Crypto in practice: What happens when you connect to a secure web server

Normal communication between a web browser and a web server is carried out through an exchange of messages in HTTP (Hypertext Transfer Protocol). These messages are translated into and from the network packets used by the lower-level network protocols.

The Secure Sockets Layer (SSL), later renamed Transport Layer Security (TLS) is a protocol that sits between application layer protocols like HTTP and the lower levels. The first TLS messages exchanged create the secure channel between client and server, and the subsequent messages are encrypted versions of the application layer messages.

This demo presents a very detailed example of how this is carried out, using TLS version 1.2, the latest descendant of SSL. In it, you will see how all the techniques we have studied in class: pseudo-random number generation, public key encryption, symmetric encryption, hash functions, message authentication codes, and digital signatures, are actually deployed in practice.

The textbook provides a very brief account of how an earlier TLS version works. The inspiration for this demo is a blog post by Doug Moser, 'The first few milliseconds of an https connection' http://www.moserware.com/2009/06/first-few-milliseconds-of-https.html. Moser carries out essentially the same procedure shown here, with an older version of TLS and a different cipher for symmetric encryption.

How the demo was created.

I set an environment variable so that the Chrome browser would log the secret key material it generates. (For obvious reasons, this is something you DON'T want to do routinely.) On a Macintosh computer, you set this variable from the command line in the Terminal utility by typing:

```
launchctl setenv SSLKEYLOGFILE full-path
```

where *full-path* is the full pathname of a text file that will be used to log the secrets.

I launched Wireshark, a network analysis tool, which captured all the packets sent from and received by the Wi-Fi connection on my computer. Then I started the Chrome browser, went to target.com, and started the process (never completed) of buying some speakers. When I went to check out, I finally reached the secure website https://www-secure.target.com. (Interestingly, Target does not encrypt the communication until it is time to pay, so it does not treat WHAT I am buying as confidential information.)

Once I reached the secure website, I shut down the browser, stopped the network capture on Wireshark and saved the result.

When you open the saved file in Wireshark, you can enter 'SSL' as a filter and click Apply; this shows you all the TLS messages. It is also possible to filter the messages so that you just see the ones associated with one conversation. This is what is shown in the screenshots below. Wireshark presents two different views of each message---one in raw hex, and another that 'understands' the protocol, and parses the message into its various meaningful components.

Let's go through these messages one by one:

Client Hello. The **Client Hello** message (Figure 1) sent from my browser to the server includes, among other things, a 4-byte time stamp, a 28-byte randomly-generated value and a list of twenty 'cipher suites', collections of cryptographic algorithms that the browser supports. For instance,

TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

means Elliptic Curve Diffie Hellman for establishing the symmetric key, Elliptic Curve Digital Signature Algorithm for signatures, AES with 128-bit keys in Galois Counter Mode (that's a new one for me!) for symmetric encryption, and SHA256 for hashing. We haven't yet established the secure channel, so the information in this message is sent in the clear. You can see from the ASCII translation of the hex dump that we are at www-secure.target.com.

The 28-byte random value will be taken together with the 4-byte time stamp that precedes it as part of the key generation process, which we'll describe later. (I do not know why the time stamp says the date is March 5, 2022. I did this in November 2014.)

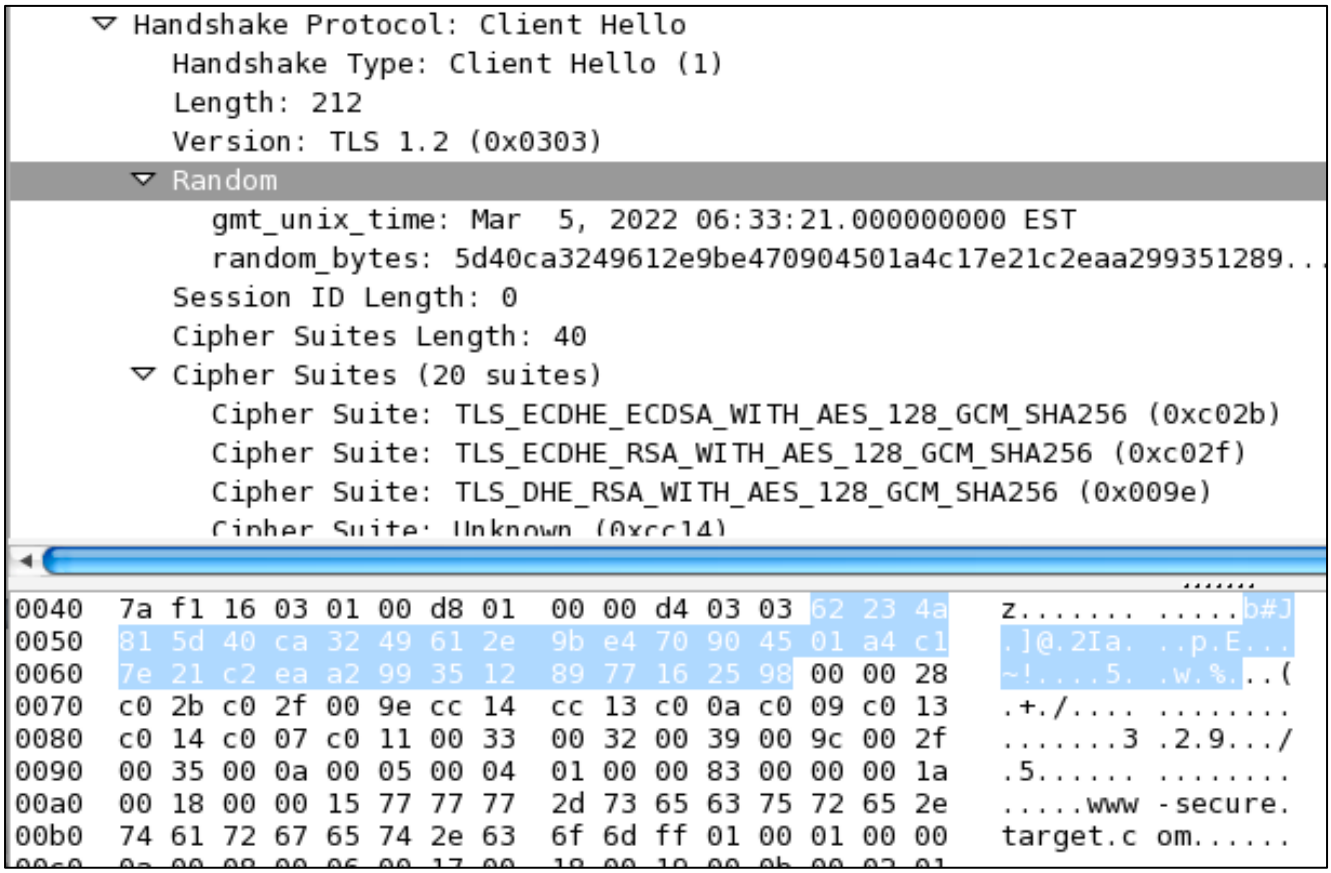

Figure 1: The Client Hello message, showing some of the supported cipher suites, the 4-byte time stamp, and the 28 bytes of random data

Server Hello. The **Server Hello** message contains a 4-byte time stamp (this time with the right date), an independently-generated random 28-byte value and the selected cipher suite, which is RSA for public-key encryption, AES - 256 in CBC mode for symmetric encryption, and SHA-1 for hashing.

