Raluca Ada Popa
Spring 2018

CS 161
Computer Security

Discussion 6

Week of February 26, 2018

Question 1 ***TLS threats*** **(10 min)**

An attacker is trying to attack the company Boogle and its users. Assume that users always visit Boogle's website with an HTTPS connection, using ephemeral Diffie-Hellman. You should also assume that Boogle does not use certificate pinning. The attacker may have one of three possible goals:

1. Impersonate the Boogle web server to a user
2. Discover some of the plaintext of data sent during a past connection between a user and Boogle's website
3. Replay data that a user previously sent to the Boogle server over a prior HTTPS connection

For each of the following scenarios, describe if and how the attacker can achieve each goal.

(a) The attacker obtains a copy of Boogle's certificate.

> **Solution:** None of the above. The certificate is public. Anyone can obtain a copy simply by connecting to Boogle's webserver. So learning the certificate doesn't help the attacker.

(b) The attacker obtains the private key of a certificate authority trusted by users of Boogle.

> **Solution:** The attacker can impersonate the Boogle web server to a user. The attacker can't decrypt past data, because the attacker doesn't learn Boogle's private key—only the CA's private key. All that the CA's private key can be used for is to create bogus certificates, which can be used to fool the client into thinking it is talking to Boogle—but doesn't allow learning past data. Replays aren't possible, due to the nonces in the TLS handshake.

(c) The attacker obtains the private key corresponding to an old certificate used by Boogle's server during a past connection between a victim and Boogle's server. Assume that this old certificate has been revoked and is no longer valid. Note that the attacker does not have the private key corresponding to current certificate.

Solution: None. Unless the attacker can figure out either a or b, the attacker will not be able to decrypt the data of past connections.

This can't be used to impersonate a Boogle server because the attacker doesn't have a fresh valid certificate corresponding to the stolen private key, and can't use the previous certificate for that key because it's been revoked.

Question 2 ***TLS protocol details*** **(20 min)**

Depicted below is a typical instance of a TLS handshake.

