Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities

Robert Merget and Juraj Somorovsky, *Ruhr University Bochum;* Nimrod Aviram, *Tel Aviv University;* Craig Young, *Tripwire VERT;* Janis Fliegenschmidt and Jörg Schwenk, *Ruhr University Bochum;* Yuval Shavitt, *Tel Aviv University*

https://www.usenix.org/conference/usenixsecurity19/presentation/merget

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities

Robert Merget[1], Juraj Somorovsky[1], Nimrod Aviram[2], Craig Young[3], Janis Fliegenschmidt[1], Jörg Schwenk[1], and Yuval Shavitt[2]

[1]Ruhr University Bochum
[2]Department of Electrical Engineering, Tel Aviv University
[3]Tripwire VERT

Abstract

The TLS protocol provides encryption, data integrity, and authentication on the modern Internet. Despite the protocol's importance, currently-deployed TLS versions use obsolete cryptographic algorithms which have been broken using various attacks. One prominent class of such attacks is CBC padding oracle attacks. These attacks allow an adversary to decrypt TLS traffic by observing different server behaviors which depend on the validity of CBC padding.

We present the first large-scale scan for CBC padding oracle vulnerabilities in TLS implementations on the modern Internet. Our scan revealed vulnerabilities in 1.83% of the Alexa Top Million websites, detecting nearly 100 different vulnerabilities. Our scanner observes subtle differences in server behavior, such as responding with different TLS alerts, or with different TCP header flags.

We used a novel scanning methodology consisting of three steps. First, we created a large set of probes that detect vulnerabilities at a considerable scanning cost. We then reduced the number of probes using a preliminary scan, such that a smaller set of probes has the same detection rate but is small enough to be used in large-scale scans. Finally, we used the reduced set to scan at scale, and clustered our findings with a novel approach using graph drawing algorithms.

Contrary to common wisdom, exploiting CBC padding oracles does not necessarily require performing precise timing measurements. We detected vulnerabilities that can be exploited simply by observing the content of different server responses. These vulnerabilities pose a significantly larger threat in practice than previously assumed.

1 Introduction

In 2002, Vaudenay presented an attack which targets messages encrypted with the Cipher Block Chaining (CBC) mode of operation [39]. The attack exploits the malleability of the CBC mode, which allows altering the ciphertext such that specific cleartext bits are flipped, without knowledge of the encryption key. The attack requires a server that decrypts a message and responds with 1 or 0 based on the message validity. This behavior essentially provides the attacker with a cryptographic oracle which can be used to mount an adaptive chosen-ciphertext attack. The attacker exploits this behavior to decrypt messages by executing adaptive queries.Vaudenay exploited a specific form of vulnerable behavior, where implementations validate the CBC padding structure and respond with 1 or 0 accordingly.

This class of attacks has been termed *padding oracle attacks*. Different forms of padding oracle attacks were demonstrated to break cryptographic hardware [6], XML Encryption [23], or web technologies like Java Server Faces [33] and ASP.NET web applications [15]. Rizzo and Duong used a padding oracle attack to steal secrets and forge authentication tokens, gaining access to sensitive data [15]. In all of these works, the attacker was able to use a direct side channel – different error messages – to instantiate a padding oracle and decrypt confidential data.

Transport Layer Security (TLS) employs CBC mode in a MAC-then-Pad-then-Encrypt scheme which makes it potentially vulnerable to these attacks. Indeed, different types of CBC padding oracles have been used to break confidentiality TLS connections [39, 4, 3, 20]. All these attacks require the attacker to perform precise timing measurements. This requirement stems from the properties of the TLS protocol; after establishing a TLS connection, all TLS error messages are sent encrypted and are of the same length. Therefore, even if an attacker is able to cause the server to send different error messages, the attacker is generally unable to distinguish between the different encrypted responses.

Since most previous analyses have only analyzed padding oracle attacks based on timing side channels, they required testing an implementation in a local environment. These evaluations uncovered many new vulnerabilities [4, 3, 20]. However, implementing a proper countermeasure to these vulnerabilities is very challenging and requires complex constant-time implementations. It is not surprising that the implementation of such countermeasures could introduce

new attacks. For example, in an attempt to fix the Lucky 13 padding oracle, the OpenSSL cryptographic library introduced a different vulnerability where OpenSSL responded with different TLS alert messages [37]. Analysis of implementations in lab settings therefore requires laborious testing for each new version of different implementations. This is obviously unrealistic, and therefore this type of analysis is performed sporadically.

Given the complexity of constant-time TLS padding verification, we expect that vulnerabilities similar to the one introduced by OpenSSL [37] could have been introduced in other implementations as well. Therefore, this work moves away from the above method of lab analyses and evaluates CBC padding oracles using large-scale Internet scans. We attempt to answer the two following questions: *How prevalent are padding oracle vulnerabilities? Are these attacks only exploitable by using timing side-channels?*

Contributions. In our work, we employ a novel scanning methodology that is capable of scanning for TLS CBC padding oracles at scale. We use this methodology to find new padding oracle vulnerabilities and perform responsible disclosures. We identify nearly 100 different padding oracles. We show that some of them can be exploitable without subtle timing side channels and thus pose a significantly larger threat in practice compared to most recently-discovered padding oracles.

New large-scale scanning methodology. Scanning at scale for padding oracles is challenging. Such scans detect vulnerabilities by sending different malformed inputs and observing server behavior. As shown by Böck et al. [9], in some cases these inputs only trigger vulnerabilities when using specific TLS versions or cipher suites. Scanning with all possible combinations of protocol versions, cipher suites and malformed inputs is not feasible since it would require an enormous number of connections to each scanned host.

We overcome this limitation by carefully selecting a set of probes, which allows for effective scans at scale. We systematically analyzed padding oracles previously described in the literature [39, 4, 3, 20, 37, 27, 25, 10, 29, 28]. We then carefully selected 25 inputs exhibiting padding oracle malformities, which we refer to as *malformed records*. These TLS records exhibit different combinations of valid and invalid padding and MAC, and are generated using the TLS-Attacker framework [37].

Even with only 25 malformed records, scanning with every combination of malformed record, TLS version and cipher suite would be impractical. We refer to these combinations as *test vectors*. We performed a preliminary scan on 50,000 random TLS hosts with all test vectors. We then reduced our test vector set, such that all vulnerabilities detected in the preliminary scan are still triggered by the reduced set. We were able to scan the Alexa Top 1 Million websites with this reduced test vector set within three days. Our scanner observes different server responses, not only in the TLS layer, but also in the TCP layer, similar to [9]. Our results indicate that about 1.83% of TLS servers are vulnerable to CBC padding oracle attacks.

Minimizing false positives. When a host first displays vulnerable behavior, we rescan it to make sure the behavior is not a scanning artifact. We only consider a host to be vulnerable if it responds identically in three separate scans to each of our test vectors. It is unlikely that hosts will be mislabeled as vulnerable under this criterion. We therefore believe our statistics for vulnerability are a conservative lower estimate.

Nearly 100 different padding oracle vulnerabilities. The detected vulnerabilities have to be clustered in order to notify different vendors. Until now, this was done manually [9]. To achieve this automatically, we re-scan vulnerable hosts against a larger set of test vectors. We refer to the set of the host responses to all test vectors as the host's *response map*. This response map is essentially a fingerprint of the host's vulnerability. We then cluster the scanned hosts according to their response maps. This process identified 93 different response maps, i.e., 93 different vulnerabilities. These vulnerabilities include different behaviors, ranging from typical padding oracles with different TLS alert messages [39], to TCP connection timeouts triggered by specific invalid MAC bytes, or closed connections observed when using invalid padding values.

We treat distinct response maps as distinct vulnerabilities. We argue that this is the natural way to count vulnerabilities since it captures the case of the same vulnerability occurring in similar, yet different implementations. Consider two hosts that respond identically to all test vectors. These hosts likely share an identical or very similar part of the implementation that causes the vulnerability to manifest with identical response maps. However, they do not necessarily share the exact same code. They may use different versions of the same TLS library, or two different libraries with a shared component.

Effective clustering of vulnerable hosts. Before we responsibly disclosed our findings to the affected parties, we grouped the vulnerable hosts by their response maps. To further refine our grouped servers, we used a novel approach based on a two-dimensional force-directed graph drawing ForceAtlas2 algorithm [21]. This algorithm allowed us to create a graph of vulnerable server hosts and thus, efficiently handle our responsible disclosure process.

New vulnerabilities that are realistic to exploit. For padding vulnerabilities to be exploitable, the attacker needs

to distinguish between different responses to correct and incorrect padding. This is usually not the case in TLS: Even if a server sends two different alert messages, the messages are encrypted, and the attacker cannot observe the difference. For this reason, most previous padding oracle attacks against TLS relied on timing measurements to distinguish between different error cases [4, 3, 20].

However, we show that many TLS implementations exhibit *observable* differences between correct and incorrect padding. For example, a server may gracefully close the TCP connection in one error case and ungracefully close it in a different case. Similarly, some servers send a different number of alert messages depending on specific padding errors. Both behaviors are easily observable.

Responsible disclosure and ethical considerations. In collaboration with affected website owners, we responsibly disclosed our findings to several vulnerable vendors. As a result of a successful attack, the attacker is able to decrypt secret values repeatedly transmitted in the TLS connection. By performing our scans, we were not able to reconstruct server private keys or other confidential data. We performed our scans with dummy data and never attempted to decrypt real user traffic.

We responsibly disclosed our findings among others to the following vendors and affected parties: IBM, Amazon, Slack, Cisco, Citrix, Oracle, Heroku, Netflix, Sonicwall, Venmo and Vine.

