The First Few Milliseconds of an HTTPS Connection

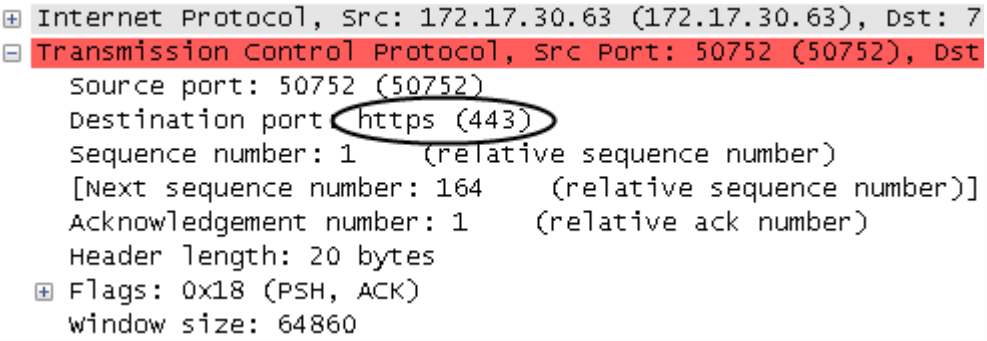
Jun 10, 2009

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.

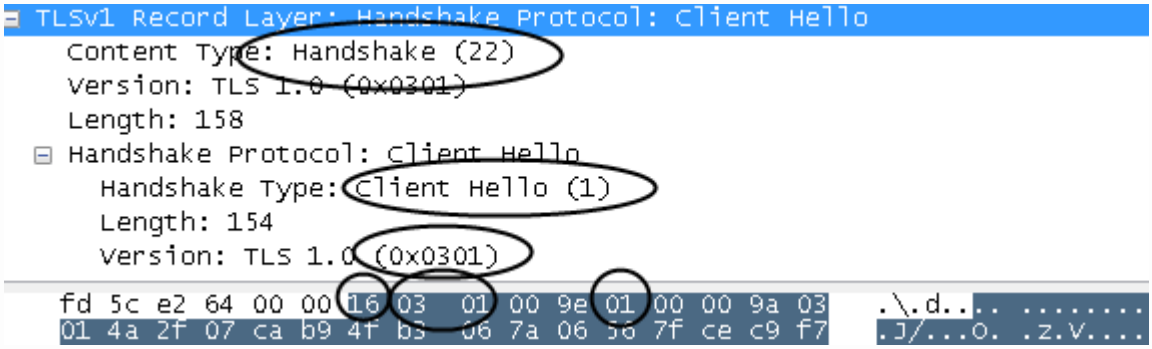
By agreement of RFC 2818, Firefox knew that “https” meant it should connect to port 443 at Amazon.com:



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

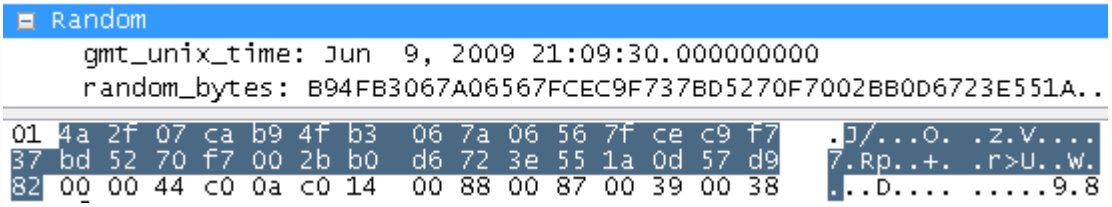
TLS wraps all traffic in “records” of different types. We see that the first byte out of our browser is the hex byte 0x16 = 22 which means that this is a “handshake” record:



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.

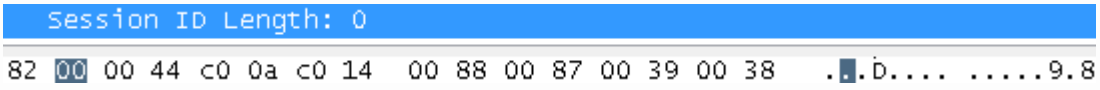
The handshake record is broken out into several messages. The first is our “Client Hello” message (0x01). There are a few important things here:

- Random:

