

MOSERWARE

JEFF MOSER'S SOFTWARE DEVELOPMENT ADVENTURES.

WEDNESDAY, JUNE 10, 2009

The First Few Milliseconds of an HTTPS Connection

Convinced from spending hours reading [rave reviews](#), Bob eagerly clicked "Proceed to Checkout" for his gallon of [Tuscan Whole Milk](#) and...

Whoa! What just happened?



In the 220 milliseconds that flew by, a lot of interesting stuff happened to make Firefox change the address bar color and put a lock in the lower right corner. With the help of [Wireshark](#), my favorite network tool, and a slightly modified debug build of Firefox, we can see *exactly* what's going on.

By agreement of [RFC 2818](#), Firefox knew that "https" meant it should connect to [port 443](#) at Amazon.com:

```

Internet Protocol, Src: 172.17.30.63 (172.17.30.63), Dst: 7
Transmission Control Protocol, Src Port: 50752 (50752), Dst
  Source port: 50752 (50752)
  Destination port: https (443)
  Sequence number: 1 (relative sequence number)
  [Next sequence number: 164 (relative sequence number)]
  Acknowledgement number: 1 (relative ack number)
  Header length: 20 bytes
  Flags: 0x18 (PSH, ACK)
  Window size: 64860

```

Most people associate HTTPS with [SSL](#) (Secure Sockets Layer) which was [created by Netscape in the mid 90's](#). This is becoming less true over time. As Netscape lost market share, SSL's maintenance moved to the Internet Engineering Task Force ([IETF](#)). The first post-Netscape version was re-branded as Transport Layer Security ([TLS](#)) 1.0 which [was released](#) in January 1999. It's rare to see true "SSL" traffic given that TLS has been around for 10 years.

Client Hello

TLS wraps all traffic in "records" of different types. We see that the first byte out of our browser is the hex byte 0x16 = 22 which [means](#) that this is a "handshake" record:

ABOUT ME



JEFF MOSER
FISHERS, INDIANA,
UNITED STATES

[VIEW MY
COMPLETE PROFILE](#)

BLOG ARCHIVE

▼ [2009](#) (5)

▼ [June](#) (1)

The First Few
Milliseconds of an
HTTPS Connection

► [April](#) (1)

► [March](#) (1)

► [February](#) (1)

► [January](#) (1)

► [2008](#) (20)

► [2007](#) (15)



Subscribe

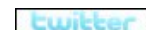
Subscribe by Email

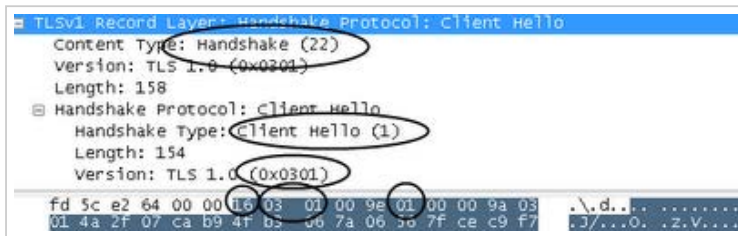
2554 readers
BY FEEDBURNER



CONTACT ME

jeff@moserware.com

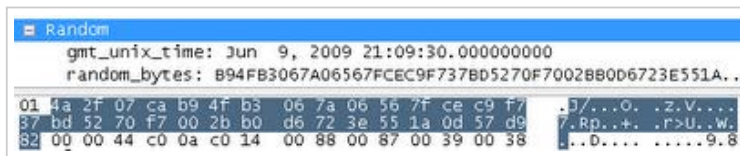




The next two bytes are 0x0301 which indicate that this is a version 3.1 record which shows that TLS 1.0 is essentially SSL 3.1.

The handshake record is broken out into several messages. The first is our "Client Hello" message (0x01). There are a few important things here:

- Random:



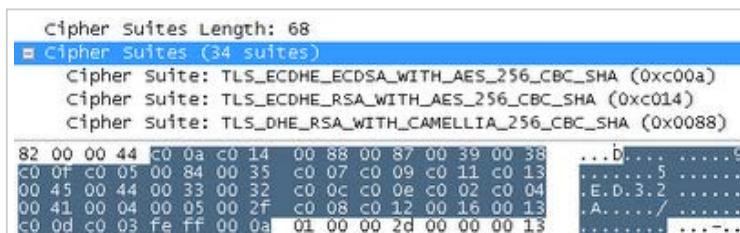
There are four bytes representing the current Coordinated Universal Time (UTC) in the Unix epoch format, which is the number of seconds since January 1, 1970. In this case, 0x4a2f07ca. It's followed by 28 random bytes. This will be used later on.

- Session ID:



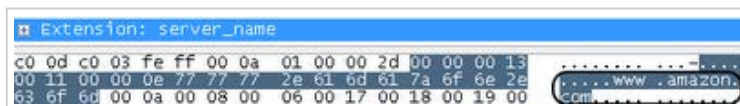
Here it's empty/null. If we had previously connected to Amazon.com a few seconds ago, we could potentially resume a session and avoid a full handshake.

- Cipher Suites:



This is a list of all of the encryption algorithms that the browser is willing to support. Its top pick is a very strong choice of "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA" followed by 33 others that it's willing to accept. Don't worry if none of that makes sense. We'll find out later that Amazon doesn't pick our first choice anyway.

- server_name extension:



This is a way to tell Amazon.com that our browser is trying to reach <https://www.amazon.com/>. This is really convenient because our TLS handshake occurs long before any HTTP traffic. HTTP has a "Host" header which allows a cost-cutting Internet hosting companies to pile hundreds of websites onto a single IP address. SSL has traditionally required a different IP for each site, but this extension allows the server to respond with the appropriate certificate that the browser is looking for. If nothing else, this extension should allow an extra week or so of IPv4 addresses.

Chat with Jeff Moser
Offline

MY RECENT RSS FINDS

[CLR via C#, 3rd Edition](#)

[The Best Bits](#)

[Flashback - A Story of Flash Memory](#)

[What does the optimize switch do?](#)

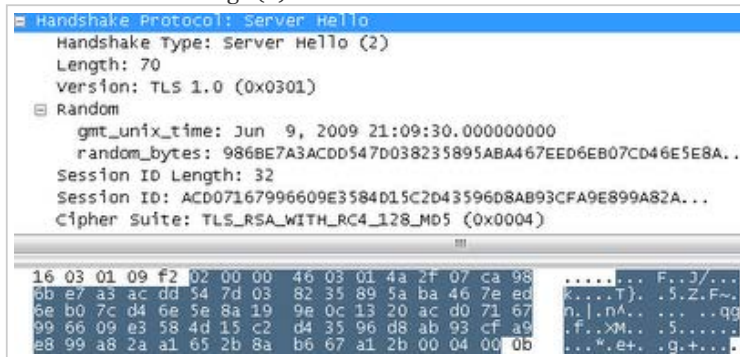
[A Working Theory About RC4](#)

[Read more...](#)

Server Hello

Amazon.com replies with a handshake record that's a massive two packets in size (2,551 bytes). The record has version bytes of 0x0301 meaning that Amazon agreed to our request to use TLS 1.0. This record has three sub-messages with some interesting data:

1. "Server Hello" Message (2):