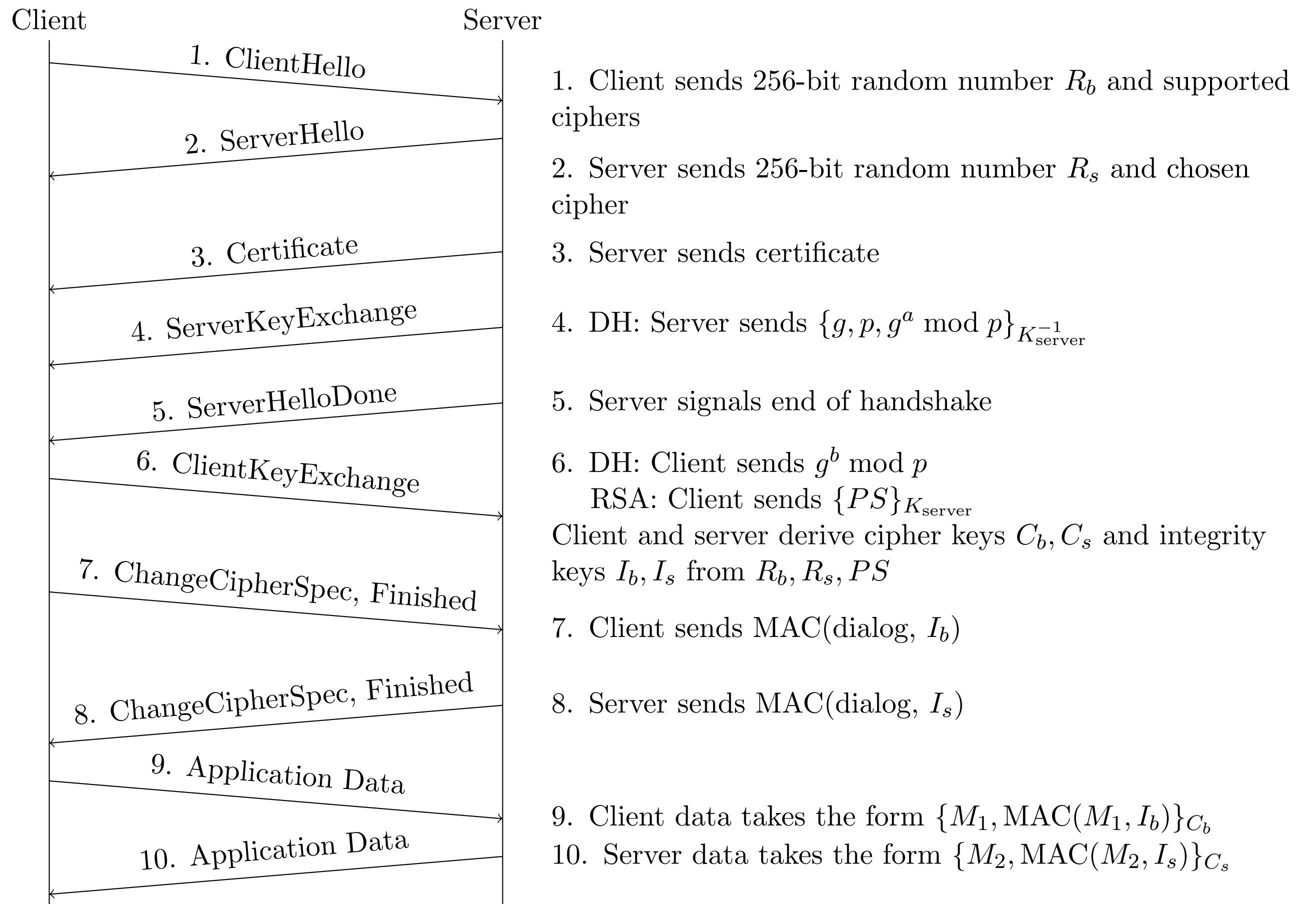


Figure 1: TLS 1.2 Key Exchange

(a) What is the purpose of the *client random* and *server random* fields?

> **Solution:** Because the master secret depends on these, they act as nonces that prevent replay attacks.

(b) ClientHello and ServerHello are not encrypted or authenticated. Explain why a man-in-the-middle cannot exploit this. (Consider both the Diffie-Hellman and RSA case.)

> **Solution:** The use of either public key encryption (RSA handshake) or a Diffie-Hellman exchange prevents an eavesdropper from learning the Premaster Secret.
>
> A MITM attacker who alters any of the values will be exposed, as follows.
>
> For the RSA handshake case, the MITM will be unable to read the Premaster

Secret sent by the client because it is encrypted using the server's public key. When the client and server exchange MACs over the the handshake dialog, the MITM will be unable to compute the correct MACs for their altered dialog because they will not know the corresponding integrity keys derived from the Master secret.

For the Diffie-Hellman case, the MITM will be unable to alter the value of $g^a \mod p$, because the client requires that the value have a correct signature using the server's public key. Because the MITM cannot alter the value, they cannot substitute $g^{a'} \mod p$ for which they know a'. Without knowledge of the exponent, the MITM cannot compute $g^{ab} \mod p$ to obtain the Premaster Secret.

(c) Note that in the TLS protocol presented above, there are two cipher keys C_b and C_s. One key is used only by the client, and the other is used only by the server. Likewise, there are two integrity keys I_b and I_s. Alice proposes that both the server and the client should simply share one cipher key C and one integrity key I. Why might this be a bad idea?

Solution: Vulnerable to **reflection** attacks: a man-in-the-middle can send a client's application data back to them. It will still verify the appropriate checks, but the user will think that this is the response from the server. Likewise, an attacker can reflect a server's response back to the client. Note that due to the existence of sequence numbers in the TLS specification, the actual attack would be a little more complicated.

(d) The protocol given above is a simplified form of what actually happens. After step 8 (ChangeCipherSpec), the protocol as described above is still vulnerable. What is the vulnerability and how could you fix this?

Solution: An attacker can perform a **replay** attack, where they send the same application data twice. The solution is to add sequence numbers (which is what the actual TLS specification does, with some extra details involved).

Question 3 ***Lists and Trees of Hashes*** **(20 min)**

BitTorrent splits large files into small file chunks which are then transmitted between peers in such a way that each peer eventually ends up with the whole file. Commonly, chunks are of size 2^8 KiB = 256 KiB.

Because you cannot trust peers, you have to verify each chunk as you download them from a peer before you start providing them to other peers. Furthermore, you want to be able to do this as soon as possible and not wait for the whole file to be downloaded. You also want to be able to know which part of the file got potentially corrupted so that you do not have to re-download the whole file.

To achieve the above properties, BitTorrent uses a Torrent file. The file contains information describing the file (or files) to be transmitted, and their chunks. You must obtain this file from a trusted source.

