whatever traffic it chooses. This attack isn't very subtle, but it can be very effective. Note that this sort of attack doesn't require very fine control of the underlying network: Techniques for TCP connection hijacking are well known (see [Joncheray1995]).

A variety of more subtle attacks are possible if the attacker has good control of the network. For instance, the attacker might be able to add and delete packets at will while preserving the connection. Then it's possible to change any record or even insert or remove records. Exactly what capabilities are required depends on what sort of modification the attacker wants to make to the traffic and what ciphers are being used.

If the traffic the attacker is modifying is encrypted using a stream cipher, the attacker can make point modifications to individual records simply by decrypting the record, changing the message text, generating a new MAC, reencrypting, and sending. Note that the attacker can't change the length of the record or all future records will be out of synch with the keystream.

On the other hand, the attacker can make arbitrary changes to any record he likes if

he's willing to intercept and modify all subsequent records. There are a number of intermediate forms of attack that require modifying fewer records, but it should be clear by now that an attacker who knows the master_secret can make the traffic look however he wants it to look.

Pretty Scary, Eh?

We hope that at this point you're convinced that allowing an attacker to get hold of the master_secret is disastrous and that it must therefore be vigorously protected. The good news is that a *correctly* written SSL implementation makes recovering the master_secret very difficult if not impossible. The next few sections describe a few easy-to-make mistakes that allow master_secret recovery.

5.10 Protecting Secrets in Memory

Unless the SSL implementation resides largely in hardware, the master_secret will live in the host's main memory. This means that any attacker who can read the memory of the SSL process can read the master_secret. Thus, it's not possible to protect your SSL connection from an attacker who has administrator privileges on your machine. However, it's possible to make mistakes that potentially allow non-administrators to access the master_secret.

Disk Storage

In general, implementations should refrain from writing secret data to disk at all. However, in some circumstances (see Chapter 9), it's necessary. In such cases, the implementation should take care to set the permissions on the files so that no other users can read the data. Moreover, the implementation should ensure that the disk files are deleted before it exits. On operating systems which have an undelete function, the program should ensure that the files are erased beyond recovery. Good practice is to scrub the data by doing three overwrite passes: the first of zeros, the second of random data, and the third of zeros. This is of particular concern on single-user operating systems such as Windows 95 and 98, where permissions offer no protection and subsequent users can read the entire disk.

Locking Memory

On operating systems with virtual memory, it may be desirable to ensure that the memory containing the master_secret (and other secrets) is never swapped to disk. If it is, a subsequent user or the administrator might be able to read that data and thus recover the secrets. Many operating systems offer subroutine calls to *lock* sensitive data so it cannot be swapped to disk. However, in some cases these calls are actually inactive stubs. Consult your operating system documentation carefully for more information on this topic.

Note that locking memory doesn't offer any protection from an administrator who reads sensitive data from the process while it's resident in memory. It merely protects against an administrator subsequently deciding to recover the data.

Core Dumps

On many operating systems (UNIX in particular) when an application generates a memory error, a *core* image is written to disk. This image contains the entire memory state of the process, including any secret information. Anyone who can read those files can of course recover that information. Programmers may wish to install exception handlers that detect when the application is about to core dump and zero sensitive information.

5.11 Securing the Server's Private Key

The server's private key is the key to the security of the handshake. In static keying modes, it is used to protect the pre_master_secret directly. In ephemeral keying modes it is used to authenticate the ephemeral key which is in turn used to protect the pre_master_secret. Consequently, although compromise of the server's private key is always bad, it's worse when static keying is used because this compromises all the traffic encrypted under that key.

Static Keying

The most vulnerable situation is when the server has a *static* key pair that is used to establish the master_secret. The most common case is the one we covered in Chapter 3. The server has a single RSA key pair with the public key in its certificate. The client uses that public key to encrypt the master_secret so that only the server can read it. However, an attacker who has the private key can easily decrypt the master_secret and perform all the attacks described previously.

In this situation, compromise of the server's static private key compromises every SSL session that was ever established with that key, because the attacker can recover the master_secret offline at a time of his choosing. In other words, the attacker can mount a purely passive attack. It's trivial to write a program that takes the server's private key and a recorded message trace and outputs the plaintext. In fact, the `ssldump` traces in Chapter 4 were generated using exactly this technique.

Although static keying is mostly used with RSA, SSL also has a number of static Diffie-Hellman modes in which the static DH key is in the server's certificate. These modes are equally vulnerable to this sort of attack. However, most uses of DH in SSL use ephemeral DH. The next section discusses the results of private key compromise when ephemeral keying (as described in Chapter 4) is used.

Ephemeral Keying

Consider the case in which the server's private key is not used for key establishment but only for signature. The server generates an *ephemeral* key pair which is used for key establishment and uses its static key pair to authenticate the ephemeral public key to the client.

When ephemeral keying is used, there are actually two flavors of private key compromise, each with different consequences. Either the static key (used for authentication) can be compromised or the ephemeral key (used for key establishment) can be compromised. First consider the case where the static key is compromised. Knowing the static key is of no value for decrypting traffic because no data is ever encrypted under this key. It's only encrypted using the ephemeral key. However, the static key is also of no use in recovering the ephemeral key.

Recovering a static authentication key isn't useless, however. An attacker who has the server's static authentication key can take over any connection (posing as the server) during the handshake phase. He can then either simply act as the server or mount a man-in-the-middle attack where he poses as the server to the client and the client to the server, shuttling data back and forth between them. Figure 5.2 shows this process.

Figure 5.2 shows this man-in-the-middle attack in action with some of the irrelevant messages trimmed. The attacker copies most of the handshake messages directly from client to server and vice versa, but changes the ServerKeyExchange message to be a signed version of his own ephemeral key rather than the server's. He signs this with the compromised private key. He then decrypts all traffic from the client and forwards it reencrypted to the server, making whatever changes he wants. There is one subtlety here in that the attacker must generate a new Finished message because the handshake hashes that the client sees will be different from those the server sees due to the changed ephemeral key.

So, compromise of a static authentication key allows the attacker to actively attack any given session during the handshake phase but not afterwards. It's interesting to note that the attacker cannot mount this form of attack if client authentication is used. Because the client's signature is over all of the handshake messages, the messages that the attacker sends to the client are different than the messages that the server thinks it's sending to the client. Therefore, the server's signature check will fail.

Even if the server machine is completely compromised and the server's private authentication key is recovered, this knowledge cannot be used to attack any session that's already been established and shut down. This property is called *perfect forward secrecy* (PFS). Note that static key establishment keys do not have this property. However, PFS is obtained only if the traffic keys themselves haven't been stored somewhere. For instance, if the keys are in the session cache, then the attacker could still decrypt the connection. Deleting unused or expired keying material regularly is critical for PFS.

By contrast, if the attacker compromises one of the server's ephemeral keys, he can use it only to attack connections that were protected using that key. For all such connections, the attacker can recover the master_secret and mount the attacks described in Section 5.9.

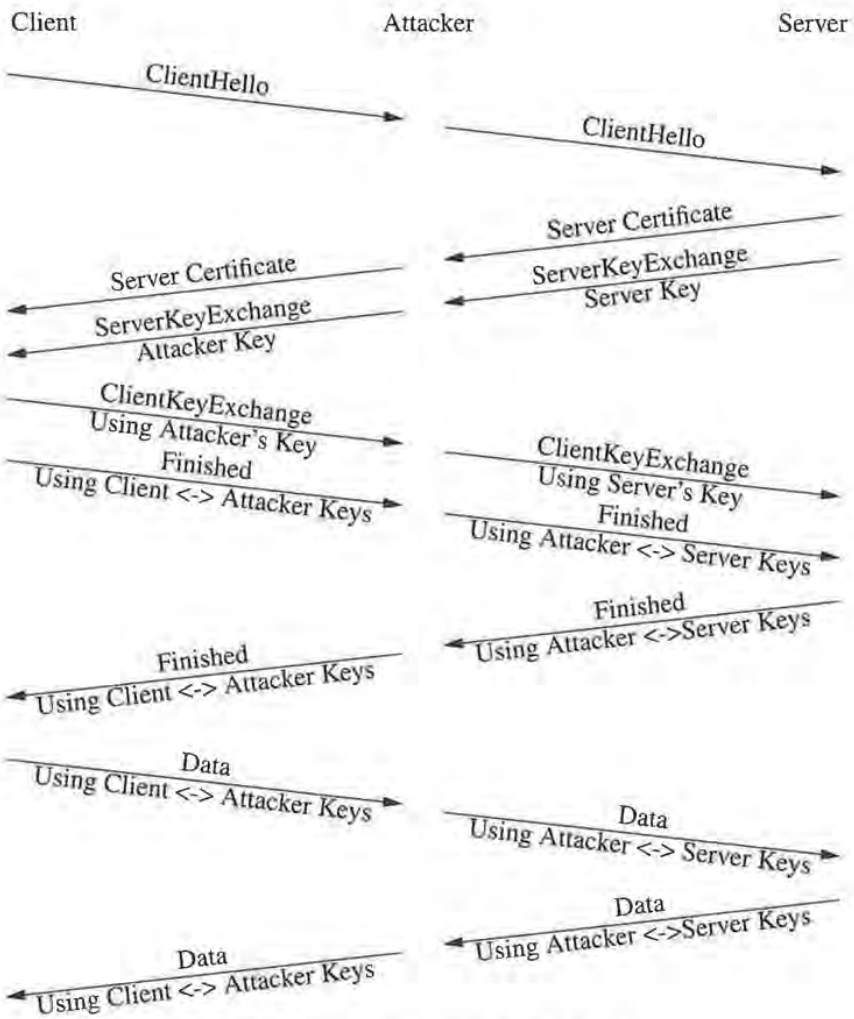


Figure 5.2 Attack using a compromised signing key

How many connections are affected depends entirely on the server's behavior. When ephemeral RSA is being used, the number of compromised connections can be very large. As we discussed in Chapter 4, generating RSA keys is expensive and servers will often generate one ephemeral RSA key when they start up and use that key until they're

shut down, days or weeks later. Consequently, an enormous amount of traffic might well be compromised when one ephemeral key is compromised. Because ephemeral RSA is currently used only with exportable RSA encryption keys (<512 bits), and we know keys of this size to be at the outer edge of breakability, this is an area of some concern.

When ephemeral DH is used, it's fairly common to regenerate the ephemeral key frequently. Some implementations (OpenSSL for instance) have a mode in which they generate a new ephemeral key for every new session. When used this way, compromise of one session has no effect on any other session.

Private Key Storage

Most SSL servers run on general purpose computers. Like most server programs, they use the computer's filesystem to store their permanent data between invocations. This usually includes the server's private key. If the server machine is secure against attack and only trusted users are allowed on the system, this is a reasonably safe practice.

However, having a completely trusted server is difficult. Many organizations can't devote a machine solely to being an SSL server, and even if they could the machine might be compromised or stolen. To partially defend against these eventualities, common practice is to encrypt the private key on disk.

At this point, the astute reader will be wondering how this makes the situation any better. We've encrypted the private key, but now we need to protect the key to encrypt it under. Of course, we could encrypt that key, but then we'd need to protect the new encryption key, and so on. So, encryption alone isn't the solution to the problem.

Of course, we could force the user to memorize the key, but memorizing 128 random bytes is really a lot of work and most users won't accept that job. The usual approach is to use a password or passphrase to generate the encryption key that we use to encrypt the private key. Passwords are easily memorizable and we have a special key derivation function that we use to generate the key from the password.

Password-Based Encryption

There are a large number of ways to transform passphrases into encryption keys. However, the strong ones share a number of common features. The reason that these functions must be different from ordinary key derivation functions is that text usually has a very low *entropy*. That is to say that given part of a word or a phrase, the rest isn't completely unpredictable. (By contrast, cryptographic keys usually are constructed so that knowing part of a key provides no information about the rest of the key.) Several techniques have been designed to counter this problem.

Users are encouraged to choose longer passphrases than the keys to be derived from them. For instance, a 128-bit key can be represented in 16 characters, but passphrases are often 50 or more characters. This allows them to be both memorable and somewhat secure. Nevertheless, passphrases are probably more guessable than keys, particularly since users often choose short passwords. In order to provide some protection even if

poor passwords are chosen, password-based key derivation functions typically use two techniques called *salting* and *iteration*.

Salting includes a public random value as part of the input to the key derivation function. This value is called a *salt*. The salt is simply stored with the key file. Even if two users use the same password, they won't have the same encryption key because they will be using different salt values. This prevents two attacks. First, the attacker might make a table of all common passwords and then could break a password merely by searching that table. Second, the attacker could collect a series of password files and search them all in parallel, thus doing the key derivation stage only once. Neither of these attacks works when salting is used.

Iteration slows down the key derivation function. Password-based key derivation functions are designed to be very slow, in order to make search of the password space expensive. Users won't notice if it takes a half a second to decrypt their private key, but this is a substantial burden if each of a hundred million passwords takes a half a second to check. Typical functions are designed with a basic primitive which can be applied (iterated) an arbitrary number of times, with the output of one stage being used as the input to the next stage. Each iteration takes some time, slowing down the entire process. Again, the iteration count is stored with the encrypted file.

The best-known algorithm for password-based encryption is RSA's PKCS #5 [RSA1993d], which included both the salt and iteration features. Unfortunately, PKCS #5 could generate keys of 160 bits at most, making it useless for 3DES. When Microsoft designed the PFX key storage standard (now PKCS #12 [RSA1999a]), they designed a new function that could generate arbitrary length keys. RSA has since updated PKCS #5 to support arbitrary length keys [RSA1999b].

Figure 5.3 shows a diagram of the salting and iteration process as used in PKCS #5 version 1. The process proceeds in two stages. First, the concatenated password and salt are hashed to produce T_1. This value is hashed repeatedly ($count - 1$ times) to produce T_n, which is the output of the function. T_n is simply equal to $Hash(T_{n-1})$. Obviously this transform cannot produce an output longer than the hash function, which is why it is not used to generate keys longer than the 160-bit output of SHA-1. The PKCS #12 and PKCS #5 version 2 transforms are similar in spirit but not identical and can produce keys of arbitrary length.

Storageless Key Recovery

Imagine that you want to run SSL on a device without any user-specific permanent storage whatsoever. There's no place to store a private key on disk, encrypted or otherwise. It's nevertheless possible for such a device to have a private key if we can derive the private key from the passphrase.

The simplest case is when using discrete log systems such as DH or DSA. Because the private key X is an arbitrary number, it's possible to simply generate it from the passphrase using the password-based KDFs we've just discussed. The group parameters

Ex.1024
Askeladden, L.L.C. / Page 220 of 788

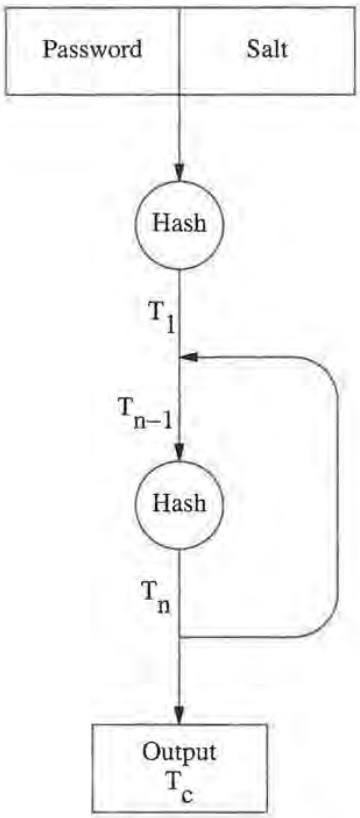


Figure 5.3 PKCS #5 transform

g, p, and q can be common to all device users and thus burned into permanent storage on the device when it's created.

This technique is less convenient for systems such as RSA where the key generation process is expensive, but it's nevertheless possible to use a passphrase to seed the PRNG that is used to generate the RSA key. As long as the passphrase is the same, the same RSA key will be generated each time.

This approach is primarily useful in situations in which a given user might use any number of different machines at any given time, such as kiosk machines or shared terminals. Because certificates can be stored in the clear on the network, the user can simply walk up to the machine and enter his passphrase.

Hardware Cryptography

Another approach to storing private keys is to use a hardware device. The device contains both permanent storage for the private key and some sort of processor capable of doing cryptographic computations. The private key is generated on the device, all private key operations are done on the device, and the private key never comes off the device.

Hardware crypto devices come in two basic flavors, permanent and removable. Permanent devices are intended to stay connected to the machine at all times, either as a card on the system bus or plugged into the back via a SCSI, parallel, or other interface. Removable devices typically are packaged as PCMCIA cards or smartcards (devices about the size of a credit card). A removable device has the obvious advantage that it can be taken away from the machine and locked up for extra security.

Either sort of device can be stolen, so the private keys are usually protected by a password or a *personal identification number* (PIN). The device won't operate correctly unless the user *logs in* with the correct PIN. Sometimes the private key is encrypted as it would be on disk and sometimes the device is simply programmed to forbid using the key unless the PIN is entered. Furthermore, the devices are typically designed to *zeroize* after some (small) number of incorrect attempts. Zeroizing means that they erase all of their sensitive internal state.

For additional security, hardware devices often incorporate a tamper resistance feature. The device is designed to detect attempts to open the case and directly access the hardware. If such an attempt is detected, the device zeroizes its memory. The idea is that even physical possession of the device is useless without the appropriate PIN. FIPS-140-1 [NIST1994b] provides standards for this sort of tamper resistance.

So, hardware devices have two primary advantages over software key files: Because the card zeroizes after some number of failed login attempts, the attacker cannot simply try all possible PINs as he can with the key file, so it's not necessary to choose a really good PIN. (This is good, because choosing good PINs is difficult.)

Second, even if the attacker does get the PIN, he needs physical access to the device. Say the attacker breaks into your server machine and guesses the PIN, but your key is in hardware. He can use the server to decrypt traffic, but once you discover the compromise, you can remove the device or change the PIN and the attack is over. With software, the attacker would have the private key and you would have to generate a completely new key pair.

Fixed devices usually are packaged (and marketed) as *cryptographic accelerators*. They're intended to be able to perform cryptographic operations faster than the host computer. As a consequence, they typically provide both increased security and a performance boost. By contrast, removable devices tend to be very slow (on the order of a few RSA operations per second) and therefore not really usable in a server environment, though they can provide convenient security and packaging for client-side keying material.

Some cryptographic accelerators have both a fixed and removable component. This provides added security because the device cannot be used without the removable key but is as fast as an ordinary fixed device.

Passphrase Entry

You may have noticed by now that both encrypted disk files and hardware devices usually require the operator to enter a passphrase in order to start the server. This is the weak link in the security of the entire system. It's extremely desirable for server systems to be able to restart cleanly without user intervention, but the requirement to enter passphrases to start the secure server makes this impossible.

There are only two known workable approaches: compromise security or compromise unattended reboot. Compromising unattended reboot is easy: require the operator to enter a passphrase to start the server. This means that the machine may restart correctly but an operator is needed on site to start the secure server portions.

Compromising security is similarly easy. You simply place the passphrase somewhere in the system startup scripts and arrange for it to be passed to the server when the server is started. The obvious disadvantage here is that anyone (for instance the administrator or attackers) who can read that file can read the password. Typically, such files are set to have very restrictive permissions, but most operating systems let the administrator read any file.

There's no known way to work around these two choices. Your organization simply has to choose one and accept the consequences.

Biometrics

Biometric authentication measures such as fingerprint identification and retinal scans are often used as secure access control measures. Although there is some doubt as to whether they can be securely implemented, they are nevertheless fairly widely used for access to secure environments and computers.

However, biometrics are singularly inappropriate for protecting private keying material. The measures we've talked about so far all depend on encrypting the private key based on some piece of data the user inputs. This piece of data must be identical each time because it is used to create an encryption key and encryption algorithms are specifically designed so that tiny errors in the key cause large errors in decryption.

However, biometrics cannot be easily used for this purpose because the characteristic being measured is not identical every time. Therefore, it's difficult if not impossible to recreate the same encryption key from the biometric each time. Biometric systems therefore have to keep the key in the clear and simply use the biometric to control access. This is inferior if the device containing the key is ever compromised, because the attacker doesn't need to decrypt the key. If the key is stored on disk somewhere, this is catastrophic.

There are two exceptions to this general rule. First, if the key is stored in trusted tamper-resistant hardware, then a biometric is safe because the hardware will zeroize the key before allowing it to be recovered. This requires having a lot of faith in your tamper resistance! Second, if a passphrase is used in conjunction with a biometric, then the key can still be encrypted using the passphrase and biometrics can be used as a secondary piece of access control. In this case, biometrics add valuable security.

Bottom Line

Understanding your threat model is critical to understanding which technique to use. For instance, if you have strong physical security for your system, you may not need to use as strong security to protect the keys. By contrast, if you're worried that your machine will be stolen or accessed by an unauthorized user, it's critical to protect the keys in hardware in order to limit the cost of compromise. This section gives you the information to evaluate various techniques, but you must choose the one that fits your security model.

5.12 Random Number Generation

The security of an SSL implementation depends completely on the quality of the random numbers it generates. Moreover, both the client and the server need to have strong random number generation. Exactly what is meant by "strong" random numbers isn't that obvious, but it's fairly obvious what it doesn't mean: The "random" number generators that most programming languages provide don't do the job. Special tools are required.

There are two general approaches to generating random data: you can have true randomness provided by some random or nearly random physical process, or you can have pseudo-randomness, which uses some algorithm to produce a stream of random-seeming numbers. In general, good hardware random number generators aren't available on most machines, so programmers settle for algorithmic *pseudo-random number generators* (PRNGs).

Whatever strategy we choose, we want to have at least the following properties.

The stream should be unbiased. The number of zero and one bit should be on average equal.

Output should be decorrelated. Knowing the value of the output of some section of the byte stream should not help you know the value of any other section.

Pseudo-Random Number Generators

The general idea with a PRNG is that you have some function that produces a stream of unpredictable bytes. In theory, this stream should be indistinguishable from a stream of bytes generated by a hardware device, such as flipping a coin.

Of course, this isn't the case because all PRNGs will eventually repeat the stream of bytes. The output of a PRNG is a function of the PRNG's internal state. Because their internal state can be no larger than the memory of the computer they are running on, they must repeat after outputting at most that many outputs. In practice, the internal state of the PRNG merely needs to be sufficiently large that an essentially infinite number of bytes can be generated without the state repeating. 256 bits of internal state is easily plenty to achieve this goal.

Typically, PRNGs are based on digest or encryption algorithms. A good example is described in [Kelsey1999]. There are a number of well-known PRNGs available, and we won't discuss the details here. However, they all have one thing in common—they need to be *seeded* with some amount of secret data. That data is used to set up their internal state. Knowing the seed data amounts to the same thing as knowing the internal state of the PRNG and allows the attacker to predict the bytestream completely. The major problem with using a PRNG is to seed it correctly.

In general, PRNGs are constructed so that they can be seeded with a large amount of low-quality random data and they will *distill* the randomness out of it. For instance, consider some data like the price of a stock from day to day. If the stock is at 85 on Monday, it's very likely that it will be somewhere between 80 and 90 on Tuesday, so the 10s digit isn't that random, but the 1s place is somewhat random and the fractions place is very random. A good PRNG can be seeded with this sort of data and collect all the good randomness out. This makes the job of collecting seed data somewhat easier.

A typical strategy for collecting seed data is to look at some fairly volatile piece of system state, such as network activity, the process table, or the screen, and distill that input into strong randomness. On an active system, this is probably good enough for generating keys to protect single sessions. Several operating systems (Windows 2000, *BSD, and Linux, at least) provide a kernel device that collects system events to continuously seed a PRNG in the background.

For high-security applications, such as private key generation, the implementations often require user input and measure the time between keystrokes or mouse movements. RFC 1750 [Eastlake1994] provides some guidance on collecting seed data.

Although most SSL implementations provide some way of seeding their PRNG from system data, they usually expect the application to provide most of the seed data and provide their own seeds only as a last-ditch effort. Even if you're using a toolkit, you may very well have to seed the PRNG yourself.

It's critical to note that a strong PRNG produces random-looking output even if it's poorly seeded. There are a number of statistical tests that are of some use in determining whether the PRNG you are using is strong. They are of little use in determining whether it has been well seeded.

Hardware Random Number Generators

The other popular strategy is to use a hardware source of randomness. Certain physical devices seem to act randomly at the quantum mechanical level, and this can be used to produce random data. The most obvious example is radioactive decay, which produces a steady stream of signals at random (but predictable on average) intervals. The time between signals (or their intensity over a given time window) can be used to produce random numbers.

Although radioactive decay is random, it's not unbiased, because the output clusters around a mean value. Some processing is usually needed to make sure that the output is unbiased.

> Imagine that you have a random number generator that is random but biased in a simple way, such as an unbalanced coin that returns 60% heads and 40% tails. John von Neumann [Neumann1951] proposed a simple way to unbias it. Collect the output in pairs. Any pair that is the same (e.g., {heads, heads} or {tails, tails}) throw away. Any pair that is different, output the first one (e.g., {heads, tails} becomes heads.) The choice of first or second is arbitrary. It's easy to convince yourself that even if the generator is biased, this trick produces a balanced stream of output, because the probability of the sequence {heads, tails} is the same as the probability of the sequence {tails, heads}.

The easiest way to do this is simply to use the output of a hardware random number generator to seed a PRNG. This overcomes the bias problem as well as ensuring that you don't have to wait for the hardware device to do its thing whenever you want random data.

Recently, Intel Corporation added a hardware-based random number generation feature to their Pentium III processors. The RNG works largely as described above. A hardware random number device based on thermal noise is built into the chip. The data is post-processed through a mixing function based on SHA-1. Initial evaluations [Kocher1999] indicate that the system is secure, provided, of course, that the machine running it has not been compromised by an attacker who is forging RNG output.

5.13 Certificate Chain Verification

We've already established that an unverifiable certificate is little better than no certificate at all. However, merely being able to construct a certificate chain to some root isn't enough to verify a certificate. A number of other checks are required in order to be sure that you're communicating with the entity you think you're communicating with.

Server Identity

As we discussed earlier, the certificate presented by the server must contain an identity which is somehow connected to the expected identity of the server. The expected identity can't be carried over the wire, but must already be known to the client at the time the connection is initiated.

In most cases, when a client wants to connect to an SSL server, the destination is specified by a DNS domain name (see [Mockapetris1987a, Mockapetris1987b]) such as `foo.example.com` or by a *Uniform Resource Locator* (URL) [Berners–Lee1994] that contains a domain name. Thus, the expected identity of the server is the domain name and that must match the identity in the certificate.

Unfortunately, the *distinguished names* (DNs) in certificates aren't really that compatible with domain names. Recall from Chapter 1 that distinguished names are structured as a series of attribute-value pairs whereas domain names are essentially a single string. Moreover, there's no DN attribute for domain names. As a consequence, most CAs use the Common Name attribute to represent the domain name. This approach requires both client and server to be aware of the convention to work correctly.

If you're using X509v3 certificates, which allow extensions, there is another approach. Version 3 certificates can contain a subjectAltName extension which contains alternate name forms for the subject of the certificate. One such form is the dNSName, which is used to represent a domain name. It's permissible to leave the subject DN field blank and use only the subjectAltName to convey identity.

It's important to note that we're not using the server's IP address to denote its identity. The reason for this is that usually the client doesn't get the IP address from a trusted source. Rather, the domain name is trusted but the IP address is looked up using the DNS (a process called *name resolution*), which is easily compromised, as described in [Bellovin1995]. Thus, you can't safely compare the IP address to the IP address in the certificate.

Unfortunately, the original BSD sockets API required that clients specify which host to connect to using the IP address (forcing them to do name resolution themselves). Many SSL APIs mimic this behavior. Thus, because the SSL implementation doesn't know the domain name, it can't check it against the certificate and this check must therefore be done at the application layer. Figure 5.4 provides an example of this procedure using PureTLS. We'll talk more about server identity checking in Chapter 7 when we discuss the design of secure protocols.

```
public static SSLSocket connect(SSLContext ctx,String host,int port)
  throws IOException {
  // Connect to the remote host
  InetAddress hostAddr=InetAddress.getByName(host);
  SSLSocket s=new SSLSocket(ctx,hostAddr,port);

  // Check the certificate chain
  Vector certChain=s.getCertificateChain();

  // Length check
  if(certChain.size()>2)
   throw new IOException("Certificate chain too long");

  // Hostname check (using Common Name)
  Certificate cert=(Certificate)certChain.lastElement();
  String commonName=dnToCommonName(cert.getSubjectName());
  if(!commonName.equals(host))
   throw new IOException("Host name does not match commonName");

  return s;
}
```

Figure 5.4 Java SSL identity checking example

Client Identity

Client identity obviously needs to be verified only when client authentication is used. Otherwise the client is completely anonymous. SSL doesn't really have anything to say about client identities. The application layer is expected to know how to interpret them. That said, the rest of the discussion of certificate chain verification holds as much for client certificate chains as for servers.

Choose Your Roots

All verified certificate chains must eventually terminate in a *root* certificate. This root certificate is configured into the application either by the application developer or by the user (or his system administrator.) For instance, most popular browsers come with upwards of 25 root certificates configured into the application.

These roots must be trusted to accurately certify the identity of the customers whose certificates they sign. If they don't, then the system becomes untrustworthy. For instance if a root provides an attacker with a certificate bearing Alice's name, then the attacker can impersonate Alice when you attempt to make a connection to his server.

In order to evaluate whether a root is acceptable, you need to evaluate the care with which they authenticate users (do they require positive physical ID, etc.) and the care with which they protect their keys. Neither Netscape nor Microsoft has published the rules that they use to determine which roots are compiled into their browsers. However, both Netscape and Microsoft allow the user to install their own roots. This is typically done by having the user dereference a Web page with a specific content type labelling it as a root certificate. Unless carefully handled, this practice represents a security vulnerability. The roots must be obtained from a trusted location over a trusted link or an attacker can convince the user to install a fake root. One common approach is to distribute a root *fingerprint*—a digest of the root key—offline and then allow the user to check that against the root when installing it.

Certificate Chain Depth

For management reasons, it's often convenient to use certificate chains that contain more than one certificate. This allows for distributed certification. For instance, imagine that Widgets, Inc. wants to issue certificates to all of its 20,000 employees at ten different locations. It might be convenient to have each location issue its own certificates.

One way to do this is to have a single root certificate for all Widgets certificates. That root would then sign the certificates for each location's CA. The location CA would then sign certificates for employees. Figure 5.5 shows a certificate hierarchy that uses this model.

In this example the Widgets Root CA certifies two location CAs, one for the Palo Alto office and one for the Seattle office. Each office has two employees, who are certified by their local CAs. Alice and Bob both work in Palo Alto and are certified by the Palo Alto CA. Charlie and Dave both work in Seattle and are certified by the Seattle CA.

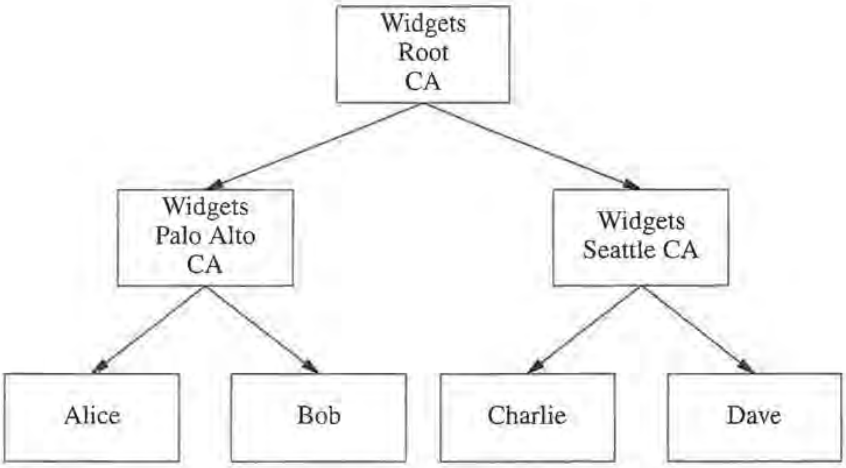


Figure 5.5 A sample certificate hierarchy

Realize that we now need to distinguish between two different classes of entities: those whom we trust to vouch for others and those whom we do not. For instance, in the above example, the certificate for the Widgets Palo Alto CA means "The Root CA asserts that this key belongs to the Widgets Palo Alto CA and this CA may in turn issue certificates." However, the certificate for Alice means "The Root CA asserts that this key belongs to Alice." The Widgets Palo Alto CA is trusted to sign certificates, but Alice is not. If Alice were allowed to sign certificates, she could create herself the certificate chain shown in Figure 5.6 and impersonate Bob.

The distinction between the two types of certificates can be made either implicitly or explicitly. Before X.509v3 certificates, the only real choice was to make it implicitly by controlling the *depth* of the certificate chain. That is to say, they would declare that certificate chains from their roots would be no more than n certificates long. In the current example, n would be 2: the location CA (signed by the root), and the user.

The primary problem with this approach is that it doesn't allow implementations to follow a single global policy. One CA might allow a depth of 2 and one might allow a depth of 1. Setting one's implementation's maximum depth to 2 allows impersonation attacks by individuals certified by the second CA, and setting it to 1 makes it impossible to verify certificate chains issued by the first CA. In practice, nearly all CAs use only one level deep certificate chains, and it's easy to add the next-level CAs as roots for those CAs who use longer certificate chains.

The newer approach is to use X.509v3 certificate extensions to indicate whether a certificate corresponds to a CA or not. In particular, PKIX (the IETF X.509 certificate profile described in [Housley1999a]) requires that the Basic Constraints extension must be present in all CA certificates. The Basic Constraints extension specifies whether or not a certificate belongs to a CA and how deep the certificate chain may extend from that path.

