Pseudo Security of Pseudo Constant Time Implementations

Anonymous Author(s)

ABSTRACT

TBD

CCS CONCEPTS

• **Security and privacy** → Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper;

KEYWORDS

template; formatting; pickling

1 IMPLEMENTATION BUGS IN LUCKY13 COUNTERMEASURES

Pseudo constant time countermeasures are very hard to get right and maintain over time. This is due both to the possibility of finding novel variants of the original attacks, and the need to manually check the timing implications of adding new features. In contrast, real constant time implementations are more robust against novel attack variants, and bugs created by supporting new features will likely be found by unit-testing. TLS 1.2 [9] added new ciphers suites based on CBC-mode for encryption and HMAC-SHA-384 for integrity. The SHA-384 hash function is considered more secure, and also has better performance on 64-bit processors, than the previously supported SHA-1 and SHA-256 algorithms. We tested if TLS implementations supporting HMAC-SHA-384 are vulnerable to timing attacks similar to the one described in [2]. All of the constant-time implementations that we checked (OpenSSL, BoringSSL, NSS) were secure. However, all of the "pseudo" constant time implementations (i.e. those only ensuring a constant number of compression function calls) had bugs making them vulnerable to attack. The reason for the bugs is that, although the SHA-384 cipher suites were added, the code responsible for adding dummy compression function calls was not updated correctly. Specifically, SHA-384 has a 128-byte block size (compared to the 64-byte blocks of SHA-256), and encodes the message length using 16 bytes (compared to 8 bytes in SHA-256). All of the extra compression function call calculations have hard-coded values appropriate for SHA-256 but not SHA-384, resulting in them using a non-constant number of calls to the SHA-384 compression function. We explain in more detail below for each of the four "pseudo" constant time implementations we studied; since the bugs are easily fixed, we do not go into great detail on how each bug leads to a plaintext recovery attack.

1.1 GnuTLS Implementation

Although the function `dummy_wait` (see Listing 1 in Appendix A) uses the correct hash block size, it also uses the hard-coded number "9". This comes from at least 1 byte for the the hash function padding and the 8 bytes used to encode the hashed message length for SHA-256. However, in SHA-384, the message length is encoded using 16 bytes, and so the correct value should be "17" rather than "9". The code includes a comment warning that this is a hash-specific fix, but it was apparently not corrected when the SHA-384 cipher suites were added.

Even more surprisingly, we discovered that the GnuTLS Implementation is vulnerable to a timing attack when SHA-256 is selected as the hash algorithm in HMAC, despite this having being patched in response to Lucky 13 [21]. The function `dummy_wait` can add at most one call to the hash compression function. However, the attack described in Section 2.3 creates a padding oracle that distinguishes between a valid pad of a large length (PadLen > 240) and an invalid padding (PadLen = 0). In that case, for GnuTLS, there will be a timing difference of 3 calls to the compression function of SHA-256, that is 3 times larger than the timing difference in the original Lucky 13 attack [3].

2 A PADDING ORACLE BASED ON TLS RECORD CACHE ACCESS PATTERN

The original Lucky 13 attack [3] exploited the time difference of the TLS record verification process for valid and invalid padding. As a mitigation, all the pseudo constant time TLS implementations added dummy compression function calls that causes the total number of compression function calls to be independent of the padding length. However, unlike proper constant time TLS implementations, the cache access pattern to the data structure holding the TLS record is still dependent on the padding length. We can exploit this cache access pattern with our novel “Just in Time Priming”attack to restore the original padding oracle of [3] in several TLS implementations – XXX , GnuTLS and . All implementations were patched against Lucky 13 [3]. We will again use a PRIME+PROBE [24] cache attack, assuming a cache side channel as in Liu et.al. [17]. Our attack works on HMAC using both SHA-384 and SHA-256, even if all the bugs in Section 1 were fixed, and all previous variants of Lucky 13 were patched correctly.

2.1 Attack Preliminaries

All of the vulnerable implementations follow this general code flow for constant time decryption:

(1) Decrypt the message, accessing all the bytes in the TLS record.
(2) Perform constant time checking of the TLS record padding, assume zero-length padding if the padding is not valid. All of the final 256 bytes of the TLS record are accessed.
(3) Calculate HMAC on the decrypted TLS record payload (excluding the padding). All bytes in the decrypted TLS record are accessed, except for the padding bytes at the end.
(4) Add extra dummy compression functions calls to make the number of calls the same in every case. The data input to these function calls is obtained from the start of the TLS record or from a dummy memory buffer. The padding bytes of the TLS record are not accessed (except for messages that are shorter than the hash block size).