2 Background

The TLS protocol provides confidentiality, integrity, and authentication on the modern Internet. The latest version of the protocol is TLS 1.3 [31]. This version is gradually being deployed as of this writing. Until TLS 1.3 is fully deployed, the latest version in widespread use is TLS 1.2 [14]. Modern clients and servers typically also support two previous versions, TLS 1.0 and 1.1 [12, 13]. In the rest of the paper, we discuss only versions 1.0 to 1.2, which are commonly used today and share a similar structure.

The TLS protocol consists of two phases. In the first phase, called the *handshake*, the client and server choose the cryptographic algorithms that will be used for the session and establish session keys. In the second phase, the peers can securely send and receive application data, which is encrypted and authenticated using the keys and algorithms established in the previous phase.

The aforementioned choice of cryptographic algorithms is called a TLS *cipher suite* [14]. More precisely, a cipher suite is a concrete selection of algorithms for all of the required cryptographic tasks. Cipher suites are named by concatenating their choices for these algorithms. For example, the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` uses RSA public-key encryption in order to establish a shared session key in the first phase, and also uses symmetric AES-CBC encryption with a 128-bit key and SHA-1-based HMACs in order to encrypt and authenticate data in the second phase.

2.1 The TLS Handshake

The client initiates the TLS handshake with a `ClientHello` message. This message advertises the TLS versions and cipher suites supported by the client. The server then responds with a `ServerHello` message specifying the selected cipher suite. It also sends its certificate in the `Certificate` message and indicates the end of transmission with the `ServerHelloDone` message. The client then generates a secret value called the `premaster secret`, encrypts it under the server's RSA key, and sends the encrypted ciphertext in a `ClientKeyExchange` message. Having shared knowledge of the premaster secret, both parties now derive symmetric encryption and MAC keys to be used in the session, based on the premaster secret. Finally, both parties send the `ChangeCipherSpec` and `Finished` messages. The `ChangeCipherSpec` message notifies the receiving peer that subsequent messages will be encrypted and authenticated under the session keys, and using the symmetric encryption and HMAC algorithms specified in the cipher suite. The `Finished` message contains an HMAC computed over all the previous handshake messages based on a key derived from the premaster secret. As this message is sent after the `ChangeCipherSpec` message, it is the first message in the session which is encrypted and authenticated using symmetric encryption and MAC. If the `Finished` message correctly decrypts and verifies on both sides, both parties can now securely exchange application data.

2.2 CBC Mode

There are many possible encryption algorithms in TLS, but we focus on the CBC encryption mode in this work. In CBC mode, each plaintext block is XOR'ed to the previous ciphertext block before being encrypted by the block cipher. Formally, if we denote plaintext blocks by $p_i, i = 0, \ldots$, ciphertext blocks by c_i and the encryption with a block cipher under key k as $Enc_k(\cdot)$, then $c_i = Enc_k(p_i \oplus c_{i-1}), i = 1, \ldots$. The above holds for all blocks except the first one, where there is no previous ciphertext block – instead, that block is XOR'ed with an initialization vector (IV) before encryption: $c_0 = Enc_k(p_0 \oplus IV)$.

CBC mode malleability. The CBC mode allows an attacker to perform meaningful plaintext modifications without knowing the symmetric key. Concretely, assume the attacker knows some block of the original plaintext p_i, and wants to alter the ciphertext such that block i instead decrypts to p'_i. The attacker can change the previous ciphertext block c_{i-1} to $c'_{i-1} = c_{i-1} \oplus p_i \oplus p'_i$. This comes at the cost of

corrupting the previous block, which now decrypts to some value that the attacker, in general, cannot predict.

Furthermore, the attacker can change the order of blocks while using this technique. If the attacker knows the plaintext block p_i and replaces ciphertext block c_j with c_i, then block j will now decrypt to $p'_j = p_i \oplus c_{i-1} \oplus c_{j-1}$.

This "malleability" property of CBC mode has been used in many cryptographic attacks, and is also a cornerstone of the attacks presented here.

2.3 TLS Record Layer

The TLS record layer encapsulates protocol messages. In essence, the record layer wraps the protocol message with a header containing the message length, message type, and protocol version. Once `ChangeCipherSpec` messages are exchanged, subsequent TLS records will encapsulate messages which are encrypted.

In our work, we focus on cipher suites using the CBC mode of operation. These cipher suites use a Message Authentication Code (MAC) to protect the authenticity of TLS records and encrypt application data using a block cipher in CBC mode (e.g., AES or 3DES). The TLS specification prescribes the *MAC-then-Pad-then-Encrypt* mechanism [14]. The encryptor first computes a MAC over the plaintext, concatenates the MAC to the plaintext, pads the message such that its length is a multiple of the block length, and finally encrypts the MAC'ed and padded plaintext using a block cipher in CBC mode.

TLS specifies the exact value of the padding bytes. The last byte of the padded plaintext specifies how many padding bytes are used, excluding that last byte. The value of the rest of the padding bytes is identical to the value of the last byte. For example, if 4 padding bytes are used including the last byte, then the value of all four bytes will be `0x03`.

To demonstrate the full process, if the encryptor encrypts five bytes of data with the `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite, he uses HMAC-SHA (whose output is 20 bytes long) and AES-CBC. After applying HMAC-SHA to the original plaintext, the concatenation is 25 bytes in length, which fits into two AES 16-byte blocks. The encryptor will typically select the minimum viable amount of padding, which would be 7 bytes in this case. The first block contains the data and the first 11 HMAC bytes. The second block contains the remaining 9 HMAC bytes and 7 bytes of padding `0x06`, see Figure 1. Note that the encryptor can also choose longer padding and append 23, 39, ...or 247 padding bytes (while setting the value of the padding bytes accordingly).

3 A Brief History of Padding Oracle Attacks

One of the main design failures in SSLv3 and TLS is the specification of the *MAC-then-Pad-then-Encrypt* scheme in CBC cipher suites. This scheme was responsible for a series of attacks on TLS implementations named padding oracle attacks. Even though the countermeasures are explicitly summarized in the TLS specification [14, Section 6.2.3.2], their correct implementation is challenging.[1]

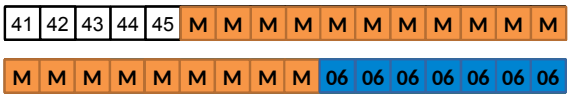

Figure 1: When processing five plaintext bytes with AES-CBC and HMAC-SHA, the encryptor needs to append 20 bytes of the HMAC-SHA output and seven bytes of padding.

3.1 Vaudenay's Padding Oracles

In 2002, Vaudenay showed that the MAC-then-Pad-then-Encrypt scheme introduces potential vulnerabilities in security protocols, in the form of so-called padding oracles [39]. The attacks leveraging these vulnerabilities are based on the malleability of the CBC mode of operation. We focus on the case of TLS.

Consider the TLS record layer when using CBC mode. After decryption, the decrypting party needs to verify the padding bytes and the MAC bytes. The natural way to implement these two checks is first to verify the padding bytes and, if they verify correctly, then verify the MAC bytes. If the padding bytes are invalid, it is natural for an implementation to emit an error message, without checking the MAC bytes. On the other hand, if the padding bytes are valid but the MAC is invalid, it is then natural to emit a (potentially different) error message.

Assume a decryptor that indeed emits two different error messages in these cases. The attacker can decrypt the last byte of any message block p_i as follows. He sets the last ciphertext block to c_i and replaces the last byte of the previous block c_{i-1} with a value between 0 and 255. If the last cleartext byte is `0x00`, then the padding will be valid (other forms of valid padding are much less likely). When the padding byte correctly verifies, the attacker detects this by observing that the decryptor emitted an "invalid MAC" error, rather than an "invalid padding" error. The attacker learns the value of the last byte of p_i after sending at most 255 ciphertexts to the decryptor.

Using his knowledge of the last plaintext byte, the attacker can proceed to decrypt the second-to-last byte of p_i. By doing so, he aims to create valid padding of length 2. More generally, using this technique, the attacker can iteratively decrypt every byte in p_i. We omit the formal description of the rest of the attack and refer the reader to [39].

[1] We note that the countermeasures summarized in [14] do not protect from timing-based attacks [4].

Note that the above attack relies on the ability to distinguish between ciphertexts decrypting to valid and invalid padding. It would therefore appear trivial for TLS implementations to prevent this attack by making sure they always emit the same error message. Indeed, Vaudenay was unaware of a way for an attacker to directly distinguish between these two cases in the context of TLS. The reason is that even if the TLS error messages differ, their distinction is impossible since they are encrypted with TLS session keys. This is one of the challenges we address in our work.

3.2 BEAST Attack Model

One question left open in Vaudenay's paper is how to exploit what he terms an "exploding oracle" – an oracle that is usable only until it first returns a negative answer. This models the problem where a TLS implementation will abort the session as soon as a message doesn't decrypt correctly. Hence, an attacker that relies on changing messages in a TLS session would not be able to continue the attack as soon as the first decryption error arises.

Canvel et al. used a model where the client repeatedly connects to the server [11], observing that this occurs due to polling behavior of email clients at the time, and exfiltrating an authentication password. The BEAST attack [34] essentially used the same model, but rather relied on the behavior of modern web browsers. In the simplest form of the BEAST model, a victim is tricked into visiting a malicious website controlled by the attacker. That website contains javascript which causes the victim browser to repeatedly connect to the victim website. Every website request then contains the user authentication cookie, which is automatically sent by the browser. This behavior allows the attacker to force the victim to repeatedly send encrypted values to the server.

Our attacks work in this model. We assume that the attacker can cause the victim client to repeatedly connect to a victim server while retransmitting the same sensitive information. We also assume the attacker is a man in the middle (MitM) and can change messages in transit. This model has now become standard in literature for modern attacks.