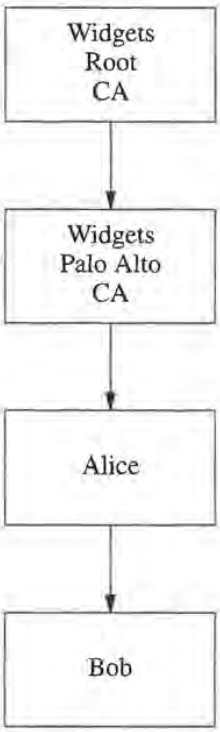


Figure 5.6 Impersonation by extending the hierarchy

Unfortunately, many currently available certificates and CAs do not follow PKIX and there are a large number of legacy certificates that don't. Consequently, if you have a closed system, you should try to use Basic Constraints, but otherwise you're pretty much stuck using hard-coded length constraints. Software that follows PKIX will reject otherwise legitimate CAs because they don't have Basic Constraints.

Key Usage Extensions

The X.509 key usage extensions restrict the purposes for which a key may be used. X.509 defines 8 key usage values that are combined into a bitmask of allowed operations. The values are: digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly, decipherOnly. No browser requires the keyUsage extension to be present but Netscape and Microsoft both

check them if they are. Figure 5.7 shows the required bits for the various types of certificate for each type of operation.

Usage	Netscape	Microsoft
SSL Server	keyEncipherment	keyEncipherment
SSL Client	digitalSignature	keyEncipherment
Certificate Signing	keyCertSign	keyCertSign

Figure 5.7 Use of X.509 extensions

Note that Microsoft and Netscape expect different bits to be set to indicate that client authentication is permitted. The confusion as to what bit to use for client authentication derives from the fact that SSL client authentication involves signing a string that the client never sees. Arguably, the client's key is being used to authenticate the key exchange, so keyEncipherment is more appropriate. Netscape took the natural approach and used digitalSignature.

To add to the confusion, Netscape also defined their own certificate usage extension: netscape-cert-type. Documentation of Netscape's certificate practices can be found at [Netscape1995a]. The netscape-cert-type extension is a bit string. The relevant values are SSL Client (bit 0), SSL Server (bit 1), and SSL CA (bit 5). There are several other values that are used for S/MIME certificates and for something Netscape calls *object signing*. It's not clear how to behave if both extensions are present and disagree. In practice, however, this is unlikely to happen.

Certificates for root CAs are compiled into the browser. Certificates for intermediate CAs should contain basicConstraints. Netscape will also accept certificates with one of the CA bits in netscape-cert-type set as an intermediate CA. Many other implementations, including older versions of OpenSSL, are far less picky and will take any certificate as a CA provided they can construct a chain. As a rule, modern implementations should either manually limit certificate chain length or insist on some indication in the certificate that the certified entity is a CA.

Not all certificates contain the keyUsage extension. Most browsers will accept such certificates for either client or server but not for an intermediate CA.

Other Certificate Extensions

Netscape also defined a number of other certificate extensions, as shown in Figure 5.8. These extensions are described in [Netscape1999a]. Most of these extensions are URLs that the verifier can dereference to get information about the certificate, such as the policy under which it was issued and its revocation status. These aren't that widely used because revocation is actually quite uncommon. There is also a netscape-ssl-server-name extension. This field can be used instead of the Common Name to identify the server to which the certificate belongs. Thus, there are at least three ways to put the server's name into a certificate, the Common Name, the subjectAltName extension, and

the netscape-ssl-server-name extension. In general, you should avoid using the netscape-ssl-server-name since it's completely nonstandard. Instead, use the Common Name for maximum compatibility or the subjectAltName for newly designed systems.

Extension	Purpose
netscape-base-url	The base for all URLs in the certificate—allows the other URLs to be shorter.
netscape-revocation-url	Contains a pointer to a URL that will return revocation information.
netscape-ca-revocation-url	Used to check revocation status of CA certificates.
netscape-cert-renewal-url	The URL that the certificate owner can use to renew the certificate.
netscape-ca-policy-url	The policy under which the certificate was issued, in text form.
netscape-ssl-server-name	The name of the SSL server.
netscape-comment	A comment to be displayed to the user.

Figure 5.8 Other Netscape certificate extensions

5.14 Partial Compromise

The security of SSL is completely dependent on the cipher suites used. Each cipher suite specifies four algorithms: a digital signature algorithm, a key establishment algorithm, a data encryption algorithm, and a message digest algorithm. Although these algorithms are not orthogonally specified, in general there are multiple choices for each type of algorithm and these choices vary in strength. Moreover, for both digital signature and key establishment algorithms, multiple key sizes are possible, and each key size represents a different security level. It's quite likely that one algorithm will be broken while the others remain secure. This section considers the effect on the security of the overall system of such a compromise.

When we talk about algorithm strength, we need to consider two factors. First, we can place an upper bound on the strength of an algorithm based on the best known attack against that algorithm. By that metric, RC2-40, RC4-40, DES, and RSA, DH, and DSS with 512-bit keys are already dangerously weak. That is to say that with a plausible amount of effort, an attacker might be able to recover a single key for such an algorithm.

The second factor is the possibility that an algorithm might be completely broken sometime in the future. That is to say, it might become possible to recover an arbitrary number of keys, regardless of key length, with a trivial amount of effort. This has happened to other algorithms. Most of the ciphers used in SSL have undergone fairly extensive analysis, so it's widely believed that we have some idea of their security properties. Nevertheless, the possibility that an algorithm might be catastrophically broken remains as a worst-case scenario.

Digital Signature Algorithms

SSL uses digital signatures for three purposes:

1. To authenticate certificates.
2. To sign ephemeral keys.
3. To sign CertificateVerify messages for client authentication.

Total compromise of a digital signature algorithm compromises all of these uses and renders the protocol completely vulnerable to active attack. The situation here is worse than compromise of a single-server private key, because compromise of a signature algorithm means that all certificates signed with that algorithm are also vulnerable. This means that any SSL connection where one accepts the compromised algorithm is subject to active attack.

Partial compromise of an algorithm is also possible. In some sense this has already happened with RSA and DSS keys of 512 bits or less, since recovering such keys is within the current capabilities of a wealthy and dedicated attacker. The effect of such a compromise depends on which key is compromised.

> Note that DSA keys can also be compromised due to incorrect usage. DSA computations require the generation of a random quantity k for each signature. This number must be kept secret. If it is revealed, or simply not randomly generated, then the DSA key can be totally compromised.

Compromise of a CA key leads to total compromise of all agents who trust that CA. Although the individual private keys corresponding to the certificates that the CA has issued are still safe, this is irrelevant because the attacker can mount an active attack and impersonate any client or server merely by signing himself a certificate with the appropriate CA key.

Compromise of a server's key allows that server to be actively attacked. We've already covered this situation previously in this chapter.

Compromise of a client's key allows the attacker to pose as the client. Note that this does *not* permit the attacker to take over connections that the client creates, but it does permit him to initiate his own connections and client authenticate as the client.

Key Establishment Algorithms

Total compromise of a key establishment algorithm means that the attacker can recover the master_secret for any connection encrypted with that algorithm and can thus mount any of the attacks described in Section 5.9. Similarly, if a key establishment key is compromised (even if the algorithm is strong), then the attacker can recover the master_secret for any session that uses a key established using that algorithm and can similarly mount any attack that involves knowing the master_secret. This is particularly

bad when the key is a static key because that means that all traffic to a given server can be broken. Figure 5.9 shows a summary of the consequences of the exposure of various kinds of keys.

Key Type	Effect of Compromise
CA	Attacker can impersonate to all agents that trust that CA.
Key Establishment (Static Keying)	Total compromise of all traffic to the server.
Server Authentication (Ephemeral Keying)	Attacker can mount an active attack on any session at handshake time.
Key Establishment (Ephemeral RSA)	Total compromise of all traffic exchanged with this key (typically hundreds of connections).
Key Establishment (Ephemeral DH)	Total compromise of all traffic exchanged with this key (typically one connection).
Client Signature Key	Attacker can impersonate this client.

Figure 5.9 Summary of effects of private key compromise

Revocation

As we saw in Chapter 1, when a private key is compromised, some CAs issue *Certificate Revocation Lists* (CRLs) which can be checked to see if a certificate has been revoked. The issue then becomes how to get the CRLs to the end users. Some protocols such as S/MIME allow CRLs to be carried along with certificates in messages. SSL does not allow this. Agents must get CRLs in some undefined way, which often means not at all. This makes compromise of keys particularly problematic for SSL.

After Goldber and Wagner [Goldberg1996] showed that Netscape's PRNG was weakly seeded, servers that had used that PRNG to generate their private keys all had to generate key pairs. Since CRLs were not widely deployed, this led to a large number of certificates for weak keys floating around with no good way to take them out of service other than waiting for them to expire.

Encryption Algorithms

The broadest range of algorithm strengths in SSL is found for the encryption algorithms. At the strong end of the spectrum, SSL supports 3DES which has an effective key length of 112 bits under the best-known attacks. Moreover, DES has undergone far more analysis than any other public cipher, so it seems unlikely that a new good attack against 3DES will be discovered. At the weak end, SSL supports 40-bit RC4, which is breakable by attackers with minimal resources—a few high-end PCs.

Imagine that the attacker manages to recover the encryption key for one side of the channel (say the client_write_key). Obviously he can now read all traffic encrypted with that key (traffic sent from the client to the server), but the scope of the attack is limited to the confidentiality of that data.

The client_write_key and server_write_key are independently derived from the master_secret, so having the client_write_key is insufficient to recover the server_write_key. Similarly, you can't backtrack from any given encryption key to the master_secret because the master_secret is much larger than the encryption keys it generates, so there is a many-to-one relationship between master_secret and encryption keys.

Similarly, confidentiality and message integrity are independently provided. Even if the attacker breaks the encryption, he can only mount a passive snooping attack. That is, he can recover the encrypted data. He can't use this information to forge traffic. For that, he has to break the MAC algorithm.

The effect of encryption algorithm compromise can be limited by occasionally rehandshaking. Even if no new key exchange is performed, just refreshing the random values will result in new encryption keys and force the attacker to re-attack the keys in order to read the new traffic. Note that because of CBC rollover, if you're using a block cipher it's important to rehandshake every 2^{32} blocks or so. Note that doing a new key exchange while rehandshaking adds little value, because most attacks on key exchange algorithms break the entire private key and not just one exchange.

Digest Algorithms

Compromise of a digest algorithm can mean a number of things. In general, because message digests destroy information (the message space is larger than the digest space), it is impossible to recover a message given the digest. However, it is not known to be impossible to produce a message that digests to a given value, although it is not known how to do so for any currently used digest. It would totally compromise such a digest if such a method were known.

The effect of being able to produce messages with a given digest would be very bad, because it would be possible to forge certificates. The attacker would start with a certificate that had a valid signature and attempt to generate a new certificate with a name and public key of his choice that digested to the same value.

Imagine that you have certificates signed with a safe digest but that both sides accept a broken digest for message protection. This does not necessarily lead to system compromise. First, SSL uses a combination of digests (both SHA-1 and MD5) to protect the handshake, so the attacker would need to have broken both in order to actively attack the handshake. Thus, you would probably still have confidentiality, even if no message integrity.

Moreover, HMAC is believed to be immune to some forms of compromise of the underlying digest [Krawczyk1996], so a partial compromise of a digest might not even lead to message integrity problems. For instance, collisions in a digest do not lead to a weakness in HMAC based on that digest.

One other case is worth mentioning. Because DSS signatures are computed over a SHA-1 digest only, if SHA-1 were broken, then any ephemeral key exchanges signed with DSS would be subject to active attack, because the attacker could forge the ServerKeyExchange. Figure 5.10 summarizes the effects of symmetric algorithm compromises.

Ex.1024
Askeladden, L.L.C. / Page 235 of 788

Algorithm	Effect of Compromise
Any digest algorithm	Attacker can forge certificates for CAs using that algorithm.
SHA-1	Attacker can mount an active attack on anything protected with DSA.
Any encryption algorithm	Confidentiality lost. Integrity intact

Figure 5.10 Summary of effects of symmetric algorithm compromise

The Weakest Link Principle

Although SSL provides a check on the handshake to prevent man-in-the-middle downgrade attacks, this does not provide total protection. Because all security is based on the master_secret, no SSL implementation can ever be more secure than the weakest key establishment mechanism that it supports. Imagine that a client supports RSA but is willing to accept certificates signed with 512-bit RSA keys. It's also willing to accept certificates with longer keys, but it doesn't distinguish between them.

An attacker who can compromise a 512-bit RSA key can thus attack connections such a client makes even though the client would be willing to support a longer key, because the client will accept a handshake made with a weaker key. If the attacker can break a key establishment (or signature) algorithm that one of the parties accepts, he can mount a man-in-the-middle attack. Note that if client authentication is required, then the attacker *also* needs to be able to forge a signature that the server will accept.

This concern does not apply to encryption algorithms. The check in the handshake protects attackers from downgrading the handshake to support a weaker encryption algorithm as long as the key establishment mechanism is strong.

5.15 Known Attacks

Although no good attacks are known on SSL itself, a number of attacks are known that are effective against specific implementations. The next several sections describe these attacks and the required countermeasures.

Of particular note are *timing cryptanalysis* and the *million message attack*. Timing cryptanalysis was developed in 1996, after SSLv2 was widely deployed and after SSLv3 had been published. The million message attack was developed after SSLv3 had seen wide deployment and TLS had been published. Neither of these attacks is known to have been used against production servers, and it's not even known that they are practical, but conscientious implementors should at least be aware of them and the appropriate countermeasures.

5.16 Timing Cryptanalysis

In 1996, Paul Kocher publicized a technique which he called *timing cryptanalysis*. The technique relies on the observation that cryptographic operations take varying amounts of time to complete depending on the keys and data that are being used. The attack is actually a general technique which must be tailored for the target cryptosystem. Kocher provided examples for RSA, DH, and DSS in his original paper [Kocher1996b].

Attack Overview

Kocher emphasized timing attacks on public key cryptosystems in his paper. To mount the timing attack, the attacker needs to time the victim using his private key. With enough samples (roughly 2500 for a 1024-bit RSA key) you can recover the entire key. The details of the attack are out of the scope of this book. Suffice it to say that Kocher describes how to mount timing attacks on RSA, DH, and DSS.

Naturally, there is some noise in these measurements due to other system activity that interferes with timing. Kocher describes how to filter out noise in the measurements, but this technique requires more samples. To be maximally effective, the attacker should be able to get accurate times for the operations in question, with as little noise added as possible.

For maximal convenience, the attacker would be on the same machine as the victim and be able to time the victim directly. Such favorable circumstances are rarely available, but network protocols such as SSL often allow the attacker to determine the same information by observing network traffic.

Applicability

Recall that the server's private key is the most valuable key to attack. Is it potentially subject to timing analysis? Yes. Consider first the static RSA case. The time between the receipt of the ClientKeyExchange and the sending of the server's ChangeCipherSpec message is primarily divided between three primary tasks: decrypting the pre_master_secret, using the key derivation function to create the keys, and processing the client's Finished message.

Because the time taken by key derivation and Finished verification is mostly constant—and, more importantly, random with respect to the server's private key operation—it can be treated as noise and filtered out with enough samples. After these are factored out, the attacker has some idea of the time taken for the private key operation and can use this to attack the server's private key.

Ephemeral keying is similarly vulnerable. Although the attacker can't get enough samples to attack the ephemeral key, he can attack the static signing key. The time between the server's receipt of the ClientHello and sending of the ServerKeyExchange is dominated by computing the signature in the ServerKeyExchange message. If DH is being used and a new DH key is generated for every session, this provides a source of random noise, but this can be filtered out. Ephemeral RSA typically reuses the same ephemeral key for every session, making this mode particularly vulnerable.

Note that this attack does not require the attacker to be active. He can collect all this information merely by monitoring the victim's network connection. The attack can be mounted with fewer samples if the attacker mounts an active attack and can therefore choose the ciphertexts that the victim decrypts. The timing attack does require the attacker to know a fair number of details of the victim's SSL implementation, but these are fairly easy to come by, either by downloading the source (e.g., OpenSSL) or reverse engineering a binary.

No reports of hostile timing attacks being mounted against operational servers have ever been published. In any case, a number of SSL implementations contain countermeasures to make this attack difficult.

Countermeasures

The most obvious countermeasure is simply to slow down all operations by a random time. This is distasteful for performance reasons and it also doesn't work that well. Precisely because the noise is random, if the attacker collects enough samples the noise can be filtered out using basic signal processing techniques.

A second approach is to make all operations take constant time. This is a feasible approach in hardware but not really in software. And again, it has undesirable performance consequences.

The most popular approach is to use *blinding* techniques. The victim transforms the data to be signed (or decrypted) in such a way that the private key operation now operates on data unknown to the attacker. The victim then *unblinds* the result to obtain the original result that would have been obtained if no blinding had been used.

5.17 Million Message Attack

In 1998, Daniel Bleichenbacher published an attack which came to be known as the *million message attack* [Bleichenbacher1998]. The attack is actually an attack on RSA using PKCS #1, but all versions of SSL that use RSA are susceptible to it. The effect of this attack is to allow the attacker to recover the pre_master_secret, for a given session. We assume that the attacker has captured the ClientKeyExchange message containing the relevant encrypted pre_master_secret. He does this by sending a series of *chosen ciphertexts* to the server. That is, he generates a series of messages and observes the server's responses. From this, he can recover the pre_master_secret.

Attack Overview

The attack proceeds as follows. The attacker generates a series of messages based on the encrypted pre_master_secret and then chooses a series of integers s and computes

$$c' = cs^e \bmod n$$

where c is the encrypted pre_master_secret and e is the RSA public exponent.

The attacker then uses the victim's server as an *oracle*: he probes it with these messages and examines the responses. The attacker initiates a handshake with the server and sends c' in his ServerKeyExchange message. When the server decrypts c' he recovers a new message m'. m' can be shown mathematically to be ms where m is the original message. However, since the server doesn't send us m', this information isn't directly useful. However, it can still be used to attack m.

The key to the attack is that the attacker needs to be able to use the victim server as an oracle to get information about m'. He can use this information to get an accurate picture of m. The information that this attack uses is whether m is correctly PKCS #1 formatted.

Recall that PKCS #1 (described in Chapter 1) contains as the first 3 bytes a zero and a block type indicator. For properly formatted public key encrypted data, these bytes will be `00 02`. About one in 2^{16} s values will yield an m' that is properly formatted. Figure 5.11 shows the entire padded pre_master_secret for a TLS connection, including the version number.

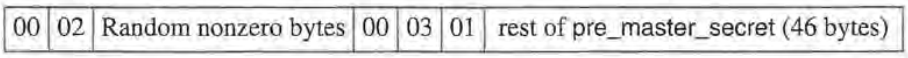


Figure 5.11 TLS Formatted pre_master_secret

The attack proceeds by finding a series of s values that produce PKCS #1 conforming plaintexts. These values can be used to narrow down the range of the original message m until it is completely determined. The exact details of how this is done are out of scope for this book, but suffice to say that around 2^{20} messages are required. (This number is around a million, hence the name *million message attack*.)

Applicability

The attack depends on the attacker being able to determine the difference between correctly and incorrectly formatted messages. Conveniently for the attacker, implementations written before this attack was discovered sent an alert (and then closed the connection) when they encountered incorrectly formatted messages. This makes it very easy to mount such an attack against such servers.

Note that SSL formatted pre_master_secret contains three pieces of verifiable information, not one. Aside from the PKCS #1 formatting, the pre_master_secret must be 48 bytes long and the first 2 bytes of the pre_master_secret must be the version number. The probability of a randomly chosen message getting all three of these right is less than 2^{-40}, so an implementation which rejected all of these errors *identically* would be impractical to attack in this fashion.

The key word in the above sentence is *identically*. Some implementations have been experimentally determined to reject all of these cases but to behave differently. Such implementations are still vulnerable to this attack.

There are no known instances of this attack being used against live servers, but Bleichenbacher says that he has tried it experimentally against one of his own servers and successfully recovered the pre_master_secret [Bleichenbacher1999].

Countermeasures

The most widely adopted countermeasure (and the one recommended in RFC-2246) is to make all three checks but not to send any alerts. If an error is detected, the implementation fills the pre_master_secret with random data and proceeds with the handshake as usual. The server will throw an alert when the client's Finished message is received and decrypts to garbage, but this is the behavior it would exhibit if m' were correctly formatted but the pre_master_secret were wrong.

Note that setting the pre_master_secret to a constant value will not work. The attacker could then use that value as the pre_master_secret and check to see if the Finished message is accepted or not.

A stronger fix is to replace PKCS #1 with a different padding algorithm that is sensitive to any damage to the plaintext. If the plaintext is padded in this way, then all of the probe messages used by this attack will automatically be rejected because they fail integrity checks. The most widely accepted example, *Optimal Asymmetric Encryption Padding* [Bellare1995] has been included in PKCS #1 version 2 [Kaliski1998a]. Unfortunately, OAEP padding is incompatible with PKCS #1 version 1 padding, so there's little incentive for SSL to move to PKCS #1 version 2.

5.18 Small-Subgroup Attack

Under certain specialized conditions, it is possible for an attacker to compromise a static Diffie-Hellman key that is used for a large number of key exchanges. This attack is generally called a *small-subgroup attack* [Lim1997].

Attack Overview

The attack arises out of an optimization that can be made to improve DH performance. Recall that the DH shared secret is computed by: $ZZ = Y^X \bmod p$. The smaller X, the faster this computation is. Choosing too small an X can compromise security (X should be about twice as long as the largest key you want to generate) but Xs a lot smaller than 1024-bits (a typical size for p) are still quite secure.

The choice recommended in the ANSI X9.42 standard [ANSI1998] is to create p such that $p = jq + 1$ where q is a large prime. X is then chosen to be smaller than q. g must be chosen so that it generates a group of order q.

Under these circumstances, the attacker can obtain information about X if he chooses his public key appropriately. He chooses his Y such that $Y \ll q$. If he can convince the victim to send him ZZ, he can obtain information about X. With enough samples, he can recover all of X. This may take as little as a hundred trials. It turns out that it is not necessary for the victim to return ZZ. Simply returning information about decryption success or failure is sufficient.

Applicability

Small subgroups are only an issue for a very small number of systems that use Diffie-Hellman keys for multiple transactions and also use small exponents (X). Because static DH is almost unused, this in practice means only those implementations that use ephemeral DH keys for multiple transactions. Of those, only some use small Xs.

Countermeasures

There are a number of countermeasures available. The simplest is to always use a new key for every transaction. This also has the advantage of providing perfect forward secrecy.

If you must use a static DH key, then there are still several countermeasures available. The simplest is to choose a long exponent, thus avoiding the vulnerability. Choosing p such that $p - 1 = 2qj$ reduces the risk so that only a single bit of the private key is leaked. This is sufficient for most purposes. As a special case, if $j = 1$ then p is referred to as a *strong prime*. A more complicated technique called *compatible cofactor exponentiation* is described in [Kaliski1998b].

5.19 Downgrade to Export

An attacker who can factor 512-bit RSA keys can downgrade a client/server pair into 512-bit ephemeral RSA mode even if both sides support 1024-bit RSA keys. The following attack was described in [Moeller1998]. The attack requires both sides to also support export mode and the attacker to be able to factor the server's ephemeral RSA key.

Attack Overview

The attack, shown in Figure 5.12, requires the attacker to act as a man-in-the-middle. He intercepts the client's ClientHello and removes all but the export ciphers. He transmits the modified ClientHello to the server which chooses an export cipher and sends a ServerKeyExchange containing his ephemeral RSA key. The handshake sniffs the ClientKeyExchange. Because he has factored the server's ephemeral RSA key, he can recover the master_secret. He then forges new Finished messages, one to the server reflecting the modified ClientHello and one to the client reflecting the original ClientHello. Once the handshake has completed, the server can simply sniff traffic. Because he has recovered the master_secret the connection is completely compromised.

Applicability

This attack requires the client and server to support strong cipher suites but be willing to negotiate export cipher suites. If both sides support only strong cipher suites, then the attack cannot succeed. If either side supports only export cipher suites, then they would

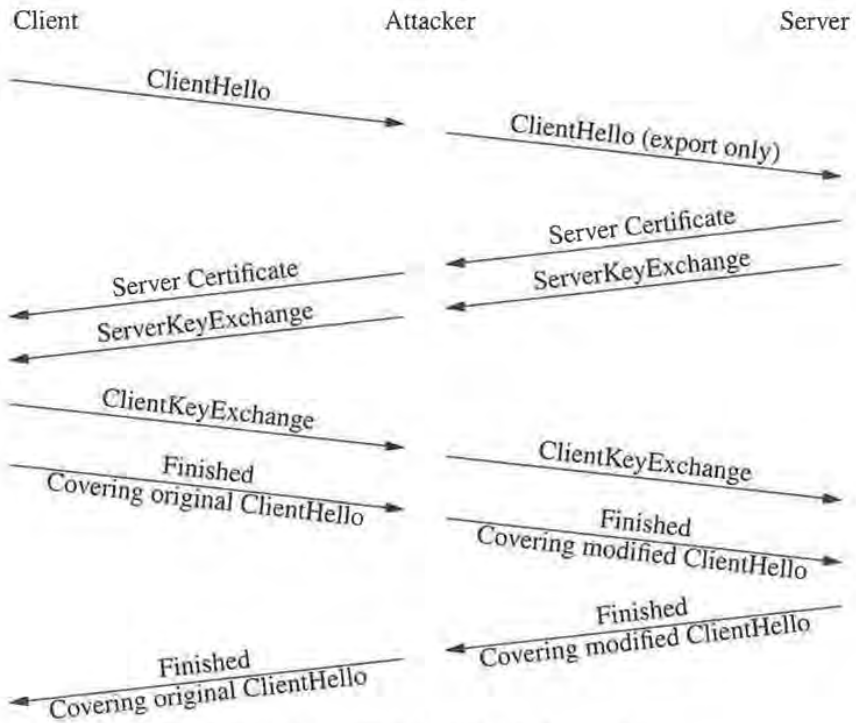


Figure 5.12 Downgrading to 512-bit ephemeral RSA

negotiate an export algorithm anyway. Moreover, the attacker must be able to factor the server's ephemeral RSA key during its lifetime. Thus, servers which frequently regenerate their ephemeral RSA keys will be safe. However, as factoring technology gets better, the safe lifetime becomes shorter.

Countermeasures

The safest countermeasure would be to fix the protocol so that the signature over the ServerKeyExchange covered more of the handshake, including the ClientHello. This would allow the client to detect that the attacker had modified the ClientHello. This change would also serve to strengthen SSL against similar attacks based on downgrading key exchange algorithms.

However, it's possible to protect against this attack without a protocol change. The ephemeral RSA cipher suites are doubly outdated. They have been replaced with the EXPORT_1024 cipher suites and strong ciphersuites can now be exported. Thus, there is no longer any real reason to support them, except for interoperability with old implementations.

In most cases, users can safely configure their browsers to support only strong

cipher suites. Nearly all servers support 1024-bit RSA so this should not cause an interoperability problem. Server administrators can protect themselves by frequently regenerating the ephemeral RSA so that it cannot be factored during its lifetime.

A Similar Attack

A similar attack is described in [Schneier1996b]. If the client supports both DH and RSA and the server supports DH, the attacker can tamper with the ServerHello to make the client believe that the server's ephemeral DH key is actually an RSA key. If the client is incautious, it will interpret the DH prime p as the RSA modulus and the DH base g as the RSA exponent. It's easy to recover the pre_master_secret from the resulting ClientKeyExchange.

For this attack to succeed the client would have to be quite careless. First, the server would likely use a DH modulus of longer than 512 bits, in which case the client should abort the handshake when it sees that the "modulus" is likely longer than 512 bits. This check is required by SSL to enforce export constraints. Second, the client would have to ignore the fact that the ServerKeyExchange contained more data than it should since it contains the server's public key Y. The moral of the story is that clients and servers should carefully check all protocol values.

5.20 Summary

This chapter has provided a description of the security properties of SSL. We start with some general guidelines for using SSL securely and then provide some detailed description of various potential attacks and how they are countered.

The master_secret is everything. Compromise of the master_secret compromises the entire protocol. An attacker who knows the master_secret can read traffic and forge messages at will.

Protect the server's private key. In the normal RSA mode and static DH mode, compromise of the server's private key leads to compromise of the master_secret. Even if ephemeral modes are used, an attacker who has the server's private key can pose as the server and attack traffic in real time.

Good randomness is essential. If either party doesn't use secure random number generation, the protocol is at risk. Good random numbers are especially important when generating long-term secrets such as private keys.

Just having a certificate isn't good enough. If you care about the identity of the other side, you must not only check that it has a valid certificate but that the certificate matches the expected identity of the other side.

Use good enough algorithms to protect your data. SSL supports a variety of algorithms with a variety of keylengths. Choose an algorithm such that the cost of breaking it is substantially higher than the value of your data.

There are no good known attacks on SSL. However, there have been a number of proposed attacks on certain implementations. Careful implementation can protect you from these.

6

SSL Performance

6.1 Introduction

This chapter discusses the performance properties of SSL. As with previous chapters, we divide the discussion into two primary parts. The first part discusses the performance properties of SSL in general terms and makes fairly broad observations about performance of various modes under various circumstances. The second part of the chapter covers the same territory in detail, using instrumented network traces and profiled implementations to show exactly where the time is going.

6.2 SSL Is Slow

The most common complaint that server administrators make about SSL is that it's slow. This criticism is not without some justice: Depending on the protocols being used, the server hardware, and the network environment, SSL connections can be anywhere between 2 and 100 times slower than ordinary TCP connections.

This performance cost translates directly into increased operational costs. Internet servers are usually heavily loaded even when insecure protocols are used. Computationally intensive security protocols necessarily mean that more server machines must be purchased and operated to maintain the same quality of service. This transition is particularly difficult when the original service ran on a single server machine. If security necessitates operation of multiple machines, this may mean arranging to share data between servers where this was previously unnecessary.

Unfortunately, some performance degradation is largely unavoidable. Cryptography is computationally expensive and the difference between simply writing data to the network and encrypting it before sending is usually nontrivial. However, it is often possible to minimize SSL's performance impact with careful systems design and configuration.

Ex.1024
Askeladden, L.L.C. / Page 245 of 788

6.3 Performance Principles

Before discussing SSL performance, we'll first discuss some general principles of system performance, with a special emphasis on network protocols. This provides a background for understanding the contribution the various pieces of SSL make to the performance of the protocol as a whole.

Amdahl's Law

The most basic rule of performance tuning is Amdahl's Law, described in [Hennessey1996]. Roughly speaking, the speedup resulting from an optimization is equal to the improvement of the speedup multiplied by the fraction of the CPU time spent in the code sped up, as shown in Figure 6.1.

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Figure 6.1 Amdahl's law

Therefore, it is of crucial importance to determine which sections of the system consume the most time, because they represent prime targets for optimization. Once we have a picture of where the time is going, our strategy becomes clear: try to improve the performance of these sections of the system, either by making them run faster or using them less often.

As a corrolary, it should be clear that even very expensive operations aren't worth speeding up unless they also happen frequently. In the case of a network protocol, this means that we're primarily concerned with operations that happen with every transaction or protocol message. We'll ignore time consumed by system startup and shutdown, even though it may represent the equivalent of hundreds of transactions, because it's insignificant over the life of a server.

The 90/10 Rule

Amdahl's law wouldn't be very useful if all parts of a system consumed equal amounts of time. In such a system, we'd have to optimize a significant fraction of the system in order to see any improvement. This is very infrequently the case. Rather, most of a system's time is spent performing a few tasks.

A useful rule of thumb, the *90/10 rule* states that 90% of the execution time of a program is spent in 10% of the code. Combining the 90/10 rule with Amdahl's law, we can see that optimizing that section of the system would contribute greatly to the performance of the system. Focusing on on the other 90% is almost useless.

A particular case of this rule is what is termed a *bottleneck*. A bottleneck occurs when a system is performing a lot of different operations, but they all have to go through some central point or operation. Any time you see congestion before a certain point and free flow afterwards, suspect a bottleneck.

The Cost of I/O

Program performance tuning is normally accomplished with the aid of profilers (software that determines the amount of CPU time consumed by pieces of a program) and instrumented software. This allows the person tuning the code to identify sections of code that consume large amounts of CPU. Systems that perform extensive input/output (I/O) in general, and network protocols in particular, add another dimension to tuning: the cost of I/O. This must be balanced against the cost of CPU.

Consider the problem of transferring a file over a network between two computers. Under most circumstances, the factor that limits the performance of this transaction isn't the load placed on the CPU by the transmitting and receiving programs but rather the performance of the network between the two machines.

A 10 megabyte (10 MB) file takes roughly 3000 seconds to transmit over a 28.8 kilobit per second (Kb/s) transmission channel. Provided that the machines at either end meet some minimal standard of processing power (essentially any commercially available machine can do the job) the speed of transmission is constant because it's limited by the speed of the communications channel. In this case, then, it might be worth expending some CPU power to compress the data before (or during) transmission in order to reduce transmission time. Nearly any machine is fast enough to compress data at much faster than 28.8 Kb/s.

The laptop on which this was written can compress roughly 1 megabyte/second. Typical text files can be compressed by a factor of between two and four, so, compressing while encrypting will reduce the transmission time to between 750 and 1500 seconds, raising the effective data transmission rate to about 100 Kb/s—quite an improvement. Notice that the bottleneck is still the network: The machine can easily compress data faster than it can be transmitted over the network. In fact, compression is so much faster that nearly all commercial modems automatically compress the data as they transmit it.

