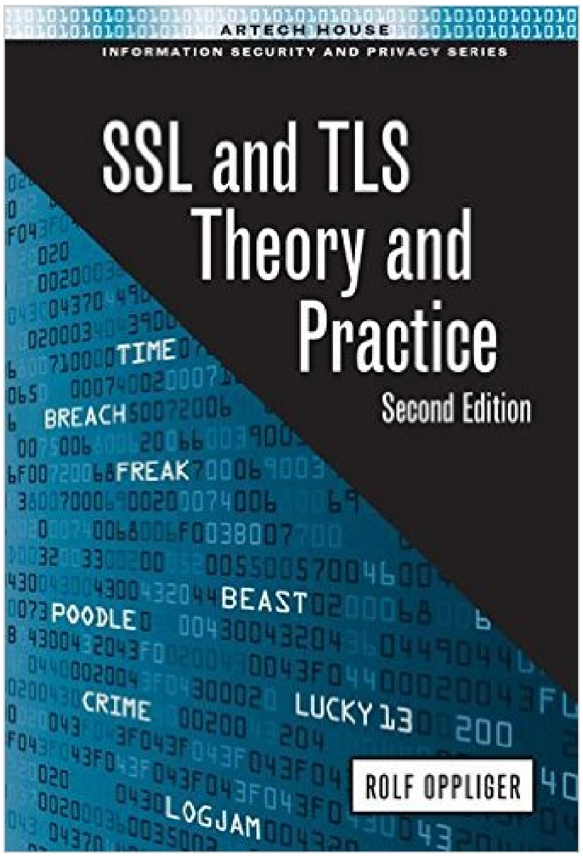
This Text is extracted from a draft version of the 3rd edition of Rolf Oppliger's upcoming book on "SSL and TLS: Theory and Practice" that is going to be published by Artech House (tentatively scheduled for 2022).

The second edition is available (ISBN 978-1-60807-8).



### 3.8.1 Renegotiation Attacks

In 2009, Marsh Ray and Steve Dispensa published a paper[52] in which they describe a vulnerability (CVE-2009-3555) that can be exploited to mount a MITM attack against an optional but usually recommended feature of the TLS protocol, namely to be able to renegotiate a session. The reasons that may make it necessary to renegotiate a session are discussed in Section 2.2.2. It may be the case that cryptographic keys need to be refreshed, cryptographic parameters need to be changed, or—most importantly—certificate-based client authentication needs to be invoked. More specifically, if a client needs to authenticate using a certificate but wants to hide its identity, then it may first establish an unauthenticated session to the server and then use this session to renegotiate another session that is authenticated with a certificate. Since the renegotiation messages are transmitted within the first session, the client certificate is encrypted during transmission and is not revealed to an eavesdropper. But keep in mind that a session renegotiation is not the same as a session resumption: While a new session with a new session ID is established in a renegotiation, an already existing and previously established session is reused in a resumption. Technically speaking, a client-initiated renegotiation can be started by having the client send a new CLIENTHELLO message to the server, whereas a server-initiated renegotiation can be started by having the server send a HELLOREQUEST message to the client. In either case, the party that receives the message can either accept or refuse the renegotiation request. If it accepts the request, then a new handshake is initiated. If, however, it refuses the request, then no handshake is initiated and a `no_renegotiation` alert message (with code 100) is returned instead.

A renegotiation attack is essentially a plaintext injection attack, meaning that the MITM tries to inject some additional data into an application data stream. If the injected data is delivered together with the rest of the data stream to the application, then some unexpected things can happen that may be exploited in an attack.

Let us consider, for example, a web setting in which TLS—and hence HTTPS—is used to secure an application. Such a setting and the outline of a renegotiation attack are overviewed in Figure 3.10. The adversary representing the MITM is located between the client (left side) and the server (right side). The client wants to establish a TLS session and sends a respective CLIENTHELLO message to the server. This message is captured by the MITM. Before forwarding it (on the client's behalf), the MITM establishes a first TLS session to the server. This session is used to send

52 When the paper was published, Ray and Dispensa were working for PhoneFactor, a then leading company in the realm of multifactor authentication. The publication appeared as a technical report of this company. In 2012, PhoneFactor was acquired by Microsoft, and hence the technical reports of PhoneFactor are no longer available on the Internet. However, there are still many Internet repositories that distribute them. You may simply search for "Renegotiating TLS."

