PRACTICAL CRYPTOGRAPHY

NIELS FERGUSON BRUCE SCHNEIER

USAA-1017

Practical Cryptography

Niels Ferguson
Bruce Schneier



Wiley Publishing, Inc.

Executive Publisher: Robert Ipsen
Executive Editor: Carol A. Long
Editorial Manager: Kathryn A. Malm
Managing Editor: Fred Bernardi

This book is printed on acid-free paper.

Copyright © 2003 by Niels Ferguson and Bruce Schneier. All rights reserved.

Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada

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, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: `permcoordinator@wiley.com`.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

ISBN: 0-471-22894-X (C)
ISBN: 0-471-22357-3 (P)

Printed in the United States of America

10 9 8 7 6 5 4 3 2

Chapter 9

Implementation Issues (I)

Now that we have come this far, we would like to talk a bit about implementation issues. Implementing cryptographic systems is sufficiently different from implementing normal programs to deserve its own treatment.

The big problem is, as always, the weakest-link property (see section 2.2). It is very easy to screw up the security at the implementation level. In fact, implementation errors (most commonly in the form of buffer overflows) are by far the biggest security problem in real-world systems. If you have been paying any attention to computer security problems over the last few years, you know what we mean. You rarely hear about cryptography systems that are broken in practice. This is not because the cryptography in most systems is any good; we've reviewed enough of them to know this is not the case. It is just easier to find an implementation-related hole than it is to find a cryptographic vulnerability, and attackers are smart enough not to bother with the cryptography when there is this much easier route.

So far in this book we have restricted our discussion to cryptography, but in this chapter we will focus more on the environment in which the cryptography operates. Every part of the system affects security, and to do a really good job the entire system must be designed from the ground up not just with security in mind, but with security as one of the primary goals. The "system" we're talking about is very big. It includes everything that could damage the security properties if it were to misbehave.

One major part is, as always, the operating system. But none of the operating systems in widespread use is designed with security as a primary goal. The logical conclusion to draw from this is that it is impossible to implement a secure system. We don't know how to do it, and we don't know anyone else who knows how to do it, either. Real-life systems include many components that were never designed for security, and that makes it impossible to achieve the level of security that we really need. So should we just give up? Of course not. When we design a cryptographic system, we do our very best to make sure that at least our part is secure. This might sound like a civil-servant mentality: all we care about is our little domain. But we *do* care about the other parts of the system; we just can't do anything about them. That is one of the reasons for writing this book: to get other people to understand the insidious nature of security, and how important it is to do it right.

Another important reason to get at least the cryptography right is one we mentioned before: attacks on the cryptography are especially damaging because they can be invisible. If the attacker succeeds in breaking your cryptography, you are unlikely to notice. This can be compared to a burglar who has a set of keys to your house. If the burglar exercises reasonable caution, how would you ever find out?

Our long-term goal is to make secure computer systems. To achieve that goal, everybody will have to do their part. Our work, and the work this book is about, is making the cryptography secure. Other parts of the system will have to be made secure, too. We don't know how to do this, but maybe other people do, or maybe we will learn in future. Until then, the overall security of the system is going to be limited by the weakest link, and we will do our utmost to ensure that the weakest link will never be the cryptography.

Another important reason to do the cryptography right is that it is very difficult to switch cryptographic systems once they've been implemented. An operating system runs on a single computer. Cryptographic systems are often used in communication protocols to let many computers communicate with each other. Upgrading the operating system of a single computer is feasible, and in practice it is done relatively often. Modifying the communication protocols in a network is a nightmare, and as a result many networks still use the designs of the 1970s and 1980s. We must keep in mind that any new cryptographic system we design today, if adopted widely, is quite likely

to still be used 30 or 50 years from now. We hope that by that time the other parts of the system will have achieved a much higher level of security.

9.1 Creating Correct Programs

The core of the implementation problem is that we in the IT industry don't know how to write a correct program or module. (A "correct" program is one that behaves exactly according to its specifications.) There are several reasons for the difficulty that we seem to have in writing correct programs.

9.1.1 Specifications

The first problem is that for most programs, there is no clear description of what they are supposed to do. If there are no specifications, then you cannot even check whether a program is correct or not. For such programs the whole concept of correctness is undefined.

Many software projects have a document called the functional specification. In theory, this should be the specification of the program. But in practice this document either does not exist, is incomplete, or specifies things that are irrelevant for the behavior of the program. Without clear specifications there is no hope of getting a correct program.

There are really three stages in the specification process:

Requirements Requirements are an informal description of what the program is supposed to achieve. It is really a "*what* can I do with it" document, rather than a "how exactly do I do something with it" document. Requirements are often a bit vague and leave details out in order to concentrate on the larger picture.

Functional specification The functional specifications give a detailed and exhaustive definition of the behavior of the program. The functional specification can only specify things that you can measure on the outside of the program.

For each item in the functional specifications, ask yourself whether you could create a test on the finished program that would determine whether that item was adhered to or not. The test can only use the external behavior of the program, not anything from the inside. If you can't create a test for an item, it does not belong in the functional specification.

The functional specifications should be complete. That is, every piece of functionality should be specified. Anything not in the functional specification does not have to be implemented.

Another way to think of the functional specification is as the basis for testing the finished program. Any item can, and should, be tested.

Implementation design This document has many names, but it specifies how the program works internally. It contains all of the things that cannot be tested from the outside. A good implementation design will often split the program into several modules, and describe their functionality. In turn, these module descriptions can be seen as the requirements for the module, and the whole cycle starts all over again for the module.

Of these three, the functional specification is without a doubt the most important one. This is the document against which the program will be tested when it is finished. You can sometimes get by with informal requirements, or an implementation design that is nothing but a few sketches on a whiteboard. But without functional specifications, there is no way to even describe what you have achieved in the end when the program is finished.

9.1.2 Test and Fix

The second problem in writing correct programs is the test-and-fix development method that is in almost universal use. Programmers write a program, and then test whether it behaves correctly. If it doesn't, they fix the bugs and test again. As we all know, this does not lead to a correct program. It results in a program that kind of works in the most common situations.

Back in 1972, Edsger Dijkstra commented in his Turing Award lecture that testing can only show the presence of bugs, never the absence of bugs [23].

This is very true, and ideally we would like to write programs that we can demonstrate to be correct. Unfortunately, current techniques in proving the correctness of programs are nowhere good enough to handle day-to-day programming tasks, let alone a whole project.

Computer scientists do not know how to solve this. Maybe it will be possible in the future to prove that a program is correct. Maybe we just need a far more extensive and thorough testing infrastructure and methodology. But even without having a full solution, we can certainly do our very best with the tools we do have.

There are some childishly simple rules about bugs that any good software engineering book includes:

- If you find a bug, first implement a test that detects the bug. Check that the bug is detected. Then fix the bug, and check that the test no longer finds the bug. And then keep running that test on every future version to make sure the bug does not reappear.
- Whenever you find a bug, think about what caused it. Are there any other places in the program where a similar bug might reside? Go check them all.
- Keep track of every bug you find. Simple statistical analysis of the bugs you have found can show you which part of the program is especially buggy, or what type of error is made most frequently, etc. Such feedback is necessary for a quality control system.

This is not even a bare minimum, but there is not a lot of methodology to draw from. There are quite a few books that discuss software quality. They don't all agree with each other. Many of them present a particular software development methodology as *the* solution, and we are always suspicious of such one-cure-does-it-all schemes. The truth is almost always somewhere in the middle.

9.1.3 Lax Attitude

The third problem is the incredibly lax attitude of most people in the computer industry. Errors in programs are just accepted as a matter of course.

If your word processor crashes and destroys a day's worth of work, everybody seems to think this is quite normal and acceptable. Often they blame the user: "You should have saved your work more often." Software companies routinely ship products with many known bugs in them. This wouldn't be so bad if they only sold computer games, but nowadays our work, our economy, and—more and more—our lives depend on software. If a car manufacturer finds a defect (bug) in a car after it was sold, they will recall the car and fix it. Software companies get away with disclaiming any and all liability in their software license, something they wouldn't be allowed to do if they produced any other product. This lax attitude means that no serious attempts are being made at producing correct software.

9.1.4 So How Do We Proceed?

Don't ever think that all you need is a good programmer or code reviews or an ISO 9001–certified development process or extensive testing or even a combination of all of them. Reality is much more difficult. Software is too complex to be tamed by a few rules and procedures. We find it instructive to look at the best engineering quality control system in the world: the airline industry. Everybody in that industry is involved in the safety system. There are very strict rules and procedures for almost every operation. There are multiple backups in case of failures. Every nut and bolt of the airplane has to be flight-qualified before it can ever be used. Anytime a mechanic takes a screwdriver to the plane, his work is checked and signed off by a supervisor. Every modification is carefully recorded. Any accident is meticulously investigated to find all the underlying causes, which are then fixed. This fanatical pursuit of quality has a very high cost. An airplane is probably an order of magnitude more expensive than it would be if you just sent the drawings to an ordinary engineering firm. But the pursuit of quality has also been amazingly effective. Flying is an entirely routine operation today, in a machine where every failure is potentially fatal. A machine where you cannot just hit the brakes and stop when something goes wrong. One where the only safe way back to the ground is the quite delicate operation of landing on one of the rare specially prepared spots in the world. The airline industry has been amazingly effective at making flying secure. We would do well to learn all we can from them. Maybe

writing correct software *does* cost an order of magnitude more than what we are used to now. But given the cost to society of the bugs in software that we see today, we are sure that it would be cost-effective in the long run.

9.2 Creating Secure Software

So far we have only talked about correct software. Just writing correct software is not good enough for a security system. The software must be secure as well.

What is the difference? Correct software has a specified functionality. If you hit button A, then B will happen. Secure software has an additional requirement: a *lack* of functionality. No matter what the attacker does, she cannot do X. This is a very fundamental difference; you can test for functionality, but not for lack of functionality. The security aspects of the software cannot be tested in any effective way, which makes writing secure software much more difficult than writing correct software. The inevitable conclusion is:

Standard implementation techniques
are entirely inadequate to create secure code.

We actually don't know how to create secure code. Software quality is a vast area that would take several books to cover. We don't know enough about it to write those books, but we *do* know the cryptography-specific issues and the problems that we see most frequently, and that is what we will discuss in the rest of this chapter.

Before we start, let us make our point of view clear: unless you are willing to put real effort into developing a secure implementation, there is no point in bothering with the cryptography at all. Designing cryptographic systems might be fun, but it is a waste of time if you implement them badly.

9.3 Keeping Secrets

Anytime you work with cryptography, you are dealing with secrets. And secrets have to be kept. This means that the software that deals with the secrets has to ensure that they don't leak out.

