Variants of the Padding Oracle Attack

Md Ariful Haque
CIISE, Concordia University
Montreal, Quebec, Canada

Sima Bagheri
CIISE, Concordia University
Montreal, Quebec, Canada

Alireza Moghaddasborhan
CIISE, Concordia University
Montreal, Quebec, Canada

Yaswanth Kalyanam
CIISE, Concordia University
Montreal, Quebec, Canada

Piyush Adhikari
CIISE, Concordia University
Montreal, Quebec, Canada

Sourov Jajodia
CIISE, Concordia University
Montreal, Quebec, Canada

Mahdieh Ghorbanian
CIISE, Concordia University
Montreal, Quebec, Canada

Shafayat Hossain Majumder
CIISE, Concordia University
Montreal, Quebec, Canada

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

In the ever-evolving digital era marked by an increase in cybercrime, the Secured Socket Layer/Transport Layer Security (SSL/TLS) protocol is de facto to secure the privacy and integrity of online data and network traffic [18]. Although these protocols are crucial for preserving confidentiality and data integrity, the dynamic nature of cybersecurity has resulted in more sophisticated attacks against SSL/TLS protocols. The Oracle padding attack and its variants are examples of SSL/TLS protocol attacks.

The earlier SSL versions (SSLv1, SSLv2, and SSLv3) introduced by Netscape between 1994 and 1996 were deprecated in 2015 due to their insecure design and numerous vulnerabilities [19] [20]. The successors of SSL are TLSv1.0, TLSv1.1, TLS v1.2, and TLSv1.3. TLSv1.3 was introduced in 2018 as a significant upgrade over TLSv1.2 removing several security vulnerabilities [21] [7].

While the SSL/TLS version was upgraded to provide more security, the Oracle padding attack became more sophisticated and evolved into several variations. In this paper, we will look at the Oracle padding attack and its variants. This document offers eight attacks against the SSL/TLS protocol, including BEAST, POODLE, drown, CRIME, BREACH, and so on (a complete list can be constructed after everyone updates this). We successfully simulated four out of the eight attacks, the details of which can be found in further sections.

II. Background

The Oracle Padding Attack, introduced in 2002 [30], is a sophisticated exploit that leverages a padding oracle algorithm to scrutinize the integrity of padding on plaintext when provided with any ciphertext. This algorithm, designed to identify valid padding, returns true if the padding is accurate; otherwise, it returns false. This vulnerability opens the door for attackers to methodically guess each padding byte of a message, ultimately leading to the decryption of the entire message [15].

In the intricate landscape of cryptographic attacks, the oracle padding attack stands out for its exploitation of the server's behavior during the decryption of encrypted text. An astute attacker can extract valuable insights, capitalizing on cues such as prolonged decryption times or error code displays. In the specific scenario of an oracle padding attack, let's envision a scenario where the attacker acquires the ciphertext through a man-in-the-middle attack.

Functioning as a padding oracle, the server issues error messages upon encountering incorrect padding. The attacker strategically tests each byte of the ciphertext, iterating through the possibilities from 0 to 255, until the server confirms the correctness of the padding. Upon successful validation, the attacker employs XOR calculations to unveil the last byte of the plaintext. This meticulous process is then systematically repeated to unveil each preceding byte [15]. The vulnerability inherent in padding oracle attacks underscores the importance of robust encryption protocols and vigilant cybersecurity practices.

III. Padding Oracle Attack and its Variants

This section delves into notable variants of the padding oracle attack, namely BEAST, CRIME, BREACH, and POODLE. Each attack is introduced with a concise overview, elucidating the prerequisites or conditions necessary for its successful execution. Subsequently, we delve into implementation details, augmenting comprehension with illustrative examples. To fortify our exploration, we conclude with effective mitigation strategies tailored to counteract each specific attack. This comprehensive discussion aims to equip readers with a holistic understanding of these diverse padding oracle attack variants, their intricacies, and the corresponding strategies to enhance cybersecurity resilience.

A. BEAST Attack

The BEAST (Browser Exploit Against SSL/TLS) attack strategically exploits vulnerabilities in SSL 3.0 [16] and TLS 1.0 [12]. Leveraging encryption and the cipher block chaining (CBC) technique, the attack aims to deduce the secret key used for encrypting plaintext [27]. A key weakness lies in the selection of the last ciphertext block as the initialization vector (IV) for the subsequent plaintext encryption.

Discovered by T. Duong and J. Rizzo in 2011 [13], the BEAST attack targets HTTPS requests, specifically seeking to decrypt and acquire authentication tokens. This intricate attack underscores the significance of robust cryptographic protocols,

urging continuous advancements in encryption methodologies to thwart evolving threats in the digital landscape. Understanding the nuances of BEAST is imperative for cybersecurity professionals striving to fortify systems against such exploits.

*1) **Overview**:* HTTPS requests often harbor critical information, such as authentication tokens, making the decryption of these requests a significant threat to victims. The BEAST attack specifically targets vulnerabilities within the SSL/TLS protocols. SSL primarily utilizes symmetric-key encryption, employing either block ciphers or stream ciphers. BEAST becomes exploitable when a block cipher is in use [13]. The vulnerability arises when the Cipher Block Chaining (CBC) mode is employed for encryption, utilizing chained initialization vectors (IVs) [14]. In this context, "chained IVs" refer to the utilization of the last block of the preceding ciphertext for encrypting the new plaintext block. The use of CBC with chained IVs is deemed insecure [10], and this vulnerability extends to SSL [9].

Consider a scenario where a man-in-the-middle attacker, intercepting TLS 1.0 traffic, can inject data into it. Armed with knowledge about the data being sent and its location in the message, the attacker can inject a meticulously crafted data block. By comparing the resulting encrypted block with the corresponding block in the actual message stream, the attacker can ascertain the correctness of their injected guess. Successfully executing this process allows the attacker to unveil the plaintext block. Figures 1 and 2 provide a visual representation of the BEAST attack environment. Understanding the intricacies of this attack is paramount for cybersecurity practitioners seeking to fortify systems against such sophisticated exploits.