Now, consider the same two machines connected by Fast Ethernet. Such a network can transmit roughly 100 megabits/s (Mb/s). In this case, then, the same file can be transmitted in about 1 second. By contrast, compressing the file before transmitting it will take roughly 10 seconds. In this case, then, we've made the situation far worse: the bottleneck is the CPU rather than the network.

As we can see, it is possible to trade off I/O bandwidth for CPU and vice versa. The optimal design and configuration for such a system depends on the relative amounts of CPU and I/O bandwidth available. If your system is deployed in a wide variety of environments, it may be advantageous to have it probe to see how much I/O and CPU are available before deciding which tradeoffs to make.

Ex.1024
Askeladden, L.L.C. / Page 247 of 788

Latency versus Throughput

"Never underestimate the bandwidth of a station wagon full of tapes."
—Dr. Warren Jackson

Recall the problem of data transmission from the previous section. This time, let's consider a somewhat larger file of 100 MB. With compression, that file will take somewhere around three hours to transmit over a 28.8 Kb/s channel. With a file this large, it might be more efficient to simply write it onto a tape (this can be done in about ten minutes) and carry (or mail) it from machine to machine.

If we mail it, it will take somewhere between two and five days to move the file from machine to machine. This doesn't seem that good when compared to 3 hours, but imagine we have many such files to transmit. Over a modem, we can only move about eight files a day. If we mail them, we can move them as fast as we can copy them onto tape: around 100 files a day. True, the first file won't arrive for a few days, but they'll arrive at a steady (and fast) rate afterwards.

This example illustrates the difference between *latency* and *throughput*. Latency is the amount of time it takes to process a given transaction from beginning to end. Throughput is the number of total transactions you can maintain over a period of time. As you can see, they're not always related.

To make this difference more clear, suppose we Federal Express the tapes instead of mailing them. We've thus reduced latency to overnight. However, we haven't had any effect on throughput at all. By contrast, if we buy another machine to write tapes with, we can increase throughput without impacting latency.

With a little thought, it's easy to come up with examples of nearly any combination of throughput and latency. Figure 6.2 shows some examples:

	Low Latency	High Latency
Low Throughput	Modems	Pager
High Throughput	LANs	Satellite

Figure 6.2 Latency/throughput variations

Servers versus Clients

As a general rule, network systems have many clients and a few servers. In such systems, slowdown at the client side generally manifests as increased latency, whereas slowdown at the server side manifests as both increased latency and decreased throughput.

Recall our now familiar example of network file transmission, with one twist. Now we have a single server serving a large number of clients. The server has a T1 Internet connection (capable of carrying roughly 1.5 Mb/s) and each client is served by a

28.8 Kb/s modem. In such an environment, the server can service approximately 50 clients at a time and keep their network connections completely full. As before, each 1 MB file takes somewhere around 300 seconds to deliver. Thus, our throughput is roughly 1.5 Mb/s, and our latency is 300 seconds.

Now imagine that we double the number of clients. The server tries to service them all equally, so they now get only half as much data as before. This doubles the latency to 600 seconds. Notice, however, that although it takes twice as long to serve each client, we're serving twice as many clients at the same time, so throughput is unaffected.

Next, imagine that we halve the speed of each client's network connection to 14.4 Kb/s. Again, latency doubles to 600 seconds, but we can serve twice as many clients at this speed, so again throughput is unaffected.

Finally, consider the impact of halving the speed of the server's Internet connection. If we keep the number of clients the same, then we can serve them at only half the previous rate, doubling latency and halving throughput. This sort of effect applies equally when the bottleneck is CPU instead of I/O. Slow clients leave server resources available for other clients. Slow servers simply slow down the system as a whole.

Note that CPU and bandwidth aren't the only scarce resources for a server. Each client consumes some server resources, especially memory, so when the number of clients gets really large, the server can be overloaded just from trying to serve them all, even if there is plenty of CPU and bandwidth to go around.

6.4 Cryptography Is Expensive

In the first section of this chapter, we established that SSL was slow. In the second section, we introduced the concepts necessary to understand the factors that affect performance. In this section, we start to apply that general knowledge to SSL. As we hinted in the first section, the primary reason why SSL is slow is that cryptography, particularly the big number operations required for public key cryptography, is extremely CPU-intensive.

Profiling OpenSSL using `gprof(1)` shows this quite clearly. Consider a simple SSL connection using TLS_RSA_WITH_3DES_EDE_CBC_SHA. The vast majority of the time on both client and server is spent on cryptographic processing. In fact, the vast majority of time is spent on a single operation: RSA decryption.

The two phases of an SSL connection, handshake and data transfer, are worth considering separately. Handshake happens only once per connection, but is comparatively expensive. Each individual data record is comparatively cheap, but for connections involving large amounts of data the cost of the data transfer phase eventually dominates the cost of the handshake. Nevertheless, the cost of data transfer is mostly due to cryptography.

The costs of the handshake differ substantially between client and server, but the data transfer phase is symmetrical. Therefore, in the rest of this section, we'll first discuss the handshake for client and server and then the data transfer phase.

Server Handshake

The performance profile for the server side of the handshake is comparatively simple. Over half of the CPU time goes to a single operation: the RSA private decryption of the pre_master_secret. A distant second is the computation of the master_secret and the key schedules from the pre_master_secret.

Client Handshake

The situation on the client is far less straightforward. The client side operations are much faster than the server side operations. Thus, in many cases the client spends most of its time simply waiting for server messages.

Once this waiting time is factored out, we find that the two most expensive operations are verifying the server's certificates and encrypting the pre_master_secret. Although both of these involve RSA operations, these are public key operations and therefore are roughly an order of magnitude faster than the private key operations the server executes. As a consequence, the client's major computational burdens are twofold: RSA computations and the ASN.1 decoding of the server's certificate.

The Bottleneck

Clearly, in this case, the server's performance limitations control the performance of the system as a whole. This situation would change somewhat if client authentication were used. In that case, the client would perform an RSA private key operation more or less equivalent to that performed by the server and this would become the dominant performance cost on the client. The server would incur almost no additional performance cost.

It's also worth noting that this particular division of labor really only applies to RSA. DSA operations are much more symmetrical between client and server, and we find that when DSS/DHE cipher suites are used, the computational load is similarly symmetrical between client and server.

However, in most production systems, the server is still the bottleneck. First, client authentication and DSS/DH are rarely used. More important, as we discussed previously, the server is accessed by many clients, whereas any client will have only a small number of connections open. Thus, even a very fast server will quickly achieve a load that slows it down below the speed of clients.

Data Transfer

In the data transfer phase, there are two relevant cryptographic operations, record encryption and the record MAC. Together, these account for the majority of the cost of data transfer. The relative importance of these depends on which precise algorithms are chosen and the implementation details. If a fast encryption algorithm (such as RC4) is chosen, then the HMAC will be the dominant cost. If a slow encryption algorithm such as 3DES is shown, then encryption and MACing will share the cost somewhat evenly. Record assembly and disassembly account for an insignificant amount of time.

6.5 Session Resumption

As we discussed in the previous section, the performance bottleneck in SSL handshakes is the public key crypto operations associated with the handshake itself. Most of these operations are associated with the exchange of the session key. The encryption and decryption of the master_secret are obviously directly related; however remember that the only need to verify the server's certificate is in order to use it for key exchange.

We can conclude from this that if we could do away with these operations, the handshake would be dramatically faster. As we discussed in Chapter 4, session resumption provides us with an opportunity to do exactly that. Because resumed sessions use the master_secret from a previous connection, we can omit the expensive public key computations.

As expected, this provides a dramatic performance improvement. In our test harness, using 512-bit RSA keys, we see a performance improvement of roughly a factor of 20 in the handshake. An even larger improvement would be observed with larger keys. More important, this improvement comes almost completely as a result of reduced load on the server. Consequently, in an environment where most sessions are resumed, we would expect to see a corresponding improvement in throughput.

Clearly, session resumption can provide a significant performance improvement. However, it's not appropriate in all cases. Rather, it's only appropriate when a reasonable number of clients will reconnect within some reasonable period of time.

The Cost of Session Resumption

The cost of session resumption is borne almost entirely by the server. As we indicated before, a client will only talk to some very small number of servers, so the cost of maintaining a cache is minimal. The server, however, needs to retain cache entries for a large number of clients, so the load is correspondingly greater.

Session resumption consumes two primary resources: memory to store the resumable sessions and CPU time to check the cache. Obviously, in an environment where resumption happens frequently, these resources have been well spent, but in an environment where sessions are never resumed, they're just waste.

The state for a resumable session requires somewhere on the order of 50–100 bytes in the session cache. This isn't a lot for any individual session, but in a server handling hundreds of transactions a minute, this could mean megabytes of data in an hour. Clearly, even if resumption pays off, it's worth limiting the size of the cache and timing out entries fairly frequently.

Accessing the cache takes time as well. The relevant issue is that the cache must be shared between multiple working threads or processes. This means that the cache must be locked, read, and then unlocked. Locking and unlocking are comparatively expensive operations. If the cache needs to be shared between processes, as with many UNIX servers, every read or write of the cache requires a context switch into the kernel. This is even more expensive. Again, if a cache entry is resumed, this time was negligible, but if sessions are never resumed, then the CPU time consumed was merely unnecessary overhead.

The moral of the story is that session resumption is nearly always worth doing on the client. On the server, however, it's only worth doing if there is a lot of repeat business within a fairly short period of time. Even in such situations, it's worth asking whether the protocol can be converted to simply leave those connections up rather than incurring even the minimal cost of a resumed handshake.

6.6 Handshake Algorithm and Key Choice

As we discussed in Section 6.4, the performance characteristics of an SSL connection depend substantially on the algorithms being used. In the handshake phase, this can dramatically affect the cost of the handshake, as well as changing the relative loads on the client and server.

RSA versus DSA

When deploying SSL systems, the choice of signature algorithm often depends on intellectual property or compatibility concerns. However, in cases where performance is the dominant concern, RSA is the algorithm of choice over DSS/DH. In general, RSA is *much* faster than DSA for verification and comparable for signature, as shown in Figure 6.3. However, this cost is paid primarily by the client, so it affects latency, not throughput.

Algorithm	Keysize	Signs/s	Verifys/s
RSA	512	342	3287
DSA	512	331	273
RSA	1024	62	1078
DSA	1024	112	94
RSA	2048	10	320
DSA	2048	34	27

Figure 6.3 Digital signature performance (Pentium II 400/OpenSSL)

However, the DSA cipher suites are nearly always run in ephemeral mode. This requires substantial extra work on both sides to generate, sign, and verify the ephemeral DH keys. This dramatically reduces throughput. In practice, DSA cipher suites tend to be between two and ten times slower than RSA cipher suites of comparable strength.

Fast or Strong, Choose One

The performance of public key algorithms declines dramatically with key size. RSA with 1024 bits is roughly four times slower than RSA with 512 bits. DSA performance

is similarly reduced by longer keys. As a consequence, choosing your private key length is a compromise between security and handshake performance. As discussed in Chapter 1, 512 bits is almost certainly too short for valuable data. However, 768-bit keys are probably strong enough for most commercial transactions and are dramatically faster than 1024-bit keys.

Ephemeral RSA

Recall that when using RSA with a strong (>512) bit key and an export cipher, the standard requires you to use a 512-bit ephemeral RSA key. In essence, this adds a single 512-bit RSA operation to each side (a public one on the client side and a private one on the server side). This has some negative effect on throughput and latency.

Note that both sides must still perform all the operations they would have performed if ephemeral RSA were not being used. The slower these operations, the smaller the proportional effect will be of using ephemeral RSA. For 1024-bit keys, we expect to see between a 10% and 25% slowdown. Note that this is not a security/performance tradeoff, as 512-bit ephemeral RSA is both weaker and slower than 1024-bit static RSA: 512-bit keys are much weaker than 1024-bit keys, and because the same 512-bit key is used for a large number of transactions, there is no compensating benefit of perfect forward secrecy.

6.7 Bulk Data Transfer

Algorithm Choice

Recall that cryptography consumes the bulk of the time in record processing. Unsurprisingly, the choice of MAC and encryption algorithms affects the performance of the system. In short, use RC4 for maximum speed and 3DES for maximum security. RC4 is about ten times faster than 3DES. Digest choice is less flexible as MD5 is being phased out. SHA-1 is fast enough in most real world scenarios, so it's a generally good choice. On modern systems, RC4 with SHA-1 is fast enough to saturate most networks.

Optimal Record Size

A surprising amount of the record computation is insensitive to the size of the data being transmitted. There is substantial fixed overhead, both from the cryptographic algorithms and from writing to the network. As a consequence, transmitting the data in small records results in bad performance. Up to a certain point, increasing the size of the record increases performance. On our test system, this point appears to be roughly 1024 bytes. Beyond this point, the advantage of using even larger blocks is small. If you're transferring bulk data, it pays to buffer it until you can encrypt fairly large records.

6.8 Basic SSL Performance Rules

Asymmetric algorithm choice. Use RSA.

Private key size. Use the shortest private keys with which you feel safe. 768 is good enough for most applications.

Symmetric algorithm choice. Use RC4 for best performance, 3DES for best security. Export versions compromise security without improving performance.

Digest algorithm choice. MD5 only improves performance about 40% over SHA-1. In most cases SHA-1 will be fast enough. Stick with SHA-1 for security.

Session resumption. Clients should always use session resumption. Servers should use it only if clients tend to reconnect within five to ten minutes.

Record size. Send data in the largest chunks possible.

6.9 The Story So Far

The general idea behind getting good performance is to perform as few operations as possible. This means minimizing the number of time-consuming operations and, whenever possible, making them faster. As we've seen, the time-consuming operations in SSL are mostly cryptographic. Session resumption allows us to remove a number of those operations and proper algorithm selection allows us to minimize the ones we still have to perform.

The first part of this chapter was intended to provide enough information to obtain good performance out of SSL as a server or client administrator. The rest of this chapter is geared toward the programmer and is intended to provide a more complete picture of the performance characteristics of SSL. We reexamine some of the material from the first part in more detail as well as discussing some of the fine points of the interaction of SSL with TCP.

6.10 Handshake Time Allocation

The next several sections provide detailed coverage of a number of different SSL handshakes. The data described in these sections was collected using a specially instrumented version of OpenSSL 0.9.4. The instrumentation records the time taken to generate and/or process each handshake message.

Test Environment

The test system is a 300 MHz Pentium running FreeBSD. The client and server programs are modified versions of the OpenSSL `s_client` and `s_server` programs. Both client and server are running on the same system, so network latency is insignificant. The instrumentation does not record the time taken to actually send and receive the network messages or to context switch between processes.

Serialization

As discussed in Chapters 3 and 4, the order of SSL messages is completely described by the standard. Moreover, at certain times the server must wait for a client message and vice versa. OpenSSL buffers network output for increased performance. On modern operating systems, network I/O requires a context switch to enter the kernel. This is expensive. Thus, it's often computationally cheaper to generate a number of messages and transmit them all at once. The traditional approach to buffering is to have a fixed size buffer and transmit the data when the buffer gets full. With SSL, it's also necessary to flush the buffer when transmitting the last message before waiting for a reply. Otherwise the implementations will deadlock.

Figure 6.4 shows this behavior in a simple RSA handshake (the same one we showed in Chapter 3). The ServerHello, Certificate and ServerHelloDone all have the same timestamp, meaning that they were transmitted to the network at the same time. Similarly, the client's ClientKeyExchange and ChangeCipherSpec were transmitted at the same time.

```
New TCP connection: speedy(3266) <-> romeo(4433)
1  0.0456 (0.0456) C>S   Handshake      ClientHello
2  0.0461 (0.0004) S>C   Handshake      ServerHello
3  0.0461 (0.0000) S>C   Handshake      Certificate
4  0.0461 (0.0000) S>C   Handshake      ServerHelloDone
5  0.2766 (0.2304) C>S   Handshake      ClientKeyExchange
6  0.2766 (0.0000) C>S   ChangeCipherSpec
7  0.2766 (0.0000) C>S   Handshake      Finished
8  0.2810 (0.0044) S>C   ChangeCipherSpec
9  0.2810 (0.0000) S>C   Handshake      Finished
10 1.0560 (0.7749) C>S   application_data
11 6.3681 (5.3121) S>C   application_data
12 7.3495 (0.9813) C>S   Alert
Client FIN
Server FIN
```

Figure 6.4 Buffering of SSL handshake messages

However, when messages are buffered in this way, it's possible that one side is waiting for a message that has already been written by the other side, but is still in the output buffer. Thus, processing on the client and the server sometimes take place sequentially and sometimes in parallel, depending on whether the handshake messages completely fill the buffers or not. If the buffers fill, the messages will be sent immediately, in which case the host on the other side can process the message in parallel with the next message being generated. If the buffers don't fill, the messages will only be sent when the sending side stops to wait for a response, in which case the messages will be processed sequentially.

The version of OpenSSL used to generate these traces has had the buffer size increased from 1024 to 4096 bytes so that all handshake messages are processed

sequentially. This makes the handshake traces much easier to read and allows us to show the CPU time used without the impact of context switching. The choice of 1024 bytes in OpenSSL is arbitrary and other implementations may use no buffering at all.

In some cases, this sort of buffering improves latency and in some circumstances it makes it worse. We'll discuss this issue in Section 6.23 when we discuss Nagle's algorithm.

6.11 Normal RSA Mode

The first mode that we'll be examining is the server-only RSA mode. Figure 6.5 shows the time allocation to process the various messages. Time allocated to client processing is in the left column and time allocated to server processing is in the right column. Indentation is used to indicate instrumentation of pieces of an operation. All times are in milliseconds. Thus, reading the server certificate consumed 8.11 ms. This included three verifies, each of which took about 2 ms.

In general, RSA operations come in two flavors: *private key operations* (digital signature and private key decryption) and *public key operations* (signature verification and public key encryption). This distinction is important in this context because private key operations are much slower than public key operations. However, all private key operations with a given key size take pretty much the same time, as do all public key operations with a given key size. We'll be using these terms throughout the discussion of RSA to avoid distinguishing between the various instances of each type of operation.

As we'd expect from the 90/10 rule and the discussion in Section 6.4, the vast majority of the CPU time is allocated to processing three messages: on the client side, the processing of the server's Certificate message and the generation of the ClientKeyExchange; on the server side, the processing of the client's ClientKeyExchange message. All the rest of the messages put together account for less than a millisecond of CPU time. We'll discuss each of the three important messages in turn.

> When we initially collected these traces, the majority of the time went to the creation of the ClientHello message. This was due to the one-time (on-demand) seeding of the OpenSSL PRNG. In reality, this is a startup cost and so we moved the seeding to a section of the code called before the handshake.

Certificate (Client)

Recall that the Certificate message contains the server's certificate chain. In order to process it, the client needs to parse and verify each certificate. Primarily this means performing RSA verifies. In this case the certificate chain is three certificates long, so three verifies are required. Each verify takes roughly 2 ms. The rest of the time goes mostly to ASN.1 parsing.

```
Client              Server
0.07 Write client_hello
                     0.15 Read client_hello
                     0.07 Write server_hello
                     0.20 Write certificate
                     0.00 Write server_hello_done
0.05 Read server_hello
8.11 Read certificate
2.47   verify
1.95   verify
2.25   verify
0.00 Read server_hello_done
2.54 Write client_key_exchange
2.26   encrypt_premaster
0.15 Write finished
                    31.01 Read client_key_exchange
                    30.79   decrypt_premaster
                     0.00 Read finished
                     0.12 Write finished
0.00 Read finished

Client: 10.95       Server: 31.59
```

Figure 6.5 RSA server-only mode time allocation (ms)

This is an unusually long certificate chain. Normal certificate chains are one certificate long (a certificate signed directly by the root), requiring only a single verify. Under such circumstances the time taken to process this message is approximately the same as the ClientKeyExchange. Also, note that the limiting factor here is the length of the *CA* keys, not the server's key. The server's key isn't really processed in this stage.

As a performance optimization, it is possible to cache the results of certificate validation, thus saving the public key verification stage. This doesn't make sense when certificate chains are short because the number of server certificates one is likely to encounter is too large to be cached. However, when long chains are used, as in this handshake, it's often worth caching CA certificates.

> By default, `s_client` provides a callback to OpenSSL's certificate processing code. This callback outputs a lot of certificate debugging information to `stderr`. We had to disconnect this callback because the I/O was consuming significant amounts of CPU time and thus skewing the instrumentation results.

ClientKeyExchange (Client)

Recall from Chapter 4 that the ClientKeyExchange contains the pre_master_secret encrypted under the server's public key. The dominant operation here is this RSA encryption. Thus, the limiting factor is the length of the server's public key.

ClientKeyExchange (Server)

Unsurprisingly, the dominant operation here is the server's RSA decryption of the pre_master_secret, which takes up essentially all of the time allocated to this message. Note that the time this operation takes is directly related to the size of the server's RSA key. The longer the key, the longer the decryption takes. Note that this is a much slower operation than the encrypts and verifies performed by the client.

Note that because the decrypt is a server operation, it is the bottleneck in SSL throughput. The cost of the decrypt is dependent on the server's choice of key length, making the server's RSA key length the most important factor limiting performance in this mode.

6.12 RSA with Client Authentication

RSA with client authentication mode is basically normal RSA mode with some extra steps. From a performance perspective, one takes all the operations that were previously performed on the server and also performs them on the client and vice versa. As a consequence, client authentication mode substantially increases the load on the client but only slightly increases the load on the server. Figure 6.6 shows an instrumented client authentication trace. In this trace, and from now on, the operations that have changed from the previous trace will be rendered in bold.

All the operations we performed in normal mode still consume roughly the same amount of CPU time. The new time-consuming operations we've added are the client-side generation of the CertificateVerify message and the server-side verification of the client's Certificate and CertificateVerify message.

The CertificateVerify message consists of an RSA-signed message. The expensive stage here is, of course, the RSA signature. The comments in the previous section about the processing of the ClientKeyExchange apply here.

On the server side, the Certificate message is processed exactly as the client processes the server's Certificate message. That is to say that the dominant cost is the verification of the digital signatures. Finally, the server needs to process the CertificateVerify. This mainly consists of checking the client's digital signature, one public key operation.

Although the total computational cost of client authentication mode is roughly double that of server-only authentication, this cost is borne disproportionately by the client. In this case, the cost to the server is roughly 20% of that of the client. As a result, using client authentication substantially increases latency but has a much smaller effect on throughput. Overall, client authentication adds:

+1 Client-side RSA private key operation
+1 Server-side RSA public key operation
+1 Server-side RSA public key operation per client certificate

```
Client                Server
 0.07 Write client_hello
                       0.15 Read client_hello
                       0.17 Write server_hello
                       0.49 Write certificate
                       0.06 Write certificate_request
                       0.00 Write server_hello_done
 0.05 Read server_hello
 9.01 Read certificate
 2.44   verify
 1.94   verify
 2.24   verify
 0.18 Read certificate_request
 0.00 Read server_hello_done
 0.42 Write certificate
 2.61 Write client_key_exchange
 2.33   encrypt_premaster
30.56 Write certificate_verify
 0.12 Write finished
                       8.65 Read certificate
                       2.38   verify
                       1.97   verify
                       2.24   verify
                      30.72 Read client_key_exchange
                      30.51   decrypt_premaster
                       2.23 Read certificate_verify
                       0.00 Read finished
                       0.14 Write finished
 0.00 Read finished

 Client: 43.075       Server: 42.66
```

Figure 6.6 RSA client authentication mode time allocation (ms)

6.13 Ephemeral RSA

As we discussed in Chapter 4, ephemeral RSA uses a temporary short RSA key to encrypt the pre_master_secret. Effectively, this adds one short RSA operation to both client and server. However, this is accomplished by adding some operations and subtracting others, as we'll see in this section. In order to do that, we need to look at each

message whose contents change in ephemeral mode. Figure 6.7 shows the trace of a handshake using ephemeral RSA.

```
Client              Server
 0.07 Write client_hello
                     0.15 Read client_hello
                     0.12 Write server_hello
                     0.49 Write certificate
                    30.45 Write server_key_exchange
                    30.43   sign
                     0.00 Write server_hello_done
 0.05 Read server_hello
 8.57 Read certificate
 2.34   verify
 1.94   verify
 2.23   verify
 2.23 Read server_key_exchange
 2.20   verify
 0.00 Read server_hello_done
 1.12 Write client_key_exchange
 0.86   encrypt_premaster
 0.11 Write finished
                     5.81 Read client_key_exchange
                     5.59   decrypt_premaster
                     0.00 Read finished
                     0.11 Write finished
 0.00 Read finished

Client: 12.2                    Server: 37.17
```

Figure 6.7 Ephemeral RSA mode time allocation (ms)

ServerKeyExchange (Server)

This message doesn't even appear in the normal RSA handshake, so whatever cryptographic processing it entails is new, and as we can see, it's quite an expensive operation. The reason for this is that we're signing the ServerKeyExchange using the server's private key, which is typically long (1024 bits or more).

Note that this does not include the time to generate the ephemeral RSA private key. On this platform it takes 200–400 ms, which is too expensive to do for each transaction. Instead, one conventionally generates the ephemeral key once a day or so. Thus, we don't count it in the per-transaction cost.

The score so far:
+1 server-side long RSA private key operation

ServerKeyExchange (Client)

On the client side we need to process the server's ServerKeyExchange message. The only interesting processing here is verifying the server's signature over the message. This adds a client side long public key operation.

The score so far:
+1 server-side long RSA private key operation
+1 client-side long RSA public key operation

ClientKeyExchange (Client)

This message existed in the normal RSA case, but in that case the client was doing a long RSA encryption. Now the client simply does a 512-bit RSA encryption, which is substantially faster. So, we've added a 512-bit RSA operation and subtracted a long RSA operation.

The score so far:
+1 server-side long RSA private key operation
+1 client-side 512-bit RSA public key operation

ClientKeyExchange (Server)

As we said above, we've substituted a short RSA operation for a short one with this message. This obviously applies on the server side as well. Thus, the final change is:

+1 client-side 512-bit RSA public key operation
+1 server-side 512-bit RSA private key operation

Because 512-bit operations are roughly 4 times as fast as 1024-bit operations, this isn't that big a difference. On this platform it works out to roughly 2 ms on the client side and 6 ms on the server side.

6.14 DSS/DHE

The following series of traces are handshakes using DSS/DHE mode. The DSA signing key and the ephemeral DH key are both 1024 bits long. As before, the client and the server are the same machine, and we've arranged the buffers so that the handshake happens sequentially.

Figure 6.8 shows the time allocation for a normal DSS/DHE handshake. The messages are the same as those for ephemeral RSA but the processing times are wildly different. This is accounted for by two factors: first, the performance characteristics of DSS and DH are dramatically different from those for RSA. Second, both sides are generating new ephemeral DH keys during the connection.

We can immediately make two observations: first, the DSS/DHE handshake is dramatically more expensive than an RSA handshake; we're spending roughly eight times as much CPU time. Second, the majority of the additional cost of the handshake is

```
Client                Server
  0.07 Write client_hello
                         0.18 Read client_hello
                         0.11 Write server_hello
                         0.26 Write certificate
                       113.90 Write server_key_exchange
                        96.67   generate_keys
                        17.15   sign
                         0.00 Write server_hello_done
  0.06 Read server_hello
  0.48 Read certificate
 21.24 Read server_key_exchange
 21.19   verify
  0.00 Read server_hello_done
196.95 Write client_key_exchange
 96.82   generate_keys
 99.86   key_agree
  0.18 Write finished
                       100.86 Read client_key_exchange
                       100.66   key_agree
                         0.00 Read finished
                         0.15 Write finished
  0.01 Read finished

Client: 219.02        Server: 215.51
```

Figure 6.8 DSS/DHE mode time allocation (ms)

borne by the client. Nevertheless, Figure 6.8 shows that the server consumes significantly more CPU (by a factor of five to seven) when using DSS/DHE than when using RSA.

ServerKeyExchange (Server)

Recall that in the ephemeral RSA ServerKeyExchange nearly all of the time was spent signing the message using the server's permanent key. In this handshake, we've added something new: the server generates the DH key on the fly, one for each connection. The primary cost of generating the ServerKeyExchange message is generating this key. In fact, a 1024 bit DSA signature is about twice as fast as the equivalent RSA signature.

It's worth examining the DH key generation process in more detail. Recall from Chapter 1 that DH (unlike RSA) relies on a publicly known group (p and g). The private key is simply a random number X. The public key (Y) is $g^X mod\ p$. The key

generation we're talking about here is simply the generation of the pair (X, Y), not the generation of the group.

The generation of the group is extremely slow, because it requires making a 1024-bit prime (p). Moreover, it is often recommended that that prime have special properties which makes it take even longer to generate. On the test machine, OpenSSL took ten minutes to generate a 1024-bit DH group. Thus, group generation is done either at server startup or (preferably) when the server is originally installed.

Certificate (Client)

The client side certificate processing is the same with DSA as with RSA; the only difference is the algorithm. As we discussed previously, DSA is substantially slower than RSA for verification (roughly ten times slower at the 1024-bit ey length). Thus, the client processing of the Certificate message is much slower when using DSA.

ServerKeyExchange (Client)

As with the Certificate message, client processing of the ServerKeyExchange for DSS/DHE differs from RSA only by substituting DSA for RSA. Thus, the processing of this message is slower purely because DSA verification is so much slower.

ClientKeyExchange (Client)

In this trace, the processing of the ClientKeyExchange message actually consists of two operations, each consuming roughly half of the CPU time: generating the client's ephemeral DH key and computing the DH shared secret (ZZ). Generating the client's ephemeral DH key is exactly the same as generating the server's ephemeral DH key, and we see that the time allocated to this operation on the client is roughly the same as that on the server.

We've seen the computation of ZZ before. Recall that we compute

$$ZZ = Ys^X \bmod p = (g^{Xs})^{Xc} \bmod p = g^{Xs \cdot Xc} \bmod p$$

As you can see from the trace, this operation takes roughly the same amount of time as the generation of the client's DH key.

Note that we've missed an opportunity for parallelism. The generation of the ClientKeyExchange does not depend on knowing ZZ. ZZ is never transmitted over the wire and is used purely to compute the symmetric message keys. Computing ZZ before we send the ClientKeyExchange simply delays transmission of the message, increasing latency.

A better approach would be to send the ClientKeyExchange and then compute ZZ. That way the client can be computing ZZ while the ClientKeyExchange is in transit and/or the server is itself computing ZZ. Note that this change would primarily affect latency; unless the server is unloaded, it would not have much of an effect on throughput.

ClientKeyExchange (Server)

ClientKeyExchange processing on the server is simply the dual of processing on the client. We compute *ZZ* using the server's private key and the client's public key. Unsurprisingly it takes roughly the same amount of time to compute.

6.15 DSS/DHE with Client Authentication

The final handshake we'll cover is DSS/DHE with client authentication. Figure 6.9 shows that as with RSA client authentication, the only significant difference from the non-client authentication mode is the addition of several new messages: the client Certificate and the CertificateVerify.

```
Client                Server
  0.07 Write client_hello
                         0.17 Read client_hello
                         0.12 Write server_hello
                         0.26 Write certificate
                       113.96 Write server_key_exchange
                        96.63   generate_keys
                        17.26   sign
                         0.02 Write certificate_request
                         0.00 Write server_hello_done
  0.06 Read server_hello
  0.46 Read certificate
 21.36 Read server_key_exchange
 21.30   verify
  0.01 Read certificate_request
  0.00 Read server_hello_done
  0.23 Write certificate
193.21 Write client_key_exchange
 96.87   generate_keys
 96.07   key_agree
 17.06 Write certificate_verify
  0.18 Write finished
                         0.42 Read certificate
                        95.98 Read client_key_exchange
                        95.79   key_agree
                        21.00 Read certificate_verify
                         0.00 Read finished
                         0.15 Write finished
  0.01 Read finished

Client: 232.69        Server: 232.13
```

Figure 6.9 DSS/DHE client authentication mode time allocation (ms)

On the client side, the only substantial operation is the DSA signature on the CertificateVerify. The server's processing of the client's Certificate message is dominated by certificate verification. Because this is a DSS certificate, the verification is fairly slow. Similarly, the server's processing of the CertificateVerify is dominated by DSA verification. Note that with DSA (unlike RSA) the primary cost of client authentication is borne by the server because DSA verification is so slow.

6.16 Performance Improvements with DH

As we've seen, DHE performance is much worse than performance with RSA. It would be reasonable to ask why anyone would ever use this mode. Until very recently, there were two reasons: first, it's free. RSA was still subject to patent in the United States; DSA is not. Second, it provides one new feature, Perfect Forward Secrecy. PFS isn't practical with RSA due to the high cost of RSA key generation, but it is practical (though expensive) with DHE. Now that RSA's patent has expired the only reason is to get PFS. PFS is sometimes valuable, so it's worth considering whether we can improve DHE performance at all. The answer is yes.

Use Smaller Private Keys

Recall that all DH operations require exponentiating some number to $X \ mod \ p$. The larger the value of X the slower this operation. A common approach is simply to use an X of approximately the same size as p. To be more specific, X is chosen in the interval $(2, p)$. It is possible that X will be much less than p, but because X is randomly chosen over this interval, it will be about as long as p. However, if a smaller X is deliberately chosen, the operations are much faster.

This optimization is allowed only under certain specific circumstances, however. Otherwise, it is possible to greatly weaken the security of the key exchange. Recall from Chapter 1 that g and p must have special properties in order to use a short X.

