



**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCE  
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION**

**BSc THESIS**

**Extending the Rupture compression attack framework  
against real world systems**

**Dimitrios P. Grigoriou**

**Supervisor :**

**Kiayias Aggelos, Associate Professor**

**ATHENS**

**JULY 2017**



**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ  
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

**Επέκταση του εργαλείου επιθέσεων συμπίεσης Rupture για  
επιθέσεις σε πραγματικά συστήματα**

**Δημήτριος Π. Γρηγορίου**

**Επιβλέπων: Κιαγιάς Άγγελος, Αναπληρωτής Καθηγητής**

**ΑΘΗΝΑ**

**ΙΟΥΛΙΟΣ 2017**

## **BSc THESIS**

Extending the Rupture compression attack framework against real world systems

**Dimitrios P. Grigoriou**

**S.N.:** 1115201100144

**SUPERVISOR:**     **Kiayias Aggelos, Assistance Professor**

## **ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

Επέκταση του εργαλείου επιθέσεων συμπίεσης Rupture για επιθέσεις σε πραγματικά συστήματα

**Δημήτριος Π. Γρηγορίου**

**A.M.: 1115201100144**

**ΕΠΙΒΛΕΠΩΝ:**      **Κιαγιάς Άγγελος, Αναπληρωτής Καθηγητής**

## ABSTRACT

The need for system protection from various attacks makes system security one of the most important technological fields in the 21<sup>st</sup> century. This work investigates attacks on compressed encrypted protocols, and develops further the Rupture framework.

New methods are being proposed that allow the recovery of a secret from a website that uses cryptographic suites, given the fact that the secret's prefix can be found multiple times in the website's plaintext. This way, in theory, we can recover the whole plaintext from a ciphertext.

At the same time, we present two new patches: one in the BetterCap framework and one in PacketFu library. With these two patches, IPv6 attacks are now more modular and easy to conduct. The BetterCap framework is the one used in the first part of the Rupture attack. The purpose of this work is to make the Rupture attack possible on IPv6 targets.

The implementation of this work aims by no means to be used with malicious purposes. On the contrary, it aims to sensitize the community about Man-in-the-Middle attacks, as well as compression side-channel attacks, given the extensive growth of technology, such as IPv6.

Rupture is a collaborative work with (alphabetical order): Dimitris Karakostas, Dionysis Zindros, Eva Sarafianou. Updated versions on the current work can be found on the following link: <https://github.com/dimriou/rupture-thesis>. Rupture repository is: <https://github.com/decrypto-org/rupture>.

**SUBJECT AREA:** Security, Cryptography

**KEYWORDS:** Compression Attacks, Rupture Framework, BREACH Attack, Man-in-the-Middle Attacks, IPv6, Neighbor Discovery Protocol

## ΠΕΡΙΛΗΨΗ

Η ανάγκη για προστασία των υπολογιστικών συστημάτων από επιθέσεις, καθιστά την ασφάλεια ως έναν από τους σημαντικότερους τεχνολογικούς τομείς του 21ο αιώνα. Η παρούσα εργασία ερευνά επιθέσεις πάνω σε συμπιεσμένα κρυπτογραφημένα πρωτόκολλα και συγκεκριμένα αναπτύσσει περαιτέρω το εργαλείο Rupture.

Προτείνονται μέθοδοι που επιτρέπουν την ανακάλυψη μυστικού κρυπτογραφημένης σελίδας, του οποίου το πρόθεμα συναντάται πολλές φορές μέσα στο κείμενο της σελίδας. Με αυτό τον τρόπο δύναται η δυνατότητα, σε θεωρητικό επίπεδο, εξαγωγής ολόκληρου του καθαρού κειμένου από το κρυπτοκείμενο.

Παράλληλα παρουσιάζονται δυο νέα επιπλέον κομμάτια στα ήδη υπάρχουσα εργαλεία BetterCap και PacketFu, μέσα από τα οποία είναι πλέον δυνατή η επίθεση σε τερματικά που χρησιμοποιούν το πρωτόκολλο IPv6, με πολύ μεγαλύτερη αυτοματοποίηση. Το εργαλείο BetterCap είναι αυτό που χρησιμοποιείται από το Rupture στο πρώτο σκέλος της επίθεσης. Μέσα από αυτήν την εργασία το Rupture θα μπορεί πλέον να στοχεύει θύματα που χρησιμοποιούν το πρωτόκολλο IPv6.

Η εργασία αυτή δεν έχει σε καμία περίπτωση ως στόχο την κακόβουλη χρήση των τεχνικών που παρουσιάζονται. Αντίθετα με την ολοένα και περισσότερη υιοθέτηση νέων τεχνολογιών, όπως το IPv6, θέλει να ευαισθητοποιήσει την κοινότητα τόσο για τις επιθέσεις Man-in-the-Middle, όσο και για εκείνες σε συμπιεσμένα κρυπτογραφημένα πρωτόκολλα.

