Plaintext-Recovery Attacks Against Datagram TLS

Nadhem J. AlFardan and Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK
{nadhem.alfardan.2009, kenny.paterson}@rhul.ac.uk

Abstract

The Datagram Transport Layer Security (DTLS) protocol provides confidentiality and integrity of data exchanged between a client and a server. We describe an efficient and full plaintext recovery attack against the OpenSSL implementation of DTLS, and a partial plaintext recovery attack against the GnuTLS implementation of DTLS. The attack against the OpenSSL implementation is a variant of Vaudenay's padding oracle attack and exploits small timing differences arising during the cryptographic processing of DTLS packets. It would have been prevented if the OpenSSL implementation had been in accordance with the DTLS RFC. In contrast, the GnuTLS implementation does *follow the DTLS RFC closely, but is still vulnerable to attack. The attacks require new insights to overcome the lack of error messages in DTLS and to amplify the timing differences. We discuss the reasons why these implementations are insecure, drawing lessons for secure protocol design and implementation in general.*

Keywords TLS, DTLS, CBC-mode encryption, padding oracle, attack, timing, OpenSSL, GnuTLS.

1 Introduction

DTLS, OpenSSL and GnuTLS: The Datagram Transport Layer Security (DTLS) protocol was first introduced at NDSS in 2004 [10]. Two years later, the Internet Engineering Task Force (IETF) assigned Request for Comments (RFC) 4347 [11] to DTLS. The aim of DTLS is to provide a datagram-compatible variant of TLSv1.1 [6] that eliminates the dependency on the Transport Control Protocol (TCP). Since its introduction, there has been a growing interest in the security services offered by DTLS. Leading implementations of DTLS can be found in OpenSSL[1] and GnuTLS[2]. Both of these provide source toolkits that implement TLS and DTLS as well as being general purpose cryptographic libraries that software developers can use. The first release of OpenSSL to implement DTLS was 0.9.8. Since its release, DTLS has become a mainstream protocol in OpenSSL. There are also a number of commercial products that have taken advantage of DTLS. For example, DTLS is used to secure Virtual Private Networks (VPNs)[3,4] and wireless traffic[5]. Platforms such as Microsoft Windows, Microsoft .NET and Linux can also make use of DTLS[6]. In addition, the number of RFC documents that are being published on DTLS is increasing. Recent examples include RFC 5415 [1], RFC 5953 [8] and RFC 6012 [13]. A new version of DTLS is currently under development in the IETF to bring DTLS into line with TLSv1.2.

Padding oracle attacks: According to [11], the DTLS protocol is based on TLSv1.1 and provides equivalent security guarantees. In particular, then, one would expect implementations of DTLS to be resilient to attacks on TLS known prior to the development of TLSv1.1, especially those attacks explicitly mentioned in RFC 4346 [6], the specification for TLSv1.1.

One such example is the padding oracle attack introduced by Vaudenay in [15] and applied to OpenSSL by Canvel *et al.* in [2]. This attack exploits the MAC-then-Pad-then-Encrypt construction used by TLS and makes use of subtle timing differences that may arise in the cryptographic processing carried out during decryption, in order to glean information about the correctness or otherwise of the plaintext format underlying a target ciphertext. Specifically, Canvel *et al.* used timing of encrypted TLS error messages in order to distinguish whether the padding occurring

*This author's research supported by an EPSRC Leadership Fellowship, EP/H005455/1.

[1] http://www.openssl.org

[2] http://www.gnu.org/software/gnutls

[3] http://www.cisco.com/en/US/products/ps10884/index.html

[4] http://campagnol.sourceforge.net

[5] http://www.cisco.com/en/US/docs/wireless/controller/7.0MR1/configuration/guide/cgi_lwap.html

[6] http://www.eldos.com/sbb/desc-ssl.php

at the end of the plaintext was correctly formatted according to the TLS standard or not. Using Vaudenay's ideas, this *padding oracle* information can be leveraged to build a full plaintext recovery attack. However, because TLS tears down the connection in the event of any error arising during cryptographic processing, and because all the messages in the attack do provoke such errors, the attack can only recover significant amounts of plaintext if the same plaintext bytes are repeated across many TLS connections at the same location in the data stream.

OpenSSL quickly addressed the attack of [2] by modifying the code to firstly make the error messages are identical and secondly to ensure that the error messages would always appear on the network at the same time, an approach we call *uniform error reporting*. These countermeasures eventually appeared in TLSv1.1 as guidance for implementors a few years later. Since the initial work in this vein, padding oracle attacks have been generalised in various ways and applied to other network protocols such as IPsec [4, 5] and application layer protocols such as ASP.NET [12, 7], further highlighting the dangers of the MAC-then-PAD-then-Encrypt cryptographic construction coupled with non-uniform error reporting.

Our attack on DTLS: Given this history, and the fact that the DTLS specification is based on that of TLSv1.1, implementations of DTLS should be immune to padding oracle attacks and their variants. Our paper shows that this is not the case for either the OpenSSL or the GnuTLS implementations of DTLS.

We first focus on OpenSSL, showing that there is a small timing difference in OpenSSL's processing of DTLS packets having valid and invalid padding fields: just like old version of OpenSSL's implementation of TLS, if the padding is invalid, then the MAC is not checked, while if the padding is valid, the MAC check is done. This results in a timing difference for processing of packets with valid and invalid padding that is on the order of a few tens of microseconds (μs) on a modern processor.

However, one major difference between TLS and DTLS is that DTLS provides no error messages when decryption encounters an error. The detection of these error messages is essential to the attacks of [2] on TLS. Thus it would appear that this timing difference *cannot* be used to build a padding oracle. This may explain why the OpenSSL code for DTLS has not been patched to remove the known timing difference.

By bringing new techniques into play, we show that the lack of DTLS error messages is not a serious impediment to the attack – we are able to exploit the DTLS extension for Heartbeat messages [14] to ensure that the timing difference shows up in the timing of Heartbeat response messages rather than error messages. In fact, any upper layer protocol which has messages that also provoke a response message with a predictable delay can be used in place of Heartbeat messages in our attack. We also introduce new techniques which *amplify* the identified timing difference. In TLS, this is easily done by using long messages, since TLS supports messages up to roughly 2^{14} bytes in size. But this is not possible in DTLS, since the maximum message size is limited by the PMTU. To overcome this, we build trains of DTLS packets which all either have valid or invalid padding and hence which all contribute to an accumulated timing difference in the same way. These trains need to be carefully injected into the network – fast enough so as to ensure each packet arrives before the processing of the previous one has completed, but not so fast that DTLS's buffer for incoming packets gets swamped. Thus the success of the attack depends on delicate, μs-level timing of network events.

Another major difference between TLS and DTLS is that, in TLS, any error arising during cryptographic processing is treated as fatal, meaning that the TLS connection is discarded in the event of any error. TLS can afford to do this because it is built on top of a reliable transport protocol, TCP. DTLS, on the other hand, cannot afford to do so, since its underlying transport protocol is UDP. This means that DTLS does *not* discard connections in the event of errors, but merely discards error-generating packets. So, in contrast to previous attacks on TLS, our attack on OpenSSL's DTLS implementation can efficiently recover as much plaintext as the adversary desires, without having to wait for the re-establishment of DTLS connections. Our attack becomes even more efficient in the situation where DTLS's anti-replay feature is disabled, which is an option within the DTLS specification.

Our attack on OpenSSL is easily prevented, by properly implementing the countermeasures in the TLSv1.1 specification on which DTLS is based. We have informed the OpenSSL development team about our attack and a fix to prevent the attack was incorporated in versions 1.0.0f and 0.9.8s of OpenSSL.

We then switch our focus to the GnuTLS implementation, and show that, even though it properly implements the countermeasures in TLSv1.1, it is still vulnerable to a partial plaintext recovery attack in its default configuration. We show that a small timing channel is introduced into the decryption process because a plaintext-dependent sanity check is carried out at an early stage during decryption, followed later by assigning a zero value to the plaintext message length in the case when this sanity check fails. This introduces a detectable timing difference that, when combined with our new techniques, allows 4 or 5 bits of plaintext to be recovered per ciphertext block. Our specific attack on GnuTLS can be prevented by assigning a proper value to the plaintext message length in the case when the plaintext-dependent sanity check fails. We have informed

the GnuTLS development team about our attack, and the code was patched in version 3.0.11 of GnuTLS.

Despite the availability of fixes, we argue that the attacks are still interesting and provide valuable lessons for protocol designers and implementors:

- To our knowledge, our attacks are the first of their kind against any implementations of DTLS. Our OpenSSL attack is also the first plaintext-recovering attack against a protocol implemented by OpenSSL since the work of Canvel *et al.* [2].
- Our attacks exploit the fact that DTLS has to be error-tolerant, but we had to find a novel means to circumvent the resulting lack of error messages.
- The DTLS specification is rather brief and refers to the TLSv1.1 specification for many details, particularly those relating to how packets are encrypted and decrypted. This then requires an implementor to cross-refer to other standards during implementation, which may lead to software that does not implement the known countermeasures.
- Our attack on the GnuTLS implementation of DTLS and TLS shows that, even if all the known countermeasures are carefully implemented, DTLS and TLS implementations may still be vulnerable to attack via subtle timing side channels.