However, the client cannot conveniently tell if the server has generated g and p appropriately because the server only sends it g and p in the ServerKeyExchange. Thus, the client cannot safely use a short X and therefore doesn't experience any speedup at all. Under most circumstances, only server performance would matter but if the client is a very limited device such as a palmtop or a cell phone, the load on the client from the DH operation may be substantial.

As a demonstration of this optimization, we modified OpenSSL, which chooses X over $(2, p)$, to choose a 256-bit X while still using a 1024-bit p. This simple change increased the speed of the server-side DH operations by roughly a factor of four, as shown in Figure 6.10.

Recall that we said in Chapter 1 that the length of X should be chosen so that it is twice the length of the desired keying material. However, in practice the strength of a DH key is also limited by the size of p. The strength of a DH key is governed by the minimum of the strength of p and the strength of X. An X of 160 bits is roughly equivalent to a 1024-bit p. Thus, with a 1024-bit prime, a 256-bit X is more than long enough. (See [Menezes1996].)

Ex.1024
Askeladden, L.L.C. / Page 265 of 788

```
Client                Server
  0.07 Write client_hello
                         0.17 Read client_hello
                         0.12 Write server_hello
                         0.26 Write certificate
                        43.14 Write server_key_exchange
                        25.93   generate_keys
                        17.14   sign
                         0.00 Write server_hello_done
  0.06 Read server_hello
  0.47 Read certificate
 21.26 Read server_key_exchange
 21.21   verify
  0.00 Read server_hello_done
192.66 Write client_key_exchange
 96.60   generate_keys
 95.79   key_agree
  0.18 Write finished
                        25.57 Read client_key_exchange
                        25.38   key_agree
                         0.01 Read finished
                         0.15 Write finished
  0.01 Read finished

Client: 214.75       Server: 69.46
```

Figure 6.10 Performance effect of short DH keys

Reuse Ephemeral Keys

We improved performance without a significant effect on security by using shorter private keys (X). However, it's possible to improve performance even further by discarding Perfect Forward Secrecy. Recall that roughly a third of the server's computational load is consumed by generating the ephemeral DH key. Reusing that key reduces the load accordingly. Unfortunately, the server must still perform the signature on the ServerKeyExchange because the signature is also computed over portions of the handshake.

Reusing ephemeral keys only reduces server load, not client load. Because the group is determined by the server, in most cases each server will have a different group, so the client must still generate a new key for each server. Nevertheless, this optimization improves both throughput and latency because it makes the server faster.

Some might object that in order to get this performance improvement, we had to give up some security. However, note that RSA mode does not provide PFS at all, so

although this sort of reuse is weaker than using a new key for each connection, it is comparable in strength to RSA. As a compromise, it's possible to regenerate the server's DH key periodically. Regenerating as frequently as every 20 or 30 connections will provide partial PFS with only a negligible impact on performance. Note that if you use a short X and reuse it for a number of connections, you must take care to prevent small subgroup attacks, as described in Chapter 5.

Precomputation for DH Key Generation

Even if we're not willing to reuse ephemeral keys, it's possible to improve key generation dramatically. In order to do so, you precompute a table of powers of the generator g. Because each public key computation requires exponentiating g, this optimization will reduce the time it takes to compute Y from X. Wei Dai [Dai2000] reports a speedup of a factor of about three for precomputed DH key generation versus ordinary DH key generation. As a special case, if the base is 2 it's possible to compute exponentials very quickly simply by left-shifting.

Static DH

A final option is to use a static DH key. Recall from Chapter 4 that it is possible to have the server's DH key in a certificate, exactly as is usually done with RSA keys. This dispenses with the ServerKeyExchange method completely, leaving the only computational load on the server as the computation of ZZ. Together with short Xs, this optimization reduces the load on the server to roughly comparable to that presented by RSA. The load on the client is still significantly greater because DSA verification is so slow. Note that if static DH is used, great care must be used to ensure that small subgroup attacks are not possible.

The primary deployment problem with static DH is that it requires static DH certificates, which are even less widely used than DSA certificates.

6.17 Record Processing

Algorithm Choice

Figure 6.11 shows the performance of OpenSSL on a Pentium 300 using various combinations of encryption and MAC algorithm. These numbers were generated by computing SSL records but not sending them to the network. Thus, they measure only record generation cost, not network overhead. They are intended as representative of the sorts of behavior you can expect. Obviously, each system and implementation will perform differently.

In short, RC4 with MD5 provides the best performance and it's roughly eight times faster than 3DES with SHA. The moral of the story is that you should use RC4 if you want reasonable security and high performance. Using MD5 instead of SHA-1 gets you about a 40% improvement.

Cipher/HASH	KB/s
RC4-MD5	15034
RC4-SHA	10831
DES-CBC-SHA	4758
DES-CBC3-SHA	2068

Figure 6.11 SSL record processing speed (Pentium 300)

Note that the processing time for the slowest cipher (3DES/SHA) is over 2MB/s. This is easily fast enough to saturate a 10 Mb Ethernet. The performance of RC4-MD5 on a reasonable machine is easily fast enough to saturate a 100 Mb Ethernet. On our test machine, even RC4 with SHA-1 will generate over 80 Mb/second of traffic. In practice, this is enough to saturate a 100 Mb network. A fast machine can easily handle the SSL record processing for even the fastest networks if RC4 is used, even with SHA-1.

It's important to note that there is no performance benefit from using the export variants of each cipher. SSL always uses the same version of each cipher and simply expands a shortened key out to the full key length. As a consequence, the export versions have exactly the same data transfer rate as the strong versions. This is different from the behavior with public key ciphers, where the export versions are notably faster.

Optimal Record Size

Both SSLv3 and TLS use HMAC (SSLv3 uses a variant, but the performance characteristics are roughly the same). As we discussed in Chapter 1, HMAC consists of a pair of nested digests. The first digest covers the key and the data. The second digest covers the key and the output of the first digest. The key is always processed as a separate digest block.

Therefore, the inner digest is dependent on the length of the message, but the outer digest isn't. Data is fed to the digests 64 bytes at a time. Up to a digest input of 64 bytes, 4 digest blocks need to be processed. For each 64 bytes after that, one more digest block is required. The point is that there is a lot of fixed overhead for the MAC.

When we encrypt the record, we need to encrypt both the MAC and the data. If we're using SHA, then this means we need to encrypt a minimum of 20 bytes. Again, there is significant fixed message overhead. Therefore, it pays to encrypt large payloads whenever possible in order to reduce the effect of the overhead. Figure 6.12 shows this quite clearly.

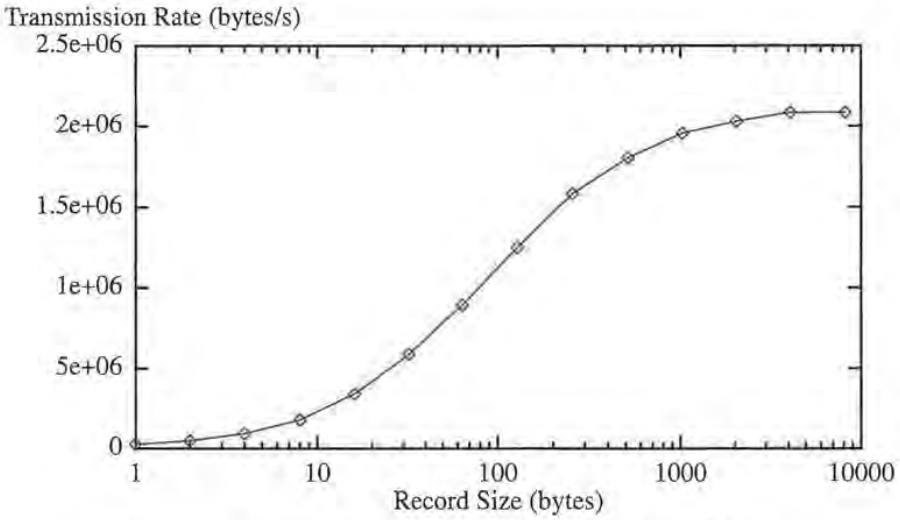


Figure 6.12 SSL performance at various message sizes (3DES/SHA-1)

The data for Figure 6.12 was collected using 3DES-CBC and SHA, but we could draw a similar picture using any set of ciphers. In order to show the effect at small record sizes, we're using a log scale for the X axis.

As Figure 6.12 shows, the performance with really small record sizes is terrible, and it gets progressively better until a point of diminishing returns around 600 bytes or so. The exact location of this point will of course depend on the implementation, operating system, compiler, etc., but the basic message should be clear. Using record sizes smaller than 32 bytes produces really terrible performance. For optimal performance, use record sizes of 1024 bytes or greater when possible.

6.18 Java

The Java performance situation is far worse than the C performance situation. Java code in general is slow, and crypto code is especially CPU intensive and therefore especially slow. Recall that on our test platform OpenSSL executes SHA-1 at roughly 22 MB/s. The fastest code for Java that the authors know of executes at roughly 2.3 MB/s on a Pentium II/450.

Operating System

Unfortunately, it's fairly difficult to get a clear picture of the situation since the performance of Java crypto code is VM and operating system dependent. DES is probably the most common algorithm implemented, and its speed varies from 37 KB/s running under Netscape Navigator on FreeBSD to 2.3 MB/s running under Internet Explorer on Windows NT. Figure 6.13 shows the performance of DES on various platforms.

Processor	OS	VM	Speed (KB/s)
Pentium II (400)	FreeBSD 3.4	Netscape 4.7	36
Pentium II (400)	FreeBSD 3.4	JDK 1.1.8	105
Pentium II (450)	Windows NT	JDK 1.2.2	2368
Pentium II (450)	Windows NT	Netscape 4.61	1856
Pentium II (450)	Windows NT	Internet Explorer 5	2334

Figure 6.13 Java DES performance on various platforms

Note that the performance of the Windows implementations is dramatically better than that of the FreeBSD implementations. Windows is one of the platforms on which Sun has delivered official JDKs, whereas the FreeBSD port is unofficial. Clearly, much more effort has gone into tuning the Windows implementations. However, even on Windows the difference between IE and Navigator is substantial.

Native Code Acceleration

Because the bottleneck in this process is the Java cryptographic primitives, an efficient and straightforward way to dramatically improve performance is to replace the Java primitives with fast native code. JNI (the Java Native Interface) provides a portable way to interface Java with native routines (typically with C code).

The GoNative Provider (see `http://www.rtfm.com/puretls/gonative.html`) implements one approach to doing this: The Java Development Kit (JDK) provides a generic cryptographic provider interface called the Java Cryptography Architecture (JCA). Programs that use the JCA call the JCA to get a specific instance (a class object) which implements a given algorithm. Providers inform the JCA of which algorithms they implement and the JCA automatically arranges to use the right provider.

The combination of the JCA and JNI is what makes this form of acceleration easy. GoNative Provider is a JCA provider which acts as a bridge to OpenSSL (using JNI). It can be installed as a provider on a given machine and made accessible to all applications transparently without the application making any special effort. Cryptographic algorithms implemented in this fashion are far faster than the standard Java algorithms and are only about 10% slower than the equivalent code written directly in C. Note that the faster the algorithm, the more significant the overhead from JNI itself. Figure 6.14 shows the performance properties of some representative algorithms (platform: Pentium 300, FreeBSD).

Algorithm	OpenSSL	Java	GoNative Provider
DES (KB/s)	6792	31	5165
3DES (KB/s)	2392	10	2142
RC4 (KB/s)	32363	132	13872
SHA-1 (KB/s)	22838	93	10014
DSA (sign/s)	60	7	48
DSA (verify/s)	49	4	40

Figure 6.14 Relative performance of Java and C

One observation to make from this table is that there is significant overhead from calling into JNI itself. This is why 3DES performance is so close between GoNative Provider and C. As a corollary to this observation, it's especially important to use large record sizes if you want to get maximum benefit from JNI acceleration.

It's also important to buffer as much data as possible in Java before passing it into the C code. For instance, imagine you're given a large chunk of data to DES. It's important to pass it to the DES code all as one block rather than calling DES on the individual 8 byte blocks. Figure 6.15 shows the effect of input block size on DES encryption with GoNative Provider. Note that Figure 6.15 only shows performance for DES, not for SSL record encryption, which would include the MAC computation. However, similar block size effects hold true for message digests and MACs.

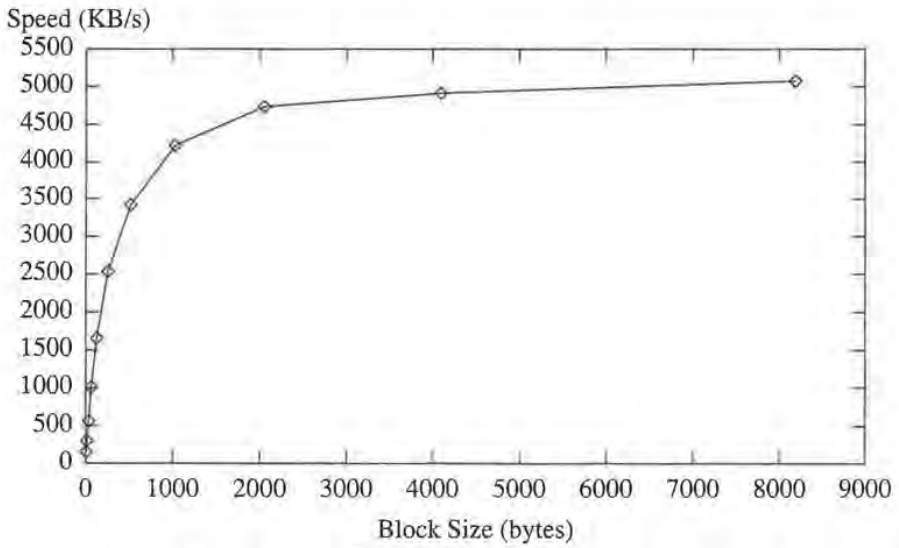


Figure 6.15 The effect of block size on JNI encryption

The end result of all this tuning is unfortunately still not as fast as pure C code, but it's not that far off, either. PureTLS with the GoNative Provider module is roughly half as fast as OpenSSL, depending on your algorithm choices.

6.19 SSL Servers under Load

Unsurprisingly, under heavy load, the cryptographic computations necessary for SSL become the bottleneck for server performance. Thus, a server that is capable of handling a given load can quickly become overloaded if those transactions are transitioned over to SSL. In order to simulate the impact of a large number of clients, we can use a *load generator*. We have a client machine (or machines) which connects to the server and makes requests. Each machine can generate a large number of requests at a given time, thus simulating a large number of users connecting to a given site. In this case, our client is a Pentium II/400 running FreeBSD 3.4.

We could perform this experiment using any SSL-enabled protocol, but for convenience we'll use HTTP over SSL. The traffic mix we've chosen simulates the behavior of a Web site when a large number of people decide to request a page at more or less the same time. In their paper on SSL performance, Abbott and Keung [Abbott1988] report that this is the scenario of most concern to their customers.

To produce this traffic mix, we arrange that each client connects to the server at a different *arrival time*. The arrival times are generated using a Poisson process with a mean of 3 seconds. Thus, all of the clients connect somewhere between 0 and 10 seconds after the start of the simulation.

Figure 6.16 shows the results of such a simulation on an ordinary HTTP server. The server is Apache 1.3.12 with mod_ssl 2.6.5 running under Linux 2.2.14-5.0 (Red Hat) on a Pentium III/650. The bar graph labelled "arrivals" represents the number of simulated client connections within each 1-second window. Thus, 23 connections were initiated in the first second of the simulation. The line graph labelled "queue" is the number of clients that were active during this time window. Thus, a client that connects at 1.5 s and disconnects at 2.5 s contributes to the queue during the 1–2 and 2–3 time windows.

When we examine Figure 6.16, we can see that the queue size closely tracks client arrivals. This is happening because the server is serving each request more or less as it comes in. This is a good indication that the server is handling the load successfully.

Compare this result to Figure 6.17, which shows the server behavior with 550 simulated HTTP/SSL clients. The cipher suite is TLS_RSA_WITH_3DES_EDE_CBC_SHA and the client has been set never to resume sessions, so every handshake requires an RSA operation. Initially, the queue size closely tracks the client arrival times, but as the number of clients increases during the simulation the queue size starts to increase dramatically. Note that although our client is significantly slower than our server, it nevertheless is capable of completely saturating the server machine. This is a nice demonstration of the asymmetry of SSL.

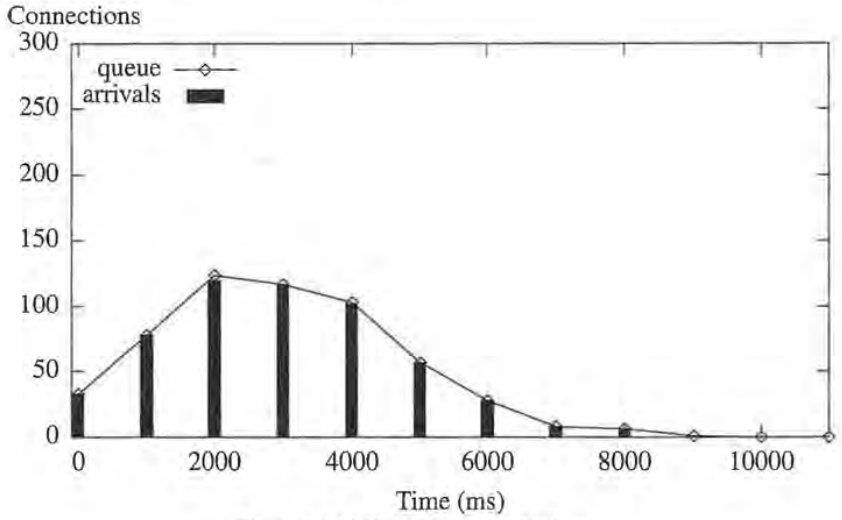


Figure 6.16 550 simulated HTTP clients

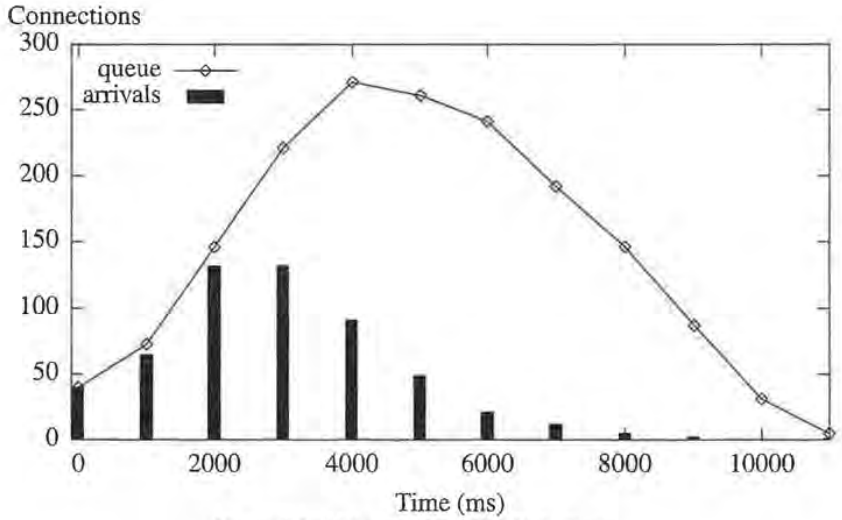


Figure 6.17 550 simulated HTTP/SSL clients

What's happening here is that the server CPU is saturated by the SSL handshakes it is already performing. Thus, more clients are arriving than are being successfully served, creating a backlog of unserved clients. As the number of new arrivals decreases, the server is able to work through the backlog of queued but unserviced connections and so the queue size slowly decreases. However, the server is only able to clear the queue because the number of arrivals trails off.

The server would not be able to handle the peak load shown in Figure 6.17 on a continuous basis. This is demonstrated by Figure 6.18, which shows the server under a continuous load of approximately 75 connections a second. Note that the size of the queue grows linearly with time, indicating that the server is not keeping up with load.

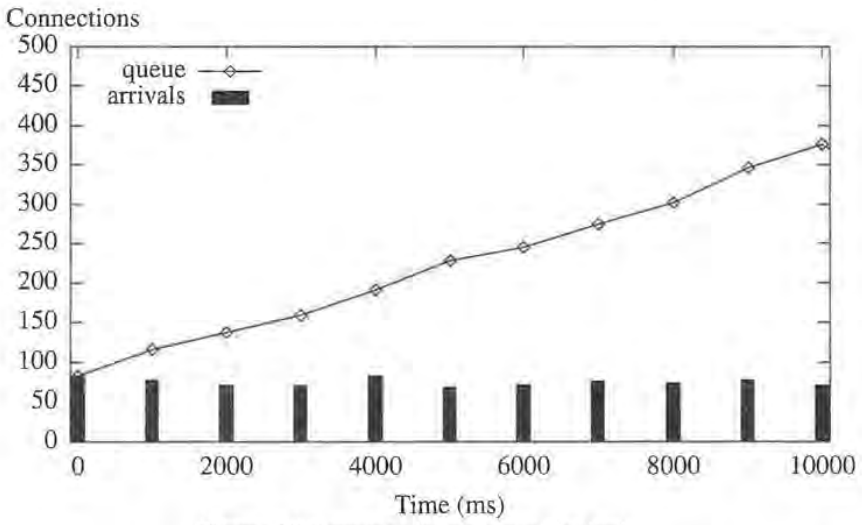


Figure 6.18 A server under continuous high load

6.20 Hardware Acceleration

Because the bottleneck in SSL performance is server CPU, it's natural to consider accelerating the SSL server by offloading the expensive crypto operations to a separate processor. Hardware acceleration not only increases the number of connections that a server can handle but also frees the server CPU to perform other tasks, such as interpreting the CGI scripts that are used to run many Web sites.

Typically, hardware accelerators are packaged as boards that may be plugged into the server machine, usually via the PCI bus on Intel architecture machines. Accelerators can also be attached to the server machine via an Ethernet connection or the SCSI bus. Accelerators are available from several manufacturers but the best known are Rainbow (`http://www.rainbow.com/`) and nCipher (`http://www.ncipher.com/`).

Figure 6.19 shows the same server as in the previous connection accelerated with a Rainbow SSL accelerator, capable of doing 200 RSA operations per second. Again, we use 550 simulated clients. However, with acceleration the queue size once again closely tracks arrival time. The server is handling the load, and thus no backlog is building up.

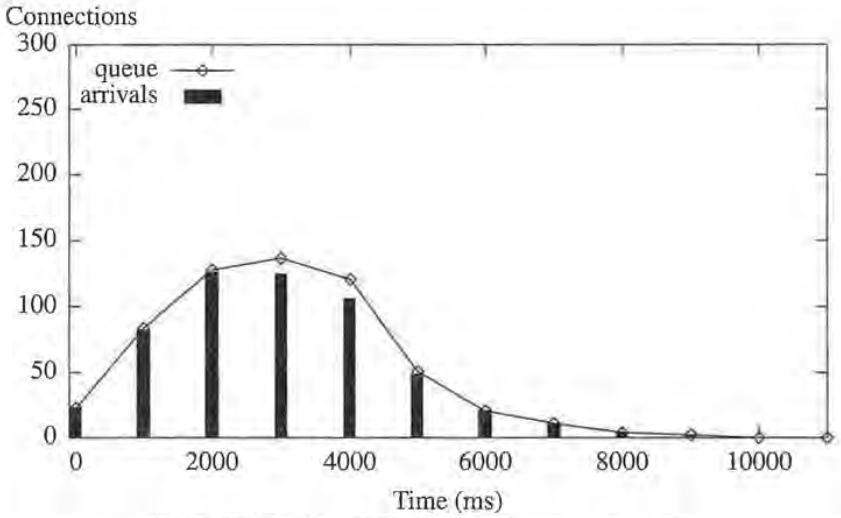


Figure 6.19 550 HTTP/SSL clients (hardware accelerated)

Because the most expensive computation in the SSL handshake is the RSA computation, accelerating RSA adds the most value. Rainbow and nCipher advertise that their high-end boxes can process up to 1000 SSL handshakes per second by accelerating the RSA computation. This acceleration is largely independent of the server CPU as long as it is reasonably powerful. By contrast, our Pentium III/650 system can perform 95 RSA operations per second. In practice, the number of connections per second it can handle will be substantially lower, because in real life the CPU would have to be running the server as well as performing crypto. Thus, 75 simulated clients per second is enough to overload the server.

Accelerators can also be used to offload symmetric operations. In most circumstances, merely offloading the asymmetric operations will be enough but if the network connection is very fast, it pays to accelerate the encryption and message digesting as well. Keung [Keung] reports that throughput nearly doubles when an accelerator is added to a 200 MHz Pentium on a 10 Mb Ethernet. Note that this effect will be less pronounced with modern faster processors. Nevertheless, if your server is handling a lot of secure traffic, it may be worth considering an accelerator to increase throughput. It's certainly worth accelerating some symmetric cryptography if you already have acceleration for asymmetric cryptography. RC4 is already so fast that it's probably not worth accelerating it at all.

Typically, the manufacturers of accelerators also provide drivers that allow popular servers to use their accelerators. A popular approach is to distribute patches or modified versions of OpenSSL that use the hardware device. This allows any OpenSSL-based system to be accelerated.

6.21 Inline Hardware Accelerators

Another approach to hardware accelerating SSL is to use a separate *inline accelerator*. The accelerator is a hardware device which sits between the clients and the servers. It listens for SSL connections from clients and accepts the SSL handshake. It then initiates an ordinary TCP connection to the server. This relieves the server of the requirement to do SSL. In order for this approach to work, the accelerator needs to have the server's keying material.

The primary virtue of an inline accelerator is that it does not require tampering with the server at all. By contrast, hardware acceleration boards must be inserted into the server, with the possibility of destabilizing it. The primary disadvantage of inline acceleration is that the coupling between the client and the server is broken. This makes it difficult for the server to make access control decisions based on the security properties of the SSL connection because there is no good way to communicate them from the accelerator to the server.

Inline accelerators are made by a number of manufacturers, including iPivot/Intel (`http://www.ipivot.com/`), F5 (`http://www.f5.com/`), and Network Alchemy/Nokia (`http://www.network-alchemy.com/`).

Configuration

Consider the problem of inserting an inline accelerator into an environment with an operational server. We must now arrange for connections from clients to terminate at the accelerator rather than the server. One approach would be to give the accelerator its own IP address. Then the DNS records could be adjusted to point to the accelerator rather than the server. Alternatively, we could give the server a new IP address and leave the DNS alone. Note that although the server and the accelerator may be on the same network segment, because traffic between them is unencrypted, that segment must be secure.

A more popular approach is to use a *transparent* configuration. In such a configuration, the accelerator must sit on a separate network physically between the client and server. The accelerator is configured to intercept a certain class of connections as if it were the server. However, it allows all other connections to simply pass through, as if it were a bridge. The transparent approach has the significant advantage that it does not require any changes to address assignments or DNS at all. You simply plug the accelerator into the network and plug the server into the accelerator, as shown in Figure 6.20.

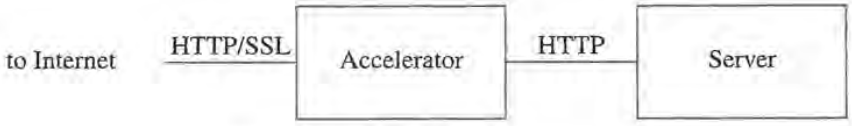


Figure 6.20 An inline SSL accelerator

Multiple Accelerators

A really busy server may have more traffic than a single accelerator can handle, even with hardware cryptography. In order to serve high loads, administrators may wish to use multiple accelerators. An additional benefit of this approach is that if one accelerator box fails, the server will still be able to handle requests, although the total number of requests it can serve will, of course, be smaller. There are two primary ways to use multiple accelerators, "chained" and "clustered."

Chained Accelerators

The first vendor to provide inline accelerators was iPivot, which has since been acquired by Intel, with their Commerce Accelerator. iPivot's box is a transparent proxy only. In order to use multiple boxes, one simply connects them in series, as shown in Figure 6.21.

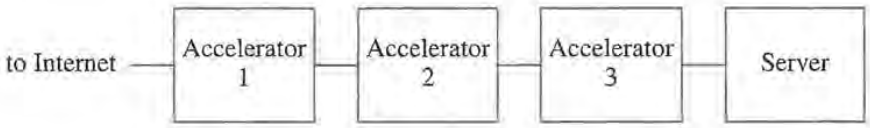


Figure 6.21 Chained SSL accelerators

If the load on a box gets too high, the box simply starts to pass traffic untouched to the next box in the chain. This process continues until all clients have been serviced. The last box in the chain can be configured to either refuse connections when the load gets too high or to pass them on to the server. The server should be configured to support SSL in order to allow it to process such requests.

iPivot's hardware is equipped with a "bypass" feature. If a box fails, the bypass feature goes into effect and all traffic is passed directly through the box as if it were not there. Thus, even if the box fails fairly catastrophically, the server will still be connected to the network and able to serve clients. However, because the server must be configured to accept raw HTTP, use of the bypass feature can lead to the server accepting HTTP connections while thinking that it's accepting decrypted HTTPS connnections. This is an obvious security problem.

Clustered Accelerators

Consider the situation where you have multiple chained accelerators and one box fails. Although the remaining accelerators will be able to handle new connections, all the SSL connections terminated at that box when it failed will be lost. Of course, neither the client nor server will know that this has happened, so when they attempt to send traffic they will get TCP RSTs. The idea behind clustered accelerators is to allow individual connections to fail over between boxes, so that no traffic is ever lost.

Full Disclosure: The author did significant development on Network Alchemy's clustered SSL accelerator under contract. This has the advantage that I am familiar with how it works and that I am able to describe it before it is released. It has the disadvantage that I have a conflict of interest. I have attempted to describe the advantages and disadvantages of the solution objectively, but the reader would do well to bear this fact in mind.

Network Alchemy/Nokia's clustered SSL accelerator, due out in Q1 2001 attempts to solve the problem of failover of active connections. Instead of connecting boxes in a chain, you connect multiple boxes in parallel, as shown in Figure 6.22. This configuration is called a cluster.

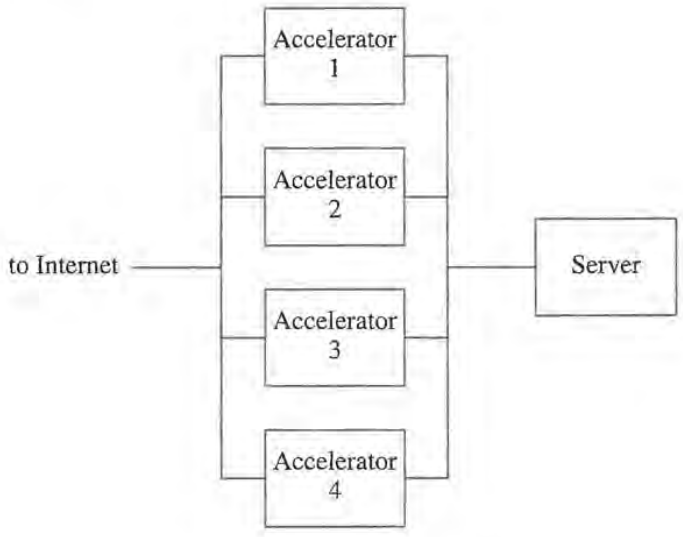


Figure 6.22 Clustered SSL accelerators

The entire cluster has a virtual ethernet MAC address and thus behaves as if it were a single piece of hardware. Each machine in the cluster listens to all traffic destined for that address. The cluster automatically distributes responsibility for handling each section of traffic between the machines in the cluster. If a box fails, its traffic is automatically picked up by the other boxes.

In order to enable failover, the clustered machines automatically propagate connection state from the machine handling the connection to the rest of the machines in the cluster. Thus, if one of the machines fails, the new machine assigned to handle the connection already has all of the connection state and can pick up where the other machine left off. Note that if the machine is handling traffic for a given connection when it fails, that IP packet will be dropped. However, when the packet is retransmitted the machine that is now handling the connection will automatically process it correctly. All the client and server will see is some delay while waiting for the retransmit.

Note that although a clustered solution handles single box failures more cleanly, it handles catastrophic failures less cleanly. As we saw in the previous section, if all the iPivot boxes fail, the network simply behaves as if they were not present. Because we want another box to pick up the load if a box fails, when individual clustered boxes fail, they fail in a disconnected state. Thus, if all the boxes in a cluster fail, the server will be denied service.

However, the probability that all the boxes in a cluster will fail is generally fairly low. The most likely situation would be some sort of power glitch that brought them all down together. In order to avoid this, the various cluster members must be on separate circuits so there is no single point of failure. By contrast, with the iPivot solution, each box is a single point of failure. Although the iPivot boxes are designed to fail safe, some conceivable problems (a broken internal wire, for instance) would bring down the entire system in such a way that bypass would not work.

The clustered solution has a number of other advantages. Because the cluster automatically shares workload, the load can be more easily and evenly distributed between cluster members. This is more difficult to accomplish with a chained solution. Also, the cluster can be configured as a single unit. Each individual box does not need to be configured with all the SSL information. You configure it as part of the cluster and the SSL configuration propagates automatically.

6.22 Network Latency

So far, we've concentrated on how SSL uses the processor, ignoring its interaction with the network. However, the characteristics of the network over which the SSL connection is running have a substantial effect on its performance characteristics. The rest of this chapter focuses on that interaction.

Although theoretically SSL can be run over any connection-oriented protocol, in practice it's nearly always run over TCP/IP. Therefore, we're going to focus almost exclusively on the performance characteristics of SSL running over TCP networks, although some of this material is also applicable to other types of networks.