3.3 POODLE

The predecessor to TLS, SSLv3, uses a similar *MAC-then-Pad-then-Encrypt* scheme. However, unlike TLS, the value of the padding bytes in SSLv3 is under-specified. The last byte of the plaintext denotes how many padding bytes are present, but the rest of the padding bytes can take any value.

Consider a message with one full block of 16 padding bytes. The last block of plaintext will have a last byte of `0x0F`, and the first 15 bytes can take any value. Therefore, an attacker can use the techniques described in Section 3.1 to replace the last block with any block whose last byte decrypts to `0x0F`, and obtain a validly padded message. This property of SSLv3 led to a devastating attack called "POODLE". See [27] for a full description of the attack.

Although POODLE relies on the under-specification of the padding bytes in SSLv3, it surprisingly also affects TLS implementations. In essence, there is nothing forcing a careless TLS developer to verify the (specified) padding bytes after decryption; a TLS implementation will interoperate just fine even if it does not check the padding bytes at all. In fact, it is *easier* for the developer to reuse the same code that handles SSLv3 padding in a TLS implementation. This has led to a variant of the POODLE attack that affects TLS implementations [25]. Even after these two high-profile discoveries, variants of POODLE continued emerging [10, 29, 28]. These works detected different TLS record processing vulnerabilities; some TLS implementations only verified the first MAC byte, the others skipped validation of specific padding bytes.

3.4 Lucky 13 and Other Timing Attacks

In 2013, AlFardan and Paterson [4] used a similar technique to break TLS confidentiality and dubbed their attack "Lucky 13". The attack relies on an important observation: Common HMAC functions require different processing times when processing inputs of different lengths. By performing clever padding byte manipulations, the attacker can force the server to execute HMAC computations on plaintexts of different lengths. This is because the padding length determines the amount of data used as input into the HMAC function. The attacker can then measure the different processing times and learn information about the padding byte. We refer the reader to [4] for the full attack description.

The fix to Lucky 13 was to change the MAC verification code in TLS implementations to be constant-time, regardless of the number of processed cleartext blocks. This is possible, but writing and maintaining such code is hard, even for experts. In 2016, Somorovsky identified a bug in the patched code of OpenSSL [37]. The bug introduced a similar and even more severe vulnerability which allowed an attacker to distinguish between two alert messages. A different message could be triggered if the decrypted message only contained two or more valid padding blocks.

Amazon's s2n TLS library was released in 2015 [24], after the Lucky 13 attack was published. s2n's developers were aware of Lucky 13 and introduced specific countermeasures that seemed to render the code constant-time, thereby preventing the attack. They also introduced randomized timing delays to make the attack more difficult, in the unexpected case that the code turned out to be vulnerable. Despite all these efforts, s2n was still vulnerable to variants of Lucky 13 [3, 35]. All vulnerabilities were found despite the code having been formally verified.

3.5 Bleichenbacher's Attack and its Variants

Bleichenbacher's attack [8] is also a form of a padding oracle attack. Rather than targeting symmetric encryption, it targets a padding scheme used in RSA encryption, called PKCS#1 v1.5. It also similarly exploits a malleability property of RSA encryption and relies on a decryptor (i.e., a server) emitting error messages in case of invalidly-padded cleartexts. The standard countermeasure is similar to that of CBC padding oracles; the server must not behave differently when encountering error states in RSA decryption. This countermeasure has become part of the TLS standard.

However, implementing the countermeasure correctly is challenging. Böck et al. scanned for vulnerable TLS servers vulnerable to Bleichenbacher's attack [9]. They found vulnerabilities in servers used by high-profile websites such as Facebook and Paypal. Interestingly, their vulnerabilities could be triggered by using different TLS protocol flows or exploiting TCP connection states (TCP resets or timeouts). As with CBC padding oracles, Bleichenbacher's attack shows a similar sequence of an attack variant being discovered every few years in different contexts [26, 22, 6].

4 Scanning and Evaluation Methodology

The ultimate goal of our research is to estimate the number and the impact of padding oracle vulnerabilities and report our findings to the responsible vendors. To accomplish this, we proceed in three steps. We first define a list of test vectors potentially triggering observable differences which result in padding oracles. We then reduce this test vector list and perform a large-scale scan. Finally, we analyze the identified vulnerabilities and responsible vendors.

4.1 Test Vector Generation

In order to detect padding oracles in implementations, we connect and send various malformed records. These records contain different malformities in regards to the padding, MAC, and application data. We then observe if there are any differences in responses, in the TLS layer, or in lower layers. An implementation that responds differently to two malformed records may be vulnerable.

It is infeasible to test with all possible malformed records. For example, a vulnerable implementation could correctly check all padding bytes unless the padding bytes are exactly 16 bytes long, in which case the implementation does not check a specific bit in the padding.[2] Since there could be up to 256 padding bytes, testing the correct validation of each bit for all possible padding lengths would require testing with $\sum_{i=1}^{256} 8i = 263,168$ different records. These records need to be tested with different cipher suites or protocol versions which makes such a comprehensive test infeasible. We therefore carefully selected a set of malformed records which are motivated by previous research.

We concede this way of selecting the set of malformed records means we can only detect vulnerabilities that are similar to known ones. However, this approach is cost-effective and well-suited to large-scale scans. Since only a limited number of messages can be sent to individual servers during large-scale scans, automatic approaches for the test vector generation, like fuzzing, are usually infeasible.

4.1.1 Malformed Records

Our malformed records are all 80 bytes in length. Equal lengths ensure that differences in responses are likely caused by a padding oracle vulnerability and are not false positives triggered by different record lengths. Unusual record lengths may lead to errors that are unrelated to decryption; for example, recent OpenSSL versions respond with a different error message if the encrypted TLS record is shorter than the MAC length. We decided to use 80 bytes to have enough room for an HMAC output combined with two full padding blocks. This allows us to construct records protected by SHA-384, whose output is 48 bytes in length. We summarize our 25 malformed records in the following paragraphs. See also Table 1 for a summary of these malformed records for the case of `TLS_RSA_WITH_AES_128_CBC_SHA`.

Flipped MAC bits. We start with a valid record containing application data, a MAC, and four padding bytes. We then create three malformed records based on this record: One by flipping the most significant bit in the first MAC byte, one by flipping a middle bit in the middle of the MAC bytes, and one by flipping the least significant bit of the last MAC byte. We chose these malformed records to detect implementations where the MAC is not completely checked. The specific bit flipping positions are motivated by the recent OpenSSL vulnerability [1], where OpenSSL only checked the least significant bit of each byte on some platforms, and by further vulnerabilities caused by incomplete MAC validations [29, 28].

Missing One MAC byte. We start with a valid record containing empty application data, but with valid MAC and padding. We then modify it to create two malformed records: One where we delete the first MAC byte, and one where we delete the last MAC byte. We then add another padding byte in both messages. These malformed records could also trigger vulnerabilities caused by incomplete MAC validations and are indirectly motivated by [28].

Missing MAC. Motivated by [37], we created two malformed records which only contain padding and do not con-

[2] The above behavior may sound contrived, but similar behaviors have been found in the wild, see e.g. [29, 28, 37].

Nr.	MAC			Padding		
	Len	Pos	Modification	Len	Pos	Modification
1	20	20	⊕ 0x01	56	–	–
2	20	11	⊕ 0x08	56	–	–
3	20	1	⊕ 0x80	56	–	–
4	19	1	DEL	56	–	–
5	19	20	DEL	56	–	–
6	0	–	–	80	ALL	0x4F
7	0	–	–	80	ALL	0xFF
8	20	–	–	60	1	⊕ 0x80
9	20	–	–	60	31	⊕ 0x08
10	20	–	–	60	60	⊕ 0x01
11	20	1	⊕ 0x80	60	–	–
12	20	9	⊕ 0x08	60	–	–
13	20	16	⊕ 0x01	60	–	–
14	20	1	⊕ 0x01	60	1	⊕ 0x80
15	20	1	⊕ 0x01	60	31	⊕ 0x08
16	20	1	⊕ 0x01	60	60	⊕ 0x01
17	20	–	–	6	1	⊕ 0x80
18	20	–	–	6	3	⊕ 0x08
19	20	–	–	6	6	⊕ 0x01
20	20	1	⊕ 0x80	6	–	–
21	20	9	⊕ 0x08	6	–	–
22	20	16	⊕ 0x01	6	–	–
23	20	1	⊕ 0x01	6	1	⊕ 0x80
24	20	1	⊕ 0x01	6	3	⊕ 0x08
25	20	1	⊕ 0x01	6	6	⊕ 0x01

Table 1: A summary of our malformed records, as constructed for `TLS_RSA_WITH_AES_128_CBC_SHA`. The columns indicate length, position, and modification for MAC and padding bytes, respectively. ⊕ denotes XOR'ing the listed value in the listed position. DEL denotes deleting one byte in the listed position.

tain a MAC at all: One where we supply exactly 80 bytes of valid padding (`0x4F`), and one where we supply 80 bytes of incomplete padding of value `0xFF`. The latter is not only missing the MAC but also contains invalid padding since if the value of the last byte is `0xFF`, there should be 256 padding bytes.

Combining valid and invalid MAC and padding. The last group of malformed records contains messages with combinations of valid and invalid MAC and padding of three types: valid MAC and invalid padding, invalid MAC and invalid padding, and invalid MAC and valid padding. For each of these three types, we create three sub-types, depending on which bit positions we flip; we flip either the most significant, middle, or least significant bit in the first, middle, or 16th byte, respectively. For each of these nine sub-types, we create one version which contains application data, and one without. The length of the application data is chosen such that the padding bytes are contained within one plaintext block, while the malformed records without application data contain more than one block of padding. This aims to detect implementations which check only the last block of padding bytes.