We expand on these themes later in the paper.

Paper organisation: Section 2 provides further background on DTLS, TLS and padding oracle attacks, as preparation for the presentation of our basic attack against the OpenSSL implementation in Section 3. Then Section 4 discusses a number of implementation issues for this attack and discusses refinements of it. Section 5 presents our experimental results demonstrating efficient and reliable recovery of full DTLS plaintexts in the OpenSSL case. Section 6 briefly discusses how similar attacks can recover partial plaintexts in the GnuTLS case. Section 7 discusses the wider implications of our work for secure network protocol design.

2 Further Background

2.1 Encryption in DTLS

A DTLS client initiates a handshake protocol with a server to agree on a number of parameters such as the cipher suite and the keys to use for a symmetric encryption algorithm and a message authentication code (MAC). After the handshake is complete, DTLS deploys a MAC-then-PAD-then-Encrypt construction, with CBC being a commonly used mode of operation. We will assume CBC-mode encryption is in use for the remainder of the paper.

When sending a DTLS packet, the sender first calculates a MAC over the DTLS payload and other parameters including a sequence number [11]. The size of the MAC output depends on the hash function used (e.g. 160 bits in the case of HMAC-SHA-1). The MAC is appended to the DTLS message. The sender then appends padding so that the payload length is a multiple of b bytes, where b is the block-size of the selected block cipher (so $b = 8$ for 3DES and $b = 16$ for AES). As with TLS, the padding consists of $p + 1$ copies of some byte value p, where $0 \leq p \leq 255$. In particular, at least 1 byte of padding must always be added. So examples of valid padding fields are: "0x00", "$0\text{x}01, 0\text{x}01$" and "$0\text{x}02, 0\text{x}02, 0\text{x}02$". The padding may extend over multiple blocks, and receivers must support the removal of such extended padding. The concatenation of DTLS message, MAC and padding is then encrypted using CBC-mode of the selected block cipher, using an explicit IV. Thus, the ciphertext blocks are formed as:

$$C_j = E_k(P_j \oplus C_{j-1})$$

where P_i are the plaintext blocks, C_0 is the IV, and k is the key for the block cipher E. The resulting ciphertext, including the IV, is then appended to a header which includes a length field and an explicit sequence number. The decryption process reverses this sequence of steps. First the ciphertext is decrypted block by block to recover the plaintext blocks:

$$P_j = D_k(C_j) \oplus C_{j-1},$$

where D denotes the decryption algorithm of the block cipher. Then the padding is checked and removed, and finally, the MAC is checked.

2.2 DTLS versus TLS

Applications that operate over the Unreliable Datagram Protocol (UDP) can easily take advantage of the security services offered of DTLS. Such applications are generally unconcerned about the session management services that TCP provides. Simple Network Management Protocol [8] is a good example of such applications. Other examples include voice and video network streaming applications. By design, DTLS is very similar to TLSv1.1 [6]. In fact, RFC 4347 [11] presents only the changes to TLSv1.1 introduced by DTLS and refers to RFC 4346 [6] for the rest of the protocol specification. According to RFC 4347, this approach has been chosen to minimize the amount of effort needed to implement the protocol. Thus, to fully understand and be able to analyse and code DTLS, the reader of RFC 4347 is expected to be familiar with TLSv1.1.

A number of changes were introduced so that the services of TLSv1.1 could be delivered over an unreliable

transport protocol like UDP. The reader can refer to [11] and [10] for the complete list of changes. We list here some of the changes relevant to our work:

- In TLS, MAC errors must result in connection termination. In DTLS, the receiving implementation may simply discard the offending record and continue with the connection. According to [11, Section 4.1.2.1], DTLS implementations should silently discard data with bad MACs, and the OpenSSL implementation takes this "discard and continue" option, with no error messages being sent on the wire. Not sending error messages clearly complicates the task of the adversary, since it is the presence of these messages (and their timings) that allowed previous attacks on TLS; however not terminating the connection in the event of an error proves to be very useful in building a reliable padding oracle that can be accessed as many times as the adversary wishes.
- Unlike TLSv1.1, fragmentation of record messages is not permitted in DTLS. Instead, a DTLS record must fit within a single lower layer datagram. Therefore, we cannot use a large size message in our attacks, a feature exploited in [2] to amplify timing differences for TLS.
- DTLS optionally supports record replay detection, whereas this is required in TLS. The technique used is the same as in IPsec's AH protocol [9], by maintaining a bitmap window of received records. Records that are too old to fit in the window and records that have previously been received are silently discarded. According to [11], the replay detection feature is optional, since packet duplication is not always malicious, but can also occur due to routing errors. In this paper, we mostly focus on the case where the DTLS anti-replay feature is disabled, but explain how to extend our attack to the case where it is enabled in Section 4.5.

2.3 Heartbeat Extension for TLS and DTLS

The Heartbeat extension [14] provides a new protocol for TLS and DTLS allowing a keep-alive functionality. This is very useful in the case of DTLS, which runs on top of unreliable transport protocols that have no concept of session management. The only mechanism available at the DTLS layer to determine if a peer is still alive is performing a costly renegotiation. The Heartbeat extension uses Heartbeat request and response messages between two entities having an established DTLS connection. A Heartbeat request message can be sent by either of the entities and is protected using the same DTLS ciphersuite and keys used for protecting other payloads. According to [14], whenever a Heartbeat request message is received, it has to be answered with a corresponding Heartbeat response message. Both messages have specific lengths that can be detected by the adversary. Although we exploit Heartbeat request messages in our attack against OpenSSL, other type of messages could also be used. The only constraint is that they should always predictably generate responses that can be detected by the adversary. We demonstrate this in our attack against GnuTLS.

2.4 Padding Oracle Attacks

The concept of a padding oracle was first introduced by Vaudenay [15]. In Vaudenay's formulation, a padding oracle is a notional algorithm which, when presented with a CBC-mode ciphertext, returns `VALID` if the underlying plaintext has padding that is correctly formatted and `INVALID` otherwise. Here, correctness is with respect to some padding scheme. For example, for TLS/DTLS padding, correctness means that the decryption of the ciphertext is a byte string ending in one of the valid padding patterns "0x00", "$0\text{x}01, 0\text{x}01$", etc. Vaudenay showed that, for certain padding schemes, repeated access to a padding oracle can be used to decrypt arbitrary target ciphertext blocks (and indeed complete ciphertexts in a block-by-block manner). His techniques apply to the TLS/DTLS padding scheme and, for completeness, we show in Algorithm 1 how to decrypt a complete block from a target ciphertext C^*, given access to a padding oracle. In this algorithm, for ease of presentation, we number the bytes of the target ciphertext block C_t^* from 0 to $b-1$ starting with the *rightmost* byte; we also use C_{t-1}^* to denote the ciphertext block preceding the target block in the ciphertext C^*. For any block B, we write $B[i]$, $0 \leq i < b$ to denote its bytes. The attack requires on average 128 and at most 256 queries to the padding oracle to decrypt each byte of the target block. The attack as presented uses 2-block ciphertexts, but is easily adapted to use longer ciphertexts simply by ensuring that blocks R, C_t^* are always placed at the end of the ciphertext.

In practice, to mount a padding oracle attack, an adversary must find some way of actually realizing a padding oracle for a specific implementation. In the original presentation for TLS in [15], Vaudenay posited that such an oracle could be built by sending a message to a TLS server and then waiting for a reply in the form of an error message. In TLSv1.0, a `decryption_failed` message would indicate a padding error, while a `bad_record_mac` message would indicate that padding was correct, but that MAC verification had failed. There are (at least) two challenges to building a TLS padding oracle in this way:

1. The two TLS errors, `decryption_failed` and `bad_record_mac`, are classified as fatal, causing the immediate termination of the TLS connection after *every* query to the padding oracle. Informally, we say

Algorithm 1: Decrypting a block using a padding oracle $\mathcal{PO}$ for TLS/DTLS.

Data: C^*_{t-1}, C^*_t
Result: $P^*_t = D_k(C^*_t) \oplus C^*_{t-1}$
Let R be a random b-byte block.;
for $i = 0$ **to** $b - 1$ **do**
  for $\texttt{byte} = 0$ *to* 255 **do**
    $R[i] = \texttt{byte}$;
    $C = R||C^*_t$;
    if $\mathcal{PO}(C) =$ `VALID` **then**
      $P[i] = R[i] \oplus C^*_{t-1}[i] \oplus i$;
      Break;
  for $j = 0$ *to* i **do**
    $R[j] = R[j] \oplus (i) \oplus (i + 1)$;
Output P;

that the padding oracle behaves as a *bomb oracle*. The adversary must wait for a new TLS connection to be established before making another query, but each new connection will have fresh keying material. This makes the attack impractical unless connections are re-established quickly. Moreover, unless the same plaintext is repeated in a known ciphertext block across many connections, the adversary can only efficiently recover the last byte of each block in the bomb oracle case.