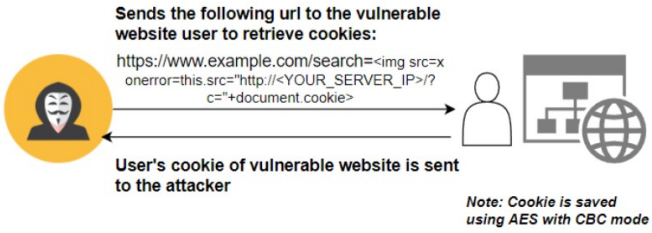

Fig. 1. Performing XSS to retrieve user's cookie

In TLS 1.0's Cipher Block Chaining (CBC) mode, the encryption of a data block involves XOR operations with the plaintext block, the preceding ciphertext block (known to the attacker), and the initialization vector (acquired by sniffing the previous message). As XOR is a reversible operation, an attacker can initiate a chosen plaintext attack. By guessing a probable data block, XORing it with the IV and the preceding ciphertext block, and injecting the result into the session, the attacker can craft a manipulated data block [13]. This vulnerability underscores the importance of robust encryption mechanisms to thwart potential exploitation.

*2) **Prerequisite for BEAST Attack**:* In executing a BEAST attack, the attacker requires both the ciphertext and control over the plaintext. In real-world scenarios, obtaining

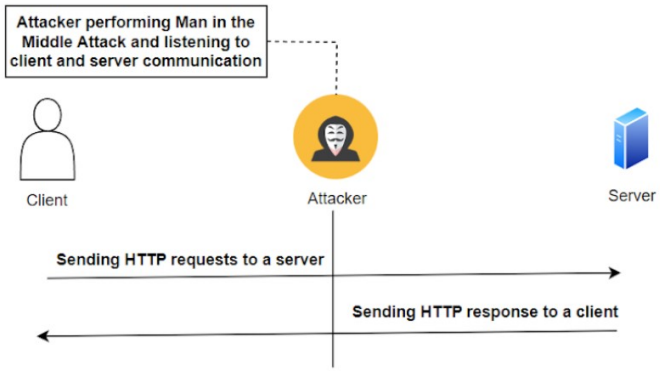

Fig. 2. Performing man in the middle attack

the ciphertext and asserting control over the plaintext can be achieved through various vulnerabilities, including cross-site scripting (XSS), man-in-the-middle (MiTM) attacks, and others. For the purposes of this report, let's consider a hypothetical web application utilizing TLS V1.0 and being susceptible to cross-site scripting (XSS) attacks. Leveraging XSS attacks, an attacker can effortlessly retrieve encrypted sensitive information such as cookies and session IDs (indicating access to ciphertext). Moreover, XSS attacks empower attackers to manipulate the plaintext by gaining control over the victim's browser, thereby escalating the impact of the BEAST attack.

*3) **Implementation:**:* Before delving into the implementation of the BEAST attack, it is crucial to understand the functioning of Cipher Block Chaining (CBC).

Cipher Block Chaining(CBC): In the CBC mode, an initialization vector (IV) is selected and XORed with the plaintext block (P). As depicted in Figure 3, the resulting XOR value undergoes encryption using a block cipher like AES, producing the cipher text (C). This newly generated cipher text becomes the initialization vector for the next encryption. The CBC mode encryption can be expressed succinctly as: C_n = C($IV \oplus Pn$); where IV is C_{n-1} except for the first block.

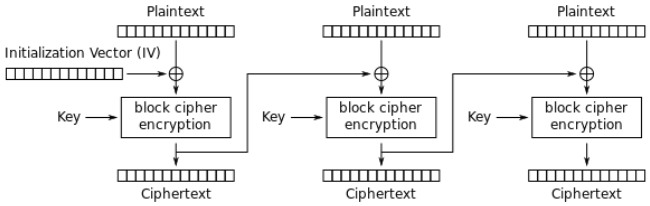

Fig. 3. CBC mode encryption

Breakdown of BEAST Attack implementation: Let us assume that P is the plain text containing n blocks of 16 bytes each [P = P_1, P_2, ..., P_n], C is the cipher text containing n blocks of 16 bytes each [C = C_1, C_1, ..., C_n]. Here, P_x and C_x are the x-th block of the plaintext and ciphertext respectively. The CBC mode encryption is as follows:

$$C_1 = E_k(IV \oplus P_1)$$
$$C_2 = E_k(C_1 \oplus P_2)$$
$$\vdots$$

As an attacker, we have access to the ciphertext blocks C_1, C_1, ..., C_n. Suppose, now we want to figure out plaintext block P_2, as it contains some sensitive information, and we have control over plaintext block P_5. So attacker crafts P_5 as follows:

$$P_5 = C_4 \oplus C_1 \oplus \text{Guess}$$

Here, Guess is a 16 bytes long block which we're trying to equate with P_2 by trial and error. Setting the P_5 block like this will result in the following scenario during encryption:

$$C_5 = E_k(C_4 \oplus P_5) = E_k(C_4 \oplus C_4 \oplus C_0 \oplus \text{Guess})$$

Here, C_4 XOR C_4 cancels out, simplifying the equation to:

$$C_5 = E_k(C_1 \oplus \text{Guess})$$

If somehow, we guessed the correct P_2, therefore the equation will turn out to be:

$$\begin{aligned} C_5 &= E_k(C_1 \oplus \text{Guess}) \\ &= E_k(C_1 \oplus P_2) \\ &= C_2 \quad [\text{if Guess} = P_2] \end{aligned}$$

Since, we know the value of C_2 we can guess i.e., by trial and error until C_5 = C_2 so that Guess = P_2. In this way, we can obtain any block of the plain text using BEAST attack. The overall scenario is visualized in 4.

Note: For guessing the plain text, we will only use 1 byte since it will be very difficult and practically infeasible to guess the 16-byte/128 bits block size used in AES, as it would result in 2^{128} tries. In the literature, a way to circumvent having to do so many tries is to use a prefixing approach. Here, the controlled plaintext block will be constructed in such a way that the first 15 bytes will be a known value, and only the last byte will be the information that needs to be extracted. Therefore, when that plaintext block gets encrypted, the attacker already knows the first 15 bytes, and he just needs to guess the last byte, which takes at most 28 = 256 tries to find out.

$$\text{Guess block} = \text{15 bytes of known prefix} \,||\, \text{1 byte of guess}$$

As an example, in the XSS scenario, since the attacker has control over the HTTP request the victim is making, he could make the user request for a URL path with a deliberately long name, such that only 1 byte of the target information falls inside a 16 byte block.

For instance, consider the Figure 5, represents an encrypted post request.

Here, the attacker is deliberately making the victim request to a path '`\AAAAAAAAAAAA`' (12 As), so that only the cookie value's first byte 'C' enters the 3rd plaintext block. The rest of the prefix '`secret_cookies=`' (15 bytes) is already known. Therefore, for the controlled plaintext byte, he takes the guess '`secret_cookies=X`' (16 bytes), where he tries all 256 ASCII values instead of the last byte 'X' until he gets the right one.

