A Mute Oracle

George Arnold Sullivan
University of Illinois at Chicago

Akshay Raj
University of Illinois at Chicago

Abstract

The Cipher Block Chaining padding oracle attack was first described by Vaudenay in 2002. It relies on an attacker being able to distinguish between correctly and incorrectly padded messages. Because of the order of the authentication and encryption functions TLS has long been vulnerable to variations on this attack. To address this and other weaknesses in TLS Kaloper-Meršinjak, Mehnert, et al. [7] introduced not-quite-so-broken TLS[1]. nqsb-TLS was developed to be a reference for and functional implementation of TLS. Our goal in this project was to install a program that utilizes their TLS implementation, to measure timing differences within this implementation, and, hopefully, to exploit them to recover plaintext messages. We did not succeed in creating such an attack, however, in the following we discuss the difficulties that we encountered while trying to carry out this attack both those we overcame and those that we have not. We then offer some thoughts on possible solutions to our current set of problems and possible extensions if we were successful.

1 Introduction

Symmetric block ciphers, such as AES or DES, are designed to take a fixed block size, say 128 bits, and output a ciphertext, typically of the same size as the plaintext block. Despite this functionality, people insist on communicating with messages that are longer than 128 bits, and often with ones that are not even multiples of 128 bits. To address these shortcomings (we will leave it to the reader to decide whether the failings are on the part of the cipher or of the user) various methods have been adapted to allow a fixed length block cipher to accommodate variable length messages.

In general there are two problems to be addressed, first, messages may be longer than a single block, and second, their length may not be an even multiple of the block length. To address the first problem a number of *modes of operation* have been developed including an alphabet soup of alternatives such as ECB, CBC, CTR, etc. that describe how a fixed length block cipher can be used to encrypt multiple blocks, often by feeding some extra information in to be encrypted with each block. To address the second problem various methods of padding out the plaintext have been developed so that the message to be encrypted *is* an even number of blocks long. The fundamental requirement of these methods is that the message needs to be padded in some way so that the process is invertible, that is, the padding can be reliably removed and the original message recovered. However, as we will see, this necessary logic in the padding can often provide a helpful foothold for an attacker.

While these two problems may seem trivial in relation to the complexity involved in designing a block cipher, weaknesses in either the encryption mode or the padding can lead to the complete breakdown of security. For example, using ECB (electronic code-book) mode, that is simply encoding each block with the same key, perhaps the most obvious solution to encrypting multiple blocks, transparently leaks information about equality between two blocks. In this paper, we will be concerned with cipher-block chaining (CBC) mode and the padding approach used in TLS, which when used in conjunction, and incorrectly, can lead to plaintext recovery if the attacker can gain access to an oracle that will tell her whether a received message has bad padding or not. Originally, such an oracle was easily available because TLS implementations would inform users whether the padding was bad or whether the MAC was bad, this is no longer the case and newer attacks, require timing or other side channels.

Our paper is structured as follows. In section 2 we describe the generic padding-oracle attack and discuss previous work. In section 3 we discuss the nqsb-TLS implementation, which was our target. In section 4 we discuss

[1] `https://nqsb.io/`

the steps we took in trying to execute the attack. In section 5 we discuss possible future directions of this work, once the problems of section 4 are overcome. Finally, in section 6 we conclude.

2 Background

We begin by discussing CBC mode encryption. In his mode of operation, the previous ciphertext block, or in the case of the first block, the IV, is XORd with the current plaintext block and then encrypted to obtain the matching ciphertext block. Figure 1 on page 3 illustrates how encryption proceeds. In this way, each ciphertext block is dependent on all the plaintext blocks that have preceded it. Changing a single bit in the IV or the plaintext will have the salubrious, in terms of security, effect of modifying all subsequent ciphertext blocks.

Going the other way, during decryption (shown in Figure 2), each ciphertext block is passed through the decryption function, and the output is XORed with the previous ciphertext block, or the IV in the case of the first block.. Since, the decryption depends only on the ciphertext of the previous block, it can be parallelized if all the ciphertext is available. Therefore, while a change in the plaintext affects all subsequent blocks, an error in the ciphertext only affects the subsequent block, making it resilient to errors. It is this dependence of the plaintext on the previous block of ciphertext, in particular the fact that it is directly XORed with it, along with the padding scheme used in TLS that allows the padding-oracle attack to proceed.