2. The two error messages are encrypted, making it more difficult for the adversary to distinguish them.

The work of Canvel *et al.* [2] addressed the second issue here, by developing a different realization for the padding oracle. Their realization relies on the fact that, for a TLS implementation, the processing of a message with valid padding may take longer than the processing of a message with invalid padding. The reason for this is that the padding is checked for validity before the MAC verification is performed, and so a TLS implementation that aborts processing immediately after detecting an error (of any kind) will exhibit a timing difference in message processing for packets with valid and invalid padding: in the former case, the MAC verification will take place, while in the latter it will not. The timing difference would then show up as a difference in the time at which the error messages appear on the network. As observed in [2], this is exactly how TLS was implemented in OpenSSL.

In the attack in [2], the timing difference was amplified by working with long messages, since these take longer to pass through MAC verification. Canvel *et al.* reported timing differences of as much as 2 milliseconds for these long messages[7]. Because of noise introduced by various sources, the padding oracle so obtained is not fully reliable, so the server had to be queried a number of times for every message and a statistical model used to analyse the observed timings. Moreover, the oracle is still a bomb oracle, so only one query per TLS connection can be made. Even so, Canvel *et al.* [2] were able to use this approach to extract TLS-encrypted passwords for an IMAP e-mail server running stunnel, an application using the OpenSSL implementation of TLS. The attack was perceived as serious enough that the OpenSSL code for TLS was updated from releases 0.9.6i and 0.9.7a, to ensure that the processing time for TLS messages is essentially the same, whether or not the padding is correct, and to send the same encrypted error message, `bad_record_mac`, in both cases. Eventually, the same countermeasures appeared in the specification for TLSv1.1 [6], with the requirement that they *must* be implemented.

3 Building A Padding Oracle for the OpenSSL Implementation of DTLS

3.1 Assumptions on the Adversary

The objective of the attack is to recover DTLS-protected plaintext. We assume that the adversary:

- Has access to the ciphertext. This can be achieved by the adversary gaining access to a network device like a switch or a router, or by ARP spoofing, or by eavesdropping in a wireless environment.
- Can send arbitrary DTLS messages to the original recipient. This can be achieved by injecting packets into the network while spoofing the IP and UDP headers.
- Is aware of the encryption algorithm's block size, b. The adversary can infer this by either monitoring the connection's handshake messages, or the size of the encrypted messages over time.
- Can detect and record a number of Heartbeat request packets.

The above assumptions apply when anti-replay is deactivated. We note that anti-replay is enabled by default for both the OpenSSL and GnuTLS implementations of DTLS, and we had to modify the server source code to disable it in our experiments. When anti-replay is activated, then we also need to assume that the adversary can stop messages of his choice from reaching their final destination. For example, the adversary may achieve this by exploiting his control

[7] We measured the MAC verification time for DTLS messages with payload sizes of up to 1456 bytes and found the time to be in the tens of μs instead.

over a router or a firewall in the data path. In presenting our attack below, we assume that anti-replay is disabled, i.e. we assume that the targeted system does *not* perform sequence number checking for incoming DTLS messages. We explain how to modify the attack to handle the case where anti-replay is enabled in Section 4.5.

3.2 A Padding Oracle for the OpenSSL Implementation of DTLS

In this section, we explain how to construct a padding oracle for the OpenSSL implementation of DTLS. This oracle can then be used in the standard way to decrypt arbitrary ciphertext blocks and thence arbitrary amounts of plaintext data, as described in Section 2.4. The key observation we use is that, in the current OpenSSL implementation of DTLS, if the padding underlying a ciphertext is valid, then the MAC on the message is checked, whereas if the padding is invalid, then the MAC is not checked and the ciphertext is rejected immediately. This contravenes the requirement for equal processing times in TLSv1.1 that is inherited by reference in the DTLS specification. As a consequence of this deviation, we would expect the processing time for a DTLS packet with invalid padding to be slightly less than that of a DTLS packet with valid padding. The actual time difference depends on a number of factors including the algorithms used, the clock-speed of the target system, the size of the DTLS packet, other processes running on the target system, and the network conditions. For example, we measured the MAC verification time on our testing machine running OpenSSL with HMAC-SHA-1 and found it to be in in the order of tens of μs – see Figure 1.

So far, this is identical to the timing side channel exploited in [2]. However, DTLS does not have any error messages, so we cannot use existing methods to observe the difference in processing times. This may explain why the implementors of DTLS in OpenSSL chose not to implement the required countermeasures. Instead, we introduce an alternative means to detect the timing difference, by exploiting Heartbeat messages. The basic idea is quite simple. Suppose we send to the target system a packet train consisting of a DTLS packet P_C carrying the ciphertext C (whose padding validity we wish to test) immediately followed by a Heartbeat request message. Then this train will result in a detectable Heartbeat response message being sent back on the network, and, assuming orderly processing on the target system, the total amount of time needed to process P_C and to produce the Heartbeat response message will reflect whether or not MAC verification was carried out when processing C. From an adversary's perspective, only send and receive times of packets can be captured, so the adversary will measure the time difference between sending the initial packet train and receiving the Heartbeat response packet, which we refer to as the *round trip time (RTT)*. If this time difference is larger than some threshold T, the adversary will assume the padding was valid (and so the MAC verification was carried out), while if it is lower than this threshold, the adversary will assume the padding was invalid. The threshold can be set by doing some initial system profiling to measure the typical timing difference between packets carrying ciphertexts having valid and invalid padding. Notice also that DTLS Heartbeat packets are not essential to building the oracle: any upper layer protocol having suitably predictable and detectable response messages can be used.

In reality, the timing of packets is influenced by many factors beyond just DTLS's cryptographic processing. Moreover, as we noted above, the timing difference will be rather small for normal-sized packets. So the DTLS padding oracle as presented would be much too error-prone. To enhance the accuracy of the oracle, the adversary can:

- Choose a specific, favourable DTLS packet payload length, l.
- Send n copies of packet P_C in a train followed by a Heartbeat request instead of just one copy of P_C. Here, the idea is that each copy of P_C will be processed in the same way, so the larger the accumulated time difference will become and the easier it will become to distinguish between valid and invalid padding. This exploits the fact that DTLS does not tear-down DTLS connections in the event of errors (recall that when the padding oracle is used in a plaintext recovery attack, all the ciphertexts sent in the attack will be invalid in some way – they will either have invalid padding or invalid MACs). It also assumes that all the packets in the train can be made to arrive at the target system in such a way that no adverse delays are introduced during the processing of these packets.
- Send m packet trains (each containing n copies of P_C), and use an applicable statistical model to analyse the observed RTTs.

Algorithm 2 describes our basic DTLS padding oracle for a ciphertext C. In the algorithm, RTT_q denotes the response time in the q-th trial, T denotes the threshold for deciding on whether C has valid or invalid padding, and simple averaging is used to process the gathered RTTs. Other statistical measures could be used in place of averaging here, an idea that we discuss in more detail in the next section. There we also explore the many practical issues that arise in building this padding oracle, addressing issues such as packet timing, system profiling, parameter selection to tune the attack, and dealing with anti-replay.

Algorithm 2: Padding Oracle for OpenSSL implementation of DTLS

Data: C
Result: `VALID` or `INVALID`
for $q = 1$ **to** m **do**
  RTT_q = **Timer(**C**)**;
RTT=Mean$(RTT_1, RTT_2, ..., RTT_m)$;
if $RTT \geq T$ **then**
  return `VALID`;
else
  return `INVALID`;

Timer(C**)**
Set T_s = current time;
Send n copies of P_C, a DTLS packet containing C, to the targeted system;
Send a Heartbeat request packet to the targeted system;
Set T_e = time when Heartbeat response packet is seen;
return $(T_e - T_s)$

4 Practical Considerations

In this section, we discuss a number of practical issues that arise in implementing our attack. All of our remarks are specific to the OpenSSL implementation of DTLS.

