Survey on Oracle Padding Attacks on Cryptographic Protocols

Anab Leila Abdi
CIISE, Concordia University
Montreal, CA

Tomas Navarro Munera
tomas.navarro@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Jainil Shah
jainil.shah@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Bergen Davis
bergen.davis@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Diego Jaramillo
diego.jaramilloalfonso@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Md Ashfaque Rahman
mdashfaque.rahman@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Alcibiades Rogelio Pumajulca
alcibiades.pumajulcasalazar@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Shrutibahen Rana
shrutibahen.rana@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

Juan Soto Chourio
juan.sotochourio@mail.concordia.ca
CIISE, Concordia University
Montreal, CA

***Abstract*—Cryptographic protocols play an essential role in protecting information across diverse applications and systems. These systems range from finance, e-commerce, transportation, shipping etc. The integrity and reliability of these protocols are crucial for safeguarding the confidentiality of the data they facilitate. While modern-day protocols are generally effective in securing systems, there are instances where they can fall victim to malicious actors. These actors employ specially crafted techniques and tools to exploit various design flaws within the definition of the protocols libraries, its implementation etc, effectively bypassing the security mechanisms of cryptographic protocols.**

The Oracle Padding Attack, along with its variants, serves as an example of assaults on cryptographic protocols, padding oracle attacks enable the retrieval of either partial or full content of the original plain-text from encrypted messages. These attacks also impact the prevalent operational method of contemporary cryptographic protocols, particularly the cipher block chaining (CBC) mode. Consequently these attacks have the potential to compromise nearly all online communication channels that rely on such protocols for security In this survey report, we investigate and analyze the Oracle Padding Attack and some of its variants. Our investigation delves into the background information of these attacks, their implementation details, and effective solutions against them

***Index Terms*—Cryptographic protocols, Oracle Padding Attack, systems**

I. Introduction

In today's modern society, a multitude of highly important operations are carried out over the internet or some electronic medium. This dependency on the digital era has correlated with an increase in cyber crimes across the technological industry. Therefore, when communicating, there is a demand for the confidentiality and integrity of components and properties of sent messages over these channels. In the case of the internet, the TLS/SSL protocol has been the conventional standard to secure communication. However, this protocol, among others, is not fully resistant to penetration. Actors can use specific implementations to penetrate and retrieve full or partial recovery of original plain-text messages. These attacks take advantage of unintended side channels revealed by cryptographic protocols. Consequently, they create an oracle that allows making inferences about the underlying plain-text by utilizing easily predictable padding bytes. This is why they are referred to as padding oracle attacks [1]. As a consequence, SSL/TLS protocols are not impervious to oracle attacks.

The SSL/TLS protocol has seen a significant change in its security and overall architecture over the last few years, culminating in its termination in 2015. The protocols were rolled out into the market in the early 1990s. However, as time progressed, the protocols were plagued by their insecure design and numerous vulnerabilities. Subsequently, the TLS standard was introduced as the successor to SSL, iterating over versions TLSv1.0 until today's current standard of TLS1.3, which has mitigated vulnerabilities that plagued its predecessors [2]

As the SSL/TLS version underwent enhancements for increased security, the Oracle padding attack progressed in sophistication and diversified into various forms. This survey delves into the Oracle padding attack and its multiple variants, presenting an examination of eight attacks

on the SSL/TLS protocol, encompassing BEAST, POODLE, DROWN, CRIME, BREACH, LUCKY13, ROBOT and SCP02 Attack.

II. Background

The oracle padding attack, introduced in the early 2000s, has proven to be an intricate attack that operates by utilizing the oracle padding algorithm to carefully examine the integrity of padding on plain-text when given any cipher-text. The padding oracle algorithm was designed to identify cases of valid padding. It operates on the condition that it returns a positive result if the padding is correct; however, it will return a negative result if incorrect. As is common with many standardized formats, this introduces a vulnerability for malicious actors to apply investigative techniques to determine the padding byte of a specific message, which directly results in the collapse of the system and the exposure of the plain-text message [3].

The oracle padding attack is a key attack within the realm of cryptographic attacks, showcasing its prominence by its ability to exploit a server's behavior during the decryption process of messages. Attackers are able to deduce sensitive information through side channel data such as prolonged decryption cycles or error code displays. [3]

In the case of the oracle padding attack, we have designed a scenario where the attacker retrieves the cipher-text through his participation in a man-in-the-middle attack. The server functions as a padding oracle, validating the padding and issuing error messages when incorrect padding is detected. The attacker will perform a series of tests on each byte of the cipher-text, iterating through the set of 0 - 255 until the server validates the padding. Once successfully validated, the attacker will perform XOR computations to reveal the last byte of the plain-text. This process is repeated iteratively to reveal preceding bytes in each instance [3]. The vulnerability within the padding oracle attack underscores the need for well-designed protocols that are regularly revised to protect against security infractions.

A. Padding Oracle History and Emergence of Variants

The earliest version of the padding oracle attack can be traced back to 1998, demonstrated in Bleichenbacher's attack, an adaptive cipher-text attack that victimizes RSA with PKCS 1 v1.5 padding [4]. However, this attack was inefficient and ineffective. This improved in 2002 when Vaudenay made a significant breakthrough in symmetric cryptography, later known as the padding oracle attack [5]. Over the course of the next decade, many faults were discovered in the implementation of security protocols such as SSL and TLS. Regrettably, these weaknesses provided opportunities for cyber assaults to pilfer vital user credentials. Notably, widely-used encryption methods such as CBC are particularly susceptible to such breaches. Surprisingly, the robustness of the encryption doesn't consistently offer protection, as certain attacks can prevail even without the attacker having knowledge of the encryption key. Neglecting this led to the promotion of several variants of the oracle padding attack. These variants include BREACH [6], POODLE, ROBOT[7], LUCKY13, BEAST, and CRIME [**?**, b8]

B. Overview of Cryptography

In this section, we provide an overview of fundamental cryptography concepts and terminology to assist readers in understanding attacks more comprehensively.

Encryption and Decryption: Encryption is the process of converting plain-text information into a secret code (cipher-text) to prevent unauthorized access. It is a crucial technique in cryptography for securing sensitive data during storage or transmission. [55] Plain-text messages, denoted as 'm', can be sequences of bytes of any length (m E 0, 18n). The corresponding cipher-text, denoted as 'c', consists of byte sequences that are multiples of 'b', known as the block size. During encryption, a mathematical algorithm (encryption algorithm 'E') and a secret key 'k' are used to transform plain-text into cipher-text. An example of the encryption process is as follows:

$$E(m, k) = c.$$

Decryption involves the reverse process of converting encrypted cipher-text back into its original and readable form (plain-text). It is the inverse operation of encryption and requires the use of a key or algorithm to transform the cipher-text back into its original content [55]. During decryption, a mathematical algorithm (decryption algorithm 'D') and the secret key 'k' are used to convert the cipher-text into plain-text. An example of the decryption process is provided below.

$$D(c, k) = M$$

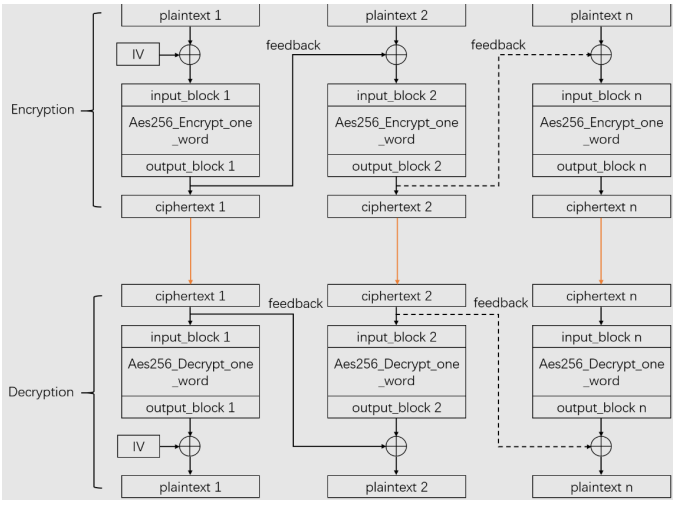

Fig. 1. Overview of CBC Encryption and Decryption [36]

C. SSL/TLS

The vulnerability leading to maximum padding oracle attacks stems from flaws in the construct of SSL or TLS protocols. Let's delve briefly into these cryptography protocols:

SSL (Secure Sockets Layer) and its successor, TLS (Transport Layer Security), were devised to fortify internet communications by ensuring privacy, authentication, and data integrity. Introduced by Netscape in 1995, SSL aimed to counter the risk of transmitting data in plain-text over the web, vulnerable to interception and unauthorized access. [30]

SSL and TLS are pivotal in securing online transactions and communications. Websites utilizing SSL/TLS display URLs starting with "HTTPS" instead of "HTTP," indicating a secure connection. These protocols encrypt transmitted data, rendering it unreadable to unauthorized entities. This encryption is especially crucial in safeguarding sensitive information like credit card details from interception and theft.

A significant feature of SSL/TLS is the authentication process initiated through a handshake between communicating devices. This handshake verifies the identity of both the client (user's device) and the server. SSL/TLS also employs digital signatures to ensure data integrity, affirming that transmitted information remains unaltered during transit.

SSL evolved into TLS over time, with the Internet Engineering Task Force (IETF) proposing updates in 1999. This transition in naming marked a change in ownership and development. Though SSL and TLS are often used interchangeably, TLS is acknowledged as the more contemporary and secure version. [30] TLS serves three primary functions: encryption, authentication, and integrity. Encryption shields data from unauthorized parties, ensuring confidentiality. Authentication validates the identities of involved parties, thwarting unauthorized access. Integrity ensures that transmitted data remains unchanged during communication. The TLS handshake, pivotal to the protocol, establishes a secure connection between the client and the server. It involves specifying the TLS version, selecting cipher suites for encryption, authenticating the server through its TLS certificate, and generating session keys for secure communication. [30]

Public key cryptography is instrumental in TLS, facilitating secure key exchange over an unencrypted channel. The server's public key, part of its TLS certificate, is utilized in authentication. Once data is encrypted and authenticated, it undergoes further protection via a message authentication code (MAC), ensuring data integrity akin to a tamper-proof seal on a product.

