Network Working Group
Internet-Draft
Expires: August 11, 2013

A. Pironti
INRIA Paris-Rocquencourt
N. Mavrogiannopoulos
KU Leuven
February 7, 2013

Length Hiding Padding for the Transport Layer Security Protocol
draft-pironti-tls-length-hiding-00

Abstract

This memo proposes length hiding methods of operation for the TLS protocol. It defines a TLS extension to allow arbitrary amount of padding in any TLS ciphersuite, and it presents guidelines and a reference implementation of record fragmentation and padding so that the length of the exchanged messages is effectively concealed within a given range of lengths. The latter guidelines also apply to the standard TLS padding allowed by the TLS block ciphers.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 11, 2013.

Copyright Notice

Table of Contents

1. Introduction

When using CBC block ciphers, the TLS protocol [RFC5246] provides means to frustrate attacks based on analysis of the length of exchanged messages, by adding extra pad to TLS records. However, the TLS specification does not define a length hiding (LH) method for applications that require it. In fact, current implementations of eager fragmentation strategies or random padding strategies have been showed to be ineffective against this kind of traffic analysis [LH-PADDING].

This document proposes a TLS extension to allow arbitrary amount of padding in any TLS ciphersuite, and in addition it presents guidelines and a reference implementation of record fragmentation and padding so that the length of the exchanged messages is effectively concealed within a range of lengths provided by the user of the TLS record protocol. The latter guidelines also apply to the standard TLS padding allowed by the TLS block ciphers.

The proposed extension also eliminates padding oracles (both in errors and timing) that have been plaguing standard TLS block ciphers [CBCTIME] [DTLS-ATTACK]

When using standard TLS block cipher padding, the goals of LH for TLS are the following:

1. Length-Hiding: use message fragmentation and the allowed extra padding for block ciphers to conceal the real length of the exchanged message within a range of lengths chosen by the user of the TLS record protocol. All messages sent with the same range use the same network bandwidth, regardless of the real size of the message itself.

2. Interoperability: an implementation sending length-hidden messages correctly interoperates with non LH-aware implementations of TLS.

3. Efficiency: the minimum required amount of extra padding is used, and the minimum number of required fragments is sent.

By design, in the standard TLS block cipher mode, only a limited amount of extra padding can be carried with each record fragment, and this can potentially require extra fragmentation for wide ranges. Moreover, no LH can be implemented for stream ciphers. To overcome these limitations, the TLS extension proposed in this document enables efficient LH both for block and stream ciphers (at the partial cost of interoperability, as both parties must implement the extension).

2. Terminology

This document uses the same notation and terminology used in the TLS Protocol specification [RFC5246].

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. TLS Extension: Extended Record Padding

The TLS extended record padding is a variant of the TLS record protocol where every record can be padded up to 2^14 bytes, regardless of the cipher being used.

3.1. Extension Negotiation

In order to indicate the support of the extended record padding, clients MUST include an extension of type "extended_record_padding" to the extended client hello message. The "extended_record_padding" TLS extension is assigned the value of TDB-BY-IANA from the TLS ExtensionType registry. This value is used as the extension number for the extensions in both the client hello message and the server hello message. The hello extension mechanism is described in [RFC5246].

This extension carries no payload and indicates support for the extended record padding. The "extension_data" field of this extension are of zero length in both the client and the server.

The negotiated record padding applies for the duration of the session including session resumption. A client wishing to resume a session where the extended record padding was negotiated SHOULD include the "extended_record_padding" extension in the client hello.

3.2. Record Payload

The translation of the TLSCompressed structure into TLSCiphertext remains the same as in [RFC5246]. When the cipher is BulkCipherAlgorithm.null, the 'fragment' structure of TLSCiphertext also remains unchanged. That is, for the TLS_NULL_WITH_NULL_NULL ciphersuite and for MAC-only ciphersuites this extension has no effect. For all other ciphersuites, the 'fragment' structure of TLSCiphertext is modified as follows.

```
        stream-ciphered struct {
            opaque pad<0..2^14>;
            opaque content[TLSCompressed.length];
            opaque MAC[SecurityParameters.mac_length];
        } GenericStreamCipher;

        struct {
            opaque IV[SecurityParameters.record_iv_length];
            block-ciphered ciphered struct {
                opaque pad<0..2^14>;
                opaque content[TLSCompressed.length];
                opaque MAC[CipherSpec.hash_size];
            };
        } GenericBlockCipher;

        struct {
            opaque nonce_explicit[SecurityParameters.record_iv_length];
            aead-ciphered struct {
                opaque pad<0..2^14>;
                opaque content[TLSCompressed.length];
                opaque MAC[CipherSpec.hash_size];
            };
        } GenericAEADCipher;
```

The padding can be filled with arbitrary data, and it is authenticated as part of the MAC. For block ciphers, the length of the pad MUST be such that the total length (i.e., the pad, the content and the MAC) are a multiple of the block size.