Having given a brief overview of how CBC works, it is helpful, for the following, to keep in mind the structure of a TLS data packet after the headers for the lower layers have been removed. Such a packet has four components: a TLS header, data, a message authentication code (MAC) and padding. Tracking the construction of a packet we begin with some data that we want to transmit. To do so we first need to calculate the values for the padding and the header because we need to know the length of the message for the header, and to calculate the MAC we need the value of the header. Although there seem to be circular dependencies here we can calculate all these values because the MAC is of a known fixed length, so its length can be included in the calculation before we know its value. Knowing the length of the other values, the padding is then calculated so that the message, the MAC, and the padding are an even multiple of the block size. It should be noted that in TLS this padding need not be the shortest such padding, any longer padding equivalent modulo the block size and less than 255 is acceptable. Whatever (allowed) value chosen, each byte of padding has the value of the length of the padding, dictating the 255 limit. Finally, once the whole packet has been constructed, everything except for the header is encrypted using a block cipher in the chosen mode of operation and the packet is ready to send.

The basic attack against CBC with the above described padding was first published by Vaudenay [10]. The attack relies on the attacker having an oracle that will tell her whether or not a message is correctly padded. In this attack, the attacker intercepts some message that has been encoded using CBC-encryption and then sends modified packets to the receiver to decrypt the plaintext. Considering the decryption diagram above we can see that if we modify a byte of the intercepted ciphertext it will have a predictable effect on the last byte of plaintext. If the attacker modifies this last byte of the penultimate block it will likely change the message to have bad padding. The key to the attack is that if she can tell when the padding is good, this most likely means that the last byte decrypted to `0x01`. In that case if the modified byte is a and the modified plaintext is `0x01` we can see from the diagram that the original plaintext byte, call it p_{16}, has the value $p_{16} = a \oplus \texttt{0x01}$. This method can be easily extended to recover the whole block and then by dropping blocks the whole message can be recovered. Once this is understood the difficulty of the attack in practic is finding a way to use, typically the server, as an oracle.

TLS is vulnerable to this kind of attack because, as described above, it first applies a MAC to the plaintext and header, and then encrypts. Unless done very carefully this order of MAC-then-encrypt violates current best practice and invokes what Moxie Marlinspike calls the cryptographic doom principle [9]. Namely the idea is that if you do any operations before checking the MAC you are likely going to leak information. In current TLS there is an extension [5] to ameliorate the problem by performing the operations in the preferred order and in TLS 1.3 the problem will be addressed by requiring authenticated modes of encryption.

As mentioned the difficulty of the attack is finding an appropriate oracle. The original attack relied on the fact that at that time the attacker was able to get error messages saying whether the MAC was bad or whether the padding was bad. Subsequent attacks have depended not upon explicit error messages, but rather on timing differences between how bad MACs and bad padding are handled. A modification of this approach was used in the Lucky 13 [1] attack, which relied not on timing differences between padding and MAC errors, but between different MAC calculation times.

In addition to side channels, other information about the messages being transferred may be enough to effect an attack. For example, a structured plaintext such as an XML document can provide sufficient leverage for the attacker as demonstrated by Jager and Somorovsky [6].