It is noted that, TLS-protected HTTPS has become standard practice for websites, with major browsers reinforcing security measures by highlighting non-HTTPS sites. The adoption of TLS encryption remains pivotal in shielding web applications from data breaches and cyber threats. [30]

III. The Padding Oracle Attack

The Padding Oracle attack exploits weaknesses in modern cryptography protocols and systems. The attack capitalizes on differences in error messages that occur during the decryption of cipher-texts. Through the investigation and analysis of these errors, malicious attackers can deduce critical data surrounding the validity of padding, allowing them to decrypt and compromise the security of the encryption process. The existence of this attack sheds light on the obstacles and challenges in the realm of securing encrypted communication systems and showcases the importance of robust cryptography practices. [9]

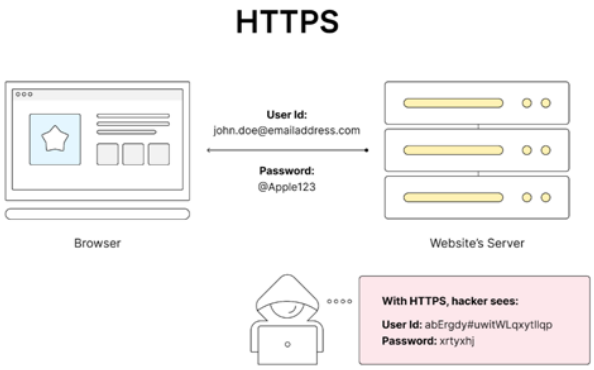

Fig. 2. Overview of HTTP/S [29]

A. Prerequisites for the attack

The success of the Padding Oracle Attack hinges on meeting specific conditions. Typically, the attacker needs access to encrypted communication and the Oracle. This access enables the attacker to discern the validity of the padding generated by the current plain-text. With this crucial information, the attacker can decrypt and reveal the plain-text, regardless of the encryption's strength or the selection of robust keys. [9] The prerequisites for executing the padding oracle attack encompass:

- Availability of an Oracle: Successful execution requires the attacker to have access to a padding oracle, a mechanism indicating whether the decrypted message's padding is correct. This could manifest as subtle error messages or other detectable behaviors.
- Cipher Block Chaining (CBC) Mode: This attack is particularly effective when encryption employs Cipher Block Chaining (CBC) mode, leveraging the interconnection of blocks to exploit padding.
- Familiarity with Padding Schemes: Understanding the padding scheme used in the cryptography protocol is essential for the attacker. Various schemes exhibit distinct error patterns that can be exploited. In our demonstration, we'll utilize the PKCS7 padding scheme.

B. Attack Analysis

The Padding Oracle Attack can target symmetric encryption using CBC mode, which employs the PKCS7 padding scheme. Initially, we'll delve into the mathematical principles underpinning the attack. Then, we'll explore the step-by-step decryption

process for one byte, followed by subsequent bytes and blocks. Our approach assumes that we can submit any cipher-text to the server, which will indicate whether the decryption yields plain-text with valid padding. [3]

a) Mathematical Foundation: In the encryption method known as Cipher Block Chaining (CBC), every block of plain-text goes through an operation called XOR (exclusive OR) with the preceding block of cipher-text before it enters the cipher. Thus, when decrypting in CBC mode, each cipher-text block is decrypted by the cipher and then subjected to XOR with the preceding cipher-text block to reveal the original plain-text.

The decryption process for a specific plain-text block, say Pn, necessitates the presence of both cipher-text blocks, Cn and Cn-1. To elucidate, to uncover P2, access to both C2 and C1 is imperative. In practical terms, C2 undergoes an initial transformation into an intermediary form denoted as I2 through the utilization of the block cipher decryption function in conjunction with the key.

The Padding Oracle attack operates by computing an temporary intermediate representation for each of the cipher-text.

This representation holds significance as it facilitates decryption. Understanding I2 is crucial because once we have it, we can derive P2 using the following transformation:

$$I_2 = C_1 \oplus P_2$$

This equation implies:

$$P_2 = I_2 \oplus C_1$$

This equation serves as a compelling motivation to uncover the intermediate representations, which ultimately facilitate complete decryption. In our scenario, since C1 is already known as part of the cipher-text, determining I2 will enable us to readily deduce P2.

b) Computation of the Last Byte and the Other Bytes of a Block: Previously mentioned, our aim is to send any ciphertext to the server and determine if the decrypted plaintext's padding is valid. This grants us the ability to submit any ciphertext to the server. Let's consider the ciphertext we're going to submit to the server as $C'_1 + C_2$. Here, C'_1 is the carefully crafted cipher block, and C_2 is the block we aim to decrypt. Initially, we designate the first 15 bytes as random, while the 16th is the byte we'll manipulate, trying values from 0 to 255. Our objective is to ensure that after decryption, the last byte of the plaintext will be valid padding.

Therefore, an iterative process takes place and so forth until the oracle confirms valid padding.

Upon determining $C'_1[16]$, alongside the knowledge that $P'_1[16] = 0x01$, we can employ the discussed equation to compute $I2[16]$, which subsequently helps determine $P2[16]$.

$$P1[16] = C1[16] \oplus I2[16] = C1[16] \oplus (C'_1[16] \oplus P'_2[16]) \quad (2)$$

This allows us to retrieve the last byte of the $P1$ block without compromising the cipher's integrity, showcasing the efficacy of the Padding Oracle attack.

Finding the other bytes of a block follows a similar process to finding the last byte. This time, the first 14 bytes, $C'_1[1..14]$, are randomized, and $C'_1[15]$ is the byte we'll iterate over with values from 0 to 255. By performing back calculations, we can deduce $C'_1[16]$.

To ascertain $C'_1[16]$ for finding the second byte, we utilize our prior knowledge of $I2[16]$ from the preceding step. Additionally, given that for valid padding, $P2[15, 16]$ must be $0x02$ according to the PKCS7 padding scheme, we determine $C'_1[16]$ as:

$$C'_1[16] = P'_2[16] \oplus I2[16] = 0x02 \oplus I2[16]$$

Subsequently, we construct C'_1 accordingly, iterating on $C'_1[15]$ while verifying with the oracle until valid padding is achieved. Once valid padding is confirmed, we apply Equation 2 again, this time for the 15th byte.

c) Computation of the Final Blocks: The requirement for a specific plaintext block, P_n, mandates the availability of both the ciphertext blocks, C_n, and C_{n-1}. This creates a challenge in uncovering the initial plaintext block, P_1, as it's typically encrypted using an initialization value, IV. However, in CBC encryption mode, particularly prevalent in TLSv1 and earlier versions, the last cipher block of the preceding ciphertext is commonly employed for encrypting the subsequent plaintext. Consequently, an eavesdropping attacker, having monitored the encryption channel over a prolonged period, could potentially discern the IV, as it essentially corresponds to the last block of the previous ciphertext.

In contrast, in more contemporary TLS/SSL iterations, this approach would prove ineffective. Thus, the attacker might resort to speculative attempts, such as employing sequences like [0,0,0,...]. Should such efforts fail, decrypting the initial plaintext block wouldn't be feasible, even employing a padding oracle attack.

d) Effectiveness of the Attack: Padding oracle attacks, once hindered by challenges like restricted system access and the intricacy of execution, have gained practicality over time, thanks to advancements in attack methodologies and cryptographic countermeasures. Despite initial hurdles, the emergence of cloud computing and sophisticated penetration testing tools has empowered attackers with more streamlined avenues to exploit padding vulnerabilities.

A noteworthy and somewhat alarming aspect of Padding Oracle attacks is their capability to succeed even against robust cryptographic protocols, even in scenarios where meticulously chosen keys are employed. As technology progresses, the persistence of padding oracle attacks remains evident, particularly in systems reliant on outdated or inadequately maintained security measures. Therefore, grasping the historical and contemporary practical dimensions of padding oracle attacks holds paramount importance in devising robust defense strategies amidst the evolving cybersecurity landscape.

IV. Variants of the Attack

A. Padding Oracle Attack Against the SCP02 Protocol

The Secure Channel Protocols (SCP) specified by GlobalPlatform aim to secure communication between smart cards (or other secure elements) and external entities (off-card entity) like card readers and servers, some of these specifications are: SCP01, SCP02, SCP03, SCP04, SCP10, SCP11. In this section, the focus is on SCP02 which until today is used and improves upon SCP01 by introducing counter measures against replay attacks and providing more robust security features. SCP02 (based on 3DES) uses dynamic session keys derived from a base key and counters.

Section Outline: Part 1 describes the SCP02 protocol in detail. Part 2 describes the regular padding oracle attack. Part 3 details how to use the Padding Oracle Attack against SCP02. Part 4 specifies the experimental settings. Part 5 lists counter measures and conclusion of the section.

Notation: A byte value is written as 7E. The value 00^i corresponds to the byte string made of i bytes equal to 00. $b_i|b_{i+1}$ refers to the concatenation of the bytes b_i and b_{i+1}. C || B refers to the concatenation of the two DES blocks C and B. ENC indicates a symmetric encryption function, while ENC^{-1} indicates the corresponding decryption function.

1) GlobalPlatform SCP02:
The protocol SCP02 is based on symmetric-key algorithms, and it aims at establishing a secure link between an "off-card entity" and a card [24]. The card and the party involved in the communication with the card (from now on "the server") share one or several sets of symmetric keys. Three keys make a set, from this set the session keys are computed every time a new channel is established. The card manages a sequence counter related to a given keyset with an initial value of 0, this value is incremented after each successful session. After the sequence number reaches its maximum value, the card should not start a session with that keyset. The commands sent by the server and the responses sent by the card are encrypted and protected with a MAC tag; although, this depends on the security level negotiated during the key exchange. The lowest security level is data integrity (regarding the commands) and only data encryption in not allowed. The plaintext is padded with a fixed string of bytes [26] where a byte equal to 80 is appended to the plaintext, then it's added as many null bytes as necessary, so the string length is multiple of a DES block. For data encryption is used 3DES in CBC mode with a null IV [25]. The MAC (8 byte) tag is computed on the command header, the plaintext, and a padding data. From figure 3 taken from [27]: The genuine header HDR can be retrieved from the header HDR' of the encrypted command. Besides, IV used for the next command is equal to the MAC tag computed on the previous command. The ciphertext and the MAC tag become then the data field of the server's command.

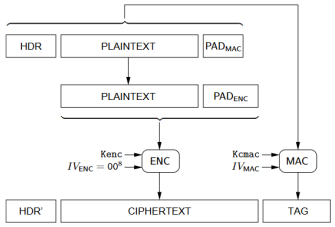

Fig. 3. Encryption and MAC computation of a command data with SCP02

2) Padding oracle attack:
In Algorithm1 the attack is described, and it is assumed that the targeted block C includes at least one byte of padding, i.e. 80. When the plaintext (to retrieve) corresponds to more than two blocks $B_0, ..., B_{k-1}, k > 2$, and the encrypted blocks are $C_0...C_{k-1}$ after the attacker applies Algorithm1 to each pair of encrypted blocks $(C_{k-2}, C_{k-1}), (C_{k-3}, C_{k-2})...(C_0, C_1), (C_{-1}, C_0)$ where C_{-1} is the CBC IV (equal to 00^8 in SCP02). For the rest of this section, it is assumed the ciphertext is made of two blocks V, C. The padding length can be found using the dichotomous algorithm proposed in [27] (which complexity is $log_2(d)$ where d is the block size), or a linear search by testing all bytes starting from the rightmost one until the decryption of the modified block yields a valid padding (this method finds the length in min(d, h + 1) steps where h is the padding length). Although, the padding length helps to shorten the duration of the attack, the attack can also retrieve the padding data as any other unknown plaintext byte.

