Padding Oracle Attacks

Rafael Fedler
Supervisor: Benjamin Hof, M.Sc.
Seminar Innovative Internet Technologies and Mobile Communications, SS 2013
Chair for Network Architectures and Services
Department of Computer Science, Technische Universität München
Email: fedler@in.tum.de

ABSTRACT

For the security of communication channels in today's networks and encryption of messages therein, applications and their users rely on cryptographic protocols. These are supposed to provide confidentiality and integrity of message contents. They are relied upon by online shopping, banking, communication, scientific applications, and many others. Design errors in standard definition documents or in the implementation of widespread libraries, however, allow for the violation of these objectives by adversaries. Specifically, *padding oracle attacks* render the partial or complete recovery of the underlying plaintext of encrypted messages possible. Such attacks also affect the most common *modus operandi* of most modern cryptographic protocols, the cipher block chaining (CBC) mode. Thus, given a corresponding design or implementation error, these attacks can affect almost all online communication channels secured by such protocols.

In this paper, we give an insight into the theoretical aspects of padding oracle attacks. We will outline all necessary background and detail prerequisites for a successful attack. An overview of resulting practical implementations in real-world applications such as Datagram TLS, among others, will also be provided. It is our intent to introduce the reader to common design flaws of cryptographic constructs in protocols that make them prone to padding oracle attacks, so that readers are able to avoid such mistakes and to assess cryptographic constructs in this regard.

Keywords

Padding oracle attacks, cryptographic protocols, cipher block chaining (CBC) mode, chosen ciphertext attacks, Datagram TLS

1. INTRODUCTION

Nowadays, many security-critical applications are carried out over the Internet or other networks. Financial transactions, signing of legally binding contracts, connecting to corporate networks via VPN, and many more applications require a significant level of security to be reliable for productive use. Usually, they demand integrity of messages and confidentiality of message contents, among other properties. For example, they may also ask for authentication or identification of the other communicating party. Standards that aim to meet such requirements are, e.g., the TLS standard, and many library and custom implementations exist. However, erroneous behavior as dictated by standards or defined by implementations may allow for partial or full recovery of an encrypted message's plaintext, or even to code execution. Such attacks exploit unwanted side channels exposed by the respective cryptographic protocols, and thus establish an *oracle* to make assumptions about the underlying plaintext using easily predictable *padding* bytes. Hence they are known as *padding oracle attacks*.

In Chapter 2, we will explain the aforementioned terminology and the theoretical background necessary to understand these plaintext recovery attacks. We will also cover the prerequisites for an adversary to carry out such an attack, and the long history of padding oracle attacks. In Chapter 3, we will explain padding oracle attacks in detail. Chapter 4 will feature a selection of such attacks which affected widespread libraries, framework, and end-user software implementations. Furthermore, we will assess the relevance for currently used software.

2. BACKGROUND

In this chapter, we provide some historical background of padding oracle attacks, and point out why they are still relevant. Subsequently, we will introduce the reader to the most basic cryptographic primitives and concepts required to understand padding oracle attacks. Finally, before continuing with padding oracle attacks themselves in detail in Chapter 3, we will explain under which circumstances these attacks are feasible for an attacker.

2.1 History and Relevance

A padding oracle attack for symmetric cryptography has first been proposed by Vaudenay in 2002 [19]. Similar attacks, however, had already been shown theoretically feasible as early as 1998 for RSA [7], though not entirely as efficient. Thus, for now more than a decade, padding oracle attacks are known. Still, standard and implementation errors facilitating such attacks have repeatedly emerged. The basic susceptibility to such attacks is derived from the MAC-then-pad-then-encrypt paradigm defined by standards, and thus cannot easily be fixed. As a consequence, relevance is still given nowadays, as by design implementations can still be prone to these attacks. Furthermore, as will be shown later in this paper, *all* block ciphers are generally prone to this kind of attack. Additionally, cipher strength is completely irrelevant for the probability of success.

2.2 Cryptographic Basics

In the following, we introduce the reader to the very basics of cryptographic primitives and operations related to encryption and decryption.

doi: 10.2313/NET-2013-08-1_11

We denote a plaintext message to be encrypted as m. m is a sequence of bytes of any length ($m \in \{0,1\}^{8n}$). The corresponding encrypted message, known as ciphertext, will be denoted as c, as is a sequence of bytes that is a multitude of b, known as the block size. The following trivial equations describe the relationship between plaintext message m and ciphertext c, where E is the encrypting, D the decrypting function, and k the (symmetric) encryption/decryption key:

$$E(m,k) = c; \qquad D(c',k) = m'$$

E is executed on the sender's side, while D is executed on the receiver's side. If c has not been altered during transmission, then $c = c'$ and thus $m = m'$.
All of the aforementioned components describe a symmetric-key cryptographic system.

