Contents

Chapter 1

Theoretical background

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

Specifically, section 1.1 describes the functionality of the gzip compression method and the algorithms that it entails. Section 1.2 covers the same-origin policy that applies in the web application security model. In section 1.3 we explain the Transport Layer Security, which is the main protocol used today to provide communications security over the Internet. Finally, in section **??** we describe attack methodologies in order for an adversary to perform a Man-in-the-middle attack, such as ARP spoofing and DNS poisoning.

1.1 gzip

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

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

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

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

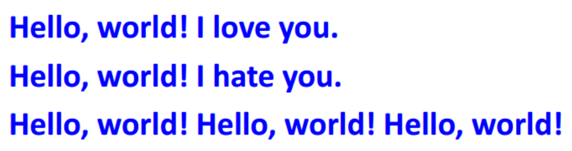
1.1.1 LZ77

LZ77 is a lossless data compression algorithm published in paper by A. Lempel and J. Ziv in 1977. [1] It achieves compression by replacing repeated occurences of data with references to a copy of that data existing earlier in the uncompressed data stream. The reference composes of a pair of numbers, the first of which represents the length of the

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

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

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



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

Figure 1.1: First we get the plaintext to be compressed.

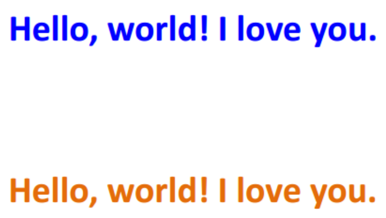


Figure 1.2: Compression starts with literal representation.

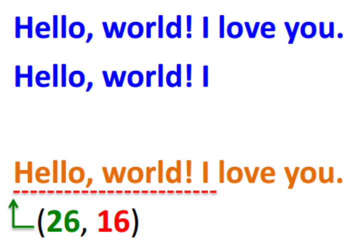


Figure 1.3: We then use a pointer at distance 26 and length 16.

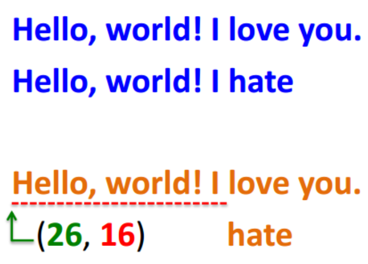
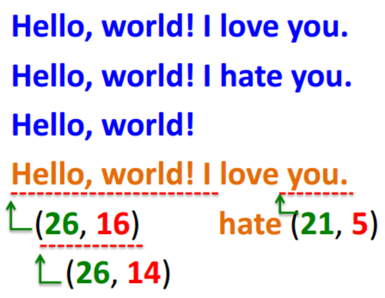
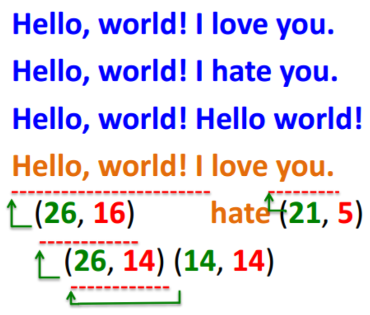


Figure 1.4: We continue with literal.



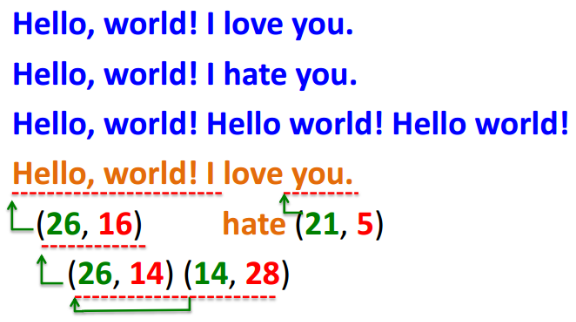
Hello, world! I love you.
Hello, world! I hate you.
Hello, world!
Hello, world! I love you.
(26, 16)
hate
(21, 5)
(26, 14)

Figure 1.5: We use a pointer pointing to a pointer.



Hello, world! I love you.
Hello, world! I hate you.
Hello, world! Hello world!
Hello, world! I love you.
(26, 16)
hate
(21, 5)
(26, 14) (14, 14)

Figure 1.6: We then use a pointer pointing to a pointer pointing to a pointer.



Hello, world! I love you.
Hello, world! I hate you.
Hello, world! Hello world! Hello world!
Hello, world! I love you.
(26, 16)
hate
(21, 5)
(26, 14) (14, 28)

Figure 1.7: Finally, we use a pointer pointing to itself.

1.1.2 Huffman coding

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

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

Chancellor on bring of second bailout for banks [2]

Frequency Analysis

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

Huffman tree

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

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

1.2 Same-origin policy

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

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

1.2.1 Cross-site scripting

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

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

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

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

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

1.2.2 Cross-site request forgery

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

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

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

1.3 Transport Layer Security

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

Chapter 2

Partially Chosen Plaintext Attack

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

2.1 Partially Chosen Plaintext Indistinguishability

2.1.1 Definition

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

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

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

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

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

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

2.1.2 IND-PCPA vs IND-CPA

Suppose the adversary submits the empty string as the chosen plaintext, a choice which is allowed by the definition of the game. The challenger would then send back the ciphertext $C_i = E(P_k, M_b||" ") = E(P_k, M_b)$, which is the ciphertext described in the IND-CPA game. Therefore, if the adversary has the ability to beat the game of IND-PCPA, i.e. if the system is not indistinguishable under partially chosen plaintext attacks, he also has the ability to beat the game of IND-CPA. Thus we have shown that IND-PCPA is at least as strong as IND-CPA. The above intuitive proof could be expressed formally in future works.

2.2 PCPA on compressed encrypted protocols

2.2.1 Compression-before-encryption and vice versa

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

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

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

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

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

2.2.2 PCPA scenario on compression-before-encryption protocol

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

$c = Encrypt(Compress(m))$

where c is the ciphertext and m is the plaintext.

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

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

where n_1, n_2, n_3 are random nonces.

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

Bibliography

[1] A. Ziv, Jacob; Lempel, “A universal algorithm for sequential data compression,” Information Theory, IEEE Transactions, vol. 23, pp. 337–343.

[2] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” Proceedings of the IEEE, vol. 40, pp. 1098–1101.