4.1 Timing and OpenSSL Cryptographic Operations

Our attack relies on detecting the time difference introduced by MAC verification that is performed for packets having valid padding but not for packets having invalid padding. Failure to detect this time difference would result in the padding oracle providing an incorrect answer. Figure 1 shows, for a variety of DTLS payload sizes, the time taken by OpenSSL in our set-up to perform decryption with 3DES or AES-256 alongside the time taken for MAC verification using HMAC-SHA-1. The hardware specifications of our set-up are listed in Section 5. We note the following features evident from this figure:

- In general, decryption is slower than MAC verification, especially in the case of 3DES.
- The MAC processing time for a single packet is on the order of a few tens of μs, which is well below that reported in [2] and below the level of jitter expected in a typical network.
- 3DES is much slower than AES-256: for a packet size of 1456 bytes, the factor is about 4. For reasons that will be explained below, using a slower decryption algorithm increases the effectiveness of the attack. Hence the attack parameters (l, m, n) may need to be tuned depending on which block cipher is in use.
- With AES-256, the processing time rapidly drops from about 50 μs to about 20 μs when the DTLS payload size reaches 512 bytes. We do not know the reason for this behaviour, but the adversary also needs to be aware of it when selecting attack parameters.

Although we have targeted HMAC-SHA-1 in our attack, the fundamentals of the attack still apply when other MAC algorithms are in use. At the time of writing, OpenSSL only supports HMAC-MD5 and HMAC-SHA-1. More detail about how packets are processed and the source of the timing difference is provided in Appendix A.

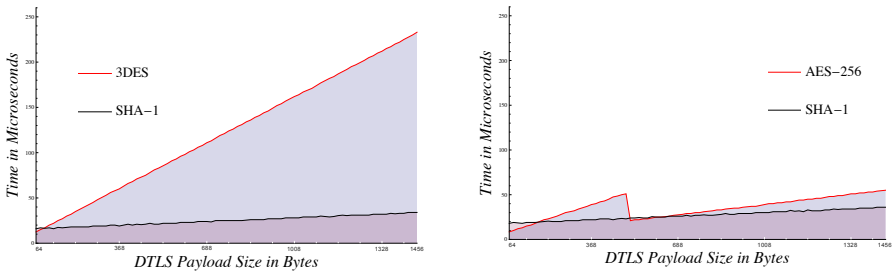
4.2 System Profiling

System profiling refers to the process by which the adversary collects information about the targeted system prior to carrying out an attack. This provides the adversary with the expected values for the RTTs (for valid and invalid padding) under some conditions such as system load, the DTLS payload length, l, and the number of packets in the train, n. This profiling in turn allows the threshold value T for the attack to be set.

Given a captured ciphertext, it is easy to construct ciphertexts having any desired length l and having either invalid or valid padding, simply by manipulating the last 2 blocks of the captured ciphertext and prepending random blocks (or truncating it if a shorter ciphertext is needed). Given such pairs of ciphertexts and a Heartbeat request message, the adversary can then construct packet trains containing the required number of packets n. These trains can then be repeatedly sent to the target system and the RTTs measured, to obtain two empirical PDFs, one for trains with validly padded packets and the other for trains with invalid padding. From these PDFs, the threshold T can be set by, for example, calculating the mean of each distribution and setting T to be the mid-point between the means. In practice, we tend to obtain small numbers of extreme outliers in such profiling experiments, and removing these before calculating the means by using a simple cut-off generally improves the performance of the attack. More sophisticated statistical methods can of course be employed, but we have found profiling followed by thresholding to be already adequate for our attacks to be successful.

4.3 An Attack Without System Profiling

System profiling is not even strictly necessary – for a given byte position i in the target block, an adversary can simply measure the RTTs for a packet train (consisting of n DTLS packets with the target ciphertext block being located



(a) 3DES and SHA-1

(b) AES-256 and SHA-1

Figure 1. Timing of cryptographic operations for DTLS payloads of sizes between 64 and 1456 bytes

at the end of each packet, followed by a Heartbeat request packet), for each of the 256 possible byte values in position i in the block preceding the target ciphertext block. Then the adversary can select as the correct byte value (i.e. the one giving valid padding) the one that maximises the RTT across the 256 measured RTT values. Accuracy can be further improved by repeating the trial for each byte m times, removing outliers, and using the maximum of the average RTTs. In fact, we have observed in our experiments that repeating the trial for each byte value m times, removing outliers, and then selecting the byte value that maximizes the *minimum* of the m measured RTTs for each byte value gives substantially higher success probabilities for the attack. We will illustrate this in Section 5 where we discuss our experimental results in more detail. This, then, is the preferred version of our attack. Note that, strictly speaking, this version of the attack does not build a padding oracle, but rather considers all possible 256 byte values simultaneously.

Even more sophisticated statistical techniques, such as sequential estimation (as in [2]) or likelihood estimation, can be used in place of averaging or selecting the minimum when processing the results of the m trials per byte. However, these more advanced approaches were not needed in order to successfully launch our attack. They could be useful in further reducing the amount of data sent or Heartbeat request messages consumed in an attack.

Finally, we note two further advantages of using an attack without profiling. Firstly, the process of profiling itself will require Heartbeat request messages to be gathered. Secondly, the attack environment may change over time during the attack itself, as varying network or server loads are experienced, for example. The attack without profiling described here automatically adjusts for such changes, at least if they do not occur within the time taken to recover a single byte of plaintext.

4.4 Measuring Success Under Budgetary Constraints

The attack is such that a byte is successfully decrypted only if all the preceding bytes in the same block are successfully decrypted. Hence, under a reasonable independence assumption, if the probability of successfully decrypting a byte is p, then the probability of successfully decrypting a block of size b will be $p_b = p^b$. For AES, $b = 16$, so for successful decryption of a whole block with a reasonable probability, we need p to be rather close to 1. For example, with $p = 0.99$ and $b = 16$ we have $p_b = 0.85$. The adversary can tune the attack parameters (l, m, n) so as to increase the success probability p of the attack and can try to find the optimal combination that results in the highest success probability. However, in practice, an adversary will have a limit on, for example, the maximum number of bytes that he wishes to send in order to recover a byte. As discussed below, when anti-replay is enabled, Heartbeat request packets (or their equivalents) will become a precious resource. Since each train consumes one such packet in this situation, it may be desirable to increase l, the packet size and n, the number of packets per train, so as to maximize the amplification effect, whilst minimizing m, the number of trains sent per byte. However, as our later experimental results will show, simply increasing l and n does not always help, especially in the case of AES-256.

4.5 Attacks with Anti-Replay Enabled

Attacking DTLS becomes slightly more complex when anti-replay is enabled. Since the OpenSSL implementation of DTLS first checks the sequence number against the anti-replay window before doing any cryptographic processing, the adversary has to take care that all packets sent in trains do not have sequence numbers that are marked as having

previously arrived. Fortunately, the anti-replay window is only updated if the MAC on a packet is successfully verified, and all the packets used in the attack will fail the MAC verification (with the exception of the Heartbeat packets), so the window is not updated as a consequence of these attack packets.

With anti-replay enabled, each Heartbeat request packet can be used only once, since its sequence number will be marked in the window as having been seen once the packet arrives. Moreover, the adversary has to ensure that the sequence number for each Heartbeat request packet used does fall within (or to the right of) the current anti-replay window, otherwise the Heartbeat request will be discarded and no response generated.

Thus Heartbeat request packets become a precious resource in the situation where anti-replay is enabled: the attack can only proceed as quickly as they become available. Hence decryption in this setting may be rather slow and "opportunistic" – every time a packet is seen on the wire by the adversary, a new packet train can be launched and a byte value tested.

Given these issues, it is apparent that the adversary should try to use as few Heartbeat request packets as possible, which means minimizing m for a given target success probability p. A further enhancement arises by building packet trains that test multiple byte values *simultaneously*. For example, the adversary could build two sets of m trains, each train containing $128n$ packets, with half of the possible byte values being tested in each train n times each. This would represent the first step in a binary search for the correct byte value, requiring only 8 steps and therefore $16m$ Heartbeat request packets to extract a byte. The number of Heartbeat requests consumed could be halved again with initial system profiling. In contrast, our basic attack would consume $256m$ Heartbeat request packets for the same result. We have not tested this version of the attack, but our experience indicates that it would work well whenever using long packet trains does not degrade performance.

Finally, we recall that packets from *any* suitable application layer protocol could be used in place of Heartbeat request packets, so long as the corresponding application always sends a detectable response packet with a predictable response time. So the success of our attack does not depend completely on the availability of Heartbeat request packets in the case where anti-replay is enabled.

5 Implementation and Results for OpenSSL

5.1 Implementation

In our laboratory set-up, we have a client, the adversary and the targeted system all connected to a 100Mbps Ethernet switch on the same VLAN. The targeted system was a machine running a single core processor operating at a speed of 1.87 GHz and having 2 GByte of RAM.

We ran version 1.0.0a of OpenSSL on the client and the server. We used the built-in OpenSSL utilities for the client[8] and the server[9], `s_client` and `s_server` respectively. `s_client` implements a generic client which connects to a remote host using DTLS, while `s_server` implements a generic server which listens for connections on a given UDP port using DTLS. We implemented the Heartbeat Extension feature by installing the appropriate OpenSSL patch[10]. We deactivated anti-replay by directly modifying the OpenSSL code.