For the secure channel we have two types of secrets: the keys and the data. Both of these secrets are transient secrets; we don't have to store them for a long time. The data is only stored while we process each message. The keys are only stored for the duration of the secure channel. Here we will only discuss keeping transient secrets. For a discussion on storing secrets long-term, see chapter 22.

Transient secrets are kept in memory. Unfortunately, the memory on most computers is not very secure. We will discuss each of the typical problems in turn.

9.3.1 Wiping State

A basic rule of writing security software: wipe any information as soon as you no longer need it. The longer you keep it, the higher the chance that someone will be able to access it. What's more, you should definitely wipe the data before you lose control over the underlying storage medium. For transient secrets, this involves wiping the memory locations.

This sounds easy to do, but it leads to a surprising number of problems. If you write the entire program in C, you can take care of the wiping yourself. If you write a library for others to use, then you have to depend on the main program to inform you that the state is no longer needed. For example, when the communication connection is closed, the crypto library should be informed so that it can wipe the secure channel session state. The library can contain a function for this, but we all know that the programmer of the application is probably not going to bother calling this function. After all, the program works perfectly well without calling this function.

In some object-oriented languages, things are a bit easier. In C++, there is a destructor function for each object, and the destructor can wipe the state. This is certainly standard practice for security-relevant code in C++. As

long as the main program behaves properly and destroys all objects it no longer needs, the memory state will be wiped. The C++ language ensures that all stack-allocated objects are properly destroyed when the stack is unwound during exception handling, but the program has to ensure that all heap-allocated objects are destroyed. Calling an operating system function to exit the program might not even unwind the call stack. And you have to ensure that all sensitive data is wiped even if the program is about to exit. After all, the operating system gives no guarantees that it will wipe the data soon, and some operating systems don't even bother wiping the memory before they give it to the next application.

Even if you do all this, the computer might still frustrate your attempts. Some compilers try too hard to optimize. A typical security-relevant function performs some computations in local variables, and then tries to wipe them. You can do this in C with a call to the `memset` function. Good compilers will optimize the `memset` function to in-line code, which is more efficient. But some of them are too clever by half. They detect that the variable or array that is being wiped will never be used again, and "optimize" the `memset` away. It's faster, but suddenly the program does not behave the same way anymore. It is not uncommon to see code that reveals data that it happens to find in memory. If the memory is given to some library without having been wiped first, the library might leak the data to an attacker. So check the code that your compiler produces, and make sure the secrets are actually being wiped.

In a language like Java, the situation is even more complicated. All objects live on the heap, and the heap is garbage-collected. This means that the finalization function (similar to the C++ destructor) is not called until the garbage collector figures out that the object is no longer in use. There are no specifications about how often the garbage collector is run, and it is quite conceivable that secret data remains in memory for a very long time. The use of exception handling makes it hard to do the wiping by hand. If an exception is thrown, then the call-stack unwinds without any way for the programmer to insert his own code, except by writing *every* function as a big try clause. The latter solution is so ugly that it is impractical. It also has to be applied throughout the program, making it impossible to create a security library for Java that behaves properly. During exception handling, Java happily unwinds the stack, throwing away the references to the objects

without cleaning up the objects themselves. Java is really bad in this respect. The best solution we've been able to come up with is to at least ensure that the finalization routines are run at program exit. The `main` method of the program uses a `try-finally` statement. The finally block contains some code to force a garbage collect, and to instruct the garbage collector to attempt to complete all the finalization methods. (See the functions `System.gc()` and `System.runFinalization()` for more details.) There is still no guarantee that the finalization methods will be run, but it is the best we've been able to find.

What we really need is support from the programming language itself. In C++ it is at least theoretically possible to write a program that wipes all states as soon as they are no longer needed, but many other features of the language make it a poor choice for security software. Java makes it very difficult to wipe the state. One improvement would be to declare variables as "sensitive," and have the implementation guarantee that they will be wiped. Even better would be a language that always wipes all data that is no longer needed. That would avoid a lot of errors without significantly affecting efficiency.

There are other places where secret data can end up. All data is eventually loaded into a CPU register. Wiping registers is not possible in most programming languages, but on register-starved CPUs like the Pentium, it is very unlikely that any data will survive for any reasonable amount of time.

During a context-switch (when the operating system switches from running one program to running the next program) the values in the registers of the CPU are stored in memory where their values might linger for a long time. As far as we know, there is nothing you can do about this, apart from fixing the operating system to ensure the confidentiality of that data.

9.3.2 Swap File

Most operating systems (including all current Windows versions and all UNIX versions) use a virtual memory system to increase the number of programs that can be run in parallel. While a program is running, not all of its data is kept in memory. Some is stored in a swap file. When the program tries to access data that is not in memory, the program is interrupted. The

virtual memory system reads the required data from the swap file into a piece of memory, and the program is allowed to continue. What's more, when the virtual memory system decides that it needs more free memory, it will take an arbitrary piece of memory from a program and write it to the swap file.

Of course, most virtual memory systems do not make any serious attempt to keep the data secret, or to encrypt it before it is written to the disk. Most software is designed for a cooperative environment, not the adversarial environment that cryptographers work in. So our problem is the following: the virtual memory system could just take some of the memory of our program and write it to the swap file on disk. The program never gets told, and does not notice. Suppose this happens to the memory in which the keys are stored. If the computer crashes—or is switched off—the data remains on the disk. Most operating systems leave the data on disk even when you shut them down properly. Typically there is no mechanism to wipe the swap file, so the data could linger indefinitely on disk. Who knows who will have access to this swap file in future? We really cannot afford the risk of having our secrets written to the swap file.[1]

So how do we stop the virtual memory system from writing our data to disk? On some operating systems there are system calls that you can use to inform the virtual memory system that specified parts of memory are not to be swapped out. Rarely do we find an operating system that supports a secure swap system where the swapped-out data is cryptographically protected. If neither of these options is available, you are out of luck. Complain loudly about the operating system, and do the best you can.

Assuming you can lock the memory and prevent it from being swapped out, which memory should be locked? All the memory that can ever hold secrets, of course. This brings up a secondary problem. Many programming environments make it very hard to know where exactly your data is being stored. Objects are often allocated on a heap, data can be statically allocated, and many local variables end up on the stack. Figuring out the details is complicated and very error-prone. Probably the best solution is to simply lock all the memory of your application. Even that is not quite as easy as it

[1] In fact, we should never write secrets to any permanent media without encrypting them, but that is an issue we will discuss later.

sounds, because you could lose a number of operating system services such as the automatically allocated stack. And locking all the memory makes the virtual memory system ineffective.

It shouldn't be this difficult. The proper solution is, of course, to make a virtual memory system that protects the confidentiality of the data. This is an operating system change, and beyond our control. Even if the next version of your operating system were to have this feature, you should carefully check that the virtual memory system does a good job of keeping secrets.

9.3.3 Caches

Modern computers don't just have a single type of memory. They have a hierarchy of memories. At the bottom is the main memory—often hundreds of megabytes large. But because the main memory is relatively slow, there is also a cache. This is a smaller but faster memory. The cache keeps a copy of the most recently used data from the main memory. If the CPU wants to access the data, it first checks the cache. If the data is in the cache, the CPU gets the data relatively quickly. If the data is not in the cache, it is read (relatively slowly) from main memory, and a copy is stored in the cache for future use. To make room in the cache, a copy of some other piece of data is thrown away.

This is important because caches keep copies of data, including copies of our secret data. The problem is that when we try to wipe our secrets, this wiping might not take place properly. In some systems, the modifications are only written to the cache and not to the main memory. The data will eventually be written to main memory, but only when the cache needs more room to store other data. We don't know all the details of these systems, and they change with every CPU. There is no way to know if there is some interaction between the memory allocation unit and the cache system that might result in some wipe operations escaping the write-to-main-memory part when the memory is deallocated before the cache is flushed. Manufacturers never specify how to wipe data in a guaranteed manner. At least, we have never seen any specifications like that, and as long as it is not specified, we can't trust it.

A secondary danger of caches is that under some circumstances a cache learns that a particular memory location has been modified, perhaps by the other CPU in a multi-CPU system. The cache then marks the data it has for that location as "invalid," but typically the actual data is not wiped. Again, there might exist a copy of our secrets that has not been wiped.

There is very little you can do about this. It is not a great danger, because in most systems only the OS code can access the cache mechanisms directly. And we have to trust the operating system anyway, so we could trust it with this as well. We are nevertheless concerned about these designs, because they clearly do not provide the functionality that is required to implement security systems properly.

9.3.4 Data Retention by Memory

Something that surprises many people is that simply overwriting data in memory does not delete the data. The details depend to some extent on the exact type of memory involved, but basically if you store data in a memory location, that location slowly starts to "learn" the data. When you overwrite or switch off the computer, the old value is not completely lost. Depending on the circumstances, just powering the memory off and back on again can recover some or all of the old data. Other memories can "remember" old data if you access them using (often undocumented) test modes [39].

Several mechanisms cause this phenomenon. If the same data is stored for a time in the same location in SRAM (Static RAM), then this data becomes the preferred power-up state of that memory. A friend of ours encountered this problem with his home-built computer long ago [9]. He wrote a BIOS that used a magic value in a particular memory location to determine whether a reset was a cold reboot or a warm reboot.[2] After a while the machine refused to boot after power-up because the memory had learned the magic value, and the boot process therefore treated every reset as a warm reboot. As this did not initialize the proper variables, the boot

[2]In those days home-built machines were programmed by entering the binary form of machine language directly. This led to many errors, and the one sure way to recover from a program that crashed was to reset the machine. A cold reboot is one after power-up. A warm reboot is the sort performed when the user presses the reset button. A warm reboot does not reinitialize all the state, and therefore does not wipe the settings the user made.

process failed. The solution in his case was to swap some memory chips around, scrambling the magic value that the SRAM had learned. For us it was a lesson to remember: memory retains more data than you think.

Similar processes happen in DRAM, although they are somewhat more complicated. DRAM works by storing a small charge on a very small capacitor. The insulating material around the capacitor is stressed by the resulting field. The stress results in changes to the material, specifically causing the migration of impurities [39]. An attacker with physical control over the memory can potentially recover this data.

It is arguable whether this is a significant threat, but we think it is important. If your computer is ever compromised (e.g., stolen) you do not want the data that you had and then wiped to be compromised as well. To achieve this goal, we have to make the computer forget information.

We can only give a partial solution, which works if we make some reasonable assumptions about the memory. This solution, which we call a Boojum,[3] works for relatively small amounts of data, such as keys. Let m be the data we want to store. Instead of storing m, we generate a random string R and store both R and $R \oplus m$. These two values are stored in different memory locations, preferably not too close together. The trick is to change R regularly. At regular intervals, say every 100 ms, we generate a new random R', and update the memory to store $R \oplus R'$ and $R \oplus R' \oplus m$. This ensures that each bit of the memory is written with a sequence of random bits. To wipe the memory, you simply write a new m with the value zero, which results in the two storage locations getting the same (random) data.