After some tries using the BEAST attack, he finds out that the last byte would be 'C', which is the first byte of the secret that he's trying to decrypt. Now, he'll shorten the requested path by 1 byte and make it '`\AAAAAAAAAA`' (11 As). Thus, the post request's block structure is showed in Figure 6:

Now for the controlled plaintext byte, the attacker takes the guess '`ecret_cookies=CX`' (16 bytes), updating it with the byte that he found from the last iteration. Again, he takes 256 tries of ascii values instead of the last byte '`X`', until he finds the next byte of the secret successfully. This keeps on and on until the attacker can decrypt the entire secret value.

4) ***Practicality:*** *:* Executing the actual attack could be challenging due to the browser's Same-Origin Policy (SOP) enforcement. Nevertheless, certain web technologies may offer a means for cross-domain communication. Doung and Rizzo highlighted various technologies that could potentially be employed to craft the attack script, including the HTML5 WebSocket API, Java URLConnection API (as showcased in their presentation by exploiting a distinct SOP bypass), and Silverlight WebClient API. It's essential to note that, apart from utilizing the Java SOP bypass vulnerability, no other avenues for exploitation have been publicly disclosed [26].

5) ***Mitigation:****:* To mitigate BEAST attack, the following approaches can be followed:

- If SSL 3.0 / TLS 1.0 version is used, an initial empty fragment into the message can be inserted in order to randomize the IV as in the case of OpenSSL. So, the attackers can not easily guess the initial vectors [22]. Though it was not very reliable.
- The TLS version should be upgraded. TLS version 1.1, preferably version 1.2 should be enabled [27].
- Disable cross-origin requests on the server side whey they aren't needed.

B. CRIME Attack

CRIME (Compression Ratio Info-leak Made Easy) is a security vulnerability against HTTPS and SPDY [5] protocols where compression is used in HTTP requests [3]. By executing this attack, HTTP requests header can be recovered, for example, cookies. We know cookies are the primary vehicle for web application authentication(after login), so if stolen, it can be used by an attacker to perform session hijacking on authenticated web sessions, allowing more threats. CRIME attack was first demonstrated by two security researchers Juliano Rizzo and Thai Duong in 2012 [25].

John Kelsey first introduced vulnerability exploitation where information leakage happens through data compression in 2002 [22]. CRIME, for the first time, gave a practical example how this attack can happen. This attack was revealed by Rizzo and Duong in 2012 at Ekoparty security conference

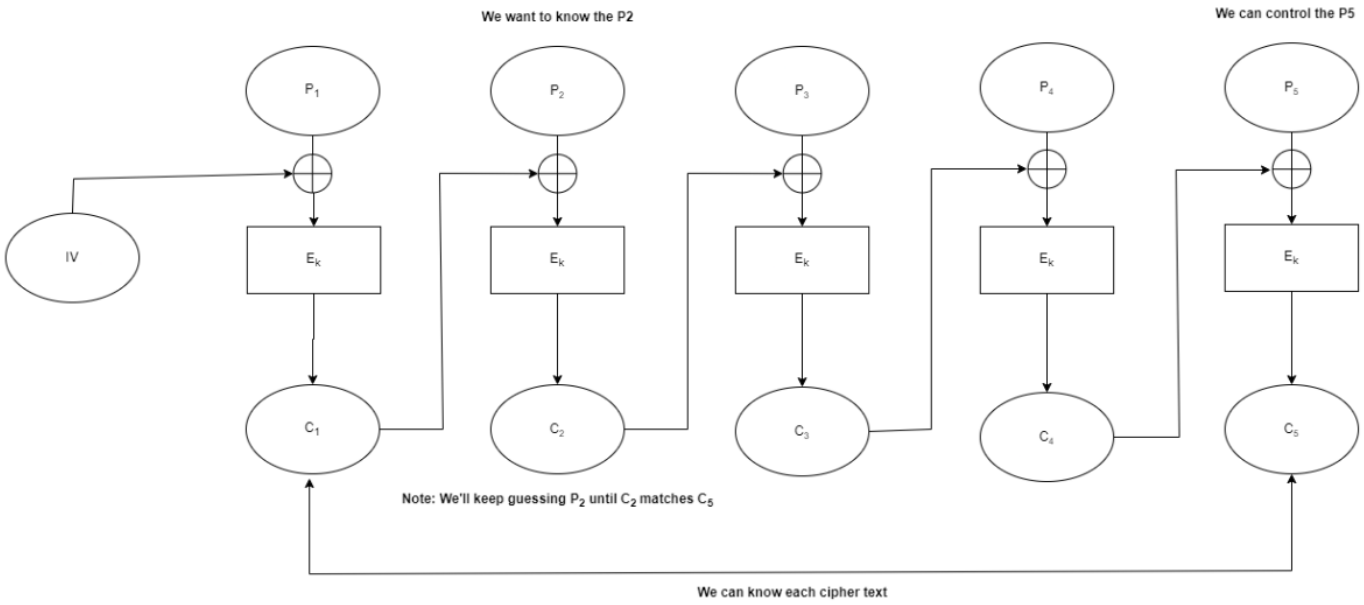

Fig. 4. BEAST attack (cap)

POST /**AAAAAAAAAA**	**AA** HTTP /1.1\r\n	secret_cookies=C	00kie1234

Fig. 5. An encrypted post request

POST /**AAAAAAAAAA**	**A** HTTP /1.1\r\n s	ecret_cookies=C0	0kie1234

Fig. 6. A modified encrypted post request

[25]. This attack was proved to be effective against a large number of protocols including SPDY [5], TLS and HTTP. the vulnerability related to this attack was assigned CVE-2012-4929 [4].

*1) **Overview:** :* HTTP requests may contain important session related information like cookies. If these cookies are leaked, an attacker can use them to hijack a session. CRIME attack is used to reveal the secrets from the HTTP requests. It exploits the compression techniques of TLS and SPDY [5] protocols when they use DEFLATE and gzip data compression schemes [29]. It relies on a combination of chosen plaintext attacks and inadvertent information leakage through compression. Attacker observes the change in size of the compressed request payload, which contains both the secret cookie that is sent by the browser to the target site, and variable content created by the attacker. When the size of the content is reduced, it can be said that it is probable that some part of the injected content matches some part of the source, giving the attacker a hint to guess the secret cookie correctly.

*2) **Prerequisite of CRIME attack:** :* Executing CRIME attack requires special environments and requirements. To execute this attack, an attacker should have and ensure the following abilities [17] [25]:

- Compression techniques, such as DEFLATE, gzip etc. are used in HTTP requests.
- He should be able to measure the size of encrypted traffic.
- Must be able to inject partially chosen plaintext into a victim's HTTP requests.