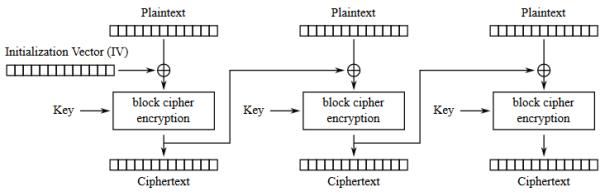

Figure 1: CBC mode encryption

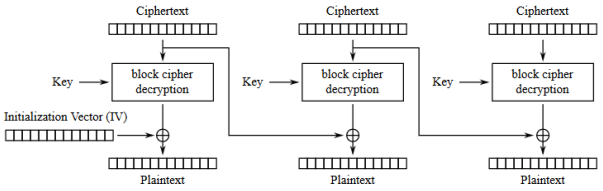

Figure 2: CBC mode decryption

3 nqsb-TLS

Having discussed the outline of the attack we now turn to the target. not-quite-so-broken (nqsb) TLS [7] is designed to provide a reference implementation of TLS. Rather than writing an implementation from scratch, it has been designed to interoperate with openssl to utilize its sophisticated architecture. nqsb-TLS is implemented using Ocaml, a functional programming language which provides type safety, memory safety, and allows concise and declarative description of complex protocol logic. The authors chose it both to take advantage of a clear description of the protocol using the code and to overcome a number of memory related attacks against TLS.

The implementation works on a client-server architecture and in addition to their library implementation they have made a number of applications using their implementation. In this paper we will largely focus on tlstunnel, for reasons we will discuss below. This application takes a request from a client and forwards it to the server, providing a secure access point for an insecure server. Based on the security choices of the library tlstunnel only speaks TLS 1.0 and above.

The presentation of nqsb-TLS notes that it is vulnerable to a timing attack along the lines of Lucky Thirteen [1]. This is because even though the implementation continues to calculate the MAC on a padding failure, there is still a small leakage in time since calculating MAC on plaintext is linear, but, is not linear when the same operation is performed over ciphertext. However, the authors argue that their implementation follows best practices and is immune to more pedestrian timing attacks.

Despite these assertions there remain a number of functions in their TLS library that seem to be susceptible to a timing attack. In particular the CBC_unpad function bails out early when bad padding is detected and we were hoping that this would be enough to execute an attack.

4 The (Incomplete) Attack

Having laid the groundwork, in this section we are going to describe our original plan for executing a padding-oracle attack on CBC-mode encryption. Then we are going to describe the successful steps we took. We will follow that by describing what did not work. Finally we will describe possible steps to overcome our current problems.

4.1 Our Plan

Our original plan for this project was to break an application that used CBC-mode encryption by using a variation of Serge Vaudenay's padding oracle attack. To do this we imagined that we would begin with some toy examples to develop a technique that would work. Then we would hopefully find an application known to be vulnerable, hone our attack on that, and finally find a new application that was vulnerable and adapt our attack to the new challenges of that implementation and carry out the attack. We now discuss how we fell short of this goal.

4.2 Original Tests

Our first step to make sure that we understood the attack on a practical level was to explore encrypting plaintext with the OpenSSL command line tool. Knowing that we needed to differentiate errors, we then modified the encrypted text in different places to see whether we could differentiate padding errors from garbled decryption. At this point we were not incorporating a MAC.

Having gotten a rudimentary understanding in this way we then wrote code in C that could take in a ciphertext, and given a padding oracle that directly responded with whether or not the padding was correct could decipher the plaintext. Eventually we expanded this code so that it could do this decryption with simple timing differences between bad padding and bad decryption, what would be a bad MAC in a more practical attack. A portion of the code for this attack can be seen in Listing 1. Expanding the attack to include timing was relatively straight-forward, however, even with our toy example where the oracle artificially waited for a set number of milliseconds we found that we could not reliably decrypt with a single query per possible byte. Presumably due to context switches, we found that sending 10 messages per byte and taking the mean was sufficient to identify the correct byte in the plaintext. While this might not be an

accurate count in this idealized case, it is clear that sending multiple messages significantly increases the time of the attack and the amount of data that must be exchanged with the server, making detection by an alert administrator more likely.

Although the toy timing attack was successful, the timing threshold currently needs to be manually set. We would like to expand this so that it could adjust its timing sensitivity based on the responses it is getting from the oracle, perhaps by sending multiple messages per byte at the beginning and then adjusting the number of messages sent as the timing becomes more apparent based on the observed variation. That is, we would have the tool automatically look for measurable timing differences.