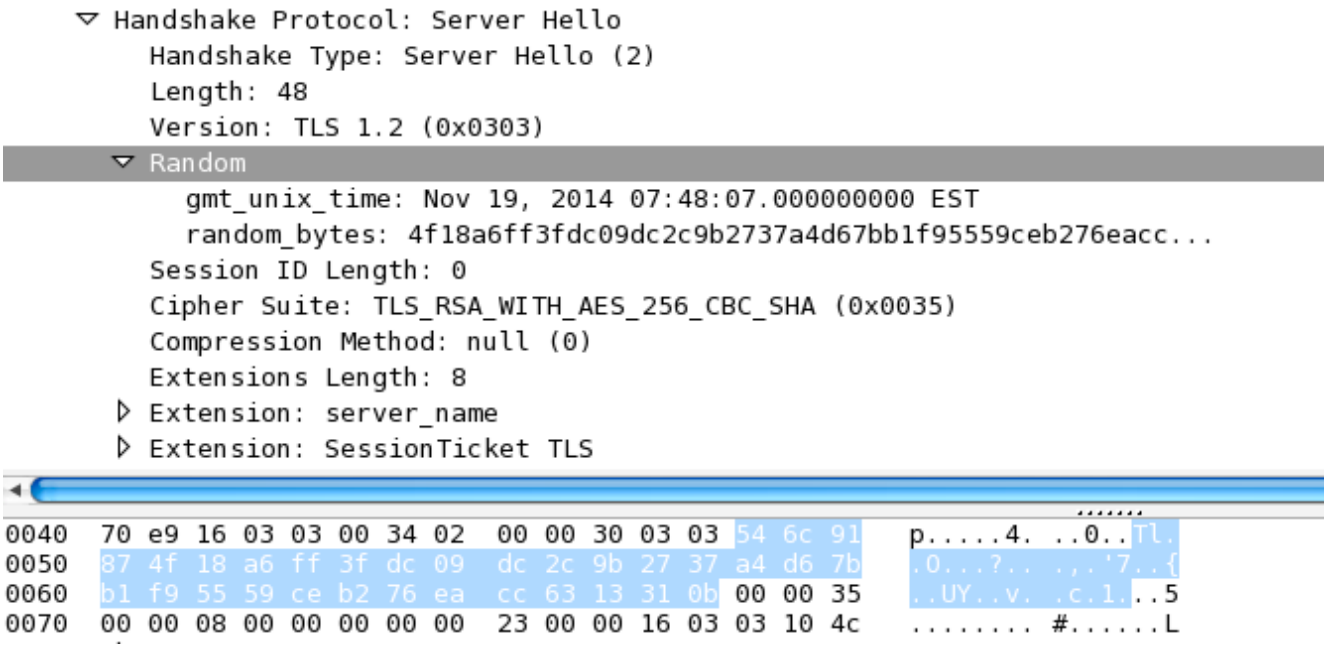

Figure 2: Server Hello message, showing the time stamp, random data, and the chosen cipher suite.

Certificates. The next message consists of **certificates** sent from the server. The three certificates in the chain match the ones reported to us in the friendlier view from the browser. (Figure 4.)

Figure 3: The list of certificates sent by the server, as displayed in Wireshark

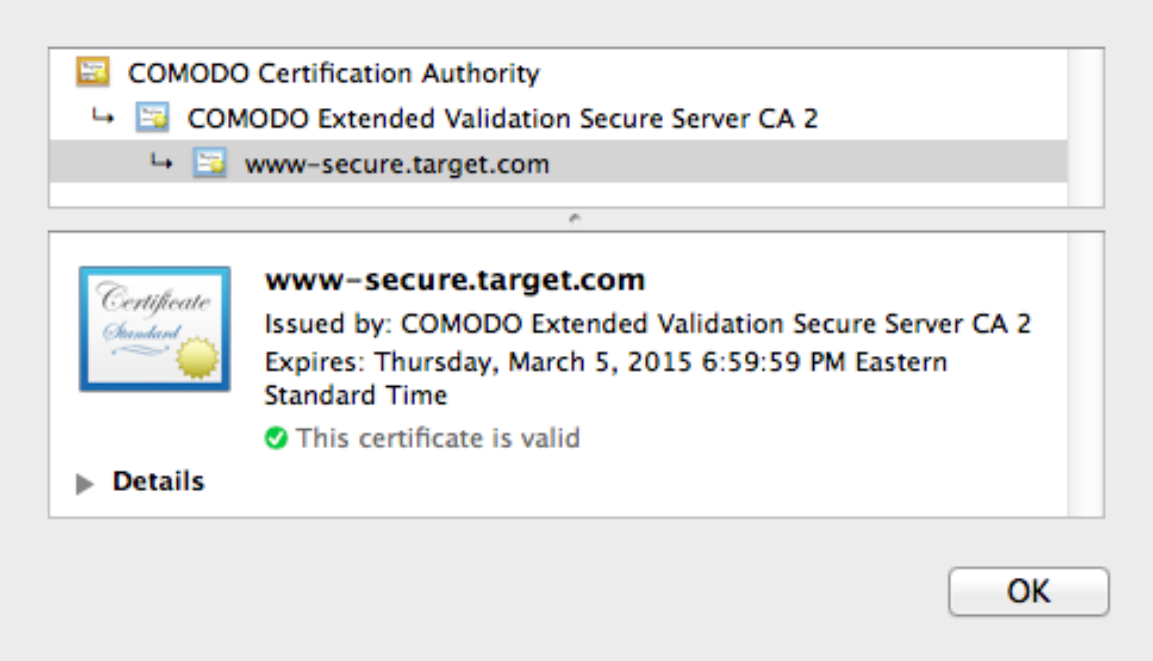

Figure 4: A same list of certificates, viewed in the browser

The purpose of the certificate is to show that the public key information it contains really does belong to Target, and thus prevent a Man-in-the-Middle attack in which we send confidential information to an attacker pretending to be Target. This is accomplished by having the certificate bear the digital signature of a trusted authority. Let's do some cryptographic calculations and independently verify the legitimacy of the certificate. The site certificate (the one at the bottom in the browser view, the first one in the Wireshark view) contains the signature. It's labeled as 'encrypted' in Wireshark's view, and as 'Signature' in the browser's view, which is what we show in Figure 5.