4.1.2 Combining Malformed Records with Protocol Versions and Cipher Suites

We use each malformed record with several TLS protocol versions and cipher suites. As previously stated, we use the term *test vector* to refer to the combination of a malformed record, protocol version, and cipher suite. As we later show, testing each malformed record with different protocol versions and cipher suites is necessary; some vulnerabilities are only triggered with such specific combinations. At first glance this is surprising, but this actually follows the findings of [9]. We conjecture that implementations may use completely different code stacks depending on the negotiated version and cipher suite, and some vulnerabilities are only present in a subset of those code stacks.

4.2 Empirical Test Vector Reduction

Depending on the configuration of the server, the above set of test vectors is quite large. Assuming a server supporting TLS 1.0 and TLS 1.1 with 10 CBC cipher suites, there would be $10 \cdot 2 \cdot 25 = 500$ test vectors. Note that every test vector requires establishing a new TLS connection and performing an expensive handshake. This large number of test vectors would not allow us to perform large-scale scans. On the other hand, removing test vectors could lead to false negatives and missing vulnerabilities. To reduce the number of test vectors without lowering the detection rate, we propose an empirical test vector reduction approach. We sample 50,000 random hosts which respond on port 443. We then perform a full scan on these hosts with the aforementioned 25 malformed records and all supported cipher suites and TLS version combinations. We can then analyze our test vector combinations and create the smallest set of test vectors detecting all padding oracle vulnerabilities. These empirical steps ensure that 1) with high probability we do not miss vulnerabilities, and 2) we can use the reduced set for large-scale analyses.

4.3 Clustering Vulnerabilities

Once we reduce the number of test vectors we can perform our full scan. For this purpose, we use one of the *Internet top lists* which typically contain a good mixture of up-to-date server implementations. Among Internet top lists, the Alexa Top 1 Million dataset contains the most significant number of hosts responding to TLS connections (about 75%) and is recommended for TLS scans [36].

After performing the TLS scan with a reduced vector set, we create a list of vulnerable hosts. We re-scan these hosts with our full test vector list. For every host, we store its *response map*. The response map describes the complete host behavior when responding to our test vectors. The response map consists of *cipher suite fingerprints*. A cipher suite fingerprint describes the server response behavior for a specific cipher suite and TLS version.

One of our major goals is to notify vulnerable vendors. For this purpose, it is necessary to group vulnerable hosts using the resulting response maps and contact their administrators to find out the vulnerable implementation version. Böck et al. performed this step manually and were able to approach the most important vendors [9]. However, such an approach is laborious and error-prone. We aim to group vulnerable implementations automatically.

Although grouping vulnerable hosts appears to be easy given all response maps, response maps differ even if they use the same vulnerable implementation version. TLS servers running identical implementations can use different configurations, enabling different cipher suites and TLS versions. For example, server A may be vulnerable to a padding oracle attack and has only one TLS cipher suite enabled: `TLS_RSA_WITH_AES_128_CBC_SHA256`. Server B is vulnerable using the same cipher suite fingerprint. However, server B is configured to use additional cipher suites as well which are not vulnerable to the attack. Are these two servers using the same implementation or just a similar one? To estimate this, we devised a novel approach based on a two-dimensional force-directed graph drawing algorithm [21]. These algorithms embed a network of nodes on a plane that allows for spatially interpreting the network. They do so by creating a two-dimensional graph which contains as few crossing edges as possible. In our approach we use the ForceAtlas2 algorithm [21]: *ForceAtlas2 simulates a physical system in order to spatialize a network. Nodes repulse each other like charged particles, while edges attract their nodes, like springs. These forces create a movement that converges to a balanced state. This final configuration is expected to help the interpretation of the data [21].*

We represent the scanning results as a graph as follows: Each node in the graph represents a host. Each pair of hosts is connected by an edge if their response maps do not include different cipher suite fingerprints for the same cipher suite.

This approach works well on our dataset, and servers exhibiting similar vulnerabilities are grouped closely. We augment the graph by coloring nodes according to their degree (i.e., their number of edges). The resulting visualization indeed allows identifying similar implementations. We show the concrete results in Section 8.

5 Large Scale TLS Scanning

We developed our padding oracle test vectors with TLS-Attacker [37], a framework for systematic analyses of TLS implementations. TLS-Attacker supports creating malicious TLS workflows and message malformities. TLS-Attacker has already been used for detecting padding oracle attacks, but only against specific implementations in lab conditions, not at scale. Our approach of creating an optimized set of test vectors was not previously included in this framework.

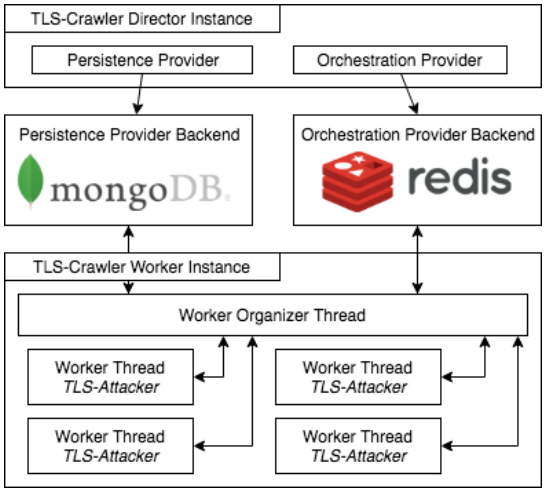

Figure 2: Our TLS scanning infrastructure is based on well-established components for data persistence and on TLS-Attacker for performing TLS evaluations.

5.1 TLS-Crawler

In order to scan a large number of hosts, we developed a framework which scans multiple servers in parallel and writes the results to a database. This allows us to parallelize the scan by using multiple machines. The database provides a querying interface for the scan data, which allows for easier analysis of the large result datasets. We call our framework TLS-Crawler.

TLS-Crawler is split into a director instance and potentially multiple worker instances. The worker instances perform the actual TLS host scans. Each worker instance implements a thread pool which distributes scanning work across available threads. The instance then bundles the results and coordinates parallelized database access. A director instance coordinates the worker instances. The director instance contains an orchestration provider responsible for the coordination and distribution of scanning tasks across workers. The results are persisted in a database using a persistence provider. We use MongoDB[3] as the persistence provider, and orchestrate instances via a Redis queue.[4] Figure 2 visualizes the TLS-Crawler architecture.

5.2 Performing the TLS Scans

Before scanning each host with test vectors, we perform a brief scan in order to learn the CBC cipher suites and TLS protocol versions supported by the host. We excluded export and anonymous cipher suites from these tests since they are already trivially broken by a MitM attacker. We then perform our scan using our set of test vectors for each CBC cipher

[3] `https://www.mongodb.com`
[4] `https://redis.io`

suite and its supported protocol version.

Previous large-scale TLS scans have mostly focused on vulnerabilities in the TLS handshake [9, 2], certificates [19], or vulnerabilities which could be triggered before the TLS handshake succeeds [17]. These previous scans only require performing a successful handshake once, usually with a commonly supported cipher suite. In contrast, in order to test for padding oracle vulnerabilities, it is necessary to perform a full TLS handshake for each tested cipher suite. This is complicated by TLS implementations exhibiting intolerances [7] which might prevent a server from completing the TLS handshake, or even responding to the initial `ClientHello` message. We tried to minimize the effect of these intolerances on our scans, but 20% of servers exhibited enough intolerances that we could not effectively scan them.

Even completing a TLS handshake does not guarantee we can effectively scan a host. For example, in some tests, the target hosts temporarily stopped responding for a few seconds. This is likely because the servers crashed or blocked our requests as part of a Denial-of-Service defense. In order to avoid false negatives from such scans, we scan multiple hosts in parallel (up to 2000) such that no host is overloaded by our requests. Additionally, we wait at least 10 seconds between scanning a host with two cipher suite/version pairs, further limiting the load on scanned hosts.

When performing these scans it is critical to select an appropriate timeout. If the timeout is too low, we might miss responses due to high server load. Conversely, a high timeout value would decrease the scanning performance. Setting a high timeout value also means we no longer distinguish between a server immediately closing the connection, and requiring a noticeable time to recover and close the connection. Additionally, the server's answers may span multiple TCP packets, so there is no simple way to ascertain the scanner has received the server's answer in full at any point in time. (Some responses do not include a TCP RST or FIN packet.) We empirically determined that a timeout of one second works well in practice, and mostly guarantees that the server did have enough time to process our record and respond. However, even when using this timeout value, we found servers that responded non-deterministically due to high load or various bugs.

To work around non-deterministic responses, we rescanned each suspected vulnerability in order to avoid false positives. We only consider a server as vulnerable if it responds identically in three separate scans to each of our test vectors.

6 Evaluation

For the scans, we used a machine with 2 Xeon E5-2683v5 CPUs (with a total of 64 cores) and 48 GB of RAM. The scan used an average of 5Mbit/s of upstream data and 15Mbit/s of downstream data.

6.1 Pre-Scanning with All Malformed Records

We performed a preliminary scan of 50,000 random TLS hosts, aimed at reducing the set of malformed records. The scan took place in October 2018 and required three days. The results confirmed that the choice of key exchange algorithm and protocol version indeed affects whether a given host exhibits CBC padding oracle vulnerabilities. We then reduced the set of malformed records. To do this, we first identified all vulnerable hosts, i.e. hosts that would be identified when scanning with the full set of malformed records. We then examined subsets of malformed records of increasing sizes, and for each subset, examined the number of hosts that would be identified when scanning only with this subset of malformed records. This process was stopped when a subset of four malformed records identified all vulnerable hosts. That is, all hosts that would be identified when scanning with the full set of malformed records, would also be identified when scanning with the reduced set of malformed records. This reduced set includes the following malformed records (all of these records are 80 bytes in length):

1. A record with missing MAC and correct padding (of value `0x4F`).

2. A record with missing MAC and incorrect padding (of value `0xFF`).

3. An empty record with no application data, with invalid padding and valid MAC. The highest bit in the first padding byte is flipped.