Η δημιουργία όλων των μεθόδων και τεχνικών είναι συνεργατική με τους (αλφαβητική σειρά): Δημήτρη Καρακώστα, Διονύση Ζήνδρο, Εύα Σαραφianού. Ανανεωμένες εκδόσεις της παρούσας εργασίας μπορούν να βρεθούν στον ακόλουθο σύνδεσμο: <https://github.com/dimriou/rupture-thesis>. Το εργαλείο Rupture βρίσκεται στο σύνδεσμο: <https://github.com/decrypto-org/rupture>.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ:** Ασφάλεια, Κρυπτογραφία

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ:** Επιθέσεις Συμπίεσης, Εργαλείο Rupture, Επίθεση BREACH, Επιθέσεις Man-in-the-Middle, IPv6, Πρωτόκολλο Neighbor Discovery

## **ΕΥΧΑΡΙΣΤΙΕΣ**

Η πτυχιακή αυτή εκπονήθηκε υπό την επίβλεψη του καθηγητή Άγγελου Κιαγιά, τον οποίο θα ήθελα να ευχαριστήσω θερμά για τη βοήθειά του, καθώς και για το γεγονός ότι μέσω της εργασίας αυτής με εισήγαγε στον κόσμο της Ασφάλειας και της Κρυπτογραφίας.

Ακόμα, θα ήθελα να ευχαριστήσω τον Διονύση Ζήνδρο, ο οποίος μου πρότεινε το θέμα της εργασίας, με καθοδήγησε κατά τη διάρκεια της εκπόνησής της και σε κάθε ευκαιρία μου μετέδιδε συνεχώς καινούριες γνώσεις.

Επίσης θα ήθελα να ευχαριστήσω τον Δημήτρη Καρακώστα για την συνεχή βοήθεια που πρόσφερε, τις χρήσιμες συμβουλές και την γενικότερη υποστήριξή του κατά την διάρκεια της εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους και την οικογένειά μου για τη στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

# CONTENTS

<b>PREFACE .....</b>	<b>12</b>
<b>1. INTRODUCTION.....</b>	<b>13</b>
1.1 Thesis Structure.....	14
<b>2. THEORETICAL BACKGROUND.....</b>	<b>15</b>
2.1 gzip.....	15
2.1.1 LZ77 .....	15
2.1.2 Huffman coding .....	17
2.2 Same-origin policy .....	18
2.2.1 Cross-site scripting.....	19
2.2.2 Cross-site request forgery.....	19
2.3 Transport Layer Security .....	20
2.4 Man-in-the-Middle .....	22
2.4.1 ARP Spoofing.....	22
<b>3. RUPTURE FRAMEWORK.....</b>	<b>24</b>
3.1 Attack Assumptions .....	24
3.2 Principles of Attack .....	24
3.3 Architecture.....	27
3.3.1 Injector.....	27
3.3.2 Sniffer .....	28
3.3.3 Client .....	29
3.3.4 Real-time .....	30
3.3.5 Backend .....	32
<b>4. BACKTRACKING .....</b>	<b>34</b>
4.1 Method Specifications.....	34
4.2 Architecture.....	36
4.2.1 Round Model .....	36
4.2.2 Backtracking Analyzer.....	36



<b>4.3 Backtracking experimental results .....</b>	<b>38</b>
<b>5. IPV6.....</b>	<b>42</b>
<b>5.1 Theoretical Background.....</b>	<b>42</b>
5.1.1 IPv6 Addressing Modes .....	42
5.1.2 IPv6 Addresses .....	44
5.1.3 Neighbor Discovery.....	45
5.1.4 Neighbor Cache States .....	46
<b>5.2 Ipv6 Patch Architecture.....</b>	<b>47</b>
5.2.1 IPv6 Parser.....	47
5.2.2 Neighbor IPv6/MAC Address discovery .....	47
5.2.3 IPv6 Packet Manipulation ( PacketFu extension patch) .....	48
5.2.4 Neighbor Discovery Spoofer .....	50
5.2.5 ip6tables Firewall .....	50
<b>6. CONCLUSION .....</b>	<b>51</b>
<b>REFERENCES .....</b>	<b>52</b>

## LIST OF FIGURES

Figure 2.1: Plaintext to be compressed.....	15
Figure 2.2: Compression starts with literal representation.....	16
Figure 2.3: Use a pointer at distance 30 and length 13 .....	16
Figure 2.4: Continue with literal .....	16
Figure 2.5: Use a pointer pointing to a pointer.....	16
Figure 2.6: Use a pointer pointing to a pointer pointing to a pointer.....	17
Figure 2.7: Huffman tree.....	18
Figure 2.8: TLS handshake flow.....	20
Figure 2.9: TLS record .....	21
Figure 2.10: Man-in-the-Middle.....	22
Figure 2.11: ARP spoofing.....	23
Figure 3.1: Sampleset.....	25
Figure 3.2: Divide & Conquer Alphabet.....	26
Figure 3.3: Scoreboard.....	27
Figure 3.4: Client code.....	30
Figure 3.5: Real-time code.....	31
Figure 3.6: work object.....	31
Figure 3.7: work-completed.....	32
Figure 4.1: Common prefix plaintext.....	34
Figure 4.2: Backtracking tree implementation.....	36
Figure 4.3: Relative probability formula.....	37
Figure 4.4: Relative probability example.....	37
Figure 4.5: Complete Backtracking graph with all types of probabilities.....	38
Figure 4.6: Backtracking execution step 1.....	39
Figure 4.7: Backtracking execution step 2.....	39

Figure 4.8: Backtracking execution step 3.....	40
Figure 4.9: Backtracking execution step 4.....	40
Figure 4.10: Backtracking execution step 5.....	41
Figure 4.11: Backtracking execution step 6.....	41
Figure 5.1: Unicast messaging.....	42
Figure 5.2: Multicast messaging.....	43
Figure 5.3: Anycast messaging.....	43
Figure 5.4: Cache state algorithm.....	47
Figure 5.5: Multicast Neighbor Solicitation Message for address resolution.....	48
Figure 5.6: Unicast Neighbor Advertisement Message for address resolution.....	48
Figure 5.7: Neighbor Solicitation message format.....	49
Figure 5.8: Neighbor Advertisement message format.....	49
Figure 5.9: Spoofed neighbor cache entry.....	50

## **PREFACE**

The scope of this thesis project is to study the vulnerabilities of online systems and more specifically compression side-channel attacks. We develop a tool that performs such attacks in real world systems.