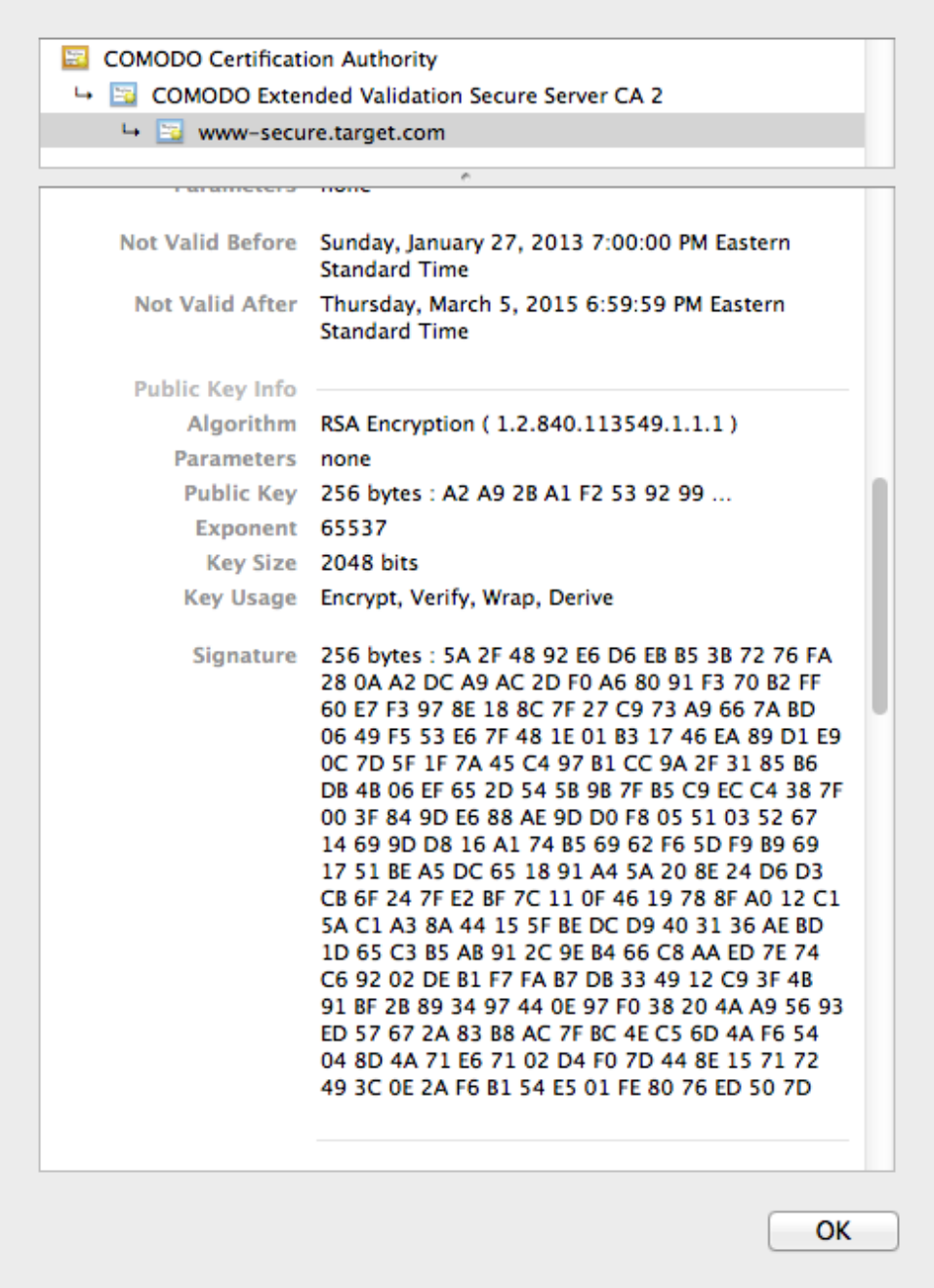

Figure 5: The signature on the site certificate

The public key for verifying this signature is on the *next* certificate in the chain, the one labeled 'Comodo Extended Validation Secure Server CA2'. The key consists of an RSA modulus and exponent, which here, as is customary, is 65537 = $2^{16} + 1$. To