4. An empty record with no application data, with valid padding and invalid MAC. The lowest bit in the first MAC byte is flipped.

Please note that we still test every TLS host with all of its supported cipher suites and TLS protocol versions.

Is the malformed record set reduction lossy? The reduced malformed record set detects all vulnerabilities detected by the larger, original malformed record set, *on the sample data of the preliminary scan*. It is natural to ask whether there are hosts that are vulnerable to a malformed record from the original set, but not to a malformed record from the reduced set. There are obviously no such hosts in the sample data, but there could be such hosts outside of the sample. If there is a large number of such hosts on the Internet, then the malformed record reduction process would be lossy, i.e. by using fewer malformed records, we detect fewer vulnerabilities in the full scan. As we now explain, this source of scanning inaccuracy is likely small enough to not materially affect our results. Put another way, the reduced set of malformed records likely detects most vulnerabilities

triggered by the full set of malformed records, not just on the sample data.

Indeed, let p denote the percentage of hosts, out of all TLS-speaking hosts, that are vulnerable to one malformed record from the full set of malformed records, but not to any malformed records from the reduced set. I.e., p describes the percentage of hosts that the reduction misses; we will now show it is rather small. In the random sample of $N = 50000$ hosts used for the preliminary scan, we did not encounter any such hosts. In order to compute the 99% confidence interval, we require $(1-p)^N = 0.01$. Solving for p, we obtain $p = 0.0092\%$. We therefore determine with 99% confidence that there are at most 0.0092% additional vulnerable hosts that our scans miss due to the malformed record reduction.

We provide an intuitive explanation of the above, for the reader's convenience. As per the above calculation, we estimate the percentage of vulnerable hosts on the Internet that would be missed because we scan with the reduced set of malformed records is 0.0092%. Censys [16] estimates there are about 42.4 million hosts which serve TLS on port 443 as of February 2019. Therefore, our estimate is that the reduction misses at most $42400000 \cdot 0.0092\% = 3900$ hosts. Intuitively, the term "99% confidence interval" means there is roughly a 1% chance that this estimate is wrong, i.e. that there are more than 3900 such hosts on the Internet.

6.2 Alexa Top Million Scan

We used the reduced set of malformed records to scan the Alexa Top Million websites. Among the top lists, Alexa Top 1 Million provides the highest percentage of hosts supporting TLS [36] and is thus suitable for large-scale TLS scans. The list likely includes most high-profile TLS implementations.

The scan required approximately 72 hours. Of the initial one million hosts, 785,295 responded on port 443. We were able to perform TLS handshakes with CBC cipher suites with 627,493 hosts. We excluded all other hosts from the evaluation. We discovered a total of 18,257 Alexa Top Million hosts (1.83%) which are vulnerable to padding oracle attacks.

The data supports our conjecture that implementations may be vulnerable on a cipher suite with one protocol version, but not vulnerable on the same cipher suite with a different protocol version. A total of 649 servers were only vulnerable in either TLS 1.0 or TLS 1.1/1.2 although the vulnerable cipher suite was supported in the other version. Similarly, in some cases, the negotiated key exchange algorithm affects whether implementations exhibit a CBC vulnerability. 601 hosts were vulnerable on one cipher suite, but not on another cipher suite with a different key exchange algorithm but the same symmetric cipher and HMAC function. A total of 3,247 hosts were vulnerable on all CBC cipher suites they supported.

After identifying vulnerable hosts, we rescanned them with the full set of test vectors to get their full response maps. As noted above, to label a host as vulnerable we require the response maps to be consistent across three different scans.

6.3 Results of Our Clustering Approach

Analyzing each vulnerable host manually is infeasible. We therefore clustered the vulnerable hosts, such that hosts exhibiting the same cipher suite fingerprints are clustered together. This minimizes the manual work required to identify the vendor (or vendors) responsible for each vulnerable behavior. We reiterate that this clustering is not trivial, as explained in Section 4.3.

We identified 93 different cipher suite fingerprints. Table 2 summarizes the 40 most common cipher suite fingerprints. Using the first row as an example, 7297 hosts responded with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` TLS alerts and timed out the connection for malformed records 11 and 12 (🕓). For all other malformed records these hosts closed the TCP connection (🚫) after sending the same TLS alerts.

We also identified four groups exhibiting behavior similar to the CVE-2016-2107 vulnerability in OpenSSL [37] (cipher suite fingerprints #41, #75, #14, and #54 in Table 2). They respond to malformed records 6 and 7 (see Table 1) with a `RECORD_OVERFLOW` TLS alert. To all other malformed records they respond with `BAD_RECORD_MAC`. These are likely unpatched OpenSSL implementations, or security appliances running older OpenSSL versions.

For vulnerable cipher suites on the same host, cipher suite fingerprints are largely consistent. Of hosts exhibiting at least one vulnerable cipher suite, 99.6% have an identical cipher suite fingerprint on all vulnerable cipher suites. We removed the remaining 0.4% of hosts to make clustering easier. However, hosts sharing the same cipher suite fingerprint on vulnerable cipher suites don't necessarily share the same implementation. As an example, consider two hosts, A and B, with two cipher suites supported by both hosts, 1 and 2. A is vulnerable on cipher suite 1 with cipher suite fingerprint X, but is not vulnerable on cipher suite 2. B is not vulnerable on cipher suite 1, but is vulnerable on cipher suite 2 with the same cipher suite fingerprint X. This difference indicates the hosts don't share the same implementation, as we would expect the shared implementation to have a consistent set of vulnerable cipher suites. (We concede that it is possible the hosts exhibit different behavior because of different configuration flags despite sharing the same implementation, but we consider this unlikely).

We denote the above situation (in its general form) as "contradictory response maps"; two hosts exhibiting the same cipher suite fingerprint on vulnerable cipher suites, but where there exists a cipher suite supported by both hosts such that one host is vulnerable on that cipher suite and the other host is not. We refer to the complement situation as "compatible response maps".

Nr.	Cipher suite fingerprint									Strength		Count
	1,2,3,20,21	4,5	6	7	8,9	10,16,19,22–25	11,12	13,14,15	17,18	R1	R2	
15	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*S*	7297
41	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{22} 🔒 ⊘	F_{22} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	4387
84	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} 🔒 ϟ	👁	*P*	2313
75	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{22}W 🔒 ⊘	F_{22}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁‍🗨	*W*	940
21	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	👁‍🗨	*W*	687
23	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20} 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*W*	458
68	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	⏲	👁	*P*	248
0	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	A ⊘	👁	*P*	194
79	⊘	⊘	⊘	⊘	⊘	⊘	F_{20}W 🔒 ⏲	⊘	⊘	👁	*W*	151
10	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	👁‍🗨	*W*	98
85	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	A ⊘	👁	*P*	83
2	ϟ	ϟ	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	76
61	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	ϟ	ϟ	ϟ	F_{20} 🔒 ⊘	ϟ	ϟ	👁‍🗨	*S*	54
6	ϟ	ϟ	ϟ	ϟ	F_{40} 🔒 ⊘	ϟ	ϟ	ϟ	F_{40} 🔒 ⊘	👁	*P*	52
62	⏲	⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	47
33	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	⏲	👁	*P*	43
31	⊘	⊘	⊘	⊘	⏲	⏲	⊘	⏲	⏲	👁	*P*	36
76	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	👁	*P*	34
77	F_{20}W 🔒 ⊘	F_{50}W 🔒 ⊘	F_{50}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁‍🗨	*S*	28
14	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{22} 🔒 ϟ	F_{22} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	👁‍🗨	*W*	24
24	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ⏲	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	👁	*W*	21
38	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	ϟ	ϟ	ϟ	F_{80} 🔒 ⊘	ϟ	ϟ	👁	*S*	19
4	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} 🔒 ⏲	👁	*P*	15
54	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{22} 🔒 ⏲	F_{22} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁‍🗨	*W*	12
74	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*W*	9
7	ϟ	ϟ	ϟ	ϟ	⊘	ϟ	ϟ	ϟ	⊘	👁	*P*	8
37	F_{20} 🔒 ⊘	F_{50} 🔒 ⊘	F_{50} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	7
51	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	⊘	👁	*W*	7
59	A ⊘	ϟ	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	👁	*S*	7
66	ϟ	ϟ	⏲	ϟ	⏲	ϟ	⏲	⏲	ϟ	👁	*W*	7
70	A ⊘	A ⊘	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	7
11	F_{20} 🔒 ⊘	⏲	⏲	⏲	⏲	⏲	F_{20} 🔒 ⊘	⏲	⏲	👁	*P*	5
42	F_{20} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{20} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	👁‍🗨	*S*	5
89	ϟ	ϟ	⏲	⊘	⊘	⊘	⏲	⊘	⊘	👁	*S*	5
3	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁	*S*	4
26	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{10} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	4
28	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	AW 🔒 ⊘	👁	*P*	4
35	ϟ	ϟ	ϟ	ϟ	F_{20}W 🔒 ⊘	ϟ	ϟ	ϟ	F_{20}W 🔒 ⊘	👁	*P*	4
73	⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	⊘	⊘	⊘	⊘	⊘	⊘	👁	*W*	4
9	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*W*	3

Table 2: Analysis of the 40 most common cipher suite fingerprints, each consisting of responses to 25 malformed records. For ease of reading, we group together malformed records for which responses are identical within each cipher suite fingerprints. We use the following notation: `Application` message (A), Fatal Alert with error code k (F_k), Warning Alert (W), connection closed (⊘), TCP reset (ϟ), timeout (⏲). We use the following TLS Alert codes: `UNEXPECTED_MESSAGE` (10), `BAD_RECORD_MAC` (20), `DECRYPTION_FAILED_RESERVED` (21), `RECORD_OVERFLOW` (22), `DECOMPRESSION_FAILURE` (30), `HANDSHAKE_FAILURE` (40), `ILLEGAL_PARAMETER` (47), `DECODE_ERROR` (50), `DECRYPT_ERROR` (51), `INTERNAL_ERROR` (80). Alerts with code `CLOSE_NOTIFY` always used the `warning` level. 🔒 denotes an encrypted response. The oracle *strength* definition is provided in Section 7; observable differences are depicted with 👁, unobservable differences with 👁‍🗨. We use *W* and *S* for weak and strong padding oracles, respectively (a strong and observable oracle is exploitable). *P* represents behavior similar to POODLE (which is also exploitable if it is observable).