Algorithm1 - Regular Padding Oracle Attack

FindBlock(V, C)
for i = n - 1 down to 0 **do**
  $b_i \leftarrow$ **FindByte**(b_{i+1}) // b_n = 80
end for
return b_0 ... b_{n-1}

FindByte(b)
for g = 00 to FF **do**
  V'$\leftarrow$ in V replace $v_i|v_{i+1}$ with $(v_i \oplus g \oplus 80)|(v_{i+1} \oplus b)$
  send V' ‖ C to the oracle O
  r $\leftarrow$ O(V'‖ C)
  if r = 1 **then**
    return g
  end if
end for

3) Attack applied to SCP02:
When the ciphertext (valid) V‖C is changed to V'‖C, the MAC verification returns an error since the ciphertext has been modified (whether the padding is valid or nor).

This generates that the smart card outputs an error code, this error code is the same regardless of the validity of the padding data. Hence, the authors [27] used the time spent by the smart card during the decryption procedure to build the oracle O, their experiments show that the card's response time (which reflects the card's computation time) when the padding data is valid is higher compared to the case when the padding data is invalid, so this suggested that the MAC tag is not verified in the latter case. This timing discrepancy allows the attacker to deduce information about the plaintext byte-by-byte, so it is clear that the attack mechanism is *timing channel*. [5]

The attack is described in algorithm 2 taken from [27], so when looking for a byte bi for each value of g, a modified ciphertext V' ‖C is sent to the targeted smart card. Each response time is collected until the guess is correct (then the trials are stopped) and the algorithm continues to byte bi-1; otherwise, g is assigned another value to test. We need to precise that for a given value g, the Stop procedure returns true as soon as a response time is lower than t_min or if the number of successive response times higher than t_min reaches K_R (the number of attempts necessary to detect a right guess). The Correct procedure returns true if the number of successive response times higher than t_min is equal to K_R. Besides, when the smart card receives a message with an invalid MAC tag (this happens when we change V ‖C into V'‖C) a cryptographic error occurs on the smart card side and the session is stopped; then a new session is started and a new cyphertext V ‖C (for the same plaintext) is obtained, so we have to consider the bytes already found and the change in V.

Algorithm2 - Padding Oracle Attack based on the card response time.

FindBlock
for i = n - 1 down to 0 **do**
  $b_i \leftarrow$ **FindByte**$(b_{i+1}...b_n)$ // b_n = 80
end for
return b_0 ... b_{n-1}

FindByte$(b_{i+1}...b_n)$
for g = 00 to FF **do**
  j = 0
  repeat
    get a new ciphertext $V||C$
    V'$\leftarrow$ in V replace $v_i|v_{i+1}...|V_n$ with $(v_i \oplus g \oplus 80)|(v_{i+1} \oplus b_{i+1})...|(V_n \oplus b_n)$
    send $R_0||...||R_{m-1}||V'||C$ as encrypted data to the smart card
    $t_j \leftarrow$ response time
    j = j + 1
  until $Stop(t_0, ..., t_{k-1})$ = true
  if Correct$(t_0, ..., t_{k-1})$ = 1 **then**
    return g
  end if
end for

4) Experimental Settings

The experimental setting in [27]: First, the attacker is located between the remote server and the card at a point where he can directly eavesdrop on SCP02 encrypted commands and send modified commands to the card. Second, the attacker can distinguish response times from valid and invalid padding. Third, the server repeatedly sets up a new secure channel with the card. Forth, the same message (secret information) is sent through every new secure channel.

Real use-case: Upload an applet into a smartphone's Sim Card/UICC through the SCP02 channel established between a remote server and the Sim Card by using the intermediary of an application on the smartphone. Furthermore, the SCP02 commands that carry the applet are embedded in a second secure channel (TLS or SCP80) established between the remote server and the mentioned application. The application sends the decrypted data (SCP02 commands) to the Sim Card. As a result, it is possible to send a command (STORE DATA) to upload an applet which may carry secret data (symmetric key). The attacker sits between the application and the smart card, and by using a Trojan (that escalates privileges) he can break the SCP02 channel and retrieve sensitive data; then, by this point the attacker is able to craft SCP02 commands sent to the Sim Card (process initiated by the legitimate application), the attacker makes sure that the valid application asks the server for a new delivery of the same message until this command is acknowledge by the Card. It is worth noting that the Trojan could work by getting access to the memory space of a legitimate application [50], escalate privileges to root access or inject malicious code into system libraries that execute processes with systems rights. After forging the SCP02 command, the attacker measures the elapse time until the Sim Card sends a response, so the attacker knows if his guess (g) is correct or not, but it can be used statistical tools to find patters in timing [51].

5) Countermeasures and Conclusion:

In [52], the author proposes that the MAC should be computed on the padded data (pad the plaintext); later, the MAC should be verified before removing the padding data. Another simple fix is to change the byte-oriented padding of the form 80 00i into a bit-oriented padding of the form $10 \cdots 0$ [53], this makes the padding oracle unfeasible. A different approach is to make responses time-invariant by simulating a MAC verification [54] (including when a padding error is detected); in general, it is suggested that the implemented decryption procedure should try to eliminate all timing channels. Finally, In [27] the authors suggest to use the command

PUT KEY (part of the smart card commands) to send secret values to the smart card; since the data field of such command corresponds to the output of a double encryption process in which: first, we encrypt with a session key different than the one used to encrypt and MAC the other commands, then encrypt with these same secure channel session keys. Accordingly, an attacker could retrieve data encrypted with this additional session key, but not the real plaintext. Finally, it is convenient to limit the number of times a server can send the same secret value to the smart card. This section has shown the feasibility of attacking the SCP02 protocol with the use of a timing channel exploitation based on the padding oracle attack; as a result, several counter measures have been made to mitigate the attack; otherwise, it is recommended to use more secure protocols such as SCP03.

B. BREACH Attack

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) is an instance of the CRIME attack against HTTP compression. It was first announced in 2013 at Black Hat USA conference by Angelo Prado, Neal Harris and Yoel Gluck [59]. The attack is a side-channel attack to HTTPS that targets HTTP compression with the aim of disclosing secrets such as CSRF tokens and victims' credentials under certain conditions [59] [60].

*1) **Overview**:* CRIME attack targeted HTTP requests, while BREACH attack targeted HTTP responses [6]. The initial CRIME attack focused on request headers and relied on TLS compression, so by disabling it, the attack was mitigated. However, this mitigation method did not work for BREACH. BREACH attack used the same concept as CRIME, but focused on secrets that are within HTTP response bodies. HTTP responses are compressed using the common HTTP compression, which allows the attack to be carried out without relying on TLS-level compression and without tampering with or downgrading SSL [37].

HTTP compression is based on DEFLATE which works by eliminating repetitions in strings of text. The more repetitions, the more potential there will be for compression to reduce the overall size [58]. It uses a combination of LZ77 and Huffman Coding. LZ77 reduces redundancy by replacing occurrences of three or more characters with pointer values to reduce space. Huffman Coding, on the other hand, replaces the more common bytes with shorter bytes, usually symbols, to optimize the description of the data to the smallest size possible [61]. BREACH works by attacking the LZ77 compression while minimizing the effects of Huffman Coding.

It is common for web applications to not only deliver secrets such as CSRF tokens in the HTTP response body, but also reflect user input such as URL parameters within the response body [40]. Since DEFLATE takes advantage of repeated strings for compression, an attacker can guess the secret, one character at time, using the reflected URL parameter in the response body. For instance, assuming the first character of the attacker's guess matches the first character of the CSRF token, DEFLATE compresses the response more efficiently, and serves as an oracle for the attacker. The attacker is then able to repeat this process to recover the entire CSRF token.

While LZ77 makes this attack easy, Huffman Coding presents a challenge for the attacker. When you replace common bytes with shorter sequences, it ultimately compresses the overall size to something smaller. The oracle in this case becomes confused as to whether the overall size is smaller due to LZ77 which would indicate a match with the attacker's string, or if the smaller size is as a result of Huffman Coding since the character is very common in the response. If isolation between the two components is not performed, the result will be too many false positives, which reduces the overall effectiveness of the attack [61].

*2) **Prerequisites for BREACH attack**:* For a web application to be considered vulnerable to the BREACH attack, it must possess the following features:

i. The server must use HTTP-level compression, for instance DEFLATE.
ii. User input should be reflected in the body of the HTTP response.
iii. The HTTP response body should reflect a secret, for instance a CSRF token.

BREACH attack is further aided if the size of the responses largely stays the same. Any noise in the channel makes the attack more difficult [40] [6].

*3) **Attack Implementation**:* BREACH attack involves a victim, an attacker, and a server as depicted in Figure 4. The victim and the attacker need to be on the same network to allow the attacker to see the victim's traffic (man in the middle).

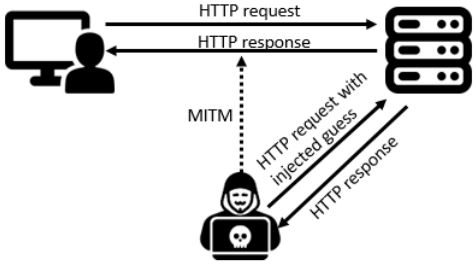

Fig. 4. BREACH attack implementation architecture

The attack leverages compression to extract data from a SSL/TLS channel by taking advantage of HTTP compression used in the HTTP response bodies. The attacker injects a guess into the HTTP request and measures the size of the compressed encrypted responses. A smaller response size indicates that the guess closely matches the secret value. The attacker then repeats this, one character at a time, while closely monitoring the size, until a perfect match is obtained [61].

Assume the HTTP request and response shown in Figure 5. Once the server verifies that the parameters submitted by the

user are correct, it sends a response back with a secret called "token" being reflected in the response.

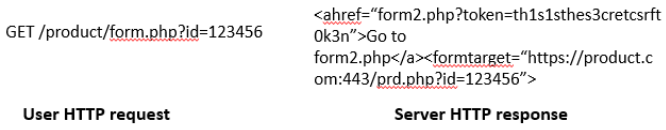

Fig. 5. HTTP request and response

This functionality is leveraged by the attacker to guess the value of the token. In the first attack request, the attacker injects a guess "token=a" into the id parameter as shown in Figure 6.

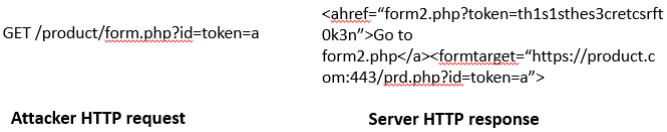

Fig. 6. Attacker's first request

The attacker then proceeds to measure the size of the response sent back. Due to HTTP compression, if the attacker's guessed value matches the first value of the actual token, the size of the response will decrease by the number of duplicate strings.

In this case, the size of the actual token "token= th1s1sthes3cretcsrftok3n" is 30. A correct guess from the attacker would result in a response size that decreased by 7. However, the attacker observes that the size decreased by 6, and can conclude that the guess was incorrect.