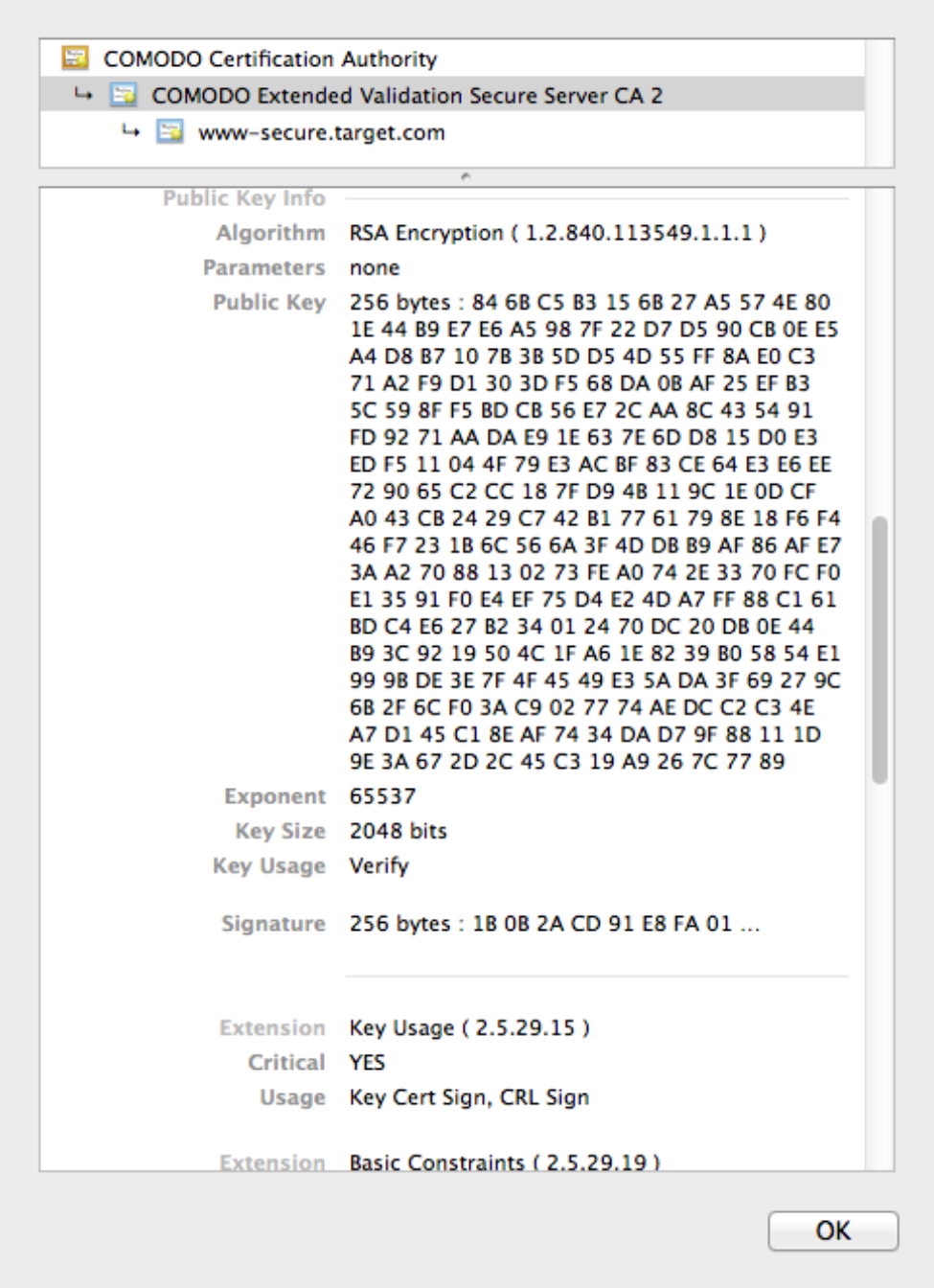

Figure 6: The public verification key of the signer of the site certificate. This key is found on the second certificate in the chain.

generate the Python code shown below, I just copied the hex signature and modulus from the certificates, removed the blanks, converted to long integers, and called Python's built-in pow function for modular exponentiation. You wouldn't notice anything special about the result if you printed it in decimal, but when we look at it in hex, we see that it contains about a gazillion (ok, about 1700) 1 bits with a few zeros tacked on in front, followed by the information the

signature was applied to. This result could not possibly arise by chance, so we must have done something right!

```
Python 2.7.6 Shell
Python 2.7.6 (v2.7.6:3a1db0d2747e, Nov 10 2013, 00:42:54)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.

>>> hex_signature='5A 2F 48 92 E6 D6 EB B5 3B 72 76 FA 28 0A A2 DC A9 AC 2D F0 A6
80 91 F3 70 B2 FF 60 E7 F3 97 8E 18 8C 7F 27 C9 73 A9 66 7A BD 06 49 F5 53 E6 7F
48 1E 01 B3 17 46 EA 89 D1 E9 0C 7D 5F 1F 7A 45 C4 97 B1 CC 9A 2F 31 85 B6 DB 4B
06 EF 65 2D 54 5B 9B 7F B5 C9 EC C4 38 7F 00 3F 84 9D E6 88 AE 9D D0 F8 05 51 03
52 67 14 69 9D D8 16 A1 74 B5 69 62 F6 5D F9 B9 69 17 51 BE A5 DC 65 18 91 A4 5A
20 8E 24 D6 D3 CB 6F 24 7F E2 BF 7C 11 0F 46 19 78 8F A0 12 C1 5A C1 A3 8A 44 15
5F BE DC D9 40 31 36 AE BD 1D 65 C3 B5 AB 91 2C 9E B4 66 C8 AA ED 7E 74 C6 92 02
DE B1 F7 FA B7 DB 33 49 12 C9 3F 4B 91 BF 2B 89 34 97 44 0E 97 F0 38 20 4A A9 56
93 ED 57 67 2A 83 B8 AC 7F BC 4E C5 6D 4A F6 54 04 8D 4A 71 E6 71 02 D4 F0 7D 44
8E 15 71 72 49 3C 0E 2A F6 B1 54 E5 01 FE 80 76 ED 50 7D'
>>> hex_signature=hex_signature.replace(' ','')
>>> signature=long(hex_signature,16)
>>> hex_modulus='84 6B C5 B3 15 6B 27 A5 57 4E 80 1E 44 B9 E7 E6 A5 98 7F 22 D7 D
5 90 CB 0E E5 A4 D8 B7 10 7B 3B 5D D5 4D 55 FF 8A E0 C3 71 A2 F9 D1 30 3D F5 68 D
A 0B AF 25 EF B3 5C 59 8F F5 BD CB 56 E7 2C AA 8C 43 54 91 FD 92 71 AA DA E9 1E 6
3 7E 6D D8 15 D0 E3 ED F5 11 04 4F 79 E3 AC BF 83 CE 64 E3 E6 EE 72 90 65 C2 CC 1
8 7F D9 4B 11 9C 1E 0D CF A0 43 CB 24 29 C7 42 B1 77 61 79 8E 18 F6 F4 46 F7 23 1
B 6C 56 6A 3F 4D DB B9 AF 86 AF E7 3A A2 70 88 13 02 73 FE A0 74 2E 33 70 FC F0 E
1 35 91 F0 E4 EF 75 D4 E2 4D A7 FF 88 C1 61 BD C4 E6 27 B2 34 01 24 70 DC 20 DB 0
E 44 B9 3C 92 19 50 4C 1F A6 1E 82 39 B0 58 54 E1 99 9B DE 3E 7F 4F 45 49 E3 5A D
A 3F 69 27 9C 6B 2F 6C F0 3A C9 02 77 74 AE DC C2 C3 4E A7 D1 45 C1 8E AF 74 34 D
A D7 9F 88 11 1D 9E 3A 67 2D 2C 45 C3 19 A9 26 7C 77 89'
>>> hex_modulus=hex_modulus.replace(' ','')
>>> modulus=long(hex_modulus,16)
>>> verify=hex(pow(signature,65537,modulus))
>>> verify
'0x1fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffff003021300906052b0e03021a05000414f418d08fb2d7ea
2ba68f96157a405ed68704a554L'
>>>
                                                              Ln: 16 Col: 4
```

Figure 7: Verification of the signature on the site certificate. The signature is applied to an SHA1 hash of the certificate, with special code prepended to say that this is an SHA-1 hash, and then padded out with a lot of 1's.