We then use a graph algorithm in order to further split host groups. For each group of hosts with an identical cipher suite fingerprint, we construct a graph where each node represents a host. We draw an edge between two hosts if and only if their response maps are compatible. We then embed the graph in a two-dimensional plane using the ForceAtlas2 algorithm, as implemented in the Gephi software.[5] The ForceAtlas2 algorithm clusters together nodes connected by an edge, so nodes with compatible response maps are clustered together. Identically configured servers which behave identically will be connected to the same nodes and will therefore have the same degree. Since these servers are connected to the same nodes, ForceAtlas2 will draw them close to one another. By coloring the nodes by their degree it becomes easy to manually spot similarly configured and identically behaving implementations in the graph. These subgroups can then be examined for candidates for manual analysis and responsible disclosure. [6]

Example for one vulnerability group. An example of this visualization is provided in Figure 3. The figure clearly shows two distinct sub-groups which do not share edges (meaning their response maps are contradictory and they likely do not share the same implementation). Hosts shown in green are vulnerable on `TLS_RSA_WITH_AES_128_CBC_SHA` and `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`, while servers shown in pink are only vulnerable to `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA` and not on `TLS_RSA_WITH_AES_128_CBC_SHA`. Interestingly the hosts in the middle of the graph (mostly in teal) do not support `TLS_RSA_WITH_AES_128_CBC_SHA` (they are vulnerable on `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`). They may share their implementation with either the green or pink group and therefore share edges with the members of both groups. Hosts in red are very similar to the pink group but do not share edges with the teal group. This means that either a third group exists, or the teal group actually belongs to the green group and the red group belongs to the pink group. Individual nodes are likely rare configurations of one of the implementations of the bigger groups. We performed a DNS lookup and determined both groups are operated by a Czech hosting company.

This approach allowed us to also contact other prominent websites in each group and ask what TLS implementation they use.

[5] `https://github.com/gephi/gephi`

[6] We note that further grouping by the server agent string could provide more insights into the different groups. However, it is also very likely that it would also falsify our results. In many cases, TLS is terminated in reverse proxies or firewalls, and the server agent string is generated on a different machine handling HTTP traffic.

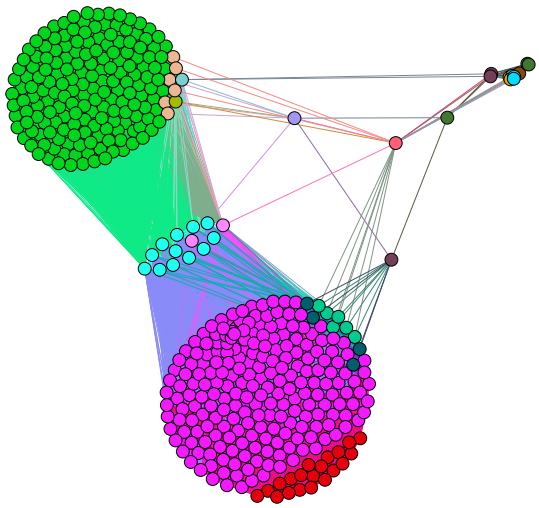

Figure 3: Visualisation of group #23 from Table 2.

Breakdown of response maps. Figure 4 visualizes the prevalence of the various cipher suite fingerprints. A few very common vulnerabilities account for the majority of vulnerable hosts. The newly-discovered vulnerabilities in Amazon/OpenSSL and Citrix account for slightly more than half of all vulnerable hosts. These are listed as #15 and #84 and described in more detail in Section 8.2. In addition, response maps #41 and #75, which likely stem from implementations based on unpatched OpenSSL versions, account for roughly a third of vulnerable hosts. Response map #23 is found in the above-mentioned Czech hosting company.

7 Realistically Exploitable Padding Oracles

Not all of the oracles we identified enable effective decryption attacks. The rest of this section explains exploitation in more detail.

The padding oracles we discovered are based on direct message side channels, i.e. on TLS implementations where two error states trigger different error responses from the TLS server. They may be exploitable in the BEAST attacker model, which relies on two assumptions: (a) the victim client visits a website under the attacker's control, which triggers HTTPS requests to the victim server, and (b) the attacker is a MitM and can observe the session and modify transmitted ciphertexts. In addition to those standard assumptions, an oracle is exploitable if it satisfies two additional requirements: *(R1) Observability* and *(R2) Perfect padding distinguishability*.

7.1 (R1) Observability

Unlike timing side channels, little attention has been paid to direct message side channels in the case of TLS, and common wisdom seems to assume they are *unobservable* to the

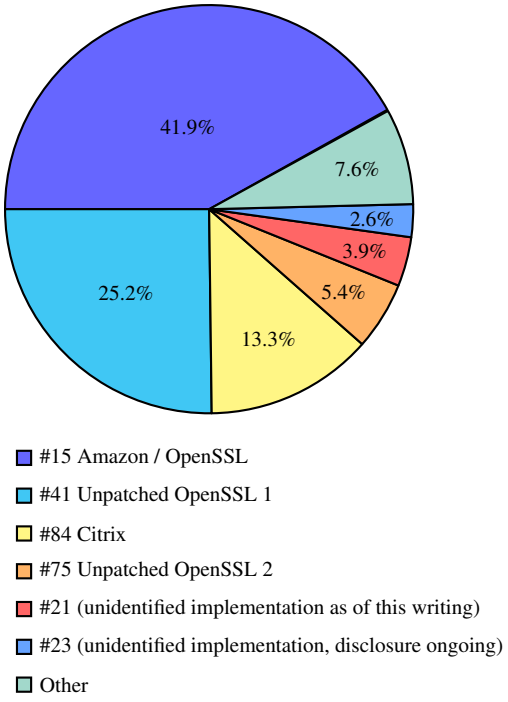

Figure 4: A visualization of the prevalence of cipher suite fingerprints. A few widely-prevalent vulnerabilities account for the majority of vulnerable hosts. Out of the above cipher suite fingerprints, #84 and #15 are exploitable. They are described in more detail in Section 8.2.

attacker. Indeed, this is true for implementations which send a single alert in all error cases and the behavior is identical except perhaps for the content of the alert message. Such behavior cannot be exploited by the attacker to create a side channel because the alert message is encrypted. However, we identified many cases where implementations do exhibit an observable difference in behavior. These observable differences can roughly be divided into two classes:

- **TCP layer.** We found implementations which leak information about the padding validity in the TCP layer. For example, in the case of Amazon, most test vectors with invalid padding caused the server to immediately close the TCP connection. However, specific, carefully crafted test vectors caused the server to abort the TLS session while keeping the TCP connection open.
- **Number of TLS records.** We observed TLS servers that responded with a different number of records based on the padding validity. While the attacker cannot decrypt these records, he is able to observe the total ciphertext length. For example, the servers from group 23 (see Table 2) responded with *one* TLS alert in the case of valid padding, while for invalid padding they responded with *two* TLS alerts.

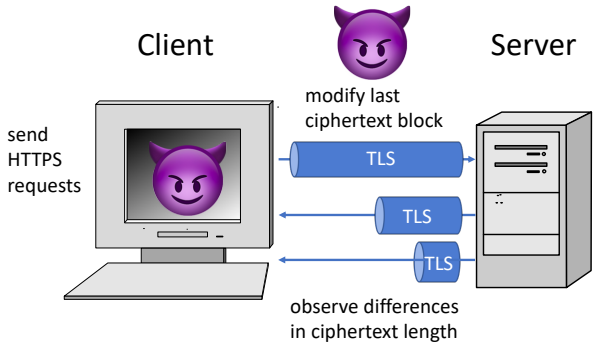

Figure 5: Exploiting observable error-based padding oracles in a BEAST scenario. Differences in total ciphertext length result from different numbers of TLS alerts being sent.