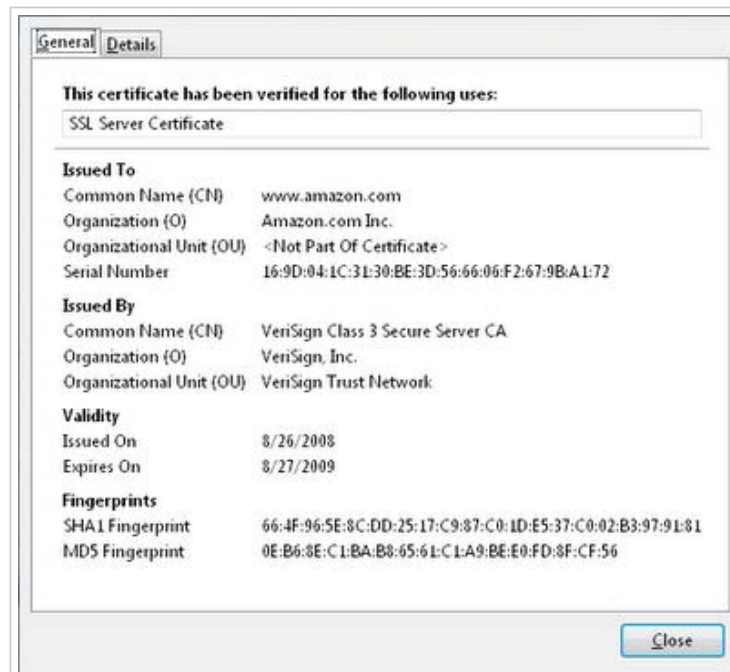


- We get the server's four byte time Unix epoch time representation and its 28 random bytes that will be used later.
- A 32 byte session ID in case we want to reconnect without a big handshake.
- Of the 34 cipher suites we offered, Amazon picked "TLS_RSA_WITH_RC4_128_MD5" (0x0004). This means that it will use the "RSA" [public key](#) algorithm to verify certificate signatures and exchange keys, the [RC4](#) encryption algorithm to encrypt data, and the [MD5](#) hash function to verify the contents of messages. We'll cover these in depth later on. I personally think Amazon had selfish reasons for choosing this cipher suite. Of the ones on the list, it was the one that was least CPU intensive to use so that Amazon could crowd more connections onto each of their servers. A much less likely possibility is that they wanted to pay special tribute to [Ron Rivest](#), who created all three of these algorithms.

2. Certificate Message (11):



- This message takes a whopping 2,464 bytes and is the certificate that the client can use to validate Amazon's. It isn't anything fancy. You can view most of its contents in your browser:



3. "Server Hello Done" Message (14):



- This is a zero byte message that tells the client that it's done with the "Hello" process and indicate that the server won't be asking the client for a certificate.

Checking out the Certificate

The browser has to [figure out](#) if it should trust Amazon.com. In this case, it's using certificates. It looks at Amazon's certificate and [sees](#) that the current time is between the "not before" time of August 26th, 2008 and before the "not after" time of August 27, 2009. It also [checks](#) to make sure that the certificate's public key is authorized for exchanging secret keys.

Why should we trust this certificate?

Attached to the certificate is a "signature" that is just a really long number in [big-endian](#) format:



Anyone could have sent us these bytes. Why should we trust this signature? To answer that question, need to make a speedy detour into [mathemagic land](#):

Interlude: A Short, Not *Too* Scary, Guide to RSA

People [sometimes wonder](#) if math has any relevance to programming. Certificates give a very practical example of applied math. Amazon's certificate tells us that we should use the RSA algorithm to check the signature. [RSA](#) was created in the 1970's

by MIT professors [Ron Rivest](#), [Adi Shamir](#), and [Len Adleman](#) who found a [clever way](#) to combine ideas spanning [2000 years of math](#) development to come up with a [beautifully simple algorithm](#):

You [pick](#) two huge prime numbers "p" and "q." Multiply them to get " $n = p \cdot q$." Next, you pick a small public [exponent](#) "e" which is the "encryption exponent" and a [specially crafted inverse](#) of "e" called "d" as the "decryption exponent." You then **make "n" and "e" public and keep "d" as secret as you possibly can** and then throw away "p" and "q" (or keep them as secret as "d"). It's really important to remember that "e" and "d" are inverses of each other.

Now, if you have some message, you just need to interpret its bytes as a number "M." If you want to "encrypt" a message to create a "ciphertext", you'd calculate:

$$C \equiv M^e \pmod{n}$$

This means that you multiply "M" by itself "e" times. The "mod n" means that we only take the remainder (e.g. "[modulus](#)") when dividing by "n." For example, 11 AM + 3 hours $\equiv$ 2 (PM) (mod 12 hours). The other recipient knows "d" which allows them to invert the message to recover the original message:

$$C^d \equiv (M^e)^d \equiv M^{e \cdot d} \equiv M^1 \equiv M \pmod{n}$$

Just as interesting is that the person with "d" can "sign" a document by raising a message "M" to the "d" exponent:

$$M^d \equiv S \pmod{n}$$

This works because "signer" makes public "S", "M", "e", and "n." Anyone can verify the signature "S" with a simple calculation:

$$S^e \equiv (M^d)^e \equiv M^{d \cdot e} \equiv M^{e \cdot d} \equiv M^1 \equiv M \pmod{n}$$

Public key cryptography algorithms like RSA are often called "asymmetric" algorithms because the encryption key (in our case, "e") is not equal to (e.g. "symmetric" with) the decryption key "d". Reducing everything "mod n" makes it impossible to use the easy techniques that we're used to such as normal [logarithms](#).

The magic of RSA works because you can calculate/encrypt $C \equiv M^e \pmod{n}$ [very quickly](#), but it is *really hard* to calculate/decrypt $C^d \equiv M \pmod{n}$ without knowing "d." As we saw earlier, "d" is derived from [factoring](#) "n" back to its "p" and "q", which is a [tough problem](#).

Verifying Signatures

The big thing to keep in mind with RSA in the real world is that all of the numbers involved have to be *big* to make things really hard to break using the [best algorithms that we have](#). How big? Amazon.com's certificate was "signed" by "VeriSign Class 3 Secure Server CA." From the certificate, we see that this VeriSign modulus "n" is 2048 bits long which has this 617 digit base-10 representation:

```
1890572922 9464742433 9498401781 6528521078 8629616064
3051642608 4317020197 7241822595 6075980039 8371048211
```

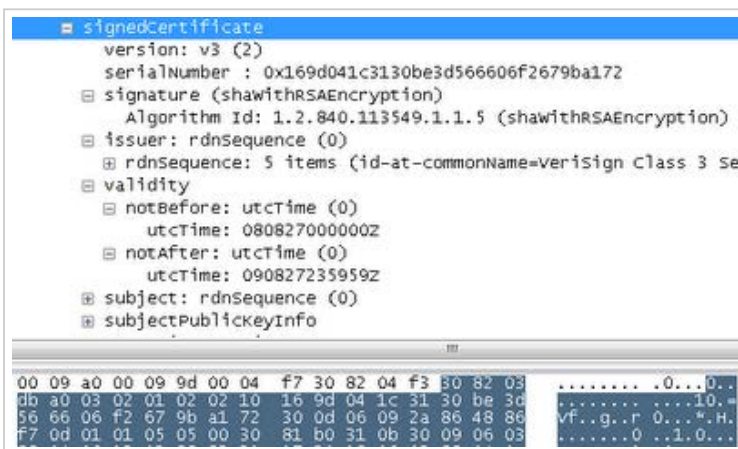
```

4887504542 4200635317 0422636532 2091550579 0341204005
1169453804 7325464426 0479594122 4167270607 6731441028 3698615569
9947933786 3789783838 5829991518 1037601365 0218058341
7944190228 0926880299 3425241541 4300090021 1055372661
2125414429 9349272172 5333752665 6605550620 5558450610
3253786958 8361121949 2417723618 5199653627 5260212221 0847786057
9342235500 9443918198 9038906234 1550747726 8041766919
1500918876 1961879460 3091993360 6376719337 6644159792 1249204891
7079005527 7689341573 9395596650 5484628101 0469658502
1566385762 0175231997 6268718746 7514321

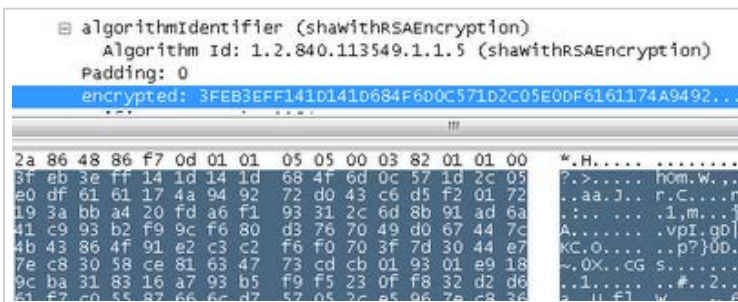
```

(Good luck trying to find "p" and "q" from this "n" - if you could, you could generate real-looking VeriSign certificates.)