The attacker then proceeds to try different values for the token. When a correct token value is guessed, in this case "token=t", the size of the response will decrease by 7. As a result, the attacker establishes that the first character of the token was successfully guessed, and accordingly retains this character as a constant. The attacker then tries different values for the second character, for instance "token=tq" as shown in Figure 7.

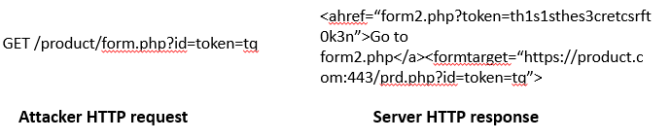

Fig. 7. Attacker's request for the second token character

The process is then repeated until the attacker successfully obtains the entire value of the token. The idea is to change the input and compare the size of the responses repetitively until the entire secret is recovered [61].

To counter the effects of Huffman Coding, an attacker can use padding as shown in Figure 8. In this instance, the characters "tq" are already known, 'd' is the attacker's next guess and is padding. This is sent with 16 possible values, and the smallest response will represent the correct guess [61].

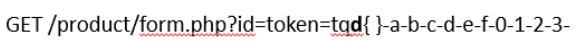

Fig. 8. Attacker's request with padding

*4) **Practicality**:* BREACH attack is easily executable in less than a minute with just a few thousand requests, with the number of requests being dependent on the size of the secret [6]. It relies on the fact that before being sent to a user, a webpage is compressed in order to save bandwidth. Although the response will be encrypted, the size of the compressed response can still be obtained [60]. The attack was observed to work on all versions of SSL/TLS [61].

BREACH was performed on Microsoft Outlook Web Access. The entire CSRF token could be reliably recovered, approximately 95% of the time, often in under 30 seconds [40].

*5) **Mitigation**:*

- **Heal the BREACH (HTB):** This mitigation method modifies the compression library used by the web server to add randomness to the size of the response [60]. As a result, the length of the compressed HTTP response is modified, and this prevents an attacker from guessing the characters of the secret token.
- **Disabling HTTP compression:** The root cause of this attack is HTTP compression. Therefore, disabling this functionality mitigates the attack. However, this method comes with a caveat that the overall performance of the web application will be significantly affected [61].
- **Separating secrets from user input:** Putting secrets in a different compression context from the rest of the response body can mitigate this attack. In this case, changes to user input do not disclose secret information in the length of the compressed HTTP response body [61].
- **Masking secrets:** This method involves generating a one-time pad P and embedding P$\|$(P$\oplus$S) in the page. P$\|$(P$\oplus$S) doubles the length of every secret and guarantees that the secret is not compressible [40]. In addition, masking the secret with a onetime random value with every request ensures that a new secret is generated every time.
- **Length hiding:** Adding random values to the compressed HTTP response body prevents an attacker from being able to calculate the size difference of the responses after compression [61].

C. BEAST Attack

BEAST (Browser Exploit Against SSL/TLS) is a attack is a security exploit against the SSL/TLS protocol, which is designed to provide secure communication over the internet. The vulnerability specifically targets the Cipher Block Chaining (CBC) mode of operation used in TLS 1.0 and earlier versions. The core issue lies in the way initialization vectors (IVs) are handled, which, in CBC mode, are combined with the plaintext before encryption to ensure that identical plaintext

blocks produce different ciphertext. However, in TLS 1.0, the IV is determined by the previous block's ciphertext, which can be predicted by an attacker in certain scenarios.

*1) **Overview**:* The BEAST attack involves a sophisticated process where an attacker injects malicious JavaScript into the victim's browser, enabling them to make repeated guesses about the content of encrypted messages. By observing the changes in the ciphertext, the attacker can infer the plaintext one byte at a time. This is possible because the attacker knows the structure of the message (such as the HTTP headers) and can manipulate the message to align the target byte at the end of a block. The attack is complex and requires precise timing and conditions, but it demonstrates a practical method to compromise the confidentiality of SSL/TLS-protected communications.

To mitigate the BEAST attack, it is recommended to use TLS 1.1 or higher, which introduces explicit IVs to prevent attackers from predicting them. Additionally, using stream ciphers like RC4, which do not use IVs in the same way as block ciphers, can also prevent this type of attack. However, given the weaknesses discovered in RC4, the best practice is to upgrade to a more secure version of TLS and use strong, modern ciphers.

*2) **Attack Implementation**:* The implementation of the BEAST attack involves several steps. First, the attacker must be able to intercept the victim's network traffic. This is typically achieved through a Man-in-the-Middle (MitM) attack, where the attacker positions themselves between the victim and the server they are communicating with.

Once the attacker can intercept the network traffic, they inject malicious JavaScript into the victim's browser. This script is designed to generate requests to the server and observe the resulting ciphertext. By carefully manipulating the plaintext and observing the resulting changes in the ciphertext, the attacker can make educated guesses about the plaintext.

The BEAST attack is particularly effective against HTTP cookies, which are often used to authenticate users. By stealing a user's cookie, an attacker can impersonate the user and gain unauthorized access to their account. [56]

To successfully carry out a BEAST attack, the attacker needs a high degree of control over the victim's browser and network traffic. This makes the attack quite complex and difficult to execute in practice. However, the existence of the BEAST attack led to significant changes in the SSL/TLS protocols to prevent similar attacks in the future. [57]

The main steps to perform the attack are as follows:

1. Initialize the test environment 2. Set the SSL/TLS version to TLS 1.0 3. Set the encryption mode of operation to CBC 4. Start secure communication between the client and the server 5. Observe the encrypted traffic to identify patterns in the Initialization Vectors (IVs) 6. Inject malicious JavaScript into the client (this is done only in a controlled test environment) 7. Generate requests to the server and observe changes in the ciphertext 8. If predictable patterns are observed in the ciphertext, then the system is vulnerable to the BEAST attack 9. Record the results 10. Clean up the test environment

D. POODLE Attack

The POODLE attack, short for Padding Oracle On Downgraded Legacy Encryption, targets a vulnerability within the SSL protocol, designated CVE-2014-3566 [12], with the aim of decrypting confidential data. Specifically, it seeks to compromise sensitive information like login details and encrypted exchanges between users and websites. [13] In the following section, we offer a comprehensive examination of this attack's mechanics and impact. Subsequently, we delve into the intricacies of the identified SSL cryptography weakness. Concluding our discussion, we outline our approach to implementation and suggest effective strategies for mitigating the risks posed by the POODLE attack.

*1) **Overview**:* SSL, or Secure Sockets Layer, stands as a cryptography protocol with the primary purpose of ensuring secure communication across networks. Its role encompasses safeguarding the confidentiality and integrity of data exchanged between a client (such as a web browser) and a server. Although SSL 3.0 is an antiquated and insecure protocol, it has largely given way to more robust alternatives like the Transport Layer Security (TLS) protocol. Various iterations of TLS, including TLS 1.0, TLS 1.1, and TLS 1.2, retain compatibility with SSL 3.0 to facilitate seamless interaction with older systems and maintain user experience continuity. Consequently, many TLS clients adopt a mechanism known as a "protocol downgrade dance" to navigate around server-side interoperability issues when dealing with legacy systems. [14] During the initial handshake, the client presents the highest protocol version available. If this attempt fails, subsequent efforts, potentially repeated, involve trials with earlier protocol versions. This downgrading process differs from standard negotiation, where a server may respond to a client's proposal of a higher version with a lower one. However, in the downgrade dance scenario, downgrading can result not only from negotiation but also from network disruptions or deliberate interference by malicious actors. Consequently, attackers may confine communication to SSL 3.0. Once the protocol is successfully downgraded, the attacker gains the ability to decrypt the SSL session content, executing decryption on a byte-by-byte basis, often leading to the establishment of numerous connections between the client and server. [15]

The US-CERT has acknowledged a design flaw discovered in how SSL 3.0 manages block cipher mode padding. The POODLE attack serves as a demonstration of how attackers can exploit this vulnerability to decrypt and extract data from encrypted transactions. [20]

The vulnerability within SSL 3.0 arises from the encryption process of data blocks using a specific encryption algorithm within the SSL protocol. The POODLE attack capitalizes on the protocol version negotiation functionality embedded in SSL/TLS to compel the usage of SSL 3.0, subsequently exploiting this newfound vulnerability to decrypt specific content within the SSL session. This decryption occurs incrementally, byte by byte, leading to the establishment of numerous connections between the client and server. [20]

Despite SSL 3.0 being an outdated encryption standard and largely superseded by TLS, most SSL/TLS implementations retain backward compatibility with SSL 3.0 to facilitate interaction with legacy systems and uphold a seamless user experience. Even in cases where both client and server support a version of TLS, the SSL/TLS protocol suite allows for negotiation of protocol versions (commonly referred to as the "downgrade dance" in other discussions). The POODLE attack exploits the tendency of servers to revert to older protocols like SSL 3.0 when a secure connection attempt fails. By inducing a connection failure, an attacker can coerce the use of SSL 3.0 and proceed with the new attack. [20]

To effectively carry out the POODLE attack, two additional prerequisites must be fulfilled:

- Firstly, the attacker needs control over segments of the SSL connection's client side, enabling manipulation of input length.
- Secondly, the attacker must gain access to the resulting cipher-text. Typically, achieving these conditions involves adopting a Man-in-the-Middle (MITM) stance, which necessitates a distinct method of attack to acquire such access.

These requirements introduce complexity to the successful exploitation of the vulnerability. Environments predisposed to MITM attacks, like public WiFi networks, mitigate some of these obstacles.

*2) **Details of POODLE Attack (CVE-2014-3566)**:* SSL 3.0 employs either the RC4 stream cipher or a block cipher in Cipher Block Chaining (CBC) mode for encryption purposes. However, it's essential to note that RC4 is known to exhibit biases. These biases imply that when the same secret, like a password or HTTP cookie, is transmitted across multiple connections and encrypted with different RC4 streams, increasing amounts of information will gradually become exposed. [15]

An issue of paramount concern with CBC encryption within SSL 3.0 arises from its block cipher padding, which lacks determinism and remains outside the purview of the Message Authentication Code (MAC). Consequently, during decryption, it becomes challenging to ascertain the integrity of the padding entirely. [15]

The exploit becomes most viable when encountering a padding block consisting of L-1 arbitrary bytes followed by a single byte with a value of L-1 (where L represents the block size in bytes). Upon processing an incoming cipher-text record $C_1, C_2, ..., C_n$ (with each C_i representing one block) and given an initialization vector C_0, the plain-text initially appears as $P_1, P_2, ..., P_n$ where $P_i = DK(C_i) \oplus C_{i-1}$ (with DK indicating block cipher decryption using the key K). Following this, the recipient conducts a validation and removal process for the padding, subsequently eliminating the MAC post-verification. Notably, the MAC size in SSL 3.0 CBC cipher suites typically amounts to 20 bytes. [15] As a result, beneath the CBC layer, an encrypted POST request assumes the following structure:

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

For executing the POODLE attack, the perpetrator positions themselves as an intermediary between the victim and the server. By intercepting and altering the SSL records transmitted by the browser, the attacker strategically manipulates them to increase the likelihood of server acceptance without rejection. Once the modified record gains acceptance, the attacker gains the capability to decrypt a single byte of the message, such as cookies. With control over both the request path and body, the attacker orchestrates requests to fulfill two specific conditions [16]:

- 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).

