طَبَقَةُ مِقْبَسِ رَجُلِ الكهفِ (طَمِرَك) إصدارة 3.1.0

De Holbewoner Socket Laag (HSL) Versie 0.1.3

The Caveman Socket Layer (CSL) Version 0.1.3

إصلاح الإنحطاط في إس إس إل أو تي إل إس

Festsetzung der Dekadenz in TLS/SSL

Fixing the Decadence in SSL/TLS

Caveman
email: toraboracaveman@gmail.com
website: `https://github.com/al-caveman/csl`

August 6, 2017

Abstract

Basically, TLS/SSL sucks. They do too much dance, and PKI and shit. Result? Too much hard-to-spot real issues, and you end up needing to trust over 300 CAs, which is probably more than the HTTPS websites that you visit per year that you care about. If you think about it, you probably care about your email, eBay, Amazon, online banking, etc, which –if you list– you will realize that they are less than 300 many SSL/TLS services that you visit. So it is practically more scalable for you to simply white-list special keys per "secure" service that you use than white-listing 300+ CAs. This is funny, cause one of the points of white-listing CAs is that you will, supposedly, never need to white-list more cause there are fewer CAs than there are secure services that you care about. But it's funny that the CAs that you white-list in your OS are more than the secure services that you care about. So the CA scalability argument is just tossed right there. Total bullshit. Fuck that. Here I fix that by proposing the Caveman Socket Layer (CSL) which has two key properties: 1) simple, and 2) works. Because of its simplicity, you will be less likely to end up having security holes because of weird unpredictable mathematical voodoo. E.g. in RSA, if your random sequences, or the random sequences of the other end, are shit, then your entire security is shit. CSL fixes that and more. Plus, CSL works just as securely, if not better.

Contents

1 Overview

We got a client, and a server that's identified as S and is listening on port P. The identifier S could be the domain name, IP address, etc. The client chooses how it identifies the server based on client's policies. Client wishes to communicate with server. Procedure goes as follows:

1. Server, when initializing the CSL service, needs to generate a good random sequence. Perhaps by using `/dev/random`, or whatever other method. We don't care about that. All we need, get enough random bits. How many bits are enough? This to be decided by the DevOps at hand. But for now, we get a random sequence, which we call R_s for now.

2. Then, after we have R_s figured out, we can start the CSL service and have it listen on port P.

3. If there is no CSL caching entry in the client about S:

   (a) Client warns user "*S is a new server, wanna learn its CSL fingerprint?*". Unless user chooses "*yes*", process will terminate right here.

   (b) Client uses S to identify how to communicate with the server. E.g. if S is the domain name, client performs a DNS lookup to get the IP address of the server.

   (c) Client opens TCP (or UDP? Whatever) connection to port P.

   (d) Client and server do the Diffie-Hellman (DH) dance to agree on a an encryption key k. From now and on, all exchanged messages by the client or the server are encrypted by using some symmetric encryption function e (e.g. e could be some variant of AES?) using key k:

      i. Client generates its own random sequence R_c.

      ii. Client sends the following TLV tuple: (init, n, R_c) to server via this connection, where "init" denotes that this is the initialization phase, n denotes size of R_c probably in octets, or whatever other more convenient unit.

iii. Server reads R_c, and sends the TLV $\big(\text{initre}, n, h(R_s{+}R_c)\big)$, where h is some lovely hashing function that tickles your fancy (e.g. SHA3), and + denotes concatenation.

(e) Client then locally caches the following information:

- S.
- R_c.
- $h(R_s + R_c)$.

(f) Client and server then proceed communicating with each other using one-time pad-*ish* encryption scheme (except for using a seeded secure PRNG, CPRNG, instead of a truly random sequence) that relies on the following indefinitely long random sequence that is defined in n long chunks of random bytes/octets/whatever unit we agreed about earlier, such that the i^{th} chunk of random bits, c_i, is defined as follows:

$$c_i = \begin{cases} e(c_{i-1}) & \text{if } i > 1 \\ h(R_s + R_c) & \text{else} \end{cases} \tag{1}$$

Note 1: total entropy we have here depends on n, which affects the size of R_c and R_s. So if we choose large n, the more security we will have.