This thesis project was elaborated during my studies in Department of Informatics and Telecommunications in the University of Athens.

The thesis was realized from April 2016 to July 2017 under the supervision of Associate Professor Kiayias Aggelos, under the guidance of PhD candidate Zindros Dionysis and with the help of Cryptography Researcher Karakostas Dimitris.

Dimitrios Grigoriou,

Athens, 24<sup>th</sup> July 2017

# 1. INTRODUCTION

As the Internet evolves and computer networks become bigger and better, network security has become one of the most important factors for society to consider. Big enterprises like Facebook, Microsoft or Google are designing and building software products that need to be protected against foreign attacks. At the same time, recent publications about massive leaks on personal info of users have changed the way people are using online services. Researchers have been constantly seeking solutions in order to protect against every possible kind of attack.

The implementation of this work aims to present the weaknesses of multiple online protocols which are currently used on the Internet and sensitize the community about Man-in-the-Middle attacks, as well as compression side-channel attacks.

Most of the data which are sent online are compressed beforehand. Our work focuses on exploiting those compression algorithms in order to extract plaintext from encrypted pages. More specifically, we extend previous attack models, such as BREACH, by creating a modular framework for conducting those types of attacks. This way we point out the weaknesses of online protocols, which are considered to be safe.

On this work, we focus on compression software gzip, which uses the DEFLATE algorithm which is a variation of Huffman [\[17\]](#) and LZ77 compression. LZ77 finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length). This technique helps conducting the attack, while Huffman compression prevents it.

HTTP (Hyper-Text Transfer Protocol) is the most common method for data communication for the World Wide Web. However it is common knowledge that HTTP data are not encrypted and should not be considered as secure. This problem, was solved by SSL (Secure Socket Layer) and its descendant TLS (Transport Layer Security), which is a cryptographic protocol that provides communication security, by encrypting data before they are sent, over a computer network.

Encryption algorithms can be split in two big categories: stream ciphers and block ciphers. In the first category, data are encrypted as a continuous stream, while on the other one they are split into multiple blocks of equal size. In case the data length is not sufficient to create a complete block, it is filled with artificial noise in order to achieve the appropriate size.

One of the most popular stream ciphers is RC4. However, this algorithm is considered to be insecure due to multiple vulnerabilities. On the other hand, AES is the most popular block cipher algorithm and it is widely used in the majority of online systems with a few variations. Although block ciphers make it harder for our attack to succeed, we describe below interesting techniques that can go around this problem and ultimately perform the attack successfully.

In order to achieve the attack against block ciphers, we use statistical methods by injecting our own artificial noise to each target block. Then we analyze the results in order to extract a decision.

Rupture's implementation is designed for modularity and easy use. Its architecture requires minimum modification by the user and gives the ability for multiple compression side-channel attacks at the same time. This tool was used in lab environment attacks.

This work introduces new techniques in order to extend Rupture and make the attack possible in real world systems. We introduce a new method for recovering target secrets with high certainty. We also create new patches in order to use Rupture in IPv6 end-points.

To sum up, this work is the extension of a continuous lab research which was introduced in the past few years and pointed out some key vulnerabilities in online systems. It is important that new methods of defense will be adopted against this attack, in order to achieve higher protection in online communications.

## **1.1 Thesis Structure**

### **Chapter 2**

This chapter provides the reader with basic information, both in technical as well as theoretical terms, which will be used later. We describe the most common compression algorithms as well as basic protocols used for secure communications. We also introduce various attacks against them.

### **Chapter 3**

In this chapter we describe the tool we created for attacks in compressed encrypted protocols. More specifically, we outline the conditions under which such an attack is carried out, we analyze the terminology used in our framework and finally we describe in detail its various parts and their functionalities.

### **Chapter 4**

This chapter introduces a new method for conducting the Rupture attack, Backtracking. This method can recover multiple secrets with common prefix and extracts the final secret with a high amount of certainty, based on probabilistic models.

### **Chapter 5**

In this chapter we introduce a patch for conducting Man-in-the-Middle attacks on IPv6 endpoints. This is done by extending the capabilities of BetterCap framework, as well as the underlying library that is used, PacketFu.

### **Chapter 6**

Conclusion

## 2. THEORETICAL BACKGROUND

In this chapter, we intend to provide the necessary background to the user to understand the mechanisms used later in the work. The description of each system is only a short introduction to familiarize the reader with the needed concepts.

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 Transport Layer Security, which is the widely used protocol that provides communications security over the Internet. Finally, in section 2.4 we describe attack methodologies, such as ARP spoofing, in order for an adversary to perform a Man-in-the-Middle attack.

### 2.1 gzip

[gzip](#) is a file format and a software application used for file compression and decompression. It is the most popular compression method on the Internet, used alongside with protocols such as HTTP and XMPP. 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.