5.2 Results

The results shown in this section reflect our specific set-up. Of course, the values would change as the set-up changes – for example, the timings are heavily dependent on the clock-speed of the processor used on the target system. However, the fundamentals of the attack would remain the same.

5.2.1 Experimentally observed PDFs:

The figures we discuss hereafter show PDFs observed in our experiments for different attack parameters and encryption algorithms. In all the figures, the x-axis represents RTTs while the y-axis represents the probability of observing these RTTs. In all figures, outliers have been removed. Each figure shows two PDFs, PDF_1 (in red) and PDF_2 (in blue), that correspond to having valid and invalid padding in the packets in the trains, respectively. We recall that l denotes the DTLS payload size, m denotes the number of trials per byte, and n denotes the number of DTLS packets per trial. Figures 2 and 3 show PDFs for n equal to 10 and varying the value of l, for 3DES and AES-256 respectively. We note the following:

- It is generally easier to distinguish between the two PDFs in the case of 3DES.
- Generally, there is an increasing overlap between the two PDFs as the value of l, the DTLS payload size, increases. This is more evident in the case of AES-256.
- In the case of AES-256, increasing l makes the PDFs much harder to distinguish. The reason for this is that the adversary spends more time preparing and sending packets as the packet size increases, while the targeted system may already have finished AES decryption and

[8] `http://www.openssl.org/docs/apps/s_client.html`
[9] `http://www.openssl.org/docs/apps/s_server.html`
[10] `http://sctp.fh-muenster.de/dtls-patches.html`

MAC verification and be waiting for the next packet. Thus long packets tend to arrive "late" at the targeted system.

Figures 4 and 5 show the PDFs for $l = 1024$ and varying the value of n, for 3DES and AES respectively. We note the following:

- In the case of 3DES, increasing the value of n helps in making the two PDFs more distinguishable. This is the case with AES-256 when small DTLS payloads are used.
- With AES, increasing the value of n when using large DTLS payloads makes the PDFs harder to distinguish. Figures 6 and 5 show this effect when AES-256 is used for $l = 256$ and $l = 1024$ respectively.
- By appropriately choosing the attack parameters, it is possible to obtain PDFs that are very easy to distinguish. For example, the last graph in Figure 6 shows the PDFs for AES-256 when $l = 256$ and $n = 160$, where the peaks are separated by more than 500μs while the distributions are entirely contained within 50 μs of the peaks.

5.2.2 Success Probability:

Table 1 shows the success probability, p, of decrypting a byte under different attack parameters (l, m, n) when AES-256 is used. We recall that the success probability for a block is then given by p^b where b is the block length in bytes.

These tables were obtained using the preferred version of our attack described in Section 4.3, where no system profiling is used, outliers are removed, and, for each byte, we use the minimum RTT value from the m values available, and then select the correct byte as being the one that gives the maximum amongst these values. Each entry in the tables is calculated using 100 runs of the attack.

We can clearly see that the probability of success increases as the number of trials, m, increases. Success probabilities p equal to 0.99 or above are easily achieved for moderate values of l, m and n, making our preferred attack both efficient and highly reliable for these parameter choices.

Table 2 shows analogous success probabilities for 3DES. Note however that in these tables, we report figures for substantially larger values of l than we did for AES-256. This is indicative of the fact that our attacks are still quite successful for 3DES even with long payloads, giving an additional amplification opportunity. As further confirmation of the practicality of our attacks,Table 3 provides success probabilities for AES-256 for $l = 192$ and various values of m and n, with the probabilities being based on 1000 runs of

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.00	0.15	0.41	0.32	0.00	0.01
2	0.02	0.24	0.32	0.40	0.00	0.01
5	0.05	0.08	0.49	0.02	0.00	0.01
10	0.04	0.12	0.36	0.00	0.01	0.01
20	0.01	0.13	0.34	0.05	0.02	0.01
50	0.10	0.33	0.38	0.03	0.00	0.01

$m = 1$

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.99	0.99	1.00	0.99	1.00	0.99
2	0.99	1.00	0.99	1.00	1.00	0.98
5	0.99	1.00	1.00	1.00	1.00	0.98
10	0.98	1.00	0.99	1.00	1.00	0.99
20	0.99	0.99	1.00	1.00	1.00	0.99
50	0.99	0.99	1.00	1.00	0.98	0.95

$m = 5$

n \ l	**128**	**160**	**192**	**224**	**256**	**288**
1	0.99	0.99	1.00	0.99	1.00	0.99
2	0.99	1.00	0.99	1.00	1.00	0.99
5	0.99	1.00	1.00	1.00	1.00	0.98
10	0.98	1.00	0.99	1.00	1.00	0.99
20	0.99	0.99	1.00	1.00	1.00	0.99
50	0.99	0.99	1.00	1.00	0.99	0.95

$m = 10$

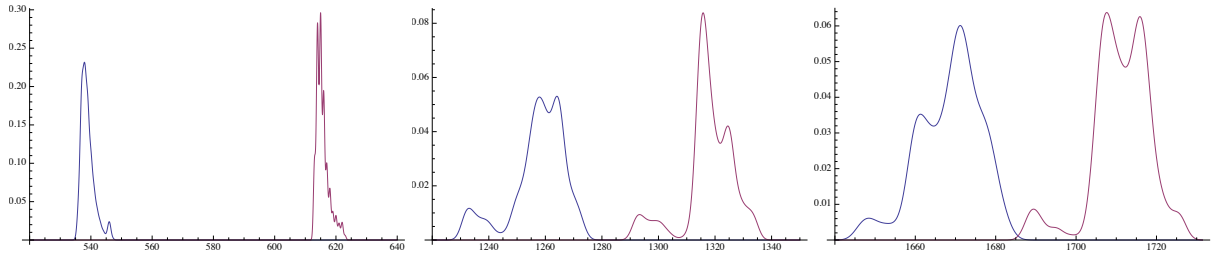
Table 1. Success probabilities per byte for AES, for various attack parameters.

the attack. For example, already for $m = 10$ and $n = 2$, the success probability is 0.996, meaning that an entire block of plaintext can be recovered correctly with probability 0.94, at a cost of (roughly) 7000 bytes of network traffic per byte.

6 Attacking the GnuTLS Implementation of DTLS

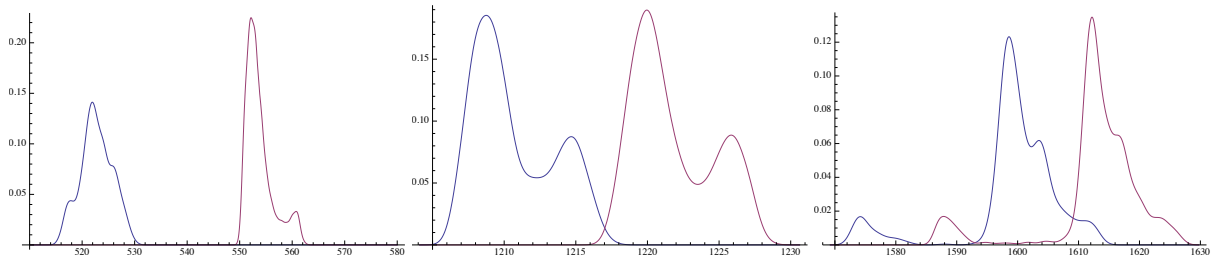
We have examined the GnuTLS implementation of DTLS, with the intention of finding similar attacks. The code for decryption[11] is such that a MAC computation is carried out whether or not the padding is validly formatted, and only then is the packet dropped. Thus a first inspection of the code would indicate that there is no timing difference in decryption processing that can be exploited in an attack.

[11] See `http://git.savannah.gnu.org/gitweb/?p=gnutls.git;a=blob;f=lib/gnutls_cipher.c`



(a) $l = 256$ (b) $l = 1024$ (c) $l = 1456$

Figure 2. 3DES – PDFs for $n = 10$ and varying l.



(a) $l = 256$ (b) $l = 1024$ (c) $l = 1456$

Figure 3. AES-256 – PDFs for $n = 10$ and varying l.

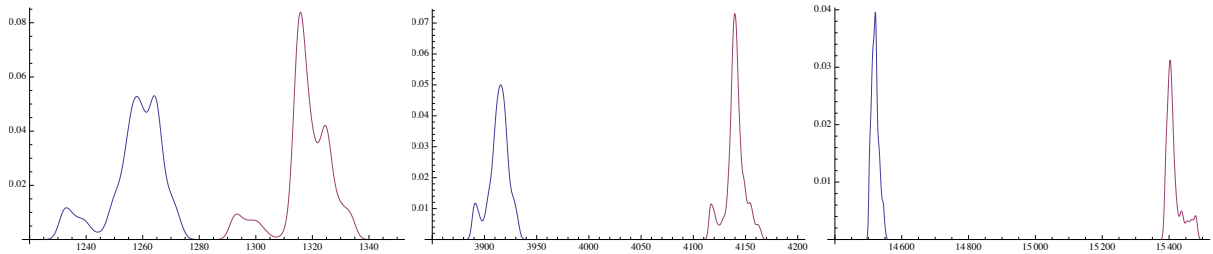
However, the code does include the lines shown in Figure 7. These lines are executed after CBC-mode decryption is complete. We explain their function next.