By this, an attacker can leverage the information leaked by compression to recover the targeted parts of the plaintext.

*3) **Implementation:** :* CRIME capitalizes on the data compression feature present in SSL and TLS protocols. Compression occurs at the SSL/TLS level, encompassing both the header and body. The compression algorithm employed by SSL/TLS and SPDY [5] is DEFLATE, which eliminates duplicate strings, thereby compressing HTTP requests. CRIME exploits the manner in which duplicate strings are eliminated, employing a systematic brute force approach to guess session tokens. Each occurrence of a duplicate string is substituted with a pointer to the initial instance of the string. Consequently, the level of redundancy in the data directly influences the compression amount. Greater data redundancy results in more compression, leading to a reduced length of the HTTP request. The attacker leverages this dynamic. The CRIME attack process is depicted below.

```
GET/ HTTP/1.1
Host: importantserver.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/100.0.0.0 Safari/537.36
...
Accept-Encoding: gzip, deflate
...
Cookie: secret=341267
```

Fig. 7. An example of HTTP request

In Figure 7, we observe an illustration of an HTTP request

from a website. The compression techniques of gzip and deflate are employed for compressing or encoding the HTTP request. The header of the request contains a cookie that holds a session secret. The CRIME attack capitalizes on the use of compression techniques for HTTP requests, fulfilling the initial condition for its execution.

```
GET/ HTTP/1.1
Host: importantserver.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/100.0.0.0 Safari/537.36
...
Accept-Encoding: gzip, deflate
...
Cookie: secret=781904
...
Cookie: secret=1 (Injected by Attacker)
```

Fig. 8. Attacker injects some text (e.g. by javascript)

In Figure 8, the attacker injects guesses into the HTTP request, satisfying the second condition for the CRIME attack. Utilizing DEFLATE compression, the system recognizes multiple occurrences of the "Cookie: secret=" part. The second instance is replaced with a small token pointing to the location of the first occurrence, reducing the request length by 15 (the length of the string "Cookie: secret="). Although the attacker cannot directly observe the data, they monitor the length changes in the request. The attacker employs a brute-force approach, systematically trying different values for the secret until the correct one is compressed. For instance, the attacker repeats the process with secret = 2, secret = 3, and so forth until the correct value is identified.

```
GET/ HTTP/1.1
Host: importantserver.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/100.0.0.0 Safari/537.36
...
Accept-Encoding: gzip, deflate
...
Cookie: secret=781904
...
Cookie: secret=7 (Injected by Attacker)
```

Fig. 9. Attacker injects some correct text (e.g. by javascript)

In Figure 9, when secret=7 the request's length decreases by 16. Thus the attacker realizes that he has retrieved the first character of the cookie. The same process is repeated until the entire cookie value is retrieved.

4) ***Practicality:*** *:* Executing the CRIME attack successfully requires a remarkably small number of requests, merely six per cookie byte [6]. Despite its minimal demand in terms of requests, the ramifications of this attack are substantial, especially when considering the widespread use of TLS across various online platforms and services. The attack's efficiency in obtaining sensitive information with such limited interaction underscores the urgency for robust countermeasures to safeguard against such threats in the realm of secure communication protocols.

5) ***Mitigation Techniques:*** *:* The mitigation techniques related to CRIME attacks are described below:

- **Disable compression:** CRIME attack was executable in TLS version 1.0 [12]. Because of that, TLS 1.1 and TLS 1.2 versions were introduced, where compression mechanisms were disabled at TLS levels and thus mitigated this problem [29].
- **Compression algorithm:** In TLS protocol version 1.2, client sends a compression algorithm in its ClientHello message, and the server picks one of them and returns it back in its ServerHello message. So, if the client offers or the server chooses **"none"** as a compression algorithm, no compression will take place [3] and thus prevent this problem.
- **Keep Browser Updated:** As newer attacks are introduced everyday, it is necessary to keep the browsers updated all time to prevent the attacks.

C. BREACH Attack

BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) is a security vulnerability against HTTPS requests when we use HTTP compression in our system. BREACH is built on the basis of CRIME attack. CRIME attack is used to recover the headers of an HTTP request by using TLS or SSL level compression whereas BREACH attack is based on HTTP responses [17]. BREACH was announced at the August 2013 Black Hat conference by security researchers Angelo Prado, Neal Harris and Yoel Gluck.

Compression side-channels attacks were not new. Over twenty years before, these attacks were described in general [22]. For the first time, CRIME showed a concrete example of how to use compression techniques to attack vulnerabilities [25]. In CRIME, an attack is executed to recover the header of an HTTP request, and since HTTP headers contain important information like cookies, it was a threatening situation. CRIME relied on TLS compression, so by disabling it, we can mitigate this attack. However, after that a new attack was introduced, that is called BREACH [17] [2]. It also utilizes compression techniques, however this attack is executed on HTTP responses. So, even if we disable TLS/SSL level compression, this attack can not be mitigated. An environment scenario is visualized in Figures 10, 11, 12, where BREACH or CRIME attacks can happen.

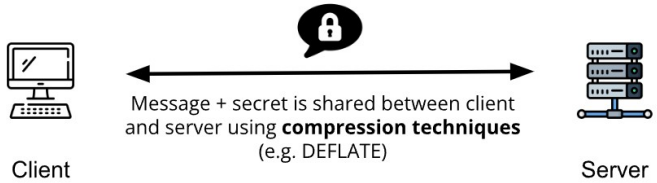

Fig. 10. Message sharing between server and client

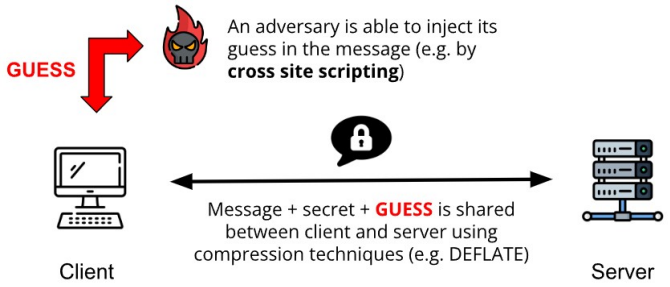

Fig. 11. Attacker can inject guess in messages

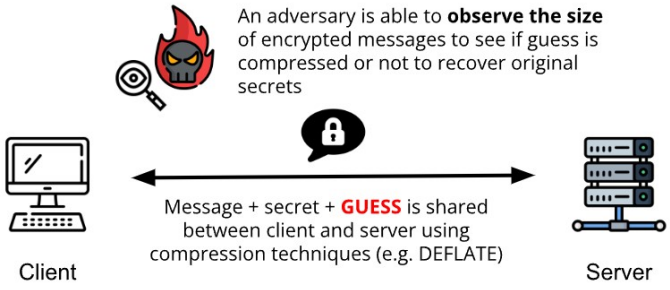

Fig. 12. An attacker can observe the size of encrypted messages

*1) **Overview:** :* Web applications many a times include sensitive information, like CSRF tokens, in their HTTP responses. Attackers can exploit these by taking advantage of the DEFLATE [11] compression algorithm used in HTTP responses, which benefits from repeated strings to shrink the payload. By injecting a "guess" character by character into a URL parameter and observing the response size, an attacker can deduce the CSRF token's characters. For instance, if the first character of the guess matches the first character of the actual CSRF token, DEFLATE compresses the response more efficiently, serving as an oracle for the attacker. This process is repeated to recover the entire CSRF token.