It is based on the DEFLATE algorithm, which is a composition of LZ77 and Huffman coding. DEFLATE could be described in short by the following compression 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 [7]. It achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair, the first of which represents the length of the repeated portion and the second of which describes the distance backwards in the stream. 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, therefore the initial appearance of a portion of data needs to have occurred at most 32 Kb up the data stream, in order to be compressed. Also, the minimum length of a text that can be compressed is 3 characters. 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! Nice to meet you.  
Hello world! Go away.  
Hello world! Hello world!

**Figure 2.1: Plaintext to be compressed.**

Hello world! Nice to meet you.

Hello world! Nice to meet you.

Figure 2.2: Compression starts with literal representation.

Hello world! Nice to meet you.  
Hello world!

Hello world! Nice to meet you.



Figure 2.3: Use a pointer at distance 30 and length 13.

Hello world! Nice to meet you.  
Hello world!

Hello world! Nice to meet you.



Figure 2.4: Continue with literal.

Hello world! Nice to meet you.  
Hello world!

Hello world! Nice to meet you.



Figure 2.5: Use a pointer pointing to a pointer.





Figure 2.6: Use a pointer pointing to a pointer pointing to a pointer.

### 2.1.2 Huffman coding

Huffman coding is also a lossless data compression algorithm developed by David A.

Huffman and published in 1952 [4]. 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 of 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.

Finally, the compression mapping needs to be included in the final compressed text so that it can be used during decompression.

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

***I find myself strangely drawn to this odd configuration of activity***

#### Frequency Analysis

<b>( ): 10</b>	<b>i: 7</b>	<b>t: 6</b>	<b>n: 5</b>	<b>o: 5</b>
<b>a: 4</b>	<b>d: 4</b>	<b>f: 4</b>	<b>y: 3</b>	<b>c: 2</b>
<b>e: 2</b>	<b>g: 2</b>	<b>l: 2</b>	<b>r: 2</b>	<b>s: 2</b>
<b>h: 1</b>	<b>m: 1</b>	<b>u: 1</b>	<b>v: 1</b>	<b>w: 1</b>

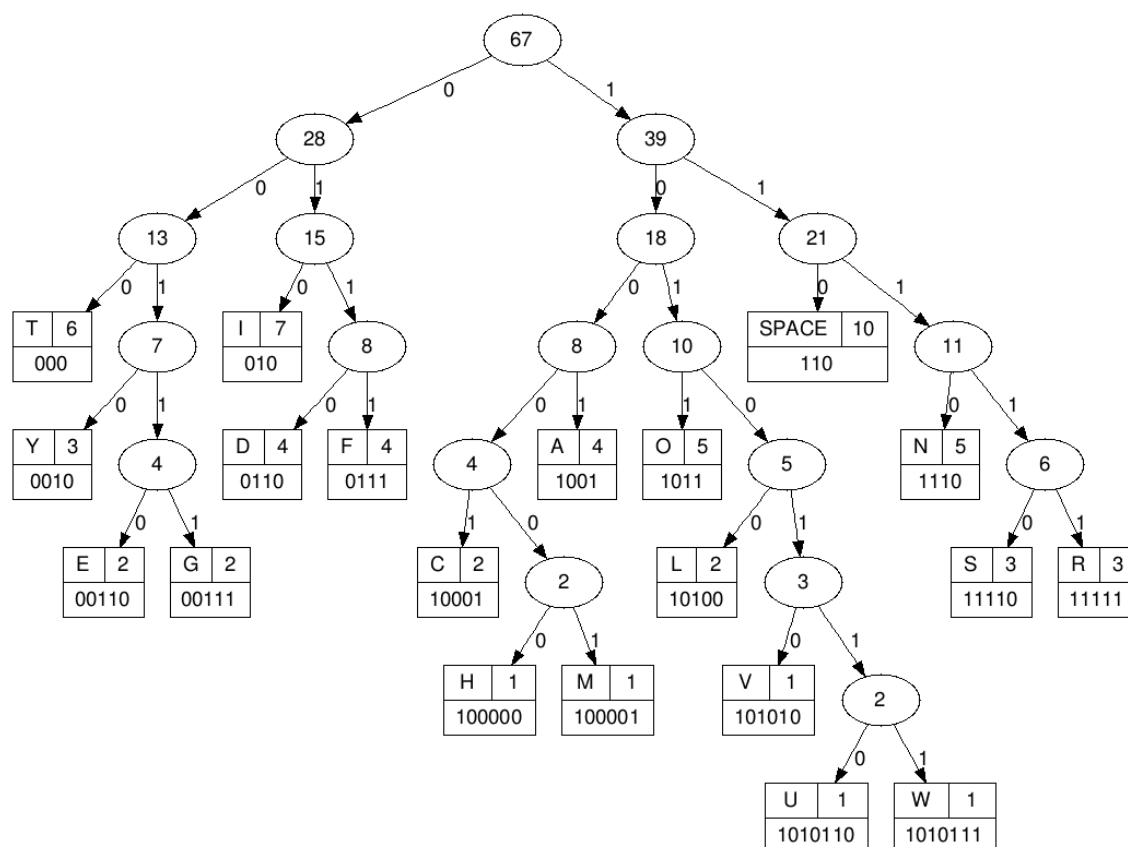


Figure 2.7: Huffman tree.

## 2.2 Same-origin policy

Same-origin policy is an important aspect of the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of [Uniform Resource Identifier](#) scheme, hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's [Document Object Model](#).

