Solutions to Selected Problems

Guide to Internet Cryptography

Companion Material

February 12, 2026

Preface

This document provides solutions to selected problems from the book *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*. The material is intended for educational use in courses and self-study.

Book website: https://link.springer.com/book/10.1007/978-3-031-19439-9

1 Chapter 11: A Short History of TLS

Problem 11.1 Handshake SSL 2.0

(a) Identify the challenge-and-response subprotocol in the SSL 2.0 handshake. How does the server authenticate itself?

(b) Could the handshake be strengthened if a modern hash function was used instead of MD5?

Solution

(a) The callenge-and-response subprotocol cosists of message CH containing the challenge r_C, and the SV message containing mac_S, which is computed over the challenge. This is combined with a key agreement protocol using RSA encryption in message CMK. This key agreement protocol isn't secure anymore, since mk can be derived with an exhaustive search attack on the 40 secret bits of mk. Assuming it was safe back in 1993, the server authenticates itself to the client through its ability to decrypt the ciphertext c contained in CMK.

(b) No. The main weakness of the protocol - which was enforced through government regulations on the use of cryptography back then - is that only 40 bits of mk are encrypted, while the other bits are transmitted in the clear.

Problem 11.2 Handshake SSL 2.0

Your goal is to impersonate an SSL 2.0 server S towards a client who supports [SSL 2.0 with] export ciphersuites. Suppose that you can manipulate the DNS to redirect all traffic destined for S to your server A.

(a) Which information do you need from S before starting the attack? How can you get this information?

(b) Is it necessary that S also supports export ciphersuites?

(c) Which information do you need to be able to perform an exhaustive key search? How would you implement this?

(d) Please sketch the full attack.

Solution

(a) You only need the certificate of the server. The certificate can be retrived by passively recording an SSL 2.0 handshake.

(b) No, S doesn't need to support export ciphersuites. But the client must offer them in its ClientHello message. If A then only includes export ciphersuites in its ServerHello message, the client is forced to select one export ciphersuite, because it must make its selcetion from the intersection of ciphersuite proposals.

(c) The adversary needs mk_{clear} (which is contained in CMK), the values r_C and r_S (sent in the clear in CH and SH, resp.), and the message CF from the client to check his guesses. For each guess mk' for the unknown 40 bits of mk_{secret}, the attacker must perform a key derivation to retrieve cwk'. This key can then be tested against the encryption of CF: If the decryption returns the correct value r_S, then the guess mk' was correct. Please note that this is an *active* attack, since the attacker forced the client into using export ciphersuites. Therefore, the exhaustive search for the correct mk' amongst the 2^{40} possibilities must be finished before the client closes the TCP connection because of a timeout.

(d) Step-by-step:

1. The attacker performs a handshake with the target server S which he wants to impersonate. This can be any TLS version (up to 1.3), beacuse all that is required here is a valid certificate *cert* from the server, and the certificate isn't bound to a specific TLS version.
2. The attacker sets his fake server A with the retrieved *cert*, and redirects the victims browser from S to A (through a man-in-the-middle attack, through routing or DNS manipulation).
3. Upon reception of the CH message, A returns a SH message indicating support for SSL 2.0 only, and containing only export ciphersuites.
4. The client is forced to choose one of these export cipehrsuites, and prepares its CMK message as indicated in Figure 11.1.
5. Upon reception of CMK, A extracts mk_{clear} and does a parallel exhaustive search for the remaining 40 bist of mk_{secret} (see (c)).
6. If this exhaustive search completes before the client aborts the TCP connection because of a timeout, A can generate the SV and SF messages, and successfully coomplete the handshake. In addition, A knows all the keys used in the SSL 2.0 session and can de- and encrypt application layer data correctly.

Problem 11.3 Cryptographic Export regulations

Please do a web search on the export regulations that were in place during the Clinton administration.

Solution

https://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States
https://www.latimes.com/archives/la-xpm-1999-sep-17-fi-11110-story.html
https://www.ams.org/notices/200204/comm-diffie.pdf
https://www.everycrsreport.com/reports/RL30836.html

Problem 11.4 Padding in TLS 1.0

Suppose you use AES-128 in CBC mode and HMAC-SHA1 on a 100-byte plaintext. Which value do the padding bytes have?

Solution

In TLS 1.0 with CBC mode, the plaintext is formatted as

$$\text{data} \parallel \text{MAC} \parallel \text{padding} \parallel \text{padding\_length}.$$

Step 1: Determine the length before padding. Given:

- Plaintext (application data): 100 bytes
- HMAC-SHA1 output: 20 bytes
- AES block size: 16 bytes

Length before padding:

$$100 + 20 = 120 \text{ bytes}.$$

Step 2: Compute required padding. The total length (including padding and the padding_length byte) must be a multiple of 16.

$$120 \bmod 16 = 8.$$

Thus, we must add

$$16 - 8 = 8 \text{ bytes}$$

of padding (including the padding_length byte).

Step 3: Determine padding byte values. In TLS 1.0, all padding bytes (including the final padding_length byte) have the same value:

$$\text{padding\_length} = 7.$$

Since the padding length field encodes the number of padding bytes *excluding itself*, and we have 8 padding bytes total:

$$8 - 1 = 7.$$

Final Answer. Eight padding bytes are added, each with value:

$$\boxed{\texttt{0x07}}$$

So the padding is:

$$\texttt{07 07 07 07 07 07 07 07.}$$

The final record length is:

$$120 + 8 = 128 \text{ bytes},$$

which is a multiple of the AES block size.

Problem 11.5 PRF function in TLS 1.0

Suppose the two PRF functions P_MD5 and P_SHA1 would each be iterated precisely five times, and you could get access to the resulting pseudorandom bit stream. Suppose further that secret was only 128 bit. Can you sketch an exhaustive key search attack on secret of complexity 2^{64}?

Solution

If we could get access to the keystream issued by P_MD5 and P_SHA1 separateley (i.e., before they are combite with XOR), we could mount separate exhaustive search attacks on s1 and s2, resp. The values label and seed are known from the RFC and the transript of the handshake, so s1 is the only unknown input to P_MD5. By iterating over all 2^{64} possible values for s1, and comparing the output of P_MD5 with the known keystream, we can identify the correct value for s1. A similar attack applies to s2.

Problem 11.6 TLS 1.3

Why can the Certificate message in TLS 1.3 be encrypted, and why is this impossible in TLS 1.2?

Solution

In TLS 1.3, the order of sending DH shares is reversed with respect to TLS 1.2: The client already sends its DH share as an extensin in the ClientHello message. So after choosing its own DH ephemeral value, the server can already derive the handshake encyption keys and encrypt the certificate. Please note that the DH share from the server must in any case be sent in the clear.

Problem 11.7 TLS 1.3 0RTT

In Figure 11.8, the client and server do not share a common key before the handshake starts. How is it possible for the client to send encrypted data along with the ClientHello message?

Solution

In a previous handshake, the TLS 1.3 server used the now integrated session ticket mechanisms to outsource the storage of the shared key k_{CS} to the client. This value k_{CS} was derived as the exporter_master_secret in that previous session (Figure 11.7); the client stored k_{CS} in a local database, togehter with an ecrypted version of it, which was sent in the NSTM message. The client can use k_{CS} as the PSK in Figure 11.7, and derive

the client_early_traffic_secret to encrypt data to send along with the ClientHello message (0-RTT data).