Lines 550-562 carry out a sanity check on the plaintext, making sure that the plaintext is long enough to contain as much padding as is indicated in the last byte of plaintext, which is assigned to variable `pad` (note that decryption is done in place, so the plaintext is held in the array `ciphertext.data}`. If this test fails, then a flag `pad_failed` is set. Line 564 calculates `length`, which should be the length of the message remaining after the padding and MAC field have been removed. Notice that this value could be negative as a result of decryption of an attacker-chosen ciphertext. Lines 576 and 577 eventually set it to `0` if this is the case. Lines 568-574 carry out the padding check, but only if the preceding sanity check did not fail. If the padding check fails, then again the flag `pad_failed` is set. Lines 582-593 perform the MAC computation (though the computed MAC value is not compared to the received MAC value until later). The code then goes on to return a negative value if any of the sanity check, padding format check or MAC verification have failed, and this eventually results in GnuTLS printing an error message to the screen. Unless the debugging level is changed, no other error messages are produced. Otherwise, if no check fails, the code returns a value indicating the length of the message.

Notice that, in the above code, if the sanity check fails, then `length` is set to `0` and the MAC check is carried out on a message consisting of just a few header fields. On the other hand, when the sanity check passes, the MAC check is carried out on a message consisting of header fields and as many message bytes are left after removal of padding and the MAC field. Since performing MAC verification on a string takes an amount of time roughly proportional to the length of that string, we see that decryption processing should be faster when the sanity test fails than it is when the sanity test passes. But the result of this sanity test depends on the value of the last plaintext byte, and hence the decryption processing time may leak information about that byte.

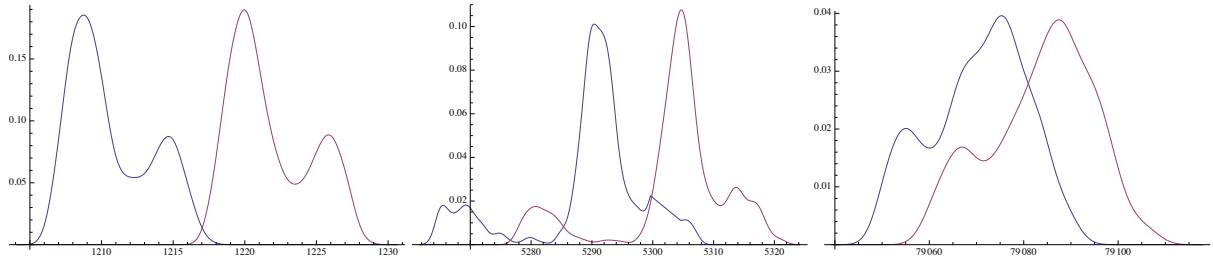
These observations lead to a partial plaintext recovery attack against GnuTLS that we explain next. For ease of presentation, we assume that the MAC size is 32 bytes (as would be produced by HMAC-SHA-256), but a similar attack would apply for 20-byte MACs. Now the padding length field `pad` is obtained from the last byte of the decrypted ciphertext (see line 550 in Figure 7). Consider an adversary who builds a DTLS packet whose encrypted payload (excluding the IV) is 160 bytes in length and that ends with two blocks R, C_t^*, where C_t^* is the target ciphertext block. Then, recalling our numbering convention for the bytes of a block and the CBC-mode decryption procedure, the sanity check in the GnuTLS code fails precisely when:

$$R[0] \oplus D_k(C_t^*)[0] > 127.$$



(a) $n = 10$ (b) $n = 40$ (c) $n = 160$

Figure 4. 3DES – PDFs for $l = 1024$ and varying n.



(a) $n = 10$ (b) $n = 40$ (c) $n = 160$

Figure 5. AES-256 – PDFs for $l = 1024$ and varying n.

Thus, if the targeted system responds quickly to the adversary's packet, he can infer that the most significant bit (MSB) of $R[0] \oplus D_k(C_t^*)[0]$ is most likely set to 1. From this, the MSB of $P_t^*[0]$, the rightmost byte of the plaintext corresponding to C_t^*, is easily deduced. The attacker can then target the second-MSB of $P_t^*[0]$, by setting $R[0]$ so that the MSB of $R[0] \oplus D_k(C_t^*)[0]$ equals 0 and then using a DTLS packet of length 96 bytes (again excluding the IV). This provides a test of the form:

$$R[0] \oplus D_k(C_t^*)[0] > 63$$

with the side information that $R[0] \oplus D_k(C_t^*)[0] \leq 127$, from which the adversary learns that the second-MSB of $R[0] \oplus D_k(C_t^*)[0]$ is set to 1 if the targeted system responds quickly. An alternative approach to this is setting $R[0]$ so that the MSB of $R[0] \oplus D_k(C_t^*)[0]$ equals 1 instead of 0 and then using a DTLS packet of length 224 bytes (again excluding the IV). This provides a test of the form:

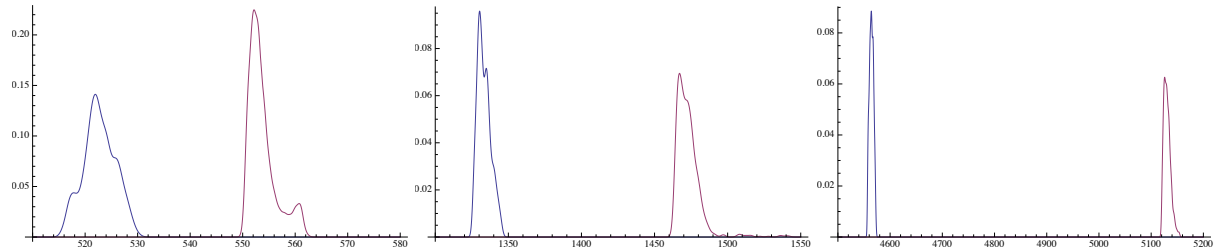
$$R[0] \oplus D_k(C_t^*)[0] > 191$$

The adversary can learn that the second-MSB of $R[0] \oplus D_k(C_t^*)[0]$ is set to 1 when the targeted system responds quickly. This alternative approach gives the adversary the opportunity to use packets with sizes that result in better success probabilities, and hence is preferable. For both approaches, iterating, the attacker can extract the 4 MSBs of $P_t^*[0]$ when the block cipher is AES, and the 5 MSBs of $P_t^*[0]$ when it is 3DES. The least significant bits (LSBs) cannot be extracted using our attack because the packet size must be a multiple of the block size b.

This provides a theoretical description of our attack. Of course, the adversary can use the same techniques as worked for OpenSSL to amplify his attack: using packet trains, multiple trials, and removal of outliers. A practical issue arises because GnuTLS does not implement the Heartbeat extension, but here we can use any application layer protocol with predictable timing differences.

We have conducted experiments to test whether the timing difference is sufficient to allow the attack for DTLS, with experimental results being presented in Figure 8 for HMAC-SHA-256 and AES-256, a ciphertext length of 176 bytes and 5 packets in a single train. Here, we see a separation between the two distributions, the red distribution for packets where the inequality "`pad > ciphertext.size - tag_size`" is satisfied and the sanity test fails, and the blue distribution for when the inequality is not satisfied and the sanity test passes.

With the second approach described above, where longer packets are used, we were able to achieve success probabilities of 0.738,0.744, 0.737 and 0.756 for individually extracting the first, second, third and fourth MSBs of the last plaintext byte, respectively, meaning that the four MSBs can be recovered correctly with probability 0.306, using (roughly) 43000 bytes of network traffic. These probabilities were achieved with $n = 5$, $m = 10$ and measured over 1000 at-



(a) $n = 10$ (b) $n = 40$ (c) $n = 160$

Figure 6. AES-256 – PDFs for $l = 256$ and varying n.

n	p
1	0.017
2	0.210
5	0.205
10	0.012
20	0.035
50	0.147

$m = 1$

n	p
1	0.961
2	0.983
5	0.983
10	0.985
20	0.989
50	0.965

$m = 5$

n	p
1	0.983
2	0.996
5	0.995
10	0.994
20	0.995
50	0.973

$m = 10$

Table 3. Success probabilities per byte for AES-256, for $l = 192$, based on 1000 trials.