To read information from this storage you read both halves and XOR them together to get m. Writing is done by XORing the new data with R and storing it in the second location.

Care should be taken that the bits of R and $R \oplus m$ are not adjacent on the RAM chip. Without information about how the RAM chip works, this can be difficult, but most memories store bits in a rectangular matrix of bits, with some address bits selecting the row and other address bits selecting the column. If the two pieces are stored at addresses that differ by 0x5555, then it is highly unlikely that the two will be stored adjacent on the chip.

[3] After Lewis Carroll's *The Hunting of the Snark* [15].

(This assumes that the memory does not use the even-indexed address bits as row number and the odd-indexed address bits as column number, but we have never seen a design like that.) An even better solution might be to choose two random addresses in a very large address space. This makes the probability that the two locations are adjacent very small, independent of the actual chip layouts of the memory.

This is only a partial solution, and a rather cumbersome one at that. It is limited to small amounts of data, as the update function would otherwise be too expensive. But using this solution ensures that there is no physical point on the memory chip that is continually stressed or unstressed depending on the secret data.

There is still no guarantee that the memory will be wiped. If you read the documentation of a memory chip, there are no specifications that prevent the chip from retaining all data ever stored in it. No chip does that, of course, but it shows that we can at most achieve a heuristic security.

We have concentrated on the main memory here. The same solution will work for the cache memory, except that you cannot control the position on the chip where the data will be stored. This solution does not work for the CPU registers, but they are used so often for so much different data that we doubt they will pose a data retention problem. On the other hand, extension registers, such as floating point registers or MMX-style registers, are used far less frequently, so they could pose a problem.

If you have large amounts of data that need to be kept secret, then the solution of storing two copies and XORing new random strings into both copies regularly becomes too expensive. A better solution is to encrypt a large block of data and store the ciphertext in memory that potentially retains information. Only the key needs to be stored in a way that avoids data retention, for example, using a Boojum. For details, see [24].

9.3.5 Access by Others

There's yet another problem with keeping secrets on a computer: other programs on the same machine might access the data. Some operating systems allow different programs to share memory. If the other program can read your secret keys, you have a serious problem. Often the shared

memory has to be set up by both programs, which reduces the risk. In other situations, the shared memory might be set up automatically as a result of loading a shared library.

Debuggers are especially dangerous. Modern operating systems often contain features designed to be used by debuggers. Various Windows versions allow you to attach a debugger to an already running process. The debugger can do many things, including reading the memory. Under UNIX it is sometimes possible to force a core-dump of a program. The core-dump is a file that contains a memory image of the program data, including all of your secrets.

Another danger comes from especially powerful users. Called *superusers*, or *administrators*, these users can access things on the machine that normal users cannot. Under UNIX, for example, the superuser can read any part of the memory.

In general, your program cannot effectively defend itself against these types of attacks. If you are careful you may be able to eliminate some of these problems, but often you'll find yourself limited in what can be achieved. Still, you should consider these issues on the particular platform you are working on.

9.3.6 Data Integrity

In addition to keeping secrets, we should protect the integrity of the data we are storing. We use the MAC to protect the integrity of the data during transit, but if the data can be modified in memory, we still have problems.

In this discussion, we will assume that the hardware is reliable. If the hardware is unreliable, there is very little you can do. If you are unsure about the hardware reliability, perhaps you should spend part of your time and memory simply to verify it, although that is really the operating system's job. One thing we try to do is to make sure the main memory on our machines is ECC (error-correcting code) memory.[4] If there is a single bit failure, then

[4]You have to make sure that all components of the computer support ECC memory. Beware of slightly cheaper memory modules that do not store the extra information but instead recompute it on the fly. This defeats the whole purpose of ECC memory.

the error-correcting code will detect and correct the error. Without ECC memory, any bit error leads to the CPU reading the wrong data.

Why is this important? There is an enormous number of bits in a modern computer. Suppose the engineering is done really well, and each bit has only a 10^{-15} chance of failing in each second. If you have 128 MB of memory, then you have about 10^{12} bits of memory, and you can expect one bit failure every 1000 seconds, or about every 17 minutes. This is an unacceptable error rate to us. The error rate increases with the amount of memory in the machine, so it is even worse if you have 1 GB of memory. Servers typically use ECC memory because they have more memory and run for longer periods of time. We like to have the same stability in all machines.

Of course, this is a hardware issue, and you typically don't get to specify the type of memory on the machine that will run the final application.

Some of the dangers that threaten data confidentiality also endanger the data integrity. Debuggers can sometimes modify your program's memory. Superusers can directly modify memory, too. Again, there is nothing you can do about it, but it is useful to be aware of the situation.

9.3.7 What to Do

Keeping a secret on a modern computer is not as easy as it sounds. There are many ways in which the secret can leak out. To be fully effective, you have to stop all of them. Unfortunately, current operating systems and programming languages do not provide the required support to stop the leakage completely. You have to do the best you can. This involves a lot of work, all of it specific to the environment you work in.

These problems also make it very difficult to create a library with the cryptographic functions in it. Keeping the secrets safe often involves modifications to the main program. And of course, the main program also handles data that should be kept confidential; otherwise, it wouldn't need the cryptography library in the first place. This is the familiar issue of security considerations affecting every part of the system.

9.4 Quality of Code

If you create an implementation for a cryptographic system, you will have to spend a great deal of time on the quality of the code. This book is not about programming, but as quality of code is typically left out of programming books, we will say a few words here.

9.4.1 Simplicity

Complexity is the main enemy of security. Therefore, any security design should strive for simplicity. We are quite ruthless about this, even though this does not make us popular. Eliminate all the options that you can. Get rid of all those baroque features that few people use. Stay away from committee designs, because the committee process always leads to extra features or options in order to achieve compromise. In security, simplicity is king.

A typical example is our secure channel. It has no options. It doesn't allow you to encrypt the data without authenticating it, or to authenticate the data without encrypting it. People always ask for these features, but typically they do not know the consequences of using partial security features. Most users do not understand enough about security to be able to select the correct security options. The best solution is to have no options and make it secure by default. If you absolutely have to, provide a single option: secure or insecure.

Many systems also have multiple cipher suites, where the user (or someone else) can choose which cipher and which authentication function to use. If at all possible, eliminate this complexity. Choose a single mode that is secure enough for all possible applications. The computational difference between the various encryption modes is not that large, and cryptography is rarely the bottleneck for modern computers. Apart from getting rid of the complexity, it also gets rid of the danger that users might configure their application to use weak cipher suites. After all, if choosing an encryption and authentication mode is so difficult that the designer can't do it, what makes you think the user understands enough to make an informed decision?

9.4.2 Modularization

Even after you have eliminated a lot of options and features, the resulting system will still be quite complex. There is one main technique of making the complexity manageable: modularization. You divide the system into separate modules, and design, analyze, and implement each module separately.

You should already be familiar with modularization; in cryptography it becomes even more important to do it right. Earlier we talked about cryptographic primitives as modules. The module interface should be simple and straightforward. It should behave according to the reasonable expectations of a user of the module. Look closely at the interface of your modules. Often there are features or options that exist to solve some other module's problems. If possible, rip them out. Each module should solve its own problems. We have found that when module interfaces start to develop weird features, it is time to redesign the software because they are almost always a result of design deficiencies.

Modularization is so important because it is the only efficient way we have of dealing with complexity. If a particular option is restricted to a single module, it can be analyzed within the context of this module. However, if the option changes the external behavior of one module, it can affect other modules as well. If you have 20 modules, each with a single binary option that changes the module behavior, there are over a million possible configurations. You would have to analyze each of these configurations for security—an impossible task.

We have found that many options are created in the quest for efficiency. This is a well-known problem in software engineering. Many systems contain so-called optimizations that are useless, counterproductive, or insignificant because they do not optimize those parts of the system that form the bottleneck. We have become quite conservative about optimizations. Usually we don't bother with them. We do create a careful design, and try to ensure that work can be done in large "chunks." A typical example is the old IBM PC BIOS. The routine to print a character on the screen took a single character as an argument. This routine spent almost all of its time on overhead, and only a very small fraction on actually putting the character

on the screen. If the interface of the routine had allowed a string as argument, then the entire string could have been printed in only slightly more time than it took to print a single character. The result of this bad design was that all DOS machines had a terribly slow display. This same principle applies to cryptographic designs. Make sure that work can be done in large enough chunks. Then only optimize those parts of your program that you can *measure* as having a significant effect on the performance.

9.4.3 Assertions

Assertions are a good tool to help improve the quality of your code.[5]

When implementing cryptographic code, adopt an attitude of professional paranoia. Each module distrusts the other modules, and always checks parameter validity, enforces calling sequence restrictions, and refuses unsafe operations. Most of the times these are straightforward assertions. If the module specifications state that you have to initialize the object before you use it, then using an object before initialization will result in an assertion error. Assertion failures should always lead to an abort of the program with ample documentation of which assertion failed, and for what reason.

The general rule is: any time you can make a meaningful check on the internal consistency of the system, you should add an assertion. Catch as many errors as you can, both your own and those of other programmers. An error caught by an assertion will not lead to a security breach.

There are some programmers who implement assertion checking in development, but switch it off when they ship the product. Who thought that up? What would you think of a nuclear power station where the operators train with all the safety systems in place, but switch them off when they go to work on the real reactor? Or a parachutist who wears his emergency parachute while training on the ground, but leaves it off when he jumps out of the airplane? Why would anyone ever switch off the assertion checking on production code? That is the only place where you really need it! If an assertion fails in production code, then you have just encountered a programming error. Ignoring the error will most likely result in some kind of

[5]We know that this is starting to sound like a programming lesson, but we find we have to repeat these things time and time again to the programmers we work with.

wrong answer, because at least one assumption the code makes is wrong. Generating wrong answers is probably the worst thing a program can do. It is much better to at least inform the user that a programming error has occurred, so that he does not trust the erroneous results of the program. Leave all your error checking on.

9.4.4 Buffer Overflows

It is an embarrassment for the IT industry that we need a section with this title. Buffer overflow problems have been known for 40 years. Perfectly good solutions to avoid them have been available for the same amount of time. Some of the earliest higher-level programming languages, such as Algol 60, completely solved the problem by introducing mandatory array bounds checking. Even so, buffer overflows cause about half of the security problems on the Internet. And still people refuse to banish them by using better tools. We consider this criminal negligence. It is comparable to a car manufacturer making the gas tank out of waxed paper. Sure, if everything goes right, there's no problem, but we'd throw the CEO into jail all the same. For some reason, large parts of our IT industry act as if they were not responsible for the consequences of their actions. (Maybe because our lawmakers let them get away with disclaimers that would be unconscionable in any other industry.) With this prevailing attitude, we sometimes wonder whether it's worth attempting something as advanced as cryptography at all.