Evidently, the payload to be signed is padded out with all those 1s before the RSA algorithm is executed. But we have to dig down a bit to figure out how to read this and complete our signature verification. The signature algorithm is described in detail in the Internet standards document defining the RSA public-key signature standard (for the curious, it's RFC3447, Section 9.2). The steps for signing are:

- Hash the message to be signed, using the hash algorithm SHA1
- prepend the bytes `30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14`

- prepend a zero byte
- prepend 218 bytes FF (1672 1s in binary)
- prepend a one byte (01 in hex, 00000001 in binary)
- prepend a zero byte (00 hex, 00000000 binary)
- raise this to the power e mod N, where e is the public RSA exponent and N the modulus of the signer's public key
-

So now we know that the hash of the signed data should match the last 160 bits (40 hex digits) of the result of our calculation, namely

F4 18 D0 8F B2 D7 EA 2B A6 8F 96 15 7A 40 5E D6 87 04 A5 54

But exactly which bytes of the certificate is the hash applied to? Obviously, it can't be the entire certificate, as this includes the signature itself. Fortunately, Wireshark points this out to us under 'signed certificate' (Figure 8). This contains the serial number, the expiration date, the name and address of the Target company, and the public key, as it must.

To finish the signature verification, I exported the raw bytes of the signed certificate and used the built-in hash library of Python to compute the SHA-1 hash. The result matches the 160 bits we extracted earlier, so the signature is accepted.

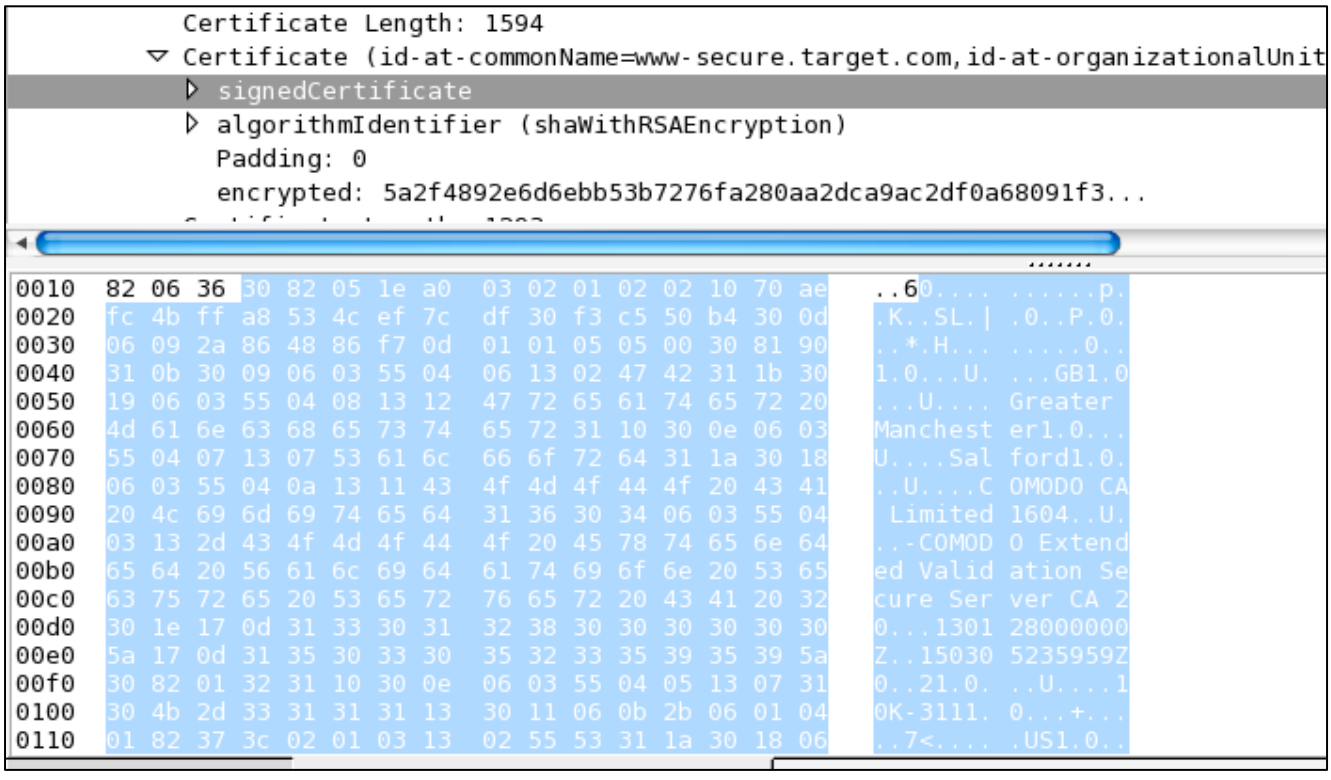

Figure 8: The portion of the certificate that is hashed before the signature is applied. The SHA-1 hash of this sequence of bytes is equal to the 160-bits string we computed earlier, so the signature is verified.

That's enough signature-verifying for this demo, but our browser is not done verifying. Why should we accept the signature of the certifying authority? Because *its* public key (on the second certificate in the chain) is signed with the public key on the last certificate (the root certificate). The root certificate, as it turns out, is `self-signed': You verify it with the public key on the same certificate!

So why should we trust the root certificate? The root certificate exactly matches one of the certificates stored in my Mac's 'keychain'. This is a small list of certifying authorities that are considered to be trustworthy. Should we believe that? It seems to be an axiom in computer security that ultimately you have to trust *somebody*.

Finishing the Handshake. The next protocol message, sent from the client to the server has three parts, called **Client Key Exchange**, **Change Cipher Spec**, and **Encrypted Handshake**. The client generates a 48-bit string called **the premaster secret.** This will be used by both the client and server to generate the symmetric keys used for the secured portion of the session. The Client Key Exchange sends the server the RSA-encrypted premaster secret, a 256-byte (2048-bit) value. This is the only place in the process where public-key encryption is used.