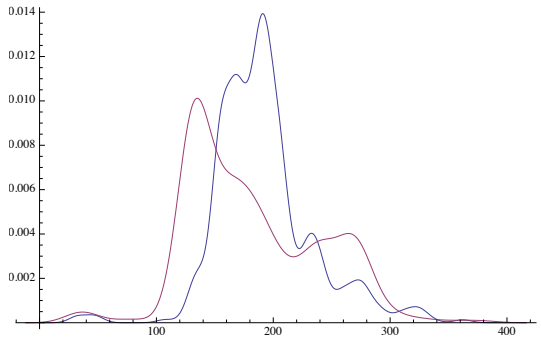


Figure 8. PDFs for AES-256 with HMAC-SHA256, $l = 176$, $n = 5$, based on 1000 trials, with outliers removed (GnuTLS).

tack runs. We used percentile filters, similar to the approach used in [3], to achieve these probability values. As expected, increasing the value of m significantly increases the success probability. For example, we were able to achieve success probabilities of 0.797 and 0.990 for recovering the four MSBs when $m = 50$ and $m = 100$, respectively.

To implement these tests, we used the same hardware set-up as the one we used for OpenSSL. We ran version 3.0.0 of GnuTLS on the client and the server. We used the built-in GnuTLS utilities for the client and the server, `gnutls_cli` and `gnutls_serv` respectively. We again disabled anti-replay by directly modifying the source code.

We have shared our findings with the GnuTLS development team. A fix to prevent our specific attack has been incorporated in version 3.0.11 of GnuTLS.

7 Discussion

We have demonstrated plaintext recovering attacks against the OpenSSL and GnuTLS implementations of DTLS. These are easily prevented by modifying the code so that the receiver's cryptographic processing time is independent of how decryption fails. However, we contend that the attacks are still interesting for a number of reasons.

Firstly, the fix to prevent our OpenSSL attack is already mandated in the specification for TLSv1.1, and is implemented in OpenSSL's implementation of TLS, but not in its implementation of DTLS. Without more insight into the software development processes followed by the OpenSSL project, we can only speculate that the experience about how to securely implement TLS's MAC-then-PAD-then-Encrypt construction was not carried over to the separate DTLS implementation. This, then, may also indicate of a lack of truly expert code review in the OpenSSL project. This is concerning given the prominence and wide application of the OpenSSL code, but also understandable given its volunteer-led effort. By contrast, GnuTLS's implementation has common code for the TLS and DTLS packet decryption procedure, meaning that countermeasures implemented for TLS are immediately carried over to DTLS. However, as we saw, even this was not sufficient to fully

```
550  pad = ciphertext.data[ciphertext.size - 1] + 1;   /* pad */
551
552   if ((int) pad > (int) ciphertext.size - tag_size)
553     {
.................
561      pad_failed = GNUTLS_E_DECRYPTION_FAILED;
562     }
563
564      length = ciphertext.size - tag_size - pad;
 .................
568    if (ver != GNUTLS_SSL3 && pad_failed == 0)
569     for (i = 2; i < pad; i++)
570      {
571        if (ciphertext.data[ciphertext.size - i] !=
572            ciphertext.data[ciphertext.size - 1])
573          pad_failed = GNUTLS_E_DECRYPTION_FAILED;
574      }
575
576  if (length < 0)
577    length = 0;
.................
582  preamble_size =
583    make_preamble (UINT64DATA(*sequence), type,
584                   length, ver, preamble);
585  _gnutls_auth_cipher_add_auth (&params->read.cipher_state, preamble,
       preamble_size);
586  _gnutls_auth_cipher_add_auth (&params->read.cipher_state, ciphertext.data,
       length);
.................
593  ret = _gnutls_auth_cipher_tag(&params->read.cipher_state, tag, tag_size);
```

Figure 7. Extract from GnuTLS decryption code, release 3.0.0

protect the GnuTLS implementation against the type of attack developed in this paper.

A second reason that the obvious and mandated countermeasures were not implemented in OpenSSL may stem from DTLS's lack of error messages, which makes the previous attacks apparently impossible against DTLS. We proved otherwise, exploiting DTLS Heartbeat request and response messages to obtain the required timing information. This kind of approach may be more widely applicable than DTLS.

A third possible explanation is that the DTLS specification relies heavily on cross-references to the TLSv1.1 specification, and indeed only gives specification details at points where TLS and DTLS differ. So an implementor needs to be familiar with both specifications in order to implement DTLS properly. We suggest that "specification by `diff`" is not a good approach to specifying secure protocols, since it requires an implementor to jump back and forth between specifications and may allow important details to fall into the gap between.

Secondly, a comparison between our attacks on DTLS and previous attacks on TLS is instructive. Our attacks are in some sense more challenging because of the lack of explicit error messages, but also easier to carry out because of DTLS's tolerance of errors, meaning that DTLS connections are not torn-down whenever an error is encountered as they are in TLS. Ultimately, this error-tolerance comes from DTLS's use of an unreliable transport protocol. For similar reasons, the anti-replay feature in DTLS is made optional in the specification. In this context, our work shows how non-security features of lower layer protocols can have a major influence on security at higher layers. This phenomenon is seemingly not that well-explored in the literature, presenting an interesting challenge for future work.

References

[1] P. Calhoun, M. Montemurro, and D. Stanley. Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification. RFC 5415, Internet Engineering Task Force, March 2009.

[2] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password Interception in a

n \ l	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.00	0.12	0.39	0.06	0.01	0.13
2	0.03	0.12	0.26	0.03	0.03	0.18
5	0.03	0.20	0.23	0.30	0.07	0.02
10	0.04	0.17	0.09	0.38	0.08	0.04
20	0.14	0.10	0.08	0.22	0.09	0.07
50	0.04	0.08	0.17	0.41	0.15	0.05

$m = 1$

n \ l	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.99	1.00	1.00	1.00	1.00	0.93
2	0.99	1.00	0.99	0.99	0.93	0.92
5	0.99	1.00	1.00	0.90	0.93	0.83
10	0.99	1.00	0.92	0.89	0.81	0.57
20	0.97	1.00	0.91	0.92	0.77	0.54
50	0.98	1.00	0.90	0.89	0.68	0.59

$m = 5$

n \ l	**128**	**256**	**512**	**1024**	**1280**	**1456**
1	0.99	1.00	1.00	0.99	1.00	0.93
2	0.99	1.00	1.00	0.99	0.93	0.92
5	0.99	1.00	1.00	0.91	0.94	0.83
10	0.98	1.00	0.93	0.89	0.81	0.57
20	0.98	1.00	0.92	0.92	0.77	0.54
50	0.99	1.00	0.91	0.89	0.68	0.59

$m = 10$

Table 2. Success probabilities per byte for 3DES, for various attack parameters.

SSL/TLS Channel. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2003.

[3] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[4] Jean Paul Degabriele and Kenneth G. Paterson. Attacking the ipsec standards in encryption-only configurations. In *IEEE Symposium on Security and Privacy*, pages 335–349, 2007.

[5] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *ACM Conference on Computer and Communications Security*, pages 493–504, 2010.

[6] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, April 2006.

[7] Thai Duong and Juliano Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2011.

[8] W. Hardaker. Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP). RFC 5953, Internet Engineering Task Force, August 2010.

[9] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, Internet Engineering Task Force, November 1998.

[10] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. In *NDSS*. The Internet Society, 2004.

[11] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, Internet Engineering Task Force, April 2006.

[12] Juliano Rizzo and Thai Duong. Practical Padding Oracle Attacks. In *4th USENIX Workshop on Offensive Technologies (WOOT'10)*, August 2010.

[13] J. Salowey, T. Petch, R. Gerhards, and H. Feng. Datagram Transport Layer Security (DTLS) Transport Mapping for Syslog. RFC 6012, Internet Engineering Task Force, October 2010.

[14] R. Seggelmann and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. Draft RFC, Internet Engineering Task Force. `http://tools.ietf.org/html/draft-ietf-tls-dtls-heartbeat-02`.

[15] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

A Timing and Packet Processing

In this appendix we look in detail at how a receiver processes a packet, with a view to building a simple model of how RTTs are affected by the attack parameters. To this end, Figure 9 shows a simplified time-line of how packet i, having valid padding, is processed by the receiver.

In the time-line we have:

- $t_{i,0}$: The time at which packet i arrives in the OpenSSL buffer. The buffer holds DTLS packets waiting to be processed.
- $t_{i,1}$: The time at which the decryption and padding check are completed for packet i.
- $t_{i,2}$: The time at which the MAC check is completed for packet i.
- $t_{i,3}$: The time at which OpenSSL is ready to process the next DTLS packet, packet $i+1$.
- OS_t: Any additional time spent by the operating system in relation to the processing of the packet. We assume this to be a constant, independent of i.

In the case of a packet with invalid padding, the MAC verification is not performed and hence we have $t_{i,2} = t_{i,1}$. Figure 10 is the analogue of Figure 9 for the case of invalid padding, and illustrates that, for a fixed DTLS packet length, the time taken to process a packet with invalid padding is less than that taken to process a packet with valid padding.