The data we've been collecting so far comes from programs running on a single machine and communicating directly through the kernel without any network in the way at all. This differs from a real network in that it's much faster. Throughput is very high and latency is very low. By contrast, in a real network, throughput will be lower and latency will be higher.

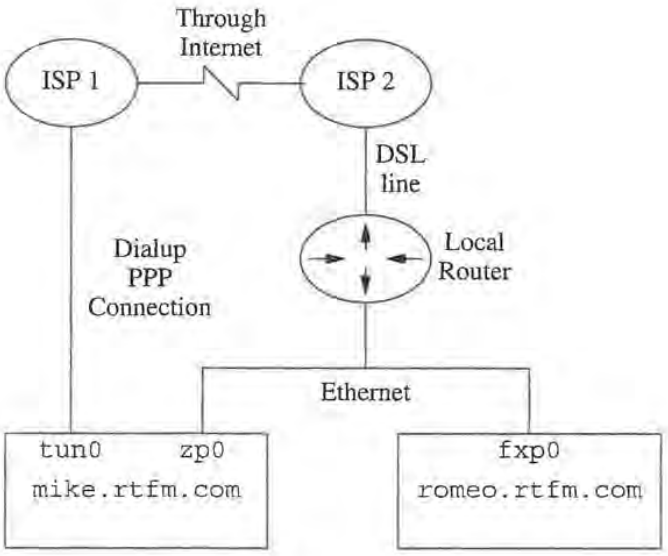


Figure 6.23 A slow test network

To demonstrate the effects of a slower network, we collected a number of traces on the network configuration shown in Figure 6.23. This network has two test machines, `romeo.rtfm.com` and `mike.rtfm.com`. `romeo` is connected to the network as usual, but `mike` has been specially configured to be *dual homed*. It has two IP interfaces. The primary network connection is the PPP dialup on interface `tun0`. All traffic between `mike` and the local network must go over the PPP interface.

However, `mike` is also connected directly to the local network via the Ethernet interface `zp0`. Under normal circumstances, traffic between `mike` and `romeo` would go through this interface. However, we've deliberately configured `zp0` with a private network to force traffic to go via PPP.

The point of this configuration is that `mike` can be configured to capture packets on both `tun0` and `zp0`. Thus, it can see each packet twice—when it appears on the PPP interface and again when it appears on the Ethernet. This allows us to directly observe the time it takes for a given packet to transit the network between `mike` and `romeo`. For instance, when a packet is sent from `romeo` to `mike` it is first transmitted from `romeo` to the local router on the Ethernet. The packet sniffer attached to `mike`'s interface `zp0` captures the packet. The packet is then sent through the DSL line to one ISP, through the Internet to another ISP, and finally down the PPP connection to `mike`. At this point the packet sniffer attached to `tun0` captures the packet.

Figure 6.24 shows a simple SSL handshake performed between `mike` (the client) and `romeo` (the server). Again, the cipher suite uses RSA for key exchange. Unlike

previous timelines, this one is drawn to scale. The vertical displacement of each arrow represents the time that it takes to transit the network. The bars on the left and right of each arrow show the actual time from transmission to delivery. Thus, the ClientHello took 112 ms to get from `mike` to `romeo`.

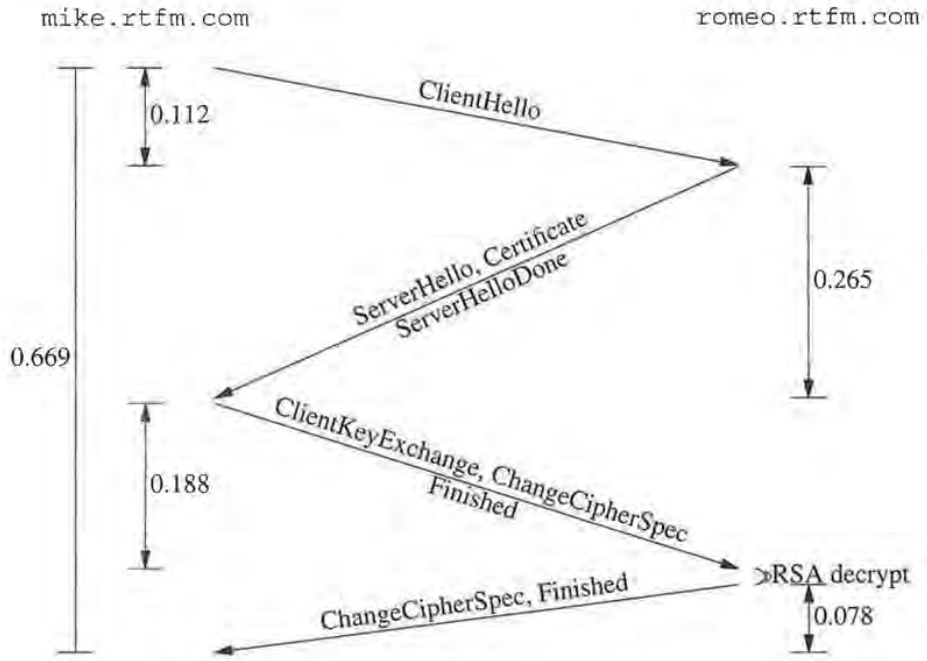


Figure 6.24 SSL handshake over a high-latency connection

Note that the total time of the handshake is vastly increased. In this case, the time from start to finish is 669 ms. A comparable handshake performed on a fast network might take 100 ms. However, the vast majority of the time is simple network latency. Packets are processed nearly as quickly as they are received. The only piece of computation that has any noticeable effect at all is the RSA decrypt, which takes perhaps 20 ms—an insignificant fraction of the total time for the handshake.

It's interesting to note that the transit time for the various handshake messages is wildly different. The combined ServerHello, Certificate, and ServerHelloDone takes 265 ms, whereas the server's ChangeCipherSpec and Finished take only 78 ms. This result is surprising at first but is easily explained: most of the latency in the system is introduced not by the Internet between the ISPs but by the PPP connection between `mike` and the first ISP. On a slow modem link, latency is dominated by transmission time. The larger the packet the longer it takes to transmit and thus the higher the latency. The Certificate is large and therefore introduces the most latency.

Ex.1024
Askeladden, L.L.C. / Page 281 of 788

Note that although increased network latency slows down individual handshakes, it generally doesn't decrease server throughput. Provided that the server's network link isn't saturated, while the server is waiting for a response from one client it can be serving others. However, this isn't to say that high handshake latency is unimportant. Because it's perceived by the user as poor performance, if we can reduce latency we can improve the user experience.

6.23 The Nagle Algorithm

Under certain circumstances, SSL interacts poorly with various TCP congestion control measures. This interaction can cause the SSL handshake to stall even though both the client and server are idle. The culprit is Nagle's algorithm described in [Nagle1984].

The Nagle algorithm is intended to reduce *tinygrams*: small packets on the wire. The problem is the TCP packet header. TCP segments have a minimum header size of 40 bytes. Thus, when one character is sent per segment, 41 bytes must be sent to the network. This can rapidly lead to congestion. Under normal circumstances, whenever programs call `write()` (or equivalent), the TCP stack simply sends out the data immediately. However, if programs call `write()` with small data segments (such as when a user is typing) and kernel sends out one packet for each such write, the network can get clogged with such packets.

The Nagle algorithm prevents this in a very simple way. When there is outstanding data that has been sent but not acknowledged, the TCP implementation does not send small packets. Instead, it buffers the data and sends the entire buffer when it receives the ACK. Under normal circumstances, this works very well, but it can cause a problem with SSL, as we'll see shortly.

The second cause of the problem is a technique called *delayed ACKs*. Rather than generating an ACK as soon as a packet is received, the TCP implementation tries to *piggyback* it on a data segment it is already sending out. To facilitate this, the TCP implementation will wait for up to 200 ms for data to be written. If none shows up within that window, then the ACK will be written to the network anyway.

A good example of this interaction is what happens when SSL sessions are resumed. Figure 6.25 shows a network trace of such a connection. Note that although the client and server are on the same machine, the data still goes through the TCP stack and so we can observe the behavior of the Nagle algorithm. The important thing to notice here is the 148 ms delay between the client sending its Finished message (line 6) and the first application data record (line 7). This is far longer than it takes to prepare the record.

What's happened here is that the client TCP stack is waiting for the Finished message to be ACKed before it sends out the first application data record (due to the Nagle algorithm). The server TCP stack isn't sending out the ACK because it's implementing delayed ACK. But the server program isn't writing because it's waiting for the client to send its first application data record: We've got a deadlock.

The deadlock is finally broken at 0.1516 seconds, when the server's delayed ACK timer goes off. The server sends the ACK out anyway (between line 6 and 7) even

```
New TCP connection: localhost(2830) <-> localhost(4433)
1 0.0003 (0.0003) C>S   Handshake      ClientHello
2 0.0028 (0.0024) S>C   Handshake      ServerHello
3 0.0028 (0.0000) S>C   ChangeCipherSpec
4 0.0028 (0.0000) S>C   Finished
5 0.0039 (0.0011) C>S   ChangeCipherSpec
6 0.0039 (0.0000) C>S   Finished
  TCP ACK arrives at 0.1516
7 0.1517 (0.14777) C>S   application_data
8 0.1530 (0.0014) S>C   application_data
```

Figure 6.25 The Nagle algorithm and session resumption

though it can't piggyback it. The client gets the ACK and at 0.1517 seconds it flushes its own buffers and sends the first record of application data. (Note that for clarity we've opted not to show the piggybacked ACKs on the data packets.)

Disabling the Nagle Algorithm

It's possible to disable the Nagle algorithm. (The sockets API uses the `TCP_NODELAY` option). If we disable it, then data will be written to the network as soon as it's presented to the TCP stack (provided there's room in the TCP window, of course). The effect is to avoid the sort of deadlock we saw in Figure 6.25. Figure 6.26 shows the same SSL handshake with the Nagle algorithm disabled on the client. As you can see, the client sends the application data immediately after sending the Finished message, drastically reducing the time it takes to complete the handshake.

```
New TCP connection: localhost(2850) <-> localhost(4433)
1 0.0006 (0.0060) C>S   Handshake      ClientHello
2 0.0016 (0.0010) S>C   Handshake      ServerHello
3 0.0016 (0.0000) S>C   ChangeCipherSpec
4 0.0016 (0.0000) S>C   Finished
5 0.0028 (0.0012) C>S   ChangeCipherSpec
6 0.0028 (0.0000) C>S   Finished
7 0.0046 (0.0018) C>S   application_data
8 0.0059 (0.0013) S>C   application_data
```

Figure 6.26 Session resumption with the Nagle algorithm disabled

The Right Layer

The obvious question to ask at this point is whether it's ever worth using the Nagle algorithm for SSL. In general, the answer is probably no. The purpose of the Nagle algorithm is to reduce the TCP overhead associated with the transmission of small datagrams.

However, SSL records are almost never small. Due to the addition of the header and the MAC, the absolute minimum size for an SSL data record is 22 bytes. Thus, if the SSL implementation packages one byte per record, we've already suffered an expansion of twenty-fold before we ever get to the TCP stack. Moreover, as we saw earlier, processing records one byte at a time is very cryptographically expensive.

Therefore, if we want to solve the tinygram problem for SSL we have to arrange to buffer data before the SSL implementation packages it into records. Exactly what is required depends on which protocol is being run over SSL, and we'll see more about that in Chapter 7.

6.24 Handshake Buffering

As we discussed in Section 6.10, some implementations buffer parts of the SSL handshake. OpenSSL uses a 1024-byte buffer. Most of the time this works fairly well, but there are a number of pathological cases where another strategy would produce significantly lower latency. Consider the DSS/DHE handshake shown in Figure 6.27. This handshake was carried out over a 10 Mb Ethernet, so transmission latency is minimal. However, we can still be hurt by the Nagle algorithm. The ServerHello, Certificate, and ServerKeyExchange cannot all fit in a single 1024-byte buffer. Thus, the ServerKeyExchange gets split into two pieces. The first piece is transmitted along with the Certificate. The second piece is buffered and sent to the operating system when the buffer is flushed after the ServerHelloDone.

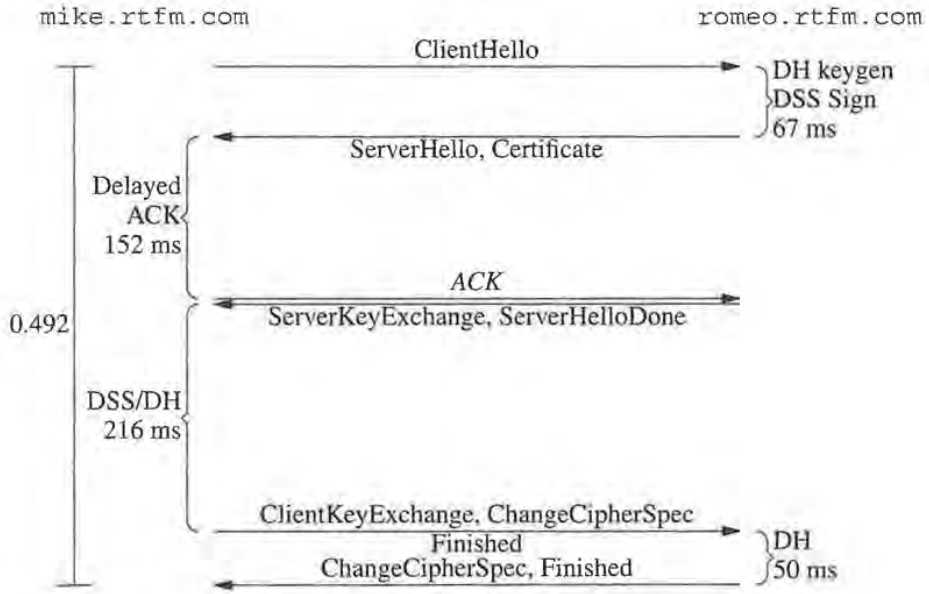


Figure 6.27 DSS/DHE with OpenSSL's normal buffers

However, because the buffer size is 1024 bytes, the first segment containing the ServerHello, Certificate, and part of the ServerHelloDone is also 1024 bytes. This is smaller than the Ethernet *maximum segment size* (MSS) and therefore this is considered to be a small packet for the purposes of the Nagle algorithm. Thus, the next segment containing the rest of the ServerKeyExchange can't be transmitted until the previous segment is ACKed. As described in the previous section, the client delays the ACK for 152 ms. During this period, both client and server are idle. As soon as the server receives the ACK it transmits the rest of the ServerKeyExchange and the ServerHelloDone.

Several features of OpenSSL conspire to produce increased latency in this handshake. First, the interaction of OpenSSL's small buffer size and the Nagle algorithm create a totally idle period where the server is waiting for an ACK and the client is performing delayed ACKs. The simple fix for this is simply to increase the buffer size. Figure 6.28 shows the same handshake except that the buffer size has been increased to 4096 bytes.



Figure 6.28 DSS/DHE with an expanded buffer

Now that we've increased the buffer size all four server messages fit in the same buffer. Thus, they are all sent at the same time. This eliminates the condition where the server waits for the client's delayed ACK, thus avoiding the temporary deadlock we saw previously. As a consequence, this handshake takes only 352 ms to complete instead of the 492 ms that we saw with the smaller buffer size.

Note that any fixed buffer size is a clumsy solution to this problem. No matter what buffer size choice we make, it's possible that the certificate will be large enough to fill it. A more tasteful solution would be to use a variable length buffer that automatically resizes and then flush the buffer manually after transmitting the ServerHelloDone. Note

that it's possible that the data may be so large that it is transmitted in more than one TCP segment. However, this will not trigger the Nagle algorithm because all but the final segment will be full-sized.

An alternative to increasing the buffer would be to simply turn off the Nagle algorithm entirely. This has much the same effect as increasing the buffer size. It does increase server load very slightly because two calls to `write()` must be made instead of one. However, the high CPU cost of the rest of the SSL handshake renders the cost of a single additional context switch insignificant by comparison.

Increasing Parallelism

The second factor contributing to the latency of this handshake is a lack of parallelism between client and server. Essentially all the time that the server spends in the first phase of the handshake is consumed generating the ServerKeyExchange message. Only after the ServerKeyExchange is generated are any messages sent from the server. Even when we use a 1024-byte buffer, it's the ServerKeyExchange that fills up the buffer and forces the ServerHello and Certificate to be transmitted.

During the generation of the ServerKeyExchange, the client is completely idle. However, there is a task that it could be performing during this time: verifying the server's certificate. Recall from Section 6.14 that DSA verification costs a significant amount of CPU time. We could reduce latency if the client performed this verification while the server was generating the ServerKeyExchange. In order to do this, we must arrange that the server sends the ServerHello and Certificate before generating the ServerKeyExchange. A simple approach would be to modify the server to send each message separately. A more sophisticated approach would be to flush the write buffer after writing the Certificate message. In either case, the Nagle algorithm would need to be turned off.

With the server suitably modified, we should expect the client to verify the server's certificate entirely before it receives the ServerKeyExchange. Thus, latency should be reduced by the time it takes to verify the server's certificate. Experimentation with a modified client reveals the expected 20 ms improvement.

The DSS/DHE case is extreme in that significant processing is required on both client and server and thus parallelism offers significant performance gains. However, it demonstrates a general principle: SSL implementors need to be carefully aware of the interaction between their implementation and the network in order to get maximal performance. In particular, whenever one side is transmitting data that requires significant processing by the other side, that data should be written to the network as soon as possible to maximimize parallelism and thereby reduce latency. A good rule of thumb is to flush network buffers before entering any time-consuming operation.

6.25 Advanced SSL Performance Rules

Avoid ephemeral keying. Ephemeral DH slows down the server. Don't use it unless you need Perfect Forward Secrecy. Even DHE mode with a fixed DH key is an improvement.

Use short DH exponents. A 256-bit DH exponent improves performance dramatically with a trivial reduction in security.

Nagle algorithm. Always turn off the Nagle algorithm.

Handshake buffering. Transmit data before entering any significant computation. Beware of the interaction of buffer flushes with Nagle.

6.26 Summary

This chapter has provided a description of the performance properties of SSL. We started with some general observations about the factors controlling SSL performance and then covered these points in more detail using instrumented software and network traces to illustrate the important points.

Most of the time goes to cryptography. Cryptographic operations are the dominating factor in SSL performance. The protocol state machine and I/O contribute negligibly to the performance of the system.

Handshakes are very expensive. For sessions that don't transmit a lot of data, the SSL handshake consumes most of the CPU time.

Session resumption can reduce the cost of handshakes. The handshake for a resumed session is much faster than the handshake for an ordinary session. If a client/server pair are going to communicate repeatedly, resuming sessions pays off.

Server performance is more important than client performance. Slow servers increase latency and reduce throughput. Slow clients mainly increase latency.

Use large records. There is a certain minimal amount of cryptographic and network overhead consumed to process any record. The larger the record size, the fewer times this overhead is incurred.

Algorithm choice matters. For fast handshake performance, use RSA. For fast data transfer use RC4. Digest algorithm performance doesn't make a big difference.

If you want acceptable performance, use C. Java crypto performance is abysmal. The performance of Java implementations can be dramatically improved by calling to C using JNI.

If you want high performance, use cryptographic hardware. For high-load servers, hardware cryptographic engines can substantially improve performance.

7

Designing with SSL

"First, figure out what you are trying to do (this is good advice under most circumstances, and it is especially apropos here)."
—NNTP Installation Guide

7.1 Introduction

So far we've focused on the mechanics of SSL itself. However, SSL isn't very useful in and of itself. Rather, it's used to carry application-layer protocols. It's these protocols that perform services for users. SSL's job is to allow those protocols to do their jobs securely while abstracting away as much security functionality as possible. SSL takes care of the mechanics of security in a generic fashion and the application-layer protocols run on top and take advantage of the security services it provides.

In practice, however, simply carrying the application data over an SSL connection often gives undesirable security and performance properties. Application-layer protocols (and protocol designers) need to know enough about SSL to be able to run on top of it securely. This chapter covers what you need to know in order to accomplish this task. As usual, we've broken the material up into two parts, with the first covering the basic concepts in overview form and the second covering some of the more technical material in more detail as well as introducing some fine points that were glossed over in the initial presentation.

All the material in this chapter addresses the problem of securing an application protocol from a protocol-level perspective. This presentation is aimed at the application protocol designer and system architect. Chapter 8 serves as a companion to this chapter and covers these techniques from a programmer's perspective, providing sample code which uses commonly available SSL implementations.

Ex.1024
Askeladden, L.L.C. / Page 289 of 788

7.2 Know What You Want to Secure

The first thing you need to figure out is what security services you want to provide. In order to do that, you need to evaluate the threat model for your application. If you don't remember what a threat model is, go back and reread Chapter 1. With the threat model in hand, figure out what are the vulnerabilities of the system you're trying to secure. Your aim should be to secure all the vulnerabilities that can be secured economically.

As a template for this exercise, recall the three security services discussed in Chapter 1: *confidentiality*, *message integrity*, and *endpoint authentication*. Not every application will need all of these services (though an application that doesn't need any isn't worth securing), and to some extent they can be provided independently. Thus, it's important to identify which services you need. Part of this process is identifying the cost of providing those services. The cheaper the service, the smaller a need justifies providing it.

It's not necessary that all security services be provided by SSL. For instance, it's quite common for systems to provide client authentication using a username and password. Although the password encryption is provided via SSL, the application protocol is responsible for transmitting, receiving, and checking it. At this point you want to identify which security services are best provided by SSL and which are best provided by the application protocol.

Confidentiality

Confidentiality is useful in most applications. However, as we saw in Chapter 6, it comes with a nonzero performance cost, particularly if RC4 is unavailable. Moreover, there are some applications that do not need confidentiality. A good example is free software download. When you're downloading software, you want to know that it arrives unmodified from the vendor, but because anyone else can download it, there's no value in keeping the contents secret.

Message Integrity

Message integrity is a required part of SSL. Moreover, it's difficult to offer any security services without message integrity. Thus, there's really not any question about whether to use the message integrity feature of SSL.

Server Authentication

The only SSL modes that don't require server authentication are the anonymous DH modes. However, it is of course possible to use self-signed (and hence unverified) certificates and get the effect of an anonymous connection with any SSL mode. This is undesirable because it leaves the connection open to a man-in-the-middle attack. Therefore, in general, server certificates are required for nearly all SSL applications.

There are certain special cases where a man-in-the-middle attack can be prevented

by the application-layer protocol. We'll discuss a technique for doing this in Section 7.16. These techniques typically require an especially tight integration between the application protocol and SSL.

Client Authentication

As we've said, using SSL pretty much locks you into certificate-based server authentication. However, certificates are not the only option for client authentication. Although SSL itself provides support only for certificate-based client authentication, many application protocols incorporate their own client authentication mechanisms, which can be run over SSL. Although these mechanisms may have been insecure when run over the net in the clear, they are often far more secure over SSL. The next section discusses some of the simpler client authentication options.

Rule of Thumb

As we've said, message integrity isn't optional, so there's no choice to be made there. In general, it's best to provide confidentiality. Many SSL implementations do not support the authentication-only modes, so you'll need to support confidentiality in service of interoperability. Moreover, it matches user expectations. Client authentication is in general a hassle for users, so avoid it unless you have some data that really needs to be restricted to specific users.

7.3 Client Authentication Options

Username/Password

The most traditional way to do client authentication is to use *username/password*. The user supplies a username and password to the network client, which in turn supplies it to the server the user wants to access. The server checks the user's password against its password database. If the password checks, then the network client is authenticated as the user.

This method is often combined with an *Access Control List* (ACL) mechanism. An ACL is simply a list of users and the permissions each user has. Each resource on the server has an ACL associated with it and when a client requests access to a resource the server checks whether the user associated with that client has permission in the appropriate ACL.

Username/password methods are vulnerable to a simple passive attack: the attacker sniffs the password off of the wire and then initiates a new session, presenting the username/password pair as his own. SSL prevents this attack because the password is sent over an encrypted channel. However, it's still possible for the attacker to mount an active attack by repeatedly guessing passwords, hoping to get a correct one.

Username/Password Variants

Because so many systems have been designed with username/password security systems, a number of complicated schemes have been designed to retain the general username/password model while improving security. The simplest example is one-time password schemes such as S/Key or SecureID cards. In general, these schemes are designed to make passwords safe in the face of arbitrary passive or active attacks on the connection carrying the password. Because SSL prevents such attacks, these schemes are generally without advantage in environments where SSL is being used. However, in a system with both insecure and secure connections, such schemes may be useful. Two examples of such schemes are described in [Jablon1996] and [Wu1998].

SSL Client Authentication

As with server authentication, client authentication in SSL depends on possession of a digital certificate. However, because of the large client-to-server ratio, this presents a larger operational challenge for client authentication than does server authentication. Certifying large numbers of clients often requires a special arrangement with a CA. To avoid making such an arrangement, many organizations choose to run their own certificate authorities, but this too presents an operational burden.

An additional problem with certificate-based client authentication is mapping certificates to user identities. ACLs can be used with certificates provided that you arrange to look up ACL entries based on the identities in certificates. This requires particular care in cases where certificate-based authentication will be used together with username/password-type authentication schemes, because the system must arrange to map both types of identity information into a common user format.

Rule of Thumb

In most circumstances it's easier to offer clients username/password-style authentication. Passwords are easier to integrate with most infrastructures and they're easier for users to understand. As an exception to this rule, automated clients are often just as easy to manage with certificates, particularly if no access control is required. A good example of such a client is a mail server, which runs unattended and should be able to authenticate itself when forwarding messages.

7.4 Reference Integrity

Server Identity

As we discussed in Chapter 5, ensuring a secure connection with SSL absolutely requires verifying the server's identity. Without this check, an active attacker can mount

a man-in-the-middle attack on the connection. Moreover, it's imperative not only that you verify the server's certificate but that you verify that the server is the server you're expecting to talk to. Otherwise, it's possible for one server with a valid certificate to intercept connections verified for another.

Naturally, comparing the server's expected identity to the server's actual identity requires having some expectation about the server's identity. Thus, whatever application protocol you're using must allow you to have some expectation as to the server's identity.

Typically, this information comes in the form of the server's domain name, which must be matched against the certificate as described in Chapter 5. The domain name is usually supplied either directly by the user (e.g., the host name in a Telnet request) or indirectly via a URL. In cases where neither of these is possible, the situation becomes much more difficult. We'll consider a workable approach for such a case in Section 7.16.

Security Properties

For security reasons, the client needs to verify that the connection properties meet its security expectations. The exact details of these expectations will depend on the type of reference that the client has for the server. For instance, with a protocol such as Telnet the client may simply have the domain name of the server. However, with a protocol such as HTTP, the client often has an indication in the URL that the protocol should be HTTP over SSL (the URL begins with `https://`).

If the reference indicates that SSL is to be used, the client needs to check that it in fact was used. If a domain name is available, the client needs to check whether the domain name in the server's certificate matches the domain name in the reference—if the server cannot be authenticated somehow, then a man-in-the middle attack is possible.

It's important to note that enforcement of secure connections must generally be done on the client side. If the client doesn't enforce it, an attacker can man-in-the-middle the client's insecure connection and initiate a secure connection to the server. Thus, the server thinks it has a secure connection and can't tell that it's to the attacker and not to the client. This sort of attack makes server enforcement of security properties extremely problematic. The one case in which this can't happen is if the server demands SSL client authentication. If SSL client authentication is used, the attacker can make a connection to the server but cannot pose as the client.

Rule of Thumb

In general, you need to provide a reference that (1) concretely identifies the server the client can expect to connect to and (2) indicates that SSL is required. The server's identity needs to be able to be automatically compared to the server's certificate. Servers should be identified via domain name, and that's what should appear in certificates.

7.5 Inappropriate Tasks

Although SSL is useful for a wide variety of security tasks, there are also a fair number of jobs for which it's inappropriate. Recall that SSL provides a secure channel between two machines and that this channel is oblivious to the data passing over it. This creates some limitations on what can be done with it. This section describes those limitations. Chapter 11 provides a brief overview of some other technologies that can fill in these pieces of the puzzle.

Nonrepudiation

One common task that SSL cannot be used for is to provide *nonrepudiation* of data. Consider making an online purchase. The transaction is performed over an SSL connection so you can be certain that you are talking to the correct merchant. Typically, the merchant gives you a receipt and because that's carried over an SSL connection you can be sure that it hasn't been tampered with. However, you can't show the receipt to anyone else and have them externally verify its contents. Thus, if the merchant claims that you forged the receipt you can't prove otherwise.

End-to-End Security

Because the SSL handshake is interactive, it's not possible to send encrypted or authenticated data to a machine to which you cannot establish a network connection. Thus, if your machine is behind a firewall, you have to punch a hole in the firewall if you want to establish an SSL connection to the other end. The other option is to establish an SSL connection to the firewall and let it establish its own connection to the end machine. This approach has the major drawback that the firewall can now read all your traffic.

In general, SSL is bad at any task that involves protecting objects in such a way that they can be stored with their cryptographic properties intact and then recovered, verified, etc., at some later time. If you have such an application, you should strongly consider the possibility that using SSL is an inappropriate approach. A better approach is to use a message security protocol such as S/MIME. We'll discuss this sort of approach in Chapter 11.

7.6 Protocol Selection

Quite commonly, it's desirable to be able to run both an insecure and a secure version of the same application protocol at the same time. If this capability is desired for a given protocol, there must be some way to differentiate between connections that are going to use the native (unsecured) and the secured version of the protocol.

We have two requirements for parallel secure and insecure operation: first, we must be able to unambiguously distinguish secure and insecure connections. Second, security-oblivious clients and servers must be able to interoperate correctly with security-aware clients and servers.

The second requirement deserves a little explanation. Naturally, agents that know nothing about security cannot interoperate securely with agents that expect security. However, an attempt by one agent to use security with a security-oblivious agent must not break the oblivious agent. It's obviously permissible for an error to be thrown, but it should be an understandable error.

One approach would simply be to have the server auto-detect based on the first bytes sent by the client. This approach depends on the SSL ClientHello and the application protocol being sufficiently different that the server can distinguish them. This is not the case for all protocols.

Moreover, even for situations where they are distinct, this approach requires the server to be able to auto-detect the difference. This requirement makes retrofitting SSL to an application protocol difficult because preexisting servers will not know about the SSL handshake and will therefore return errors, or stall, which can be worse. Although auto-detection might be acceptable (though tasteless) when designing a new protocol, it's completely unacceptable when securing an existing protocol.

There are two common approaches to protocol selection: the *separate ports* approach and the *upward negotiation* approach.

Separate Ports

Most Internet protocols have what's called a *well-known port*. For instance, Telnet connections are on port 23 and SMTP (mail) connections are on port 25. See [IANA] for a list of all the currently assigned port numbers. This means that traffic for that protocol is expected to arrive at the server on that port. The *Internet Assigned Numbers Authority* (IANA) controls registration of these port numbers, thereby ensuring that no two registered protocols can share the same port. Obviously, if two different protocols were to collide on the same port, we could encounter the sort of errors that we discussed in the previous section.

A separate ports strategy, then, means that we assign a different TCP port to the secure version of the protocol from the insecure one. A server can figure out whether the connection is to be secure or insecure simply by seeing which TCP port the client is connecting on. It then knows whether to expect the SSL ClientHello or to start processing the application protocol immediately.

The separate ports strategy's main problem is that it doesn't scale well. The number of available ports is large (around 65000) but not infinite. If we define two ports for each protocol the number space will be depleted twice as quickly. Moreover, this sets a dangerous precedent. Imagine that we wish to add some other new feature. Will we define a new port for it as well? Do we need to define a nonsecure and secure version of this new feature? The potential for combinatoric explosion is substantial.

Upward Negotiation

Normally, when we want to add new protocol features to an existing protocol, we find some way to introduce them into the existing protocol in a backward-compatible fashion. Only when major protocol changes are afoot do we use a new TCP port. For

instance, in SSL we can add new ciphers quite easily simply by defining new cipher suite numbers and their associated semantics. The protocol automatically determines whether these ciphers are usable or not.

Many other protocols allow for extensions of this sort. Therefore, it's natural to use these mechanisms to introduce security. In a typical such mechanism, one side will indicate that it supports SSL and the other side will indicate that it wants it. The SSL handshake will then occur and the rest of the application protocol will run over the secured connection.

Figure 7.1 shows a typical such negotiation in abstract form. In the first message, the client indicates that it's prepared to speak TLS. In the second message, the server agrees to speak TLS and tells the client to go ahead. In the next message, the client sends its ClientHello. The client and server complete the handshake and then the client sends the first application protocol message over the new TLS connection. This message is shown in italics to indicate that it's encrypted.

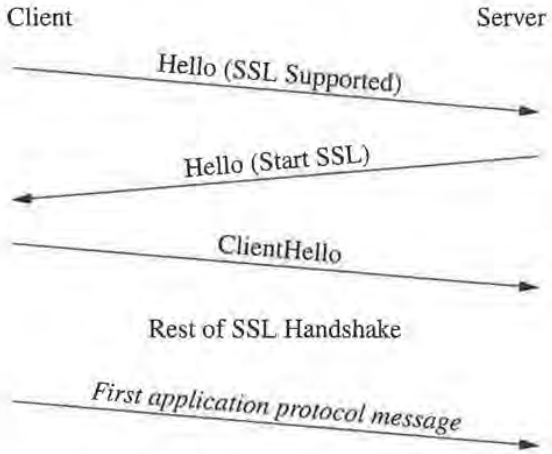


Figure 7.1 Upward negotiation to TLS

Upward negotiation requires that the protocol already have some mechanism for negotiating extensions. This is a common but by no means universal feature. If the protocol does not have such a mechanism, one will have to be inserted. Because the changeover to SSL happens during a normal protocol exchange, more care must be taken to ensure adequate security. No sensitive data should be passed until the SSL session has been negotiated and clients and servers may need to be configured to operate correctly (and safely) if the other side rejects the switch to SSL.