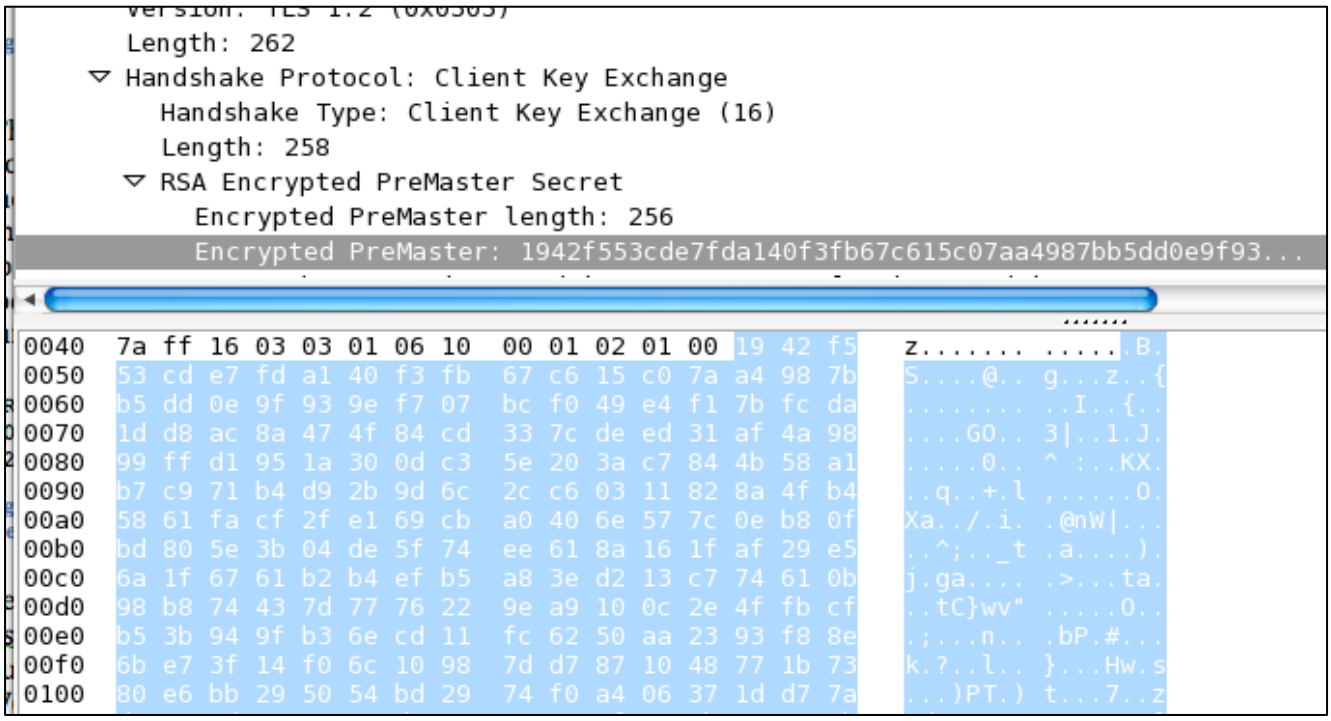

Figure 9: Client Key exchange message, showing RSA-encrypted premaster secret.

What is the actual premaster secret? Of course it cannot be sent directly to the server; that's why it's encrypted. It does not show up in these views in Wireshark, and in fact is *should not be stored*. But Chrome allows you to take the potentially dangerous step (presumably for debugging) of logging the secret key material, so this where we use the key log file. During the session I monitored, a lot of different secure connections were made, so the key log file contains more than 100 entries. But you can use a tool like grep to search for an entry using the first few hex digits of the encrypted secret. This turns up the following entry:

```
RSA 1942f553cde7fda1
03037baa39916a8d8a15eff048ecf32c9b3b828bc288bea2383a2531328c4172428ebf1ddf1252a0
2bfb51b1ea728aa7
```

Figure 10: An entry from the keylog file, showing the start of the RSA-encrypted premaster secret, and the premaster secret itself.

The first field gives just the first 8 bytes of the pre-master secret. The second is the premaster secret itself, a 48-byte (96 hex digits) integer generated using a cryptographically secure random number generator.

It would be nice to demonstrate the RSA encryption transforming the premaster secret into the encrypted version. We can get the public encryption key from the server's certificate, but remember that RSA plaintexts are always padded with random data prior to encryption, and we cannot recover these random bits. Of course the server, which has the RSA private key, can decrypt the message, strip away the padding, and obtain the premaster secret. (In the blog post cited at the beginning of these notes, the blogger worked with a debug build of the Firefox browser: he was able to insert debug statements and capture the random padding data, and thus independently reproduce the RSA encryption of the premaster secret.)

The premaster secret is used to compute something called the **master secret,** a 48-byte block from which the symmetric keys will be derived.

The **Change Cipher Spec** message informs the server that 'everything I send you from now on will be encrypted'.

Finally, the client hashes all of the handshake messages that have been exchanged up to this point, and 'encrypts' this. It is not actual encryption per se, but rather the result of applying the pseudo-random function (see below) to a seed value derived from the master secret and the hash of the handshake messages. The result is the **Encrypted Handshake Message**. The server should be able to repeat this computation by (a) decrypting the encrypted premaster secret; (b) using this to generate the master secret; (c) repeating the computation of the encrypted handshake message. This serves to authenticate the data sent from the client up to this point.

The next message from the server also has 3 parts, **New Session Ticket**, **Change Cipher Spec**, and **Encrypted Handshake Message**. The first provides a means for the client to start a new session without a full handshake and key exchange, by presenting the ticket to the server. We will ignore this---not every TLS server implements this feature. Change Cipher Spec again tells

the client that all subsequent messages will be encrypted, and the Encrypted Handshake Message uses the master secret and a hash of all the preceding handshake messages to derive a random string. This is different from the Encrypted Handshake Message sent by the client, because the hash includes one additional handshake message. Once again, the client can verify that this has been done correctly. The authenticated, secure connection has now been established. The handshake is over.

Generating the keys, and encrypting application data

The next message we see from Wireshark is labeled Application Data, and it contains encrypted application data sent from the client to the server (Figure 11). By pointing Wireshark to the keylog file, we can see its decrypted contents (Figure 12), an HTTP message beginning with

```
GET /checkout_process?catalogId=10051&langId=-1....
```