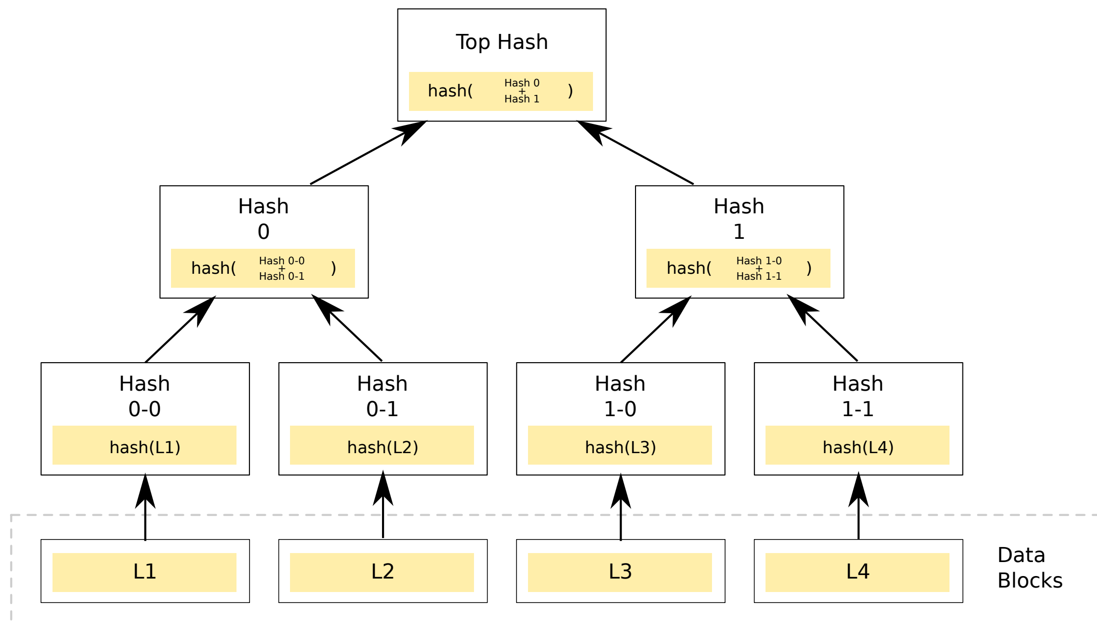
(a) Initially, a Torrent file contained a list of SHA-1 hashes for each chunk. How large is such a list for a 10 GiB large file, if one SHA-1 hash takes 160 bits? (Note: 10 GiB $= 10 * 2^{10} * (4 * 256)$ KiB)

> **Solution:** Number of chunks: 10 GiB / 256 KiB $= 10 \cdot 2^{30} / 2^{18} = 10 \cdot 2^{12} = 40 \cdot 2^{10}$. Hence, size of the list: $40 \cdot 2^{10} \cdot 20$ B = 800 KiB

(b) One way to make Torrent files smaller is to instead store only a hash of the hash list (top hash, or root hash) in the file and retrieve the hash list itself from a peer. Why would we want to make a Torrent file smaller? What is a downside of this approach?

> **Solution:** We want to reduce the size of Torrent files in order to minimize load on the servers hosting them.
>
> The downside is that we have to retrieve the whole list before we can verify any chunk, because we first have to verify the list itself against the top hash. But transmitting the hash list takes longer than transmitting the first chunk, since 800 KiB is larger than 256 KiB. Therefore, we cannot start providing the first chunk to others as soon as we have it. We also want to avoid transmissions larger than a chunk size.
>
> Moreover, hashes are not very compressible, so compression does not help much.

(c) One approach to address the issue of the size of the hash list is to split it into chunks. However, you would then need a hash list of those chunks. A better approach is to generalize this idea and use a data structure called a hash tree or Merkle tree:



Now you do not need the whole hash list in advance to verify one chunk. Instead, you can ask your peer to provide you with some hashes along with the chunk just received.

Suppose you just received chunk L2 from a peer. Which and how many hashes do you need to verify if you correctly received chunk L2? How would you generalize which and how many hashes you need for each chunk? (Hint: This might be useful to implement efficient updates for part 3 of Project 2.)

Solution:

To verify if L2 was correctly received, we need hash 0-0, hash 1, and the top hash. Since the top hash is provided in the Torrent file, we need just two hashes from the peer: hash 0-0 and hash 1.

In general, we need the hash of the received chunk's sibling, the uncle of the received chunk's hash, and so on until the top hash. The number of required hashes is logarithmic in the total number of chunks.

For the 10 GiB file, with each chunk we get the following size of hashes:

$$\left\lceil log_2\left(40 \cdot 2^{10}\right)\right\rceil = 16$$
$$16 \cdot 20 \text{ B} = 320 \text{ B}$$

The downside is that now for all chunks overall we have to get $40 \cdot 2^{10} \cdot 16 \cdot 20$ B = 12.5 MiB of hashes.

Example from the BitTorrent specs: Hashes required for verifying chunk P8 are marked with *. You can compute the hash for the chunk P8 yourself, and you have the top hash provided in a Torrent file.