But those are all things we cannot change. We can give you advice on how to write good cryptographic code. Avoid any programming language that allows buffer overflows. Specifically: don't use C or C++. And don't ever switch off the array bounds checking of whichever language you use instead. It is such a simple rule, and it will probably solves half of all your security bugs.

9.4.5 Testing

Extensive testing is always part of any good development process. Testing can help find bugs in programs, but it is useless to find security holes. Never

confuse testing with security analysis. The two are complementary, but different.

There are two types of tests that should be implemented. The first is a generic set of tests developed from the module's functional specifications. Ideally, one programmer implements the module and a second programmer implements the tests. Both work from the functional specifications. Any misunderstanding between the two is a clear indication that the specifications have to be clarified. The generic tests should attempt to cover the entire operational spectrum of the module. For some modules, this is simple; for others, the test program will have to simulate an entire environment. In much of our own code the test code is about as big as the operational code, and we have not found a way of significantly improving that.

A second set of tests are developed by the programmer of the module itself. These are designed to test any implementation limits. For example, if a module uses a 4 KB buffer internally, then extra tests of the boundary conditions at the start and end of the buffer will help to catch any buffer-management errors. Sometimes it requires knowledge of the internals of a module to devise specific tests.

We frequently write test sequences that are driven by a random generator. We will discuss PRNGs extensively in chapter 10. Using a PRNG makes it very easy to run a very large number of tests. If we save the seed we used for the PRNG we can repeat the same test sequence, which is very useful for testing and debugging. Details depend on the module in question.

Finally, we have found it useful to have some "quick test" code that can run every time the program starts up. In one of Niels's recent projects, he had to implement AES. The initialization code runs AES on a few test cases and checks the output against the known correct answers. If the AES code is ever destabilized during the further development of the application, this quick test is very likely to detect the problem.

9.5 Side-Channel Attacks

There is a whole class of attacks that we call side-channel attacks [49]. These are possible when an attacker has an additional channel of information about

the system. For example, an attacker could make detailed measurements of the time it takes to encrypt a message. If the cryptography is embedded in a smart card, then the attacker can measure how much current the card draws over time. Magnetic fields, RF emissions, power consumption, timing, and interference on other data channels can all be used for side-channel attacks.

Not surprisingly, side-channel attacks can be remarkably successful against systems that are not designed with these attacks in mind. Power analysis of smart cards is extremely successful [57].

It is very difficult, if not impossible, to protect against all forms of side-channel attacks, but there are some simple precautions you can take. Years ago, when Niels worked on implementing cryptographic systems in smart cards, one of the design rules was that the sequence of instructions that the CPU executed could only depend on information already available to the attacker. This stops timing attacks, and makes power analysis attacks more complicated because the sequence of instructions that is being executed can no longer leak any information. It is not a full solution, and modern power analysis techniques would have no problem breaking the smart cards that were fielded in those days. Still, what we did was about the best that could be done with the smart cards of the day. Resistance against side-channel attacks will always come from a combination of countermeasures—some of them in the software that implements the cryptographic system, and some of them in the actual hardware.

Preventing side-channel attacks is a rat race. You try to protect yourself against the known side channels, and then a smart person somewhere discovers a new side channel, so then you have to go back and take that one into account as well. In real life, the situation is not that bad, because most side-channel attacks are difficult to perform. Side channels are a real danger to smart cards because the card is under full control of the adversary, but only a few types of side channels are practical against most other computers. In practice, the most important side channels are timing and RF emissions. (Smart cards are particularly vulnerable to measuring the power consumption.)

9.6 Conclusion

We hope this chapter has made it clear that security does not start or stop with the cryptographic design. All aspects of the system have to do their part to achieve security. This is why security people are universally hated; they stick their noses into absolutely everything, go around telling people how to do their work, and then forbid a lot of very useful features just because they are insecure.

Implementing cryptographic systems is an art in itself. The most important aspect is the quality of the code. Low-quality code is the most common cause of real-world attacks, and it is rather easy to avoid. In our experience, writing high-quality code takes about as long as writing low-quality code, if you count the time from start to finished product, rather than from start to first buggy version. Be fanatical about the quality of your code. It can be done, and it needs to be done, so go do it!

Ideally we would redesign our entire environment, including our programming language and operating system, with security as a primary goal. We'd love to work on this project, so contact us if you are willing to spend a few million dollars on a computer you can *really* trust.

Chapter 14

Introduction to Cryptographic Protocols

Cryptographic protocols consist of an exchange of messages between participants. We've already seen a simple cryptographic protocol in chapter 12.

Protocols are probably the most difficult part of cryptography. The main problem is that as a designer or implementer, you are not in control. Up to now we have been designing a system and have had control over the behavior of various parts. Once you start communicating with other parties, you have no control over their behavior. The other party has a different set of interests than you do, and he could deviate from the rules to try to get an advantage. When working on protocols, you must assume that you are dealing with the enemy.

14.1 Roles

Protocols are typically described as being executed by Alice and Bob, or between a customer and a merchant. Names like "Alice," "Bob," "customer," and "merchant" are not really meant to identify a particular individual or organization. They identify a role within the protocol. If Mr. Smith wants to communicate with Mr. Jones, he might run a key agreement protocol. Mr. Smith could take the role of Alice, and Mr. Jones the role of Bob. The

next day the roles might be reversed. It is important to keep in mind that a single entity can take on any of the roles.[1] This is especially important to remember when you analyze the protocol for security. We've already seen the man-in-the-middle attack on the DH protocol. In that attack, Eve takes on the roles of both Alice and of Bob. (Of course, Eve is just another role, too.)

14.2 Trust

Trust is the ultimate basis for all dealings that we have with other people. If you don't trust anybody with anything at all, why bother interacting with them? For example, buying a candy bar requires a basic level of trust. The customer has to trust the merchant to provide the candy and give proper change. The merchant has to trust the customer to pay. Both have recourse if the other party misbehaves. Shoplifters are prosecuted. Cheating merchants risk bad publicity, lawsuits, and getting punched in the nose.

There are several sources of trust:

Ethics Ethics has a large influence in our society. Although very few, if any, people behave ethically all the time, most people behave ethically most of the time. Attackers are few. Most people pay for their purchases, even when it would be laughably easy to steal them.

Reputation Having a "good name" is very important in our society. People and companies want to protect their reputation. Often the threat of bad publicity gives them an incentive to behave properly.

Law In civilized societies there is a legal infrastructure that supports lawsuits and prosecution of people who misbehave. This gives people an incentive to behave properly.

Physical Threat Another incentive to behave properly is the fear of harm if you cheat and are caught. This is one of the sources of trust for drugs deals and other illegal trades. The threat can be physical violence, or other actions.

[1] In protocols with three or more participants, it is even possible for a single person to take on more than one role at the same time.

MAD A cold war term: Mutually Assured Destruction. In milder forms, it is the threat to do harm to both yourself and the other party. If you cheat your friend, she might break off the friendship, doing you both harm. Sometimes you see two companies in a MAD situation, especially when they file patent infringement lawsuits against each other.

All of these sources are mechanisms whereby a party has an incentive not to cheat. The other party knows this incentive, and therefore feels he can trust his opponent to some extent. This is why these incentives all fail when you deal with completely irrational people: you can't trust them to act in their own best interest, which undermines all these mechanisms.

It is hard to develop trust over the Internet. Suppose Alice lives abroad and connects to the ACME Web site. ACME has almost no reason to trust Alice; of the mechanisms of trust we mentioned, only ethics remains. Legal recourse against private individuals abroad is almost impossible, and certainly prohibitively expensive. You can't effectively harm their reputation, threaten them, or even threaten them with MAD.

There is still a basis of trust between Alice and ACME, because ACME has a reputation to protect. This is important to remember when you design a protocol for e-commerce. If there are any failure modes (and there always are), the failure should be to ACME's advantage, because ACME has an incentive to settle the matter properly by manual intervention.[2] If the failure is to Alice's advantage, the issue is less likely to be settled properly. Furthermore, ACME will be vulnerable to attackers who try to induce the failure mode and then profit by it.

Trust is not a black-and-white issue. It is not that you either trust someone or you don't trust him. You trust different people to different degrees. You might trust a friend with $100 but not with your lottery ticket that just won a $5,000,000 prize. We trust the bank to keep our money safe, but we get receipts and copies of canceled checks because we don't fully trust their administration. The question "Do you trust him?" in incomplete. It should be "Do you trust him with X?"

[2]Almost all telephone, mail, and electronic commerce to individuals follows this rule by having the customer pay for the order before it is shipped.

14.2.1 Risk

Trust is fundamental to business, but it is usually expressed as risk rather than trust. Risk can be seen as the converse of trust. Risks are evaluated, compared, and traded in many forms.

When working on cryptographic protocols it is easier to talk in terms of trust than in terms of risks. But a lack of trust is simply a risk, and that can sometimes be handled by standard risk-management techniques such as insurance. We talk about trust when we design protocols. Always keep in mind that business people think and talk in terms of risks. You'll have to convert between the two perspectives if you want to be able to talk to them.

14.3 Incentive

The incentive structure is another fundamental component of any analysis of a protocol. What are the goals of the different participants? What would they like to achieve? Even in real life, analyzing the incentive structure gives insightful conclusions.

Several times every week we get press reports that announce things like, "New research has shown that ..." Our first reaction is always to ask: who paid for the research? Research whose results are advantageous to the party who paid for it is always suspect. Several factors are at play here. First, the researchers know what their customer wants to hear, and know they can get repeat contracts if they produce "good" results. This introduces a bias. Secondly, the sponsor of the research is not going to publish any negative reports. Publishing only the positive reports introduces another bias. Tobacco companies published "scientific" reports that nicotine was not addictive. Microsoft pays for research that "proves" that open source software is bad in some way. Don't ever trust research that supports the company that paid for it.

The authors are personally quite familiar with these pressures. During our many years as consultants, we performed many security evaluations for paying customers. We were often harsh—the average product we evaluated was quite bad—and we rarely wrote positive evaluations for components. That

didn't always make us popular with our customers. One of them even called Bruce and said: "Stop your work and send me your bill. I've found someone who is cheaper and who writes better reports." Guess which meaning of 'better' was intended here? The only reason we could be unbiased was that we had enough work. If work is scarce and you have to put food on the table, the temptation is great to bite your tongue and say whatever your client wants to hear.