The following table explains same-origin-policy. We assume that we want access from *http://www.example.com/dir/page.html* to each of the following URLs:

<a href="http://www.example.com/dir/page2.html">http://www.example.com/dir/page2.html</a>	<b>success</b>
<a href="http://www.example.com/dir2/other.html">http://www.example.com/dir2/other.html</a>	<b>success</b>
<a href="http://www.example.com:81/dir/other.html">http://www.example.com:81/dir/other.html</a>	different port
<a href="https://www.example.com/dir/other.html">https://www.example.com/dir/other.html</a>	different protocol
<a href="http://en.example.com/dir/other.html">http://en.example.com/dir/other.html</a>	different host
<a href="http://example.com/dir/other.html">http://example.com/dir/other.html</a>	different host

This mechanism is particularly significant for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions. The lack of same origin policy would result in the compromise of data confidentiality or integrity.

Despite the use of same-origin policy by modern browsers, there still 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 (XSS) and cross-site request forgery (CSRF) are described in the following subsections.

### 2.2.1 Cross-site scripting

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. That way, same-origin policy can be bypassed and sensitive data handled by the vulnerable website may be compromised.

XSS attacks can generally be categorized into two categories: *stored and reflected*.

Stored XSS Attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum. The victim then retrieves the malicious script from the server when it requests the stored information.

Reflected XSS Attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

For further information on XSS refer to [\[10\]](#).

### 2.2.2 Cross-site request forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. An attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

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 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.

## 2.3 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol that provides communications security over a computer network, allowing a server and a client to communicate in a way that prevents eavesdropping, tampering or message forgery.

TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The Record Protocol provides connection security, while the Handshake Protocol allows the server and client to authenticate each other and negotiate encryption algorithms and cryptographic keys before any data is exchanged.

One category of TLS attack is compression attacks [13]. Such attacks exploit TLS-level compression in order to decrypt ciphertext. In this work, we extend the usability and optimize the performance of such an attack, [BREACH](#).

### 1.1.1 TLS handshake

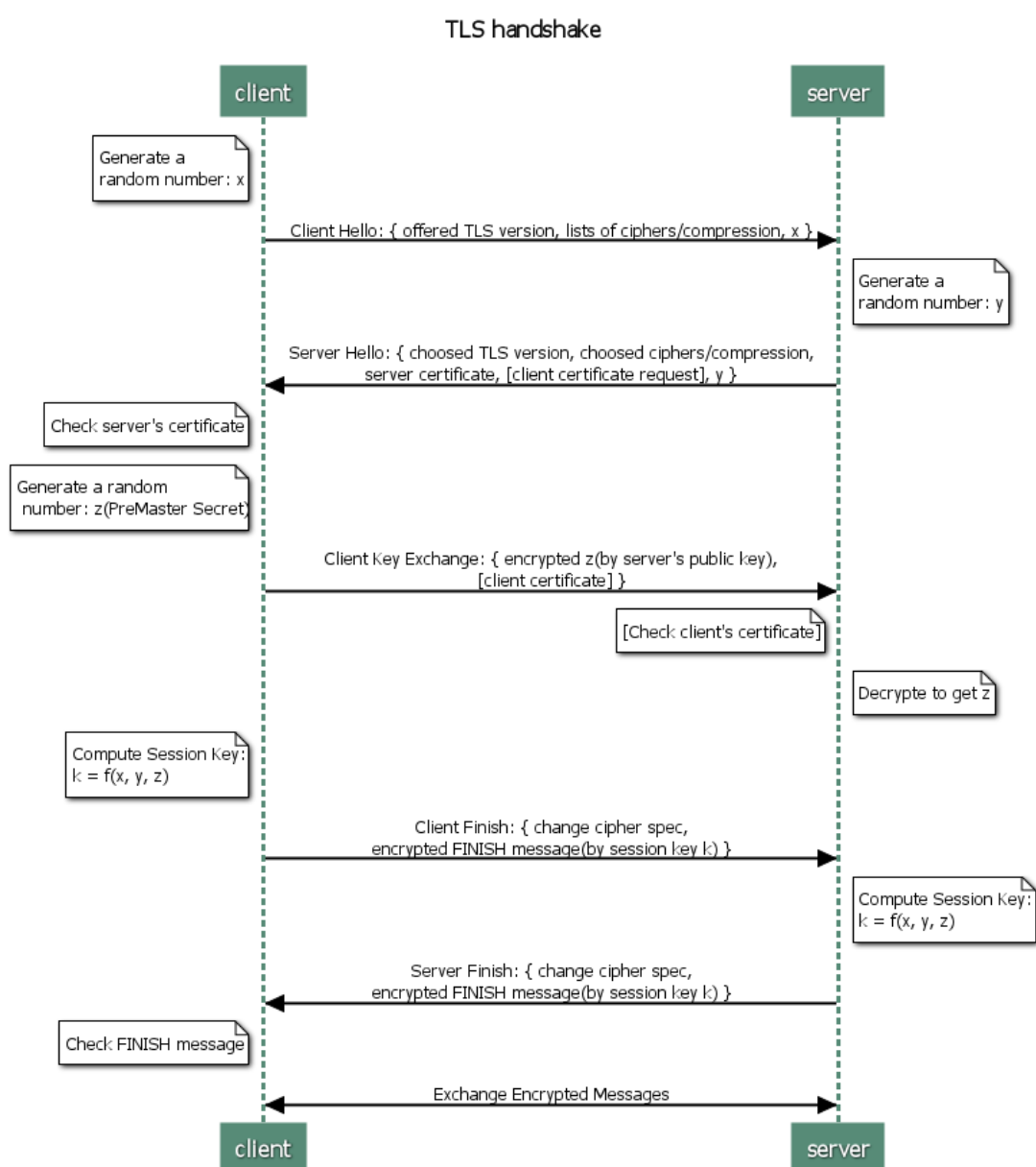


Figure 2.8: TLS handshake flow.