Rule of Thumb

In general, it's much easier to design and implement a separate ports strategy. If you're designing a protocol for largely internal usage or you're working with a protocol you can't change (not an uncommon case) use separate ports. The only time when it's really worth paying the design and implementation cost for an upward negotiation strategy is when you're designing for broad standardization.

7.7 Reducing Handshake Overhead

Recall from Chapter 6 that the primary performance cost in most SSL connections is the handshake. As an application protocol designer, the primary thing that you can do to reduce this overhead is to minimize the number of new SSL handshakes required by your protocol.

If you're designing a new protocol, reducing new connections is a first-order consideration. However, even if you're simply securing a preexisting protocol, it's important to bear it in mind. Some protocols (e.g., HTTP and SMTP) can process multiple transactions over the same protocol connection. This is a good idea in general, but is especially important with SSL.

7.8 Design Strategy

So far, we've discussed the major design considerations in securing a protocol using SSL. This section provides a broad overview of the design steps you should be going through and the order you should be going through them.

Identify your threat model. The first step in any security design is to figure out what sorts of security threats you're trying to protect against. You need to figure out the ways in which your data is sensitive. This procedure leads directly into the next step.

Figure out what you want to secure. First, figure out which security services you wish to provide in the abstract. Next, figure out which ones you can provide using SSL and which ones will have to be provided externally. Now is the time to consider the possiblity that SSL is inappropriate for your application.

Choose a protocol selection strategy. The first design decision you need to make is how to select secure versus insecure mode. As we've discussed, a separate ports strategy is easier to implement, but has some scalability problems.

Decide how (whether) to offer client authentication. If your threat model involves authenticating users, you'll need to use some sort of client authentication. If your application already uses username/password authentication, you may want to reuse this authentication model. Even if you choose to use certificates, you may want to leverage the username/password infrastructure for access control.

Identify servers and provide reference integrity. The next step is to figure out how servers will be authenticated. Almost certainly this will be done with certificates, but the

important task is to figure out how to compare references with the identity in the certificates. This requires identifying what sort of references you have and arranging for the certificates to mirror them.

Optimize connection semantics. Finally, you want to take a look at performance considerations. At this protocol layer most of what you can do is arrange that the number of connections is kept to a minimum and that sessions are resumable where possible. Moreover, you want to make sure that closure happens correctly, both because this reduces rehandshake and to provide security against truncation attacks.

7.9 The Story So Far

The first part of this chapter dicussed general strategy for securing protocols using SSL. In general, it's possible to do a very basic job of securing a protocol with little effort, but there are a number of important details that should be considered. As we've seen, failure to consider some of them leaves substantial security holes while others merely result in bad performance.

The purpose of the first part was to provide an overview of the required tasks. The second part of this chapter discusses in detail various strategies for executing them. Most of the strategies that we'll discuss have been deployed effectively in one or more protocols and are part of the standard SSL design toolbox.

7.10 Separate Ports

The most popular method of operating parallel secured and unsecured servers is to use separate ports for each protocol variant. Implementing a separate ports strategy is trivial. You simply assign a new port to the secure variant of your protocol and arrange for your server to listen on both the non-secure port and the secure port. This can be implemented by having one server process listen to both ports or one process for each port. There's no need to be able to distinguish SSL traffic from ordinary traffic, because anything that comes in on the secure port must be SSL. Any insecure traffic that comes in on the secure port results in an error, as does any secure traffic on the non-secure port.

The only tricky issue is arranging for the client to behave correctly. First, the client has to know to use SSL. Second, it has to know to go to the secure port. If the secure port is well-known then the choice of port may be obvious, but otherwise the client will have to be specially notified. Finally, the client needs to know how to behave if it can't negotiate a secure conection. Does it fall back to an insecure connection or report failure? We'll cover this issue a little later in the chapter when we discuss reference integrity.

The primary advantage of this strategy is that it doesn't require any real modification to the application-layer protocol. You need to describe its behavior over SSL but all the protocol commands stay the same. Moreover, extant unsecured implementations simply work as is. There's no risk of bizarre interactions between secure and insecure implementations because they never interact at all.

The primary weakness of a separate ports strategy is that it consumes TCP ports twice as fast as an upward negotiation strategy. This is of special concern because many protocol designers prefer well-known ports with numbers less than 1024. On UNIX machines, only processes with effective user ID of 0 (the root user) can open ports with port numbers smaller than 1024. Thus, receiving a packet from a port less than 1024 was thought to imply that the sender had the permission of the root user. Similarly, connecting to a server on a low-numbered port is taken to mean that the server is officially sanctioned by the machine's owner because only a process owned by the root user could listen on that port.

Obviously, in the face of the Internet threat model, this is a laughable distinction. It's trivially easy to forge packets with low port numbers. Moreover, many other operating systems do not enforce the low-port-number convention and there's no way to tell whether you're talking to a machine with such an OS or not. Nevertheless, many designers have a sentimental attachment to low-numbered ports and worry about depleting them.

Another problem with this strategy is that it requires modifications to some firewalls. Many packet filtering firewalls are set not to allow TCP connections except on specific ports. Such firewalls will have to be modified to support this strategy. Finally, if you're designing a protocol for IETF standardization, you should be wary of a separate ports strategy. As of this writing, the IESG was strongly encouraging protocol designers to use upward negotiation.

7.11 Upward Negotiation

An upward negotiation strategy uses the same port for secure connections as for nonsecure connections. This obviously has the advantage that it doesn't require allocating an extra port, eliminating the problems associated with that.

Furthermore, upward negotiation adds an important new feature: automatic discovery. Either the client or server can offer security, and if the other side accepts, then it can be automatically negotiated. By contrast, a separate ports strategy more or less requires the client to know that security is available. This feature adds substantial resistance to passive sniffing attacks. However, if not carefully implemented, it can make the protocol vulnerable to active downgrade attacks, as we'll see later.

An upward negotiation strategy requires substantial modifications to the code on both client and server. If the protocol doesn't have a mechanism for adding extensions, one must be added. If it does, support must be added for negotiating security. This doesn't always work correctly the first time it's designed; the mechanism in HTTP, called Upgrade, had not really been used prior to its use for TLS, and some problems were encountered as a result, as we'll discuss in Chapter 8.

Second, it's important to define the semantics of a failed handshake. Simply because both client and server support SSL does not mean that they support compatible cipher suites or keying material. If the SSL handshake fails, then the applications can disconnect and reconnect insecurely, try to continue with an insecure connection without reconnecting, or simply signal an error. The second approach requires being sure that

the network buffers on either side are in a well-defined state so that SSL data is not inadvertantly processed as application protocol data.

Upward negotiation can come at a performance cost as well. When secure and insecure protocols run over separate ports, then the first packet of data is the ClientHello. However, if upward negotiation is employed, then the client and server must first establish that SSL is permitted. This can take up to one round-trip time to establish (it might be less than one round-trip if you piggyback the SSL ClientHello on the first protocol message).

Finally, although packet filtering firewalls do not need to be modified to support an upward negotiation strategy, the effect on application-layer firewall proxies is substantial. Because the application-layer proxy must be able to recognize the upward negotiation commands and get out of the way, this often requires not simply reconfiguration of the proxy but a change to the code as well. This can be a substantial deployment barrier.

7.12 Downgrade Attacks

Both methods of protocol selection are susceptible to an active downgrade attack. In this attack, the attacker convinces the client or the server that an SSL connection is not possible, thus forcing the parties to fall back and communicate via an insecure channel if they are to communicate at all. It's important to note that none of these attacks work by downgrading the SSL handshake itself. That's prevented by the digests in the Finished message. Rather, they work by making it appear as if it's impossible to negotiate an SSL connection. Thus, no SSL protocol measures can protect against them.

Separate Ports

The simplest attack to mount on a separate ports negotiation is simply to make it appear that the server isn't listening on the appropriate port at all. This is trivial to do for any attacker who can inject packets into the network. The attacker simply waits for the client to send the TCP SYN to open the connection and then forges an RST packet in response. There's no way for the client to detect that the RST came from the attacker rather than the legitimate server, so the TCP stack returns an error to the client code.

Under normal circumstances this would be a simple denial-of-service attack and of minimal concern. However, the client's (or user's) behavior can make it something far worse. The problem is illustrated by many Web pages for secure servers. Such servers want to be able to support SSL but also don't want to deny non-SSL transactions. For instance, amazon.com's Web server contains the following text:

> If you received an error message when you tried to use our secure server, sign in using our standard server. If you select the secure server, the information you enter will be encrypted.

If the user follows these directions literally and clicks on the standard server link, an attacker can trivially force any connection to an insecure mode by simulating an error

when the user tries the secure link. The situation can be made even worse if clients automatically fall back to insecure modes. In either case, the attacker has achieved much more than a denial-of-service attack. It's an active confidentiality and integrity attack.

Naturally, this isn't the only way that an attacker can make it appear that an SSL connection cannot be created. Other options would be to forge a ClientHello with no cipher suites or a server alert claiming that the negotiation was impossible.

Upward Negotiation

Downgrade attack on a protocol that uses an upward negotiation strategy is a little different from one using a separate ports strategy, but the theory is the same. The attacker forges a message from the server (or the client) indicating that it is unwilling to use SSL. This means that the attacker must be able to take over the TCP connection, but this isn't that difficult. Figure 7.2 shows the flow of protocol messages here.

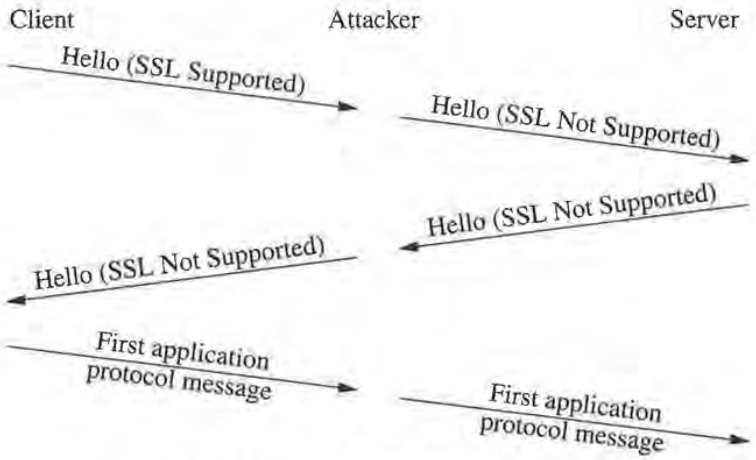


Figure 7.2 A downgrade attack on upward negotiation

As with the working upward negotiation example shown in Figure 7.1, the client attempts to form a TCP connection to the server. However, this time it forms a connection to the attacker. In Figure 7.2 we show the attacker connecting to the server posing as the client, but in many cases the attacker can just pose as the server. The client requests SSL but the attacker refuses. What happens next depends on how the client is configured.

In many cases, the client will be configured to proceed without SSL if it can't negotiate it. If all goes well, we'll *opportunistically* negotiate cryptography, but we won't worry if we can't negotiate it. In an environment where some but not all servers speak

SSL and the client has no prior knowledge about any server, this is the only sensible and interoperable way to configure it.

However, when it's configured this way, the client is completely open to this downgrade attack. As we show in Figure 7.2, it simply proceeds with an ordinary transaction. Thus, opportunistic cryptography provides excellent protection against passive attack but little protection against active attack. One partial solution to this problem (recommended in RFC 2487 [Hoffman1999a]) is to remember servers with which you have previously negotiated secure connections and insist on secure connections to them. This doesn't help when you're first encountering a new server, but it will protect against this downgrade attack against servers you've already contacted.

Even if the server requires SSL, the attacker can often mount a downgrade attack by posing as the server to the client and proxying the client's requests. This will work as long as the client doesn't demand SSL. The only case in which the server can enforce SSL successfully is if it requires SSL and client authentication. In that case the attacker will not be able to pose as the client. However, this requires all clients to have certificates as well as to support SSL. The result will be that the vast majority of clients will not be able to connect, which is often unacceptable.

Countermeasures

As we've seen, both protocol selection strategies can be vulnerable to downgrade attacks. With a separate ports strategy, all it takes is a little user inattention for security to be completely lost. The fact that legitimate servers encourage manual downgrading as a response to error makes this attack even easier.

When an upward negotiation strategy is being used, the situation is even worse. Under attack, the primary virtue of upward negotiation—automatic upgrade—becomes a weakness—automatic downgrade. Because references do not need to contain an indication that security is required in order to interoperate, it's quite common not to introduce such references at all. As a consequence, there's no way to automatically enforce security, leaving the protocol completely open to a downgrade attack.

There are two countermeasures required for this sort of attack, one technical and one social. The technical measure is straightforward; references to servers must include the fact that it is a secure site and some minimal expectations about what sorts of connection you can negotiate. The fact that SSL protects its negotiation provides some assistance here. If you can connect to the site and verify its certificate, then you can be sure that you have not been subject to a downgrade attack.

The social problem is harder. Users and programmers must be taught not to make insecure connections when secure connections fail. If you see a reference that indicates security, it's almost certainly not safe to reference it insecurely. This sort of behavior is difficult to teach users, but without it systems are ridiculously vulnerable.

7.13 Reference Integrity

The goal of reference integrity is to ensure that the server that you're connecting to is the same one that the user (or the program acting for some user) intended to connect to. This is a difficult task for several reasons.

First, the identities that servers present (distinguished names in certificates) aren't very user-friendly. More important, users aren't very motivated to pay attention to the identity of the servers they're connecting to, so the process has to be automated. Finally, many attempts to make software more user-friendly have resulted in representations of identity that are inherently difficult for software to parse. For instance, it's very easy for software to match domain names. However, software is increasingly hiding the domain names in favor of "Internet keywords," the name of the Web page, or a user's personal name. These name forms are much more difficult to map to domain names.

The overriding requirement for reference integrity is that there be a chain of trust between the identity that the application layer expects and the application layer that the certificate presented by the server. Optimally, this would consist of a single link: the identity in the certificate would match the application-layer identity. Unfortunately, this is only possible in some cases.

Failing that, the application-layer identity must be securely linkable to some other identity form. This identity form must in turn be linkable to a third identity form. This process continues until the identity can be compared to the server's certificate. Naturally, if this chain is at all long, it must be mechanically verifiable; users simply will not take the time to verify it. The major alternate forms we need to be concerned with are *IP addresses* (i.e., raw network addresses), *alternate DNS names*, and *human readable names*.

IP Addresses

As we discussed in Chapter 5, the requirement for a chain of verifiable identities is why certificates generally should not contain IP addresses. In general, DNS does not provide a secure way of mapping names to IP addresses. Secure DNS [Eastlake1999] provides some promise of fixing this problem but deployment has so far been underwhelming.

As a special case, if application-layer identities are expressed in terms of IP addresses, it's acceptable for certificates to contain IP addresses, because they can be securely compared. However, in general this is bad practice for several reasons. First, IP addresses are often obtained by dereferencing DNS names using `nslookup` or some similar tool. As we've just said, that's insecure. Second, they're subject to renumbering both when service providers change and when configurations change, so they may not be as persistent as domain names. It's difficult for users to know when such renumberings are legitimate and when they are not, so they are easily tricked into accepting spoofed connections.

Alternate DNS Names

Mapping domain names to IP addresses is not the only situation in which DNS insecurity comes into play. For a number of reasons, it's common for a single DNS name to map to a number of other DNS names. The two most common cases are CNAME and MX records. CNAMEs are usually used to provide a layer of indirection between a published name and a machine's actual name. MX records are used for e-mail routing, and we'll discuss their special problems in Chapter 10 where we discuss SMTP.

Two of the most common uses for CNAME records are simple aliasing and load balancing. Imagine that you run a very small company which is big enough to want its own Web site but not big enough to run it yourself. Instead, you farm that job out to your ISP. Nevertheless, you'd like your customers to be able to refer to your company's Web site by your company name, like so: `www.example.com`. In actuality, though, this machine is run by your ISP and it already has a name: `web1.isp.com`. A CNAME provides an alias between these two names.

> CNAME stands for *canonical name*. The idea here is that `web1.isp.com` is the canonical name of the machine and the CNAME record contains the canonical name.

Now when customers enter `www.example.com` into their browser, it automatically detects that it really needs to talk to `web1.isp.com` and so looks up its address and connects to it. If the ISP decides to move the Web site, it simply changes the target of the CNAME. Better yet, if you decide to host the Web site yourself, you just change the CNAME to point to the machine you're hosting it on, for instance `server.example.com`.

Now, imagine that your load gets so high that your single machine can't handle it any more. CNAMEs help here too because you can arrange for the name server to return a different CNAME every time, thus sharing the load between multiple servers.

Whenever we have this form of indirection, we need to carefully consider which name should appear in the certificate. From a security perspective the answer is clear but from an operational perspective it's difficult. Because DNS lookups cannot be trusted, it's not possible to create a chain of trust from the name `www.example.com` to any of these names pointed to by CNAMEs. Thus, the address that appears in the certificate *must* be `www.example.com`.

Unfortunately, using the original name (the CNAME) in the certificate is operationally problematic. In the case where the ISP is running your Web server, this would require them to obtain multiple certificates (one for each customer) or a single certificate with multiple aliases (multiple Common Names or subjectAltNames). In the case where you're running multiple servers, you'll need to share your certificate (and private key!) between all of those servers, thus increasing your security risk.

In the Web context, a common fix for this problem is to have an opening page that is accessible via HTTP only and have that point to a secure server using a URL that matches the secure server's one certificate. So, in this example the link that actually takes you to the secure server might be a link to `https://secure1.isp.com`. Of course, this sort of mismatch between URLs and the user's expectations has its own security issues which we'll take up partially in the next section and more in Chapter 9 when we discuss HTTP over SSL.

Human Readable Names

The final form of identity to consider is human readable names. Many application protocols and clients provide references to users that are in themselves representative but are nearly meaningless to the software. For instance, it's very common for e-mail clients to display user's names instead of their e-mail addresses. Similarly, users who view data in Web browsers often look at the title of the document or the content of a link rather than at the underlying URL.

Since this data is not securely bound to the server's certificate, it forms an inadequate basis for trust. Occasionally, an implementation will have access to a trustable database that maps these names onto network names. (Even Secure DNS would not do this, however.) Then and only then can these names be trusted. Otherwise, users need to be presented with the opportunity to view the identity that was verified in untranslated form. Only then can full security be provided.

7.14 Username/Password Authentication

As we stated previously, authenticating users (and clients) presents a difficult problem. From a security perspective, it would be ideal to simply deploy certificates for every client we wished to authenticate. However, this presents deployment problems in many situations, so it's important to have other options available.

Username/Password

The most obvious (and commonly used) approach is simple passwords. As we stated earlier, transmitting passwords over SSL removes the major problem with passwords—passive sniffing attacks—but it still leaves open some other more difficult (but still possible) forms of attack.

There are three major types of attack worth considering. First, note that the security of the channel depends on the client forming a secure connection to the right server. This means that the client must carefully check the server's certificate. If this check is not performed the channel is subject to a man-in-the-middle attack and the attacker can recover the user's password.

A second variety of attack is a *password guessing* attack. As we said in Chapter 5, users often choose very poor passwords (see [Klein1990]) such as their username, first name, date of birth, etc. Such passwords are easily guessed. Therefore, an attacker can simply initiate a connection to the server and try each of these easily guessed passwords in sequence. Such an attack might not succeed with any given user, but if there is a large user base, the probability that at least one user will choose a bad password is high. There are a number of somewhat effective countermeasures against password guessing, including proactive password checking, limited tries, and slowing down each password check (to make guessing more time consuming), which we will not discuss here.

A final weakness with passwords is that they're transferable. When you authenticate to a server you give it your password. If you've used the same password on multiple machines (many users do) then any one of these servers can impersonate you to another

server. As you can see, although passwords are better over SSL than in the clear, they still have substantial weaknesses.

7.15 SSL Client Authentication

The other obvious approach is to use certificate-based SSL client authentication (SSL doesn't have any other kind of client authentication). This approach doesn't have any of the weaknesses we've just described for password-based authentication. In particular, there is no way that an attacker can use the attacker's client authentication information to impersonate the client—although it is of course possible for an attacker to trick the client into making an authenticated request for the attacker.

As we've said, the primary problem with certificate-based client authentication is operational. Each client who can authenticate needs to be issued a certificate. Moreover, in most cases you still need to maintain a user list on the server side, both to support access control and to support user deletion.

Certificate Issuance

The primary problem with issuing certificates is arranging to authenticate users before you issue them certificates. For obvious reasons, if you really want to map a person's physical identity to a certificate, you can't issue certificates online but rather have to see them physically. However, in most circumstances a weaker form of authentication will suffice. For instance, users might be required to provide a credit card number.

Once you've authenticated users, you need to arrange that only that user can get a certificate with that user's name. One straightforward approach to certificate issuance is to issue temporary passwords through some out-of-band mechanism. For instance, in a corporate setting, you might simply call them on the phone to deliver the password, thus providing authentication of a sort and password security in one. The password is then provided along with the certificate request (encrypted under an SSL connection, naturally). Note that if you already have a username/password-type infrastructure in place you can simply use those passwords to transition to certificates.

Access Control

In some circumstances merely knowing that a user has a certificate is enough, but in most cases administrators want to be able to provide different levels of access to different classes of users. A number of approaches have been described for embedding this information in certificates. [Blaze1999] and [Ellison1999] describe two such systems. However, in most cases it's simpler to maintain *Access Control Lists* (ACLs) that describe the actions permitted for each certificate holder. The primary advantage of this approach is that permissions can be changed simply by changing ACLs without issuing any new certificates to the clients.

This approach has the virtue that it permits you to have a mixed certificate and password infrastructure. In order to do this, all you need is a function that maps users authenticated via certificates to users in the username/password infrastructure.

Figure 7.3 shows a simple system that uses this approach. There are actually two types of lists: a user list (which also stores passwords) and ACLs. Normally, there would be one ACL for each object in the system, but we've shown only one for simplicity. In this infrastructure, Alice can authenticate by either password or certificate, Bob can authenticate by certificate only, and Charlie can authenticate by password only. ACL entries then refer to entries in the user list.

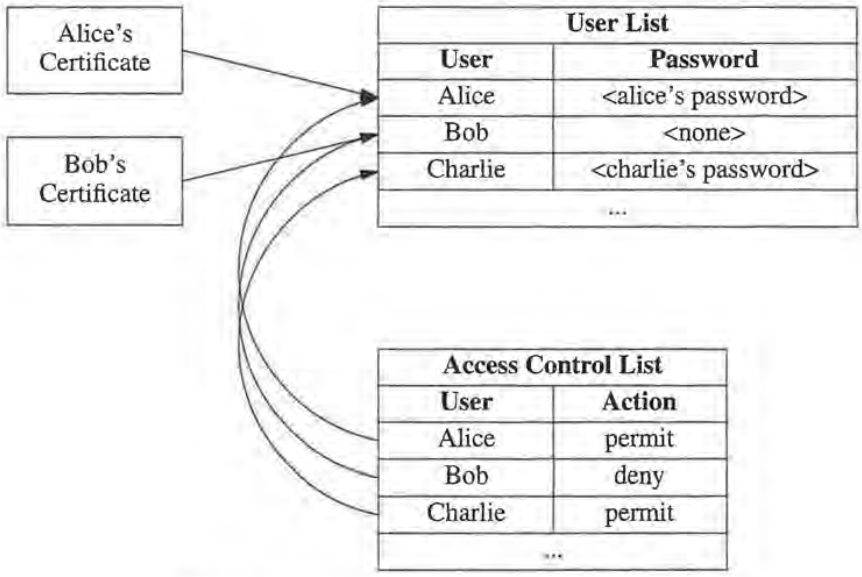


Figure 7.3 A mixed password and certificate infrastructure

Note that you can allow such users to authenticate by either method or flag users with certificates in such a way that password authentication is not allowed. The primary problem with this approach is that it requires user data to be maintained in two locations: the access control list and the CA. Thus, to create a user you have to both issue a certificate and modify the user list. Naturally, these databases can get out of synch, causing maintenance problems.

Revocation

In a dual architecture as described above, removing users is simple. You remove the entry in the access control list and do nothing else. Users can authenticate, but because their certificate doesn't map to any user, they have no permissions. The ACL entries become dangling references that can be garbage collected at some convenient later time. If you're not using ACLs (or some analagous technology), then the alternative is *Certificate Revocation Lists* (CRLs). However, as discusssed in Chapter 1, CRLs also present a significant operational problem which is exacerbated by the lack of direct support for CRLs in SSL.

Host to Host Communication

There is one special case in which certificate based authentication can be used with only minimal overhead: host to host channels. Imagine that you want to establish a secure tunnel (for a virtual private network) between two machines. In this case all you care about is the identity on the other end. In this case, the same identity checking procedures that you use for server authentication can be used for client authentication.

Moreover, if all you're interested in is authenticating the machine that is the source of a connection, you don't need to worry about access control (and therefore don't need any ACLs). Revocation is still a modest issue, but since most implementations ignore it for servers, it doesn't make much of a difference if you ignore it for clients as well.

7.16 Mutual Username/Password Authentication

So far, we've assumed that servers always authenticate via certificates. As we said in Section 7.1, this is absolutely required for secure password transmission using normal methods. However, SSL does allow operational modes in which the server remains anonymous: the SSL_DH_anon cipher suites. Under most circumstances, these cipher suites are completely vulnerable to an active attack. However, it's possible to combine passwords and anonymous cipher suites to provide a somewhat secure protocol.

Man-in-the-Middle

As discussed in Chapter 1, the primary weakness of DH is to active man-in-the-middle attack. In normal SSL operational modes, this attack is prevented by having the server sign its DH key with its static signing key. This prevents an attacker from mounting such an attack. However, if we're using anonymous mode, the server's DH key is no longer signed and so the attack again becomes possible. The only means of authentication we have left is the password shared by client and server. Somehow we have to leverage that to build a trusted connection.

One approach would be to simply use the password directly to sign the ServerKeyExchange message. Obviously, this would be a MAC rather than a digital signature, but it would have the same effect. The primary drawback to this approach is that

it would require a change to SSL, because a MAC is not one of the defined methods for signing the ServerKeyExchange. Historically, there has been resistance to adding such password mechanisms directly to SSL. Thus, proposals have been made to solve this problem at the application layer.

An Approach That Won't Work

Consider the classic DH man-in-the-middle attack, shown in Figure 7.4. The attacker intercepts the DH public keys from client and server and substitutes its own public key. It is thus able to agree on a shared key (*ZZ*) with both client and server. Note that although the attacker is able to compute pairwise keys with both client and server, the *ZZ*s aren't the same because the client and server private keys are not the same. Because SSL uses *ZZ* as the pre_master_secret, the client and server will compute different pre_master_secrets. It seems that one might be able to use this fact to protect against the man-in-the-middle attack.

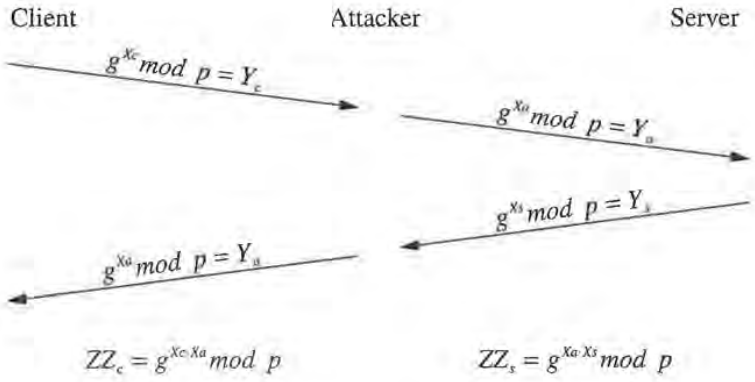


Figure 7.4 Classic DH man-in-the-middle attack

One approach that has been suggested is to use the password to compute a MAC over the master_secret (this is equivalent to computing one over the PreMasterSecret). The two sides would then exchange MACs and if they didn't match they would know they were under attack. The attacker does not know the password and therefore can't compute new MACs using it, even though he knows both ZZs.

Although this approach prevents the classic man-in-the-middle attack, there are variants for which it does not work. It turns out to be possible (by appropriate choice of forged keys) to force *ZZ* (as computed by both client and server) to be the same *and* to be one of a very small set of values. The signatures over *ZZ* will thus be the same and the attacker can just forward them from client to server and vice versa.

Ex.1024
Askeladden, L.L.C. / Page 309 of 788

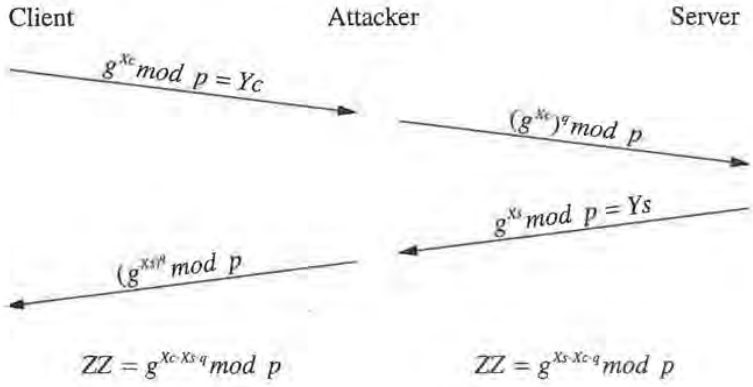


Figure 7.5 A more sophisticated man-in-the-middle attack

Figure 7.5 shows one sort of this attack. Consider the case where $p = 2q + 1$ where q is also prime and R is a relatively small number. It turns out that under these circumstances $g^q = -1$. Now, if the attacker replaces each of the public keys in transit (X) with X^q we force the resulting shared key (ZZ) to be $g^{2Xs \cdot Xy \cdot q}$. Note that ZZ is the same on either side. This is useful because

$$g^{Xs \cdot Xy \cdot q} = (g^q)^{Xs \cdot Xy} = (-1)^{Xs \cdot Xy} = 1 \text{ or } -1$$

Thus the attacker has managed to force ZZ to be either 1 or −1 and can simply try them both. This attack can be extended to DH groups where $p \neq 2q + 1$.

An Approach That Will Work

We've seen that MACing the PreMasterSecret won't work. Although MACing the ServerKeyExchange would work, it requires modifying SSL, which is problematic. However, we can indirectly MAC the DH public keys by MACing the Finished message. Because the Finished message includes digests of the public keys this transitively MACs the public keys themselves. Because the public keys seen by the client and the server are not the same, this will detect this attack. This is a variant of the procedure used in the protocol STS, described in [Diffie1992]. The specific variant described here was first suggested by Bodo Moeller on the OpenSSL mailing list.

Figure 7.6 shows this process. After the SSL connection has been formed, the applications send MACs of the Finished message as their first application data. Note that it's important that each side MAC its own finished message. This method allows the construction of a secure channel without certificates, which is bootstrapped via a shared password. Moreover, both client and server know that the other side knows the

password, providing for mutual authentication. This is in a sense superior to username/passwords with certificates because the server is forced to prove that it knows the password. Note that SSL does not have a mechanism for negotiating this technique. Client and server have to agree on it via some mechanism external to SSL.

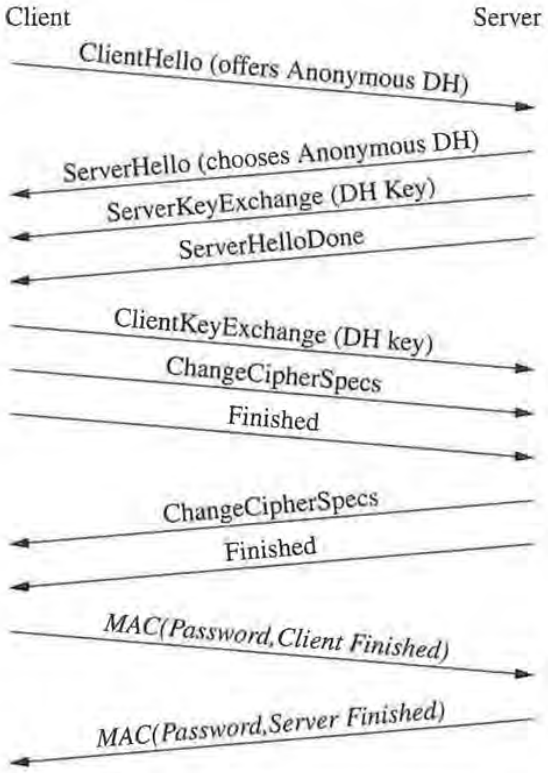


Figure 7.6 Password-based mutual authentication

An Active Dictionary Attack

Although this method is good, it does not provide total security for the password. It is possible for the attacker to mount an active attack against the connection and potentially recover the password. The attacker mounts an ordinary man-in-the-middle attack and thus shares a secret with both client and server. He then collects the MACed Finished message from either the client or server and uses this to mount his attack.

The attack is a simple variant of the dictionary search attack discussed in Chapter 5. The attacker knows the contents of the finished message but doesn't know the password used to compute the MAC. He thus tries out various potential passwords until he finds one that produces the right MAC.

Note that this attack is far superior to the password guessing attack available for regular username/password authentication because it can be carried out mostly offline. The only portion of the attack that's active is the collection of the Finished message. By contrast, the normal password guessing attack requires the server seeing every attempt.

