Lucky 13: Timing Attacks against TLS and DTLS Protocols

Martin Turk

mt5555@student.uni-lj.si

Abstract. The Transport Layer Security (TLS) protocol provides a secure channel between two applications to ensure privacy and integrity of the exchanged data. DTLS is a variant of TLS built for use cases where cost of establishing sessions is too high. With the wide adoption of the protocols many experts are seeking for their vulnerabilities. In this report we present distinguishing and plaintext recovery attacks against TLS and DTLS. The attacks exploit the well known vulnerability of CBC-mode during decryption and can–with careful statistical analysis–recover plaintext data. We overview experimental results obtained in controlled environment and discuss practicality of the attacks and some possible countermeasures.

Key words: CBC-mode encryption, TLS, DTLS, timing attack, distinguishing attack, plaintext recovery

1 INTRODUCTION

In today's world, where part of our life is basically on the Internet, data confidentiality and integrity is of great importance. The Transport Layer Security (TLS) and Datagram TLS (DTLS) protocols aim to provide just that.

TLS is the most common secure communication protocol on the Internet today. Its main task is establishing a secure end-to-end channel between applications for safe data transmission. It has originated from its predecessor–the SSL Version 3.0 protocol in 1999 and was developed by T. Dierks and C. Allen [3]. Whereas SSL was intended for safe end-to-end communication between browsers, TLS was made to be used more generally for securing a wide variety of applications such as e-commerce transactions, virtual private networks (VPN), mobile applications and much more. Today the latest version of TLS protocol is TLS Version 3.0 [7] but has not yet been completely adopted as many legacy systems still use older technology and protocols.

The DTLS protocol is based on TLS but operates on UDP instead of TCP [8] and is greatly appreciated when the cost of establishing all the TLS sessions is too high. For example, DTLS is used for video streaming where large amount of data is transmitted but is not problematic if some of the frames are not received.

TLS and DTLS are comprised of several subprotocols. The important two are (D)TLS Handshake Protocol, which is used for session key establishment, ciphersuite negotiation, authentication and more, and the Record Protocol, which uses symmetric key cryptography like block and stream ciphers to establish secure channel for data transmission. The attack described in this paper exploits the vulnerabilities of the later. The Record Protocol uses MAC-Encode-Encrypt (MEE) principle, where the MAC tag is first computed from the plaintext data. Because we concentrate on CBC-mode encryption the encoding step concatenates MAC tag, plaintext data, and encryption padding. In last step the encoded plaintext is encrypted using DES, 3DES or AES. This configuration of the protocols is referred to as MEE-TLS-CBC and is more precisely described in Section 2.

Due to popularity of the protocols scientists and experts continue to study their security and have found many attacks over the years. The cryptographic attacks against the TLS Record Protocol have largely inspired its evolution. T. Duong and J. Rizzo achieved full plaintext recovery against TLS 1.0 in 2011 with the so-called BEAST attack [6] where an attacker must first gain access to a chosen plaintext capability, perhaps by inducing the user to download malware into his browser. There are well known attacks which all emerge from the fact that the padding is added after the MAC has been computed and so form unauthenticated data in the encoded plaintext [11], [4], [10].

In Sections 3 and 4 the distinguishing attack and full plaintext recovery attacks [1] developed and published by N. J. AlFardan and K.G. Paterson in 2013 are described. A variant of the attacks exist against Amazon's implementation of TLS [2] which incorporate some of the countermeasures provided in [1] such as random waiting period in case of MAC failure. The authors M. R. Albrecht and K. G. Paterson show that those countermeasures are not sufficient. The attacks exploit well-known vulnerabilities of CBC-mode encryption [11]. Decryption needs to check if the format is valid. Validity of the format is easily leaked from

communication protocols in a chosen ciphertext attack since the receiver usually sends an acknowledgment or an error message. This is a side channel, or, in our case, timing side channel. This is because the attacks exploit the fact that, when badly formatted padding is encountered during decryption, a MAC check must still be performed, assuming zero-length pad, to prevent the known timing attacks. It turns out that zero-length pad assumption does not prevent the timing side channel. Various factors like the size of the header, MAC tag, and block can be aligned such that there will be a time difference in the time that it takes to process TLS records with valid and invalid padding pattern. This difference is measured as error message appearance on the network. This timing side channel can be used many times to obtain many timing samples. With use of some statistics plaintext data can be revealed. Some practical considerations of the attacks are mentioned and some solutions are provided as well.

The last chapter presents some results that N. J. AlFardan and K.G. Paterson obtained in [1] in controlled environment. We conclude that Lucky 13 against TLS is most likely not possible in real-world scenario (due to requiring too many TLS sessions), whereas DTLS is vulnerable against it.

2 TLS AND DTLS PROTOCOLS

The (D)TLS cryptographic protocol is probably the most widely-used client-server secure communication protocol on the Internet today. Its primary objective is to provide privacy, authentication, and data integrity between two communicating applications, and overall end-to-end security against an active man-in-the-middle (MITM) attacker. Given the immense number of communication channels established every second, it is fair to say that (D)TLS is one of the most important real-world deployments of cryptography that exist.