VeriSign's "e" is $2^{16} + 1 = 65537$. Of course, they keep their "d" value secret, probably on a safe hardware device protected by retinal scanners and armed guards. Before signing, VeriSign checked the validity of the contents that Amazon.com claimed on its certificate using a real-world "handshake" that involved [looking at several of their business documents](#). Once VeriSign was satisfied with the documents, they used the [SHA-1](#) hash algorithm to get a hash value of the certificate that had all the claims. In Wireshark, the full certificate shows up as the "signedCertificate" part:



It's sort of a misnomer since it actually means that those are the bytes that the signer is *going to sign* and not the bytes that already include a signature.



The actual signature, "S", is simply called "encrypted" in Wireshark. If we raise "S" to VeriSign's public "e" exponent of 65537 and then take the remainder when divided by the modulus "n", we get this "decrypted" signature hex value:

```

0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

```

```

FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF
FFFFFFFF00302130 0906052B 0E03021A 05000414C19F8786
871775C60EFE0542 E4C2167C830539DB

```

Per the [PKCS #1 v1.5 standard](#), the first byte is "00" and it "ensures that the encryption block, [when] converted to an integer, is less than the modulus." The second byte of "01" indicates that this is a private key operation (e.g. it's a signature). This is followed by a lot of "FF" bytes that are used to pad the result to make sure that it's big enough. The padding is terminated by a "00" byte. It's followed by "30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14" which is the [PKCS #1 v2.1 way](#) of specifying the [SHA-1](#) hash algorithm. The last 20 bytes are SHA-1 hash digest of the bytes in "signedCertificate."

Since the decrypted value is [properly formatted](#) and the last bytes are the same hash value that we can calculate independently, we can assume that whoever knew "VeriSign Class 3 Secure Server CA"'s private key "signed" it. We implicitly trust that only VeriSign knows the private key "d."

We can repeat the process to verify that "VeriSign Class 3 Secure Server CA"'s certificate was signed by VeriSign's "Class 3 Public Primary Certification Authority."

But why should we trust *that*? There are no more levels on the trust chain.



The top "VeriSign Class 3 Public Primary Certification Authority" was signed by *itself*. This certificate has been built into Mozilla products as an implicitly trusted good certificate since version [1.4 of certdata.txt](#) in the Network Security Services (NSS) library. It was checked-in on September 6, 2000 by Netscape's Robert Relyea with the following comment:

"Make the framework compile with the rest of NSS. Include a 'live' certdata.txt with those certs we have permission to push to open source (additional certs will be added as we get permission from the owners)."

This decision has had a relatively long impact since the certificate has a validity range of January 28, 1996 - August 1, 2028.

As Ken Thompson explained so well in his "[Reflections on Trusting Trust](#)", you ultimately have to implicitly trust somebody. There is no way around this problem. In this case, we're implicitly trusting that Robert Relyea made a good choice. We also hope that [Mozilla's built-in certificate policy](#) is reasonable for the other built-in certificates.

One thing to keep in mind here is that all these certificates and signatures were simply used to form a trust chain. On the public Internet, VeriSign's root certificate is implicitly trusted by Firefox long before you go to any website. In a company, you

can create your own root certificate authority (CA) that you can install on everyone's machine.

Alternatively, you can get around having to pay companies like VeriSign and avoid certificate trust chains altogether. Certificates are used to establish trust by using a trusted third-party (in this case, VeriSign). If you have a secure means of sharing a secret "key", such as whispering a long password into someone's ear, then you can use that pre-shared key (PSK) to establish trust. There are extensions to TLS to allow this, such as [TLS-PSK](#), and my personal favorite, [TLS with Secure Remote Password \(SRP\) extensions](#). Unfortunately, these extensions aren't nearly as widely deployed and supported, so they're usually not practical. Additionally, these alternatives impose a burden that we have to have some other secure means of communicating the secret that's more cumbersome than what we're trying to establish with TLS (otherwise, why wouldn't we use *that* for everything?).

One final check that we need to do is to verify that the host name on the certificate is what we expected. [Nelson Bolyard](#)'s comment in the [SSL_AuthCertificate](#) function explains why:

```
/* cert is OK. This is the client side of an SSL connection.
 * Now check the name field in the cert against the desired hostname.
 * NB: This is our only defense against Man-In-The-Middle (MITM) attacks! */
```

This check helps prevent against a [man-in-the-middle](#) attack because we are implicitly trusting that the people on the certificate trust chain wouldn't do something bad, like sign a certificate claiming to be from Amazon.com unless it actually was Amazon.com. If an attacker is able to modify your DNS server by using a technique like [DNS cache poisoning](#), you might be fooled into thinking you're at a trusted site (like Amazon.com) because the address bar will look normal. This last check implicitly trusts certificate authorities to stop these bad things from happening.

Pre-Master Secret

We've verified some claims about Amazon.com and know its public encryption exponent "e" and modulus "n." Anyone listening in on the traffic can know this as well (as evidenced because we are using Wireshark captures). Now we need to create a random secret key that an eavesdropper/attacker can't figure out. This isn't as easy as it sounds. In 1996, researchers figured out that [Netscape Navigator 1.1](#) was [using only three sources](#) to seed their pseudo-random number generator (PRNG). The sources were: the time of day, the process id, and the parent process id. As the researchers showed, these "random" sources aren't that random and were relatively easy to figure out.

Since everything else was derived from these three "random" sources, it was possible to "break" the SSL "security" in 25 seconds on a 1996 era machine. If you still don't believe that finding randomness is hard, just [ask the Debian OpenSSL maintainers](#). If you mess it up, all the security built on top of it is suspect.

On Windows, random numbers used for cryptographic purposes are generated by calling the [CryptGenRandom](#) function that hashes bits [sampled from over 125](#)

[sources](#). Firefox uses this function along with some bits derived from [its own function](#) to seed its [pseudo-random number generator](#).

The 48 byte "pre-master secret" random value that's generated isn't used directly, but it's very important to keep it secret since a lot of things are derived from it. Not surprisingly, Firefox makes it hard to find out this value. I had to compile a debug version and set the [SSLDEBUGFILE](#) and [SSLTRACE](#) environment variables to see it.

In this particular session, the pre-master secret showed up in the SSLDEBUGFILE as:

```
4456: SSL[131491792]: Pre-Master Secret [Len: 48]
03 01 bb 7b 08 98 a7 49 de e8 e9 b8 91 52 ec 81 ...{...I.....R..
4c c2 39 7b f6 ba 1c 0a b1 95 50 29 be 02 ad e6 L.9{.....P)....
ad 6e 11 3f 20 c4 66 fo 64 22 57 7e e1 06 7a 3b .n.? .f.d"W~..z;
```

Note that it's not completely random. The first two bytes are, [by convention](#), the TLS version (03 01).