Consider an attacker $\mathscr{A}$ who can distinguish between the two cases of `valid_padding` and `invalid_padding` based on the validity of the last padding byte (see Figure 6). The attacker decrypts an HTTPS session cookie as follows:[7]

1. $\mathscr{A}$ lures the victim client to load a web page he controls. This web page contains JavaScript code which sends HTTPS requests to the victim server, with a URL of $\mathscr{A}$'s' choice.
2. $\mathscr{A}$ observes the first TLS handshake and determines if the negotiated cipher suite is vulnerable to padding oracle attacks. If not, he aborts.
3. If a vulnerable cipher suite is used, $\mathscr{A}$ instructs the client to send another HTTPS request, modifying the URL such that the first character of the session cookie is the last byte in cipher block c_i.
4. As a MitM, $\mathscr{A}$ intercepts the ciphertext $(c_1, ..., c_i, ..., c_n)$ and modifies it such that c_i becomes the last ciphertext block, for example by replacing c_n with c_i.
5. Decryption of this last block c_i is a pseudorandom transform, so the padding will likely be invalid, triggering an observable `invalid_padding` error event.
6. In about 1 out of 256 requests, the padding will randomly be valid. When the padding is valid, it is most likely to be one byte in length, as depicted in Figure 6. The preceding bytes will be parsed by the TLS server as MAC data, and will be invalid with overwhelming probability. In this case, $\mathscr{A}$ observes a `valid_padding` error event, and computes the first character of the HTTPS

[7] We present here a more general form of the attack, which is also applicable to POODLE-style oracles. This form requires 256 sessions on average in order to decrypt one plaintext byte [27]. For oracles which completely disregard the MAC, there is a faster form which requires 128 sessions on average to decrypt one plaintext byte.

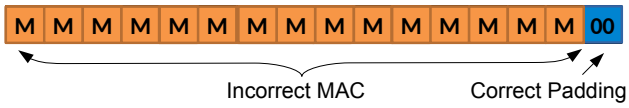

Figure 6: Our attacks rely on a vulnerable server that delivers different responses based on the validity of the last padding byte.

session cookie as $c_{n-1}[-1] \oplus c_{i-1}[-1]$, where the $[-1]$ operator denotes taking the last byte of a block.

7. $\mathscr{A}$ then prepares another HTTPS URL where the second character of the session cookie is shifted to the last byte of c_i, and starts again with step 3.

7.2 (R2) Perfect Padding Distinguishability

In the above example, we considered a simple oracle that allows for distinguishing between `valid_padding` and `invalid_padding` based on the validity of the last padding byte. However, even when providing different responses, implementations do not necessarily expose such simple oracles. For example, an older OpenSSL version responds with a different alert message only in the specific case of an empty record containing at least two full valid padding blocks [37]. We identified vulnerable implementations that only respond differently to ciphertexts containing several valid padding or MAC bytes. Such vulnerabilities are less likely to be exploitable since using the algorithm above, the attacker would need to perform far more than 256 oracle queries to decrypt each byte. The attacker may be able to overcome this limitation by inserting bytes of his choice directly after the cookie value. Due to the malleability property of CBC, it is only possible to insert one block of successive chosen data. Therefore, CBC allows for the creation of practical exploits if the number of chosen padding bytes is smaller than the block size.[8]

Therefore, in our impact estimation, we take a conservative approach. To consider a vulnerable implementation as exploitable, we require that it responds with `valid_padding` to ciphertexts with at most one block of valid padding. We call such oracles *strong* and refer to other oracles as *weak*. In addition to these two oracles, we consider oracles which do not correctly validate the complete CBC padding and only validate the MAC. We refer to such oracles as *POODLE oracles*. These oracles could also be exploited by applying attacks similar to POODLE.

Column *R2* in Table 2 identifies the oracle strength. For example, servers with the second most prevalent cipher suite fingerprint (#41) respond to malformed records #6 and #7 from Table 1 with a `RECORD_OVERFLOW`. In all other cases, the servers send the `BAD_RECORD_MAC` alerts. We consider this group to be weak since the attacker needs to send more than one block of valid padding to trigger the `RECORD_OVERFLOW` alert with a malformed record #6 or #7.

We consider servers with cipher suite fingerprint #2 to be strong oracles. The servers from this group respond with a TCP connection reset (ϟ) if they receive a malformed record with a valid padding (see malformed records #20 and #21). There are also several groups with behavior similar to POODLE. These groups ignore modifications in the MAC bytes and respond differently to malformed records #8, #9, #17, and #18.

7.3 Exploitability

We consider observable POODLE and observable strong oracles as exploitable. We consider all other oracles as non-exploitable. However, note that weak oracles may be exploitable using more advanced techniques. Our estimate of the number of exploitable hosts is, therefore, a conservative lower estimate.

Estimation of exploitable hosts. Our scan identified 18,257 hosts vulnerable to padding oracle attacks. Of those, 11,225 (61.4%) exhibit observable vulnerabilities that allow an attacker to distinguish between two malformed records. See also column *R1* in Table 2. At least 10,688 hosts provided strong or POODLE-styled oracles, which is 58% of vulnerable hosts. See also column *R2* in Table 2. In total, 10,501 hosts are practically exploitable, i.e. they meet both requirements.

Are CBC cipher suites negotiated? Most modern browsers support AEAD cipher suites. If a vulnerable server prefers AEAD cipher suites, they would likely be negotiated, and this precludes CBC attacks. 31,651 hosts or 4.03% only support RC4 or CBC cipher suites. Most modern browsers have disabled support for RC4 cipher suites due to [30], so modern browsers would likely negotiate CBC cipher suites with these hosts. Of those hosts, 1,400 were vulnerable to padding oracle attacks.

8 Findings

In this section we review our assumptions and present notable vulnerabilities we found in different implementations.

8.1 Do our Initial Assumptions Hold?

We performed our scans under the assumption that scanning with different cipher suites and protocol version is necessary

[8] Decrypting parts of the cookies with weak oracles or exploiting weak oracles could also be possible with extended techniques. We do not analyze the exploitability of these more complex oracles. Such an analysis would likely need to be done manually for each oracle and would need to consider specific browser behaviors.

in order to detect vulnerable hosts. As explained below, our findings confirm this assumption.

Is scanning with different protocol versions necessary? Böck et al. found that some servers exhibit RSA padding oracle vulnerabilities only on some of the protocol versions they support [9]. As noted in Section 3.5, we suspected the same holds for CBC padding vulnerabilities. Our findings confirm this assumption: We identified at least 744 hosts that support the same cipher suite in both TLS 1.0 and 1.2, but are vulnerable when using that cipher suite only in one of those versions. In some cases the vulnerable protocol version is the newer version, and in other cases, the older one. As an example of the former case, vine.co was vulnerable using TLS 1.2 with the `TLS_RSA_WITH_3DES_EDE_CBC_SHA` cipher suite, but was not vulnerable when using the same cipher suite in TLS 1.0.

Surprisingly, when only one protocol version is vulnerable with the same cipher suite, there are more cases where the newer version is vulnerable. Out of those 744 hosts, 120 hosts are vulnerable in TLS 1.0 but not in TLS 1.2, and 624 are vulnerable in TLS 1.2 but not in TLS 1.0.

Is scanning with different cipher suites necessary? Böck et al. also found that scanning with different cipher suites is necessary to detect as many vulnerabilities as possible [9]. In the above work, this finding held even when scanning with cipher suites using different symmetric ciphers, while the vulnerability was in the (theoretically unrelated) RSA implementation.

We find similar behavior in our results. We identified at least 601 hosts with two cipher suites, one vulnerable and one secure, where the only difference between the two cipher suites is the *key exchange algorithm*. This finding is unintuitive, as one would expect an implementation to be uniformly vulnerable or secure on all cipher suites with the same symmetric cipher. To give one example, one website is secure when using `TLS_RSA_WITH_AES_256_CBC_SHA256` with TLS 1.2, but is vulnerable when using `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`, also with TLS 1.2.

Rationale behind the server behaviors. Both behaviors may seem unintuitive but are actually expected. Many implementations take completely different code paths depending on the negotiated cipher suite or protocol version. These code paths may, for example, rely on hardware acceleration or use an optimized assembly implementation when possible. It is therefore likely (and, as we see, common) to find implementations that exhibit vulnerabilities only in some of the supported cipher suites and protocol versions, even when the same symmetric cipher is used.

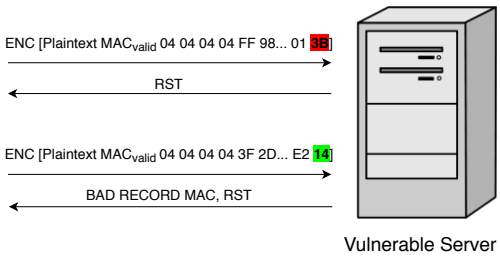

Figure 7: Behavior of Citrix implementations with cipher suite fingerprint #84.

8.2 Notable Vulnerabilities

In our scans we identified multiple devices from Cisco, two different IBM servers, and multiple devices from Sonicwall and Oracle. In the following, we describe specific vulnerabilities we identified and responsibly disclosed in Citrix, OpenSSL, and IBM servers.

Our disclosure is still an ongoing process. Our recent findings and the current state of countermeasures implemented by affected vendors are summarized on `https://github.com/RUB-NDS/TLS-Padding-Oracles`.

Amazon/OpenSSL. With the help of the Amazon security team, we identified a vulnerability (cipher suite fingerprint #15) which was mostly found on Amazon servers and Amazon Web Services (AWS). Hosts affected by this vulnerability immediately respond to most records with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` alerts, and then close the connection. However, if the hosts encounter a zero-length record with valid padding and a MAC present, they do not immediately close the TCP connection, regardless of the validity of the MAC. Instead, they keep the connection alive for more than 4 seconds after sending the `CLOSE_NOTIFY` alert. This difference in behavior is easily observable over the network. Note that the MAC value does not need to be correct for triggering this timeout, it is sufficient to create valid padding which causes the decrypted data to be of zero length. Therefore, we classify this as a strong oracle which is also exploitable.

Further investigations revealed that the Amazon servers were running an implementation which uses the OpenSSL 1.0.2 API. In some cases, the function calls to the API return different error codes depending on whether a MAC or padding error occurred. The Amazon application then takes different code paths based on these error codes, and the different paths result in an observable difference in the TCP layer. The vulnerable behavior only occurs when AES-NI is not used.