It was originally deployed as a Secure Socket Layer (SSL) Protocol by a company named Netscape. Later when SSL Version 3.0 was introduced the IETF adopted the protocol, upgraded it, and specified it as (D)TLS 1.0 which corresponds to SSL 3.1. Although TLS 1.3 has been released in 2018 this report focuses on TLS 1.2 as it still remains the *de facto* standard. TLS 1.3 is more efficient in computation and provides better security but it takes time for its widespread adoption.

The following subsections provide a brief description of how (D)TLS protocol works and what it consists of. The ephasis is put on the **(D)TLS Record Protocol** which uses CBC-mode encryption that the attack described in this report exploits. Both protocols are built nearly the same and for that reason the TLS protocol is described and the differences between the two are stated where needed. More detailed descriptions of TLS and DTLS protocols can be found in [5] and [9], respectively.

2.1 TLS Protocol Architecture

The TLS protocol operates as an intermediate layer between the transport and the application layer, and consists of several (sub)protocols distributed in two layers (see Figure 1). The two core (sub)protocols are the *TLS handshake protocol*, which is responsible for authentication and key generation, and the *TLS record protocol*, which provides a secure channel for data transmission.

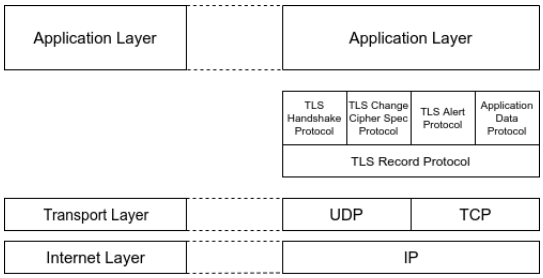

Figure 1.: The two-layer structure of TLS (sub)protocols.

2.2 TLS Handshake Protocol

The TLS handshake protocol is layered on top of TLS record protocol (see 2.5) and is executed as soon as client-server connection is established. It allows a client and a server to authenticate each other, negotiate cipher suites and (optionally) compression methods, decide on protocol version, and generate secrets (private keys) using public-key cryptography algorithms. The protocol comprises four sets of messages that are exchanged between client and a server as seen in 4.

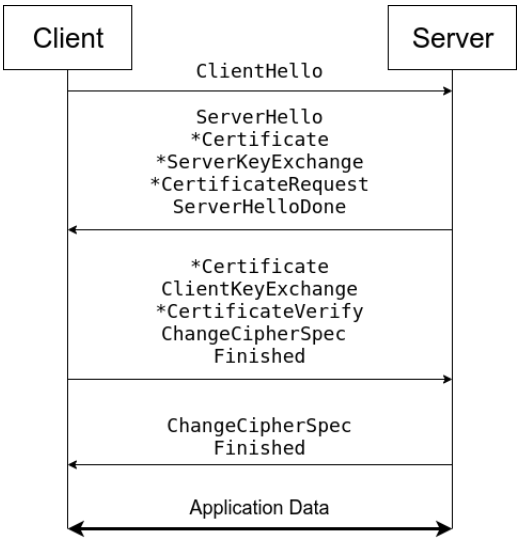

Figure 2.: The TLS handshake protocol. Star symbol (*) indicates optional or situation-dependent messages that are not always sent. The execution of the protocol is from top to bottom.

Once the TLS handshake is complete and the client and server have established a secure connection, they can start exchanging application-layer data packages using **application data protocol**. In short, application data protocol takes application data and feeds it into the TLS record protocol for fragmentation, compression, encryption, and encapsulation. The produced TLS records are then sent to the other party, where the application data is decrypted, verified, decompressed, and reassembled.

2.3 TLS Change Cipher Spec Protocol

The TLS change cipher spec protocol indicates the change in ciphering strategy–i.e., notifies that the parameters (application keys) established by the handshake protocol have changed. The protocol is a simple message which is compressed and encrypted using current keys, and sent during the handshake negotiation when the security parameters have been agreed upon.

2.4 TLS Alert Protocol

The TLS alert protocol allows communicating parties to signal potential problems to each other through alert message, which consists of two bytes. The first byte indicates the severity of a problem (`warning` or `fatal`), and second indicates the degree of severity or description of the alert (e.g. `handshake_failure`). If the alert message is `fatal`, the current TLS session is terminated immediately. Like other messages, alert messages are encrypted and compressed, as specified by the current security parameters.

2.5 TLS Record Protocol

The TLS record protocol is in charge of cryptographically protecting the application-layer data. It uses a MAC-Encode-Encrypt (MEE) construction and at the encryption step we focus on block cipher encryption in CBC mode. We refer to this setting of protocol as MEE-TLS-CBC. Other two encryption possibilities are stream ciphers and authenticated encryption with associated data (AEAD).

2.5.1 Encryption: In the record processing the application data is pushed through four layers of TLS record protocol to obtain the corresponding records. This process is illustrated in Figure 3.

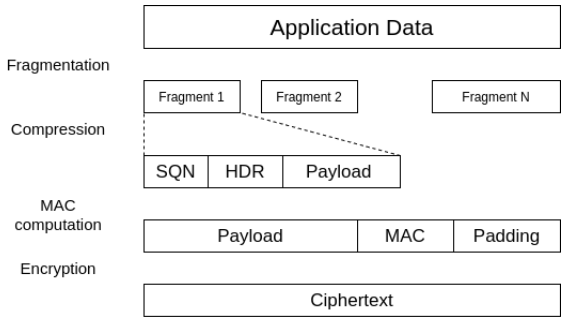

Figure 3.: (D)TLS encryption procedure.