We see exactly the same problem in other areas. As we write this book, the press is filled with stories about the accounting and banking industries. Analysts and accountants were writing reports favorable for their clients rather than unbiased evaluations. We blame the incentive structure that gave these people a reason to bias their reports. Looking at the incentives is quite instructive, and something we've both done for years. With a bit of practice it is surprisingly easy, and it yields valuable insights. And yes, it makes you more cynical of people's motives.

If you pay your management in stock options, you give them the following incentive structure: increase the share price over the next three years and make a fortune; decrease the share price, and get a golden handshake. It is a "Heads I win a lot, tails I win a little" incentive, so guess what some managers do? They go for a high-risk short-term strategy. If they get the opportunity to double the amount they gamble they will always take it, because they will only collect the winnings and never pay the loss. If they can inflate the share price for a few years with bookkeeping tricks they will, because they can cash out before they are found out. Some of the gambles fail, but others pay the bills.

A similar thing happened with the savings and loans industry in the United States in the 1980s. The federal government liberalized the rules, allowing S&Ls to invest their money more freely. At the same time, the government guaranteed the deposits. Now look at the incentive structure. If the investments pay off, the S&L makes a profit, and no doubt management gets a nice bonus. If the investments lose money, the federal government pays off the depositors. Not surprisingly, a bunch of S&Ls lost a lot of money on high-risk investments—and the federal government picked up the bill.

Fixing the incentive structure is often relatively easy. For example, instead of the company itself paying for the audit, the stock exchange can arrange

and pay for the audit of the books. Give the auditors a significant bonus for every error they find and you'll get a much more accurate report.

Examples of undesirable incentive structures abound. Divorce lawyers have an incentive to make the divorce very acrimonious, as they are paid for every hour spent fighting over the estate. It is a safe bet that they will advise you to settle as soon as the legal fees exceed the value of the estate.

In American society, lawsuits are common. If an accident happens, every participant has a great incentive to hide, deny, or otherwise avoid the blame. Strict liability laws and huge damage awards might seem good for society at first, but it greatly hinders our ability to figure out why the accident happened, and how we can avoid it in future. Liability laws that are supposed to protect consumers make it all but impossible for a company like Firestone to admit that there is a problem with their product so we can all learn how to build better tires.

Cryptographic protocols interact in two ways with incentive structures. First of all, they rely on incentive structures. Some electronic payment protocols do not stop the merchant from cheating the customer, but provide the customer with proof of the cheating. This works because the merchant has an incentive not to have people out there with proofs that they were cheated. The proof could be used either in a court case or just to damage the reputation of the merchant.

Cryptographic protocols also change the incentive structure. They make certain things impossible, removing them from the incentive structure. They can also open up new possibilities and new incentives. Once you have online banking, you create an incentive for a thief to break into your computer and steal your money.

At first, incentives look like they are mostly materialistic, but that is only part of it. Many people have nonmaterialistic motives. Most computer break-ins are not done for any material gain, but just for fun, status, or bragging rights. In personal relationships, the most fundamental incentives have little to do with money. Keep an open mind, and try to understand what drives people. Then create your protocols accordingly.

14.4 Trust in Cryptographic Protocols

The function of cryptographic protocols is to minimize the amount of trust required. Let's repeat that. The function of cryptographic protocols is to minimize the amount of trust required. This means minimizing both the number of people who need to trust each other and the amount of trust they need to have.

One powerful tool for designing cryptographic protocols is the paranoia model. When Alice takes part in a protocol, she assumes that all other participants are conspiring together to cheat her. This is really the ultimate conspiracy theory. Of course, each of the other participants is making the same assumption. This is the default model in which all cryptographic protocols are designed.

Any deviations from this default model must be explicitly documented. It is surprising how often this step is overlooked. We sometimes see protocols used in situations where the required trust is not present. For example, most secure Web sites use the SSL protocol. The SSL protocol requires trusted certificates. Browsers will often accept any certificate, and a certificate is easy to get. The result is that the user is communicating securely with a Web site, but she doesn't know which Web site she is communicating with. Numerous scams against PayPal users have exploited this vulnerability.

It is very tempting not to document the trust that is required for a particular protocol, as it is often "obvious." That might be true to the designer of the protocol, but like any module in the system, the protocol should have a clearly specified interface, for all the usual reasons.

From a business point of view, the documented trust requirements also list the risks. Each point of required trust implies a risk that has to be dealt with.

14.5 Messages and Steps

A typical protocol description consists of a number of messages that are sent between the participants of the protocol and a description of the computations that each participant has to do.

Almost all protocol descriptions are done at a very high level. Most of the details are not described. This allows you to focus on the core functionality of the protocol, but it creates a great danger. Without careful specifications of all the actions that each participant should take, it is extremely difficult to create a safe implementation of the protocol.

Sometimes you see protocols specified with all the minor details and checks. Such specifications are often so complicated that nobody fully understands them. This might help an implementer, but anything that is too complicated cannot be secure.

The solution is, as always, a modularization. With cryptographic protocols, as with communication protocols, we can split the required functionality into several protocol layers. Each layer works on top of the previous layer. All the layers are important, but most of the layers are the same for all protocols. Only the topmost layer is highly variable, and that is the one you always find documented.

14.5.1 The Transport Layer

Network specialists must forgive us for reusing one of their terms here. For us cryptographers, the transport layer is the underlying communication system that allows parties to communicate. This consists of sending strings of bytes from one participant to another. How this is done is irrelevant for our purposes. What we as cryptographers care about is that we can send a string of bytes from one participant to the other. You can use UDP packets, a TCP data stream, e-mail, or any other method. In many cases, the transport layer needs some additional encoding. For example, if a program executes multiple protocols simultaneously, the transport layer must deliver the message to the right protocol execution. This might require an extra destination field of some sort. When using TCP, the length of the message needs to be included to provide message-oriented services over the stream-oriented TCP protocol.

To be quite clear, we expect that the transport layer transmits arbitrary strings of bytes. Any byte value could occur in the message. The length of the string is variable. The string received should of course be identical to the string that was sent; deleting trailing zero bytes, or any other modification, is not allowed.

Some transport layers include things like magic constants to provide an early detection of errors or to check the synchronization of the TCP stream. If the magic constant is not correct on a received message, the rest of the message should be discarded.

There is one important special case. Sometimes we run a cryptographic protocol over a cryptographically secured channel like the one we designed in chapter 8. In cases like that, the transport layer also provides confidentiality, authentication, and replay protection. That makes the protocol much easier to design, because there are far fewer types of attacks to worry about.

14.5.2 Protocol and Message Identity

The next layer up provides protocol and message identifiers. When you receive a message, you want to know which protocol it belongs to and which message within that protocol it is.

The protocol identifier typically contains two parts. The first part is the version information which provides room for future upgrades. The second part identifies which particular cryptographic protocol the message belongs to. In an electronic payment system there might be protocols for withdrawal, payment, deposit, refund, etc. The protocol identifier avoids confusion among messages of different protocols.

The message identifier indicates which of the messages of the protocol in question this is. If there are four messages in a protocol you don't want there to be any confusion about which message is which.

Why do we include so much identifying information? Can't an attacker forge all of this? Of course he can. This layer doesn't provide any protection against active forgery; rather, it detects accidental errors. It is important to have good detection of accidental errors. Suppose you are responsible for maintaining a system, and you suddenly get a large number of error messages. Differentiating between active attacks and accidental errors such as configuration and version problems is a valuable service.

Protocol and message identifiers also make the message more self-contained, which makes much of the maintenance and debugging easier. Cars and airplanes are designed to be easy to maintain. Software is even more complex—all the more reason why it should be designed for ease of maintenance.

Probably the most important reason to include message identifying information has to do with the Horton Principle. When we use authentication (or a digital signature) in a protocol, we typically authenticate several messages and data fields. By including the message identifying information we never run the risk that a message will be interpreted in the wrong context.

14.5.3 Message Encoding and Parsing

The next layer is the encoding layer. Each data element of the message has to be converted to a sequence of bytes. This is a standard programming problem and we won't go into too much detail about that here.

One very important point is the parsing. The receiver must be able to parse the message, which looks like a sequence of bytes, back into its constituent fields. This parsing must not depend on contextual information.

A fixed-length field that is the same in all versions of the protocol is easy to parse. You know exactly how long it is. The problems begin when the size or meaning of a field depends on some context information, such as earlier messages in the protocol. This is an invitation to trouble.

Many messages in cryptographic protocols end up being signed or otherwise authenticated. The authentication function authenticates a string of bytes, and usually it is the simplest solution to authenticate the message at the level of the transport layer. If the interpretation of a message depends on some contextual information, the signature or authentication is ambiguous. We've broken several protocols due to this type of failure.

A good way to encode fields is to use Tag-Length-Value or TLV encoding. Each field is encoded as three data elements. The tag identifies the field in question. The length is the length of the value encoding, and the value is the actual data to be encoded. The best-known TLV encoding is ASN.1 [43], but it is so incredibly complex and badly specified that we shy away from it. A subset of ASN.1 could be very useful.

A newer alternative is XML. Forget the XML hype; we're only using XML as a data encoding system. As long as you use a fixed Document Template Definition (DTD), the parsing is not context-dependent, and you won't have any problems.

14.5.4 Protocol Execution States

In many implementations, it is possible for a single computer to take part in several protocol executions at the same time. To keep track of all the protocols requires some form of protocol execution state. The state contains all the information necessary to complete the protocol.

Implementing protocols requires some kind of event-driven programming, as the execution has to wait for external messages to arrive before it can proceed. This can be implemented in various ways, such as using one thread or process per protocol execution, or using some kind of event dispatch system.

Given an infrastructure for event-driven programming, implementing a protocol is relatively straightforward. The protocol state contains a state machine that indicates the type of message expected next. As a general rule, no other type of message is acceptable. If the expected type of message arrives, then it is parsed and processed according to the rules.

14.5.5 Errors

Protocols always contain a multitude of checks. These include verifying the protocol type and message type, checking that it is the expected type of message for the protocol execution state, parsing the message, and performing the cryptographic verifications specified. If any of these checks fail, we have encountered an error.

Errors need very careful handling, as they are a potential avenue of attack. The safest procedure is not to send any reply to an error and immediately delete the protocol state. This minimizes the amount of information the attacker can get about the protocol. Unfortunately, it makes for an unfriendly system, as there is no indication of the error.

To make systems usable, you often need to add error messages of some sort. If you can get away with it, don't send an error message to the other parties in the protocol. Log an error message on a secure log so that the system administrator can diagnose the problem. If you *must* send an error message, make it as uninformative as possible. A simple "There was an error" message is often sufficient.

One dangerous interaction is between errors and timing attacks. The attacker Eve can send a bogus message to Alice and wait for her error reply. The time it takes Alice to detect the error and send the reply often contains detailed information about what was wrong and exactly where it went wrong.