For the various ciphers the data are authenticated as follows.

Standard Stream Ciphers:

```
            MAC(MAC_write_key, seq_num +
                TLSCompressed.type +
                TLSCompressed.version +
                TLSCompressed.length +
                TLSCiphertext.fragment.GenericStreamCipher.pad +
                TLSCompressed.fragment);
```

Block Ciphers:

```
            MAC(MAC_write_key, seq_num +
                TLSCompressed.type +
                TLSCompressed.version +
                TLSCompressed.length +
                TLSCiphertext.fragment.GenericBlockCipher.pad +
                TLSCompressed.fragment);
```

AEAD Ciphers:

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce,
                             pad + plaintext,
                             additional_data);
```

Implementation note: With block and stream ciphers, in order to avoid padding oracles, decryption, MAC verification and payload decoding SHOULD be executed in the following order:

1. Decrypt TLSCiphertext.fragment.

2. Verify the MAC.

3. Split plaintext from pad.

4. A Length Hiding Mechanism for TLS

In order to send length-hidden messages, a user of a LH-TLS implementation provides the plaintext to be sent together with a range (low,high), meaning that an attacker can at most learn that the real plaintext length is between low and high.

The LH mechanism described in the rest of this document applies both to standard TLS block ciphers and the extended record padding of section Section 3.

4.1. Range Splitting

Not all user-provided ranges can be conveyed in a singe TLS record fragment. A LH-TLS implementation uses a fragmentation algorithm that takes a message with a desired length range (low,high) and breaks it up into n suitably sized ranges each of which can be conveyed in a single TLS record fragment. The Range and FragmentRange are defined as follows.

```
struct {
  uint32 low;
  unit32 high;
} Range;

struct {
  uint16 low;
  unit16 high;
} FragmentRange;
```

If the difference between Range.high and Range.low is greater than the maximum allowed padding size for a single fragment, or if their value is greater than the maximum fragment size, the given range must be split into multiple smaller FragmentRange structures each of which can be conveyed into a single TLS record.

Range.low MUST be less or equal to Range.high. Declaring Range.low and Range.high as uint32 allows to send messages of length up to 2^32: TLS implementations MAY use larger data types for these fields. A FragmentRange, that can be conveyed in one record, MUST have both values of FragmentRange.low and FragmentRange.high not exceeding 2^14 (or the negotiated maximum value of TLSPlaintext.length [RFC6066]).

A TLS implementation applies the range splitting algorithm starting from the user-provided Range structure, resulting into a sequence of FragmentRange structures. For each FragmentRange structure, it transmits a TLS record adhering into the limits of the corresponding FragmentRange. When a block cipher is in use, on each record the

implementation computes n bytes of minimal padding (the minimum amount of padding required to get block alignment) pretending the length of the plaintext is FragmentRange.high. The total padding added to the current fragment is finally n plus the difference between Range.high and the real plaintext length.

This document does not mandate any specific algorithm to split a Range into multiple FragmentRange ranges. The only constraint is that the sum of the obtained sequence of ranges equals the range given as input. Implementations may use non-deterministic splitting algorithms to change the shape of the traffic each time messages with the same range are exchanged.

A reference range splitting algorithm is provided in the following.

```
// The maximum allowed TLSPlaintext.length
uint16 FS = 2^14;
// Maximum padding size:
// p = 255 for standard TLS block ciphers;
// p = 2^14 for extended record padding
uint16 PS = p;
// Length of the padlen:
// pl = 1 for standard TLS block ciphers;
// pl = 2 for extended record padding
uint8  PL = pl;
// Note: Block size is 0 for stream ciphers
uint8  BS = SecurityParameters.block_length;
// MAC size
uint8  MS  = SecurityParameters.mac_length;