In Section 3, we defined RTT to be the time taken between sending the first packet in a train to receiving a Heartbeat response packet. Next, we analyse the different contributions to RTT. As an example, the time-line in Figure 11 shows a train made of two identical data packets (so $n = 2$), both having valid padding, followed by a Heartbeat request packet, which then provokes a Heartbeat response packet. In Figure 11 we have:

- T_s: The time at which the adversary sends the first DTLS packet, packet 1.
- T_f: The time at which the Heartbeat response packet is sent by the receiver
- T_e: The time at which the Heartbeat response packet is received by the adversary.
- $t_{1,0} - T_s$: The time it takes for packet 1 to reach the receiver.
- $T_e - T_f$: The time it takes for the Heartbeat response packet to reach the adversary after being sent by the targeted system.
- $T_e - T_s$: The RTT for the packet train.

Figure 11 shows the second data packet, packet 2, arriving **after** the completion of processing of packet 1 , i.e. so that $t_{2,0} > t_{1,3}$. The same applies to the Heartbeat request packet arriving after the completion of processing of packet 2. In this situation, the receiver enters a wait state until the next packet arrives and the arrival time of a packet and its processing start time are the same. In general, this situation results in some or all of the timing difference arising because of the MAC verification being "absorbed" into the wait state of the receiver, and hence is sub-optimal in terms of detecting the time difference.

In the opposite situation, where packet 2 arrives before processing of packet 1 is complete, packets are buffered. Then packet 2 is immediately available for processing at the receiver as soon as processing of packet 1 is complete, and none of the MAC verification time is absorbed. The buffer is managed by OpenSSL and its maximum size is 100 DTLS packets. Figures 12 and 13 illustrate this situation for packet trains having valid and invalid padding, respectively, with the white boxes representing the amount of time spent by packets in the buffer. It is evident from these figures how the time arising from MAC verification (in the case of valid padding) accumulates packet-by-packet to create an amplified time difference in the RTT for the train.

The upshot of this analysis is that, from the adversary's perspective, it is desirable to select the attack parameters so that the receiver's buffer always contains some (but not too many) packets. In this way, the receiver is never waiting for a packet to arrive and the MAC processing time accumulates across the whole packet train.

We have experimentally verified the essential basic correctness of this model for packet processing in the following way. Let RTT_1 denote the RTT for a train that uses packets having valid padding, and let RTT_2 denote the RTT for a train that uses packets having invalid padding. Let δ denote the time difference between the two RTTs, so that:

$$\delta = RTT_1 - RTT_2$$

Then, if we artificially inject delays in between packets from the train as they leave the adversary's machine, we would expect to see the value of δ steadily decrease and eventually reach zero as the size of the artificial delay increases. Figure 14 shows the results of such an experiment which confirms this behaviour. Somewhat surprisingly, Figure 14 also shows that adding small artificial delays can actually increase the time difference δ, making this difference in RTTs *easier* for the adversary to detect.

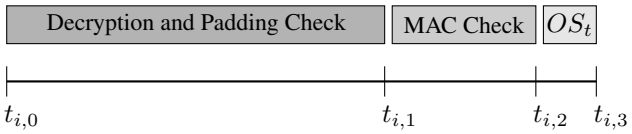


Figure 9. Packet processing time-line – valid padding

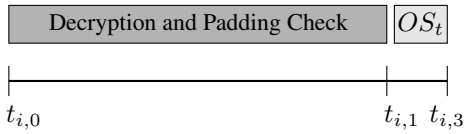


Figure 10. Packet processing time-line – invalid padding

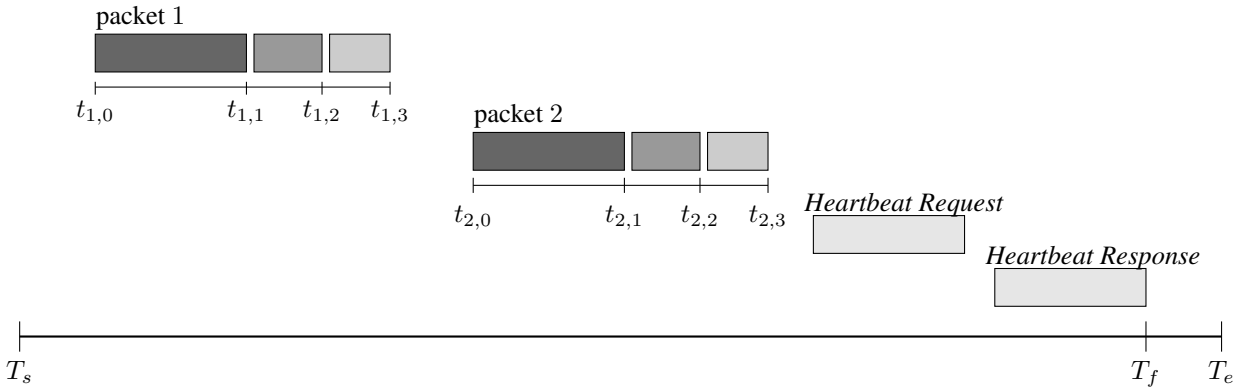


Figure 11. Time-line for a train with $n = 2$ (not to scale).

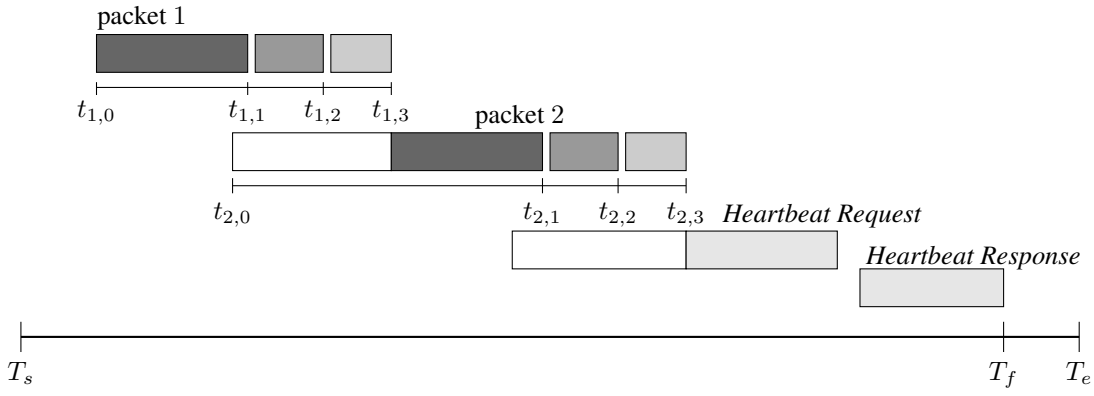


Figure 12. Time-line for packet train with valid padding and packet buffering.

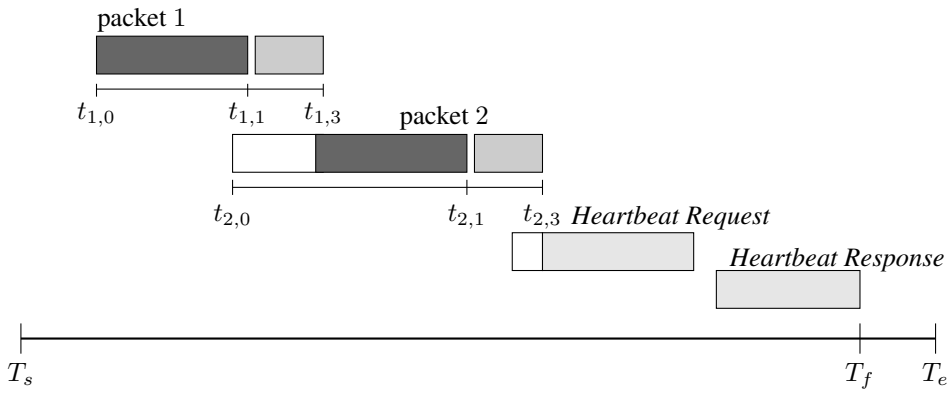


Figure 13. Time-line for packet train with invalid padding and packet buffering.

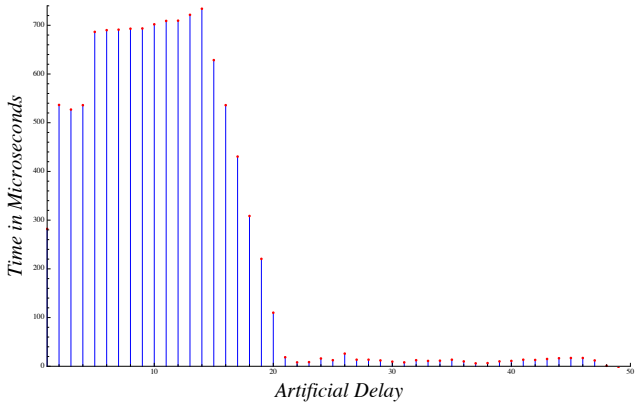


Figure 14. Value of δ, the difference in RTTs for valid and invalid padding, against artificial delay.