*2) **Prerequisites of BREACH attack:** :* This attack may not always be feasible. To successfully execute this attack, a web application must have the following features:

- A server has to use HTTP-level compression (GZIP / DEFLATE)
- The user-input should be reflected in HTTP response bodies
- There should be secrets (e.g., CSRF token, PII) in HTTP response bodies

It would be extra helpful for an attacker, if the response sizes remain the same most of the time. Because, the size of the responses measured by the attacker can be quite small. Any noise can make the attack difficult. The attack can be executed by any cipher suite. It's easier in the stream cipher than the block cipher. In block cipher, extra steps have to be taken.

*3) **Implementation:** :* BREACH attack is quite similar to CRIME attack with subtle differences. This attack also leverages compression to extract data from a SSL/TLS channel. However, its focus is not on SSL/TLS compression; rather it exploits HTTP compression. Here, the attacker tries to exploit the compressed and encrypted HTTP responses instead of requests as was the case with the CRIME attack. HTTP response compression applies only to the body of the response and not the header. The attack works by injecting data into the HTTP request and analyzing the length of the HTTP responses. Any variation in length of the response indicates a successful guess. The process of BREACH attack is described below:

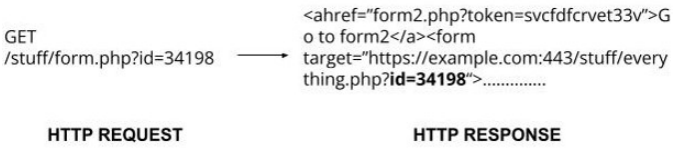

Fig. 13. An example of HTTP get and response request

In Figure 13, user input in http request id=34198 is reflected in the http response that we get and the attacker leverages this very functionality to guess the 'token' value character by character. Here we can see there is some secret token in the http response, and here the protocol uses compression http response technique (**not shown in figure).** So, this protocol can be a vulnerability source for the BREACH attack as it follows all the conditions that are necessary for a BREACH attack.

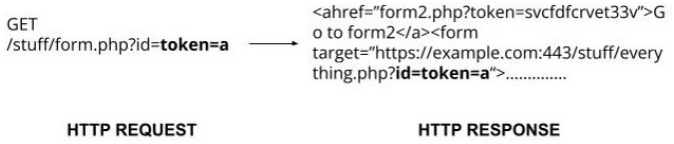

Fig. 14. An attacker trying to inject some parameters

In Figure 14, The attacker injects the token value in the id parameter. If the value of the attacker injected token matches that of the actual token then the length of the response decreases by the length of the duplicate strings due to HTTP compression. In our case, for a successful attack the length will decrease by 20 as the length of the string '`token=svcfdfcrvet33v`" is 20. Initially the attacker only knows the '`token=`' part of the string. This the attacker can know by simply logging into the application himself. The attacker initiates the attack by sending the following value as the value of the token '`token=a`'. The length of the response will decrease by 6 as the value of the length of the following string '`token=`'.

If the token input 'a' would have been correct, the length change should have been decreased by 7. So, the attacker can inspect the length of the response and can verify that the guessed token value was incorrect. The attacker then does the same for different token values. For example, in Figure 15, if the attacker chooses '`token=s`'. The request and response will look like:

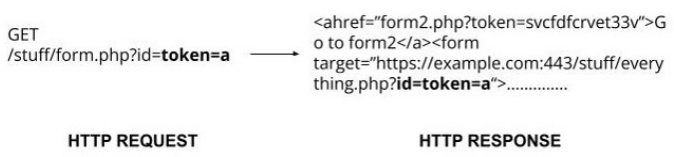

Fig. 15. An attacker injected correct some parameters

Since the first character of the attacker's guess matches the actual value of the token the response length decreases by 7, the attacker concludes that he has successfully guessed the first character of the token. The attacker then keeps the first character constant and varies the second character and repeats the entire process again. The attacker does this until he has successfully guessed the entire value of the token.

4) ***Practicality:*** *:* The BREACH attack can be exploited with just a few thousand requests and can be executed in under a minute [2]. In the case of OWA (Microsoft's Outlook Web Access), we are able to reliably (95% of the time) recover the entire CSRF token, and often in under 30 seconds [17]. Though, the number of requests might depend on the secret size. If the HTTP response body follows all the criteria of prerequisites of BREACH attack, then the system can be vulnerable against it.

5) ***Mitigation Techniques:*** *:* BREACH utilizes the compression technique used in HTTP protocols. So, even if we turn off the TLS compression, it will not make any difference to the BREACH attack, which can still perform chosen-plaintext attacks against the HTTP payload [17]. The techniques [17] [2] [1] that can be used to mitigate BREACH attacks are:

- **Disabling Compression:** If we disable compression at the HTTP level, it will completely eliminate the side-channel, thus the attack can not happen. But, it will have a huge negative impact on the performance.
- **Cross Site Request:** Whenever the referrer header indicates a cross-site request or when the header is not present, the HTTP compression has to be disabled [24].
- **Separating Secrets from User Input:** In this approach, the user input is put into a different compression context than that of application secrets. This way solves the problem completely.
- **Masking Secrets:** Here a one time pad P is generated for each request, embed the secret as P$\|(P \oplus S)$ in the page. It doubles the length of every secret and guarantees that the secret is not compressible.
- **More Aggressive CSRF Protection:** It requires a valid CSRF token for all requests that reflect user input**.**
- **HTB(Heal-the-BREACH):** It is the most effective mitigation technique [23]. It modifies the server-side gzip compression library to add randomness to the size of the response content. This randomness precludes BREACH from guessing the correct characters in the secret token. HTB protects all websites and pages in the server with minimal CPU usage and minimal bandwidth increase.

Last but not least, it is always safe to monitor the traffic to detect any attempted attacks on the network.