First, the application-layer data is **fragmented** into blocks of 2^{14} bytes or less. Next, each fragment may optionally be **compressed** using the compression algorithm defined in the current session state. The combined use of compression and encryption is known to be dangerous, as it introduces some new vulnerabilities that may be exploited by specific compression-related attacks. For this reason the use of compression is not recommended and has been completely removed in TLS 1.3.

In the third step a **Message Authentication Code (MAC)** is computed. Let R denote an individual (optionally compressed) fragment to which we refer to as a record. A record is simply a byte sequence of length zero or more. The sending party also maintains an 8-byte sequence number `SQN` which is incremented for each record sent. Aditionally, it constructs a 5-byte field `HDR` which includes a 2-byte version field, a 1-byte type field, and a 2-byte length field. Let T denote a MAC tag calculated over the byte sequence $\texttt{SQN}||\texttt{HDR}||R$. Depending on the algorithm can the size of T be 16 bytes in case of HMAC-MD5, 20 bytes in case of HMAC-SHA-1, or 32 bytes in case of HMAC-SHA-256. We denote size of T as t.

In the last step the record is **encrypted**. First the record is encoded into plaintext $P = R||T||\texttt{pad}$, where `pad` is a seqence of padding bytes such that the block size b divides the length of P. Size of b depends on the selected block cipher–i.e., $b = 8$ for 3DES and $b = 16$ for AES. The padding is made up of $p+1$ concatenations of some byte value p, where $0 \leq p \leq 255$ (e.g., "0x00", "0x01||0x01"). Finally, the encoded record P is encrypted using CBC-mode of the selected block cipher as:

$$C_i = E_K(P_i \oplus C_{i-1})$$
$$C_0 = \texttt{IV}$$

were `IV` can be explicit (randomly generated) or chained depending on the version of TLS and DTLS. P_i corresponds to i-th block of P and K is the key that the block cipher uses. After each block of plaintext is encrypted the resulting ciphertext C consists of concatenated ciphertext blocks C_i. Before the ciphertext C is transmitted to the receiver the `HDR` is prepended to it. While in TLS the sequence number is not sent as part of the message in DTLS it is. Moreover DTLS always uses randomly generated `IV` while some versions of TLS use chained `IV`.

2.5.2 Decryption: To decrypt the received message $\texttt{HDR}||C$ we need to reverse the encryption process. The receiver first extracts the ciphertext C from the message and checks whether the block cipher's block size b divides the length of ciphertext and if C is large enough to contain at least a zero-byte record, a MAC tag of size t, and at least one byte of padding.

Then the ciphertext C is decrypted block by block

using decoding algorithm D to recover the plaintext blocks:

$$P_i = D_K(C_i) \oplus C_{i-1}$$
$$C_0 = \texttt{IV}$$

In next step the padding is removed. The padding format has to be the same as specified in encryption process, otherwise some attacks are possible which exploit this vulnerability. Usually last byte of plaintext P is taken which tells us the length of padding `padlen`. Then padding corresponds to last $\texttt{padlen} + 1$ bytes, which are removed if the size of P is sufficient enough– i.e., bytes for MAC tag and at least zero-length record needs to remain.

Finally, the MAC tag can be recomputed and compared with the MAC tag in the plaintext P. In case of TLS a MAC tag is checked using the header information and a sequence number that is kept by the receiver, whereas in DTLS the sequence number is optionally checked for replays. What if the padding format is incorect? To avoid known timing side-channel attacks the MAC should still be computed. But in this case it is unclear where the padding ends and where the MAC tag is located (the plaintext cannot be parsed). The solution TLS and DTLS provide is to assume that there is no padding and take last t bytes of the plaintext P to be a MAC tag. The remaining bytes are taken as record R and MAC verification is performed on $\texttt{SQN}||\texttt{HDR}||R$.

If all goes well, the decrypted plaintexts are optionally decompressed and assembled back together to the application-layer data that was sent. Else, if padding is incorrect and error occurs during decryption TLS and DTLS may handle it differently. TLS always treats it as fatal and terminates the session and deletes all the security parameters. DTLS may consider such errors as non-fatal and the decryption process would continue, which make the attacks this report describes easier to apply.

2.6 How is MAC actually computed?

In this section a detailed description of computation of a MAC tag is provided in order to understand how the time differences between decrypting a message with correct and incorrect padding format may be detected and how they might be used to reveal some information about plaintext length.

We already mentioned possible HMAC-H algorithms, where H is MD5, SHA-1, or SHA-256. Given a message M and key K, the MAC tag T is computed as follows. The hash function H used by HMAC is iteratively applied twice, as

$$T = H((K \oplus \texttt{opad})||H((K \oplus \texttt{ipad})||M)),$$

where `opad` (outer padding) and `ipad` (inner padding) are specific 64-byte values. The key K is zero-padded to make its size 64 bytes before XOR operations are executed. Every hash function H is iterative and is applied to every 64-byte chunk of a message M. The chunks are then compressed and chained into the next iteration step. In addition, every message M is encoded where an 8-byte sequence followed by padding is appended to it before it is hashed. The padding is at least 1 byte in length and extends the message to a $(56 \bmod 64)$-byte boundary.