Here is a good illustration of the dangers of these interactions. Years ago, Niels worked with a commercially available smart card system. One of the features was a PIN code that was needed to enable the card. The four-digit PIN code was sent to the card, and the card responded with a message indicating whether the card was now enabled or not. Had this been implemented well, it would have taken 10,000 tries to exhaust all the possible PIN codes. The smart card allowed five failed PIN attempts before it locked up, after which it would require special unlocking by other means. The idea was that an attacker who didn't know the PIN code could make five attempts to guess the four-digit PIN code, which gave her a 1 in 2000 probability of guessing the PIN code before the card locked up.

The design was good, and similar designs are widely used today. A 1 in 2000 chance is good enough for many applications. But unfortunately the programmer of that particular smart card system was a bit careless. To verify the four-digit PIN code, the program first checked the first digit, then the second, etc. The card reported the PIN code failure as soon as it detected that one of the digits was wrong. The weakness was that the time it took the smart card to send the "wrong PIN" error depended on how many of the digits of the PIN were correct. A smart attacker could measure this time and learn a lot of information. In particular, the attacker could find out at which position the first wrong digit was. Armed with that knowledge, it would take the attacker only 40 attempts to exhaustively search the PIN space. (After 10 attempts the first digit would have to be right, after another 10 attempts the second, etc.) After five tries her chances of finding the correct PIN code rose to 1 in 143. That is much better than the 1 in 2000 chance she should have had. If she got 20 tries, her chances rose to 60%, which is a lot more than the 0.2% she should have had.

Even worse, there are certain situations where having 20 or 40 tries is not infeasible. Smart cards that lock up after a number of failed PIN tries always reset the counter once the correct PIN has been used, so that the user gets another five tries to type the correct PIN the next time. Suppose

your roommate has a smart card like the one described above. If you can get at your roommate's smart card, you can run one or two tries before putting the smart card back. Wait for him to use the card for real somewhere, using the correct PIN and resetting the failed-PIN attempt counter in the smart card. Now you can do one or two more tries. Soon you'll have the whole PIN code because it takes at most 40 tries to find it.

Error handling is too complex to give you a simple set of rules. This is something we as a community do not know enough about yet. At the moment, the best we can say is: be very careful.

14.5.6 Replay and Retries

A replay attack occurs when the attacker records a message and then later sends that same message again. Message replays have to be protected against. They can be a bit tricky to detect, as the message looks exactly like a proper one. After all, it *is* a proper one.

Closely related to the replay attack is the retry. Suppose Alice is performing a protocol with Bob, and she doesn't get a response. There could be many reasons for this, but one common one is that Bob didn't receive Alice's last message and is still waiting for it. This happens in real life all the time, and we solve this by sending another letter or e-mail, or repeating our last remark. In automated systems this is called a retry. Alice retries her last message to Bob and again waits for a reply.

So, Bob can receive replays of messages sent by the attacker and retries sent by Alice. Somehow Bob has to deal properly with them and ensure correct behavior without introducing a security weakness.

Sending retries is relatively simple. Each participant has a protocol execution state of some form. All you need to do is keep a timer and send the last message again if you do not receive an answer within a reasonable time. The exact time limit depends on the underlying communication infrastructure. If you use UDP packets (a protocol that uses IP packets directly), there is a reasonable probability that the message will get lost, and you want a short retry time, on the order of a few seconds. If you send your messages over TCP, then TCP retries any data that was not received properly using its own time-outs. There is little reason to do a retry at the cryptographic

protocol level, and most systems that use TCP do not do this. Nevertheless, for the rest of this discussion we are going to assume that retries are being used, as the general techniques of handling received retries also work even if you never send them.

When you receive a message you have to figure out what to do with it. We assume that each message is recognizable so that you know which message in the protocol it is supposed to be. If it is the message that you expect, then there is nothing out of the ordinary and you just follow the protocol rules. Suppose it is a message from the "future" of the protocol, i.e., one that you only expect at a later point in time. This is easy; ignore it. Don't change your state, don't send a reply, just drop it and do nothing. It is probably part of an attack. Even in weird protocols where it could be part of a sequence of errors induced by lost messages, ignoring a message has the same effect as the message being lost in transit. As the protocol is supposed to recover from lost messages, ignoring a message is always a safe solution.

That leaves the case of "old" messages; messages that you already processed in the protocol you are running. There are three situations in which this could occur. In the first one, the message you receive is the previous one you responded to, and it is identical to the message you responded to. In this case, the message is probably a retry, so you send exactly the same reply you sent the first time. Note that the reply should be the same. Don't recompute the reply with a different random value, and don't just assume that the message you get is identical to the first one you replied to. You have to check.

The second case is when you receive a message that has the same message identification as the message you last responded to, but the message contents are different. For example, suppose in the DH protocol Bob receives the first message from Alice, and then later receives another message that claims to be the first message in the protocol, but which contains different data. This situation is indicative of an attack. No retry would ever create this situation, as the resent message is never different from the first try. Either the message you just received is bogus, or the earlier one you responded to is bogus. The safe choice is to treat this as a protocol error, with all the consequences we discussed. (Ignoring the message you just received is safe, but it means that fewer forms of active attacks are detected as such. This has a detrimental effect on the detection and response parts of the security system.)

The third case is when you receive a message that is even older than the previous message you responded to. There is not much you can do with this. If you still know the original message you received at that phase in the protocol, you could check if it is identical. If it is, ignore it. If it is different, you have detected an attack and should treat it as a protocol error. Many implementations do not store all the messages that were received in a protocol execution, which makes it impossible to know whether the message you receive now is identical or not to the one originally processed. The safe option is to ignore these messages. You'd be surprised how often this actually happens. Sometimes messages get delayed for a long time. Suppose Alice sends a message that is delayed. After a few seconds, she sends a retry that does arrive, and both Alice and Bob continue with the protocol. Half a minute later, Bob receives the original message. This is a situation in which Bob receives a copy of—in protocol terms—a very old message.

Things get more complicated if you have a protocol in which there are more than two participants. These exist, but are beyond the scope of this book. If you ever work on a multiparty protocol, think carefully about replay and retries.

One final comment: it is impossible to know whether the last message of a protocol arrived or not. If Alice sends the last message to Bob, then she will never get a confirmation that it arrived. If the communication link is broken and Bob never receives the last message, then Bob will retry the previous message but that will not reach Alice either. This is indistinguishable to Alice from the normal end of the protocol. You could add an acknowledgment from Bob to Alice to the end of the protocol, but then this acknowledgment becomes the new last message and the same problem repeats. Cryptographic protocols have to be designed in a way that this ambiguity does not lead to insecure behavior.

Chapter 22

Storing Secrets

We discussed the problem of storing transient secrets, such as session keys, back in section 9.3. But how do we store long-term secrets, such as passwords and private keys? We have two opposing requirements. First of all, the secret should be kept secret. Second, the risk of losing the secret altogether (i.e., not being able to find the secret again) should be minimal.

22.1 Disk

One of the obvious ideas is to store the secret on the hard drive in the computer or on some other permanent storage medium. This works, but only if the computer is kept secure. If Alice stores her keys (without encryption) on her PC, then anyone who uses her PC can use her keys. Most PCs are used by other people, at least occasionally. Alice won't mind letting someone else use her PC, but she certainly doesn't want to grant access to her bank account at the same time! Another problem is that Alice probably uses several computers. If her keys are stored on her PC at home, she cannot use them while at work or while traveling. And should she store her keys on her desktop machine at home or on her laptop? We really don't want her to copy the keys to multiple places; that only weakens the system further.

A better solution would be for Alice to store her keys on her PDA. A PDA is less likely to be lent out, and it is something that she takes with her

everywhere she goes. (An alternative for the PDA would be her cell phone or wristwatch, but to use them requires updates to the infrastructure that are outside our scope.)

You'd think that security would improve if we encrypt the secrets. Sure, but with what? We need a master key to encrypt the secrets with, and that master key needs to be stored somewhere. Storing it next to the encrypted secrets doesn't give you any advantage. This *is* a good technique to reduce the number and size of secrets though, and it is widely used in combination with other techniques. For example, a private RSA key is several thousand bits long, but by encrypting it with a symmetric key we can reduce the size of the required secure storage by a significant factor.

22.2 Human Memory

The next idea is to store the key in Alice's brain. We get her to memorize a password, and encrypt all the other key material with this password. The encrypted key material can be stored anywhere—maybe on a disk, but it could also be stored on a Web server where Alice can download it to whatever computer she is using at the moment.

Humans are notoriously bad at memorizing passwords. If you choose very simple passwords, you don't get any security. There are simply not enough simple passwords for them to be really secret: the attacker can just try them all. Using your mother's maiden name doesn't work very well; her name is quite often public knowledge—and even if it isn't, there are probably only a few hundred thousand surnames that the attacker has to try to find the right one.

A good password must be unpredictable. In other words, it must contain a lot of entropy. Normal words, such as passwords, do not contain much entropy. There are about half a million English words—and that is counting all the very long and obscure words in an unabridged dictionary—so a single word as password provides at most 19 bits of entropy. Estimates of the amount of entropy per character in English text vary a bit, but are in the neighborhood of 1.5–2 bits per letter.

We've been using 256-bit secret keys throughout our systems to achieve 128 bits of security. In most places, using a 256-bit key has very little additional cost. However, in this situation the user has to memorize the password (or key), and the additional cost of larger keys is high. Trying to use passwords with 256 bits of entropy is too cumbersome; therefore, we will restrict ourselves to passwords with only 128 bits of entropy.[1]

Using the optimistic estimate of 2 bits per character, we'd need a password of 64 characters to get 128 bits of entropy. That is unacceptable. Users will simply refuse to use such long passwords.

What if we compromise and accept 64 bits of security? That is already very marginal. At 2 bits of entropy per character, we need the password to be at least 32 characters long. Even that is too long for users to deal with. Don't forget, most real-world passwords are only 6–8 letters long.

You could try to use assigned passwords, but have you ever tried to use a system where you are told that your password is "719327582742994690 5186"? Or how about "aoekjk3ncmakwe"? Humans simply can't remember such passwords, so this solution doesn't work. (In practice users will write the password down, but we'll discuss that in the next section.)

A much better solution is to use a passphrase. This is similar to a password. In fact, they are so similar that we consider them equivalent. The difference is merely one of emphasis: a passphrase is much longer than a password.