In our attack, we will try to distinguish between two cases: long valid padding and invalid padding. We will explain in Section 2.3

Algorithm 1 Message Access Attack

```
1:  function MessageAccessPadOracle(Valid TLS records, Attack TLS record)
2:      LastBytesCache ← FindPtrCache(Valid TLS record[End])
3:      Send attacker's TLS record to target
4:      Delay to synchronize to the start of the HMAC verification
5:      Prime(MsgCache)      ▷ evict end of TLS record from cache
6:      Delay till maximum time for HMAC calculation
7:      if Probe(MsgCache) then          ▷ end of TLS record was accessed
8:          return 0                     ▷ padding was invalid
9:      else
10:         return 1                     ▷ padding was valid
11:     end if
12: end function
```

how an oracle yielding this information can be used to recover plaintext bytes. Consider the cache access pattern from the beginning of the HMAC verification. In the case of invalid padding, the code typically assumes zero-length padding, and the HMAC verification will access all of the TLS record bytes (possibly excluding the last byte). However, if the padding is valid and large (e.g. PadLen = 255, in which case there are 256 bytes of padding), the HMAC verification will not access the last PadLen + 1 bytes of the TLS record.

2.2 "Just in Time Priming"Attack Description

Our "Just in Time Priming"attack, then, exploits this difference in the access pattern using a PRIME+PROBE [24] cache attack. We synchronize the PRIME part of the attack, to run in parallel to the HMAC verification process, that is, after the padding check but before the HMAC verification is done. The maximum TLS record size is ca. 2^{14} bytes, corresponding to about 2^8 compression function calls for a 64-byte hash block size. By working with ciphertexts of this size, we can force the HMAC verification to take a long time to complete. This makes the synchronization of the attack relatively easy.

The attack (described in Algorithm 1) has four main parts:

(1) Finding the cache sets containing the last bytes of the TLS record.
(2) Sending the attack TLS record. The TLS record is constructed to have long valid padding, except possibly in the first padding byte. This is the byte we try to recover in the attack.
(3) Delaying till the HMAC verification begins. This occurs after the decryption and padding check is finished (and takes a constant amount of time regardless of the padding).
(4) In parallel to the HMAC verification, we Prime the end of the TLS record to evict it from the cache.
(5) After the end of the HMAC verification calculation, we Probe the cache line of the last few bytes of the TLS record. If it was accessed, we know that the padding was invalid; otherwise it was valid.

2.3 Constructing Attack TLS Records

It remains to explain how we construct the TLS records used in the attack, and how we use the results of the oracle to recover plaintext bytes (HTTP cookie bytes in this case). We rely on techniques first explained in [11]: we use HTTP pathname padding and the ability to choose plaintext bytes that are placed after the cookie in the HTTP request to ensure that the plaintext in the TLS record contains 16 consecutive blocks in which the first block has the form:

$$p^*||\text{"\r"}||\text{"\n"}||\texttt{0xFF}||\ldots||\texttt{0xFF},$$

and the remaining 15 blocks consist solely of values `0xFF`. Here p^* is the last byte of the cookie and the target of the first step of the attack, while "\r", "\n" represent ASCII characters inserted after the cookie by HTTP. Note that these plaintext blocks are *almost* correct padding of maximum length; of course they are incorporated into a TLS record containing an HMAC tag and correct TLS padding. Let $C^*_0, \ldots, C^*_{15}$ denote the matching ciphertext blocks in the resulting TLS record, and let C^*_{-1} denote the preceding ciphertext block.

The attack TLS record is then constructed as:

$$\texttt{HDR}||L||C^*_{-1} \oplus \Delta||C^*_0||\ldots||C^*_{15}$$

where `HDR` is a suitable header, L is a long random block sequence that brings the TLS record up to the maximum size, and Δ is a mask with bytes:

$$\delta||(\text{"\r"} \oplus \texttt{0xFF})||(\text{"\n"} \oplus \texttt{0xFF})||\texttt{0x00}||\ldots||\texttt{0x00}.$$

Here, the first mask byte creates a value $p^* \oplus \delta$ in the first position of the block decrypting C^*_0, while the second and third mask bytes force values `0xFF` in the corresponding positions. It should now be clear that, when decrypted, this TLS record will have correct padding of length 256 if and only if $p^* \oplus \delta = \texttt{0xFF}$. The attacker then uses TLS records of this form with distinct values of δ in the attack of Algorithm 1; after 128 attempts on average and 256 in the worst case, the value of δ producing correct padding will be identified.