We had in fact previously tested the vulnerable OpenSSL code manually, in lab settings, but had not identified this vulnerability. This is because the vulnerability only manifests

under a combination of specific conditions: subtle interactions between OpenSSL and external code, and only when AES-NI is not used, which is rare nowadays. We view this as an illustrative example of the usefulness of large-scale scans in detecting vulnerabilities that lab tests may sometimes miss.

We suspect this OpenSSL behavior underlies a number of similar vulnerabilities we identified, not only vulnerability #15. Therefore, we hope that once OpenSSL releases a patch, other vulnerabilities will be fixed as a result. The issue was assigned CVE-2019-1559.

The IBM vulnerabilities. We found multiple vulnerabilities in servers hosted by IBM. One of the vulnerabilities is described by cipher suite fingerprint #77 in Table 2. Affected servers respond with a `BAD_RECORD_MAC` alert if either the MAC or the padding is incorrect. If the padding is correct and the MAC is incomplete or not present, the server responds with a `DECODE_ERROR` alert. The latter behavior occurs even if the records are too short to contain a MAC, as long as the record contains at least two blocks of ciphertext, independently of the used MAC algorithm. An attacker can send only two blocks with an IV, which guarantees there is not enough room for a MAC. This provides the attacker with a classic CBC padding oracle. We therefore consider this a strong oracle. Since the alerts are encrypted, we classify this vulnerability as unobservable, and the oracle is therefore not exploitable.

The IBM security team decided to disable CBC cipher suites on the affected servers and to only support AES-GCM.

Citrix. The described vulnerability is identified by cipher suite fingerprint #84 in Table 2. The vulnerable implementation first checks the last padding byte and then verifies the MAC. If the MAC is invalid, the server closes the connection. This is done with either a connection timeout or an `RST`, depending on the validity of the remaining padding bytes. However, if the MAC is valid, the server checks if all other remaining padding bytes are correct. If they are not, the server responds with a `BAD_RECORD_MAC` and an `RST` (if they are valid, the record is well-formed and is accepted). We visualize this behavior in Figure 7. This behavior can be exploited with an attack similar to POODLE. Since the oracle is also observable, we consider this group as exploitable. We first detected this vulnerability in Amazon Web Services. In cooperation with the Amazon security team, we determined that Citrix Application Delivery Controller (ADC) and NetScaler Gateway are responsible for this behavior. The vulnerability was assigned CVE-2019-6485.

9 Related Work

We now highlight past work that focused on large-scale scans for vulnerabilities on the modern Internet. For a survey of related work on padding oracle attacks, we refer the reader to Section 3. ZMap [18] is a network scanner capable of reaching high scanning speeds. Durumeric et al. [17] used ZMap to scan the IPv4 address space to quantify the impact of the Heartbleed vulnerability [32]. Heninger et al. [19] scanned TLS and SSH for weak keys generated using insufficient entropy. Adrian et al. [2] introduced the Logjam vulnerability and used Internet-wide scanning to quantify its effects, depending on attacker computational resources. Aviram et al. [5] introduced the DROWN vulnerability and similarly used Internet-wide scanning to quantify its effects. Böck et al. [9] performed large-scale scans for Bleichenbacher's vulnerability, while also observing side channels such as changes in the TCP connection state, as we do here. Valenta et al. [38] scanned for known vulnerabilities in elliptic curve implementations, searching for a combination that could enable a powerful attack named CurveSwap.

10 Conclusions and Future Work

This work demonstrates that padding oracle vulnerabilities still exist on the modern Internet and will likely continue to threaten users' security. These vulnerabilities are often hard to detect: they may rely on subtle side channels or require specifically-crafted inputs in order to trigger.

In the past, major new TLS attacks had positive effects on the ecosystem. For example, the work by Adrian et al. [2] resulted in an "enforcement" effort, where major browsers changed their behavior and refused to connect to servers with weak DH parameters. It is an interesting open question how the security community can better help server operators detect and remediate more subtle kind of vulnerabilities (CBC oracles in particular, and other classes of vulnerabilities in general).

One solution in the context of CBC oracles would be to disallow CBC cipher suites altogether. Recently, major browser vendors have declared their intention to remove support for the old 1.0 and 1.1 TLS versions. This forces many server operators to upgrade their implementations or change configuration. Indeed, a case could be made that browser vendors can also remove support for CBC cipher suites, forcing again server operators to upgrade. These changes are not without their costs; they usually require notice of months in advance, may require coordination between browser vendors, and obviously, create additional work for server operators.

Our results again confirm that large-scale scans make it feasible to uncover a large variety of security vulnerabilities, previously not detected by lab testing. We believe that our approach is of general interest when performing large-scale scans, not only in the context of TLS. One open question is how to identify vulnerable implementation versions and their vendors. In the SSH and IPsec protocols, these data are typically transmitted as message fields in the protocol.

Transmitting such data in TLS would make disclosure easier, but on the other hand would lead to privacy issues and easier fingerprinting.

Acknowledgments

We would like to thank Dennis Felsch who assisted us with our hardware and network infrastructure, and our anonymous reviewers for many insightful comments. Additionally, we would like to thank the Amazon, Citrix and OpenSSL teams for their professional responses and help with disclosure.

Nimrod Aviram was supported by a scholarship from The Israeli Ministry of Science and Technology, a scholarship from The Check Point Institute for Information Security, and a scholarship from The Yitzhak and Chaya Weinstein Research Institute for Signal Processing. Juraj Somorovsky was supported by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1). Robert Merget was supported by the German Federal Ministry for Economic Affairs and Energy with initiative "IT-Sicherheit in der Wirtschaft", through the SIWECOS project.

References

[1] Openssl security advisory. CVE-2018-0733.

[2] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTA, L., ET AL. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 5–17.

[3] ALBRECHT, M. R., AND PATERSON, K. G. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), pp. 622–643.

[4] ALFARDAN, N. J., AND PATERSON, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`.

[5] AVIRAM, N., SCHINZEL, S., SOMOROVSKY, J., HENINGER, N., DANKEL, M., STEUBE, J., VALENTA, L., ADRIAN, D., HALDERMAN, J. A., DUKHOVNI, V., KÄSPER, E., COHNEY, S., ENGELS, S., PAAR, C., AND SHAVITT, Y. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), pp. 689–706.

[6] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., STEEL, G., AND TSAY, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO* (2012), Canetti and R. Safavi-Naini, Eds.

[7] BENJAMIN, D. Tls ecosystem woes, Jan. 2018. Real World Crypto Symposium.

[8] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[9] BÖCK, H., SOMOROVSKY, J., AND YOUNG, C. Return of bleichenbacher's oracle threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 817–849.

[10] BÖCK, H. A little POODLE left in GnuTLS (old versions), Nov. 2015. `https://blog.hboeck.de/archives/877-A-little-POODLE-left-in-GnuTLS-old-versions.html`.

[11] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology - CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Aug. 2003.

[12] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507.

[13] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507.

[14] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[15] DUONG, T., AND RIZZO, J. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *IEEE Symposium on Security and Privacy* (2011).

[16] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security* (Oct. 2015).

[17] DURUMERIC, Z., LI, F., KASTEN, J., AMANN, J., BEEKMAN, J., PAYER, M., WEAVER, N., ADRIAN, D., PAXSON, V., BAILEY, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference* (2014), ACM, pp. 475–488.

[18] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Zmap: Fast internet-wide scanning and its security applications.

[19] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices.

[20] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[21] JACOMY, M., VENTURINI, T., HEYMANN, S., AND BASTIAN, M. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE 9*, 6 (06 2014), 1–12.

[22] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings* (2012), S. Foresti and M. Yung, Eds., LNCS, Springer.

[23] JAGER, T., AND SOMOROVSKY, J. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)* (Oct. 2011).

[24] LABS, A. W. S. s2n: An implementation of the tls/ssl protocols.

[25] LANGLEY, A. The POODLE bites again, Nov. 2014. `https://www.imperialviolet.org/2014/12/08/poodleagain.html`.

[26] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium, San Diego, USA* (August 2014).

[27] MÖLLER, B., DUONG, T., AND KOTOWICZ, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[28] PETTERSSEN, Y. The POODLE has friends.

[29] PETTERSSEN, Y. There are more POODLEs in the forest.

[30] POPOV, A. Prohibiting RC4 Cipher Suites. RFC 7465 (Proposed Standard), Feb. 2015.

[31] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[32] RIKU, ANTTI, MATTI, AND MEHTA. Heartbleed, cve-2014-0160, 2015. `http://heartbleed.com/`.

[33] RIZZO, J., AND DUONG, T. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–8.

[34] RIZZO, J., AND DUONG, T. Here Come The XOR Ninjas, May 2011.

[35] RONEN, E., PATERSON, K. G., AND SHAMIR, A. Pseudo constant time implementations of tls are only pseudo secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 1397–1414.

[36] SCHEITLE, Q., HOHLFELD, O., GAMBA, J., JELTEN, J., ZIMMERMANN, T., STROWES, S. D., AND VALLINA-RODRIGUEZ, N. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *Internet Measurement Conference (IMC'18), IMC'18 Community Contribution Award* (Boston, USA, Nov. 2018), ACM, pp. 478–493.

[37] SOMOROVSKY, J. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1492–1504.

[38] VALENTA, L., SULLIVAN, N., SANSO, A., AND HENINGER, N. In search of curveswap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), IEEE, pp. 384–398.

[39] VAUDENAY, S. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology — EUROCRYPT 2002*, vol. 2332 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Apr. 2002.