D. POODLE Attack

The Padding Oracle On Downgraded Legacy Encryption (POODLE) attack attempts to exploit SSL protocol vulnerability (i.e., CVE-2014-3566) to decrypt sensitive information, such as login credentials or encrypted communication between a user and a website. In this section, we first provide an overview of the attack and its functionality. Next, we elaborate on the existing SSL cryptography vulnerability. Finally, we discuss our implementation and propose the mitigation methods recommended for the POODLE attack.

1) ***Overview****:* SSL, Secure Sockets Layer, is a cryptographic protocol designed to provide secure communication over a network. It ensures the confidentiality and integrity of data exchanged between a client (e.g., web browser) and a server. SSL 3.0, as an outdated and insecure protocol, has largely been replaced by more secure protocols such as Transport Layer Security (TLS) protocol. Several implementations of TLS (e.g., TLS 1.0, TLS 1.1, and TLS 1.2) maintain backward compatibility with SSL 3.0 to ensure smooth interaction with legacy systems and guarantee a seamless user experience. Therefore, in order to work with legacy systems, many TLS clients implement a protocol downgrade dance to work around server side interoperability bugs. In the initial handshake attempt, the client offers the highest version of the protocol. If this initial handshake fails, there is a subsequent attempt (potentially repeated) where earlier protocol versions are tried. This process of downgrading the protocol is different from proper protocol version negotiation. In standard negotiation, if the client proposes a high version like TLS 1.2, the server might respond with a lower version like TLS 1.0. However, in the case of downgrade dance, the downgrade can occur not only due to negotiation but also due to network glitches or intentional interference by active attackers. Thus, the attacker will confine to SSL 3.0. As soon as the protocol is downgraded, the attacker is able to decrypt the SSL session content. The decryption is done byte by byte and will generate a large number of connections between the client and server.

2) ***Details of POODLE Attack (CVE-2014-3566)****:* Encryption in SSL 3.0. utilizes either the RC4 stream cipher or a block cipher in CBC (Cipher Block Chaining). RC4 is acknowledged to have biases, meaning that if the same secret (such as a password or HTTP cookie) is transmitted over multiple connections and encrypted with various RC4 streams, more and more information will gradually leak.

The critical problem with CBC encryption in SSL 3.0. is that its block cipher padding lacks determinism and is not covered by the Message Authentication Code (MAC). Therefore, the integrity of padding cannot be fully verified during decryption.

The vulnerability is most exploitable when an entire block of padding having L-1 arbitrary bytes followed by a single byte with the value L-1 (L is the block size in bytes). When processing an incoming ciphertext record $C_1, C_2, ..., C_n$, (where each C_i represents one block) and given an initialization vector C_0,

the plaintext is initially $P_1, P_2, ..., P_n$ as $P_i = DK(C_i) \oplus C_{i-1}$ (where DK denotes block cipher decryption using the key K). Subsequently, the recipient checks and removes the padding and after verification, MAC is removed. The MAC size in SSL 3.0. CBC cipher suites is typically 20 bytes. Consequently, below the CBC layer, an encrypted POST request will appear as follows:

$$POST\ /path\ Cookie : name = value...body \| 20\text{byte MAC} \| \text{padding}$$

To launch the POODLE attack, the attacker acts as a man in the middle between victim and server. The attacker intercepts and modifies the SSL records sent by the browser in such a way that there is a high chance that the server does not reject the record. If the modified record is accepted, the attacker can decrypt one byte of the message (e.g., cookies). The attacker controls both the request path and the body, enabling them to induce requests in a manner that satisfies the following two conditions:

- The padding fills an entire block (encrypted into C_n)
- The cookies' first unknown byte appears as the final byte in an earlier block (encrypted into C_i).

The attacker then replaces C_n by C_i and forwards the modified SSL record to the server. Usually, the server will reject this record, and the attacker will simply try again with a new request. Occasionally (on average, once in 256 requests), the server will accept the modified record. The attacker then concludes that $DK(C_i)[15] \oplus Cn - 1[15] = 15$, indicating that $P_i[15] = 15 \oplus Cn - 1[15] \oplus Ci - 1[15]$. Therefore the first previously unknown byte of the cookie is exposed and the attacker proceeds to the next byte by changing the sizes of the request path and body simultaneously, ensuring the request size remains the same while the position of the headers shifts. This process continues until the attacker has decrypted as much of the cookies as desired.

*3) **Implementation and mitigation**:* Fig. 16 illustrates the architecture of our implementation. As seen, the attacker is placed as a man in the middle between the victim browser and the HTTPS server. The request generator runs in the victim browser and was implemented as static JavaScript file *(POODLEClient.js)*. The HTTP server serves the static request generator code, and answers requests from the request generator regarding the parameters of the generated HTTPS requests to the attack target server. The Python module *http.server* was used here. The TLS Proxy modifies the intercepted TLS packets of the generated HTTPS requests, and checks the server response. The module *socketserver* was used for TLS proxy. Since TLS proxy and the HTTP server are running on different threads, the Python classes Manager and BaseManager from multiprocessing.managers were used to create object instances that are accessible from different threads and even remotely

In order to perform POODLE, we followed the following steps. First, we degrade TLS protocol usage to SSL 3.0. by disrupting the TLS handshake attempts. Second, we increased the POST body size by one byte until the cipher text extends

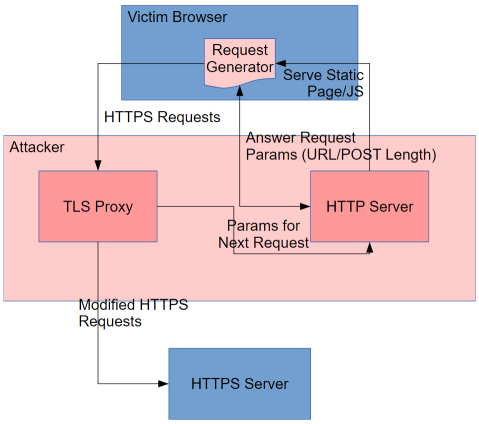

Fig. 16. Enter Caption

by a block size. This justifies the URL and POST length such that the last block of the ciphertext is padding. Finally, we perform the copy operation on every generated TLS packet and calculate the leaked byte if the server accepts the modified packet.