7.17 Rehandshake

As discussed in Chapter 4, it is possible to initiate a new SSL handshake once the SSL handshake has already been completed. This is primarily useful if your server is configured such that different types of protocol requests require different types of security. Because the SSL connection is negotiated before the first byte of application data is written, there is no way of knowing which security properties to apply to the connection during the first handshake. Therefore it is sometimes a useful strategy to initiate a second handshake during the connection in light of this new information.

Client Authentication

One common reason to renegotiate is to require client authentication. Many environments allow access to some resources by any client but require authentication to access others. Requiring SSL client authentication for all connections would exclude some clients and is therefore undesirable; however, it's a requirement for other connections.

One approach would be to use some variant of the separate ports approach and have a different port for connections that required client authentication. However, a more elegant approach is to allow clients to connect and request client authentication if a client requests access to a protected resource. This tactic is used by a number of Web servers.

The primary issue with renegotiation is arranging for it to happen at the right time. The SSL specification provides no real guidance as to when renegotiation is to occur, saying only that:

> This message may be ignored by the client if it does not wish to renegotiate a session, or the client may, if it wishes, respond with a no_renegotiation alert. Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation will begin before no more than a few records are received from the client.

In the client authentication case the situation is fairly simple. The server merely wants to ensure that the client is authenticated before it provides it any restricted services. Thus,

the server should send a HelloRequest and then wait for the client to start the handshake. Any requests that the server receives between the time the HelloRequest is sent and the handshake completes should be queued and processed once the client has authenticated. Naturally, if authentication fails, the connection is closed or at least the requests are rejected.

Cipher Suite Upgrading

Another use of renegotiation is to upgrade to a stronger cipher suite. As with client authentication, the server might discover that the client is asking for especially sensitive data and wish to use stronger ciphers. The timing for cipher suite upgrading is more problematic. Imagine that some traffic to and from a server is more sensitive than other traffic. The server might choose to ordinarily negotiate DES or RC4 but upgrade to 3DES when this sensitive traffic is being passed. Renegotiation can be used for this purpose as well. It could also be used to negotiate a new connection with a new ephemeral DH key to ensure Perfect Forward Secrecy.

Timing can be much trickier in this case. If all you're concerned with is confidentiality for data the server transmits, then the server can simply transmit its HelloRequest and wait as it did for client authentication. However, if data transmitted by the client must be confidential, the situation becomes more problematic because there is no way to force the client to renegotiate immediately. About the only thing the server can do is terminate the connection if the client starts sending sensitive data before renegotiating. Because the client will transmit up to the size of its TCP window, even if the server sends an error as soon as it receives the first sensitive byte, quite a lot of sensitive data will likely be transmitted under the less-secure cipher suite.

Replenishment of Keying Material

One concern that implementors often have with SSL is whether they should renegotiate to replenish their keying material, typically to prevent large-scale attacks on long-lived connections. If really large amounts of data are being transferred, new keys must be occasionally generated to prevent CBC rollover. From an application perspective, timing isn't that critical because environments where this is a concern typically are moving megabytes of data. A kilobyte more or less sent before renegotiation isn't that big a deal.

Client Behavior

The discussion in the preceding sections suggests several cases in which servers will transmit HelloRequest messages and then stop waiting for a new connection. As a consequence, clients that receive HelloRequest messages should attempt to initiate a new handshake fairly immediately. Certainly, they should do so immediately if they have sent all their traffic and the server remains idle. Failing to do so risks deadlock.

7.18 Secondary Channels

Some protocols are designed to run over more than one TCP connection. FTP [Postel1985] provides a good example. The client initiates a connection to the server and logs in. This connection is called the *control connection*. Various commands are issued over this connection. Each time a file is transferred between the client and server, it's transmitted over a second connection called the *data connection*.

Typically, the data connection is created by the server. The client chooses an ephemeral port number and tells the server over the control connection. The server then opens a connection to that port and uses a secondary connection from the server to the client in order to transmit the data.

This case is confusing because the server is doing an active open: that is, it's acting the way traditional TCP clients do. Which side is the TLS client? FTP over SSL has never been standardized, but several drafts have been published and they take a stand on this issue (see [Ford-Hutchinson2000). The client *always* acts as the TLS client, regardless of who does the active open.

This choice makes the situation somewhat simpler because some implementations will do session resumption only in the same mode in which the session was created (i.e., if they were the TLS client for the first session, they must be the client for resumed sessions). More important, however, is to realize that the choice has to be made if you are to have interoperability.

7.19 Closure

Many protocols use the TCP connection close to indicate that no more data is on its way. From a security perspective, this is problematic because TCP FINs can be forged quite easily. To fix this problem, SSL provides its own closure mechanism: the close_notify alert. However, this is not the only secure approach. If the application-layer protocol has its own indication that the connection is about to close—and it's carried over SSL—then the close_notify is superfluous from a security perspective.

All protocols should do a full SSL closure handshake if for no other reason than that the specification requires it. Moreover, if you don't do it, you're technically forbidden to resume the connection—although many implementations violate this rule. That said, it's important to specify in your protocol design what the consequences of failure to complete the closure handshake are. In particular, some cases represent security risks and some do not.

Incomplete Close

Many protocols contain states in which one side knows that there will be no more data coming from the other side. Consider the case of data being transferred via FTP. The data channel is one-way only, so when the transmitting end is finished that means that there will be no more data transmitted at all.

In such cases, many implementations send a close_notify and then immediately

Ex.1024
Askeladden, L.L.C. / Page 314 of 788

close the connection. This generates a condition which RFC 2818 [Rescorla2000] calls an *incomplete close*. When the other side of the connection receives the close_notify, it responds with its own close_notify alert as required by section 7.2.1 of the TLS specification. However, because the connection has already been closed by the side that sent the first close_notify, the response is a TCP RST segment.

We've been using the term "closed" very loosely here. TCP supports a *half-close* in which one side is finished sending data but is still ready to receive. In the sockets API, this is implemented using the `shutdown()` call with the `how` value set to 1. Most other APIs provide similar functionality.

Most applications, however, simply use `close()`, which instructs the system not to accept any more data from the other side as well. This is the cause of the RST segment: the server received traffic after it had already closed the connection.

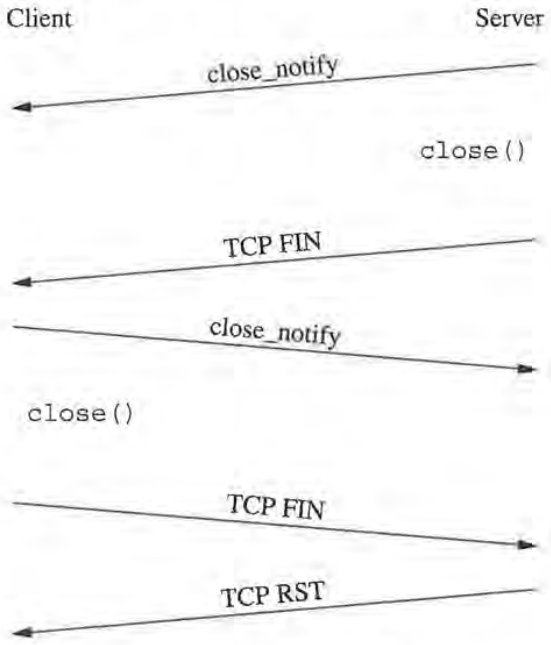


Figure 7.7 An incomplete close

Figure 7.7 shows a typical case. The server knows that the connection is finished and generates a close_notify. The server then calls `close()`, closing the connection completely (generating a FIN in the process). The client has no way of knowing that the server is not ready to receive more data (this might just be half-close) and sends its close_notify in response, followed by its own FIN. Because the socket is already closed

Ex.1024
Askeladden, L.L.C. / Page 315 of 788

on the server side, the server responds with an RST segment. Another way to get an RST is if the client has already written more data when it gets the close_notify. Note that RFC 2246 requires an implementation to stop transmitting data once it receives a close_notify, but the data might already be in the network buffers or on the wire.

An incomplete close does not necessarily present a security problem. In order to see this, consider the situation at both sides of the connection. The server knows that there will be no more protocol data (or at least that it isn't interested in any more) or it wouldn't have sent the close_notify in the first place. If there were somehow more data, this would represent a problem with the application protocol rather than the SSL connection. Whatever application data led the implementation to believe that there was no more data should have been protected just as securely as the close_notify would have been.

Now consider the view from the client's side of the connection The client knows for a fact that there will be no more data from the server because it's received an appropriate close_notify. It has generated an appropriate close_notify and had it rejected.

Premature Close

The second variety of closure failure to consider is the *premature close*. In premature close, one side closes the TCP connection (i.e., sends a FIN) without first sending a close_notify. Unlike an incomplete close, a premature close may very well represent a security threat. Recall that the close_notify exists to prevent a truncation attack. Without seeing the close_notify, the receiver cannot distinguish between a FIN segment generated by the sender and a FIN segment forged by an attacker, so a truncation attack may well be in progress.

It's worth distinguishing between two cases. In the first, the application protocol has its own end-of-data markers. If these markers are received before the FIN, then the receiver should assume that there has been an implementation error on the sender's side; this error presumably caused either the end-of-data markers to be sent too early or the close_notify to be omitted. As long as the end-of-data markers were sent over the SSL channel, they cannot have been forged.

By contrast, if the protocol does not contain end-of-data markers—or if it contains them but they were not received prior to the FIN—then it is very possible that a truncation attack is under way. It is not possible to distinguish this case from implementation errors on the sender's side. In this case, the application must treat this as an error. Under no circumstances should a session that ends in a premature close be resumed; the SSL specification clearly forbids it.

Note that failing to resume an improperly closed connection doesn't really add any security value. A premature close may indicate that the connection is under some attack, but it doesn't indicate that any of the keying material has been compromised.

Figure 7.8 summarizes the correct behavior for closure problems.

Case	Generate Error	Resume Session
Incomplete Close	No	Yes
Premature Close after End of Data	No	No
Premature Close w/o End of Data	Yes	No

Figure 7.8 Correct behavior for closure problems

7.20 Summary

This chapter has discussed the design of application protocols that use SSL. Although it's possible to achieve some security with a naive approach, getting maximal security requires paying attention to a number of important and tricky details.

Not all security services need to be provided via SSL. Although SSL is useful for securing a wide variety of protocols, there are some services it cannot perform. In particular, it can't provide security for static data. Sometimes it is useful to combine SSL with other security measures.

Both protocol selection mechanisms have value. Separate ports is easier to implement, but upward negotiation is somewhat more tasteful and has dramatically better security against passive attack.

Authenticate servers with certificates. We've discussed one special case in which servers can be authenticated using a password, but in general it's best to stick with server certificates.

Set client security expectations. In order to prevent active attacks, clients need to know what kind of security to expect from the server. This means references must include both the expected server identity and some indication that the server requires security.

Certificates are harder than passwords, but also stronger. Client authentication based on certificates presents significant deployment problems but is also far more secure than passwords, even when they are transmitted over SSL.

Define rehandshake semantics. Rehandshake can be a valuable tool, but the lack of clear specification for the timing of rehandshake can result in race conditions. Designers need to clearly specify the application protocol states in which rehandshake is allowed.

Closure is tricky. Some protocols have their own end-of-data markers, making the SSL close_notify unnecessary. Other protocols do not, making the close_notify vital to prevent a truncation attack. Designers need to carefully examine the closure behavior of their particular protocol.

8

Coding with SSL

8.1 Introduction

In previous chapters we've concentrated on SSL from a protocol perspective. For protocol designers and systems architects, this is the appropriate level of analysis. However, even the best protocol design is useless unless it can be implemented. Therefore, this chapter concentrates on SSL from the implementation perspective.

Chapter 7 examined a series of design problems commonly encountered when working with SSL. Most of these problems can be approached using one of several well-known design patterns. Similarly, implementors working with SSL need to perform many of the same implementation tasks regardless of which application protocol they're working with. This chapter is devoted to describing these tasks and to programming idioms that have proved successful in executing them.

8.2 SSL Implementations

Because SSL implementations are readily available for most common languages, most programmers will be using a toolkit rather than implementing SSL themselves. We assume that you have such a toolkit available. Although each SSL toolkit has its own *Application Programming Interface* (API), the interfaces designed for each language typically share a common style, making code written for one toolkit instructive even if you are using a different one.

For our purposes, we've chosen two freely available SSL toolkits, one in C and one in Java. For C, we've chosen OpenSSL, derived from Eric Young's SSLeay toolkit. For Java, we've chosen PureTLS, written by the author of this book. Both toolkits have APIs that are fairly typical for their respective languages. Moreover, both are readily available for download on the Internet. See Chapter 2 for details.

Ex.1024
Askeladden, L.L.C. / Page 319 of 788

8.3 Sample Programs

The best way to learn programming techniques is to write programs. To that end, we'll write two programs—a client and server pair—in both C and Java for a total of four programs. Our intent is to demonstrate programming techniques rather than to provide a detailed guide to programming with PureTLS or OpenSSL, so in cases where the OpenSSL-based and PureTLS-based code is substantially identical we'll choose one version. However, where the idioms are dramatically different between C and Java, we'll show examples for both languages.

Platform Issues

The C programs were developed with OpenSSL 0.9.4 on FreeBSD. They also work with OpenSSL 0.9.5a. Only the most basic OpenSSL functions are used so they should continue to work with most past and future versions. They should compile and run without modification on FreeBSD and other UNIX and UNIX-like operating systems. Some modification will probably be required in order to compile them on Windows.

The Java programs were developed with PureTLS 0.9b1 on FreeBSD with JDK 1.1.8. They have been tested with JDK 1.2.2 on Windows NT. They should compile and run with future versions of PureTLS.

We've deliberately chosen to emphasize UNIX over Windows for a number of reasons. First, SSLeay's predecessor, OpenSSL was designed for UNIX and thus OpenSSL's interface more closely matches that of UNIX than that of Windows. Moreover, networking APIs even on non-UNIX systems are typically based on sockets, so writing sockets-based networking code is widely applicable. Windows's networking interface, Winsock, is based on sockets but is not identical to it.

The second reason is that Microsoft provides its own API for SSL: SChannel, described in [Microsoft2000]. SChannel's interface is rather different from those provided by other SSL implementations so showing example programs using it would be of limited use to non-Windows programmers. By contrast, code written using OpenSSL is of broad applicability. The UNIX code we show is sufficiently generic that it should be portable to Windows without too much trouble.

The final reason is simple taste. In the opinion of the author, the UNIX programming API is cleaner and allows a simpler exposition of the SSL issues with a minimum of operating specific issues.

Client Program

Our client is the simplest kind of interactive client. It initiates an SSL connection to the server on the other end and then simply forwards data between the user and the server. In other words, it reads data from the keyboard and sends that data to the server over the

SSL connection. Similarly, it reads data from the server over the SSL connection and writes that data to the screen.

Despite its simplicity, this sort of program can serve a useful debugging purpose. Many Internet protocols (SMTP, HTTP, IMAP, ...) consist of simple ASCII commands and responses. Thus, a simple client can be used to contact the server directly for debugging purposes. The UNIX `telnet` program—which opens a simple TCP connection—is often used for this sort of server debugging.

This client does have one idiosyncracy: data is sent to the server in *line-by-line* mode on many systems. The UNIX terminal driver (and some others) by default buffers keyboard data until the user presses the return key. This behavior can be changed but it doesn't serve any useful demonstration purpose to do so.

Server Program

The server program is a simple *echo server*. It waits for TCP connections from clients. When it accepts one it negotiates an SSL connection. Once the connection is negotiated, it reads data from the client and then retransmits it right back. Finally, when the client closes the connection the server closes its side of the connection.

Obviously, this isn't that useful a service—although standard UNIX systems do come with an echo server that does this job in an insecure fashion—but this program allows us to demonstrate the essential features of a network server: waiting for connections, servicing them, and cleaning up afterwards.

Programs in Action

Figure 8.1 shows the interaction of our simple client and server programs. The user types data on the keyboard and it's transmitted to the server. The server simply echos the data back to the client. Finally, the client program prints the data to the screen.

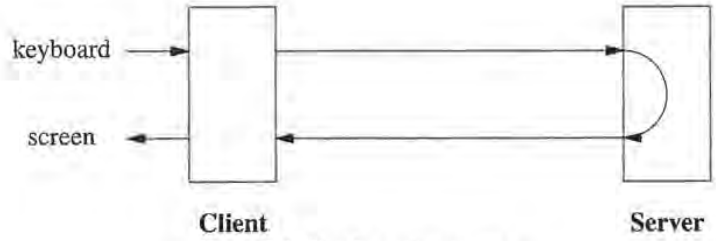


Figure 8.1 Client/server interaction

Figure 8.2 shows a demonstration run of our Java client (although the output of the C client would be nearly identical.)

```
[63] java SClient          Started the program
line 1                     Typed at keyboard
line 1                     Printed by client program
line 2                     Typed at keyboard
line 2                     Printed by client program
[64]                       Pressed Control-D to end program.
```

Figure 8.2 Sample run of `SClient`

Program Presentation

We've chosen to present only source code fragments in this chapter to illustrate the techniques we're discussing. Appendix A contains a complete source code listing for all the example programs as well as a road map to the various source files. You can tell which source file a given fragment comes from by examining the boundary lines at the top and bottom of the listing.

8.4 Context Initialization

The first task to perform when writing a program that uses SSL is to set up a context that will be used by the system. Most implementations do this by having a context object which the programmer needs to initialize. This context object is then used to create a new connection object for each new SSL connection. It is these connection objects which are used to do SSL handshakes, reads, and writes.

This approach has two advantages. First, the context object allows many structures to be initialized only once, saving time. In most applications, every SSL connection will use the same keying material, root list, etc. Rather than reloading this material for every connection, we simply load it into the context object at program startup. When we wish to create a new connection, we can simply point that connection to the context object.

The second advantage of having a single context object is that it allows multiple SSL connections to share data. The most obvious case where this is relevant is for SSL session resumption. Session data can be stored in the context object so that you can automatically resume a session merely by creating a new connection with the same context object.

Ex.1024
Askeladden, L.L.C. / Page 322 of 788

Client Initialization

In general, a client will have to perform the following initialization tasks.

Load CAs. In nearly every SSL connection the server provides its certificate to identify itself. Obviously, in order to verify this certificate, the client needs to have a list of the CAs that it trusts to sign server certificates. Typically this list is stored in a file on disk somewhere and the client needs to load it.

Load client keying material. If the client has keys that it is using for client authentication, it will need to load them and their associated certificates as well. As we discussed in Chapter 5, these will usually be either in a file on disk or on some hardware device. In either case, the program will likely need to provide a password to access the keys.

Seed random number generator. Because every application environment is different, it is often difficult for toolkit programmers to generate good random seed data. For instance, screen or mouse data is often a good source of randomness, but some environments have neither a screen nor a mouse. Thus, many toolkits leave random number seeding up to the applications programmer. This usually means finding some source of randomness on your system and feeding it to the toolkit.

Set allowable cipher suites. Most toolkits have some set of cipher suites that they are prepared to negotiate by default. This may not suit your application and you should be prepared to change them. Many toolkits allow you to do this either at context setup time or on a per-connection basis.

The exact number of operations required to perform these actions depends on the particular toolkit you're using. For instance, SPYRUS's TLSGold loads all of its keying material from a single file in one operation whereas OpenSSL requires a number of different loading operations.

Server Initialization

In general, server initialization is a superset of client initialization. Although it's technically possible for a server that doesn't require client authentication not to have a list of CAs it trusts, it's good practice to check one's own certificates when initializing. This protects against data corruption and user error. Thus, loading a CA list is a good idea for servers as well. In addition, some server-specific initialization may be required.

Set DH group. If you're planning on using ephemeral DH mode, then the server needs to be loaded with the DH group to use. Since DH group generation is so slow, it's desirable to do this at startup time. Frequently, you'll want to store a long-term DH group on disk and then load it up at this time rather than generate a new group every time you start up.

Set ephemeral RSA keys. Similarly, if you plan to use ephemeral RSA (pretty much a requirement if you're using long RSA keys and supporting export cipher suites) you may want to load an ephemeral RSA key at this time. A lot of toolkits will generate one automatically as needed, but this approach avoids incurring that performance cost later.

Java Initialization

Demo.java

```
1   import COM.claymoresystems.sslg.*;
2   import COM.claymoresystems.ptls.*;

3   public class Demo {
4       public static final String host=  "localhost";
5       public static final int port=  4433;
6       public static final String root=  "root.pem";
7       public static final String random= "random.pem";

8       static SSLContext createSSLContext(String keyfile,String password){
9         SSLContext ctx=new SSLContext();

10        try {
11          ctx.loadRootCertificates(root);
12          ctx.loadEAYKeyFile(keyfile,password);
13          ctx.useRandomnessFile(random,password);
14        } catch (Exception e){
15          throw new InternalError(e.toString());
16        }

17        return ctx;
18      }
19  }
```

Demo.java

Figure 8.3 Demo source code

Demo.java, shown in Figure 8.3, contains all the generic initialization code we'll be using for PureTLS. It also contains a number of global definitions for things like the port number we're using. Our client and server programs are all subclasses of Demo.

Import other packages

1–2 The public interface of PureTLS lives in two Java packages. COM.claymoresystems.sslg contains interfaces and COM.claymoresystems.ptls contains the implementations.

Set constants

4–7 In a real program, variables like the locations of files, host names, and port numbers would be in a configuration file or input by the user. For our purposes we just hardwire them into the code.

Initialize context

9–18 PureTLS uses the `SSLContext` object to store its context. Once the context object is created, we need to load the root certificates and the key file we'll be using. Note that PureTLS requires a password to load its random file because the file is encrypted on disk. This prevents an attacker who can read the random file from reproducing the random stream.

C Initialization

`initialize_ctx()`, shown in Figure 8.4, serves the same purpose for OpenSSL as the `Demo` class does for PureTLS. This function is considerably longer than `Demo`, partly due to error checking each call: Java's exception mechanism does this for us with PureTLS. However, OpenSSL's setup is also somewhat more complicated, which complicates our example code.

Initialize global data

44–52 Before we can even create an SSL context we need to initialize the OpenSSL library itself using `SSL_library_init()`. The rest of this code allows us to display somewhat mnemonic error strings instead of error numbers by initializing the error list.

Initialize context

53–69 OpenSSL calls its SSL context variable an `SSL_CTX` (which presumably stands for SSL context). The initialization code is fairly straightforward, but a few fine points are worth comment. First, note the use of the `meth` parameter to `SSL_CTX_new()`. The purpose of this parameter is twofold. First, it allows us to set the SSL version that this context is prepared to negotiate.

More important, this technique allows us to link in only those sections of code that we need. Each `method` references the various functions and object files needed to implement it. In most systems, the linker links in only those library functions that are transitively referenced by `main()`. (That is to say, those functions that are referenced by `main()` or referenced by a function referenced by `main()`, and so on.) Thus, if you only specify `SSLv3_method()`, the SSLv2 and TLS functions are never even linked into the final object, reducing binary size. To write a client or server that would work with any SSL version you might use `SSLv23_method()`.

Also, notice the use of a password callback in lines 59–60. In PureTLS, passwords are provided directly to the library via API calls. In OpenSSL 0.9.4, OpenSSL, the library requests passwords from the user via a callback. In this case, we've replaced the default callback (which prompts the user via the terminal) with one that simply returns our hardwired password. In later versions of OpenSSL, it is possible for the programmer to pass in the password directly via an API call.

Common C Code

Our demo programs also use a number of other common subroutines and variables found in `common.c`. The function prototypes for these functions are found in `common.h`, which is included in all our other source files. `common.h` also includes a

common.c

```
38  SSL_CTX *initialize_ctx(keyfile,password)
39    char *keyfile;
40    char *password;
41    {
42      SSL_METHOD *meth;
43      SSL_CTX *ctx;
44      if(!bio_err){
45        /* Global system initialization*/
46        SSL_library_init();
47        SSL_load_error_strings();
48        /* An error write context */
49        bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
50      }
51      /* Set up a SIGPIPE handler */
52      signal(SIGPIPE,sigpipe_handle);
53      /* Create our context*/
54      meth=SSLv3_method();
55      ctx=SSL_CTX_new(meth);
56      /* Load our keys and certificates*/
57      if(!(SSL_CTX_use_certificate_file(ctx,keyfile,SSL_FILETYPE_PEM)))
58        berr_exit("Couldn't read certificate file");
59      pass=password;
60      SSL_CTX_set_default_passwd_cb(ctx,password_cb);
61      if(!(SSL_CTX_use_PrivateKey_file(ctx,keyfile,SSL_FILETYPE_PEM)))
62        berr_exit("Couldn't read key file");
63      /* Load the CAs we trust*/
64      if(!(SSL_CTX_load_verify_locations(ctx,CA_LIST,0)))
65        berr_exit("Couldn't read CA list");
66      SSL_CTX_set_verify_depth(ctx,1);
67      /* Load randomness */
68      if(!(RAND_load_file(RANDOM,1024*1024)))
69        berr_exit("Couldn't load randomness");
70      return ctx;
71    }
```

common.c

Figure 8.4 `initialize_ctx()`: initialization for OpenSSL

number of include files which we'll need in the other source files. Last, it includes a number of hardwired constants for things like the port number and the root file. We've reproduced `common.h` in Figure 8.5 for your viewing pleasure.

common.h

```
1   #ifndef _common_h
2   #define _common_h

3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <errno.h>
6   #include <sys/types.h>
7   #include <sys/socket.h>
8   #include <netinet/in.h>
9   #include <netinet/tcp.h>
10  #include <netdb.h>
11  #include <fcntl.h>
12  #include <signal.h>

13  #include <openssl/ssl.h>

14  #define CA_LIST "root.pem"
15  #define HOST "localhost"
16  #define PORT 4433
17  #define BUFSIZZ 1024

18  extern BIO *bio_err;
19  int berr_exit (char *string);
20  int err_exit(char *string);

21  SSL_CTX *initialize_ctx(char *keyfile, char *password);
22  void destroy_ctx(SSL_CTX *ctx);

23  #endif
```

common.h

Figure 8.5 `common.h`

Server Initialization

Servers may require additional initialization steps that clients do not. As we discussed earlier, servers that need to do ephemeral keying will probably want to load either the DH group or the RSA key that they want to use. Also, in OpenSSL you need to explicitly activate session resumption for servers, which is done in lines 23–24. Figure 8.6 shows the additional server initialization steps required for OpenSSL.

sserver.c

```
19          /* Build our SSL context*/
20          ctx=initialize_ctx(KEYFILE,PASSWORD);
21          load_dh_params(ctx,DHFILE);
22          generate_eph_rsa_key(ctx);

23          SSL_CTX_set_session_id_context(ctx,(void*)&s_server_session_id_context,
24            sizeof s_server_session_id_context);
```

sserver.c

Figure 8.6 OpenSSL server initialization

8.5 Client Connect

Once the client has initialized the SSL context, it's ready to connect to the server. SSL toolkit APIs are typically closely modeled after normal networking APIs. Thus, C APIs closely resemble the Berkeley Sockets API with a series of SSL calls analagous to sockets calls. Java APIs typically subclass `java.net.Socket`—although Sun's JSSE [JavaSoft1999] takes a rather different approach which we won't show here.

Java Client Connect

PureTLS implements a class `SSLSocket` which subclasses `java.net.Socket`. As a consequence, it's trivial to make a simple SSL connection using code nearly identical to that you would use to make an ordinary socket connection. The primary additional wrinkle is that we need to check the distinguished name in the server's certificate against the hostname we want to connect to, as shown in Figure 8.7.

Extract Common Name

13–26 In order to check the server's certificate, we'll need to extract the Common Name from the DN. The `dnToCommonName()` method takes care of this. Recall that distinguished names are a sequence of *relative distinguished names* (RDNs). RDNs themselves are a list of *attribute value assertions* (AVAs). AVAs are just attribute-value pairs. In PureTLS, DNs are represented as `Vectors`. Their content RDNs are also `Vectors`, and the AVAs themselves are represented as arrays of `Strings` with the 0th element being the attribute and the first element being the value. Thus, this code takes the CN AVA out of the first (most local) RDN in the DN.

Connect to server

30 The connect itself is trivial. In the Java world, the connect is performed when you create a new `Socket`. `Socket` provides a constructor that automatically performs name resolution so we don't even need to look up the server's IP address.

Client.java

```
13          private static String dnToCommonName(DistinguishedName dN)
14            throws IOException {
15            Vector dn=(Vector)dN.getName();
16            Vector rdn=(Vector)dn.lastElement();

17            if(rdn.size()!=1)
18              throw new IOException
19                ("DN forms with multiple AVAs per RDN are unacceptable");

20            String[] ava=(String [])rdn.firstElement();

21            if(ava.length!=2)
22              throw new IOException("Bogus AVA array");

23            if(!ava[0].equals("CN"))
24              throw new IOException("CN must be most local AVA");

25            return ava[1];
26          }

27          public static SSLSocket connect(SSLContext ctx,String host,int port)
28            throws IOException {
29            // Connect to the remote host
30            SSLSocket s=new SSLSocket(ctx,host,port);

31            // Check the certificate chain
32            Vector certChain=s.getCertificateChain();

33            // Length check
34            if(certChain.size()>2)
35              throw new IOException("Certificate chain too long");

36            // Hostname check (using Common Name)
37            Certificate cert=(Certificate)certChain.lastElement();
38            String commonName=dnToCommonName(cert.getSubjectName());
39            if(!commonName.equals(host))
40              throw new IOException("Host name does not match commonName");

41            return s;
42          }
```

Client.java

Figure 8.7 PureTLS client connection

Check server certificate

31–42 Most of this function is taken up by checking the server's domain name against the server's certificate. Note that since we used the domain name to create the socket, PureTLS could theoretically perform this check for us. However, this would reduce programmer flexibility. The check shown here is about the most trivial possible. Note that we check the certificate chain length rather than relying on the Basic Constraints extension. Ideally, if Basic Constraints were universally available, PureTLS would check them for us. (It doesn't.)

C Client Connect

The OpenSSL API is considerably more complicated to use than the PureTLS API. In part this reflects the fact that Berkeley Sockets is substantially more complicated than Java networking. Additionally, we'll see that the OpenSSL API is somewhat more flexible than the PureTLS API.

Unlike PureTLS, OpenSSL requires us to create a TCP connection between client and server on our own and then use the TCP socket to create an SSL socket. We use the function `tcp_connect()` to do this job. This function (shown in Figure 8.8) should be familiar to anyone who's ever done any TCP programming: we use `gethostbyname()` to resolve the IP address of the server and then use `socket()` and `connect()`.

client.c

```
3   int tcp_connect()
4   {
5     struct hostent *hp;
6     struct sockaddr_in addr;
7     int sock;

8     if(!(hp=gethostbyname(HOST)))
9       berr_exit("Couldn't resolve host");
10    memset(&addr,0,sizeof(addr));
11    addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
12    addr.sin_family=AF_INET;
13    addr.sin_port=htons(PORT);

14    if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
15      err_exit("Couldn't create socket");
16    if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
17      err_exit("Couldn't connect socket");
18    return sock;
19  }
```

client.c

Figure 8.8 `tcp_connect()` function

Separating the TCP connection from the SSL connection has both advantages and disadvantages. The primary disadvantage is obvious: it's more difficult to program. However, functions like `tcp_connect()` are fairly common C idiom and any competent network programmer has likely written them many times. Moreover, if you're working with a preexisting network application it will already have such code and integrating this style of SSL API will be correspondingly easier.

However, separating the TCP connect and the SSL handshake means that the SSL API doesn't know the DNS name or port that the server is expected to have. This means that such a toolkit cannot (even in principle) automatically check the server's certificate against the DNS name. The application code must do the check. Also, as we'll see later, without the server's domain name, the client cannot know which session to attempt to resume, requiring the programmer to take care of client-side session resumption.

Requiring the programmer to do the TCP connect does have a number of flexibility advantages. Because an upward negotiation strategy requires that the application protocol have access to the socket in order to negotiate the transition to SSL, this sort of API is required if you want to do upward negotiation. Moreover, as we'll see in Chapter 9, some forms of proxying require that the application protocol have access to the socket before the SSL handshake.

C Client SSL Handshake

The `main()` function of `sclient.c`, shown in Figure 8.9 (see next page), is responsible for connecting to the server and performing the SSL handshake.

***Create* `SSL_CTX`**

18–19 The first thing we need to do is construct the `SSL_CTX` object using `initialize_ctx()`. No further initialization is required on the client.

Connect to the server

20–21 We use our `tcp_connect()` function to connect to the server.

SSL handshake

22–28 Once we've created the TCP connection to the server, we need to turn on SSL. OpenSSL uses the `SSL` object to represent an SSL connection. The most important thing to notice about this stage is that we don't directly attach the `SSL` object to the socket. Rather, we create a `BIO` object using the socket and then attach the `SSL` object to the `BIO`. OpenSSL uses `BIO` objects to provide a layer of abstraction for I/O. As long as your object meets the `BIO` interface, it doesn't matter what the underlying I/O device is.

This abstraction layer provides even more flexibility than simply separating the TCP connect and the SSL handshake: It's perfectly possible with OpenSSL to do SSL handshakes on devices that aren't sockets at all—provided that you have an appropriate `BIO` object. For instance, one of the OpenSSL test programs connects an SSL client and server purely through memory buffers. A more practical use would be to support some protocol that can't be accessed via sockets. For instance, you could run SSL over a serial line.