Basic operator: XOR. The basic operation for most of today's symmetric cryptographic algorithm is the XOR operator, which may be preceded or followed by a permutation or substitution of any input operators, including the key k and the plaintext message m. The reason for using the XOR operator is that it is an involution, i.e., applying XOR to any bit o is reversible by applying XOR with the same parameter bit $p \in \{0,1\}$ once more: $XOR(XOR(o,p),p) = o$. XOR is often denoted as $\oplus$.

Symmetric cipher types. Cryptographic algorithms are commonly referred to as *ciphers*. Symmetric ciphers can be divided into two classes: Stream ciphers and block ciphers. The former encrypt plaintext messages by iteratively combining input data with a pseudorandom keystream. The latter take chunks of input data, known as blocks, and encrypt those, one block at a time. The attacks described in this paper concern *block ciphers*.

2.2.1 Cipher Block Chaining

Block ciphers can operate in different modes. These modes allow for the application of block ciphers to input data larger than the block size. Furthermore, they offer different services, such as encryption, integrity, and authenticity, and vary in their susceptibility to different attacks. The trivial mode of operation is the electronic codebook (ECB) mode, where each block is encrypted independently using the same key. This, however, raises security problems. For example, two identical plaintext messages will always translate to the same ciphertext. Also, knowing only parts of the underlying plaintext message for a ciphertext allows the attacker to directly deduce parts of the key, called a known-plaintext attack. One of the most common modes, however, is the cipher block chaining (CBC) mode, which is illustrated in Figure 1. In this mode, the ciphertext block C_i generated by the cipher does not only depend on the encryption key and the plaintext input block M_i, but also on the ciphertext block C_{i-1} which has been encrypted previously. To this end, each plaintext input block is XORed with the last ciphertext block. Formally, this means:
$C_i = E(M_i \oplus C_{i-1}, k)$, where C_i is the ciphertext block i and M_i is the plaintext block i. The first ciphertext block C_1 is generated using an initialization vector serving as C_{i-1}, i.e., C_0. Oftentimes, the IV is predefined by the implementation or transmitted in plain text during session initiation.

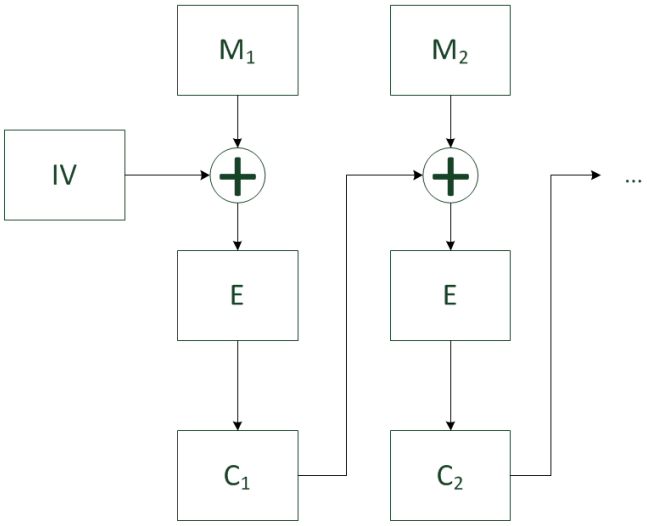

Figure 1: CBC mode encryption [12]

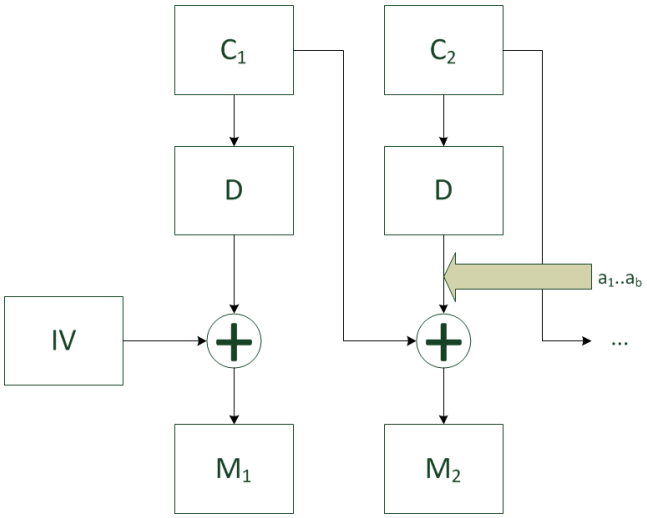

Figure 2: CBC mode decryption [12]