The timing profile of HMAC algorithm depends on the hash function H used. A 55-byte messages or smaller can be encoded in single 64-byte block, meaning that HMAC will use total of 4 compression function evaluations–2 for inner and 2 for outer hash operation. A message M containing from 56 up to 119 bytes can be encoded into two 64-byte blocks, meaning that HMAC will use total of 5 compression function evaluations– 3 for inner and 2 for outer hash operation. In general, an extra compression function evaluation is needed for each additional 64 bytes of message data. Typically, a single compression function evaluation time is measured in nanoseconds.

Now lets remember that in TLS the MAC is computed on plaintext after the padding is removed. This implies that total running time of decryption process might reveal some information about the size of the plaintext, maybe even up to 64-byte accuraccy. Next chapter describes a simple distinguishing attack which exploits this vulnerability and serves as a warm up for the plaintext recovery attacks described in Chapter 4.

3 DISTINGUISHING ATTACK

The goal of distinguishing attack is to distinguish the true ciphertext from a random sequence. The attacker can choose two messages, say m_1 and m_2, where one is encrypted into ciphertext C and given to him. His task is to find which message was encrypted. It is trivial for an attacker to find such message if m_1 and m_2 are of different sizes, so we assume both are equally long.

In following sections we show one construction of such attack, analyze it, and discuss about applying it in the real life scenario.

3.1 Construction of the Attack

First, lets describe the setting the attack will be constructed in. We use the block cipher AES with block size $b = 16$ and explicit `IV`. The attack would be very similar for 3DES with $b = 8$. The MAC tag T is computed using one of the HMAC-H algorithms, where H is either MD5, SHA-1, or SHA-256. The attacker chooses messages m_1 and m_2 as shown in Figure 4. Message m_1 has 32 arbitrary bytes and 256-byte padding of byte value `0xFF`. Message m_2 has 287 arbitrary bytes and byte `0x00` in the end as a padding. Because $b = 16$ both

messages can be divided into exactly 18 blocks (288/16 = 18).

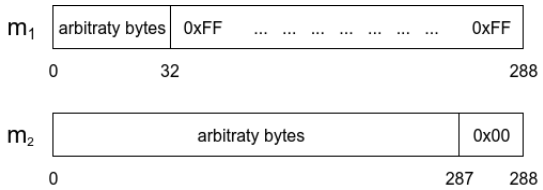

Figure 4.: The structure of messages used in distinguishing attack on MEE-TLS-CBC construction of TLS.

Attacker submits m_1 and m_2 for encryption in CBC mode and receives a ciphertext $\texttt{HDR}||C$. Here C is an encryption of encapsulation of one of the messages, say m_d ($d = \{1, 2\}$), a MAC tag T, and some padding $\texttt{pad}$. Because messages are chosen to fit in exactly 18 blocks, the additional bytes $T||\texttt{pad}$ are encrypted in separate blocks from m_d. Now attacker can form a new ciphertext $\texttt{HDR}||C_{\text{new}}$, where C_{new} contains the first 288 bytes of C–i.e., discards the blocks that contain $T||\texttt{pad}$ but keeps the same 16-byte $\texttt{IV}$.

The attacker now sends $\texttt{HDR}||C_{\text{new}}$ for decryption and observes the bytes at the end of P_{new} he received:

- If P_{new} ends with the valid 256-byte padding format $\texttt{0xFF...0xFF}$, the encrypted message was m_1. The padding is then removed and the remaining 32 bytes of plaintext correspond to a message and a MAC tag. Size of the message and MAC tag depends on the hash function H (see Table 1a). It is highly likely that MAC verification will fail and that the attacker will receive an error message.
- If P_{new} ends with the valid 1-byte padding format $\texttt{0x00}$, the encrypted message was m_2. The padding is removed and the remaining 287 bytes of plaintext correspond to a message and a MAC tag. Size of the message and MAC tag depends on the hash function H, as seen in Table 1b. Again, it is highly likely that MAC verification will fail and that the attacker will receive an error message.

3.2 Analysis of HMAC Algorithm

Now lets investigate how many evaluations of compression function of HMAC algorithm occur in both cases–when m_1 is encrypted and when m_2 is encrypted.

In case of m_1 the message processed by, lets say HMAC-MD5, has 13 bytes for header and at most 16 bytes for MAC tag. Then total of 4 compression function evaluations are required (2 for inner and 2 for outer compression).

In case of m_2 the message processed by one of the possible HMAC-H algorithms has 13 bytes for header and at least 255 bytes for message. Recall that 4 evaluations are needed for messages of at most 55 bytes in length, and then one more evaluation per every additional 64 bytes of a message is required. It is easy to see that at least 8 compression function evaluations are needed for m_2–at least 4 more than for m_1.

H	message [byte]	MAC tag [byte]
MD5	16	16
SHA-1	12	20
SHA-256	0	32

(a) A case when m_1 was encrypted.

H	message [byte]	MAC tag [byte]
MD5	271	16
SHA-1	267	20
SHA-256	255	32

(b) A case when m_2 was encrypted.

Table 1.: Possible message sizes given hash function H used by HMAC algorithm and corresponding sizes of MAC tags.