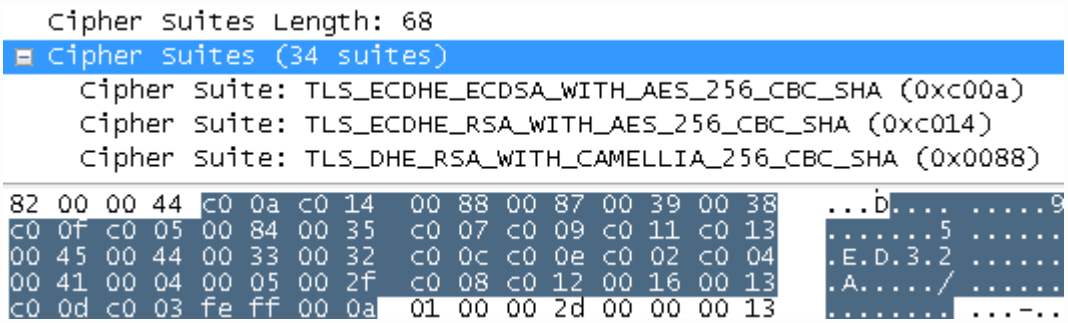
  There are four bytes representing the current Coordinated Universal Time (UTC) in the Unix epoch format, which is the number of seconds since January 1, 1970. In this case, 0x4a2f07ca. It’s followed by 28 random bytes. This will be used later on.

- 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:

```
Extension: server_name
c0 0d c0 03 fe ff 00 0a  01 00 00 2d 00 00 00 13   ........ ...-....
00 11 00 00 0e 77 77 77  2e 61 6d 61 7a 6f 6e 2e   .....www .amazon.
63 6f 6d 00 0a 00 08 00  06 00 17 00 18 00 19 00   com..... ........
```

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.

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):

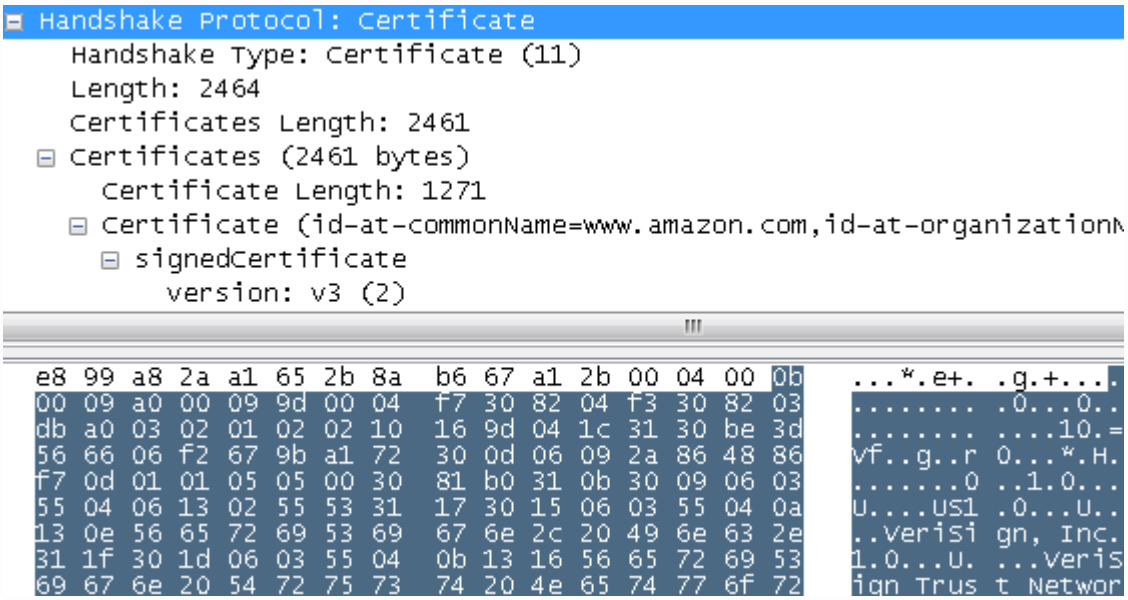
```
Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 70
    Version: TLS 1.0 (0x0301)
  Random
      gmt_unix_time: Jun  9, 2009 21:09:30.000000000
      random_bytes: 986BE7A3ACDD547D038235895ABA467EED6EB07CD46E5E8A..
    Session ID Length: 32
    Session ID: ACD07167996609E3584D15C2D43596D8AB93CFA9E899A82A...
    Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
```

```
16 03 01 09 f2 02 00 00  46 03 01 4a 2f 07 ca 98   ........ F..J/...
6b e7 a3 ac dd 54 7d 03  82 35 89 5a ba 46 7e ed   k....T}. .5.Z.F~.
6e b0 7c d4 6e 5e 8a 19  9e 0c 13 20 ac d0 71 67   n.|.n^.. ... ..qg
99 66 09 e3 58 4d 15 c2  d4 35 96 d8 ab 93 cf a9   .f..XM.. .5......
e8 99 a8 2a a1 65 2b 8a  b6 67 a1 2b 00 04 00 0b   ...*.e+. .g.+....
```