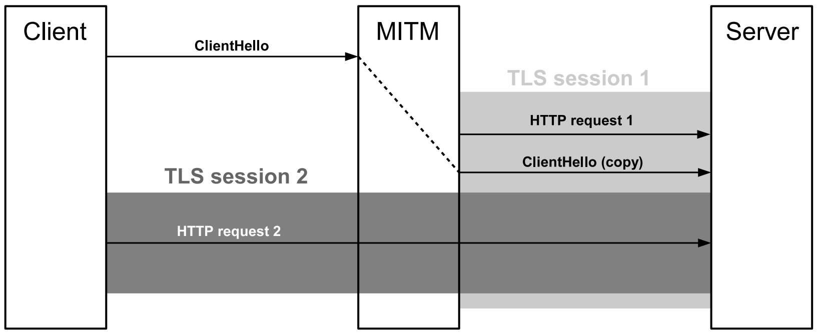


**Figure 3.10** The TLS renegotiation attack (overview).

an HTTP request message to the server (denoted HTTP message 1 in Figure 3.10). Let us assume that this request comprises the following two header lines (where only the first line is terminated with a new line character):

```
GET /orderProduct?deliverTo=Address-1
X-Ignore-This:
```

The meaning of these header lines will soon become clear. Many SSL/TLS implementations don't pass decrypted data immediately to the application. Instead they wait for other data arrive and pass them collectively to the application. This behavior is actually exploited in the attack.

Immediately after having sent this HTTP message to the server, the MITM forwards the client's original CLIENTHELLO message to the server, again using session 1. From the server's perspective, this message looks as if the client that originates the message wanted to renegotiate the session. So a second TLS session is established between the client and the server, and the use of TLS renegotiation suggests that this second TLS session is tunneled through the first TLS session. This also means that the handshake of the second TLS session is oblivious to the MITM, and that the MITM can only send back and forth handshake messages (through TLS session 1). If the handshake succeeds, then a second TLS session is established between the client and the server, and this session is now end-to-end protected, meaning that the adversary cannot decrypt the transmitted data. Let us assume that the server authenticates the client, and that the server issues a cookie as a bearer token used for client authentication. Due to the end-to-end protection, the adversary cannot access the cookie to misuse it directly. But see what happens if the client uses the end-to-end protected TLS session 2 to send another HTTP request message—labeled "HTTP request 2" in Figure 3.10—to the server. In this case, the two messages may be concatenated and collectively passed to the application. From the application's viewpoint, it is therefore no longer possible to distinguish the two messages—they both appear to originate from the same source, i.e., the client. If, in our example, HTTP request 2 started with the following two header lines

```
GET /orderProduct?deliverTo=Address-2
Cookie: 7892AB9854
```

then the two messages would be concatenated as follows:

```
GET /orderProduct?deliverTo=Address-1
X-Ignore-This: GET /orderProduct?deliverTo=Address-2
Cookie: 7892AB9854
```

The `X-Ignore-This:` header is an invalid HTTP header and is therefore ignored by most implementations. Since it is not terminated with a new line character, it is concatenated with the first line of HTTP request 2. This basically means that the entire line is going to be ignored and hence that the requested product is going to be delivered to `Address-1` instead of `Address-2`. Note that the cookie is provided by the client and is therefore valid, so there is no reason not to serve the request. The problem is not the cookie, but the fact that some unauthenticated data is mixed with authenticated data and passed together to the application. This leads to a situation, in which the application falsely suggests that either data is authenticated. Also note that there are many possibilities to exploit this vulnerability and to come up with respective (renegotiation) attacks. Examples are given in the original publication and many related articles, as well as a recommended reading that was posted shortly after the original publication.[53]