This description explains how to recover the last byte of the cookie. Further bytes can be recovered by shifting the position of the cookie by altering the length of the pathname in the HTTP request so that the last 2, 3, ... bytes are present at the start of the block underlying C^*_0. We also update Δ as needed to force correct padding `0xFF` in all but the first byte of this block. This approach will recover up to 14 bytes of the cookie; the remaining bytes seems to remain inaccessible using these techniques (trying to extend further would push the "\r" and "\n" characters into the next block, where they could not be turned into correct padding by XOR masking).

We close this description by noting that the above attack with long padding patterns can be applied to the original Lucky 13 setting, quadrupling the timing differences there and so making them substantially easier to detect (at the cost of limiting how much plaintext can be recovered). This enhancement to Lucky 13 seems to have been missed by the authors of [3], though they used a similar idea in their distinguishing attack.

2.4 Proof of Concept on xxxx TLS implementation

We implemented out attack on the xxx TLS implementation.

2.4.1 Attack results. We implemented our attack in a similar way to the attack in Section XXXX.We prepared two types of decrypted TLS record. The first one with 256 bytes of valid padding and the second one being identical, except for the first byte of

padding which was change to a different value. We called the XXXX function multiple times with both types of data. Before each call to the function, another "attack" thread was run in parallel to preform the cache priming during the HMAC verification. After the function returned, we checked if the cache line of the last bytes in the TLS record is in the cache or not. For TLS records with valid padding we saw a hit probability of ≈ 0.025. For TLS records with invalid padding we saw a hit probability of ≈ 0.998. Using the same calculations as in Section **??** this translates to $n = 4$ and an expected total number of cache attack executions (and TLS connections) of 512 per byte.

2.4.2 GnuTLS. The same vulnerability also applies to the GnuTLS code in function `decrypt_packet` (see Listing 2 in Appendix A). If the padding is invalid the function sets `pad` to 0. For constant time, extra compression function are executed via `dummy_wait` (see Listing 1 in Appendix A). If the padding was invalid, the function does nothing and returns. If the padding was valid, but the HMAC verification fails, the extra compression function _gnutls_auth_cipher_add_auth is called multiple times with the pointer `data`, pointing to the start of the TLS record.

Note that unlike other implementations, the extra compression functions are only called when the verification process fails, so the decryption time on valid messages is not constant. This may leak the real size of the encrypted messages, but cannot be used to recover plaintext bytes.

REFERENCES

[1] ICSI Certificate Notary. (????).

[2] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *Advances in Cryptology – EUROCRYPT 2016, Part I (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9665. Springer, Heidelberg, 622–643. https://doi.org/10.1007/978-3-662-49890-3_24

[3] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 526–540.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Thomas Peyrin (Ed.), Vol. 9783. Springer, 163–184. https://doi.org/10.1007/978-3-662-52993-5_9

[5] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.). ACM Press, 85–96.

[6] Andrey Bogdanov, Ilya Kizhvatov, Kamran Manzoor, Elmar Tischhauser, and Marc Witteman. 2016. Fast and Memory-Efficient Key Recovery in Side-Channel Attacks. In *SAC 2015: 22nd Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Orr Dunkelman and Liam Keliher (Eds.), Vol. 9566. Springer, Heidelberg, 310–327. https://doi.org/10.1007/978-3-319-31301-6_19

[7] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. 2003. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology – CRYPTO 2003 (Lecture Notes in Computer Science)*, Dan Boneh (Ed.), Vol. 2729. Springer, Heidelberg, 583–599.

[8] Liron David and Avishai Wool. 2017. A Bounded-Space Near-Optimal Key Enumeration Algorithm for Multi-subkey Side-Channel Attacks. In *Topics in Cryptology – CT-RSA 2017 (Lecture Notes in Computer Science)*, Helena Handschuh (Ed.), Vol. 10159. Springer, Heidelberg, 311–327.

[9] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). (Aug. 2008), 104 pages. https://doi.org/10.17487/RFC5246 Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

[10] Joey Dodds. 2016. Verifying s2n HMAC with SAW. (2016). https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/

[11] Thai Duong and Juliano Rizzo. 2011. Here come the ⊕ Ninjas. Unpublished manuscript. (2011).