/* Returns the maximum pad that can be added for a fragment,
 * given that at least 'len' bytes of plaintext will be
 * transferred.
 */
uint16 max_lh_pad(uint16 len)
{
    uint16 this_pad = min(PS,FS-len);
    if (BS == 0) {
        return this_pad;
    } else {
        uint8 overflow = (len + this_pad + MS + PL) % BS;
        if (overflow > this_pad) {
            return this_pad;
        } else {
            return this_pad - overflow;
        }
    }
}
```

```
FragmentRange split_range(Range *total)
{
    FragmentRange f;

    if (total.high == total.low) {
        // "Point" range, no real LH to do:
        // just implement standard fragmentation.
        uint16 len = min(total.high,FS);
        f.low  = len;
        f.high = len;
        total->low  -= len;
        total->high -= len;
    } else if (total.low >= FS) {
        // More bytes to send than a fragment can handle:
        // send as many bytes as possible.
        f.low  = FS;
        f.high = FS;
        total->low  -= FS;
        total->high -= FS;
    } else {
        // We are LH: add as much padding as necessary
        // in the current fragment.
        uint16 all_pad = max_lh_pad(total->low);
        all_pad = min(all_pad, total->high - total->low);
        f.low  = total->low;
        f.high = total->low + all_pad;
        total->low   = 0;
        total->high -= total->low + all_pad;
    }

    return f;
}
```

This algorithm if called multiple times creates a list of FragmentRange structures, carrying all the payload up to Range.low, followed by a sequence of fragments carrying either padding or the remaining part of the message that exceeds Range.low.

4.1.1. Fragmenting Plaintext into Records

There are many ways to fragment the message content across a sequence of FragmentRanges. This document does not mandate any fragmentation algorithm. In the following, a fragmentation algorithm that tries to put as many bytes as possible in the first fragments is provided.

```
/* len: The total real plaintext length to be sent.
 * r0: a range that can be conveyed in one fragment,
 *     as returned by split_range.
 * r1: the remaining range used to send the remaining data
 * Returns: the number of bytes of plaintext to be sent
 *          in the next fragment with range r0.
uint16 fragment(uint32 len, FragmentRange r0, Range r1)
{
    return min(r0.high, len - r1.low);
}
```

4.1.2. Adding the Length Hiding Padding

If 'len' is the real plaintext length to be sent in a record fragment with range FragmentRange, a LH-TLS implementation MUST add at least FragmentRange.high - len bytes of padding to that record fragment (plus, if needed, some additional padding required to get block alignment).

If the split_range and fragment functions above are used, then the difference FragmentRange.high - len is always smaller than the maximum available padding size (including further block alignment padding).

4.1.3. A Length Hiding API

Finally, a LH-aware TLS implementation MAY use the algorithms described in sections Section 4.1 and Section 4.1.1 to offer a LH TLS API similar to the following, where it is assumed that a TLS_send(data, len, target_length) function sends a single TLS record fragment adding the necessary padding to match the target_length, as explained in section Section 4.1.2.

```
uint32 message_send(opaque data, Range total)
{
    FragmentRange current;
    uint16 current_len, sent = 0;

    while (total.high != 0) {
        current = split_range(&total);
        next_len = fragment(data.length - sent, current, total);
        sent += TLS_send(&data[sent], next_len, current.high);
    }