Afterward, the intruder substitutes C_n with C_i and dispatches the altered SSL record to the server. Typically, the server will deny this record, prompting the attacker to retry with a fresh request. Occasionally, approximately once in every 256 requests, the server will approve the altered record. At this juncture, the attacker deduces that $DK(C_i)[15] \oplus Cn-1[15] = 15$, signifying that $P_i[15] = 15 \oplus Cn - 1[15] \oplus Ci - 1[15]$. Consequently, the initial undisclosed byte of the cookie is uncovered, prompting the attacker to proceed to the subsequent byte by adjusting the dimensions of the request path and body concurrently, ensuring that the request size remains consistent while the header's position shifts. This iterative process persists until the attacker decrypts the desired portion of the cookies. [15]

*3) **Systems Affected**:* Every system and software employing Secure Socket Layer (SSL) 3.0 with cipher-block chaining (CBC) mode ciphers could potentially face vulnerabilities. Nonetheless, the POODLE (Padding Oracle On Downgraded Legacy Encryption) attack highlights this vulnerability predominantly within the domain of web browsers and servers, presenting a common avenue for exploitation.

Furthermore, certain implementations of Transport Layer Security (TLS) are also susceptible to the POODLE attack.

*4) **Implementation**:* The architecture of our implementation is depicted in Fig. 9. In this setup, the attacker assumes the role of a middleman situated between the victim's browser and the HTTPS server. Within the victim's browser, the request generator operates, implemented as a static JavaScript file named *(POODLEClient.js)*. The HTTP server hosts the static request generator code and responds to requests from the generator concerning the parameters of the generated HTTPS requests directed to the target server under attack. For this purpose, the Python module *http.server* was utilized. The TLS Proxy intercepts and modifies the TLS packets generated by the HTTPS requests and monitors the responses from the server. Implementation-wise, the module *socketserver* was employed for the TLS proxy functionality. Given that the TLS proxy and the HTTP server operate on different threads, Python classes such as Manager and BaseManager from *multiprocessing.managers* were leveraged to instantiate objects accessible from various threads and even remotely. [17]

To execute the POODLE attack, we proceeded with the subsequent steps. Initially, we downgraded the TLS protocol to SSL 3.0 by interfering with the TLS handshake process. Subsequently, we augmented the size of the POST body by

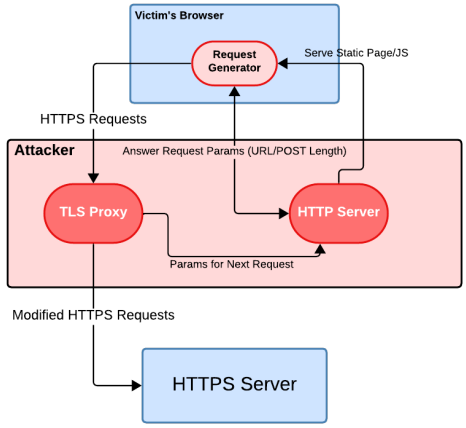

Fig. 9. POODLE Attack Implementation Architecture

one byte until the ciphertext expanded by a block size. This adjustment was made to align the URL and POST length in a manner that the final block of the ciphertext constituted padding. Ultimately, we conducted a copy operation on each generated TLS packet and determined the leaked byte upon the server's acceptance of the modified packet. [17]

As the latest iterations of Mozilla Firefox and Google Chrome remain secure, we opted to set up Firefox v37.0.0 in our testing environment, which was Kali Linux. Additionally, to render the browser susceptible to the POODLE attack, configuration adjustments were necessary, specifically involving enabling SSL downgrade, achieved in the following manner:

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

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

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

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

A custom script named `TestHTTPServer.py` was developed to serve as the target server for the attack. This script forwards incoming connections to the HTTP server. Additionally, a script titled *POODLE-dev.sh* was created to initiate the *httpserver*, *sslserver*, and *attacker-nodebug* components. The attack is initiated by requesting the URL `http://localhost:8000` from a browser vulnerable to exploitation. Consequently, it takes approximately 240 seconds on average to leak 8 bytes of data [17].

*5) **Mitigation Techniques:**:* Within TLS servers, the utilization of the $TLS_FALLBACK_SCSV$ parameter guarantees that SSL 3.0 is exclusively employed when interfacing with legacy systems, thereby thwarting protocol downgrades and mitigating the risk of POODLE attacks. Additionally, to fortify against the POODLE attack, it is imperative to deactivate SSL 3.0 support on both servers and browsers.

E. DROWN Attack

The DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) security breach leverages weaknesses within SSLv2 to decipher RSA encrypted communications. It specifically focuses on servers retaining support for the outdated and vulnerable SSLv2 protocol, despite their predominant usage of more recent TLS versions. This susceptibility often arises from servers maintaining SSLv2 compatibility for legacy purposes.

The attack methodology is depicted as follows: the attacker monitors numerous connections between the targeted client and web server, which involves capturing traffic over an extended period or enticing the user to access a website that rapidly establishes multiple connections to another site in the background. These connections can utilize any SSL/TLS protocol version, including TLS 1.2 with the RSA key exchange method [20], [21], [27].

Subsequently, the attacker initiates multiple connections to the SSL 2.0 server and transmits modified handshake messages containing alterations to the RSA ciphertext extracted from the victim's connections. The server's response to each of these probes hinges on whether the modified ciphertext decrypts into a plaintext message with the correct structure. This response pattern aids the attacker in uncovering secret keys, which are then exploited for the victim's TLS connections [20], [21], [27].

Thus, for contemporary servers and users supporting the SSL 2.0 protocol, DROWN poses a significant threat. Servers are susceptible to DROWN attacks for the following reasons [20], [21]:

- Utilizing SSL 2.0 connections.
- Employing its private key on any other server permitting SSL 2.0 connections, even across different protocols.

*1) **Overview**:* In contemporary times, the majority of up-to-date servers eschew SSLv2 in favor of alternative protocols such as TLS. However, a study conducted in 2016 revealed that among 36 million HTTPS servers examined, a notable 6 million still maintained support for SSLv2 [20]. This deprecated protocol becomes a vulnerability exploited by attackers who capitalize on the SSLv2 handshake process to decipher encrypted key ciphertext.

*2) **SSLv2 handshake**:* During the SSLv2 handshake process, the client initiates communication by transmitting the clientHello message and awaits the ServerHello message from the server.

Following this exchange, the client proceeds to encrypt the master key using the server's key before sending it back. Upon receipt, the server encrypts the cipher-text and, should the message format comply with protocol standards, encrypts a new message using the symmetric key to transmit to the client. However, if the format proves to be invalid, the server resorts to generating a random key for encrypting the message instead. This particular aspect of the protocol presents an opportunity for attackers to assess the validation of the message format by sending a message twice. A discrepancy in the keys for the

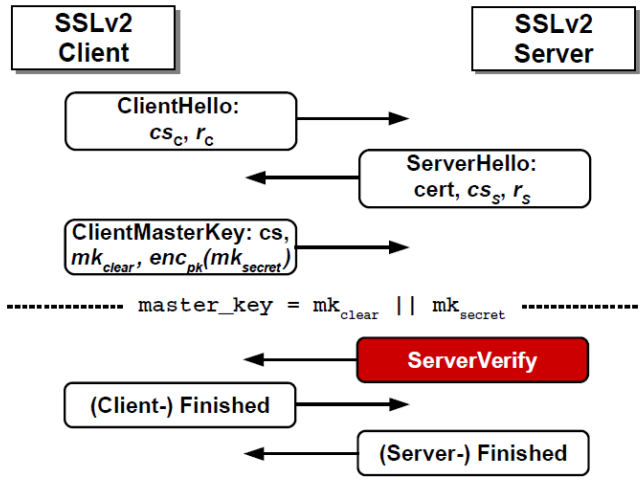

Fig. 10. SSLv2 connection handshake [20]

responses signifies that the server has generated two distinct random keys, indicating an invalid format. [22]

In the context of the DROWN attack, assailants exploit the SSLv2 protocol to disrupt TLS connections. Illustrated in Figure 11, when a client lacking SSLv2 support sends a request to a server using the TLS protocol, there remains a possibility that the server supports SSLv2 or that another server supporting SSLv2 shares the same key with the former server. In such scenarios, the attacker intercepts and captures the traffic between the client and the server. Subsequently, the attacker sends the captured traffic to the SSLv2 server multiple times, each time with slight modifications aimed at gathering information. Leveraging this acquired information, the attacker can decrypt the RSA cipher-text, employing the Bleichenbacher attack for decryption.

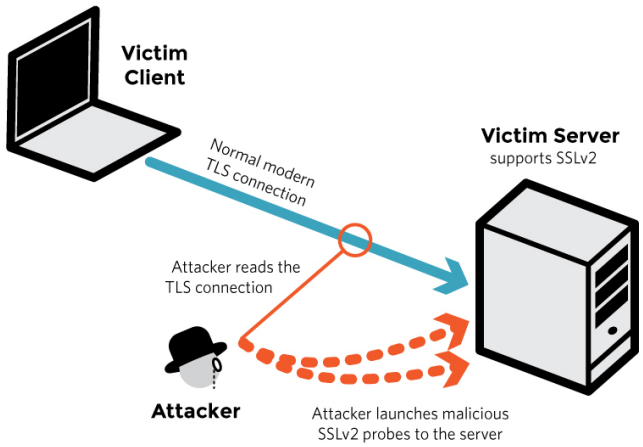

Fig. 11. Attacker uses SSLv2 to break TLS [20]

*3) **Details of decryption using Bleichenbacher**:* Bleichenbacher's padding oracle attack constitutes an adaptive chosen ciphertext exploit targeting PKCS#1 v1.5, the RSA padding standard utilized in SSL and TLS protocols. This attack facilitates the decryption of RSA ciphertexts by exploiting a server's ability to differentiate between correctly and incorrectly padded RSA plaintexts, earning it the moniker "million-message attack."

Fig. 12. PKCS#1 v1.5 block format for encryption [20]

In the encryption padding process of PKCS#1 v1.5, depicted in Figure 12, data undergoes encryption with a predefined format. Upon receiving encrypted data, the server decrypts it and assesses the format's validity. If any issues arise with the format, the server returns a response of 1; otherwise, it returns 0 for a properly formatted input.

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

Commencing with the cipher-text c0, the attacker transmits it to the oracle server and observes the result. Subsequently, the attacker iteratively adjusts the cipher-text, narrowing down the potential solution set. The modified cipher-text then undergoes further evaluation.