[12] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.), Vol. 9721. Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[13] P. Gutmann. 2014. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard). (Sept. 2014), 7 pages. https://doi.org/10.17487/RFC7366

[14] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[15] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 549–564. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[16] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *CoRR* abs/1801.01207 (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207

[17] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 605–622. https://doi.org/10.1109/SP.2015.43

[18] Colm MacCarthaigh. 2015. AWS Security Blog - s2n and Lucky 13. (2015). https://aws.amazon.com/blogs/security/s2n-and-lucky-13/

[19] Daniel P. Martin, Luke Mather, Elisabeth Oswald, and Martijn Stam. 2016. Characterisation and Estimation of the Key Rank Distribution in the Context of Side Channel Evaluations. In *Advances in Cryptology – ASIACRYPT 2016, Part I (Lecture Notes in Computer Science)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10031. Springer, Heidelberg, 548–572. https://doi.org/10.1007/978-3-662-53887-6_20

[20] Daniel P. Martin, Jonathan F. O'Connell, Elisabeth Oswald, and Martijn Stam. 2015. Counting Keys in Parallel After a Side Channel Attack. In *Advances in Cryptology – ASIACRYPT 2015, Part II (Lecture Notes in Computer Science)*, Tetsu Iwata and Jung Hee Cheon (Eds.), Vol. 9453. Springer, Heidelberg, 313–337. https://doi.org/10.1007/978-3-662-48800-3_13

[21] Nikos Mavrogiannopoulos. 2013. Time is money (in CBC ciphersuites). (2013). https://nikmav.blogspot.co.uk/2013/02/time-is-money-for-cbc-ciphersuites.html

[22] Ralph Charles Merkle, Ralph Charles, et al. 1979. Secrecy, authentication, and public key systems. (1979).

[23] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE Bites: Exploiting The SSL 3.0 Fallback. (September 2014). https://www.openssl.org/~bodo/ssl-poodle.pdf

[24] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006 (Lecture Notes in Computer Science)*, David Pointcheval (Ed.), Vol. 3860. Springer, Heidelberg, 1–20.

[25] Kenneth G. Paterson and Nadhem J. AlFardan. 2012. Plaintext-Recovery Attacks Against Datagram TLS. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society.

[26] Stephen Schmidt. 2017. AWS Security Blog - s2n Is Now Handling 100 Percent of SSL Traffic for Amazon S3. (2017). https://aws.amazon.com/blogs/security/s2n-is-now-handling-100-percent-of-of-ssl-traffic-for-amazon-s3/

[27] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 1492–1504.

[28] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS.... In *Advances in Cryptology – EUROCRYPT 2002 (Lecture Notes in Computer Science)*, Lars R. Knudsen (Ed.), Vol. 2332. Springer, Heidelberg, 534–546.

[29] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. 2013. An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks. In *SAC 2012: 19th Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Lars R. Knudsen and Huapeng Wu (Eds.), Vol. 7707. Springer, Heidelberg, 390–406. https://doi.org/10.1007/978-3-642-35999-6_25

[30] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, Kevin Fu and

Jaeyeon Jung (Eds.). USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

A SOURCE CODE

Listing 1: GnuTLS's extra compression call calculation

```
static void dummy_wait(record_parameters_st * params, gnutls_datum_t * plaintext,
        unsigned pad_failed, unsigned int pad, unsigned total){
    ...
    /* This is really specific to the current hash functions.
    * It should be removed once a protocol fix is in place.
    */
    if ((pad + total) % len > len - 9 && total % len <= len - 9) {
        if (len < plaintext->size)_
            gnutls_auth_cipher_add_auth(&params->read.cipher_state,
            plaintext->data, len);
```

Listing 2: GnuTLS's pad check and HMAC verification

```
decrypt_packet(gnutls_session_t session, gnutls_datum_t * ciphertext,
    gnutls_datum_t * plain, content_type_t type, record_parameters_st * params,
    gnutls_uint64 * sequence) {
    ...
    pad = plain->data[ciphertext->size - 1];     /* pad */
    ...
    for (i = 2; i <= MIN(256, ciphertext->size); i++) {
        tmp_pad_failed |= (plain->data[ciphertext->size - i] != pad);
        pad_failed |= ((i <= (1 + pad)) & (tmp_pad_failed));
    }

    if (unlikely (pad_failed != 0 || (1 + pad > ((int) ciphertext->size - tag_size)))) {
        /* We do not fail here. We check below for the
        * the pad_failed. If zero means success.
        */
        pad_failed = 1;
        pad = 0;
    }
    length = ciphertext->size - tag_size - pad - 1;
    ...
    ret = _gnutls_auth_cipher_add_auth(&params->read.ctx.tls12, plain->data, length);
    if (unlikely(gnutls_memcmp(tag, tag_ptr, tag_size) != 0 || pad_failed != 0)) {
        /* HMAC was not the same. */
        dummy_wait(params, plain, pad_failed, pad, length + preamble_size);
```