Perhaps Alice could use the passphrase, "Pink curtains meander across the ocean." That is nonsensical, but fairly easy to remember. It is also 38 characters long, so it probably contains about 57–76 bits of entropy. If Alice expands it to "Pink dotty curtains meander over seas of Xmas wishes," she gets 52 characters for a very reasonable key of 78–104 bits of entropy. Given a keyboard, Alice can type this passphrase in a few seconds, which is certainly much faster than she can type a string of random digits. We rely on the fact that a passphrase is much easier to memorize than random data. Many mnemonic techniques are based on the idea of converting random data to things much closer to our passphrases.

[1] For the mathematicians: passwords chosen from a probability distribution with 128 bits of entropy.

Some users don't like to do a lot of typing, so they choose their passphrases slightly differently. How about "Wtnitmtstsaaoof,ottaaasot,aboet."? This looks like total nonsense; that is, until you think of it as the first letters of the words of a sentence. In this case we used a sentence from Shakespeare: "Whether 'tis nobler in the mind to suffer the slings and arrows of outrageous fortune, or to take arms against a sea of troubles, and by opposing end them." Of course, Alice should not use a sentence from literature; literary texts are too accessible for an attacker, and how many suitable sentences would there be in the books on Alice's bookshelf? Instead, she should invent her own sentence, one that nobody else could possibly think of.

Compared to using a full passphrase, the initial-letters-from-each-word technique requires a longer sentence, but it requires less typing for good security because the keystrokes are more random than consecutive letters in a sentence. We don't know of any estimate for the number of bits of entropy per character for this technique. Maybe somebody can do the research and write a paper on various ways of choosing passphrases.

Passphrases are certainly the best way of storing a secret in a human brain. Unfortunately, many users still find it difficult to use them correctly. And even with passphrases, it is extremely difficult to get 128 bits of entropy in the human brain.

22.2.1 Salting and Stretching

To squeeze the most security out of a limited-entropy password or passphrase, we can use two techniques that sound as if they come from a medieval torture chamber. These are so simple and obvious that they should be used in every password system. There is really no excuse not to use them.

The first is to add a *salt*. This is simply a random number that is stored alongside the data that was encrypted with the password. If you can, use a 256-bit salt.

The next step is to *stretch* the password. Stretching is essentially a very long computation. Let p be the password and s be the salt. Using any

cryptographically strong hash function h, we compute

$$\begin{aligned} x_0 &:= 0 \\ x_i &:= h(x_{i-1} \,\|\, p \,\|\, s) \qquad \text{for } i = 1, \ldots, r \\ K &:= x_r \end{aligned}$$

and use K as the key to actually encrypt the data. The parameter r is the number of iterations in the computation, and should be as large as practical. (It goes without saying that x_i and K should be 256 bits long.)

Let's look at this from an attacker's point of view. Given the salt s and some data that is encrypted with K, you try to find K by trying different passwords. Choose a particular password p, compute the corresponding K, decrypt the data and check whether it makes sense. If it doesn't, then p must have been false. To check a single value for p you have to do r different hash computations. The larger r is, the more work the attacker has to do.

In normal use, the stretching computation has to be done every time a password is used. But remember, this is at a point in time where the user has just entered a password. It has probably taken several seconds to enter the password, so using 200 ms for password processing is quite acceptable. Here is our rule to choose r: choose r such that computing K from (s, p) takes 200–1000 ms on the user's equipment. Computers get faster over time, so r should be increasing over time as well. Ideally, you determine r experimentally when the user first sets the password, and store r alongside s. (Do make sure that r is a reasonable value, not too small or too large.)

How much have we gained? If $r = 2^{20}$ (just over a million) then the attacker has to do 2^{20} hash computations for each password she tries. Trying 2^{60} passwords would take 2^{80} hash computations, so effectively using $r = 2^{20}$ makes the effective key size of the password 20 bits longer. The larger r you choose, the larger the gain.

Look at it another way. What r does is stop the attacker from benefiting from faster and faster computers, because the faster computers get, the larger r gets, too. It is a kind of Moore's law compensator, but only in the long run. Ten years from now, the attacker can use the next decade's technology to attack the password you are using today. So you still need a decent security margin and as much entropy in the password as you can get.

This is another reason to use a key negotiation protocol with forward secrecy. Whatever the application, it is quite likely that Alice's private keys end up being protected by a password. Ten years from now, the attacker will be able to search for Alice's password and find it. But if the key that is encrypted with the password was only used to run a key negotiation protocol with forward secrecy, then the attacker will find nothing of value. Alice's key is no longer valid (it has expired), and knowing her old private key does not reveal the session keys used ten years ago.

The salt stops the attacker from taking advantage of an economy of scale when she is attacking a large number of passwords simultaneously. Suppose there are a million users in the system, and each user stores an encrypted file that contains her keys. Each file is encrypted with the user's stretched password. If we did not use a salt, then the attacker can attack as follows: guess a password p, compute the stretched key K, and try to decrypt each of the key files using K. The stretch function only needs to be computed once for every password, and the resulting stretched key can be used in an attempt to decrypt each of the files.

This is no longer possible when we add the salt to the stretching function. All the salts are random values, so each user will use a different salt value. The attacker now has to compute the stretching function once for each password/file combination, rather than once for each password. This is a lot more work for the attacker, and it comes at a very small price for the users of the system.

So how large a salt value do you need? We can't be bothered to do a detailed analysis. Maybe you can get away with 128 bits (if there are no possible birthday attacks), but why bother? Bits are cheap, and using a 256-bit salt is simpler for everybody.

By the way, do take care when you do this. We once saw a system that implemented all this perfectly, but then some programmer wanted to improve the user interface by giving the user a faster response as to whether the password he had typed was correct or not. So he stored a checksum on the password, which defeated the entire salting and stretching procedure. If the response time is too slow, you can reduce r a bit, but make sure that there is no way to recognize whether a password is correct or not without doing at least r hash computations.

22.3 Portable Storage

The next idea is to store key material outside the computer. The simplest form of storage is a piece of paper with passwords written on it. Most people have that in one form or another, even for noncryptographic systems like Web sites. Many users have at least half a dozen passwords to remember—we certainly have—and that is simply too much, especially for systems where you use your password only rarely. So to remember passwords, users write them down. The limitation to this solution is that the password still has to be processed by the user's eyes, brain, and fingers every time it is used. To keep user irritation and mistakes within reasonable bounds, this technique can only be used with relatively low-entropy passwords and passphrases.

As a designer, you don't have to design or implement anything to use this storage method. Users will use it for their passwords, no matter which rules you make and however you create your password system.

A more advanced form of storage would be a portable memory of some form. This could be a memory-chip card, a floppy disk, a magnetic stripe card, or any other kind of digital storage. Digital storage systems are always large enough to store at least a 256-bit secret key, so we can eliminate the low-entropy password. The portable memory becomes very much like a key. Whoever holds the key has access, so this memory needs to be held securely.

22.4 Secure Token

A better—and more expensive—solution is to use something we call a secure token. This is a small computer that Alice can carry around. The external shape of tokens can differ widely, ranging from a smart card (which looks just like a credit card), to an iButton, USB dongle, or PCMCIA card. The main properties are a nonvolatile memory (i.e., a memory that retains its data when power is removed) and a CPU.

The secure token works primarily as a portable storage device, but with a few security enhancements. First of all, access to the stored key material can be limited by a password or something similar. Before the secure token will let you use the key, you have to send it the proper password. The

token can protect itself against attackers that try a brute-force search for the password by disabling access after three or five failed attempts. Of course, some users mistype their password too often, and then their token has to be resuscitated, but you can use longer, higher-entropy passphrases or keys that are far more secure for the resuscitation.

This provides a multilevel defense. First of all, Alice protects the physical token; for example, by keeping it in her wallet or on her key chain. An attacker has to steal the token to get anywhere, or at least get access to it in some way. Then the attacker needs to either physically break open the token and extract the data, or find the password to unlock the token. Tokens are often tamper-resistant to make a physical attack more difficult.[2]

Secure tokens are currently one of the best and most practical methods of storing secret keys. They can be relatively inexpensive and small enough to be carried around conveniently.

One problem in practical use is the behavior of the users. They'll leave their secure token plugged into their computer when going to lunch or to a meeting. As users don't want to be prompted for their password every time, the system will be set to allow hours of access from the last time the password was entered. So all an attacker has to do is walk in and start using the secret keys stored in the token.

You can try to solve this through training. There's the "corporate security in the office" video presentations, the embarrassingly bad "take your token to lunch" poster that isn't funny at all, and the "if I ever again find your token plugged in unattended, you are going to get another speech like this" speeches. But you can also use other means. Make sure the token is not only the key to access digital data, but also the lock to the office doors, so that users have to take their token to get back into their office. Fix the coffee machine to only give coffee after being presented with a token. This motivates employees to bring their token to the coffee machine and not leave it plugged into their computer while they are away. Sometimes security consists of silly measures like these, but they work far better than trying to enforce take-your-token-with-you rules by other means.

[2]They are tamper-resistant, not tamper-proof. Tampering is always possible; tamper-resistance merely makes tampering more expensive.

22.5 Secure UI

The secure token still has a significant weakness. The password that Alice uses has to be entered on the PC or some other device. As long as we trust the PC this is not a problem, but we all know PCs are not terribly secure, to say the least. In fact, the whole reason for not storing Alice's keys on the PC is because we don't trust it enough. We can achieve a much better security if the token itself has a secure built-in UI. Think of a secure token with a built-in keyboard and display. Now the password, or more likely a PIN, can be entered directly into the token without the need to trust an outside device.

Having a keyboard on the token protects the PIN from compromise. Of course, once the PIN has been typed, the PC still gets the key, and then it can do anything at all with that key. So we are still limited by the security of the whole PC.

To stop this, we have to put the cryptographic processes that involve the key into the token. This requires application-specific code in the token. The token is quickly growing to a full-fledged computer, but now a trusted computer that the user carries around. The trusted computer can implement the security-critical part of each application on the token itself. The display now becomes crucial, since it is used to show the user what action he is authorizing by typing his PIN. In a typical design, the user uses the PC's keyboard and mouse to operate the application. When, for example, a bank payment has to be authorized, the PC sends the data to the token. The token displays the amount and a few other transaction details, and the user authorizes the transaction by typing her PIN. The token then signs the transaction details and the PC completes the rest of the transaction.

In reality, tokens with a secure UI are too expensive for most applications. Maybe the closest thing we have is a PDA, such as a Palm. However, people download programs onto their PDAs, and a PDA is not designed from the start as a secure unit, so perhaps the PDA is not significantly more secure than a PC.

This is an excellent example of the conflict between security and functionality. People like to be able to download programs and run them any time

they want. They also like to be able to trust their computer with valuable information. Given the current state of the art in operating system security, it is not possible to combine in a single machine the flexibility of downloadable programs with the security that we need. And given the choice between security and downloading a program that will show dancing pigs on the screen, users will choose dancing pigs just about every time.