Decryption in CBC mode is done accordingly, by XORing each ciphertext after with the previous ciphertext executing the decryption function to obtain the correct plaintext. This is formally described by the equation $M_i = D(C_i,k) \oplus C_{i-1}$ and visualized in Figure 2. Please note that, after decryption of a ciphertext, only XOR operations are carried out which are not dependent on any secret key or cryptographic algorithms. In Figure 2, the parameters for these XOR operations are C_{i-1} and an intermediate value denoted as a.

2.2.2 Padding

As CBC mode requires blocks of fixed size as input, but plaintext data input may come in arbitrary sizes, data needs to be appended in order for the input blocks to be aligned in size. This needs to be done in such a way that padding can be distinguished from the true payload so it can be reliably removed after decryption. Several standards describe different methods how padding can be performed. For example, the Public Key Cryptography Standard (PKCS) #7

doi: 10.2313/NET-2013-08-1_11

[16] defines that each added byte should be the value of the number of bytes to be added in padding, i.e., if there need to be 5 bytes added, each of these bytes will have the value `0x05`. Even messages being a multiple of the block size need to be padded. In this case, one full block of padding with bytes of value b are appended, with b being the block size. Thus, at least the last byte of a message needs to be padding and padding must always be present.
Standard-conforming padding, in addition to the design principles behind most implementations of cryptographic standards, make portions of the plaintext easily guessable. One correctly guessed byte at the end of a message tells an adversary the value of a number of other bytes, with the number of bytes determined by the guessed byte's value. This is an important factor in the feasibility of padding oracle attacks.

2.3 Message Authentication Codes

In short, Message Authentication Codes (MACs) are cryptographic checksums ensuring the authenticity of a message. MAC values can be generated from keyed hash functions (HMAC), constructs based on regular hash functions such as MD5, or some block cipher modes. They compute a checksum over a given input by also including a key in the calculation. The MAC is added to a message and can be used for validation of this message by its receiver. When implemented correctly, this ensures not only authenticity, but also integrity of a message. As an adversary does not know the secret key established by the communicating parties, he cannot alter the message contents without detection.

3. PADDING ORACLE ATTACKS

Generally, a padding oracle is an algorithm that provides an adversary with information about the validity of the padding of the underlying plaintext, when presented with any ciphertext [3]. If the padding is valid (cf. Section 2.2.2), such an oracle returns 1; if it is invalid, it returns 0. This can be used to guess all padding bytes of a message very efficiently, and ultimately serve as a stepping stone to decrypt whole messages.
Generally, padding oracles as described in this paper try to decrypt any ciphertext message fed to them, and return whether or not the padding is valid. It is important to note that they will execute the decryption function for an adversary, even though they will not return the plaintext to the adversary. They will, however, state whether the plaintext has valid padding. By using the statements on the validity of the output plaintext, an adversary can directly deduce the values of the intermediary value output by the decryption function, prior to XORing with the previous cipher block (cf. Figure 2).

3.1 Last Byte Oracle

In general, due to the nature of decryption in CBC mode, it is most practical to first guess the last byte of a message. Building upon our padding oracle, we can determine if any random ciphertext consisting of a byte sequence determined by us has – when decrypted to plaintext – a valid padding [19]. In the most likely case, the oracle will return 1 if our last byte translates to a `0x01` in plaintext, meaning that only 1 byte of the message is padding.
To determine the last byte of a block C_n, one can carry out the following operations. For this, we will construct a message M consisting of the block of encrypted data we want to decrypt (C_n), and a forged block prepended to the encrypted data we want to decrypt (C'_{n-1}). This message is then $M' = C'_{n-1}||C_n$, where $||$ denotes concatenation. We will denote all bytes of the intermediate value returned directly by the decryption function before XORing with the previous cipher block as $a = a_1..a_n$, and the decoded plaintext resulting from our forged message as $p = p_1..p_n$.
(1) Select a random byte string $C'_{n-1} = c'_1, ..., c'_b$, equaling in size the block length of the cipher
(2) The last byte of this string is XORed with an integer i . In each step, i is incremented from 0 to 255 to assume all possible byte values.
(3) Upon each incrementation of i, the padding oracle O is fed with our randomly chosen byte string C'_{n-1} prepended to the block C_n we want to decrypt. If $O(C'_{n-1}||C_n) = 1$, we have correct padding, otherwise we continue with the next incrementation of i. The padding is most likely to be `0x01`, given that in any other case, one or more bytes would have to have such a value that padding is valid, i.e., the previous byte would need to be `0x02`, the two previous bytes `0x03 0x03`, and so forth.