Ex.1024
Askeladden, L.L.C. / Page 331 of 788

sclient.c

```
10  int main(argc,argv)
11    int argc;
12    char **argv;
13    {
14      SSL_CTX *ctx;
15      SSL *ssl;
16      BIO *sbio;
17      int sock;

18      /* Build our SSL context*/
19      ctx=initialize_ctx(KEYFILE,PASSWORD);

20      /* Connect the TCP socket*/
21      sock=tcp_connect();

22      /* Connect the SSL socket */
23      ssl=SSL_new(ctx);
24      sbio=BIO_new_socket(sock,BIO_NOCLOSE);
25      SSL_set_bio(ssl,sbio,sbio);
26      if(SSL_connect(ssl)<=0)
27        berr_exit("SSL connect error");
28      check_cert_chain(ssl,HOST);

29      /* read and write */
30      read_write(ssl,sock);

31      destroy_ctx(ctx);
32    }
```

sclient.c

Figure 8.9 `sclient.c` `main()` function

Once we've established the SSL connection, we check the certificate chain in the now familiar fashion: we check the Common Name against the host name. The function `check_cert_chain()`, shown in Figure 8.10, is analogous to the certificate check we showed in Figure 8.7. Note, however, that we don't check the certificate chain length here. Instead, the length check is done automatically by OpenSSL as long as we set the maximum chain length using `SSL_CTX_set_verify_depth()`, as we did in `initialize_ctx()`.

client.c

```
25  void check_cert_chain(ssl,host)
26    SSL *ssl;
27    char *host;
28    {
29      X509 *peer;
30      char peer_CN[256];

31      if(SSL_get_verify_result(ssl)!=X509_V_OK)
32        berr_exit("Certificate doesn't verify");

33      /*Check the cert chain. The chain length
34        is automatically checked by OpenSSL when we
35        set the verify depth in the ctx

36        All we need to do here is check that the CN
37        matches
38      */
39      /*Check the common name*/
40      peer=SSL_get_peer_certificate(ssl);
41      X509_NAME_get_text_by_NID(X509_get_subject_name(peer),
42        NID_commonName, peer_CN, 256);
43      if(strcasecmp(peer_CN,host))
44      err_exit("Common name doesn't match host name");
45    }
```

client.c

Figure 8.10 `check_cert_chain()` function

8.6 Server Accept

The code for the server side of the SSL connection is conceptually similar to the client side. As before, the toolkit APIs mimic the programming idioms for the languages they're written in. Thus, the Java code uses a subclass of `ServerSocket` and the C code uses `accept()` and `SSL_accept()`.

Java Server Accept

`java.net` uses a class called `ServerSocket` to accept connections as a server. `ServerSocket.accept()` returns a `Socket`. `Sockets` behave identically whether they were created using `ServerSocket.accept()` or `new Socket()`. PureTLS mimics this using a class called `SSLServerSocket`. The code to do an accept using PureTLS is shown in Figure 8.11.

Server.java

```
23          SSLServerSocket listen=new SSLServerSocket(ctx,port);

24          while(true){
25            SSLSocket s=(SSLSocket)listen.accept();

26            // Process this connection in a new thread
27            ReadWrite rw=new ReadWrite(s,s.getInputStream(),
28              s.getOutputStream());
29            rw.start();
30          }
```

Server.java

Figure 8.11 PureTLS server accept loop

This code is fairly simple. After we've created our `SSLServerSocket` we enter an infinite loop, calling `accept()` and then serve the resulting socket. The return value of `accept()` is a `Socket`. In fact, it's possible to SSL-enable a server even if it's completely SSL oblivious merely by passing it an `SSLServerSocket` rather than a `ServerSocket`. This is only possible because the PureTLS classes extend the `java.net` classes.

In practice, however, most applications want to be able to use `SSLSocket`-specific methods so they cast the return value of `accept()` to an `SSLSocket`, as we do in line 25. If we wanted, for instance, to discover the cipher suite we negotiated, this cast would be necessary. In line 25 we cast the return value of `accept()` to an `SSLSocket` because `accept()` returns a `Socket`. If we were willing to write all of our code simply in terms of `Socket`'s methods, then we would not make this cast, but would rather operate solely in terms of `Socket` methods.

C Server Accept

As with the C client accept, OpenSSL requires the programmer to perform the TCP accept. With the TCP socket in hand, we then create `BIO` and `SSL` objects, exactly as

we did to connect. However, instead of calling `SSL_connect()` we instead call `SSL_accept()`, as shown in Figure 8.12.

sserver.c

```
90          sock=tcp_listen();

91          while(1){
92            if((s=accept(sock,0,0))<0)
93              err_exit("Problem accepting");

94            sbio=BIO_new_socket(s,BIO_NOCLOSE);
95            ssl=SSL_new(ctx);
96            SSL_set_bio(ssl,sbio,sbio);

97            if((r=SSL_accept(ssl)<=0))
98              berr_exit("SSL accept error");

99            echo(ssl,s);
100         }
```

sserver.c

Figure 8.12 OpenSSL server accept loop

One difference between the C code and the Java code that isn't instantly apparent is that `sserver` can handle only one client at a time. By contrast, `Server` (our Java server) can handle an arbitrary number of clients. The difference is hinted at in line 29 of Figure 8.11: The `ReadWrite` class that we use to echo data between client and server is subclassed from `Thread`. When we call `start()` it forks off a new thread to handle the new client, leaving the original thread to handle new clients.

We could get a somewhat similar effect by calling `fork()` in `sserver`. However, we'd need to make a few more changes to get as clean a result as we get in Java. We'll cover this topic in more detail in Section 8.8.

8.7 Simple I/O Handling

Once we've established a SSL connection, we'd like to be able to send data over it. In principle, this can be done in a fashion analogous to sending data over TCP sockets. In practice, using SSL introduces some subtleties that need to be dealt with. We'll start with a simple task and build toward progressively more complex cases.

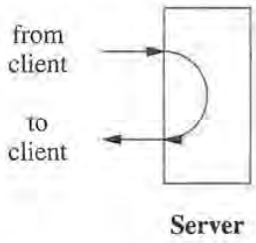


Figure 8.13 Server I/O

The simplest example of SSL I/O in these programs is the server side. Figure 8.13 reprises the server-side of our diagram from Figure 8.1. As you can see, the server's job is very simple: read data from the client and echo it back. If there's no data available to write to the client, do nothing. Figure 8.14 shows this code.

The implementation of `echo()` should be familiar to anyone who's ever written a TCP server. It's mostly just the standard TCP read/write loop with calls to `SSL_read()` and `SSL_write()` replacing calls to `read()` and `write()`. This provides us with a nice illustration of how OpenSSL does things without having to concern ourselves with the new complexities introduced by SSL; we'll consider those in Section 8.9.

`echo()` is passed the `SSL` object to read from in the argument `ssl`. It's also passed the socket that `ssl` is attached to in the argument `s`. We don't use `s` for most of the function but we need it when we close down the socket at the very end.

Read data

10–20 The first thing we do in our loop is read some data from the client side of our SSL connection. This is easily done by choosing a buffer of appropriate size and calling `SSL_read()`. Note that buffer size isn't really that important here. The semantics of `SSL_read()`, like the semantics of `read()`, are that it returns the data that is available, even if it's less than the requested amount. On the other hand, if no data is available, then the call to read *blocks*—the call waits until some data is available and returns that.

The choice of `BUFSIZZ`, then, is basically a performance tradeoff. The tradeoff is quite different here than when we're simply reading from normal sockets. In that case, each call to `read()` requires a context switch into the kernel. Because context switches are expensive, programmers try to use large buffers to reduce them. However, when we're using SSL the number of calls to `read()`—and hence context switches—is largely determined by the number of records the data was written in rather than the number of calls to `SSL_read()`.

For instance, if the client wrote a 1000-byte record and we call `SSL_read()` in chunks of 1 byte, then the first call to `SSL_read()` will result in the record being read

echo.c

```
3   void echo(ssl,s)
4     SSL *ssl;
5     int s;
6     {
7       char buf[BUFSIZZ];
8       int r,len,offset;

9       while(1){
10        /* First read data */
11        r=SSL_read(ssl,buf,BUFSIZZ);
12        switch(SSL_get_error(ssl,r)){
13          case SSL_ERROR_NONE:
14            len=r;
15            break;
16          case SSL_ERROR_ZERO_RETURN:
17            goto end;
18          default:
19            berr_exit("SSL read problem");
20        }

21        /* Now keep writing until we've written everything*/
22        offset=0;

23        while(len){
24          r=SSL_write(ssl,buf+offset,len);
25          switch(SSL_get_error(ssl,r)){
26            case SSL_ERROR_NONE:
27              len-=r;
28              offset+=r;
29              break;
30            default:
31              berr_exit("SSL write problem");
32          }
33        }
34      }
35    end:
36      SSL_shutdown(ssl);
37      SSL_free(ssl);
38      close(s);
39    }
```

echo.c

Figure 8.14 `echo()`: a simple server echo routine

in and the rest of the calls will just read it out of the SSL buffer. Thus, the choice of buffer size is less significant when we're using SSL than with normal sockets.

> Note that if the data were written in a series of small records, you might want to read all of them at once with a single call to `read()`. OpenSSL provides a flag `SSL_CTRL_SET_READ_AHEAD` that turns on this behavior.

Note the use of the `switch` on the return value of `SSL_get_error()` in line 12. The convention with normal sockets is that any negative number (typically –1) indicates failure and that one then checks `errno` to determine what actually happened. Obviously `errno` won't work here since that only shows system errors and we'd like to be able to act on SSL errors. Also, `errno` requires careful programming in order to be thread safe.

Instead of `errno`, OpenSSL provides the `SSL_get_error()` call. This call lets us examine the return value and figure out whether an error occurred and what it was. In this case, we're concerned with only one of three cases.

The return value was positive. In this case, we've read some data and we set the `len` value to that length in order to write it later.

The return value was zero. This does *not* mean that there was no data available. In that case, we would have blocked as discussed above. Rather, it means that the socket is closed and there will never be any data available to read. Thus, we exit the loop.

The return value was something negative. This signals some kind of error, but we need to use `SSL_get_error()` to find out exactly what. Since this program has no error handling, we simply default to `berr_exit()` which prints an error message and exits. A more sophisticated program would have more switch branches to determine exactly what kind of error had occurred and take appropriate action.

Note that in this case, since we handle all error conditions the same, we could simply have checked the return value for being positive, negative, or zero, and acted accordingly. Indeed, another perfectly good idiom would be to only enter the switch statement if the value were negative, handling the non-error cases directly inline. Also note that `SSL_get_error()` is an OpenSSL specific feature. Some other toolkits choose to use the return value directly to indicate the error value.

Write data

21–33 By the time we get to this section of the code we've definitely read some data from the client. That data is stored in `buf` (starting at position 0) and is of length `len`. The job of this code is to echo that data back to the client (i.e., to write it to `ssl`).

At first glance, the outer `while()` loop may seem puzzling. After all, there was no such loop when we were reading data from the network. The reason for this is simple: we want to write all the data to the network. The semantics of `SSL_write()` guarantee only that some data will be written, not necessarily all of it. The return value tells us how much. Thus, if not all the data is written in one call, we need to loop until we've written it all.

However, we want to write the data that we haven't written, not rewrite the data

we've already written. In order to do this, we need to maintain a pointer to the end of the data that we've written (and hence to the beginning of the data we need to write). Thus, we keep the variable `offset`. At the beginning of this loop it is set to zero because we've written no data. However, once we write some data, we increment it by the length of the data written and decrement `len` accordingly. Finally, when `len` is zero (and `offset` points to the end of the buffer) we exit the loop.

Why is it so important to write all the data? Consider the case where the user types something and then waits for the server to respond. Now, the server goes to write the data but only succeeds in writing part of it. Without the loop at line 23, the server would jump back to line 9 and try to read data from the client. However, the client is waiting for the server, so there isn't any data to read. The server just blocks in the `SSL_read()`. We've now got a deadlock because the client is waiting for the server but the server is waiting for the client—even though there is something the server could be doing: writing the data.

The way out of this situation, of course, is for the server to finish writing all of its data before it waits for the client. Although ultimately in this situation the server will have to write all of its data first, there are other ways to arrange that than looping around the `SSL_write()`. We'll see another approach to this problem in Section 8.8.

> In practice, this call to `SSL_write()` should always write all the data before returning. By default, OpenSSL insists on this behavior. However, if the `SSL_MODE_ENABLE_PARTIAL_WRITE` flag is set, then it will return after writing one record even if there is more than one record's worth of data in the write buffer. Thus, if this flag was set and we were working with larger blocks than 1024 bytes, we might very well see partial writes.

As we did when reading, we use `SSL_get_error()` to process the return value. However, we're only concerned with two cases here:

The return value was positive. In this case, we wrote some data.

The return value was non-positive. In this case, some error occurred. Note that we shouldn't ever see a zero return value. If no data can be written then we'll block. If the socket is closed we should see an actual error.

Close the connection

36–39 When we reach the bottom of the loop then we know that we've read all the data that the client will send, so we are ready to close the socket. We call `SSL_shutdown()` to send the close_notify and then free the `SSL` structure and close the socket.

8.8 Multiplexed I/O Using Threads

The simple I/O discipline we saw in the previous section works fine for the server because its only source of input is the client. However, this approach is inadequate for the client. Consider Figure 8.15, which shows the inputs and outputs of the client. The

client has to listen simultaneously to two sources of input, the keyboard and the server. Input from the keyboard and the server can appear *asynchronously*. That is to say that they can appear in any order. This means that a simple read/write discipline like we had for the server is fundamentally inadequate.

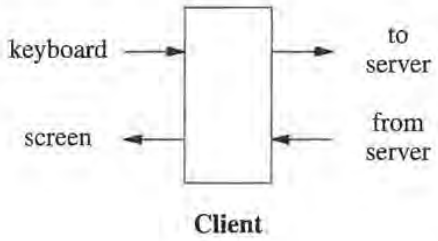


Figure 8.15 Client I/O

It's easy to see this, even with our simple server. Consider an I/O discipline analogous to the one we used in the server, represented by the pseudo-code in Figure 8.16.

```
1        while(1){
2          read(keyboard,buffer);
3          write(server,buffer);

4          read(server,buffer);
5          write(screen,buffer);
6        }
```

Figure 8.16 A broken client I/O discipline

Now, imagine that we have a slow server. The user types some input at the keyboard, which the client sends to the server. However, the server is slow and takes a while to respond. In the meantime, the user has typed in even more data—but the client can't read or write it because it's stuck at line 4, waiting to read from the server. Finally, the server writes the data and the client writes it to the screen. This frees the client to read the data from the client and send it to the server. Already, this behavior is kind of unpleasant. We could be writing data to the server but we can't because it's slow. On the other hand, this doesn't directly affect the user experience because our protocol is so simple—it's just unaesthetic. Now let's consider an example in which this behavior would cause serious problems: remote login.

Consider the case in which you're remotely logged into a machine and you request a directory listing. Your request is a single line (`ls` on UNIX boxes) but the response is a large number of lines. In general, these lines will be written in more than one write. Thus, it may very well take more than one read in order to read them from the server. However, if we use the I/O discipline in Figure 8.16, we'll run into a problem.

We read the command from the user and the first chunk of the server's response, but after that we get deadlocked. The client is waiting in line 2 for the user to type something but the user is waiting for the rest of the directory listing. We're deadlocked. Recognize that this is the dual of the deadlock we saw in Section 8.7 and the solution is the same: we need to read all the available data from the server.

However, the precise solution we employed before won't work. Recall that calling `SSL_read()` when there is no data available simply blocks waiting for the server. We can't have that because we'd just get stuck at the end of the server's data when what we actually need to do is read from the keyboard. Similar arguments can be used to show that no possible arrangement of sequenced reads and writes can work correctly. Instead, we need some way to ensure that we read data from either the keyboard or the server, depending on which one is ready.

The traditional solution to this problem depends on which language you're using. In languages like Java, the solution is to use threads. In languages like C, where threads support is nonexistent or non-standard, more complicated mechanisms have been developed. This section shows a threaded solution written in Java. The next section shows a solution in C using `select()`.

Figure 8.17 shows the `main()` method for our Java client. Lines 12–14

SClient.java

```
11          public static void main(String []args)
12            throws IOException {
13            SSLContext ctx=createSSLContext(keyfile,password);
14            SSLSocket s=connect(ctx,host,port);

15            // This thread reads from the console and writes to the server
16            ReadWrite c2s=new ReadWrite(s,System.in,s.getOutputStream());
17            c2s.start();

18            // This thread reads from the server and writes to the console
19            ReadWriteWithCancel s2c=new ReadWriteWithCancel(s,
20              s.getInputStream(),System.out,c2s);
21            s2c.start();
22            s2c.setPriority(Thread.MAX_PRIORITY);
23            s2c.join();
24          }
```

SClient.java

Figure 8.17 Java client `main()`

just call the context and socket creation code we've seen before, so we ignore them. What's important is the rest of the function.

Start threads

15–23 The general structure of the client is simple. Instead of having a single read/write loop like the one in Figure 8.16, we have two such loops, each operating in a separate thread. Individually, the loops simply block when they don't have any data to read. However, this doesn't result in deadlock because the thread scheduler automatically runs whichever thread has data to read or write.

We accomplish this by instantiating two `ReadWrite` objects. (`ReadWriteWithCancel` is a subclass of `ReadWrite` that we use to get the right behavior during connection closure. We'll examine this further in Section 8.10. For now think of `ReadWriteWithCancel` as `ReadWrite`.) `ReadWrite` subclasses `Thread` so when we call `start()` it automatically starts as a new thread.

> Line 22 shouldn't be necessary. It's a workaround for a problem in the JDK 1.1.8 thread scheduler. Without it, the `c2s` thread starves the `s2c` thread. This hack is not necessary in JDK 1.2.X. The `join()` in line 23 causes this thread to wait for the `s2c` thread to finish. This isn't strictly necessary in this case but would be if we wanted to know when the connection to the server was finished.

Figure 8.18 shows the relevant section of `ReadWrite`. We've omitted the closure code because we'll treat that in Section 8.10. The constructor simply creates the object and assigns the arguments to instance variables. Note that the arguments we provide include an `InputStream` and an `OutputStream`. These are standard Java I/O classes which PureTLS subclasses. Thus, this code is totally oblivious to PureTLS. It could be used exactly as-is with standard sockets!

ReadWrite.java

```
 8    public class ReadWrite extends Thread {
 9        protected SSLSocket s;
10        protected InputStream in;
11        protected OutputStream out;

12        /** Create a ReadWrite object

13            @param s the socket we're using
14            @param in the stream to read from
15            @param out the stream to write to
16        */
17        public ReadWrite(SSLSocket s,InputStream in,OutputStream out){
18          this.s=s;
19          this.in=in;
20          this.out=out;
21        }
```

```
22              /** Copy data from in to out.*/
23              public void run() {
24                byte[] buf=new byte[1024];
25                int read;

26                try {
27                  while(true){
28                    // Check for thread termination
29                    if(isInterrupted())
30                      break;

31                    // Read data in
32                    read=in.read(buf);

33                    // Exit if there is no more data available
34                    if(read==-1)
35                      break;

36                    // Write the data out
37                    out.write(buf,0,read);
38                  }

39                } catch (IOException e){
40                  // run() can't throw IOException
41                  throw new InternalError(e.toString());
42                }

43                // Finalize
44                onEOD();
45              }
```

ReadWrite.java

Figure 8.18 `ReadWrite` I/O handling

Read and write

22–45 Because `ReadWrite` is subclassed from `Thread` it has certain automatic behaviors. In particular, it inherits a method called `start()`. The `start()` method automatically creates a new thread and then calls `run()`. This is why the code that actually does the work of `ReadWrite` is in the `run()` method.

The `run()` method itself is simple. We simply `read()` off of the `InputStream` (`in`) and write to the `OutputStream` (`out`). Unlike OpenSSL, PureTLS guarantees that when you call `write()` all of the data will be delivered to the network before it returns. Thus, we don't need to even loop around `write()` the way we did in Figure 8.8. Obviously, this behavior would be disastrous if the whole program blocked on `write()`; because we're operating in our own thread, there's no problem at all.

PureTLS handles network problems differently from OpenSSL. Standard Java practice is to have network errors delivered via exceptions and PureTLS follows this practice. Thus, when we read data in lines 32–35 we don't need to check for read errors. All we need to do is distinguish a successful read from an end of data. This is easy since `read()` returns the length of the data read or a −1 for end of data. If we read some data, we fall through to the write code at line 37. Otherwise, we exit the loop and close the connection.

As we said, network errors are handled by throwing an `IOException`. In this program, we simply catch these errors, treat them as fatal, and throw an `InternalError` in response. In a real program, we'd make some attempt to determine fatal from nonfatal errors and report them to the user.

8.9 Multiplexed I/O with `select()`

Although the threaded approach works fine for Java, it is far less satisfactory for C. Threads support is not part of the C standard and so it's left up to the platform to provide whatever threading mechanisms it chooses. Windows provides fairly complete threading support and threads are the multiplexing method of choice on that platform.

Unfortunately, the situation on UNIX is far worse. UNIX threads support is not standardized and, worse yet, many C-based libraries on UNIX aren't thread safe. If you want portable code on UNIX, you have to avoid threads. This means that if you want to multiplex I/O you need to use `select()`. Although this use of `select()` is a pretty common UNIX idiom, the interaction of `select()` with SSL is far from clean and requires understanding some subtleties we've so far only hinted at.

Read

The basic problem we're facing is that SSL is a record-oriented protocol. Thus, even if we want to read only one byte from the SSL connection, we still need to read the entire record containing that byte into memory. Without the entire record in hand, the SSL implementation can't check the record MAC and so we can't safely deliver the data to the programmer. Unfortunately, this behavior interacts poorly with `select()`, as shown in Figure 8.19.

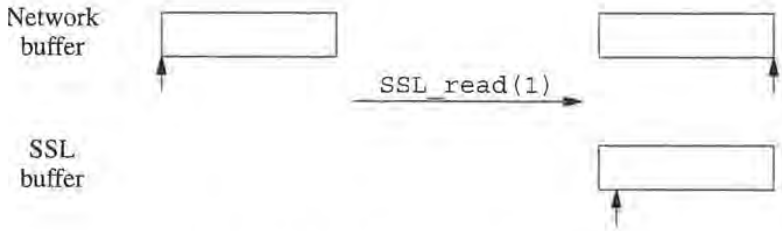


Figure 8.19 Read interaction with SSL

The left-hand side of Figure 8.19 shows the situation when the machine has received a record but it's still waiting in the network buffers. The arrow represents the read pointer which is set at the beginning of the buffer. The bottom row represents data decoded by the SSL implementation but not yet read by the program (the *SSL buffer*). This buffer is currently empty so we haven't shown a box. If the program calls `select()` at this point, it will return immediately indicating that a call to `read()` will succeed. Now, imagine that the programmer calls `SSL_read()` requesting one byte. This takes us to the situation at the right side of the figure.

As we said earlier, the SSL implementation has to read the entire record in order to deliver even a single byte to the program. In general, the application does not know the size of records and so its reads will not match the records. Thus, the box in the upper right-hand corner shows that the read pointer has moved to the end of the record. We've read all the data in the network buffer. When the implementation decrypts and verifies the record, it places the data in the SSL buffer. Then it delivers the one byte that the program asked for in `SSL_read()`. We show the SSL buffer in the lower right-hand corner. The read pointer points somewhere in the buffer, indicating that some of the data is available for reading but some has already been read.

Consider what happens if the programmer calls `select()` at this point. `select()` is concerned solely with the contents of the network buffer, and that's empty. Thus, as far as `select()` is concerned there's no data to read. Depending on the exact arguments it's passed, it will either return saying that there's nothing to read or wait for some more network data to become available. In either case we wouldn't read the data in the SSL buffer. Note that if another record arrived then `select()` would indicate that the socket was ready to read and we'd have an opportunity to read more data.

Thus, `select()` is an unreliable guide to whether there is SSL data ready to read. We need some way to determine the status of the SSL buffer. This can't be provided by the operating system because it has no access to the SSL buffers. It must be provided by the toolkit. OpenSSL provides exactly such a function. The function `SSL_pending()` tells us whether there is data in the SSL buffer for a given socket. Figure 8.20 shows `SSL_pending()` in action.

The logic of this code is fairly straightforward. `select()` has been called earlier, setting the variable `readfds` with the sockets that are ready to read. If the SSL socket is ready to read, we go ahead and try to fill our buffer. Once we've read some data, we write it to the console. Then we check with `SSL_pending()` to see if the record was longer than our buffer. If it was, we loop back and read some more data.

Note that we've added a new branch to our `switch` statement: a check for `SSL_ERROR_WANT_READ`. What's going on here is that we've set the socket for *nonblocking* operation. Recall that we said in Section 8.7 that if you called `read()` when the network buffers were empty, it would simply block (wait) until they weren't. Setting the socket to nonblocking causes it to return immediately saying that it *would have* blocked.

To understand why we've done this, consider what happens if an SSL record arrives in two pieces. When the first piece arrives, `select()` will signal that we're ready to read. However, we need to read the entire record in order to return any data so this is a

read_write.c

```
43          /* Now check if there's data to read */
44          if(FD_ISSET(sock,&readfds)){
45            do {
46              r=SSL_read(ssl,s2c,BUFSIZZ);
47              switch(SSL_get_error(ssl,r)){
48                case SSL_ERROR_NONE:
49                  fwrite(s2c,1,r,stdout);
50                  break;
51                case SSL_ERROR_ZERO_RETURN:
52                  /* End of data */
53                  if(!shutdown_wait)
54                   SSL_shutdown(ssl);
55                  goto end;
56                  break;
57                case SSL_ERROR_WANT_READ:
58                  break;
59                default:
60                  berr_exit("SSL read problem");
61              }
62            } while (SSL_pending(ssl));
63          }
```

read_write.c

Figure 8.20 Reading data using `SSL_pending()`

false positive. Attempting to read all that data will block, leading to exactly the deadlock we were trying to avoid. Instead, we set the socket to nonblocking and catch the error, which OpenSSL translates to `SSL_ERROR_WANT_READ`.

Write

When we're writing to the network we have to face the same sort of inconsistency that we had when reading. Again, the problem is the all-or-nothing nature of SSL record transmission. For simplicity's sake, let's consider the case where the network buffers are mostly full and the program attempts to perform a modest-sized write, say 1K. This is illustrated in Figure 8.21.

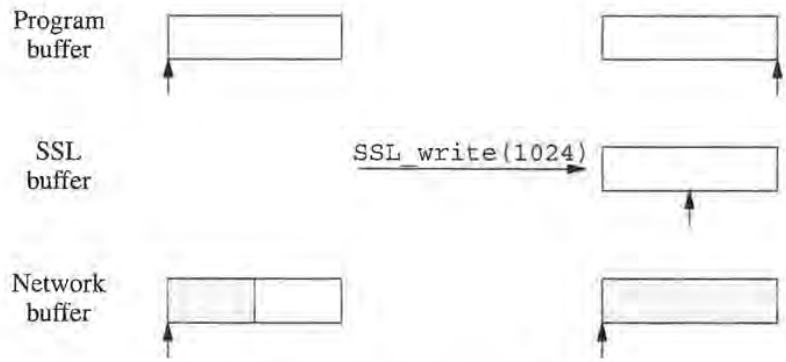


Figure 8.21 Write interaction with SSL

Again, the left-hand side of the figure represents our initial situation. The program has 1K to write in some buffer. The write pointer is set at the beginning of that buffer. The SSL buffers are empty. The network buffer is half-full (the shading indicates the full region). The write pointer is set at the beginning. We've deliberately obscured the distinction between the TCP buffers and the size of the TCP window because it's not relevant here. Suffice it to say that the program can safely write 512 bytes without blocking.

Now, the program calls `SSL_write()` with a 1024-byte block. The toolkit has no way of knowing how much data it can write safely, so it simply formats the buffer as a single record, thus moving the write pointer in the program buffer to the end of the buffer. We can ignore the slight data expansion from the SSL header and MAC and simply act as if the data to be written to the network was 1024 bytes.

Now, what happens when the toolkit calls `write()`? It successfully writes 512 bytes but gets a *would block* error when it attempts to write to the end of the record. As a consequence, the write pointer in the SSL buffer is moved halfway across—indicating that half of the data has been written to the network. The network buffer is shaded to indicate that it's completely full. The network write pointer hasn't moved.

We now need to concern ourselves with two questions: first, how does the toolkit indicate this situation to the user and, second, how does the user arrange that the SSL buffer gets flushed when space is available in the network buffer? The kernel will automatically flush the network buffer when possible so we don't need to worry about arranging for that. We can use `select()` to see when there is more space available in the network buffer and we should therefore flush the SSL buffer. There are at least two strategies for handling this situation. We'll describe the method that OpenSSL uses and then describe an alternate strategy.

OpenSSL Write Handling

Once OpenSSL has received a *would block* error from the network, it aborts and propagates that error all the way up to the application. Note that this does *not* mean that it throws away the data in the SSL buffer. This is impossible because part of the record might already have been sent.

In order to flush this buffer, the programmer must call `SSL_write()` again with the same buffer that it called the first time (it's permissible to extend the buffer but the start must be the same.) OpenSSL automatically remembers where the buffer write pointer was and only writes the data after the write pointer. Figure 8.22 shows this process in action.

read_write.c

```
66          /* Check for input on the console*/
67          if(FD_ISSET(fileno(stdin),&readfds)){
68            c2sl=read(fileno(stdin),c2s,BUFSIZZ);
69            if(c2sl==0){
70              shutdown_wait=1;
71              if(!SSL_shutdown(ssl))
72                return;
73            }
74            c2s_offset=0;
75          }

76          /* If we've got data to write then try to write it*/
77          if(c2sl && FD_ISSET(sock,&writefds)){
78            r=SSL_write(ssl,c2s+c2s_offset,c2sl);

79            switch(SSL_get_error(ssl,r)){
80              /* We wrote something*/
81              case SSL_ERROR_NONE:
82                c2sl-=r;
83                c2s_offset+=r;
84                break;

85              /* We would have blocked */
86              case SSL_ERROR_WANT_WRITE:
87                break;

88              /* Some other error */
89              default:
90                berr_exit("SSL write problem");
91            }
92          }
```

read_write.c

Figure 8.22 Client to server writes using OpenSSL

Read from console

66–75 The first thing we need to do is have some data to write. Thus, we check to see if the console is ready to read, and if so read whatever's there (up to `BUFSIZZ` bytes) into the buffer `c2s`, placing the length in the variable `c2sl`.

Write to network

76–92 If `c2sl` is nonzero and the network buffers are (at least partially) empty, then we have data to write to the network. As usual, we call `SSL_write()` with the buffer `c2s`. As before, if we manage to write some but not all of the data, we simply increment `c2s_offset` and decrement `c2sl`.

The new behavior here is that we check for the error `SSL_ERROR_WANT_WRITE`. This error indicates that we've got unflushed data in the SSL buffer. As we described above, we need to call `SSL_write()` again with the same buffer, so we simply leave `c2sl` and `c2s_offset` unchanged. Thus, the next time `SSL_write()` is called it will automatically be with the same data.

> OpenSSL actually provides a flag called `SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER` which allows you to call `SSL_write()` with a different buffer after a would block error. However, this merely allows you to allocate a new buffer with the same contents. `SSL_write()` still seeks to the same write pointer before looking for new data.

An Alternative Nonblocking Strategy

SPYRUS's TLSGold toolkit (a commercial C-based toolkit) implements a somewhat different strategy for handling nonblocking I/O. As long as the toolkit is able to completely encode the program write buffer, it returns success—even if it can't completely flush the SSL buffer. The only case in which it returns indicating that it would block is if there is already data in the SSL buffers and it can't flush them entirely. This behavior more closely matches the semantics of UNIX sockets.

However, because the caller has no indication that the data actually went to the SSL buffer and not to the network directly, he has to take positive action. TLSGold provides a call `SSL_IsMoreWriteData()` which performs the same function for writes that we saw `SSL_pending()` perform for reads. Thus, when programming TLSGold, the programmer needs to check `TSW_SSL_IsMoreWriteData()` and attempt to flush the buffers when the network is ready to write. Flushing the write buffers is accomplished by calling `TSW_SSL_Write()` with a zero-length buffer.

A Complete `select()`-based Solution

Figure 8.23 shows the complete code of `read_write()`. We've seen most of these pieces before but it's useful to examine the function as a whole.

read_write.c

```
8   void read_write(ssl,sock)
9     SSL *ssl;
10    {
11      int width;
12      int r,c2sl=0,c2s_offset=0;
13      fd_set readfds,writefds;
14      int shutdown_wait=0;
15      char c2s[BUFSIZZ],s2c[BUFSIZZ];
16      int ofcmode;

17      /*First we make the socket nonblocking*/
18      ofcmode=fcntl(sock,F_GETFL,0);
19      ofcmode|=O_NDELAY;
20      if(fcntl(sock,F_SETFL,ofcmode))
21        err_exit("Couldn't make socket nonblocking");


22      width=sock+1;

23      while(1){
24        FD_ZERO(&readfds);
25        FD_ZERO(&writefds);

26        FD_SET(sock,&readfds);

27        /*If we've still got data to write then don't try to read*/
28        if(c2sl)
29          FD_SET(sock,&writefds);
30        else
31          FD_SET(fileno(stdin),&readfds);

32        r=select(width,&readfds,&writefds,0,0);
33        if(r==0)
34          continue;

35        /* Now check if there's data to read */
36        if(FD_ISSET(sock,&readfds)){
37          do {
38            r=SSL_read(ssl,s2c,BUFSIZZ);

39            switch(SSL_get_error(ssl,r)){
40              case SSL_ERROR_NONE:
41                fwrite(s2c,1,r,stdout);
42                break;
```