Since current versions of Mozilla Firefox and Google Chrome are safe, we installed Firefox v37.0.0 on our test environment (i.e., Kali Linux). Moreover, in order to make the browser vulnerable to the POODLE attack, we have to change the configuration and enable the SSL downgrade as follows:

$$security.tls.version.min = 0$$

$$security.tls.version.max = 0$$

$$security.tls.version.fallback - limit = 0$$

$$security.ssl3.*\_rc4\_* = false$$

A script `TestHTTPServer.py` was created as attack target server. It forwards the received connections to the HTTP server. The script *POODLE-dev.sh*, starts *httpserver*, *sslserver*, and *attacker-nodebug* components. The attack is performed by requesting the URL `http://localhost:8000` from a vulnerable browser. As a result, the average time required to leak 8 bytes is around 240 seconds.

E. LUCKY 13 Attack

The Lucky 13 attack is a cryptographic vulnerability that exploits the implementation of the Transport Layer Security (TLS) protocol. First discovered in 2013, this attack targets the padding scheme used in CBC-mode ciphers, revealing vulnerabilities in how servers respond to incorrect padding in the decryption process. By capitalizing on the timing discrepancies in error messages generated during this operation, attackers can glean information about the plaintext, potentially compromising the confidentiality of secure communications. The Lucky 13 attack sheds light on the challenges of secure protocol design, emphasizing the constant need for vigilance

and refinement in cryptographic implementations to counter evolving threats.

*1) **Overview**:* Essentially, this attack exploits a timing discrepancy in the processing of Transport Layer Security (TLS) messages. Specifically, it targets the difference in processing speed between TLS messages with at least two bytes of correct padding compared to those with only one byte of correct padding or those with incorrect padding. TLS messages with two bytes of padding are processed marginally quicker, and this variance becomes noticeable through the timing of received TLS error messages.

To exploit this, the attacker sends manipulated ciphertexts to the target location in the plaintext stream across multiple TLS sessions, which in turn triggers an error message in the network.

By conducting this attack repeatedly and meticulously analyzing the resulting responses, the attacker can distinguish between these varying padding scenarios. Subsequently, by examining the timing of the packets, the attacker can initiate a plaintext-recovery attack, focusing on the MAC verification process in the event of malformed CBC padding.

*2) **Prerequisites of the Attack**:* For the LUCKY 13 attack to be successful, several conditions must be met. The target system must be using TLS 1.0, TLS 1.1, or DTLS with block ciphers in CBC mode, the attacker needs to be suitably positioned to intercept and modify network traffic between the client and the server, and they must be able to accurately measure the time taken by the server to respond to different types of modified requests.

*3) **Implementation**:* The implementation of LUCKY 13 involves the following steps described below.

- **Interception and Modification:** Intercepting network traffic and modifying the TLS/DTLS packets, particularly the padding and MAC (Message Authentication Code).
- **Timing Analysis:** Sending these modified packets to the server and measuring the time taken for the server to respond.
- **Statistical Analysis:** Analyzing the timing data to distinguish between correctly and incorrectly padded messages, thereby gradually revealing information about the plaintext.

*4) **Practicality**:* The practicality of the LUCKY 13 attack is limited due to its reliance on precise timing measurements and the need for a considerable number of attempts to gather sufficient data. However, in scenarios where an attacker can obtain accurate timing information and can repeatedly interact with the server, this attack could be practical and dangerous.

*5) **Mitigation Techniques**:* To mitigate the LUCKY 13 attack, several measures can be taken. Implementing constant-time algorithms for MAC and padding checks to eliminate timing discrepancies, upgrading to TLS 1.2 or higher, which supports more secure cipher modes like Galois/Counter Mode (GCM) that are not vulnerable to this attack, and applying patches provided by vendors and correctly configuring TLS/DTLS implementations to avoid fallback to vulnerable versions or configurations are some of the most important solutions.

F. Drown Attack

DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) is a security attack that exploits vulnerabilities in SSLv2 to decrypt RSA encrypted connections. It targets servers that still support the obsolete and insecure SSLv2 protocol, even if they primarily use newer TLS versions. Many servers have SSLv2 enabled for backwards compatibility.

*1) **Overview:** :* These days most modern servers don't support SSLv2 and use other protocols like TLS. Yet in 2016, out of 36 million HTTPS servers, 6 million support SSLv2 [8]. The attacker takes advantage of SSLv2 handshake in order to decrypt the key ciphertext.

2) SSLv2 handshake: In SSLv2 handshake, the client sends clienthello message and receives ServerHello message. Client then sends masterKey encrypted with server's key. At this point the server encrypts the ciphertext and if the format of the message was valid, encrypts new message with the symmetric key and sends it to the client. In case of not valid, It generates a random key and encrypts the message with that. This allows the attacker to figure out the validation of the format by sending a message twice. If the keys for the responses are the different, it means server, generated two different random keys, so the format was not valid.

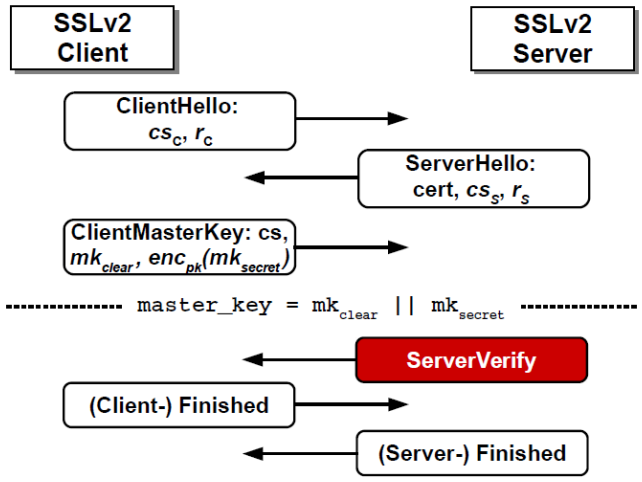

Fig. 17. SSLv2 connection handshake [8]

In the Drown attack, attacker uses SSLv2 protocol in order to break TLS connection. As shown in Figure 18, the client which doesn't support SSLv2 sends a request to a server using TLS protocol. Although this connection is through TLS, server might support SSLv2 too. Or in some cases there might be another server supporting SSLv2 that shares the same key with the former server. In this case the attacker intercepts and captures the traffic between the client and the server. Then it sends the traffic to the SSLv2 server multiple times, each time with a little of modification in order to

collect some information.Using this information it can then decrypts the RSA ciphertext.The decryption step is done using bleichenbacher attack.