3.2 Padding Length Detection

Once the last byte has been set such that an adversary's composed message $M' = C'_{n-1}||C_n$ has valid padding, it has to be determined how many bytes of valid padding exist at the end of $M'_n = D(C_n) \oplus C'_{n-1}$. This way, at least one byte of plaintext will be definitely discovered; if lucky, one may even guess multiple bytes correctly. To determine the padding length, the adversary will iterate over all bytes of C'_{n-1}, starting with the first, and XOR it with all possible values $\in [0, 255]$. If any byte alteration leads to the padding oracle returning 0, it is clear where the padding stops.
Once the length of the padding of our resulting plaintext p has been determined, the values of the corresponding bytes in the intermediate value a directly follow: If the padding has length 1, $a_b = r_b \oplus$ `0x01`; if length 2, $a_{b-1}a_b = r_{b-1}r_b \oplus$ `0x02 0x02`, and so on. The correct plaintext bytes of C_n can be acquired by XORing with the correct, non-forged C_{n-1}.

3.3 Block Decryption Oracle

Constructing an oracle for guessing the bytes of a whole block, and thus being able to decrypt it by XORing all guessed intermediate bytes $a_1..a_b$ with the unforged previous cipher block is trivial when implemented iteratively. The adversary already knows the value of one or more bytes at the end. He can easily construct a valid padding extending one byte further to the beginning of the cipher block to be decrypted. For this, the adversary needs to increase all encoded bytes by 1 so they form a valid padding which is "off by 1". E.g., if the last three bytes have already been decoded, a valid padding would be `0x03 0x03 0x03`. Setting these bytes now to `0x04 0x04 0x04` allows for the breaking of the byte right in front in an identical fashion to the Last Byte Oracle described above.

This methodology can be generalized not only to blocks, but to whole messages, enabling an adversary to decrypt whole encrypted conversations between two communicating parties. This approach is extremely efficient: To guess each byte, an adversary will require 2^7 steps on average. Thus, a block of 8 bytes takes only 1024 guesses on average; blocks of 16 bytes take 2048 tries, accordingly.

3.4 Cipher Independence

All operations we can directly influence with our forged parameter r in $m = r||C_n$ are XOR operations. They are not dependent on a secret key or any other cryptographic operations that are cipher-specific. A cipher, in our case, operates as a black box that supplies the adversary with the intermediate values $a = a_1..a_b$; its implementation is irrelevant. The only cipher parameter of importance is the block length. Consequently, cipher strength does not matter in this attack: No matter how strong a cipher, it can still be broken in linear time.

3.5 Padding Scheme Independence

As all padding schemes follow some system where padding needs to be distinguishable from plaintext payload, padding oracle attacks apply for all padding schemes and not only for the one mentioned above. An adversary, however, needs to know which padding scheme is used to be able to guess padding bytes.

3.6 Prerequisites for a Successful Attack

To mount a padding oracle attack successfully, an adversary must have access to (a) the packets he wants to decrypt, and (b) the padding oracle. Such a scenario is given when he can not only read packets between the two communicating parties, but also impersonate at least one of them to send packets to the oracle. For example, this is the case when an adversary assumes control of a hop en route of the communication packets, or if he can successfully conduct a man-in-the-middle attack in the local network of one of the two parties. He can then both intercept packets and also send forged packets on behalf of the party whose local network he is in while carrying out the man-in-the-middle attack.

3.7 Implementation and Practical Aspects

To implement a padding oracle, an adversary needs to have access to an entity that leaks one bit of information: Whether padding is valid, or not. Most commonly, this will be the case when an adversary has direct access to the communication between parties A and B which he aims to decrypt. Not only needs he to be able to read packets, but also to send them on behalf of at least one of the communicating parties: This way, by assuming the other party's identity, he can send them to the remote site and record the response. For example, in the case of TLS, an adversary will send packets on behalf of party A to party B. Party B will reply with error messages if the padding of the forged message was invalid. In the case of DTLS, there are slight timing differences in the handling of packets depending on the validity of padding. These differences have to be observed by an attacker, and also the packets to trigger these responses need to be sent by him. In practice, this can be achieved by man-in-the-middle attacks in the local network of one of the communicating parties. [3]
Effectively, padding oracle attacks are side channel attacks. Depending on the implementation an adversary wants to attack, he needs to interpret side channel information that he himself can provoke by assuming a position in between the communication parties. The type of information that will be leaked is implementation-dependent.