    return sent;
}
```

This interface requires the TLS implementation to internally buffer

the entire application message. Alternatively, a LH TLS implementation MAY directly expose the split_range and fragment functions to the user, to avoid internal buffering. Note that it is only necessary to know the desired plaintext range to execute the split_range function, not the real plaintext size nor its content.

4.2. Applicability

If a TLS-LH mechanism is used in a TLS session, then TLS record protocol compression MUST be disabled. Compression is known to leak substantial information about the plaintext, including its length [COMPLEAK], which defeats the purpose of LH. Moreover, since in TLS compression happens after fragmentation, and the compression ratio is not known a priori, it is impossible to define a precise fragmentation strategy when compression is in place.

Length hiding can only work when some padding can be added before encryption, so that an attacker cannot distinguish whether the encrypted data are padding or application data. Hence, LH can only be used with block ciphers in standard TLS, and with any cipher when the extended record padding is used. In any case, length hiding MUST NOT be used with TLS_NULL_WITH_NULL_NULL or MAC-only ciphersuites.

5. Security Considerations

The LH scheme described in this document is effective in hiding the length of the exchanged messages, when an attacker observes the total bandwidth exchanged by a client and server using TLS. Crucially, the split_range algorithm, which determines the traffic shape and total bandwidth, MUST NOT depend on the real message length, but only on the Range.low and Range.high values, which are public.

Similarly, only the application knows when the recipient of the message is expected to react, upon receiving the message. For example, a web browser may start loading a hyperlink contained in an HTML file, as soon as the hyperlink is received, before the HTML file has been fully parsed. By using a callback for the implementation of the fragment function, a LH-aware application using a TLS-LH library can decide how much data to send in each fragment. An application should consider the TLS LH mechanism effective only to conceal the length of the message exchanged over the network.

Yet, an application on top of TLS could easily leak the message length, by performing visible actions after a known amount of bytes has been received. Hiding the length of the message at the application level is outside the scope of this document, and is a complex information flow property that should carefully considered when designing a LH-aware implementation. Even the way the bytes are transferred from the TLS library to the application could leak information about their length.

5.1. Length Hiding with standard TLS block ciphers

Section 6.2.3.2, Implementation note, of [RFC5246] acknowledges a small timing channel, due to the MAC timing depending on the length of each TLSCiphertext.content. Usage of large ranges with the LH scheme amplifies this timing channel, up to make it exploitable [LH-PADDING], because shorter messages within a range will be processed faster than longer messages in the same range. Implementations supporting the LH scheme SHOULD implement a MAC algorithm whose execution time depends on the length of the TLSCiphertext.content plus the length of the padding, thus eliminating this timing channel.

5.2. Length Hiding with extended record padding

Since the padding is always included in the MAC computation, attacks that utilize the current CBC-padding timing channel (e.g., [DTLS-ATTACK]) are not applicable.

In a way, the extended record padding can be seen as a special way of

encoding application data before encryption (where application data given by the user are prefixed by some padding). Hence, previous security results on standard TLS block and stream ciphers still apply to the extended record padding.

5.3. Mitigating Denial of Service

The TLS protocol allows zero-length fragments of Application data, and these are exploited by the TLS length-hiding mechanism proposed in this document. If the user is notified of such zero-length fragments, normally this poses no problem. However, some TLS implementations will keep reading for the next fragment if a zero-length fragment is received. This exposes implementations (especially server-side ones) to distributed denial of service (DoS) attacks, where a network of attackers connect to the same host and send a sequence of zero-length fragments, keeping the host busy in processing them. This issue gets amplified when the "extended_record_padding" extension is used, because MAC computation includes a possibly large amount of padding.

Implementations that keep reading for the next fragment when a zero-length one is received, and that are concerned by such DoS attacks, MAY implement a DoS countermeasure. For example, they could accept 'n' zero-length fragments in a row, before notifying the user or returning an error. This conflicts with the requirements of a length-hiding mechanism, where zero-length fragments are used to conceal the real plaintext length. The value of 'n' SHOULD be chosen to satisfy the tradeoff so that it is the smallest number of fragments that can convey the required LH padding. Normally, this value is application specific, so TLS implementations that implement this mitigation SHOULD let 'n' be set by the application.

6. IANA Considerations

This document defines a new TLS extension, "extended_record_padding", assigned a value of TBD-BY-IANA (the value 48015 is suggested) from the TLS ExtensionType registry defined in [RFC5246]. This value is used as the extension number for the extensions in both the client hello message and the server hello message. The new extension type is used for certificate type negotiation.

7. Normative References

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[DTLS-ATTACK] Nadhem, N. and K. Paterson, "Plaintext-recovery attacks against datagram TLS.", Network and Distributed System Security Symposium , 2012.

[LH-PADDING] Pironti, A., Strub, P., and K. Bhargavan, "Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures.", INRIA Research Report 8067 , 2012.

[CBCTIME] Canvel, B., Hiltgen, A., Vaudenay, S., and M. Vuagnoux, "Password Interception in a SSL/TLS Channel", Advances in Cryptology -- CRYPTO , 2003.

[COMPLEAK] Kelsey, K., "Compression and information leakage of plaintext", Fast software encryption , 2002.

Authors' Addresses

Alfredo Pironti
INRIA Paris-Rocquencourt
23, Avenue d'Italie
Paris, 75214 CEDEX 13
France

Email: alfredo.pironti@inria.fr

Nikos Mavrogiannopoulos
Dept. of Electrical Engineering ESAT/COSIC KU Leuven - iMinds
Kasteelpark Arenberg 10, bus 2446
Leuven-Heverlee, B-3001
Belgium

Email: nikos.mavrogiannopoulos@esat.kuleuven.be