Listing 1: Toy Timing Attack

```
// loop through blocks backwards b
// is the length of the ciphertext
for(b; b > 0; b -= IVSIZE)
  // make a copy of target
  memcpy(test_ct, ct, b + IVSIZE);
  for (j = 16; j < 32; ++j)
    // set already deduced bytes
    // to right value for next
    for (i = 16; i < j; ++i)
      test_ct[b +IVSIZE - 1 - i]
          ^= (j - 16 ^ j - 15);
    // try all possible byte values
    for(i = 0; i < 256; ++i)
      // try next byte
      test_ct[b + IVSIZE - 1 - j]
        += 1;
      // start timer
      gettimeofday(&start, NULL);
      // call padding oracle 10
      // times
      for (l = 0; l < 10; ++l)
        timing_oracle(test_ct +
          IVSIZE, key, test_ct, b);
      // end timer
      ...
      // calculate elapsed time
      ...
      // if longer than others
      // we found the right byte
        nextchar =
          test_ct[b+IVSIZE-1 - j] ^
          j-15^ct[b+IVSIZE-1-j];
        pt[k++] = nextchar;
        // get padding
        if (b == len && j == 16)
          padding = nextchar;
        // move on to next byte
        break;
```

4.3 nqsb and Expanding our attack

Our planned next step was to replicate an old attack, however, we had difficulty finding concrete discussions of an attack on a broken implementation that we could replicate. Therefore, perhaps a little early, moved on to the next step and started trying to figure out how to leverage small timing effects in nqsb to execute an attack.

We envisioned three main steps in this part. First, we needed to set up a server that was running an implementation of nqsb-TLS. Second, we needed to modify a TLS client so that we could send manipulated messages to the server. Third, we needed to fiqure out how to insert ourselves into a TLS connection to use the server as an oracle. Given our lack of familiarity with OCaml we decided to begin with setting up the OCaml code.

Most of the packages that the developers of nqsb-TLS have written are available on opam, which is a package manager for OCaml code. Here we briefly describe the problems we encountered using their software. First we tried to set up their full server implemented on top of MirageOS using first 32-bit and then 64 bit installations of Ubuntu 14.04. After multiple reinstallations, trying various versions of OCaml, and installing and removing various dependencies we were unable to get the server to build properly and had to abandon it.

Fortunately, the authors had also implemented tlstunnel (available through the website in footnote 1), which uses their TLS implementation to provide secure access to an established, insecure server. This application we were able to install using opam. Once we had done this it was straightforward to test it by using python SimpleHTTPServer as the backend and OpenSSL command line tools as a client. Doing this we were successfully able to securely connect to the server throught the tunnel. Realizing that to do an attack it could be helpful to keep the connection open we also successfully tested this setup using the netcat tool as the backend.

Having successfully replicated their setup on our machines the next step was to write a program that could connect to the server and implement the attack or at least do some timing of different responses. As a first step, we successfully wrote a client in C, using the OpenSSL libary to connect to the tunnel and send messages to the server, through the tunnel. Once the connection was established we used the `SSL_write()` function to transmit data.

With this client working and talking TLS 1.2 with the tunnel we used Wireshark to capture and examine the packets we were sending to better understand their structure. Since we were using netcat it was easy to send a number of short messages of known content to the server. With a single block or less of plaintext we saw 69 byte TLS messages. The first five bytes were the header leav-

ing a 64 byte ciphertext, later we will discuss the differences when using 1.0.

Having gotten to this point we thought that the immediate next step would be relatively straight-forward. First, we would establish a connection and then within that connection we would send a message. Then to simulate an attack we would send a number of variations of that message to test out the vulnerability of the server to padding attacks by measuring the timing. Then we would then see to what extent this could be turned into a more realistic attack perhaps using methods similar to [4] or by the use of the chained IVs in TLS 1.0 to allow the attacker to interject himself into the connection and decode the message.

However, we immediately ran into another technical problem of how to send modified messages. Because of the high-level nature of the OpenSSL once a connection has been established all communication is through the `SSL_write()` function or similar BIO functions, so our program could only send good messages it could not send bad ones to probe the server and carry out the attack.