4. SAMPLE ATTACKS

It has been shown that various very widespread implementations of cryptographic protocols are or were prone to padding oracle attacks. Among them are the TLS and DTLS implementations of OpenSSL and GnuTLS [19, 3] and various software frameworks such as ASP.NET [10] and Ruby on Rails [17]. Of those, we will provide some detail on the attacks against TLS, which was the attack originally envisioned in [19]; against DTLS, which is a rather recent one; and one against ASP.NET, which gained some fame as it affected many websites, was easily exploitable, and could even lead to code execution [10]. All of these attacks have in common that they concerned widely used implementations to be found on many systems and in a lot of end-user software. However, the implementation of the padding oracle itself varied in significant aspects.

4.1 SSL/TLS

The first padding oracle attack on a symmetric cipher was published by Vaudenay in 2002 [19], though it is somewhat similar in concept to earlier attacks on RSA [7, 13]. While TLS implementations, abiding to standard, send out error messages directly indicating that padding of a forged message injected by an adversary is incorrect, this instantly terminates the connection. Standard dictates that any cryptographic error ought to be fatal. Thus, an SSL/TLS oracle, as proposed in [19], is a *bomb oracle*. It either returns 1 for correctly padded messages, or in the other case, explodes, thus terminating the session. A reestablished session will have new key material, and thus lead to different intermediate results (formerly in this paper referred to as $a = a_1..a_b$). Consequently, an attacker can only decrypt the last byte of a block with probability $\frac{1}{256}$, the last two bytes with a probability of $\frac{1}{512}$, and so on. Provided an adversary has knowledge about the underlying plaintext messages and their reoccurence, this may still be used for a successful attack to incrementally decrypt such messages. A multisession attack was developed in [8] to recover passwords (or other plaintext snippets) from a TLS-secured connection. The attack is extended even to non-distinguishable error messages by means of timing measurements. Error messages may be uniform in content, but not in the time it takes to generate them. This is exploited by this attack.

4.2 DTLS

Datagram TLS is a variant of TLS to be operated on top of an unreliable, connectionless protocol, i.e., in virtually all cases UDP. DTLS was standardized much later than 2002, hence the original padding oracle attack affecting SSL/TLS had already been known and countered by sending back uniform error messages for errors in cryptographic operations in TLS 1.1[1], on which the initial DTLS version was based. As there is no distinctive message indicating a padding error anymore, this side channel option is eliminated. However, at the time of the discovery of this attack, there were slight to significant timing differences in the handling of packets with invalid padding as compared to those with valid padding. Most interesting about this attack is that DTLS, unlike SSL/TLS, does not terminate the connection

[1]"Handling of padding errors is changed to use the bad_record_mac alert rather than the decryption_failed alert to protect against CBC attacks." [15]

doi: 10.2313/NET-2013-08-1_11

on cryptographic errors, as the underlying transport protocol is considered unreliable and might have bit errors. A padding oracle for DTLS is thus not a bomb oracle. On the other hand, the oracle cannot rely on captured error messages.
Two notable implementations of DTLS exist, one being part of GnuTLS and the other of OpenSSL. While the former followed TLS 1.1 very closely, thus also implementing time-uniform error reporting against timing side channel attacks, the latter did not adhere to the standard as closely when the padding oracle attack on DTLS was discovered. In either case, however, plaintext recovery was possible; with OpenSSL full, with GnuTLS partial recovery.

4.2.1 OpenSSL Padding Oracle

Prior to the fix which prevents the padding oracle attack described in [3], OpenSSL did not comply to the TLS 1.1 standard which DTLS is based upon in most parts. While TLS 1.1 dictates that processing time for all messages should be equal whether or not a messages is malformed, the OpenSSL DTLS implementation only conducted MAC verification of a message if the padding of the decrypted message is valid. This leads to significant timing differences in DTLS packet handling.
However, as DTLS implementations are not required to send out error messages, this difference in the time required to process packets had to be measured in another way. In [3], the authors chose to exploit heartbeat messages – periodic messages ensuring that the remote host of a DTLS connection is still up and reachable. For this, an adversary will send a sequence of packets to the oracle, directly followed by a heartbeat message. If padding of the packet sequence is valid, then MAC verification will be performed, leading to a higher delay of the heartbeat response. On the other hand, if the padding was invalid, then no MAC verification will be performed and the heartbeat response will return quicker. However, noise such as network congestion or routing choosing different paths may be introduced into measurements. Furthermore, in general the timing differences will be very small. To increase the reliability of his measurements, an adversary may carry them out repeatedly. Also, to gather typical response times, an adversary can carry out system profiling prior to utilizing the oracle. This is achieved by sending multiple packet sequences of different lengths to the oracle and record the response time.