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

Upon receiving a response of 1, the attacker proceeds to adjust the cipher-text accordingly. Otherwise, he can deduce for some value r, $2B \leq m_0 s^{-r} \mod N < 3B$. And can figure out the possible range form as below:

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

In contrast to Bleichenbacher's attack, which targets TLS using a 384-bit key length, DROWN focuses on vulnerabilities associated with short secret keys typically found in export-grade cryptography, where the key length is a mere 40 bits. Furthermore, in TLS, the server autonomously selects the cipher suite type, leaving us with limited insight into the precise length of the key.

*4) **Implementation**:*

a) Intercepting and recording: Capture approximately 1000 TLS connections by intercepting network traffic.

b) Morph TLS connection: As the captured connection lacks compatibility with the SSLv2 oracle for decryption, efforts are made to identify the SSLv2 format for the cipher-text.

c) decrypyt the key using bleichenbacher method: Transmit modified cipher-text multiple times to progressively narrow down potential solutions until only one remains viable.

*5) **Mitigation**:* Various measures can be taken to reduce the susceptibility to this attack:

a) Complete Disabling of SSLv2: Eliminating SSLv2 entirely helps thwart potential exploits associated with its vulnerabilities. It's advisable to deactivate SSLv2 on both client and server ends whenever feasible.

b) Verification of Public Key Usage: Employ distinct public/private key pairs for SSLv2 and TLS implementations if both are necessary. This precaution is particularly crucial as the DROWN attack relies on key reuse.

c) Updating OpenSSL: Vulnerabilities are present in OpenSSL versions before 1.0.2f and 1.0.1r. Ensure systems are upgraded to newer OpenSSL versions with appropriate patches to mitigate risks.

F. CRIME Attack

CRIME (Compression Ratio Info-leak Made Easy) is a security vulnerability exploitation technique against the HTTPS and SPDY [32] protocols when compression is used in HTTP requests to transport data [31]. When this attack is executed and successful, the HTTP requests header can be retrieved and recovered completely, then later, that information can be used by an attacker to perform other types of attacks. CRIME was exposed for the first time in 2012 by security researchers Juliano Rizzo and Thai Duong at Ekoparty security conference [9]. It showcased, for the first time, a real-life example on how information leakage occurs through data compression first reported in 2002 by John Kelsey [33]. The vulnerability that made the attack possible is CVE-2012-4929 [34].

*1) **Overview**:* Some HTTP requests could contain significant authentication related information such as cookies. When those cookies are leaked, an attacker is able to use them to hijack the session. The main objective of the CRIME attack is to reveal the content from those HTTP requests by exploiting the compression algorithms of the TLS and SPDY [32] protocols when the use DEFLATE [35] and gzip is needed [38]. The latter two are data compression file formats, but in the case of DEFLATE, it uses a combination of LZ77 [39] and Huffman coding.

To better explain how Deflate's compression LZ77 works, the following example is provided with a compressed string:

$$\text{Security is } s(-14, 7)$$

In the LZ77 decompression algorithm, when the number -14 is inserted as a byte value in the string, it implies on moving the pointer 14 positions to the left from that point. The new position corresponds to the letter 'e' in the word "security". Then, the algorithm will copy the next seven characters starting from that position position. In this example, the seven characters form the string "ecurity", which is then substituted into the original string in place of the pair of numbers (-14,7). As a result, the decompressed string is formed.

$$\text{Security is security!}$$

When the LZ77 compression is occurring, when the second occurrence of a string happens, instead of storing the entire string again, a pointer is used to refer back to the location where the string first appeared, along with the length of the string. These pointers can even refer to other pointers, creating a chain of references, as illustrated in Figure 13.

CRIME relies on a blend of chosen plain-text attacks and unintentional data exposure through compression. The attacker

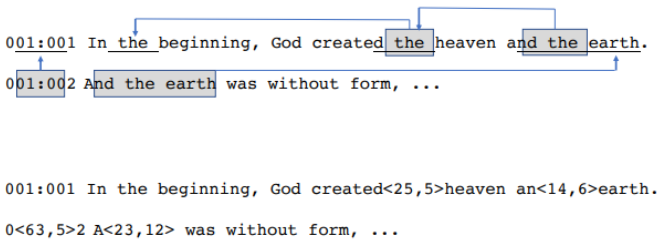

Fig. 13. Complex example of LZ77 compression.

monitors alterations in the size of the compressed request payload, which contains both the confidential cookie (in which in this scenario is a session token) sent from the browser to the destination site and dynamic content created by the attacker. When the content size reduces, it suggests that a segment of the injected content possibly aligns with a portion of the original, offering the attacker clues to potentially deduce the secret cookie accurately.

*2) **Prerequisite for CRIME**:* Executing the CRIME attack relies on specific and special environments and requirements. To execute this attack successfully, the attacker needs to have and ensure the following abilities [40] [9]:

- Compression knowledge, specially with DEFLATE, and gzip, etc. as they are used in HTTP requests.
- The attacker need to be able to measure the size of encrypted traffic.
- Must have the ability to inject a chosen plain-text into a victim's HTTP requests.

If all the conditions are met, an attacker can exploit the information disclosed by compression to reconstruct specific segments of the plain-text, such as a secret cookie that could contain a session token, by analyzing the changes in the size of the compressed data.

*3) **Implementation**:* CRIME takes advantage of the data compression functionality within SSL and TLS protocols to successfully compromise a victim. Compression happens at the SSL/TLS layer, covering both the header and body of the data. The compression algorithm utilized by SSL/TLS and SPDY [32] is DEFLATE, which eliminates duplicate strings, effectively compressing HTTP requests. CRIME exploits the method by which duplicate strings are removed, employing a systematic brute force strategy to estimate session tokens. Each instance of a duplicate string is replaced with a pointer to the initial occurrence of that string. As a result, the degree of redundancy in the data directly impacts the compression ratio. Higher data redundancy leads to more compression, resulting in a shorter length for the HTTP request. The attacker utilizes this behavior on their advantage to achive their goal. The CRIME attack process is illustrated below.

In Figure 14, We notice an example of an HTTP request sent from a website. Compression methods like gzip and DEFLATE are utilized to compress or encode this HTTP request. Within the request header lies a cookie storing a session secret. CRIME takes advantage of the utilization of compression

Fig. 14. An example of HTTP request [9]

methods in HTTP requests, meeting the prerequisite for its execution.

Fig. 15. Attacker injects some text (e.g. by javascript) [9]

In Figure 15, the attacker injects various strings into the HTTP request; Therefore, it meets the second condition for the CRIME attack. Using DEFLATE compression, the system identifies multiple instances of the "Cookie: secret=" part. The second occurrence is substituted with a compact token indicating the position of the first occurrence, causing a reduction of the request length by 15 characters (the length of the string "Cookie: secret="). Even though the attacker cannot directly access the data, they monitor changes in the request length. Utilizing a brute-force strategy, the attacker systematically tests different values for the secret until the correct one is compressed. For example, the attacker iterates through secret = 2, secret = 3, and so on until discovering the correct value.

Fig. 16. Attacker injects some correct text (e.g. by javascript) [9]

In Figure 16, when the attacker sets secret=7, they notice a decrease in the request length by 16. This indicates that the attacker has successfully extracted the first character of the cookie. Following this observation or hint, the attacker repeats the same process, until the entire cookie value is successfully retrieved.

*4) **Practicality**:* The CRIME attack is remarkably efficient, requiring only a small number of requests—just six per byte of the cookie [41] to be able to be successfully executed and obtain the desired value inside the header. Despite its minimal demand for requests, the impact of this attack is significant, particularly given the mainstream adoption of TLS across numerous services. The attack's effectiveness in acquiring sensitive information with minimal interaction underscores the necessity for robust countermeasures to defend against such threats in secure communication protocols.

*5) **Mitigation Techniques**:* The mitigation techniques recommended to counter CRIME attacks are the following:

- **Disabling compression:** CRIME attack was present in TLS version 1.0 [42]. To mitigate it, TLS 1.1 and TLS 1.2 versions were introduced, where compression mechanisms were disabled at TLS levels. [38].
- **Compression algorithm:** In TLS protocol version 1.2, the client specifies a compression algorithm in the ClientHello message, and the server selects one from the offered list, responding with its choice in the ServerHello message. If either the client offers or the server selects the **"none"** compression algorithm, no compression occurs [31], effectively mitigating the issue discussed.
- **Keep Browser Updated:** Everyday, new attacks occur and appear with new approaches to bypass the security measures implemented to mitigate known attacks, it is necessary to keep the browsers updated all time to prevent the attacks.

G. ROBOT Attack (Return Of Bleichenbacher's Oracle Threat)

It was demonstrated by Hanno Bock, Juraj Somorovsky and Craig Young [45], that the 1998 Bleichenbacher's RSA vulnerability still affects an important quantity of today's important web hosts, including Paypal, Facebook, Cisco, Citrix, Radware, F5, Erlang, Bouncy Castle, and WolfSSL among others They demonstrate a practical exploitation of facebook.com HTTPS certificate by signing a message with its private key.

Because SSL servers were issuing error codes for padding errors in PKCS 1 v1.5, malicious users were able to take advantage of an adaptive chosen-ciphertext attack vulnerability. They could exploit error messages to breach TLS confidentiality in this way [46]

Through numerous iterations of a message containing the TLS session of a third party and the ability to distinguish between the two error codes, the attacker was eventually able to piece the session back together bit by bit. If the attacker manages to obtain user credentials during that TLS session, the breach can commence. Since then, Bleichenbacher to the point that this version only needs tens of thousands of tries. Every now and then, the Bleichenbacher attack makes an appearance. In 2016, Filippo Valsorda discovered one in the Python-RSA library. In 2012, a German team discovered one in XML encryption. [47]

Only two user classes are affected by the vulnerability, RSA is still used by those with legacy systems (Windows XP) or by organizations that expressly forbid forward secrecy due to

requirements for passive monitoring. End users are the ones most at risk in this category, which includes many financial institutions. The researchers assert that the server can sign any message with their proof-of-concept code. Getting the oracle to sign an active TLS handshake would be the obvious course of action, enabling a legitimate TLS man-in-the-middle attack similar to LOGJAM. However, it is unlikely that an attacker could execute a MitM in real time. The largest drawback of the optimized Bleichenbacher attack is that it still needs tens of thousands.