Note 2: these random bits are not exchanged during the connection, or per connection. They are only generated locally off the initial sequence $c_1 = h(R_s{+}R_c)$, which is taken from the cache as exchanged during the initialization phase when S was not found in the local cache.

Note 3: the generation of the seeded CPRNG can, obviously, be done in an online manner as you are sending more and more data.

(g) Upon connection termination, while using the encryption one-time pad-*ish* session above, the following is done:

i. Client generates a new random sequence, R_d.
ii. Client sends TLV tuple (init, n, R_d) to server.
iii. Server responds by $h(R_s + R_d)$.
iv. Client updates its local cache regarding server S as follows:
- S.
- $R_c := R_d$.
- $h(R_s + R_c) := h(R_s + R_d)$.

where := is assignment.

If there is a CSL caching entry in the client about S:

(a) Client connects to S at port P.

(b) Client sends TLV tuple (auth, n, R_c) to server.

(c) Client then sends the following TLV, encrypted using the one-time pad-*ish* algorithm described above and the cached R_c and $h(R_s + R_c)$ values: (init, n, R_c).

(d) Within this one-time pad-*ish* encryption channel, the server responds: $\hat{h}(R_s + R_c)$

(e) If $\hat{h}(R_s + R_c) \neq h(R_s + R_c)$:

i. Warn user and terminate connection. Perhaps maybe ask user if he/she wants to re-negotiate a new server fingerprint? This is left to the user interface side, but for now, we can say at least "warn".

Else, i.e. if $\hat{h}(R_s + R_c) = h(R_s + R_c)$:

i. Client generates a new random sequence R_e.

ii. Client sends (init, n, R_e) to server.

iii. Server replies $h(R_s + R_e)$.

iv. Client updates cache.

v. Client and server communicate using the one-time pad-*ish* algorithm with the indefinitely long CPRNG that is generated off the seed $h(R_s + R_e)$.

vi. Upon connection termination, the client updates the cache entry.

Note 4: note how the client and server avoid the use of DH for when there is a CSL cache entry. DH was only used when there was no CSL cache entry. This should bring more speed gains.

Note about SNI: everything happens as usual, except for sending the TLV $(\text{fwd}, n, \text{www.domain.com})$ first.

2 Messages Format

All CSL data are communicated using messages that are formated as per the TLV tuple | t (8 bits) | n (32 bits) | d (length bits) |, with large-endian encoding.

3 Message Types

3.1 Initial Encryption Channel

- $t = 0$.
- $n \in \{1, 2, \ldots, 2^{32}\}$.
- $d \in \{0, 1, \ldots, 8\}^n$.

- Usage: used to signal that the sending peer is interested in finding a shared key based on d and DH, and then to start communicating over a symmetric encrypted channel that uses the shared key.
- Behaviour:
    1. Client starts the DH process locally to calculate the number that is to be publicly shared with the server. This number is sent as d_c along with its size n_c using this message type.
    2. When the server receives this message, it, too, starts its own DH process to send its own publicly-shared number as d_s along with its length n_s to the client, by using the same message type. The server then computes the shared key k_s. Now, any data the server receives from the client must be encrypted by k_s, and any data the server sends to the client must be encrypted by k_s.
    3. When the client receives the number d from the server, the client then also computes the shared key k_c. Now, any data the client receives from the server must be encrypted by k_c, and any data the client sends to the server must be encrypted by k_c. Thank to the way DH works, we know that $k_s = k_c$, and a symmetric encryption channel can start.

4 Discussion

If you think about it, you will realize that this is better than SSL/TLS for practical purposes for whatever SSL/TLS is used for today. You can ask me questions via email or IRC. You can get IRC info via email.

Minor refinements are needed about the message formats, or about how new CSL server fingerprints are negotiated. But the general idea remains the same. If you think deep enough, you'd realize that this is superior to SSL/TLS as it avoids the stupid PKI architecture, much faster, and much simpler algorithm that significantly minimizes the chance of leaving mathematical loopholes undetected.