Hence, we can assume that time required to produce decryption error message in case of m_2 is a little larger than in case of m_1. And by little we mean couple of microseconds. In theory this timing difference can be exploited for a distinguishing attack on the MEE-TLS-CBC construction used in TLS.

3.3 Real-Life Application

In the analysis of the attack we only considered time taken by compression functions, whereas there are many processing operations to take into account. For example, the time taken to remove the padding is different for two messages being processed; padding removal for m_1 takes longer than for m_2. This reduces the time difference in MAC checking we described in analysis. These ignored time differences are smaller than in MAC checking and are not that important, thus the attack is still possible.

In real-life application of this attack network jitter might cause problems when measuring such small time differences. An error message might have a small delay of couple microseconds before the attacker receives it, which is of same proportion as time differences measured for MAC verification. On the other hand when attack is performed against restricted environment (e.g. 8-bit or 16-bit processor, or smartphone) the time differences might be quite large due to slow operations and network jitter would not affect the attack much. Another example where the jitter might be significally reduced is virtual environment, namely cloud. The attacker could run a separate process on the machine performing TLS decryption, and iteratively apply the attack across number of sessions, with same message being encrypted in each session, and extract the timing signal (using statistical analysis).

In DTLS this attack can be applied in similar fashion. Here, because DTLS might not treat decryption errors as

fatal, the attacker also needs to send a message that provokes the DTLS response (besides the ciphertext). Any timing difference arising from decryption of ciphertext then shows up as difference in the arrival time of the response messages.

The timing signal can also be amplified if multiple messages containing same ciphertext are sent at once. Then the attacker could observe cumulative timing difference, because all messages will be processed in the same way. For instance, if record R_1 takes 5 microseconds and recrod R_2 takes 6 microseconds to decrypt, the time diference is 1 microsecond. If both records are then each sent 5 times consecutively the time taken for decryption of batch of R_1 records will be 25 microseconds and 30 microseconds for batch of R_2 records. The time difference is now 5 microseconds, which is easier to detect.

This attack is resistent to some other safety mechanisms in (D)TLS as well. The authors of [1] successfully implemented distinguishing attack against OpenSSL implementation of TLS and their results are presented in 5.

4 Plaintext Recovery Attacks

The attacks can be seen as an advenced version of padding oracle attack [11]. Specifically, if TLS cipher suite configuration uses HMAC-SHA-1 algorithm for computing a MAC tag and certain message lengths are carefully chosen, then TLS records containing one byte of correct padding or invalid padding format will take slightly longer to process than TLS records containing at least two bytes of correct padding. Coincidentally, this is possible because of the relation between the length of TLS header, plaintext and MAC tag, and the cipher's block boundary and the hash compression function's block boundary. The timing difference is measured in hash function compression evaluations. In our case the timing difference corresponds to one evaluation or few hundred clock cycles on a modern processor. The processing time of a single record is measured as arrival time of error messages during decryption. By repeating the attack sufficiently often and using careful statistical processing, the noise arising from network jitter and other sources can be overcome and the different padding conditions can be differentiated from one another. Thus an attacker can distinguish messages containing at least two bytes of correct padding from all other patterns. At this point, a variant of the standard padding oracle attack can be carried out.

The weakness of leaking the validity of padding format from protocols in form of time difference is called a timing side channel.

4.1 Construction of the Attack

In construction of this attack we focus on breaking the TLS protocol, with details for DTLS to follow. Furthermore, for easier presentation, we assume that the block cipher is AES ($b = 16$); plaintext will be divided in fewer bigger blocks, the CBC mode uses explicit IVs, the MAC algorithm is HMAC-SHA-1 ($t = 20$) and zero-length pad in case of incorrect padding (for MAC verification). If the block cipher is 3DES ($b = 8$) the non-`IV` blocks in ciphertext and plaintext are doubled. Other setting variations are attacked in similar manner with only more possibilities to try for recovering bytes. Normally, we assume that the adversary is capable of eavesdropping on (D)TLS-secured channels and can inject any messages into the channel.

Now, say the attacker intercepted a ciphertext C^{att}, of what form should it be? Or rather, how many non-`IV` blocks should C^{att} have? Again, remember that MAC computation of 55 bytes takes one evaluation of hash compression function less than for byte sequence of length from 56 to 119. So **the attacker aims to modify C^{att} such that the corresponding plaintext P^{att} will have enough padding that MAC will be computed on 55 bytes or less**. Because CBC mode of encryption is used, the attacker has to modify second-to-last block of C^{att} in order to change last block of P^{att}. Finally, if we truncate 13 bytes of header (`SQN` and `HDR`) and add 20 bytes for MAC tag the ciphertext needs to have at least $55 - 13 + 20 = 62$ bytes. The closest larger multiple of 16 (we use AES) is 64, meaning we need to add 2 bytes and C^{att} will have 4 non-`IV` blocks.