*1) **Investigation**:* Bleichenbacher calculated in his initial publication that it takes roughly one million queries in order to crack any given ciphertext. As a result, "million message attack" was another name for the attack. However, the effectiveness of the attack varies. based on the oracle's "strength" as given. Generally, for each valid Oracle response, the attack algorithm discovers a new interval. If the decrypted ciphertext begins with 0x0002, this occurs. If the oracle gives a negative answer for some decrypted ciphertexts that begin with 0x0002, it is deemed to be "weaker", in this case, the attacker must ask additional questions because the new interval cannot be found. This may occur, for instance, if the implementation rigorously verifies the length of the unpadded key or the PKCS 1 v1.5 format, which requires that the first 8 bytes after 0x0002 be non-zero. It was just considered 2 types the "weak" and "strong" oracle types. An arbitrary ciphertext can be decrypted using the "strong" oracle in an average of less than a million queries. An implementation that yields true if the decrypted ciphertext begins with 0x0002 and contains a 0x00 at any point can serve as such an oracle. An implementation that verifies whether the 0x00 byte is positioned correctly can supply the weak oracle, which leads to an attack with millions of queries. The original Bleichenbacher algorithm is what was employed. Bleichenbacher's attack is referred to as a decryption attack in the majority of the studies. It's worth noting that the attack makes it possible to carry out random RSA private key operations. When an attacker gains access to an oracle, they can use the server's private RSA key to sign any message in addition to decrypting ciphertexts. An appropriate hash function and encoding are used by the attacker to process the message before generating a signature using the server's private key. When generating a PKCS 1 v1.5 signature for message M, for instance, the encoded outcome will look like this:

$$EM = 0x0001||0xFF...FF||0x00||ASN.1(hash(M))$$

A cryptographic hash function is indicated by hash(). It is necessary to encode the hash function's output using ASN.1. After that, the attacker configures EM to be an input for the Bleichenbacher algorithm. He treats the message that needs to be signed almost like an intercepted ciphertext. This process culminates in a legitimate signature for M.

In order to conduct an efficient scan with the fewest number of requests feasible while still enabling to exploit all known vulnerabilities and maybe discover new ones. In order to achieve this, it was modeled an initial scanner after the methods found in Bleichenbacher's original publication as well as the findings of the subsequent studies. With the help of PKCS 1 v1.5 messages that were stored in ClientKeyExchange and had a different format, this scanner executed a basic TLS-RSA handshake. It was to locate the first TLS implementations that are vulnerable with this method. Before informing vendors and site operators of the behavior, more investigation was done to find potential false positives. Through manual analysis, it was able to expand the previous TLS scans and identify new problems.

The ClientKeyExchange contained PKCS 1 v1.5 messages were separated in 5 different formats to cause different server behaviors. notation as follow: For byte concatenation, use ||, two TLS version bytes are represented by "version" and rnd[x] indicates a function that creates a non-zero padding string whose inclusion fills the message to achieve the RSA key length. pad() indicates a non-zero random string of length x:

Strategy	Vector
Correctly formatted TLS message.	M1 = 0x0002 || pad () || 0x00 || version || rnd [46]
Incorrect PKCS #1 v1.5 padding.	M2 = 0x4117 || pad ()
0x00 at wrong position.	M3 = 0x0002 || pad () || 0x0011
Missing 0x00.	M4 = 0x0002 || pad ()
Wrong TLS version.	M5 = 0x0002 || pad () || 0x00 || 0x0202 || rnd [46]

Fig. 17. PKCS 1 v1.5 selected vectors from Bleichenbacher's work

It was noted that the responses of multiple implementations varied according to the TLS protocol flow that was constructed. Certain servers processed a ClientKeyExchange message differently when it was sent alone compared to when it was sent along with ChangeCipherSpec and Finished.

In some configurations, the device would immediately terminate the handshake and close the connection upon receiving an invalid ClientKeyExchange without sending any more messages. Otherwise, the device waited for subsequent ChangeCipherSpec and Finished messages after processing correctly formatted ClientKeyExchange. The scans also demonstrated that timing or TLS alert numbers alone should not be taken into account when choosing an appropriate side-channel. Monitoring connection status and timeout problems is also essential.

As per the TLS standard, servers that receive invalid ClientKeyExchange messages ought to proceed with the TLS handshake and reply with an identical TLS alert every time. During the analysis, it was found that a number of servers consistently returned the same TLS alerts. However, some when processing an invalid ClientKeyExchange returned an additional TLS alert.

The scanner was performed to 1 million domains from the Alexa Top list. As a result, 27,965 hosts were vulnerable including sites as Facebook, F5, Citrix, Radware, Cisco ACE, Earlang, Bouncy Castle, WolfSSL.

Also, a proof-of-concept attack was created to enable signing and decrypting messages using a server's vulnerable key. The code searches for Bleichenbacher vulnerabilities on the host. Attempts were made to identify various signals provided by the server and automatically modify the oracle accordingly.

Several more connections to a server are necessary for the attack to be successful. To establish these connections more quickly, the attack code makes use of the `TCP_NODELAY` flag and TCP Fast Open when it is available. More Oracle queries can be executed per second thanks to the reduction of latency and connection overhead.

An attacker can use the server's private key to carry out operations on a host that is vulnerable. It does, however, take some time to execute because the attack typically requires several tens of thousands of connections. This has an effect on how the attack is perceived. TLS facilitates various types of key exchanges with RSA, including: Static RSA key exchanges, in which the client encrypts a secret value, and forward-secrecy enabled key exchanges, utilizing elliptic curve Diffie Hellman, in which RSA is only used for signing.

The attack has disastrous effects if a static RSA key exchange is used. The Bleichenbacher oracle can be used by an attacker to decrypt traffic that they have passively recorded. The servers that are most vulnerable are those that only facilitate static RSA key exchanges.

*2) **Mitigation**:* The following precautions should be taken in order to reduce vulnerabilities to the ROBOT attack[x]:

- Choose safer key exchange techniques like Elliptic Curve Diffie-Hellman (ECDHE) or Diffie-Hellman (DHE) instead of cipher suites that exchange keys using RSA encryption.
- On web applications use security headers like Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS).
- Make sure the RSA keys for the TLS certificates are long enough (2048 bits) and that they are generated securely.
- Enable Perfect Forward Secrecy (PFS).
- Put in place error-handling procedures that prevent the disclosure of decryption failure details.
- Make sure your web applications and infrastructure are routinely scanned and audited for known vulnerabilities and TLS and RSA encryption configuration problems
- Include firewalls for web applications (WAF)
- Use network segmentation when possible.
- Do routine penetration tests and vulnerability assessments.

*3) **Conclusion**:* Following Bleichenbacher's initial assault, the TLS designers determined that maintaining the susceptible encryption modes while incorporating countermeasures was the optimal approach. Subsequent studies revealed that these countermeasures were insufficient, which prompted the TLS designers to include more intricate countermeasures.

An oracle based on various TLS alerts was used in Bleichenbacher's 1998 original work. ROBOT investigation modified it so that it can now distinguish between different error types, such as duplicate TLS alerts, timeouts, and connection resets. Additionally, it was found that could identify more vulnerable hosts by employing a shortened message flow in which the ClientKeyExchange message is sent without a ChangeCipherSpec and Finished message.

H. LUCKY 13 Attack

The Lucky 13 vulnerability, identified by CVE-2013-0169, exposes significant flaws in the implementation of widely used versions of the Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) protocols, including TLS 1.1, TLS 1.2, DTLS 1.0, and DTLS 1.2, as employed in prominent products such as OpenSSL, OpenJDK, and PolarSSL [43]. This vulnerability capitalizes on timing side-channel weaknesses inherent in the processing of malformed CBC padding, particularly concerning MAC check requirements. Exploiting these vulnerabilities enables remote attackers to conduct distinguishing and plaintext-recovery attacks via statistical analysis of timing data from meticulously crafted packets, thereby compromising the confidentiality of secure communications. The discovery of the Lucky 13 vulnerability underscores the ongoing challenge in designing robust cryptographic protocols and emphasizes the constant need for vigilance and refinement to counter emerging threats. In response to these vulnerabilities, countermeasures are proposed to bolster the resilience of TLS and DTLS implementations against potential exploits. These findings prompt a deeper examination of the cryptographic design of TLS and DTLS [44], with the aim of enhancing their resilience and mitigating the risks posed by timing side-channel vulnerabilities. Ultimately, the Lucky 13 vulnerability serves as a stark reminder of the perpetual arms race between attackers and defenders in the realm of cryptographic protocols.

*1) **Prerequisite for LUCKY 13**:* The Lucky 13 attack targets implementations of the Transport Layer Security (TLS) protocol, particularly those utilizing the CBC (Cipher Block Chaining) mode of operation. To understand the prerequisites of this attack, it's essential to delve into the intricacies of how TLS handles encryption and integrity verification [44].

- **Understanding TLS and CBC Mode:** Firstly, one needs a solid grasp of TLS, which is a cryptographic protocol widely used for securing communication over computer networks. Within TLS, CBC mode is often employed for symmetric encryption, where each plaintext block is XORed with the previous ciphertext block before encryption. This chaining process is integral to the security of the encryption.
- **Vulnerability in MAC Verification:** The Lucky 13 attack exploits a vulnerability in how TLS implementations verify the integrity of messages encrypted using CBC mode. Specifically, it focuses on the MAC (Message Authentication Code) verification step, which aims to ensure that the message hasn't been tampered with during transmission.
- **Timing Side-Channel:** The attack leverages a timing side-channel, meaning it exploits the variation in the time taken to process different inputs. In the context of Lucky 13, this timing discrepancy arises due to how TLS implementations handle MAC verification failures. When the MAC verification fails, the TLS server may respond differently depending on which part of the MAC

computation failed, inadvertently revealing information about the validity of the padding.

- **Padding Oracle:** Lucky 13 effectively transforms this timing side-channel into a padding oracle, a term used in cryptography to describe a vulnerability where an attacker can glean information about the plaintext by observing differences in how a system responds to valid and invalid padding.
- **Practical Exploitation:** To execute the Lucky 13 attack successfully, one needs a deep understanding of cryptography, network protocols, and the specific TLS implementation being targeted. Additionally, access to a network where TLS traffic can be intercepted and analyzed is crucial for practical exploitation.

*2) **Mitigation:**:* Mitigating the Lucky 13 attack necessitates a comprehensive approach aimed at addressing the vulnerabilities exploited by the attack while upholding the integrity and security of the TLS protocol. A fundamental strategy involves ensuring uniformity in the timing of MAC verification, regardless of the validity of the padding. This uniformity eliminates the timing side-channel utilized by Lucky 13, thus impeding adversaries' attempts to extract sensitive information. Furthermore, introducing randomized delays for MAC verification failures obfuscates timing information, rendering it more arduous for attackers to discern meaningful data. Employing generic error messages for MAC verification failures, rather than specific ones, further diminishes the information available to potential adversaries, thereby mitigating the risk of exploiting the padding oracle vulnerability. Additionally, considering the adoption of authenticated encryption modes such as GCM or CCM affords comprehensive confidentiality and integrity protection, obviating the necessity for separate MAC verification and nullifying the vulnerability exploited by Lucky 13. Regularly updating TLS implementations with the latest security patches and protocol updates is paramount for staying abreast of known vulnerabilities, including those related to Lucky 13. Lastly, fostering security awareness and education among developers and system administrators enhances their comprehension of timing side-channels and padding oracle attacks, fostering the adoption of more secure coding practices and configurations for TLS deployments [43] [44].