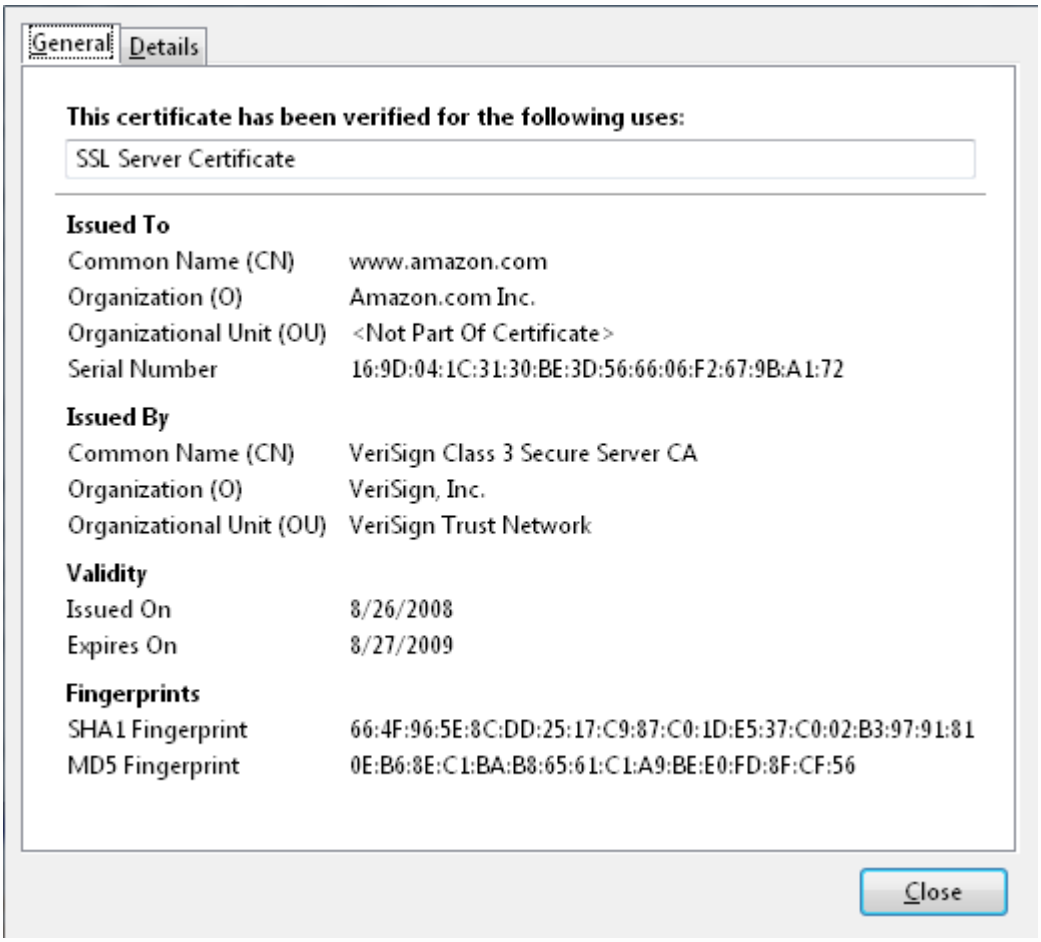
   - We get the server's four byte time Unix epoch time representation and its 28 random bytes that will be used later.
   - A 32 byte session ID in case we want to reconnect without a big handshake.
   - Of the 34 cipher suites we offered, Amazon picked "TLS_RSA_WITH_RC4_128_MD5" (0x0004). This means that 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.

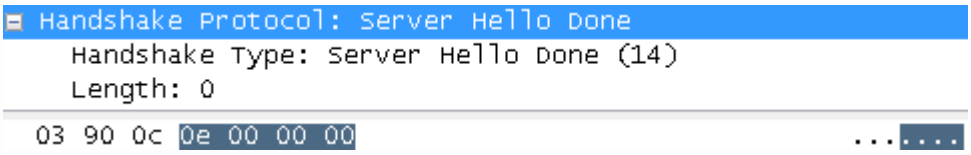
2. Certificate Message (11):



   - 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):



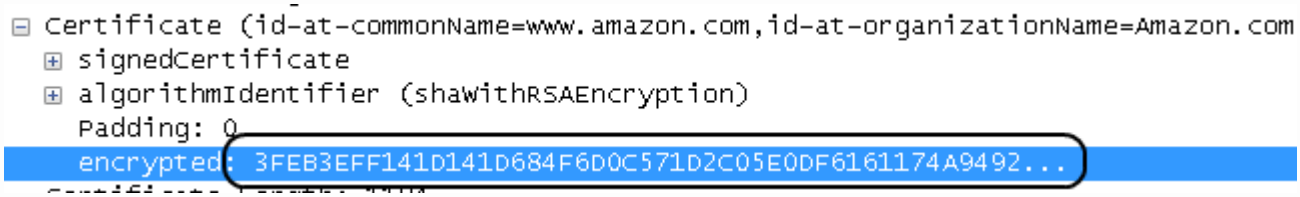
   - This is a zero byte message that tells the client that it’s done with the “Hello” process and indicate that the server won’t be asking the client for a certificate.