4.2.2 GnuTLS Padding Oracle

As the GnuTLS implementation adhered closely to the TLS 1.1 standard, the above approach for OpenSSL does not hold. However, sanity checks in the implementation limit validation of a message's MAC to its header fields if padding is found to be incorrect, setting the message's effective payload length for MAC computation to 0. In either case, whether or not padding is valid, MAC computation is performed – however, in case of invalid padding, the processing time is shorter. These processing time differences, however, are very small on modern machines. As a consequence, variation in packet transit time introduces significant noise into this method.
As with the OpenSSL padding oracle implementation, an adversary may choose large messages to cause a maximum timing difference or carefully timed packet sequences. Using these methods only a partial plaintext recovery is possible. Using statistical analysis methods and increasing the number of guesses drastically, single bytes could be recovered with probabilities up to 0.99. However, the amount of network traffic necessary for this partial plaintext recovery makes the attack rather unpractical.

4.3 ASP.NET

In 2010, it was found out that the whole ASP.NET framework was vulnerable against a padding oracle attack in such a way that any data on a webserver running ASP.NET was publicly accessible. The resulting vulnerability was identified as MS10-070 [14] and CVE-2010-3332 [2]. It affected every installation of ASP.NET, as the error was inherent to the framework itself, and not to specific web applications on top of ASP.NET. The attack is based on an adversary being able to turn a decryption oracle into an encryption oracle [10], which will not be covered in detail here. ASP.NET allows a developer to make the *navigation* of a website encrypted, meaning that the actual web resource location identifiers of a request are encrypted. This can aid in hiding the underlying structure of a web application. Resources are served to a user by *WebResource.axd* and *Script Resource.axd*. The format for a request is as follows: `WebResource.axd?`*d*`=encrypted_id&t=timestamp`, with *d* being the identifier of a web resource, specified by its relative path in a web application. If an adversary is capable of transmitting a self-chosen valid *d* parameter, he may access any resource of the web application. *d* is supplied to a client as part of web resources the client already accessed to allow for navigation, and is computed by the ASP.NET server using a private key known only to the server. The oracle is implemented by distinguishable error messages returned by the server: a 404 error code implies valid padding, and 500 is returned otherwise. [10]
Now, using the encryption oracle mentioned above, the attacker constructs a *d* parameter such that it points to resources storing crucial information crucial to the security of the web application and server. This includes, for example, the keys used for encrypting and decrypting data on the server, including valid *d* parameters. With the help of this oracle, *d* parameters can be constructed for any path of the web application. In the case of ASP.NET, the *web.config* resource contains cryptographic keys and sometimes even login credentials. Using cryptographic keys, an adversary may obtain validated sessions with the web application. Furthermore, the `App_Code` directory in ASP.NET applications contains source code files. [10]
As can be seen, when an application trusts encrypted data supplied by a client for serving resources, and if a padding oracle can be constructed, then an oracle attack may not be used only to decrypt ciphertexts, but to assume control over the application and gain access to otherwise protected crucial resources.

5. ANALYSIS

In this section, we will detail which common mistakes in cryptographic constructs and protocol designs lead to padding oracles. These can, as mentioned before, not only be used to decrypt messages, but also be leveraged to obtain encryption oracles as well to inject arbitrary encrypted data [17]. After presenting potential remedies to these problems, we will give an assessment of the relevance of such attacks for end-users.

5.1 Common Design Flaws and Fixes

Three prominent mistakes in the design of cryptographic constructs can be identified:

1. Unauthenticated, encrypted messages: Attackers may inject arbitrary messages into a communication channel and the remote side will treat them as if they were from the legitimate communication partner.

2. If messages are authenticated and their integrity protected, this only applies to the plaintext and not to the full ciphertext messages (including padding). This allows adversaries to tamper with the ciphertext and mount chosen-ciphertext attacks which padding oracle attacks are. This has been referred to as the "MAC-then-encrypt" or "MAC-then-pad-then-encrypt" paradigm by other researchers.