Renegotiation attacks are conceptually related to cross-site request forgery (CSRF) attacks, so, in general, any protection mechanism put in place against CSRF attacks also helps mitigating renegotiation attacks. Furthermore, because the renegotiation feature is optional (as mentioned above), a simple and straightforward possibility to mitigate such attacks is to disable the feature and not support renegotiation in the first place. Less strictly speaking, it may be sufficient to only disable client-initiated renegotiation (as a side effect, this also protects the server against some DoS attacks). But since disabling renegotiation is not always possible, people have come up with other protection mechanisms. Most importantly, the IETF TLS WG has developed a mechanism that provides handshake recognition, meaning that it

53 http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html.

must be confirmed that when renegotiating both parties have the same view of the previous handshake, i.e., the one that is being renegotiated. As introduced in Section 3.4.1.19, there is a distinct TLS extension, i.e., the `renegotiation_info` extension specified in [30], that serves this purpose. In essence, it binds the renegotiated handshake to the previous one. In our example from Figure 3.10, handshake 2 has no previous handshake, and if it had, then it would be different from handshake 1 and unknown to the adversary. This seems to mitigate the attack.

To invoke the secure renegotiation mechanism, the client and the server must both include the `renegotiation_info` extension in their hello messages. In the initial handshake, the extensions are empty, meaning that the extensions comprise no data. The client and server only signal to each other that they support the extension. If, at some later point in time, the client wants to securely renegotiate, it adds another `renegotiation_info` extension to its CLIENTHELLO message. This time, the extension comprises data, namely the client-side `verify_data` field of the FINISHED message that was sent in the handshake for the session it wants to renegotiate. Remember from Section 3.2.4 that the `verify_data` field comprises a hash value of all messages sent in a handshake before the FINISHED message. It thus stands for a particular handshake and a particular session. If the server is willing to securely renegotiate this session, then it sends back a SERVERHELLO message with a `renegotiation_info` extension that comprises both the client-side `verify_data` field of the client's FINISHED message and the `verify_data` field of the server's FINISHED message—both from the previous handshake. Intuitively, by including one or two `verify_data` fields from the previous handshake, the parties ensure that they both have the same view of the previous handshake, and hence that they actually renegotiate the same session.

There is an interoperability issue: Because some SSL 3.0, TLS 1.0, and sometimes even TLS 1.1 implementations have problems gracefully ignoring empty extensions at the end of hello messages, people have come up with another possibility to let a client signal to a server that it supports secure renegotiation. Instead of sending an empty `renegotiation_info` extension in the CLIENTHELLO message, the client includes a special signaling cipher suite value (SCSV), i.e., `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` with byte code 0x00FF, in the list of supported cipher suites [30]. This also tells the server that it is willing and able to support secure renegotiation. The actual secure renegotiation then remains the same (and hence the secure renegotiation extension still needs to be supported by either party). The only point in using `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` is to avoid any empty extension at the end of a hello message. This should make some implementations more stable.

The goal of the `renegotiation_info` extension is to link the session that is being renegotiated to the handshake of the previous session. This is to mitigate the renegotiation attack and its variants. A client that is tricked into renegotiation does not automatically provide the `verify_data` field from the previous handshake. Also, if a MITM modified the CLIENTHELLO message to supply the required data, then the respective message modification and integrity loss would be detected at the TLS level.

Between 2010 and 2014, it was therefore believed that the renegotiation attack was successfully mitigated. In 2014, however, it was shown that this was not completely the case, and that in some situations a renegotiation attack remains feasible, even if the secure renegotiation mechanism empowered by the `renegotiation_info` extension is put in place [66]. The respective attack employs three handshakes and is therefore called *triple handshake attack*, or *3SHAKE attack*, in short. The attack is fairly sophisticated and we only briefly sketch it here.

In a triple handshake attack, the MITM is to cleverly exploit the following two weaknesses of the TLS handshake protocol:

- First, the MITM can establich two sessions—one between the client and the MITM and the other between the MITM and the server—that share the same master key and session ID. If, for example, RSA is used for key exchange, then the MITM can simply decrypt the premaster secret from the client and reencrypt it with the public key of the server. The random values are left unchanged, and the session ID from the server is relayed back to the client. After the handshake, the master keys and session IDs of the two sessions are the same. But while the client is tricked into thinking that it has established a session to the server (where in fact it has a session with the MITM), the server thinks that it is connected with an unauthenticated client and has no way of detecting the existence of the MITM. Technically speaking, this type of attack has been known as an *unknown key-share attack* [67].
- Second, if an unknown key-share attack is followed by a session resumption, then the resulting two sessions do not only share the same master key and ID, but they also share the same `verify_data` fields in their respective FINISHED messages. This is mainly due to the fact that certificates are not used (and hence do not appear) in a session resumption.