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 *R*ivest, Adi *S*hamir, and Len *A*dleman 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*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 ≡ 2 (PM) (mod 12 hours). The recipient knows “d” which allows them to invert the message to recover the original message:

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

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:

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

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

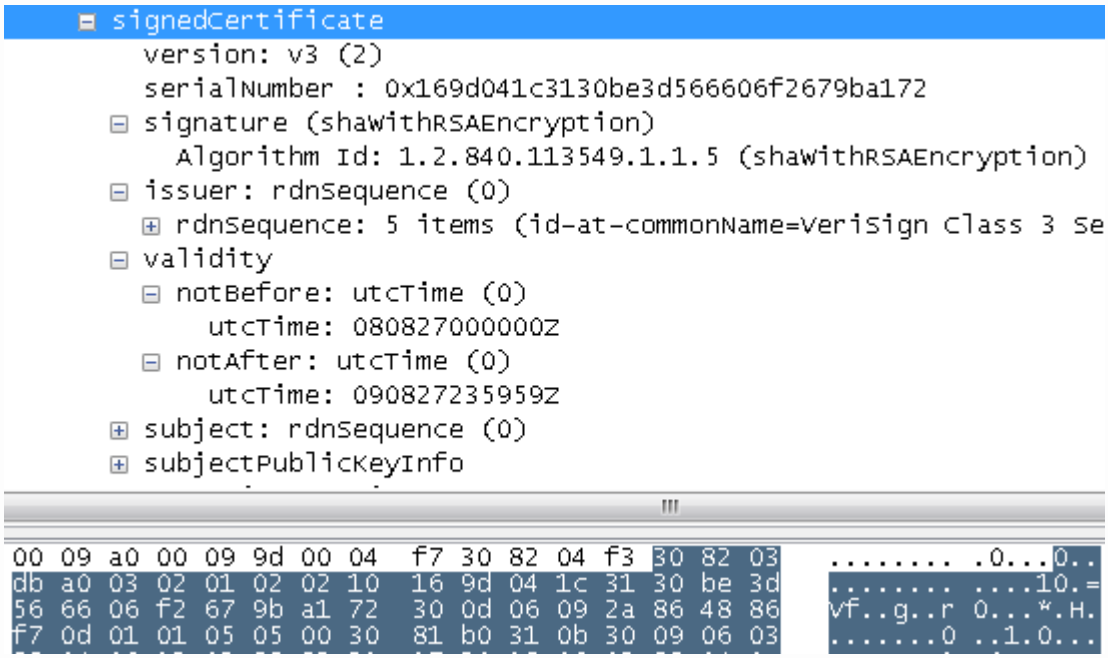
The big thing to keep in mind with RSA in the real world is that all of the numbers involved have to be *big* to make things really hard to break using the best algorithms that we have. How big? Amazon.com’s certificate was “signed” by “VeriSign Class 3 Secure Server CA.” From the certificate, we see 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
```

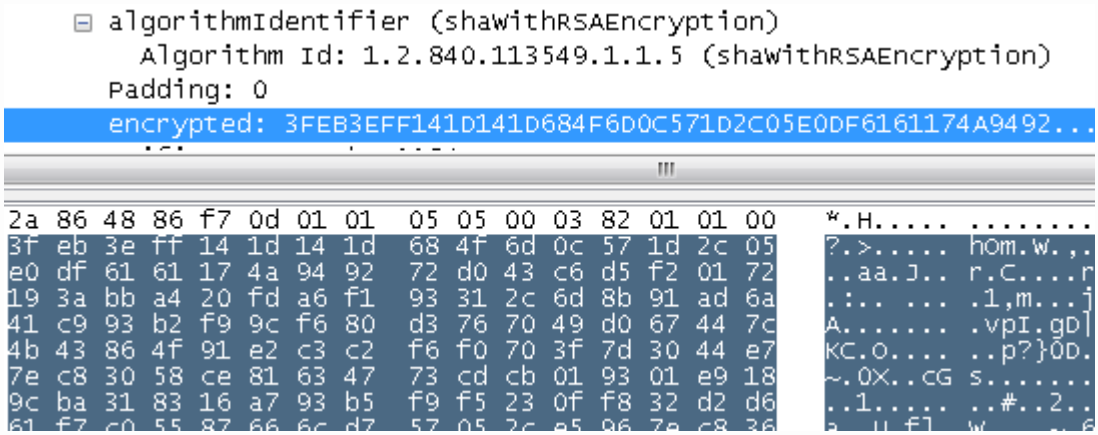
```
3091993360 6376719337 6644159792 1249204891 7079005527 7689341573 9395596650
5484628101 0469658502 1566385762 0175231997 6268718746 7514321
```

(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:

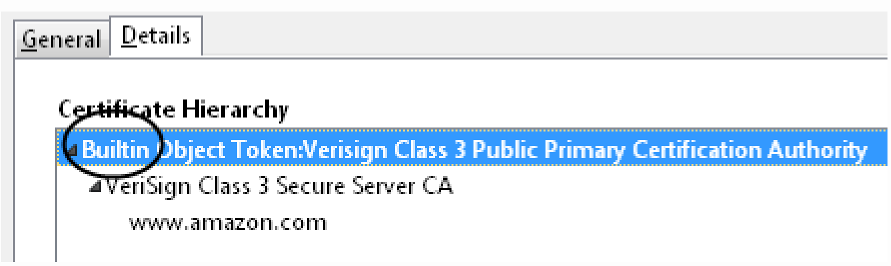
```
0001FFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFF00302130
0906052B0E03021A 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 f0 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.