The above sequence diagram presents the functionality of a TLS handshake. The client and the server exchange the basic parameters of the connection such as the highest

TLS protocol version, a random number, a list of suggested cipher suites and suggested compression methods. The server provides the client with all the necessary information in order to validate and use the asymmetric server key to compute the symmetric key that will be used for the rest of the communication. The client computes the Pre-MasterSecret and sends it 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 applies only in the basic TLS handshake. Client-authenticated and resumed handshakes are quite different, although they are not relevant for the purpose of this work.

### 1.1.2 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:

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:  $256 \_ (bits15::8) + (bits7::0)$ .

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

A [Man-in-the-Middle](#) attack is a type of cyber attack where a malicious actor inserts themselves into a conversation between two parties, impersonates both and gains access to information that they were trying to send to each other. A man-in-the-middle attack allows a malicious actor to intercept, send and receive data meant for someone else, or not meant to be sent at all, without either outside party knowing until it is too late.

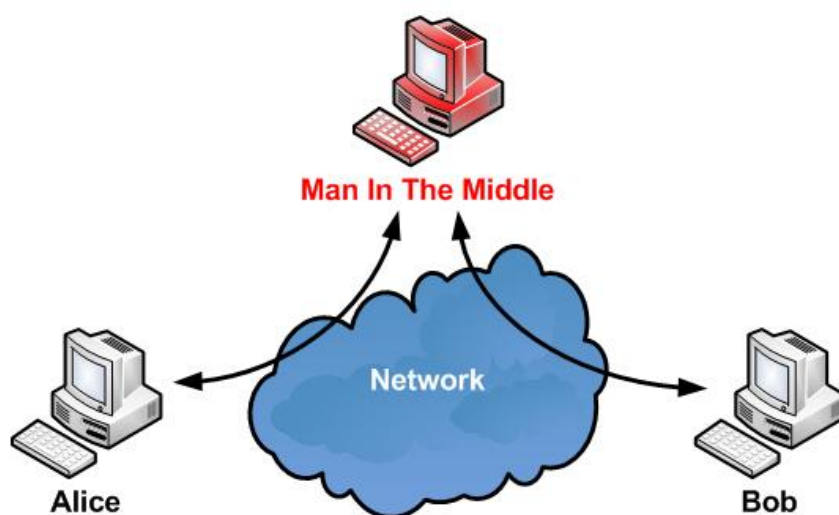


Figure 2.10: Man-in-the-Middle.

MitM attacks can be mitigated using end-to-end encryption, mutual authentication or PKIs. However, some attacks are still feasible against poorly configured end-points.

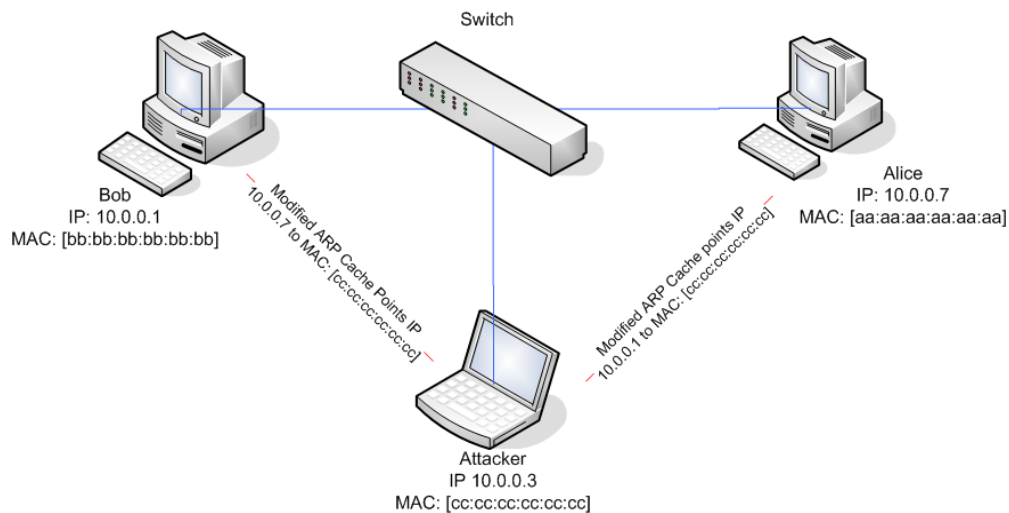
Below we describe one such attack, ARP Spoofing.

### 2.4.1 ARP Spoofing

ARP spoofing [8] is a type of attack in which an attacker sends falsified ARP (Address Resolution Protocol)[11] messages over a local area network. This results in the linking of an attacker's MAC address with the IP address of a legitimate computer or server on the network. This enables the attacker to begin receiving any data that is intended for that IP address. ARP spoofing can enable malicious parties to intercept, modify or even stop data in-transit.

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.



**Figure 2.11: ARP spoofing.**

### 3. RUPTURE FRAMEWORK

In this chapter, we describe our framework for conducting compression side-channel attacks, Rupture. Rupture [3] is a service-based architecture system which contains multiple independent components.