3. Fixed or plaintext IV. While the IV only directly affects the first block of ciphertext in an encrypted message, the IV being unknown to an adversary in any attack on CBC mode has significant positive effects: An adversary may reconstruct all blocks except for the first plaintext block. The first plaintext block often contains crucial protocol parameters in a header. If those parameters are unknown, any attacks that require knowledge about these parameters are prevented. Also, if sanity checks are performed on a header, e.g., the sequence number of a message being within a certain range for replay attack protection, an attacker cannot circumvent such measures. Furthermore, attacks allowing for recovery of arbitrary plaintext can be mounted by attackers who are able to predict the IV [5]. Thus, we conclude that the IV is a resource which is not well enough protected.

These three issues can be easily addressed in theory.
Issue 1 can be resolved by making authentication of messages compulsory, which usually also protects integrity. This case is also supported by other publications [6]. As noted in by Black and Urtubia [6], authenticity and privacy are often regarded as different objectives. However, given the nature of ciphertext processing, they are in fact strongly correlated. Using MACs on every message would prevent tampering with ciphertext, and injection of arbitrary forged messages into conversations.
If authentication is already used, it needs to be done on the full payload that leaves a host, and not only on the underlying plaintext, as also demanded by [19]. This addresses Issue 2.
In fact, Issues 1 and 2 can be resolved effectively and efficiently by using *authenticated encryption*, a paradigm for cipher modes which provides confidentiality, authenticity and integrity in a single mode of operation [6]. By providing these services out of the box, developers no longer need to combine them themselves, reducing the potential for mistakes. Some of these modes are the Offset Codebook Mode (OCB), the Counter with CBC-MAC (CCM) mode, and the EAX mode.
To resolve Issue 3, the IV should be agreed upon by the communicating parties during session establishment, instead of the IV being fixed or transmitted in plaintext. However, the IV must not be encrypted using the same key as all other payload data, as an attacker knowing the IV can otherwise use it to decrypt other messages. This is a consequence of the way many cipher modes combine message blocks using XOR. Thus, the IV must be encrypted using a different key.

5.2 Relevance for End Users

As of June 2013, widely used software such as Mozilla Firefox and Thunderbird [1], Oracle and Internet Explorer are still using TLS 1.0. Though TLS 1.1 and 1.2 are implemented, some bugs remain and many servers fail at handshake, preventing it from being enabled by default. However, TLS 1.0 is still susceptible to padding oracle attacks. Hence in theory, users of all major browsers are prone to padding oracle attacks, though a padding oracle for TLS 1.0 is a bomb oracle. Generally, many client and server applications often do not feature the latest TLS version 1.2 or even 1.1, depending on their implementation, or in the case of libraries, against which library version they were linked. If the server does not feature TLS $>$ 1.0, the client also will not be able to use it. In general, the protocol mostly used today is TLS 1.0, which is prone to the attacks presented in this paper. The most widely used SSL/TLS library OpenSSL, on the other hand, received fixes of these issues in releases 0.9.8s and 1.0.0f [3].

5.3 Lessons Learned

Apart from the design flaws addressed in Section 5.1, there is another lesson to be learned. Even one single bit leaked to an attacker can lead to the compromise of information. This also holds for web applications. Side channels should be avoided at all costs to prevent oracles of any kind. In practice, this would mean disabling distinguishable error messages in deployment stage that are not necessarily required for protocols to function properly. There are many other examples where an attacker can gain access to a system only though 1 bit of information leaked per request, e.g., in blind SQL injection attacks. Thus, we consider it to be crucial to generally disable distinguishable error messages after development phase wherever possible.

6. RELATED WORK

Much research has been done in the area of padding oracle attacks. Very notable and with the highest relevance for practice is the work by J. Rizzo and T. Duong. Their publications include papers on practical plaintext recovery attacks against SSL/TLS such as the BEAST [11] and CRIME [18] attacks, and a tool capable of conducting such attacks dubbed Padding Oracle Exploit Tool (POET) [9]. They also discovered the ASP.NET attack covered in Section 4.3.
AlFardan and Paterson, authors of the attacks against DTLS presented in Section 4.2, also recently developed another attack on (D)TLS nicknamed Lucky Thirteen after the thirteen byte TLS header [4]. In the case of TLS, it is currently not very efficient as it requires multiple sessions and is very susceptible to packet transit time differences, but may be enhanced by man-in-the-browser attacks like BEAST or other means. This attack is, again, a timing side channel-based oracle attack.

7. SUMMARY AND CONCLUSION

In this paper, we reviewed the theory behind padding oracle attacks and introduced the reader to three practical implementations. Two of those can be considered classical, as

doi: 10.2313/NET-2013-08-1_11

they aim to recover plaintext of live communication, targeting DTLS. The third attack showed how unauthenticated encrypted data can be used to manipulate applications and compromise them completely. Following the attacks, we provided the reader with an assessment of typical design errors that lead to such attacks and present approaches at fixing those. Finally, we pointed out the relevance for end users based on the prevalence of TLS versions in end user software.