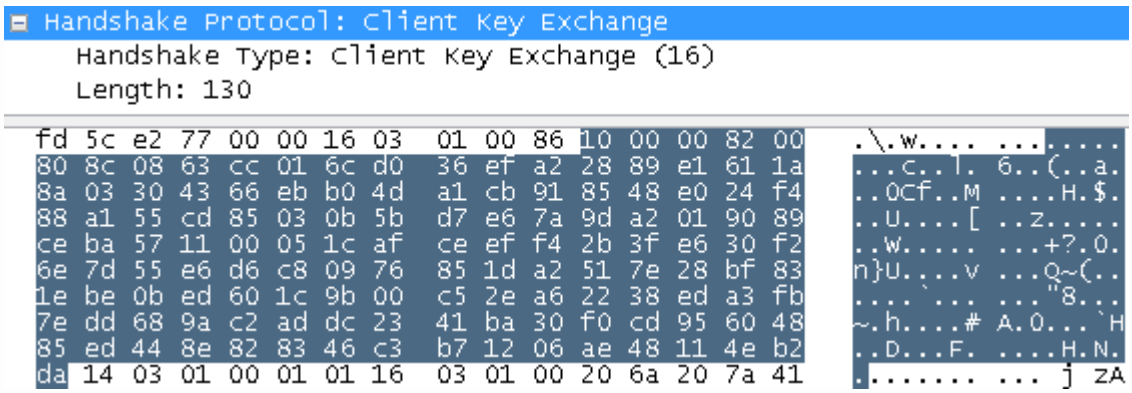
Again, Firefox makes it hard to see these random values. I had to insert debugging statements into the padding function to see what was going on:

```
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);
```

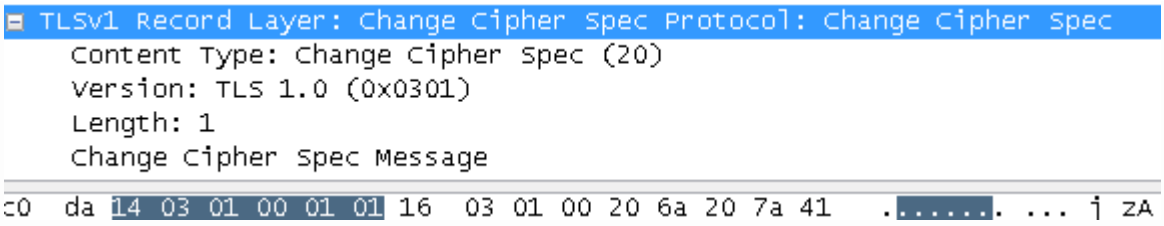
In this session, the full padded value was:

```
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 D0 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 F0 64 22 57 7E E1 06 7A 3B
```

Firefox took this value and calculated "C ≡ M^e (mod 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 be 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 *216* + *2*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 B0
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:

```
HMAC_MD5(Key, m) = MD5((Key ⊕ opad) ++ MD5((Key ⊕ ipad) ++ m)
```

(The ⊕ 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.