*3) **Conclusion:**:* In conclusion, mitigating the Lucky 13 attack demands a concerted effort to address vulnerabilities in TLS implementations while maintaining the protocol's security and functionality. By implementing strategies such as ensuring consistency in MAC verification timing and introducing randomized delays for verification failures, organizations can significantly reduce the risk of exploitation. The adoption of authenticated encryption modes like GCM or CCM offers a comprehensive solution by providing both confidentiality and integrity protection without relying on separate MAC verification. Regular updates and patches to TLS implementations are essential for staying ahead of evolving threats, including those posed by Lucky 13. Moreover, promoting security awareness among developers and administrators fosters a culture of vigilance and best practices, further enhancing the resilience of TLS deployments. Overall, a multifaceted approach that combines technical measures, protocol updates, and education is necessary to effectively mitigate the Lucky 13 attack and bolster the security of network communications.

V. Conclusion

In data protection, cryptographic protocols play a critical role in safeguarding various operational sectors by ensuring the confidentiality of data transmissions. Despite their importance, modern cryptographic protocols are not immune to exploitation by malicious actors. These adversaries exploit design flaws within the protocols, their libraries, or implementations, effectively bypassing security measures.

In this report, we analyzed the Oracle padding attack and its variants, revealing that some of these attacks exploited deprecated versions of TLS and SSL protocols, specifically TLSv1.1 and TLSv1.2. Through our analysis, we implemented a subset of these attacks, which are available in our repository []. It's important to note that the upgraded TLS v1.3 offers significantly enhanced security for internet browsing.

To remain vigilant against attackers, it is imperative for us as a collective to ensure that our protocols and security mechanisms are continuously updated to counter the evolving threats in the cyber sector. This ongoing commitment to staying abreast of emerging risks is crucial in maintaining the integrity and security of our data infrastructure.

References

[1] R. Fedler,"Padding Oracle Attacks,"Supervisor: B. 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,2013.

[2] IETF. The transport layer security (tls) protocol version 1.3, 2018. RFC 8446, Section 1.1.

[3] Rafael Fedler. Padding oracle attacks. Network, 83, 2013.

[4] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs 1. In Advances in Cryptology— CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18, pages 1–12. Springer, 1998

[5] 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.

[6] Breach attack website. https://www.breachattack.com/.

[7] 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.

[8] Hanno B¨ock, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's oracle threat. In 27th USENIX Security Symposium (USENIX Security 18), pages 817–849,2018.

[9] Juliano Rizzo and Thai Duong. The crime attack. Retrieved September 21, 2012, via Google Docs: https://tinyurl.com/rmtua57h.

[10] T. Duong and J. Rizzo. Padding oracles everywhere (presentation slides). In Ekoparty 2010, 2010.http://netifera.com/research

[11] Klima, V., Rosa, T.: Side Channel Attacks on CBC Encrypted Messages in the PKCS 7 Format. Cryptology ePrint Archive, Report 2003/098 (2003)

[12] Cve-2014-3566. NVD. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566

[13] POODLE. Wikipedia. https://en.wikipedia.org/wiki/POODLE.

[14] Downgrade attack. Wikipedia. https://en.wikipedia.org/wiki/Downgrade attack.

[15] Poodle. Google Blog. https://www.openssl.org/ bodo/ssl-poodle.pdf.

[16] What is the poodle attack? Acunetix. https://www.acunetix.com/blog/web-security-zone/what-is-poodle-attack/.

[17] Poodle implementation. Thomas Patzkes Personal Website. https://patzke.org/implementing-the-poodle-attack.html.

[18] Bodo Möller, Thai Duong and Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback, 2014.

[19] Abeer E. W. Eldewahi, Tasneem M. H. Sharfi, Abdelhamid A. Mansor, Nashwa A. F. Mohamed, Samah M. H. Alwahbani. SSL/TLS attacks: Analysis and evaluation. 2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE), 2015. 10.1109/ICCNEEE.2015.7381362.

[20] https://www.cisa.gov/news-events/alerts/2014/10/17/ssl-30-protocol-vulnerability-and-poodle-attack

[21] 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), 2016.

[22] O. Ivanov, V. Ruzhentsev and R. Oliynykov, "Comparison of Modern Network Attacks on TLS Protocol," 2018 International Scientific Practical Conference Problems of Infocommunications. Science and Technology, Kharkiv, Ukraine, 2018, pp. 565-570, doi: 10.1109/INFOCOMMST.2018.8632026.

[23] Drees JP, Gupta P, Hüllermeier E, Jager T, Konze A, Priesterjahn C, Ramaswamy A, Somorovsky J. Automated detection of side channels in cryptographic protocols: DROWN the ROBOTs!. InProceedings of the 14th ACM Workshop on Artificial Intelligence and Security 2021 Nov 15 (pp. 169-180).

[24] GlobalPlatform, "GlobalPlatform Card Specification Version 2.3.1," Mar. 2018, Reference GPC-SPE-034, [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/05/GPC-CardSpecification-v2.3.1-PublicRelease-CC.pdf.

[25] ISO/IEC 10116:2017, "Information technology - Security techniques - Modes of operation for an n-bit block cipher," [Online]. Available: https://www.iso.org/obp/ui/iso:std:iso-iec:10116:ed-4:v1:en

[26] ISO/IEC JTC 1/SC 27, "ISO/IEC 9797-1:2011 - Information technology - Security techniques - Message Authentication Codes (MACs) - Part 1: Mechanisms using a block cipher," [Online]. Available: https://www.iso.org/standard/50375.html

[27] Avoine, G., Ferreira, L. (2018). Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018(2), 149–170. https://doi.org/10.13154/tches.v2018.i2.149-170

[28] "The DROWN Attack" [Online]. Available: https://drownattack.com

[29] What Is HTTPS and How Does It Work? [Explained]," Semrush Blog. Accessed: Mar. 17, 2024. [Online]. Available: https://www.semrush.com/blog/what-is-https/

[30] Oppliger, Rolf. SSL and TLS: Theory and Practice. Artech House, 2023.

[31] CRIME. Wikipedia. https://en.wikipedia.org/wiki/CRIME.

[32] Spdy: An experimental protocol for a faster web. Chromium Developer Documentation. Retrieved 2009-11-1.

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

[34] Cve-2012-4929. NVD. https://nvd.nist.gov/vuln/detail/CVE-2012-4929

[35] P. Deutsch. Deflate compressed data format specification. Technical report, RFC 1951, RFC Editor, 1996

[36] https://tinyurl.com/39turfzc

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

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

[39] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3):337–343, 1977

[40] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crimeattack. https://www.breachattack.com/resources/BREACH - SSL, gone in 30 seconds.pdf, 2013. Unpublished manuscript.

[41] Crime attack: Compression ratio info-leak made easy. Threatpost. https://threatpost.com/crime-attack-uses-compression-ratio-tls-requests-side-channel-hijack-secure-sessions-091312/77006/. 2012

[42] T. Dierks. The tls protocol version 1.0 – rfc 2246. Technical report, IETF Request For Comments, 1999

[43] https://nvd.nist.gov/vuln/detail/CVE-2013-0169

[44] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA,

[45] Return Of Bleichenbacher's Oracle Threat (ROBOT) https://robotattack.org/ Hanno Bck, Juraj Somorovsky1,2, and Craig Young 3 1Ruhr-Universit¨at Bochum 2Hackmanit GmbH 3Tripwire VERT December 12, 2017.

[46] Prevent SSL ROBOT attacks. Veracode Docs. https://docs.veracode.com/r/prevent-ssl-robot.

[47] NETWORK SECURITY Stack Ranking SSL Vulnerabilities: The ROBOT Attack https://www.securityweek.com/stack-ranking-ssl-vulnerabilities-robot-attack/

[48] ROBOT website https://robotattack.org/

[49] ROBOT Attack (Breitenbacher RSA) By Nikhil Anilkumar Reviewed by Neda Ali Published on 20 Oct 2023 https://beaglesecurity.com/blog/support/vulnerability/robot-attack-breitenbacher-rsa.htm

[50] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in Proc. 13th Int. Conf. Inf. Secur. (ISC 2010), M. Burmester, G. Tsudik, S. S. Magliveras, and I. Ilic, Eds., vol. 6531 of Lecture Notes in Computer Science, Boca Raton, FL, USA, Oct. 25–28, 2011, pp. 346–360. Springer, Heidelberg, Germany.

[51] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on amazon's s2n implementation of TLS," in Proc. EUROCRYPT 2016, Part I, M. Fischlin and J. Coron, Eds., vol. 9665 of Lecture Notes in Computer Science, Vienna, Austria, May 8–12, 2016, pp. 622–643. Springer, Heidelberg, Germany.

[52] S. Vaudenay, "Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS...," in Proc. EUROCRYPT 2002, L. R. Knudsen, Ed., vol. 2332 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, Apr. 28 – May 2, 2002, pp. 534–546. Springer, Heidelberg, Germany.

[53] J. Black and H. Urtubia, "Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption," in Proc. 11th USENIX Security Symposium, USENIX Security 2002, Berkeley, CA, USA, 2002, pp. 327–338. USENIX Association.

[54] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password interception in a SSL/TLS channel," in Proc. CRYPTO 2003, D. Boneh, Ed., vol. 2729 of Lecture Notes in Computer Science, Santa Barbara, CA, USA, Aug. 17–21, 2003, pp. 583–599. Springer, Heidelberg, Germany.

[55] C. Boyd, "Modern data encryption," Electronics & Communication Engineering Journal, vol. 5, no. 5, pp. 271-278, 1993.

[56] Rizzo, J., & Duong, T. (2011). Here Come The Ninjas. ekoparty Security Conference.

[57] Goodin, D. (2011). BEAST exploits pay a visit to Facebook. The Register.

[58] Goodin, Dan. "Gone in 30 Seconds: New Attack Plucks Secrets from HTTPS-Protected Pages." Ars Technica, 1 Aug. 2013. Available: https://arstechnica.com/information-technology/2013/08/gone-in-30-seconds-new-attack-plucks-secrets-from-https-protected-pages/.

[59] Alupotha, Jayamine & Prasadi, Sanduni & Alawatugoda, Janaka & Ragel, Roshan & Fawsan, Mohamed. (2017). Implementing a proven-secure and cost-effective countermeasure against the compression ratio info-leak mass exploitation (CRIME) attack. 1-6. 10.1109/ICIINFS.2017.8300359. Available: https://www.researchgate.net/publication/323352314_Implementing_a_proven-secure_and_cost-effective_countermeasure_agains_the_compression_ratio_info-leak_mass_exploitation_CRIME_attack.

[60] R. Palacios, A. F. Fernández-Portillo, E. F. Sánchez-Úbeda and P. García-De-Zúñiga, "HTB: A Very Effective Method to Protect Web Servers Against BREACH Attack to HTTPS," in IEEE Access, vol. 10, pp. 40381-40390, 2022, doi: 10.1109/ACCESS.2022.3166175. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9754554

[61] P. G. Sarkar, S. Fitzgerald, "Attacks on SSL. A Comprehensive Study of BEAST, CRIME, TIME, BREACH, LUCKY 13 & RC4 Biases". Available: https://d.cxcore.net/InfoSec/ssl_attacks_survey.pdf