22.6 Biometrics

If we want to get really fancy, we can add biometrics to the mix. You could build something like a fingerprint or iris scanner into the secure token. At the moment, biometric devices are not very useful. Fingerprint scanners can be made for a reasonable price, but the security they provide is abysmal. In 2002, cryptographer Tsutomu Matsumoto, together with three of his students, showed how he was able to consistently fool all the commercially available fingerprint scanners he could buy, using only household and hobby materials [64]. Even making a fake finger from a latent fingerprint (i.e., the ones you leave on every shiny surface) is nothing more than a hobby project for a clever high-school student.

The real shock to us wasn't that the fingerprint readers could be fooled. It was that fooling them was so incredibly simple and cheap. What's worse, the biometrics industry has been telling us how secure biometric identification is. They never told us that forging fingerprints was this easy. Now suddenly a mathematician (not even a biometrics expert) comes along and blows the whole process out of the water. There are two possibilities: either the industry knew about this and has been lying to us, or they didn't know about it, and they are therefore obviously incompetent. You really have to wonder how secure other biometric systems are.

Still, even though they are easy to fool, fingerprint scanners can be very useful. Suppose you have a secure token with a small display, a small keyboard, and a fingerprint scanner. To get at the key, you need to get physical control of the token, get the PIN, and forge the fingerprint. That is more work for the attacker than any of our previous solutions. It is probably the best practical key storage scheme that we can currently make. On the other

hand, this secure token is going to be rather expensive, so it won't be used by many people.

Fingerprint scanners could also be used on the low-security side rather than the high-security side. Touching a finger to a scanner can be done very quickly, and it is quite feasible to ask the user to do that relatively often. A fingerprint scanner could thus be used to increase the confidence that the proper person is in fact authorizing the actions that the computer is taking. This makes it more difficult for employees to lend their password to a colleague. Rather than trying to stop sophisticated attackers, the fingerprint scanner could be used to stop casual breaches of the security rules. This might make a more important contribution to security than trying to use the scanner as a high-security device.

As a side note, Matsumoto's work might have serious repercussions for the use of fingerprint evidence in criminal cases. It turns out to be relatively easy to make a fake finger from a latent fingerprint. So how about putting a bit of sweat on the fake finger and using it to place someone else's fingerprint somewhere? Up to now this has been the stuff of spy movies, but it might be rather easy to do. Suddenly, the fingerprints on the murder weapon don't mean nearly as much as they used to. It could easily take a decade or more for the legal system to adapt to the idea that fingerprints can be forged with relative ease; meanwhile it remains a perfect tool for framing someone.

22.7 Single Sign-On

Because the average user has so many passwords, it becomes very appealing to create a single sign-on system. The idea is to give Alice a single master password, which in turn is used to encrypt all the different passwords from her different applications.

To do this well, all the applications must talk to the single sign-on system. Any time an application requires a password, it should not ask the user but rather the single sign-on program for it. In practice, this just plain doesn't work. There is no widely-used standard for this process, and until there is, it won't happen automatically. Just think of all the different applications that would have to be changed to automatically get their passwords from the single sign-on system.

A simpler idea is to have a small program that stores the passwords in a text file. Alice types her master password and then uses the copy and paste functionality to copy the passwords from the single sign-on program to the application. Bruce designed a public domain program called Password Safe to do exactly this. But it's just the digital version of the piece of paper that Alice writes her passwords on. It is useful, and an improvement on the piece of paper if you always use the same computer, but not the ultimate solution that the single sign-on idea would really like to be.

22.8 Risk of Loss

But what if the secure token breaks? Or the piece of paper with the passwords is left in a pocket and run through the washing machine? Losing secret keys is always a bad thing. The cost can vary from having to reregister for each application to get a new key, to permanently losing access to important data. If you encrypt your Ph.D. thesis that you have been working on for five years with a secret key and then lose the key, you no longer have a Ph.D. thesis. You just have a file of random-looking bits. Ouch!

It is hard to make a key storage system both easy to use and highly reliable. A good rule of thumb, therefore, is to split these functions. Keep two copies of the key—one that is easy to use, and another one that is very reliable. If the easy-to-use system ever forgets the key, you can recover it from the reliable storage system. The reliable system could be very simple. How about a piece of paper in a bank vault?

Of course, you want to be careful with your reliable storage system. By design, it will quickly be used to store all of your keys, and that would make it a very tempting target for an attacker. You'll have to do a risk analysis to determine whether it is better to have a number of smaller reliable key storage places or a single large one.

22.9 Secret Sharing

There are some keys that you need to store super-securely—for example, the private root key of your CA. As we have seen, storing a secret in a

secure manner can be difficult. Storing it securely and reliably is even more difficult.

There is one cryptographic solution that can help in storing secret keys. It is called *secret sharing*, which is a bit of a misnomer because it implies that you share the secret with several people. You don't. The idea is to take the secret and split it into several different shares. It is possible to do this in such a way that, for example, three out of the five shares are needed to recover the secret. You then give one share to each of the senior people in the IT department. Any three of them can recover the secret. The real trick is to do it in a manner such that any two people together know absolutely nothing about the key.

Secret sharing systems are very tempting from an academic point of view. Each of the shares is stored using one of the techniques we talked about before. A k-out-of-n rule combines a high security (at least k people are necessary to retrieve the key) with a high reliability ($n-k$ of the shares may be lost without detrimental effect). There are even fancier secret sharing schemes that allow more complex access rules, along the lines of (Alice and Bob) or (Alice and Carol and David).

In real life, secret sharing schemes are rarely used because they are too complex. They are complex to implement, but more importantly, complex to administrate and operate. Most companies do not have a group of highly responsible people who distrust each other. Try telling the board members that they will each be given a secure token with a key share, and that they will have to show up at 3 a.m. on a Sunday in an emergency. Oh yes, and they are not to trust each other, but to keep their own shares secure even from other board members. They will also need to come down to the secure key-management room to get a new key share every time someone joins or leaves the board. In practice, this means that using the board members is out. The CEO isn't very useful for holding a share either, because the CEO tends to travel quite a bit. Before you know it, you are down to the two or three senior IT management people. They could use a secret sharing scheme, but the expense and complexity make this unattractive. Why not use something much simpler, such as a safe? Physical solutions such as safes or bank vaults have several advantages. Everybody understands how they work, so you don't need extensive training. They have already been tested extensively, whereas the secret-reconstruction process is hard to test because

it requires such a large number of user interactions—and you really don't want to have a bug in the secret-reconstruction process that results in you losing the root key of your CA.

We will not explain how secret sharing schemes operate in detail. That explanation requires quite a bit of mathematical background, and since secret sharing schemes are so rarely used, there is no point in including them here.

22.10 Wiping Secrets

Any long-term secret that we store eventually has to be wiped. As soon as a secret is no longer needed, its storage location should be wiped to avoid any future compromise. We discussed the problems of wiping memory in section 9.3. Wiping long-term secrets from permanent storage is much harder.

The schemes for storing long-term secrets that we discussed in this chapter use a variety of data storage technologies: hard disk, paper, floppy disk, magnetic stripe card, EPROM, EEPROM, flash memory, or battery-maintained RAM. None of these storage technologies comes with a documented wiping functionality that guarantees that the data it stored is no longer recoverable.

22.10.1 Paper

Destroying a password written down on paper is typically done by destroying the paper itself. One possible method is to burn the paper, and then grind the ashes into a fine powder, or mix the ashes into a pulp with just a little bit of water. Shredding is also an option, although many shredders leave the paper in large enough pieces that reconstructing a page is relatively easy.

22.10.2 Magnetic Storage

Magnetic media are very hard to wipe. There is surprisingly little literature about how to do this; the best paper we know of is by Peter Gutmann [39], although the technical details of that paper are probably outdated now.

Magnetic media store data in tiny magnetic domains; the direction of magnetization of a domain determines the data it encodes. When the data is overwritten, the magnetization directions are changed to reflect the new data. But there are several mechanisms that prevent the old data from being completely lost. The read/write head that tries to overwrite old data is never exactly aligned, and will tend to leave some parts of the old data untouched. Overwriting does not completely destroy old data. You can think of it as repainting a wall with a single coat of paint. You can still vaguely see the old coat of paint under it. The magnetic domains can also migrate away from the read/write head either to the side of the track or deeper down into the magnetic material, where they can linger for a long time. Overwritten data is typically not recoverable with the normal read/write head, but an attacker who takes apart a disk drive and uses specialized equipment might be able to retrieve some or all of the old data.

In practice, repeatedly overwriting a secret with random data is probably the best option. There are a few points to keep in mind:

- Each overwrite should use fresh random data. Some researchers have developed particular data patterns that are supposed to be better at wiping old data, but the choice of patterns depends on the exact details of the disk drive. Random data might require more overwriting passes for the same effect, but it works in all situations and is therefore safer.
- Overwrite the actual location that stored the secret. If you just change a file by writing new data to it, the file system might decide to store the new data in a different location, which would leave the original data intact.
- Make sure that each overwrite pass is actually written to disk and not just to one of the disk caches. Disk drives that have their own write-cache are a particular danger, as they might cache the new data and optimize the multiple overwrite operations into a single write.
- It is probably a good idea to wipe an area that begins well before the secret data and that ends well after it. Because the rotational speed of a disk drive is never perfectly constant, the new data will not align perfectly with the old data.

As far as we know, there is no reliable information on how many overwrite passes are required, but there is no reason to choose a small number. You only have to wipe a single key. (If you have a large amount of secret data, store that data encrypted under a key, and only wipe the key.) We consider 50 or 100 overwrites with random data perfectly reasonable.

It is theoretically possible to erase a tape or disk using a degaussing machine. However, modern high-density magnetic storage media resist degaussing to such an extent that this is not a reliable wiping method. In practice, users do not have access to degaussing machines, so this is a nonissue.

Even with extensive overwriting, you should expect that a highly specialized and well-funded attacker could still recover the secret from the magnetic medium. To completely destroy the data, you will probably have to destroy the medium itself. If the magnetic layer is bonded to plastic (floppy disk, tape), you can consider shredding and then burning the media. For a hard disk, you can use a belt sander to remove the magnetic layer from the platters, or use a blowtorch to melt the disk platters down to liquid metal. In practice, you are unlikely to convince users to take such extreme measures, so repeated overwriting is the best practical solution.

22.10.3 Solid-State Storage

Wiping nonvolatile memory, such as EPROM, EEPROM, and flash, poses similar problems. Overwriting old data does not remove all traces, and the data retention mechanisms we discussed in section 9.3.4 are also at work. Again, repeatedly overwriting the secret with random data is the only practical solution, but it is by no means perfect. As soon as the solid-state device is no longer needed, it should be destroyed.