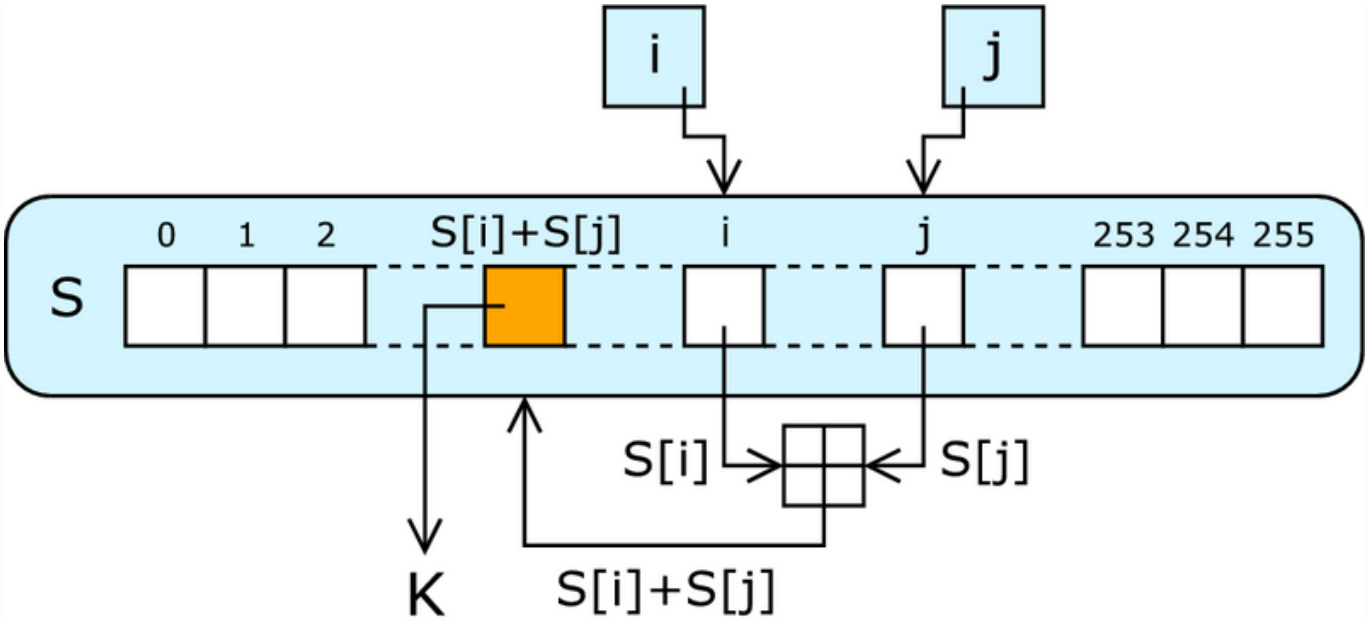
All that's left is to encrypt these bytes.

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 F0 AE CB C4 …", we'll get what appears in the encrypted portion that we can see in Wireshark:

```
⊟ TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 32
    Handshake Protocol: Encrypted Handshake Message

    da 14 03 01 00 01 01 16  03 01 00 20 6a 20 7a 41   ........ ... j zA
    66 0b d6 6c f7 be 3f 41  de d0 28 c2 4a 61 2d b5   f..l..?A ..(.Ja-.
    30 16 73 c1 e6 31 7a 42  22 2c 2f 96               0.s..1zB ",/.
```

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!

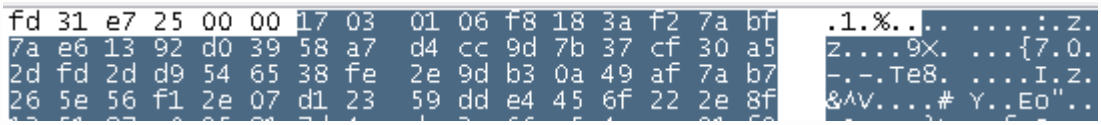
Now, 220 milliseconds after we started, we're finally ready for the application layer. We can now send normal HTTP traffic that'll be encrypted by the TLS layer with the RC4 write instance and decrypt traffic with the server RC4 write instance. In addition, the TLS layer will check each record for tampering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