Next, let C' and C^* be second-to-last and last block of C^{att}, respectively. The attacker wishes to recover P^*, which is the plaintext block that corresponds to C^*. The following holds:

$$P^* = D_K(C^*) \oplus C'.$$

If C^* is the first block of a ciphertext then C' may be the last block of the preceding ciphertext or the `IV`. Then we have

$$C^{\mathrm{att}} = \mathtt{HDR}||C_0(=\mathtt{IV})||C_1||C_2||C'||C^*$$

For any block B of ciphertext or plaintext we write $B = [B_0 B_1 ... B_{b-1}]$, where B_i corresponds to i-th byte of B. For example, we have $C^* = [C_0^* C_1^* ... C_{b-1}^*]$.

We said the attacker will wants to modify second-to-last block, C', of C^{att}. He can achieve this by simply XOR-ing C' with some 16-byte block X of his choice. We denote modified ciphertext C^{att} with block X as $C^{\mathrm{att}}(X)$ and corresponding 64-byte plaintext as $P^{\mathrm{att}}(X)$. We have

$$C^{\mathrm{att}}(X) = \mathtt{HDR}||C_0||C_1||C_2||C' \oplus X||C^* \text{ and}$$
$$P^{\mathrm{att}}(X) = P_1||P_2||P_3||P_4,$$

where

$$P_4 = D_K(C^*) \oplus (C' \oplus X) = P^* \oplus X \qquad (1)$$

is an important relation between unknown, target plaintext block P^* and plaintext of injected message. What happens during decryption? Receiving party decrypts $C^{\mathrm{att}}(X)$ block by block, removes padding and performs MAC verification, which is extremely likely to fail and produce an error message. There are three possible scenarios that can occur:

1) P_4 has valid padding format of length 1 byte (i.e. ends with `0x00`). After padding is removed the next 20 bytes are taken as a MAC tag T leaving $64 - 1 - 20 = 43$ bytes of plaintext as the record R. To verify the MAC a 13-byte header is added to R meaning MAC verification will be performed on 56-byte message.
2) P_4 has invalid padding format. In this case zero padding is assumed; last 20 bytes are taken as T and MAC verification is performed on $44 + 13 = 57$ bytes.
3) P_4 has valid padding format of length at least 2 bytes. In this case two or more bytes are removed as padding and the next 20 bytes are taken as T, meaning the record R will have 42 bytes or less. Adding the header the MAC verification will be performed on at most 55-byte message.

The first two scenarios will take 5 evaluations of the compression function for SHA-1 to verify the MAC, while third scenario requires only 4 evaluations. Hence, **we would like to distinguish first two scenarios from the third one by timing the appearance of the error message on the network**. Here we can see how a 13-byte header together with 20-byte MAC are crucial for making this timing side channel possible.

The most likely padding pattern in third scenario is of length 2. Then the attacker can set the mask X so that he triggers scenario 3 after sending $C^{\mathrm{att}}(X)$ for decryption. At this point the attacker knows that P_4 ends with `0x01||0x01` and can recover the last 2 bytes of P^* using the equation from (1). If padding is longer than 2 bytes the attack can be repeated by modifying the third-to-last byte of the mask X for recovering last 3 bytes. The same principle is applied for recovering longer paddings.

This is possible because the attacker can freely select and send $C^{\mathrm{att}}(X)$. The byte-recovery procedure is simple; for every possible combination of X_{14} and X_{15} in X submit $C^{\mathrm{att}}(X)$ which will surely produce scenario 3. Note that in the worst case scenario at most 2^{16} combinations are tried. Here the scenarios in which X falls are decided with simple statistical test such as basic percentile test or averaging over recorded times.

After recovering last 2 bytes of P^* an efficient attack for recovering the remaining bytes exists. It is called padding oracle attack [11]. In essence, one iteration of an attack–for example–to extract third-to-last byte goes as follows: set X_{14} and X_{15} to the recovered bytes so that P_4 ends with `0x02||0x02`. Then generate X for all possible X_{13} in the same fashion as before, except that now at most 2^8 iterations are required. This will produce a ciphertext that fits the third scenario, meaning third-to-last byte of P_4 was set to `0x02`. Now P^*_{13} can be recovered. For recovering the complete P^* at most $14 \cdot 2^8$ trials are needed.

4.1.1 Real-Life Application: Shortly, this variant of the attack most likely could not be carried out in uncontrolled environment. First major problem is that the timing difference between the scenarios is extremely small–measured in processor clock cycles–and can easily be hidden by network noise. This problem can be solved by executing the attack in parallel with several sessions.

Second major problem is that TLS sessions are destroyed after every attack. This problem can again be solved by executing the attack in multi session manner where the same plaintext is repeated in the same position over every session.

4.2 Lucky 13 and the BEAST

Due to large consumption of TLS sessions a new variant of an attack was proposed in combination with the BEAST attack [6]. Here the attack is restricted to web browser communicating with web server over TLS where TLS-protected HTTP cookies are targeted. To perform this attack the user must first download malware into his browser. This can be accomplished in various different ways. The malware can then automatically initiate all the TLS sessions needed for the attack where the browser will automatically append the targeted HTTP cookie plaintext in each target ciphertext block. Then a plaintext recovery attack with partially-known plaintext **(partial plaintext recovery attack)** can be carried out. In this attack the attacker knows one of the 2 last bytes of P^*, say P^*_{14}, and can then set the initial value of X such that $X_{14} = P^*_{14} \oplus$ `0x01`, so that when $C^{\mathrm{att}}(X)$ is decrypted, the second-to-last byte of P_4 already equals `0x01`. Then he iterates over 2^8 possibilities to find such X_{15} that P_4 has its last two bytes equal to `0x01||0x01` which triggers scenario 3. The rest of the bytes are recovered as described before. This attack will establish at most $15L \cdot 2^8$ sessions, where L is the number of trials used for each X.