At first we thought that this would have a relatively straight-forward solution and began reading the, somewhat unsatisfactory, OpenSSl documententation to see how we could get inside the `SSL_write()` function and modify a message or get a copy of it before transmission. During this investigation we discovered that the function does not use publicly available buffers and we could not find a way to access the buffer.

Our next idea was that if we could get access to the current symmetric keys for encryption and MACing we cound construct our own TLS packets and after the handshake write our own code to transmit the data messages, keeping track of the relevant data. This approach would not be very stable because of a lack of support for renegotiation or handling any error conditions, but we hoped that it would be sufficient for our purposes. Unfortunately, while we were able to access data structures and 'getter' functions to read various pieces of information about the connection, such as the names of the current ciphers and the number of messages, we were not able to find a way to access the keys. In short, we were stymied again.

Having been unable to find a way to use OpenSSL for our purposes we began looking for another TLS implementation that allowed lower level access. After briefly looking through a number of TLS implementations at essentially the same level as OpenSSL, such as GnuTLS and wolfSSL, we found a book-length implementation [3] of a TLS 1.0 client from the ground up by Joshua Davies. First, we began trying to see whether we could implement enough of the code ourselves to patch together a client with our existing OpenSSL code. However, this proved to be a daunting and error-prone task and we fortunately found on GitHub [8] an implementation of the code from this book. Using this code we intended to get a connection and then modify the write functions so that we could send arbitrary modifications of legitimate messages. We had no problem getting the code to compile (we will leave the wisdom of downloading and running essentially random code from GitHub for another paper[2]) and quickly made some adjustments so that it could send some simple messages through the tlstunnel we had set up.

However, having made the changes we found that the new client would start the handshake with the server, but could not successfully complete it, complaining of a message from the server containing an unrecognized cipher suite. We then went back to the original downloaded code, which purportedly executes a `GET` request using TLS 1.0 to the passed in url and tested it against a site that we had found previously that only spoke TLS 1.0. After this failed with a different error we tried the client against a number of sites in the Alexa top 50. While we received a number of new errors, including from sites that did not have TLS enabled, we were not able to complete the handshake with any of them. We are not sure whether this is a bug in the code, evolutions in how most servers implement TLS since [3] was written in 2011, or from some other cause, but we had reached another dead end.

After carrying out these tests we implemented a number of changes to the client code, including modifying the cipher suites it offered and trying to ignore various warnings, but were unsuccessful in getting a working client.

4.4 Man in the Middle

Our final effort at getting the attack working, or at least to get some timing data, was to write a man-in-the-middle client. The MitM acted as a server for an OpenSSL command line client and initially simply passed through messages, acting as a clinet to one end of the tlstunnel. The idea was that since we were unable to modify the packets within our previous clients, we would instead intercept the packets and then modify some of them before passing them along. We would do this by allowing the handshake messages and the first data message to pass through the MitM, and then begin to act as the client to try to implement a padding oracle attack.

Initially, after some debugging, we were successful in using the new MitM client to pass through the handshake and record messages in both directions to allow the OpenSSL client to connect to and communicate with the tunnel using TLS 1.0. To better understand the packets we again used Wireshark. When using TLS 1.2 we noted

[2] See `http://gkoberger.github.io/stacksort/` for an xkcd inspired sorting variation on this

that the packets were 69 bytes which made sense with a five byte header, in this case, using TLS 1.0, however they were 74 bytes, leaving an additional five bytes unaccounted for. To carry out the attack we would need to fully understand this packet.

We planned on proceeding by saving a message from the client and then modifying it to send modified messages, using chained-IVs, to the server to get timing information and hopefully decrypt the message. However, our initial test of this approach thwarted that hope.

To see whether we could see the expected error messages we began by sending a duplicate message from the MitM. We did get the expected bad record MAC alert, but then realized that this error is always treated as fatal, presumably to thwart just such shenanigans. This immediately terminates the connection, making this avenue of approach a challenging one. To overcome this problem we would need a more sophisticated attack that can cross over broken sessions such as that in [2]. If we could modify that attack this approach might be promising.