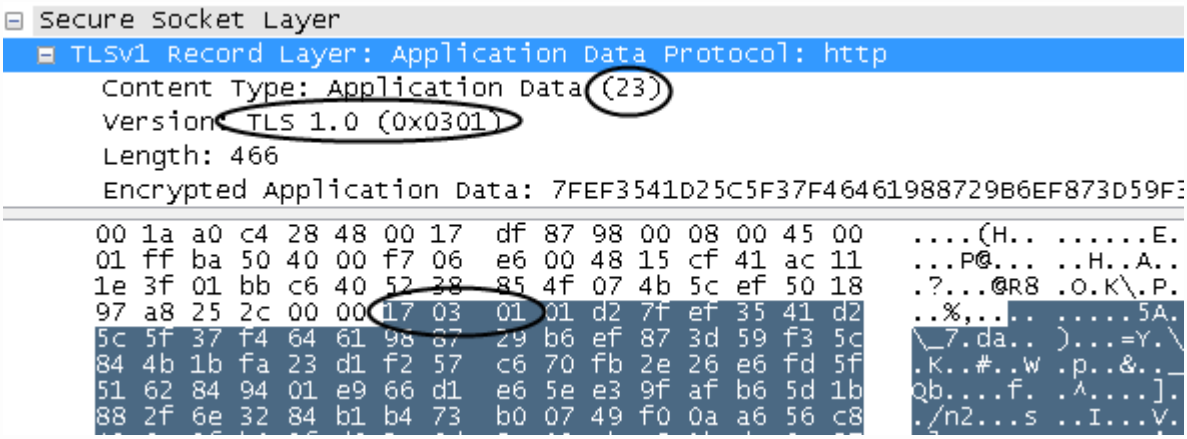
```
GET /gp/cart/view.html/ref=pd_luc_mri HTTP/1.1
Host: www.amazon.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.10) Gecko/20
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: 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:

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.

UPDATE: I wrote a program that walks through the handshake steps mentioned in this article. I posted it to GitHub.

updated version any time soon? Thanks for all the hard work you've put into this.

1 ^ | v • Reply • Share ›

Jeff Moser Mod ➚ Hang • 6 years ago

A lot has changed due to various cryptographic attacks in the past few years: RC4 is no longer used due to initialization weaknesses and RSA is no longer recommended because of advancing factoring attacks. The industry has shifted towards 256 bit (or higher) elliptic curve Diffie-Hellman (ECDHE) for the public key portion (and to get forward secrecy) and authenticated encryption such as AES in Galois Counter Mode (GCM) for the symmetric portion. For example, if you go to Amazon using a modern browser today, you'll see this ECDHE + AES 128 in GCM.

TLS 1.2 got rapid adoption after problems with RC4 and CBC mode ciphers. TLS 1.3 will likely tighten up some minor things as well as strongly advocate elliptic curve ECDHE + AES GCM and perhaps a stream cipher option such as ChaCha20 + Poly1305. You're starting to see this now in Chrome. Any site that doesn't support these newer suites is marked as using "obsolete cryptography."

It'd probably take a fair bit of time to update this post, but it's something I'd consider. Upvote this comment if you're interested.

53 ^ | v • Reply • Share ›

Aleksandar Kostadinov ➚ Jeff Moser • 3 years ago

And with EC### we are waiting for quantum computers to break it..

^ | v • Reply • Share ›

PotHix ➚ Jeff Moser • 6 years ago • edited

An update would be really great!

BTW, I really want to try your code but it doesn't have a README on how to run it. Does it still works?

Thank you for sharing your knowledge! :)

^ | v • Reply • Share ›

Jeff Moser Mod ➚ PotHix • 6 years ago

Thanks for your interes!. You can still use the TLS Analyzer if you clone the repo and open

it up using Visual Studio and click run or press F5. I'm currently using Visual Studio 2015 and had to do an automated conversion of the project when opening it, but otherwise it ran fine.

To be candid, it's a poorly designed UI that's a bit hard to use/understand. Most of the fun stuff happens by following along the code in the debugger.

1 ^ | v • Reply • Share ›

PotHix ➚ Jeff Moser • 6 years ago

Cool! Any chance to get it to run under Linux? :P

^ | v • Reply • Share ›

Jeff Moser Mod ➚ PotHix • 6 years ago

Maybe with Mono, but it's probably not worth it vs just browsing the code. If I did it again, I'd probably do it in TypeScript + React.

^ | v • Reply • Share ›

PotHix ➚ Jeff Moser • 6 years ago

Fair enough. Thanks! :)

^ | v • Reply • Share ›

Hang ➚ Jeff Moser • 6 years ago

Thank you for taking the time to reply...you are a very knowledgeable and thorough person. Would totally love to see an update at some point in the future. Thank you for taking the time to do this. You're awesome!

^ | v • Reply • Share ›

yvenu • 8 years ago

Nice explanation. Thanks.

1 ^ | v • Reply • Share ›

chetan kapoor • 3 years ago

This is very interesting and detailed explanation that you will find on internet at one place.

^ | v • Reply • Share ›

Marcos de Benedicto • 3 years ago

Excellent technical article, best explanation!

Excellent technical article, best explanation!

Reply • Share ›

刘曦光 • 3 years ago

Why does https need time stamp during handshaking?

Reply • Share ›

刘曦光 • 3 years ago

Unbelievable, every http request will follow by a hmac hash value to verify the package.

Reply • Share ›

Jay R Del • 3 years ago

Very very nice job. Well done. I hope you also had another post about Diffie-Hellman with AES GCM encryption.
Thank you

Reply • Share ›

Bartosz Deni • 4 years ago

Impressive

Reply • Share ›

常红亮 • 4 years ago

nice

Reply • Share ›

nEosAg • 4 years ago

This is great in-depth article, waiting for the update. Thanks for sharing!

Reply • Share ›

Müller Manfred • 4 years ago

If somebody has a packet capture of the connection setup described before, could he decrypt the whole following conversation? Let's say I am setting up a HTTPS connection using a public wi-fi hotspot and the operator of the hotspot captures all traffic. In this case HTTPS could not guarantee me any security. Is this assumption right?

Reply • Share ›

nullqubit ➚ Müller Manfred • 4 years ago

No because you can't decrypt unless you have the private key, which only the real server has.

1 Reply • Share ›

Müller Manfred ➚ nullqubit • 4 years ago

If this was true the browser could not decrypt the message either, as it does not have the private key.

Reply • Share ›

nullqubit ➚ Müller Manfred
• 4 years ago • edited

Both the browser and the server have a public-private key pair (4 keys total). Data encrypted using a public key can only be decrypted by the associated private key. The browser and server exchange public keys, so they use each other's key to encrypt. The browser can decrypt the message using its own private key (Because it was encrypted using the public key that was given to the server during the handshake). In the same way, the server can decrypt the message using its own private key. It is important to know that the private keys never go through the wire, so a man in the middle cannot decrypt the data.