Trading Secrets

We now need to get this secret value over to Amazon.com. By Amazon's wishes of "TLS_RSA_WITH_RC4_128_MD5", we will use RSA to do this. You *could* make your input message equal to just the 48 byte pre-master secret, but the Public Key Cryptography Standard (PKCS) #1, version 1.5 RFC [tells us](#) that we should pad these bytes with *random* data to make the input equal to exactly the size of the modulus (1024 bits/128 bytes). This makes it harder for an attacker to determine our pre-master secret. It also gives us one last chance to protect ourselves in case we did something really bone-headed, like reusing the same secret. If we reused the key, the eavesdropper would likely see a different value placed on the network due to the random padding.

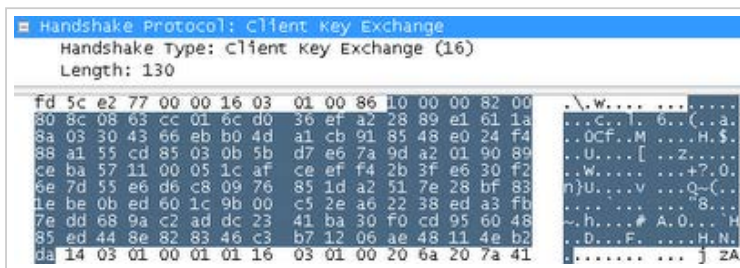
Again, Firefox makes it hard to see these random values. I had to insert debugging statements into [the padding function](#) to see what was going on:

```
wrapperHandle = fopen("plaintextpadding.txt", "a");
fprintf(wrapperHandle, "PLAINTEXT = ");
for(i = 0; i < modulusLen; i++)
{
    fprintf(wrapperHandle, "%02X ", block[i]);
}
fprintf(wrapperHandle, "\r\n");
fclose(wrapperHandle);
```

In this session, the full padded value was:

```
00 02 12 A3 EA B1 65 D6 81 6C 13 14 13 62 10 53 23 B3 96 85 FF 24 FA CC
46 11 21 24 A4 81 EA 30 63 95 D4 DC BF 9C CC Do 2E DD 5A A6 41 6A 4E
82 65 7D 70 7D 50 09 17 CD 10 55 97 B9 C1 A1 84 F2 A9 AB EA 7D F4 CC 54
E4 64 6E 3A E5 91 A0 06 00 03 01 BB 7B 08 98 A7 49 DE E8 E9 B8 91 52
EC 81 4C C2 39 7B F6 BA 1C 0A B1 95 50 29 BE 02 AD E6 AD 6E 11 3F 20 C4
66 Fo 64 22 57 7E E1 06 7A 3B
```

Firefox took this value and [calculated](#) " $C \equiv M^e \pmod n$ " to get the value we see in the "Client Key Exchange" record:



Finally, Firefox sent out one last unencrypted message, a "Change Cipher Spec" record:



This is Firefox's way of telling Amazon that it's going to start using the agreed upon secret to encrypt its next message.

Deriving the Master Secret

If we've done everything correctly, both sides (and only those sides) now know the 48 byte (256 bit) pre-master secret. There's a slight trust issue here from Amazon's perspective: the pre-master secret just has bits that were generated by the client, they don't take anything into account from the server or anything we said earlier. We'll fix that by computing the "master secret." [Per the spec](#), this is done by calculating:

```
master_secret = PRF(pre_master_secret, "master secret",
    ClientHello.random + ServerHello.random)
```

The "pre_master_secret" is the secret value we sent earlier. The "master secret" is simply a string whose [ASCII](#) bytes (e.g. "6d 61 73 74 65 72 ...") are used. We then concatenate the random values that were sent in the ClientHello and ServerHello (from Amazon) messages that we saw at the beginning.

The PRF is the "Pseudo-Random Function" that's also [defined in the spec](#) and is quite clever. It combines the secret, the ASCII label, and the seed data we give it by using the keyed-Hash Message Authentication Code ([HMAC](#)) versions of both [MD5](#) and [SHA-1](#) hash functions. Half of the input is sent to each hash function. It's clever because it is quite resistant to attack, even in the face of [weaknesses in MD5 and SHA-1](#). This process can feedback on itself and iterate forever to generate as many bytes as we need.

Following this procedure, we obtain a 48 byte "master secret" of

```
4C AF 20 30 8F 4C AA C5 66 4A 02 90 F2 AC 10 00 39 DB 1D E0 1F CB E0
E0 9D D7 E6 BE 62 A4 6C 18 06 AD 79 21 DB 82 1D 53 84 DB 35 A7 1F C1 01
19
```

Generating Lots of Keys

Now that both sides have a "master secrets", the spec [shows us](#) how we can derive all the needed session keys we need using the PRF to create a "key block" where we will pull data from:

```
key_block = PRF(SecurityParameters.master_secret, "key expansion",
SecurityParameters.server_random + SecurityParameters.client_random);
```

The bytes from "key_block" are used to populate the following:

```
client_write_MAC_secret[SecurityParameters.hash_size]
server_write_MAC_secret[SecurityParameters.hash_size]
client_write_key[SecurityParameters.key_material_length]
server_write_key[SecurityParameters.key_material_length]
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

Since we're using a [stream cipher](#) instead of a [block cipher](#) like the Advanced Encryption Standard ([AES](#)), we don't need the Initialization Vectors (IVs). Therefore, we just need two Message Authentication Code ([MAC](#)) keys for each side that are 16 bytes (128 bits) each since the specified MD5 hash digest size is 16 bytes. In addition, the RC4 cipher uses a 16 byte (128 bit) key that both sides will need as well. All told, we need $2 \times 16 + 2 \times 16 = 64$ bytes from the key block.

Running the PRF, we get these values:

```
client_write_MAC_secret = 80 B8 F6 09 51 74 EA DB 29 28 EF 6F 9A B8 81
Bo
server_write_MAC_secret = 67 7C 96 7B 70 C5 BC 62 9D 1D 1F 4A A6 79 81
61
client_write_key = 32 13 2C DD 1B 39 36 40 84 4A DE E5 6C 52 46 72
server_write_key = 58 36 C4 0D 8C 7C 74 DA 6D B7 34 0A 91 B6 8F A7
```

Prepare to be Encrypted!

The last handshake message the client sends out is the "[Finished message](#)." This is a clever message that proves that no one tampered with the handshake and it proves that we know the key. The client takes all bytes from all handshake messages and puts them into a "handshake_messages" buffer. We then calculate 12 bytes of "verify_data" using the pseudo-random function (PRF) with our master key, the label "client finished", and an MD5 and SHA-1 hash of "handshake_messages":

```
verify_data = PRF(master_secret, "client finished",
MD5(handshake_messages) + SHA-1(handshake_messages)) [12]
```

We take the result and add a record header byte "0x14" to indicate "finished" and length bytes "00 00 0c" to indicate that we're sending 12 bytes of verify data. Then, like all future encrypted messages, we need to make sure the decrypted contents haven't been tampered with. Since our cipher suite in use is TLS_RSA_WITH_RC4_128_MD5, this means we use the MD5 hash function.

Some people get paranoid when they hear MD5 because it has some weaknesses. I certainly don't advocate using it as-is. However, TLS is smart in that it doesn't use

MD5 directly, but rather the **HMAC** version of it. This means that instead of using MD5(m) directly, we calculate:

$$\text{HMAC_MD5}(\text{Key}, m) = \text{MD5}((\text{Key} \oplus \text{opad}) ++ \text{MD5}((\text{Key} \oplus \text{ipad}) ++ m))$$

(The $\oplus$ means **XOR**, ++ means concatenate, "opad" is the bytes "5c 5c ... 5c", and "ipad" is the bytes "36 36 ... 36").

In particular, we calculate:

```
HMAC_MD5(client_write_MAC_secret, seq_num + TLSCompressed.type +
TLSCompressed.version + TLSCompressed.length +
TLSCompressed.fragment));
```

As you can see, we include a sequence number ("seq_num") along with attributes of the plaintext message (here it's called "TLSCompressed"). The sequence number foils attackers who might try to take a previously encrypted message and insert it midstream. If this occurred, the sequence numbers would definitely be different than what we expected. This also protects us from an attacker dropping a message.

All that's left is to encrypt these bytes.

RC4 Encryption

Our negotiated cipher suite was TLS_RSA_WITH_RC4_128_MD5. This tells us that we need to use **Ron's Code #4 (RC4)** to encrypt the traffic. **Ron Rivest** developed the RC4 algorithm to generate random bytes based on a 256 byte key. The algorithm is so simple you can actually memorize it in a few minutes.

RC4 begins by creating a 256-byte "S" byte array and populating it with 0 to 255. You then iterate over the array by mixing in bytes from the key. You do this to create a state machine that is used to generate "random" bytes. To generate a random byte, we shuffle around the "S" array.

Put graphically, it looks like this:



To encrypt a byte, we **xor** this pseudo-random byte with the byte we want to encrypt. Remember that xor'ing a bit with 1 causes it to flip. Since we're generating random numbers, on average the xor will flip half of the bits. This random bit flipping is effectively how we encrypt data. As you can see, it's not very complicated and thus it runs quickly. I think that's why Amazon chose it.

Recall that we have a "client_write_key" and a "server_write_key." The means we need to create two RC4 instances: one to encrypt what our browser sends and the other to decrypt what the server sent us.

The first few random bytes out of the "client_write" RC4 instance are "7E 20 7A 4D FE FB 78 A7 33 ...". If we xor these bytes with the unencrypted header and verify message bytes of "14 00 00 0C 98 Fo AE CB C4 ...", we'll get what appears in the encrypted portion that we can see in Wireshark:



The server does almost the same thing. It sends out a "Change Cipher Spec" and then a "Finished Message" that includes all handshake messages, including the *decrypted* version of the client's "Finished Message." Consequently, this proves to the client that the server was able to successfully decrypt our message.

Welcome to the Application Layer!

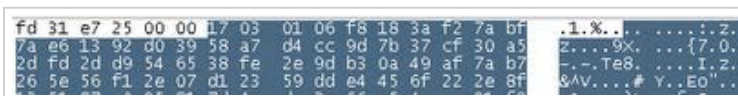
Now, 220 milliseconds after we started, we're finally ready for the application layer. We can now send normal HTTP traffic that'll be encrypted by the TLS layer with the RC4 write instance and decrypt traffic with the server RC4 write instance. In addition, the TLS layer will check each record for tampering by computing the HMAC_MD5 hash of the contents.

At this point, the handshake is over. Our TLS record's content type is now 23 (0x17). Encrypted traffic begins with "17 03 01" which indicate the record type and TLS version. These bytes are followed by our encrypted size, which includes the HMAC hash.

Encrypting the plaintext of:

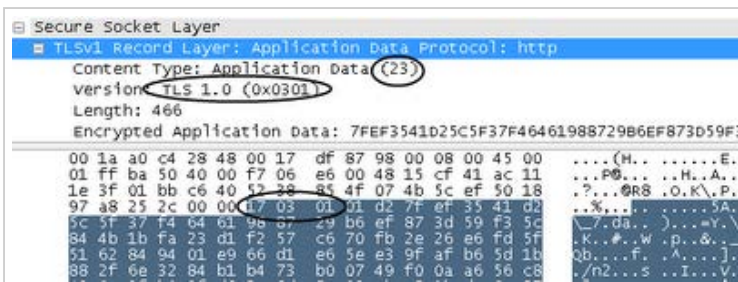
```
GET /gp/cart/view.html/ref=pd_luc_mri HTTP/1.1
Host: www.amazon.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.10)
Gecko/2009060911 Minefield/3.0.10 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
...
```

will give us the bytes we see on the wire:



The only other interesting fact is that the sequence number increases on each record, it's now 1 (and the next record will be 2, etc).

The server does the same type of thing on its side using the server_write_key. We see its response, including the tell-tale application data header:



Decrypting this gives us:

```
HTTP/1.1 200 OK
Date: Wed, 10 Jun 2009 01:09:30 GMT
Server: Server
...
Cneonction: close
Transfer-Encoding: chunked
```

which is a normal HTTP reply that includes a non-descriptive "Server: Server" header and a misspelled "[Cneonction: close](#)" header coming from Amazon's load balancers.

TLS is just below the application layer. The HTTP server software can act as if it's sending unencrypted traffic. The only change is that it writes to a library that does all the encryption. [OpenSSL](#) is a popular open-source library for TLS.

The connection will stay open while both sides send and receive encrypted data until either side sends out a "[closure alert](#)" message and then closes the connection. If we reconnect shortly after disconnecting, we can re-use the negotiated keys (if the server still has them cached) without using public key operations, otherwise we do a completely new full handshake.

It's important to realize that application data records can be *anything*. The only reason "HTTPS" is special is because the web is so popular. There are lots of other TCP/IP based protocols that ride on top of TLS. For example, TLS is used by [FTPS](#) and [secure extensions to SMTP](#). It's certainly better to use TLS than inventing your own solution. Additionally, you'll benefit from a protocol that has withstood careful [security analysis](#).

... And We're Done!

The very readable [TLS RFC](#) covers many more details that were missed here. We covered just one single path in our observation of the 220 millisecond dance between Firefox and Amazon's server. Quite a bit of the process was affected by the `TLS_RSA_WITH_RC4_128_MD5` Cipher Suite selection that Amazon made with its ServerHello message. It's a reasonable choice that slightly favors speed over security.

As we saw, if someone could secretly factor Amazon's "n" modulus into its respective "p" and "q", they could effectively decrypt all "secure" traffic until Amazon changes their certificate. Amazon counter-balances this concern this with a short one year duration certificate:

Validity	
Issued On	8/26/2008
Expires On	8/27/2009

One of the cipher suites that was offered was "TLS_DHE_RSA_WITH_AES_256_CBC_SHA" which uses the [Diffie-Hellman key exchange](#) that has a nice property of "[forward secrecy](#)." This means that if someone

cracked the mathematics of the key exchange, they'd be no better off to decrypt another session. One downside to this algorithm is that it requires more math with big numbers, and thus is a little more computationally taxing on a busy server. The "Advanced Encryption Standard" ([AES](#)) algorithm was present in many of the suites that we offered. It's different than RC4 in that it works on 16 byte "blocks" at a time rather than a single byte. Since its key can be up to 256 bits, many consider this to be more secure than RC4.

In just 220 milliseconds, two endpoints on the Internet came together, provided enough credentials to trust each other, set up encryption algorithms, and started to send encrypted traffic.

And to think, all of this just so Bob can buy milk.

POSTED BY JEFF MOSER AT [8:57 AM](#)

80 COMMENTS:

 Drew said...

Great post, Jeff. I'm curious to know if there is a way to disable certain algorithms in FF? Could you cut down that list of 34 algorithms? What will Amazon choose if you make FF refuse to use TLS_RSA_WITH_RC4_128_MD5?

[JUNE 10, 2009 10:01 AM](#)

 Jeff Moser said...

Drew: You can disable cipher suites by typing "about:config" in your address bar and then filter for "RSA"

You'll see a huge list, including "security.ssl3.rsa_rc4_128_md5", if you set this to false and then go back to Amazon.com, you'll see that it now picks "TLS_RSA_WITH_AES_256_CBC_SHA"

Amazon is willing to spend extra time using 256 bit AES, but they still really don't like the expensive Diffie-Hellman key exchange :)

[JUNE 10, 2009 10:06 AM](#)

 Eddie P said...

Thanks.. now i feel as dumb as a rock... excuse me while i go stare blankly at the walll

[JUNE 10, 2009 10:18 AM](#)

 Will Shaver said...

This is a fantastic post. Thanks for taking the time to write it!

[JUNE 10, 2009 10:26 AM](#)

 robburke said...

Really superb post -- tremendously informative and an engaging read!! Thank you
Jeff.

JUNE 10, 2009 10:56 AM

 Anonymous said...

I learned something, thank you!

JUNE 10, 2009 11:11 AM

 Anonymous said...

Thanks for this post, Jeff. Great stuff.

JUNE 10, 2009 11:53 AM

 Jim C said...

I love it when somebody like yourself takes the time to make an ambitious post like
this one - thanks for contributing this body of information to the searchable web.

- you've likely helped more people than you think you have...

JUNE 10, 2009 12:00 PM

 sriram srinivasan said...

Thanks much for the lovely presentation. This is great work.

JUNE 10, 2009 12:23 PM

 Eric Frenkiel said...

Fantastic post, Jeff! Your ability to clearly explain difficult concepts is unmatched
=> I look forward to dropping by more often.

JUNE 10, 2009 12:28 PM

 Anonymous said...

Thanks very much for taking the time to post this article.. Truly fascinating and
informative read.

JUNE 10, 2009 12:29 PM

 Mr T's Fashion Consultant said...

Great article, very informative. I like Wireshark too. Can you tell us how you
launched Wireshark (command line options or whatever) to get the captured data?
Thanks for any info.

JUNE 10, 2009 12:39 PM

 Adam said...

At least with newer versions of FireFox, you left out an important part of many new
ecommerce HTTPS connections - OSCP verification of the validity of the certificate

from the CA itself, in real time.

The particular Amazon cert you used didn't make use of this (only the new EV certs do), but for the 'fully secure' new status bar and color change in FF3+ and IE8, you have to have an EV cert with valid OCSP responder.

JUNE 10, 2009 12:41 PM

 Gabe Sumner said...

> If nothing else, this extension should allow an extra week or so of IPv4 addresses.

:) Made me chuckle!

JUNE 10, 2009 12:44 PM

 chkno said...

Note: [RFC 4217 FTP with TLS](#) is more often referred to as FTPS rather than SFTP. SFTP is more commonly used to refer to the [SSH File Transfer Protocol](#).

JUNE 10, 2009 12:57 PM

 John Doe said...

DH is not preferred because its costlier, it only does the key-exchange so authentication still needs to be done using DSA or RSA (hence called TLS_DHE_RSA_WITH_AES_256_CBC_SHA), while RSA can be used for both key-exchange and auth. DH additionally might require 'server-key-exchange' SSL messages. Lastly, most crypto-cards support a full RSA handshake operation which can't be used for DH.

JUNE 10, 2009 12:58 PM

 C.Santini said...

Thank you! Great post!!
my little contribute for RSA understanding:
[RSA encryption by hand](#) :)

JUNE 10, 2009 1:22 PM

 Anonymous said...

Thank you.

JUNE 10, 2009 1:30 PM

 Tony Oppenheim said...

Wonderful post! Coincidentally I've just been reading Steven Levy's great book "Crypto: How the Code Rebels Beat the Government Saving Privacy in the Digital Age" which tells the backstory of how RSA and many of the other encryption methods we now use came to be.

Thanks for posting!

JUNE 10, 2009 1:39 PM

 Anonymous said...

I read about week ago about SRP protocol. It is so brilliant. It would be really nice, if it will be supported in browsers and web servers.

Nice post.

JUNE 10, 2009 2:24 PM

 Anonymous said...

Excellent write up. Thank you for putting this together!!

JUNE 10, 2009 2:25 PM

 Anonymous said...

What an excellent walkthrough. I was geekily exhilarating. You took the time to annotate this beautifully so the many might enjoy it. Wunderbar.

JUNE 10, 2009 3:37 PM

 Anonymous said...

always mind your p's and q's

JUNE 10, 2009 3:42 PM

 JeremyH said...

Nice post. One other thing that goes on even before the HTTPS connect is a DNS lookup to connect to a server called `www.amazon.com`. Mozilla will first see if '`www.amazon.com`' is in the DNS cache and if not will start to recursively resolve the domain name, starting at the root ('.'), then the TLD ('.com.'), then the domain ('amazon.com.'). It all happens before the initial TCP connection. This looks like this:

```
; <<>> DiG 9.4.2 <<>> +trace www.amazon.com
;; global options: printcmd
. 297443 IN NS J.ROOT-SERVERS.NET.
. 297443 IN NS L.ROOT-SERVERS.NET.
. 297443 IN NS D.ROOT-SERVERS.NET.
. 297443 IN NS G.ROOT-SERVERS.NET.
. 297443 IN NS I.ROOT-SERVERS.NET.
. 297443 IN NS K.ROOT-SERVERS.NET.
. 297443 IN NS M.ROOT-SERVERS.NET.
. 297443 IN NS B.ROOT-SERVERS.NET.
. 297443 IN NS F.ROOT-SERVERS.NET.
. 297443 IN NS H.ROOT-SERVERS.NET.
. 297443 IN NS C.ROOT-SERVERS.NET.
. 297443 IN NS A.ROOT-SERVERS.NET.
```

. 297443 IN NS E.ROOT-SERVERS.NET.

:: Received 512 bytes from 68.87.71.226#53(68.87.71.226) in 32 ms

com. 172800 IN NS a.gtld-servers.net.

com. 172800 IN NS b.gtld-servers.net.

com. 172800 IN NS c.gtld-servers.net.

com. 172800 IN NS d.gtld-servers.net.

com. 172800 IN NS e.gtld-servers.net.

com. 172800 IN NS f.gtld-servers.net.

com. 172800 IN NS g.gtld-servers.net.

com. 172800 IN NS h.gtld-servers.net.

com. 172800 IN NS i.gtld-servers.net.

com. 172800 IN NS j.gtld-servers.net.

com. 172800 IN NS k.gtld-servers.net.

com. 172800 IN NS l.gtld-servers.net.

com. 172800 IN NS m.gtld-servers.net.

:: Received 492 bytes from 199.7.83.42#53(L.ROOT-SERVERS.NET) in 60 ms

amazon.com. 172800 IN NS udns1.ultradns.net.

amazon.com. 172800 IN NS udns2.ultradns.net.

:: Received 116 bytes from 192.35.51.30#53(f.gtld-servers.net) in 102 ms

www.amazon.com. 887 IN NS ns-923.amazon.com.

www.amazon.com. 887 IN NS ns-921.amazon.com.

www.amazon.com. 887 IN NS ns-912.amazon.com.

www.amazon.com. 887 IN NS ns-911.amazon.com.

:: Received 180 bytes from 204.74.101.1#53(udns2.ultradns.net) in 20 ms

www.amazon.com. 60 IN A 72.21.207.65

:: Received 48 bytes from 72.21.192.209#53(ns-921.amazon.com) in 23 ms

JUNE 10, 2009 4:20 PM

 spdalton said...

Brilliant post! Was a great way to see real application for the introduction to networking course I'm taking. Thanks!

JUNE 10, 2009 4:54 PM

 Anonymous said...

Great post! Learnt something new today. Thanks.

JUNE 10, 2009 5:31 PM

 Anonymous said...

Great post!

JUNE 10, 2009 6:32 PM

 Anonymous said...

FTP over SSL is called FTPS. There's enough confusion already regarding SFTP vs. FTPS so please fix your mistake.

JUNE 10, 2009 7:10 PM

 Vicente said...

Nice article!

JUNE 10, 2009 9:57 PM

 Jan Michael Yu said...

Awesome post. My mind went blank halfway through but I'm saving this for a later read. I wish classes were taught like this in school.

JUNE 10, 2009 10:31 PM

 Jeff Moser said...

Will Shaver, robburke, Anonymous #1/#2/#3/#4/#5/#7/#10/#11, sriaram srinivasan, and Vicente: Thanks for the positive feedback! I really appreciate all of them. As I was writing this post over the past four weeks, I was worried that it'd be too long and no one would read it (even after I cut a few thousand words). It's been very encouraging to see the response.

Eddie P: That wasn't the intent -- hopefully TLS is more exciting than staring at a wall :)

Jim C: Thanks for the encouragement. My hope is that this might encourage others to see the beautiful protocols that underly everyday Internet usage.

Eric Frenkiel - It'd be great to see you back. Feel free to click the "Subscribe" buttons on the right hand side (either via RSS or email)

Mr T's Fashion Consultant: I downloaded Wireshark from wireshark.org, installed it, and then ran it. I then clicked "Capture", then "Interfaces" and clicked "Start" on the interface that was generating packets. Once you do this, it'll start to capture all traffic. I then ran the debug Firefox build (Minefield) and went to Amazon. After all this I clicked "Capture", then "Stop". In the protocol column, I found the first "TLSv1" protocol packet and then right-clicked and then clicked on "Follow TCP Stream." Everything else came from looking at the specific packets and seeing the Wireshark decoding of the bytes.

Adam: Very true on OCSP's, CRL's, and the EV extended checks.

Gabe Sumner: Glad you enjoyed it - I tried to keep things lighthearted :-)

chkno/Anonymous #12: You're right. I've gone ahead and made the change from SFTP to FTPS.

John Doe: Point taken on DH for just key exchange and RSA for authentication. This is honestly what I meant as I had linked to the appropriate Wikipedia articles. What I was trying to indicate was DH is nice from a "forward secrecy" perspective. The "downsides" if you can call them that, are more math == more CPU and you have ephemeral keys which means that a network admin has a harder time snooping on the traffic.

C.Santini: Thanks. What do you think of the math with huge numbers? :)

Tony Oppenheim: Sounds like an interesting book. Did anything in particular stick out from the backstory that you thought was interesting?

Anonymous #6: SRP is one of those hidden Internet gems. It's too bad it's not more popular (and therefore, available in more products).

Anonymous #8: Nice use of "geekily exhilarating" :)

Anonymous #9: I don't mind the p's and q's, just so long as they're prime.

JeremyH: Yep, the DNS query answer came in right before the Client Hello. One thing that I didn't cover here is DNS-SEC, which uses certificates to sign responses. It's not widely deployed (yet), but it helps to prevent against DNS cache poisoning.

spdaltan: Glad it was helpful. Computers offer the ultimate lab science. I'd highly encourage you to play around with Wireshark to see what's happening on your own network. I think it's fun to see what's actually going on.

Jan Michael Yu: Don't worry if it takes a little for it to sink in. It took me a few weeks of exploring to feel comfortable with it. Feel free to ask follow-up questions if something doesn't make sense.

JUNE 10, 2009 11:20 PM

 Dom said...

You no doubt have one already but you could probably submit this as a Masters paper and come away with Honors from most top universities. Nice work!

JUNE 11, 2009 12:13 AM

 sanjaya said...

wow.. this is amazing though takes time to get into bits and bytes, but now have some idea about how protocol works.

JUNE 11, 2009 12:42 AM

 Anonymous said...

Great post! Thanks dude!

JUNE 11, 2009 1:14 AM

 pats said...

Everytime i read that secure traffic flows thro' a tunnel, are all these data packets goes through 'tunnel' ? ... my defination of tunnel is something that connects client and server and no one can read actual data flowing throgh since it is encrypted ... if wrong correct me..

JUNE 11, 2009 2:23 AM

 Tony Oppenheim said...

@Jeff Moser: "Did anything in particular stick out from the backstory that you thought was interesting?"

Mostly that the NSA did everything they could to keep RSA and other private sector encryption methods from coming into use. It's thru the efforts of really just a few dedicated/brilliant/eccentric/obsessed people that we have the level of security for commercial use we enjoy today. If it wasn't for them e-commerce as we know it wouldn't be happening.

Or at least that's how it seems from my reading. Though I'm no expert in this field.

JUNE 11, 2009 2:49 AM

 Anonymous said...

This is monumental work, Jeff. I would love to read more from you.

JUNE 11, 2009 2:50 AM

 Siu said...

Thank you for the post! Very useful!

I will need to print this to understand everythig.

JUNE 11, 2009 5:07 AM

 Anonymous said...

Thanks a lot this enlightening article. It is both detailed and technically correct, as well as understandable. Very well structured and written from a didactical point of view.

JUNE 11, 2009 5:32 AM

 Anonymous said...

Awesome article. Please make a pretty PDF version of it!

JUNE 11, 2009 6:04 AM

 John Doe said...

@Jeff - Should submit it to phrack. Would be real cool to have a phrack entry!

JUNE 11, 2009 6:10 AM

 Niranjan said...

Excellent post, very informative and easy to understand!

JUNE 11, 2009 6:13 AM

 Nguyen said...

such a good research :D

JUNE 11, 2009 6:44 AM

 earthgecko said...

Fascinating article, thanks. Found the RSA part very interesting. It does just go to demonstrate how much data is flying around... and how quick it does fly around.

JUNE 11, 2009 7:29 AM

 Anonymous said...

You are totally awesome for this post, great work!!! Oh, and I almost forgot, Firefox rules! Kudos

JUNE 11, 2009 7:39 AM

 Pdizzle said...

Great post thanks a lot. This ties in well with the SSCP class I just took.

JUNE 11, 2009 7:40 AM

 Dave said...

I read you article while wearing my Three Wolf Moon T-Shirt (link below), and I was actually able to understand the math ! It's amazing, because the last math class I had was in my senior year of elementary school. Thanks to your wonderful explanation of the SSL handshake, I can now encrypt and decrypt human conversation in real time. I will try using what I have learned here to decrypt the mathematical language of the wolves in order to discover their plan to reach the moon again before the Russians can beat us there. Thank you !

[Three Wolf Moon T-Shirt](#)

JUNE 11, 2009 8:16 AM

 Anonymous said...

Interesting hack. What are you going to do with this knowledge? Hack Firefox users who access Paypal?