When dealing with HTTP cookies its basic access authentication, the username and password are Base64 encoded, meaning that each byte of plaintext has only 64 possible values. This results in significant reduction of space of possibilities (2^6 per byte), leaving this variant of the attack with at most $15L \cdot 2^6$ established sessions to decrypt a block.

4.3 Applying the Attacks to DTLS

The attacks on DTLS are very similar to those on TLS with few bigger differences. As discussed in 3.3 the timing differences can be amplified and DTLS responses might need to be provoke. The latter difference implies that the entire attack against DTLS can be executed in single session without having to repeat same plaintext in the same position in the plaintext in every session.

Furthermore, there is no more waiting for Handshake Protocol to reestablish the session. These differences make execution of these attacks in uncontrolled environments in real world possible.

5 Experimental Results

In this chapter experimental results from [1] are shortly presented. Results were obtained in controlled environment where OpenSSL 1.0.1 was used on the client and the server which were connected to the same VLAN.

5.1 Distinguishing Attack (TLS)

The authors executed an attack described in Section 3 and obtained distribution of timing values of encryption of m_1 and m_2 shown in Figure 5. Looking at mean values of both distributions one could easily distinguish between them–i.e., between m_1 and m_2.

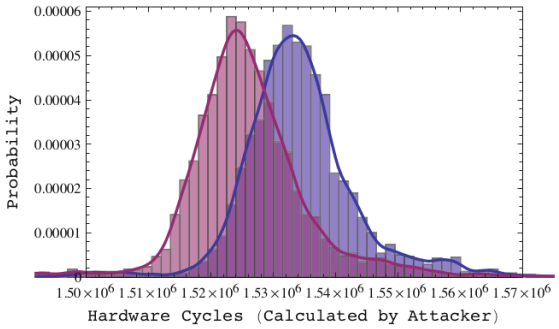

Figure 5.: Distributions of processing time measurements of m_1 (left, red) and m_2 (right, blue). The processing times for m_1 are clearly smaller than for m_2.

For the attack a simple thresholding test was used. After profiling and measuring L timing samples outliers were filtered out. Then if the median value is smaller than some threshold T return 0 and 1 otherwise. The test results are in Table 2. Apparently the attack is reliable even for small amount of samples; even for $L = 1$ the success probability is quite high whereas for $L = 2^7$ the attack will always work.

5.2 Plaintext Recovery Attacks (TLS)

This attack was not implemented due to too large time consumption when establishing and destroying TLS sessions. For example, the attack would need a total of $L \cdot 2^{16}$ trials which can make up to 2^{23} trials if $L = 2^7$ is taken (if we want the attack to be successful). This would take about 64 hours.

L	Success Probability
1	0.756
2	0.769
4	0.858
8	0.914
16	0.951
32	0.983
64	0.992
128	1

Table 2.: OpenSSL distinguishing attack success probabilities.

5.3 Plaintext Recovery Attacks (DTLS)

For attacks against DTLS the authors used amplification techniques where the attacker sends a batch (n) of constructed packages, a DTLS Heartbeat request immediately after and then waits for Heartbeat response (instead of error message). The attack is repeated L times for each mask value. For the attack in this setup $n = 10$ is a good choice but should be larger if the server and client (attacker) are further apart. Note that for $n = 1$ we closely simulate what would happen with TLS–only the network noise tends to be higher with DTLS.

When recovering P^*_{15} when P^*_{14} is known and $n = 10$ the attack is very effective as shown by percentile-based success probabilities in Figure 6. For $L = 8$ the attacks will recover unknown plaintext byte with only 2^{11} trials.

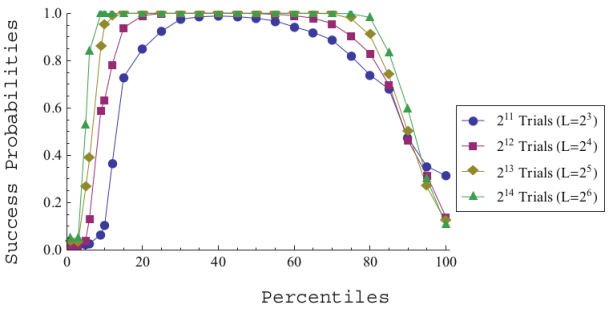

Figure 6.: Percentile-based success probabilities in case of OpenSSL DTLS partial plaintext recovery where P^*_{14} is known and $n = 10$.

The authors implemented a 2-byte recovery attack against OpenSSL DTLS which is essential for full text recovery attack (and the harder part). For $n = 10$ and $L = 8$ (2^{19} trials) the attack is again very effective, with success probability of recovering P^*_{14} and P^*_{15} is 0.93 (see Figure 7). Hence, it is highly probable that the full plaintext attack can be executed.

To conclude, for $n = 1$ the attack serves as experimental model for TLS. From results in Figure 8 we see

that 2-byte recovery is very reliable if 2^{23} ($L = 2^7$) trials are used. $L = 2^6$ also gives promising results with success probability of more than 0.8 but for $L = 2^5$ the success probabilty already falls to around 50%.