4.5 Possible Solutions

While we have not been able to create a functioning attack client we can see some possible ways forward. The first solution would be a better understanding of OpenSSL so that we could gain access to the private connection data to access the keys or be able to modify the buffer. That being said, we have spent substantial time looking at OpenSSL documents and are not hopeful about finding such a solution without new understanding about how to look. Second, we could take the TLS implementation derived from [3] and implemented by Li [8] and try to debug it and turn it into a working implementation where we had full access to the keys. Third, we found suggestions that other functional implementations, such as Microsoft's SChannel, may allow more access to lower-level functions that would facilitate our approach. The reason we took our original line doing the attack within a legitimate connection, was that while it was not realistic, it seemed like the most straightforward way to get realistic timing measurements and it proved much harder than we were anticipating.

An alternative to this approach would be to begin by making the attack even less realistic and simply time the running of the relevant functions from their implementation, artificially constructing messages that only have a bad MAC and ones that have a bad padding and see whether there are timeable differences. Once this was done we could then build from there. One reason we did not take this approach originally was our lack of familiarity with OCaml.

Alternatively we could build off our MitM client approach and try to expand this into a functioning attack using more sophisticated approach. As discussed in [2] this would require applying knowledge about the plaintext to execute an attack.

5 Next Steps

When we overcome the problems in the last section there remain a number of issues with converting what we would have then into a successful attack. In this section we will discuss some possible application of our near attack and discuss how we can expand upon it.

If we develop a functioning client with the ability to get timing from different TLS implementations it is possible that it could be used to test various TLS implementations for timing vulnerabilities suggesting whether or not they might be vulnerable to a padding oracle attack. In addition to different implementations we would be able to test them with different numbers of hops between the client and the server to see how realistic attacks depending on timing might be. Finally, we could explore to what extent increasing the number of messages sent to discover each byte, as discussed above, is able to overcome distances.

Having bumped into various safeguards in TLS as we developed our attack the next step would be to better understand nqsb-TLS and fully understand which improved methods would be most likely to be fruitful against it. As discussed previously this could be related to how to overcome closed connections in the wake of MAC-record errors and how to leverage information about the plaintext as discussed in [6] to mount an attack.

6 Conclusion

Although our efforts were not successful, we came away with a much better understanding of how CBC encryption works and a better appreciation of the difficulties of mounting an actual attack, as opposed to just understanding how it works in theory. Being stymied by a wide variety of problems helped us to better understand the workings of TLS and its various implementations as we progressed. As we neared the end of this project we continued to think that we had nearly solved our problems, but everything we thought of seemed to lead nowhere.

One lesson from this project is that we should have tried to make our attack more realistic on known vulnerabilities before trying to extend it to a new system. Had we found an implementation known to be vulnerable and honed our skills there, we likely would have had a more successful and productive attempt at breaking nqsb. We look forward to using the experience from this project for more successful cracking in the future.

References

[1] AL FARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 526–540.

[2] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password interception in a ssl/tls channel. In *Advances in Cryptology-CRYPTO 2003*. Springer, 2003, pp. 583–599.

[3] DAVIES, J. *Implementing SSL / TLS Using Cryptography and PKI*. IT Pro. Wiley, 2011.

[4] DUONG, T., AND RIZZO, J. Here come theŁ ninjas. *Unpublished manuscript 320* (2011).

[5] GUTMANN, P. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), Sept. 2014.

[6] JAGER, T., AND SOMOROVSKY, J. How to break xml encryption. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 413–422.

[7] KALOPER-MERŠINJAK, D., MEHNERT, H., MADHAVAPEDDY, A., AND SEWELL, P. Not-quite-so-broken tls: lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 223–238.

[8] LI, W. sslbook. `https://github.com/liweinan/recipies/tree/master/sslbook`, 2013.

[9] MARLINSPIKE, M. The cryptographic doom principle. `https://moxie.org/blog/the-cryptographic-doom-principle/`. Accessed: 2016-04-27.

[10] VAUDENAY, S. Security flaws induced by cbc paddingapplications to ssl, ipsec, wtls... In *Advances in CryptologyEUROCRYPT 2002* (2002), Springer, pp. 534–545.