Combining these two weaknesses, the MITM can establish two sessions—one to the client and another to the server—that use the same `verify_data` field in their FINISHED messages and hence also share the same `renegotiation_info` values. This, in turn, means that the renegotiation attack can be mounted again (as a 3SHAKE attack), and that the `renegotiation_info` extension does not really solve the problem. Referring to its name, the 3SHAKE attack comprises three steps or handshakes, respectively.

- In step 1, the MITM mounts an unknown key-share attack to establish two sessions that share the same master key and session ID. As is usually the case in a MITM attack, the first session is between the client and the MITM, and the second session is between the MITM and the server.
- In step 2, the MITM waits until the client resumes its session. In contrast to a renegotiation that requires the `verify_data` from the FINISHED message of the previous handshake, only the session ID and the master key are required to resume a session. The MITM can do this also for his or her session to the server. In the end, there are two fully synchronized sessions that share the same `verify_data` field. After these preparatory steps, the MITM can actually mount the renegotiation attack.
- In step 3, the MITM first sends HTTP request 1 to the server and then has the server trigger a renegotiation that requires client-side authentication using, for example, a certificate. The respective handshake takes place between the client and the server. To mitigate "normal" renegotiation attacks, the client adds the `renegotiation_info` extension to its CLIENTHELLO message. The data of this extension refers to the `verify_data` field of the previous session to the MITM. According to what we have said before, this is the same value as the `verify_data` field of the previous session between the MITM and the server. So it can be verified by the server, and the server thus believes that the previous session to the MITM is the one that is being renegotiated. When the now authenticated client then sends an HTTP request 2 to the server, both requests are concatenated and collectively passed to the application. This means that the same attack becomes feasible, and hence that we are back where we started.

The bottom line is that triple handshake attacks can still be mounted in spite of the `renegotiation_info` extension. The underlying problem is that an unknown key-share attack combined with a subsequent session resumption allows a MITM to establish two fully synchronized sessions. This is true even if the sessions are bound to different server certificates (the session from the client to the MITM is bound to the MITM's certificate, whereas the connection from the MITM to the server is bound to the server certificate). One way to mitigate the unknown key-share attack is to make the generation of the master key not only depend on the premaster secret and random values selected by the client and the server, but also on the server certificate. Even more generally, it is reasonable to make the master secret depend on all handshake messages that have been exchanged previously. Because the CERTIFICATE message comprises the server certificate, this ensures that the master secret also depends on the server certificate, and hence that an unknown key-share attack is infeasible to mount.

The `extended_master_secret` extension specified in RFC 7627 [28] follows this line of argumentation (cf. Section 3.4.1.19). It ensures that the master secret that is generated—let us call it *extended master secret*—is unique. In either case, it ensures that different sessions established to different servers have different extended master secrets. Remember from Section 3.1.1 that a master secret is normally generated as

```
master_secret =
   PRF(pre_master_secret,"master secret",
       client_random + server_random)
```

If the `extended_master_secret` extension is negotiated, then the master secret is generated in a slightly different way:

```
master_secret =
   PRF(pre_master_secret,"extended master secret",
       session_hash)
```

Note that this construction uses another label and that the client and server random values are replaced with a session hash. This is a hash value computed over the concatenation of all handshake messages (including their type and length fields) sent or received, starting with the CLIENTHELLO message and up to and including the CLIENTKEYEXCHANGE message. This includes, among other things, the client and server random values and the server certificate. Since TLS 1.2, the hash function is the same that is also used to compute the FINISHED message. For all previous versions of the TLS protocol, the hash function employs the concatenation of MD5 and SHA-1. We have seen and discussed this before.

If a TLS session is established with both the `renegotiation_info` and `extended_master_secret` extensions in place, then it is reasonable to assume that the respective session is going to be secure against all types of renegotiation attacks. At least nobody has found a possibility to mount yet another attack in this setting.