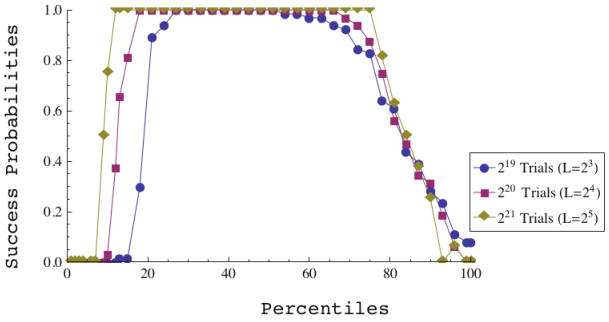

Figure 7.: This figure shows percentile-based success probabilities for OpenSSL DTLS 2-byte (P^*_{14} and P^*_{15}) recovery, where $n = 10$.

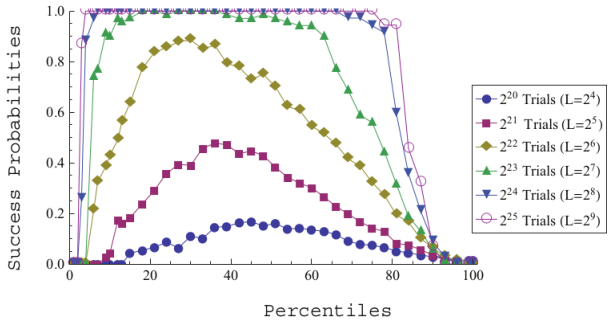

Figure 8.: This figure shows percentile-based success probabilities for OpenSSL DTLS 2-byte (P^*_{14} and P^*_{15}) recovery, where $n = 1$.

6 Conclusion

In this project we provided some main building blocks of the TLS and DTLS protocols, namely Record Protocol and described a timing side channel attack against it. It turns out that relationship between MAC tag size, block size and 13-byte header enables the attacker to construct ciphertexts in such way that, when decrypted, information about validity of padding is leaked. This information is in the form of arrival of error messages on the network in case of TLS or Heartbeat response in the case of DTLS. After enough trials it is possible to recover plaintext.

We have discussed about drawbacks of this attack in 3.3 and 4.1.1. The biggest drawbacks are extremely small timing differences which can easily be concealed by network jitter and enormous amount of sessions (in case of TLS). The latter causes the attack against TLS to be unrealistic as even in controlled environment the attack would take about 64 hours. On the other hand attacks against DTLS are possible due to only one session needed and amplification techinquest that make timing differences easier to detect.

Some possible countermeasures are adding random time delays to the decryption process which would make statistical analysis of timing differences useless. This was shown to not solve the problem [2]. One possible solution is that every decryption process would take the same amount of time no matter the configuration parameters and sizes.

In case of TLS RC4 stream cipher could be used instead of CBC-mode encryption but has some other vulnerabilities.

Switching from MEE-TLS-CBC to using a dedicated authenticated encryption algorithm like AES-GCM or AES-CCM which were standardised for use in TLS will make the attacks described here useless. This novelties however can bring implementation errors or other, yet unknown, attacks.

References

[1] Al Fardan, N. J. and K. G. Paterson: *Lucky thirteen: Breaking the tls and dtls record protocols*. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[2] Albrecht, M. R. and K. G. Paterson: *Lucky microseconds: A timing attack on amazon's s2n implementation of tls*. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 622–643. Springer, 2016.

[3] Allen, C. and T. Dierks: *The TLS Protocol Version 1.0*. RFC 2246, January 1999. https://rfc-editor.org/rfc/rfc2246.txt.

[4] Canvel, B., A. Hiltgen, S. Vaudenay, and M. Vuagnoux: *Password interception in a ssl/tls channel*. In Boneh, D. (editor): *Advances in Cryptology - CRYPTO 2003*, pages 583–599, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg, ISBN 978-3-540-45146-4.

[5] Dierks, T. and E. Rescorla: *The transport layer security (tls) protocol version 1.2*. RFC, 5246:1–104, 2008.

[6] Duong, T. and J. Rizzo: *Beast: Surprising crypto attack against https*. Blog, September, 42:45–47, 2011.

[7] F., Alan O., Philip K., and Paul C. K.: *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101, August 2011. https://rfc-editor.org/rfc/rfc6101.txt.

[8] Islam, M., M. Hossain, M. Hasan, M. Shahjalal, Y. M. Jang, *et al.*: *Design and implementation of high-performance ecc processor with unified point addition on twisted edwards curve*. Sensors, 20(18):5148, 2020.

[9] Modadugu, N. and E. Rescorla: *The design and implementation of datagram tls*. In *NDSS*, 2004.

[10] Paterson, K. G., T. Ristenpart, and T. Shrimpton: *Tag size does matter: Attacks and proofs for the tls record protocol*. In Lee, D. H. and X. Wang (editors): *Advances in Cryptology – ASIACRYPT 2011*, pages 372–389, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg, ISBN 978-3-642-25385-0.

[11] Vaudenay, S.: *Security flaws induced by cbc padding—applications to ssl, ipsec, wtls...* In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 534–545. Springer, 2002.