Padding oracle attacks are an example of the importance of correct implementation of cryptography. They demonstrate how the strongest cryptosystems can be broken in linear time if an attacker can tamper with intermediate results in algorithms. This stresses the importance of ciphertext integrity protection as well. Furthermore, even one bit of leaked information – in this case whether a message is correctly padded or not – is sufficient to conduct full plaintext recovery attacks. Side channels can be exploited to gather such leaked information remotely. In general, any cryptographic processing behavior should be perfectly indistinguishable from the outside and completely independent of input.

As CBC mode is generally prone to such attacks, we encourage the use of secure block cipher modes such as Offset Codebook Mode (OCB), Counter with CBC-MAC (CCM) mode, or EAX mode. These modes provide integrity and authentication of ciphertext by default. This way they also prohibit attackers from injecting arbitrary ciphertext messages, as authentication would fail.

Currently, users of popular browsers are in theory still at risk of padding oracle attacks. An example is the recent Lucky Thirteen Attack. Passwords or other crucial information may be recovered from encrypted communication, but not larger parts of plaintext. While this may be fixed by browser developers in the foreseeable future by enabling the most recent TLS version by default, servers are also required to implement this version correctly. As of now, this is often not the case. Nevertheless, the most recent padding oracle attacks are not trivial to conduct and require the attacker to be en route or in the local network of one of the two communicating parties. Thus, though generally feasible, such attacks will not become a widespread threat. Targeted intrusions and advanced persistent threats (APT) might apply them however, as APTs usually act locally and over an extended period of time.

8. REFERENCES

[1] Firefox Bug 733647: Implement TLS 1.1 (RFC 4346) in Gecko (Firefox, Thunderbird), on by default. `http://bugzilla.mozilla.org/show_bug.cgi?id=733647`; retrieved June 1, 2013.

[2] CVE-2010-3332 ("ASP.NET Padding Oracle Vulnerability"), September 2011. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3332`.

[3] N. J. AlFardan and K. G. Paterson. Plaintext-recovery attacks against datagram tls. In *Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[4] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. 2013.

[5] G. V. Bard. The vulnerability of SSL to chosen plaintext attack. 2004.

[6] J. Black and H. Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *Proceedings of the 11th USENIX Security Symposium*, pages 327–338. USENIX Association, 2002.

[7] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology—CRYPTO'98*, pages 1–12. Springer, 1998.

[8] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a SSL/TLS channel. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599. Springer Berlin Heidelberg, 2003.

[9] T. Duong and J. Rizzo. Padding oracles everywhere (presentation slides). In *Ekoparty 2010*, 2010. `http://netifera.com/research/poet/PaddingOraclesEverywhereEkoparty2010.pdf`.

[10] T. Duong and J. Rizzo. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *IEEE Symposium on Security and Privacy 2011*, pages 481–489, 2011.

[11] T. Duong and J. Rizzo. Here come the ⊕ ninjas, May 13 2011. Manuscript published online: `http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf`.

[12] C. Eckert. *IT-Sicherheit: Konzepte, Verfahren, Protokolle*. Oldenbourg, 2013.

[13] J. Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS# 1 v2.0. In *Advances in Cryptology – CRYPTO 2001*, pages 230–238. Springer, 2001.

[14] Microsoft. Microsoft Security Bulletin MS10-070 (originally published Sept 28, 2010), September 2011. `http://technet.microsoft.com/de-de/security/bulletin/ms10-070`.

[15] Network Working Group, IETF. RFC 4346: The Transport Layer Security (TLS) Protocol Version 1.1, April 2006. `https://www.ietf.org/rfc/rfc4346.txt`.

[16] Network Working Group, IETF. RFC 5652: Cryptographic Message Syntax (CMS), September 2009. `https://www.ietf.org/rfc/rfc5652.txt`.

[17] J. Rizzo and T. Duong. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies, WOOT*, volume 10, pages 1–8, 2010.

[18] J. Rizzo and T. Duong. The crime attack (presentation slides). In *Ekoparty 2012*, 2012. `http://netifera.com/research/crime/CRIME_ekoparty2012.pdf`.

[19] S. Vaudenay. Security flaws induced by cbc padding – applications to ssl, ipsec, wtls... In L. R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–545. Springer Berlin Heidelberg, 2002.