The section 3.1 describes in detail the assumptions needed in order to orchestrate the attack. The section 3.2 describes the terminology used in our framework and its governing principles. Finally, the section 3.3 presents the multiple components thoroughly. While the components are designed to be able to run independently on different networks or computer systems, the attack can be performed by running all subsystems on an individual system. We provide appropriate scripts to conduct such attacks easily.

#### 3.1 Attack Assumptions

The attack framework assumes a *target* service to be attacked. Typically this target service is a web service which uses TLS. Specifically, we are targeting services that provide HTTPS end-points. However, this assumption can be relaxed and attacks against other similar protocols are possible. Any protocol that exchanges encrypted data on the network and for which a theoretical attack exists can in principle be attacked using Rupture. We designed Rupture to be a good playground for experimentation for such new attacks. Examples of other encrypted protocols for which attacks can be tested include SMTP and XMPP.

The attack also assumes a user of the target service for which data will be decrypted, the *victim*. The victim is associated with a particular target.

There are two underlying assumptions in our attack: The injection and the sniffing assumptions. These are often achieved through the same means, although not necessarily.

The injection assumption states that the adversary is able to inject code to the victim's browser for execution. This code is able to issue adaptive requests to the target service.

Injection in Rupture is achieved through the *injector* component. The code that is injected is the *client* component.

The sniffing assumption states that the adversary is able to observe network traffic between the victim and the target. This traffic is typically ciphertexts. Sniffing is achieved through the *sniffer* component.

Both the *sniffer* and *client* will be described in section 3.3.

#### 3.2 Principles of Attack

The attack takes place by first injecting client code into the victim's computer using the injector. The client then opens a command-and-control channel to the real-time service,



which forwards work from the backend to the client. The real-time service facilitates the communication between the client and backend and the backend module makes all the decisions for the attack. Further description for the real-time and the backend is provided in 3.3.

When a client associated with a victim asks to work, the backend passes a work request to the real-time service, which passes it to the client. These work requests ask the client to perform a series of network requests from the victim's computer to the target web application. As these requests are made from the victim's browser, they contain authentication cookies which authenticate the user to the target service. As such, the responses contain sensitive data, but that data is not readable by the client due to same-origin policy.

When a response arrives from the target web app to the victim's computer, the encrypted response is collected by the sniffer on the network. The encrypted data pertaining to one response is a *sample*. Each work asks for multiple requests to be made, and therefore multiple samples are collected per work. The set of samples collected for a particular work request are a *sampleset*.

```
def _sampleset_to_work(self, sampleset):
    return {
        'url': self._url(sampleset.candidatealphabet),
        'amount': self._victim.target.samplesize,
        'alignmentalphabet': sampleset.alignmentalphabet,
        'timeout': 0
    }
```

Figure 3.1: Sampleset

A successful attack completely decrypts a portion of the plaintext. The portion of the plaintext which the attack tries to decrypt is the *secret*. That portion is identified by an initially known prefix which distinguishes it from other secrets. This prefix is typically 3 to 5 bytes long. A prefix of such a length is required to bootstrap the attack due to the LZ77 implementation. Each byte of the secret can be drawn from a given *alphabet*, the secret's alphabet. For example, some secrets only contain numbers, and so their alphabet is the set of numbers [0-9].

At each stage of the attack, a prefix of the secret is known, because that portion of the secret has already been successfully decrypted. The known prefix gets extended until the whole secret becomes known, at which stage the attack is completed.

When a certain prefix of the secret is known, the next byte of the secret must be decrypted. The attack initially assumes the next unknown byte of the secret exists in the secret's alphabet, but slowly drills down and rejects alphabet symbols until only one candidate symbol remains. At each stage of the attack on one byte of the secret, there is a certain *known alphabet* which the next byte can belong to. This known alphabet is a subset of the secret's alphabet.

To drill down on the known alphabet, one of two methods is employed. In the *serial method*, each symbol of the known alphabet is tried sequentially. In the *divide & conquer method*, the alphabet is split into two *candidate alphabet* subsets which are tried independently.

```
def _build_candidates_divide_conquer(self, state):
    candidate_alphabet_cardinality = len(state['knownalphabet']) / 2

    bottom_half = state['knownalphabet'][:candidate_alphabet_cardinality]
    top_half = state['knownalphabet'][candidate_alphabet_cardinality:]

    return [bottom_half, top_half]
```

**Figure 3.2: Divide & Conquer Alphabet**

The above figure shows how the initial alphabet is divided into two equal alphabets each of which will be tested separately.

The attack is conducted in *rounds*. In each round, a decision is made about the state of the attack and more becomes known about the secret. In a round, either the next byte of the secret becomes known, or the known alphabet is drilled down to a smaller set. In order to compare various different candidate alphabets, the attack executes a series of steps to collect *batches* of data collection for each round.

In each batch, several samples are collected from each probability distribution pertaining to a candidate alphabet, forming a *sampleset*. When samplesets of the same amount of samples have been collected for all the candidate alphabets, a batch is complete and the data is analyzed. The analysis is performed by the *analyzer* which statistically compares the samples of different candidates and decides which candidate is optimal, i.e. contains the correct guess. This decision is made with some *confidence*, which is expressed in bytes. If the confidence is insufficient, an additional batch of samplesets is collected, and the analysis is redone until the confidence value surpasses a given threshold.