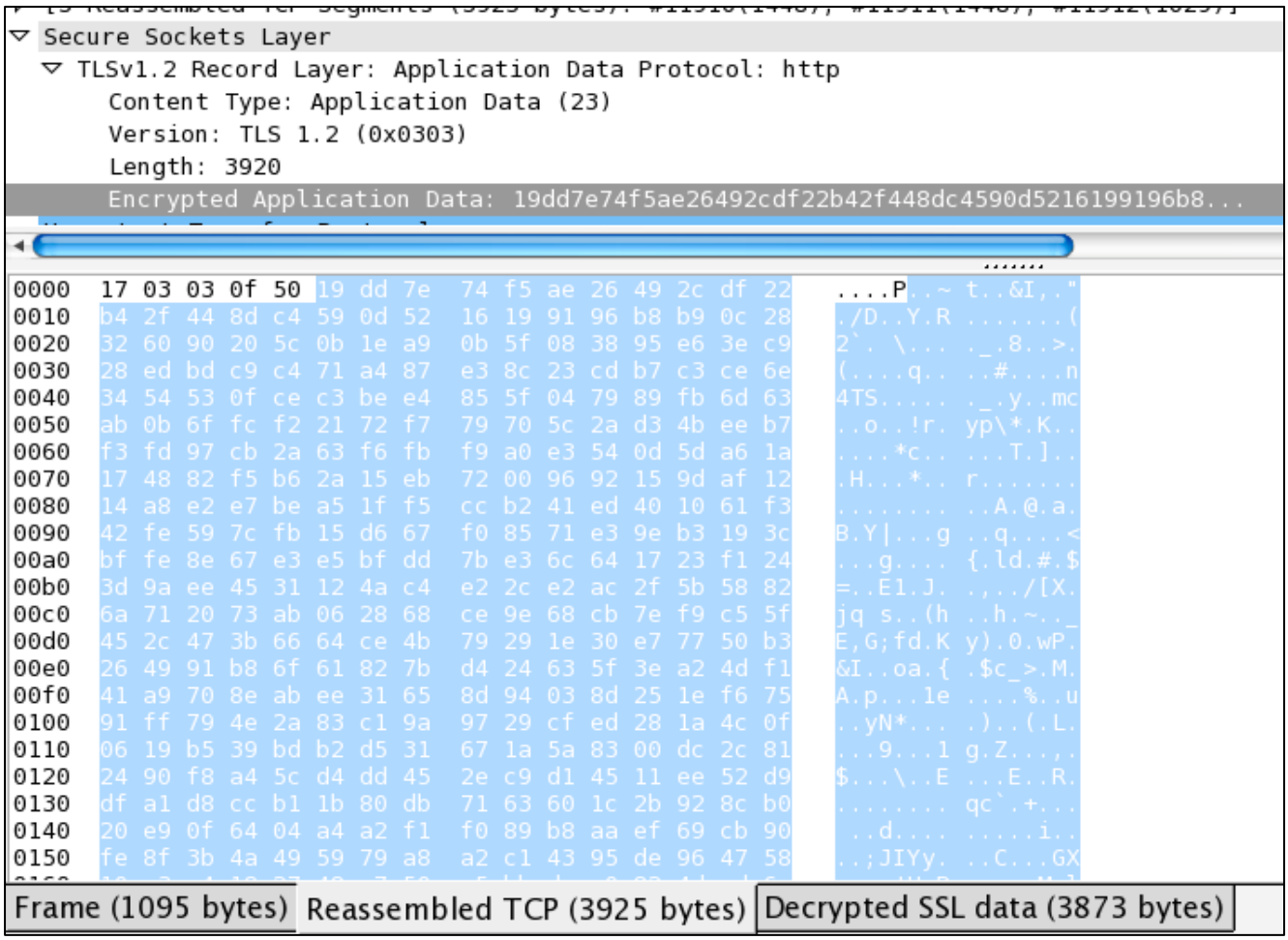

Figure 11: The first message from client to server that is encrypted with AES---ciphertext view.

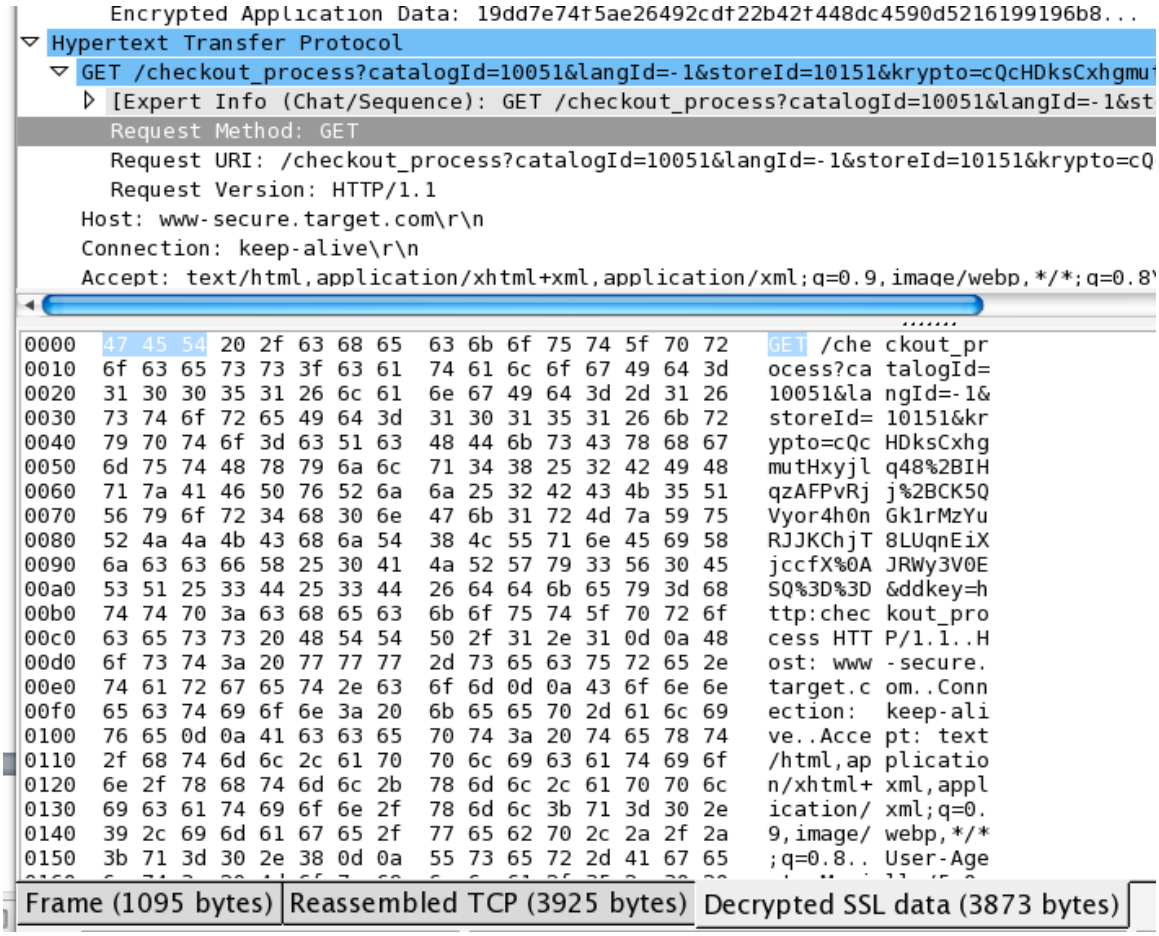

Figure 12: ..and plaintext view

Let's look in detail at the steps the server at Target takes to decrypt the encrypted application data.