JUNE 11, 2009 8:20 AM

 Jeremy said...

Great post, thank you very much. I'm a graduate student in computer science, but my focus is databases so I don't often get to see the networking end of the wire.

JUNE 11, 2009 8:28 AM

 Jeff Moser said...

Sanjaya, Anonymous #12/#13/#14/#15, Siu, John Doe, Niranjan, Nguyen, earthgecko, Pdizzle, Jeremy: Thanks for the kind words and spending the time to comment. I appreciate it.

Dom: That'd be nice, but I'm sure it's a bit tougher than that :) No masters for me, just a Bachelor's in Computer Science and Math. I'm just a guy that likes exploring with Wireshark.

Pats: Tunnels allow normally unencrypted traffic to appear as encrypted when on the wire. SSH is a popular example. You can sort of think of TLS/SSL as a secure "tunnel" for HTTPS traffic.

Tony Oppenheim: Thanks for sharing that. I read on Wikipedia that Clifford Cocks invented a similar system to RSA 4-5 years earlier for the UK intelligence agency GCHQ, but didn't really implement or deploy it. I always [enjoy the history](#) behind the technology.

Anonymous #15: Any advice on what you'd like a "prettier PDF version" to look like? Like a magazine (two-columns?)

Dave: glad the shirt gave you its claimed super powers :) It'd be nice if practical math like RSA was taught in high school. I think it'd be more engaging... at least a few weeks of it.

Anonymous #16: I have no hacking intents... it's more of just seeing all these protocols that are used everyday as having an elegance to them.

JUNE 11, 2009 8:46 AM

 Anonymous said...

Wow,
this explanation ROCKS!

Can you please do the same for SIP/RTP kind of traffic (in combination with wireshark)?

JUNE 11, 2009 8:56 AM

 Sasi said...

wow !! Clear & Well explained ... Thanks Jeff.

JUNE 11, 2009 9:09 AM

 Anonymous said...

Great post, thanks! BTW this post has made it to the front page of digg.com

JUNE 11, 2009 9:45 AM

 Waan said...

Great post,Jeff.... Thank you!

JUNE 11, 2009 10:19 AM

 Anonymous said...

I think i learned more from this than my Cisco Security class... and we spent like 3 weeks on this...

JUNE 11, 2009 10:25 AM

 JaKe said...

A PDF would be great! Just a simple straight from the page print of the text, graphics and hyperlinks. Nothing fancy, like two-column, required, the graphics won't let you anyway ;).

JUNE 11, 2009 10:50 AM

 CVillalobos said...

Okay. That was cool thanks for the info. Detailed and a fun way to start the morning.

JUNE 11, 2009 11:00 AM

 Anonymous said...

Great Info, great article. Thank you so much..

JUNE 11, 2009 11:25 AM

 Anonymous said...

Great Article, very intresting and useful Jeff!!!

JUNE 11, 2009 12:02 PM

 Ankur said...

Good stuff, Jeff! Truly informative.

JUNE 11, 2009 1:31 PM

 Television Online said...

Possibly the most intense milliseconds that you'll never actually notice.

JUNE 11, 2009 3:05 PM

 Anonymous said...

wow, you have no life

JUNE 11, 2009 3:41 PM

 k@beza said...

Oh my god !! Oh my god !! You can see the Matrix !!
Fuc*ing insaneeeeeeeeeeeeeee !

JUNE 11, 2009 3:43 PM

 sulumits retsambew said...

very deep explanation , thank you

JUNE 11, 2009 4:16 PM

 Peter Vatistas said...

Awesome post! I'm a big fan of Wireshark.

JUNE 11, 2009 4:27 PM

 David Thomson said...

I read one comment about SRP. I implemented this in a web browser called SupraBrowser

<http://sourceforge.net/projects/suprabrowser>

It supports encryption of all traffic using 3DES (with pluggable algorithms), based on a Diffie-Hellman key exchange augmented by the result of a bi-directional SRP authentication (<http://srp.stanford.edu>).

JUNE 11, 2009 4:27 PM

 tarmo said...

Good stuff. Well explained.

JUNE 11, 2009 4:38 PM

 Anonymous said...

Thanks a lot for the post, it was very informative.

JUNE 11, 2009 6:15 PM

 Rajat Swarup said...

Some additional information:

For really large primes p, q such that $n = pq$ the values of the public exponent e and private exponent d are computed as :

$$e \times d = 1 \bmod (p-1) \times (q-1)$$

The value $(p-1) \times (q-1)$ is also called as the Euler's totient function. :-)

JUNE 11, 2009 6:16 PM

 Rajat Swarup said...

Btw...I forgot to mention...this was an awesome post! :-)

JUNE 11, 2009 6:17 PM

 Anonymous said...

Thanks for a thorough and ... just great post. Awesome!

JUNE 11, 2009 6:24 PM

 Anonymous said...

@Mr.Moser... So for example, the 'client_write_mac_secret' is an associative array, whose hash_value's value is part or all of the key_block? Am I understanding that part correctly?

By the way, great post! And I don't mean great in the way in which it is normally cheaply used in our day to day activities. Unless blogger.com goes down, I think this post will be indexed highly on the internets after you and I are long gone. You might want to back this info up in a couple other places chief!

JUNE 12, 2009 12:21 AM

 Anonymous said...

Outstanding article. In the STARTTLS E-Mail world it is neccessary to force Wireshark to interpret the SMTP stream as SSL, as they don't seem to have found a robust way to detect the transition from plain to encrypted yet. Once this is done, everything works just the same as described here.

JUNE 12, 2009 12:25 AM

 Anonymous said...

@Mr.Moser... just for clarification on a typo in my last post a few moments ago, when I said "hash_value"'s value ...i meant "hash_size"'s value. Please clarify if you have a minute. If you explain this one, I will understand the others. It's an associative array, correct? But how is that value populated or determined. I'm foggy on that.

JUNE 12, 2009 12:28 AM

 Anonymous said...

wow,

this article is perhaps a reply to a loosely based question 'dad, where did I come from?'

and your explanation is a very good elaboration of facts and figures.

great effort and good job.

well done and yes I am enlightened.

JUNE 12, 2009 2:38 AM

 **Rajasekhar** said...

I think this is one of most clearly written blog-articles I have read in recent times. Thanks a ton for the tech details. I am yet to get a full grasp of it, but when I do, it will definitely add to my tech armor :) Great work!

JUNE 12, 2009 3:27 AM

 **Suebtas** said...

Awesome tutorial SSL article.

JUNE 12, 2009 3:29 AM

 **Mary-Frances Beesorchard** said...

Thank You! This article makes such a clear study-plan for beginners.

JUNE 12, 2009 7:23 AM

 **Jeff Moser** said...

Anonymous #19/#20/#21/#22/#24/#25/#29, Sasi, Waan, CVillalobos, Ankur, sulumits retsambew, Peter Vatistas, tarmo, Rajasekhar, Suebtas, Mary-Frances Beesorchard: Thanks for the kind comments and for stopping by.

Anonymous #18: SIP/RTP is unencrypted and Wireshark handles it fairly well on its own. I'd encourage you to download Wireshark and just take a few seconds of captures and then explore it in Wireshark. That'd probably be more insightful.

JaKe: Per your request, I created [a PDF version of this post](#). Does this work for you?

Television Online: Yeah, I typically don't notice them most times, but now I respect those milliseconds more.

David Thomson: Glad to see use of SRP.

Rajat Swarup: Yep. If you check out the links I had in the RSA section, you'll see that I linked to the totient function along with Fermat's Little Theorem, the Chinese Remainder theorem, and the original RSA paper that outlines this as well.

Anonymous #26/#28: The "client_write_MAC_secret" are the first "hash_size" bytes (in this case, 16 bytes since it's MD5) of "key_block". The "key_block" is the output of the Pseudo Random Function (PRF) that I mentioned. You don't need to hash it any more. This is the "key" of the keyed-HMAC version of MD5 that I described in the post.

For completeness:

client_write_MAC_secret are bytes 0..[SecurityParameters.hash_size] - 1 of the PRF generated "key_block"

server_write_MAC_secret are bytes [SecurityParameters.hash_size] ..

[SecurityParameters.hash_size + SecurityParameters.hash_size] - 1 of the PRF generated "key_block"
client_write_key are bytes [SecurityParameters.hash_size + SecurityParameters.hash_size]..[SecurityParameters.hash_size + SecurityParameters.hash_size + SecurityParameters.key_material_length] -1 of the PRF generated "key_block"
server_write_key are bytes [SecurityParameters.hash_size + SecurityParameters.hash_size + SecurityParameters.key_material_length]..
[SecurityParameters.hash_size + SecurityParameters.hash_size + SecurityParameters.key_material_length + SecurityParameters.key_material_length] - 1 of the PRF generated "key_block"

Does that help? Feel free to check out the RFC I linked to if you want even more details.

Anonymous #27: It's a little trickier to detect a transition. A good heuristic would be to look for the the TLS version bytes and content types... along with seemingly random data.

JUNE 12, 2009 8:32 AM

 Jimmy In Madrid (J.I.M) said...

Amazing! great work there!

JUNE 12, 2009 9:24 AM

[Post a Comment](#)

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)