3) Special Drown: except for the general Drown attack,

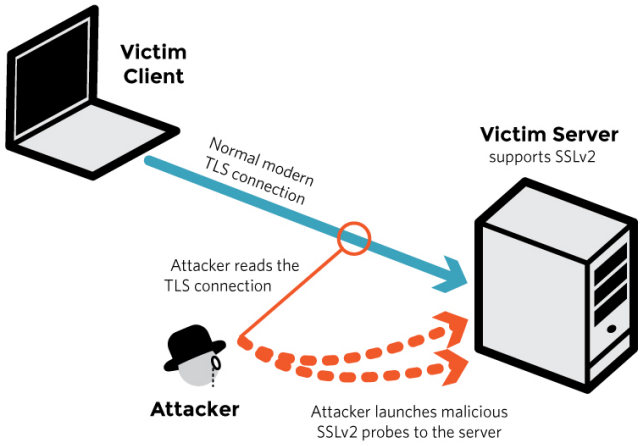

Fig. 18. Attacker uses SSLv2 to break TLS [8]

4) Details of decryption using Bleichenbacher: bleichenbacher's padding oracle attack is an adaptive chosen ciphertext attack against PKCS#1 v1.5,the RSA padding standard used in SSL and TLS. It enables decryption of RSA ciphertexts if a server distinguishes between correctly and incorrectly padded RSA plaintexts, and was termed the "million-message attack".

As shown in Figure 19, in PKCS#1 v1.5 encryption padding data being encrypted has a specific format. When data is being sent to the server, it decrypts the data and validates the format. If the format has a problem the server returns 1, in case of proper format it returns 0.

00	02	Padding String	00	Data Block

Fig. 19. PKCS#1 v1.5 block format for encryption [28]

In other words the valid format would be: $2B \leq m \leq 3B-1$ where $B = 2^{8*(L(m)-2)}$.

The attacker starts with ciphertext c0 and sends it to the oracel server. Based on the result it modifies the ciphertext step by step so that the possible solutions set become smaller. The new ciphertext would be:

$$c = (c_0 \cdot s^e) \mod N = (m_0 \cdot s)^e \mod N$$

By receiving 1, the attacker modifies ciphertext again. Otherwise he can deduce for some value r, $2B \leq m_0 s^{-r} \mod N < 3B$. And can figureout the possible range for m as below:

$$(2B + rN)/s \leq m_0 < (3B + rN)/s$$

Despite bleichenbacher that attacks TLS directly with key length of 384 bit, Drown attacks deals with short secret for export grade crypto which means the key length is 40 bits. Also in TLS, the server chooses type of cipher suites, so we don't have information about exact length of the key.

5) Implementation:

- **Interception and record:** Intercept network traffic and capture about 1000 TLS connections.
- **Morph TLS connection :** The connection captured is not compatible with SSLv2 oracel and it can not decrypt it. So it is needed to find SSLv2 format for the ciphertext.
- **decrypyt the key using bleichenbacher:** send modified ciphertext multiple times in order to narrow down the possible answers and continue till only one solution remains.

6) Mitigation: There are ways to mitigate the risk of this attack:

- Disable SSLv2 entirely: This prevents exploitation of SSLv2 vulnerabilities. Disable SSLv2 on both clients and servers where possible.
- Check public key reuse: Use different public/private key pairs for SSLv2 and TLS if both are required. Special DROWN relies on reusing keys.
- Patch OpenSSL: OpenSSL prior to 1.0.2f and 1.0.1r are vulnerable. Upgrade to newer patched OpenSSL versions.

REFERENCES

[1] Breach. Wikipedia. https://en.wikipedia.org/wiki/BREACH.
[2] Breach attack website. https://www.breachattack.com/.
[3] Crime. Wikipedia. https://en.wikipedia.org/wiki/CRIME.
[4] Cve-2012-4929.
[5] Spdy: An experimental protocol for a faster web. Chromium Developer Documentation. Retrieved 2009-11-13.
[6] Crime attack: Compression ratio info-leak made easy. Threatpost, 2012. Retrieved from https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-77006/.
[7] Acunetix. History of tls/ssl - part 2, Year of publication.
[8] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. {DROWN}: Breaking {TLS} using {SSLv2}. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, 2016.
[9] G.V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on ssl. In *SECRYPT*, pages 7–10, 2006.
[10] W. Dai. An attack against ssh2 protocol. Email to the `ietf-ssh@netbsd.org` email list, 2002.
[11] P. Deutsch. Deflate compressed data format specification. Technical report, RFC 1951, RFC Editor, 1996.
[12] T. Dierks. The tls protocol version 1.0 – rfc 2246. Technical report, IETF Request For Comments, 1999.
[13] Thai Duong and Juliano Rizzo. Here come the ⊕ ninjas. Unpublished manuscript, 2011.
[14] M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. Technical report, Technical report, DTIC Document, 2001.
[15] Rafael Fedler. Padding oracle attacks. *Network*, 83, 2013.
[16] A. Frier, P. Karlton, and P. Kocher. The ssl 3.0 protocol. Technical Report 18:2780, Netscape Communications Corp, 1996.
[17] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. https://www.breachattack.com/, 2013. Unpublished manuscript.
[18] IETF. The tls protocol version 1.0, 1999. RFC 2246.
[19] IETF. Prohibiting secure sockets layer (ssl) version 2.0, 2011. RFC 6176.
[20] IETF. Deprecating secure sockets layer version 3.0, 2015. RFC 7568.
[21] IETF. The transport layer security (tls) protocol version 1.3, 2018. RFC 8446, Section 1.1.

[22] John Kelsey. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[23] Rafael Palacios et al. Htb: A very effective method to protect web servers against breach attack to https. *IEEE Access*, 10:40381–40390, 2022.

[24] Alfredo Pironti and Nikos Mavrogiannopoulos. Length hiding padding for the transport layer security protocol. Technical report, Internet-Draft draft-pironti-tls-length-hiding-00, IETF Secretariat, 2013.

[25] Juliano Rizzo and Thai Duong. The crime attack. Retrieved September 21, 2012, via Google Docs: https://docs.google.com/document/d/1l31tFHbUxZcblU1DWXq3nD4BG8bMr9pKu6JHHNz3m2g/pub, 2012.

[26] Pratik Guha Sarkar and Shawn Fitzgerald. Attacks on ssl: A comprehensive study of beast, crime, time, breach, lucky 13 & rc4 biases. https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf, 2013. [June, 2014].

[27] Ashutosh Satapathy and Jenila Livingston. A comprehensive survey on ssl/tls and their vulnerabilities. *International Journal of Computer Applications*, 153(5):31–38, 2016.

[28] Gao Si, Chen Hua, and Fan Limin. Padding oracle attack on pkcs# 1 v1. 5: Can non-standard implementation act as a shelter?

[29] Preeti Sirohi, Amit Agarwal, and Sapna Tyagi. A comprehensive study on security attacks on ssl/tls protocol. In *2016 2nd international conference on next generation computing technologies (NGCT)*. IEEE, 2016.

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