1 Reply • Share ›

geek07 • 4 years ago

I had to control my laugh when you made those Firefox jokes but this "And to think, all of this just so Bob can buy milk." made me laugh so dam hard.

Reply • Share ›

Arunprashad Selvaraj • 4 years ago

Wow.The best explanation on TLS handshake i ever read

Reply • Share ›

griso • 5 years ago

Great article!

Reply • Share ›

Andreas Leeb • 5 years ago

Wow, thank you a lot! This helps me a lot, as I have to hold a presentation about HTTPS in school. I haven't found a source with so many details yet. Keep up the good work, it's amazing!

Reply • Share ›

Ketchup • 5 years ago

Amazing article. The explanation is most explicit :)

Thanks so much !

Reply • Share ›

Vishal Ranpariya • 5 years ago

Where we can get certificate Validity details "Issued on" and "expired on" in the wireshark pcap?

Reply • Share ›

Kegan Thorrez • 6 years ago

I noticed a couple problems. The biggest one is that you say integer factoring is a "tough problem" and link to the NP wikipedia page. It is true that integer factorization is in NP, but that does not mean it is tough. Every easy problem is also in NP so being in NP does not mean tough. NP Hard means tough, but integer factorization has not been proved to be NP Hard. There could be a fast integer factorization algorithm developed tomorrow (although extremely unlikely).

Also you say "48 byte (256 bit) pre-master secret".

And "216 + 216 = 64" which appears to be a markdown error and should say "2*16 + 2*16 = 64" but the asterisk was interpreted as italic.

Reply • Share ›

micman997 ➚ Kegan Thorrez • 6 years ago

yep, 48 byte = 256 bit?? i am wondering too. but still very nice and helpfull article.

Reply • Share ›

ankurkher • 6 years ago • edited

I may be asking a very basic question but just want to verify that "Root Certificate" also gets verified or not?

What I believe is, the root certificate which is sent to the user/client contains CA's(Root CA's) Public key. So, does the client contacts the root CA to ask root CA to validate itself to the client before client starts making a connection or not and if yes, how does it work?

Reply • Share ›

Jeff Moser Mod ➚ ankurkher • 6 years ago

The root CA is built into the web browser itself and is implicitly trusted. Thus, it provides an end to the trust chain. See the "But why should we trust that?" section in the post.

Reply • Share ›

ceclinux • 6 years ago

The best post I have ever seen related to HTTPS. Great explanation in every detail with convincing wireshark packet analysis, unpacking the HTTPS box and show how it works. Thanks.

Reply • Share ›

linoge • 6 years ago

This is really amazing.

Reply • Share ›

yvenu • 8 years ago

"client sends out because the certificate from Verisign has Amazon's public key. Thus, the client would use that public key (and not one an attacker generated)."

Hi Jeff,
Does this mean Certificate from Verisign (which is pre-loaded in to browser) will have public keys of all sites it signed.

Reply • Share ›

Jad Nehme ➚ yvenu • 6 years ago

no , maybe it has some certificates that were previously verified by the browser, for performance purposes, but it is not a necessety. Each time you receive a certificate from a new server, you verify it by decrypting it's signature with the issuer's public key (could be verisign or any other CertificationAuthority "CA") and by comparing the result with the hash of the information in the certificate.

Reply • Share ›

KP • 8 years ago

Great blog post this. It seems that the encryption of the http packets happens in the transport layer below it, so even if a site is running on https, can any browser extension or anyone who has access to the DOM manage to read the form fields like passwords or credit card info etc. before the encryption happens ? Have been wondering about that

Reply • Share ›

Tang Chaobin ➚ KP • 6 years ago

The SSL/TLS doesn't provide security for things like that, it is mainly designed against Man-in-the-Middle attack, which is very easy to happen. What you said about attacker sniffing things on your webpage falls into a category of hack

known as cross site request forgery, where a program (typically javascript, yes, like a browser extension) can cause the unwanted actions to be executed on a web page user trusts, e.g., the code can read your password and submit it to another place.

 • Reply • Share ›

Anonymous • 8 years ago

Very interesting article, and I imagine it will be looked at many times in light of the recent US and UK spy agencys claiming to be able to crack HTTPS.

My questions is, where does the private key reside in HTTPS transactions, if its on my local machine, why cant i see it/where its stored.

Many thanks.

 • Reply • Share ›

rudie dirkx • 8 years ago

How long will a handshaked session last? The endpoints didn't agree on a TTL/Keep-alive thing... Is it until either point denies the current encryption, which will trigger a new handshake? What about HTTP? Keep-alive is usually 300s, which means a new socket after that. New handshake or reuse previous?

Thanks. Great to finally see in such detail what the heck my browser is doing all the time.

 • Reply • Share ›