We first have to know how the symmetric encryption key is generated from the premaster secret. (The definitive source for this is the Internet Standards document RFC 5246 describing the TLS 1.2 protocol.) At the heart of this process is the **pseudo-random function**, which is obtained by iterating a hash-based message authentication code (HMAC) based on the SHA256 hash function. The use of SHA256 is an innovation in TLS 1.2---earlier versions used a combination of MD5 and SHA-1. The Python code in Figure 13 is an implementation of the pseudo-random function. The HMAC produces data in 32-byte blocks. By iterating the process enough times, you can get as many pseudo-random bits as you want. You should be aware that the first three arguments to PRF are sequences of bytes, treated as strings. Thus the addition operator that appears in the code below is concatenation of these strings, and not addition of numbers.

The pseudorandom function is used together with the 48-byte premaster secret and the random data exchanged in the hello messages to generate the **master secret.** This is another 48-byte string. The client_random and server_random arguments in the code below are 32 bytes long: they consist of the 4-byte time stamp in each Hello message concatenated with the 28 random bytes.

The master secret is in turn combined with the 32-byte random strings from the Hello messages using the pseudo-random function to produce the **keyblock.** The size and structure of the key block depend on the cipher suite used. In our example, we are encrypting with AES 256 in CBC mode and using the SHA-1 hash function for message authentication. The key block in this case consists of four keys: A 20-byte key used by the client to compute SHA-1-based HMAC authentication tags, followed by a 20-byte key used by the server for the same purpose, then a 32-byte (256-bit) AES key used by the client for encrypting data (and the server for decrypting), followed by another 32-bit AES key used by the server for encrypting and the client for decrypting. (Note that in contrast to the setup we described at the start of the course, each party uses one symmetric key for encrypting and another for decrypting.) The code for computing the keyblock is shown in Figure 15. In our example, it is the third component of the keyblock that is used to encrypt data from the client to the server.

For what it's worth, the master secret is

a7bb34756fd93a981a9e60469da3848cf409c0c9a65983bf8de61f5c3d2f4170a76fd77415a80bff
20cf37eac6053d55

and the client write key is

4d04afed676a500ca0f5088a7e887deb53fc69e54b88e5500dda732beda6c2fd

This consists of 64 hex digits, thus 32 bytes or 256 bits.

Finally, we set this key on the encrypted application data seen in Figure 11, using the implementation of AES in the pycrypto package. The message is 3920 bytes long. Since we are in CBC mode, the first 16 bytes is the initialization vector, and the remaining 3904 bytes consist of ciphertext for us to decrypt. The decrypted version, also 3904 bytes, begins with the HTTP message shown above in Figure 12. The last few bytes of the decrypted message are shown in Figure 16 (Python displays the result in ASCII, with non-printing characters are rendered in hex preceded by `\x'.)

These final bytes consist of the byte value 10 (the ASCII encoding of the newline character \n) repeated 11 times. This is padding that was added to the plaintext to make it consist of a whole number of 16-byte blocks. The twenty bytes preceding this, which are mostly nonprintable characters, constitute the MAC tag. The actual plaintext HTTP message ends with the characters `pkyvprod1\r\n\r\n`. So the true plaintext is 3904-20-11=3873 bytes long: This is what is

displayed in Figure 12. (We will not reproduce the other step taken by the server, which is to verify the 20-byte MAC tag.)

Summary. The entire process, from the Client Hello message to the transmission of the first encrypted application data, took 0.24 seconds. Let's recap the cryptographic methods we saw used:

- *Secure random number generation* was used to produce the initial 28-byte random blocks sent by both client and server, and to generate the premaster secret.
- *RSA digital signatures* were used to authenticate the server to the client. The client verifies the signature on the server's certificate using the public verification key on the certificate authority's certificate.
- *RSA encryption* was used by the client to encrypt the premaster secret with the server's public key, obtained from the server's certificate.
- *AES symmetric encryption*, with 256-bit keys and 128-bit blocks, was used to send encrypted HTTP messages between client and server.
- *Cryptographic hash functions* were used in three different places: as part of the pseudo-random function, to hash the server's certificate for signing/signature verification, and to compute and verify the MAC tag on the encrypted message.

```
#We define the PRF as specified in the standard, except we add a fourth
parameter
#for the number of blocks to output.

def prf(secret,label,seed,numblocks):
    seed=label+seed
    output = ''
    a=hmac.new(secret,msg=seed,digestmod=hashlib.sha256).digest()
    for j in range(numblocks):
        output += hmac.new(secret,msg=a+seed,digestmod=hashlib.sha256).digest()
        a=hmac.new(secret,msg=a,digestmod=hashlib.sha256).digest()
    return output
```

Figure 13: The pseudorandom function in TLS v 1.2.

```
#Compute the master secret from the premaster secret.
#premaster_secret, client_random and server_random have string type, that
#is they are sequences of bytes represented as strings,
#so the addition in this code is concatenation of strings.

#We want 48-bit output, so we need to call prf to produce two
#32-bit blocks

def master_secret(pms,client_random,server_random):
    out=prf(pms,"master secret",client_random+server_random,2)
    return out[:48]
```

Figure 14: Computation of the master secret from the premaster secret.

```
#generate the key block.  We will need 20+20+32+32=104 bytes, so we need
#to generate 4 blocks and partition.  We are just doing the case AES256CBC-
SHA

def keyblock(ms,client_random,server_random):
    u=prf(ms,"key expansion",server_random+client_random,4)
    return (u[:20],u[20:40],u[40:72],u[72:104])
```

Figure 15: Computation of the keyblock from the master secret.

```
%3B%20s_sq%3Dtargetcomprod%252Ctargetusglobal%253D%252526pid%25253Dcheckout%2525
253A%25252520view%25252520%25252526%25252520manage%25252520cart%252526pidt%25253
D1%252526oid%25253Dhttp%2525253A%2525252F%2525252Fwww.target.com%2525252Fcheckou
t_process%252526ot%25253DA%3B;
tgtakalb=pkyvprod1\r\n\r\n\xb9L&(\x95\x05`w\xc8\rY\xc6\x8fbH\xc1\x10\xd9\x8a\xc7
\n\n\n\n\n\n\n\n\n\n\n\n
```

Figure 16: The end of the decrypted application data. The last 11 bytes are padding, and the 20 bytes before that are the MAC tag.