Once enough batches have been collected for a decision to be made with good confidence, the round of the attack is completed and more information about the secret becomes known. Each round at best collects one bit of information of the secret.

```

#####
Candidate scoreboard:
d: 16576
a: 16592
c: 16592
b: 16592
e: 16592
g: 16592
f: 16592
i: 16592
h: 16592
k: 16592
j: 16592
m: 16592
l: 16592
o: 16592
n: 16592
q: 16592
p: 16592
s: 16592
r: 16592
u: 16592
t: 16592
w: 16592
v: 16592
y: 16592
x: 16592
z: 16592
#####
Decision:
state: {'knownalphabet': u'abcdefghijklmnopqrstuvwxyz', 'knownsecret': u'imperd'}
confidence: 1.0
#####

```

Figure 3.3: Scoreboard

The above figure shows the results of the analysis of one batch. For each candidate letter we present a number, which is the total length of the samplesets for the specific letter. The candidate letters are presented in an ascending order of the total length.

Under the scoreboard is the decision which consists of the known alphabet, the possible knownsecret and the confidence describing how sure we are regarding the possible knownsecret.

### 3.3 Architecture

#### 3.3.1 Injector

The injector component is responsible for injecting code to the victim's computer for execution. In our implementation, we assume the adversary controls the network of the victim. Our injector injects the client code in all unauthenticated HTTP responses that the victim receives. This Javascript code is then executed by the victim's browser in the context of the respective domain name. We use BetterCap [1] to perform the HTTP injection. The injection is performed by ARP spoofing the local network and forwarding all traffic in a Man-in-the-Middle manner. It is simply a series of shell scripts that use the appropriate BetterCap modules to perform the attack.

As all HTTP responses are infected, this provides increased robustness. The injected client code remains dormant until it is asked to wake up by the command-and-control channel. This means that the user can switch between browsers, reboot their computer, close and reopen browser tabs, and the attack script will continue to be injected.

As long as one tab with the client script is open, the attack can keep running.

The injector component needs to run on the victim's network and as such is lightweight and stateless. It can be easily deployed on a machine such as a Raspberry Pi, and can

be used for massive attacks, for example at public networks such as coffee shops or airports. Multiple injectors can be deployed to different networks, all controlled by the same central command-and-control channel.

### 3.3.2 Sniffer

The sniffer component is responsible for collecting data directly from the victim's network. As the client issues chosen plaintext requests, the sniffer collects the respective ciphertext requests and ciphertext responses as they travel on the network. These encrypted data are then transmitted to the backend for further analysis.

Our sniffer implementation runs on the same network as the victim. It is a Python program which uses `scapy` [2] to collect network data.

Our sniffer runs on the same machine as our injector and utilizes the injector's ARP spoofing to retrieve the data from the network. Other sniffer alternatives include sniffing data on the target network side, or on the ISP or router point if the adversary has such a level of access.

The sniffer exposes an HTTP API which is utilized by the backend for controlling when sniffing starts, when it is completed, and to retrieve the data that was sniffed. This API is described below.

#### Backend ↔ Sniffer (HTTP)

The Python backend application communicates with the sniffer server, in order to initiate a new sniffer, get information or deletes an existing one. The sniffer server implements a RESTful API for communication with the backend.

**/start** is a POST request that initializes a new sniffer. Upon receiving this request, the sniffer service should start sniffing.

The request contains a JSON with the *source\_ip*, (the IP of the victim on the local network) and the *destination\_host* (the hostname of the target that is being attacked).

The backend returns *HTTP 201* if the sniffer is created correctly. Otherwise, it returns *HTTP 400* if either of the parameters is not properly set, or *HTTP 409 - Conflict*, if a sniffer for the given arguments already exists.

**/read** is GET request that asks for the network capture of the sniffer.

The GET parameters are the *source\_ip* (the IP of the victim on the local network) and the *destination\_host* (the hostname of the target that is being attacked).

The backend returns *HTTP 200* with a JSON that has a field *capture*, which contains the network capture of the sniffer as hexadecimal digits, and a field *records*, that contains the total amount of captured TLS application records. In case of error, *HTTP 422 Unprocessable Entity* is returned if the captured TLS records were not properly formed on the sniffed network, or *HTTP 404* if no sniffer with the given parameters exists.

**/delete** is a POST request that asks for the deletion of the sniffer

The request contains a JSON with the *source\_ip* (the IP of the victim on the local network) and *destination\_host* (the hostname of the target that is being attacked).

The backend Returns *HTTP 200* if the sniffer was deleted successfully, or *HTTP 404* if there is no sniffer with the given parameters.

### 3.3.3 Client

The client is written in Javascript and runs in a different context from the target website. Thus, it is subject to same-origin policy and cannot parse the plaintext or encrypted responses. However, the encrypted requests and responses are available to the sniffer through direct network access.

The client contains minimal logic. It connects to the real-time service through a command-and-control channel and registers itself there. Afterwards, it waits for work instructions by the command-and-control channel, which it executes. The client does not take any decisions or receive data about the progress of the attack other than the work it is requested to do. This is intentional so as to conceal the workings of the adversary analysis mechanisms from the victim in case the victim attempts to reverse engineer what the adversary is doing. Furthermore, it allows the system to be upgraded without having to deploy a new client at the victim's network, which can be a difficult process.

As a regular user is browsing the Internet, multiple clients will be injected in insecure pages and they will run under various origins. All of them will register and maintain an open connection through a command-and-control channel with the real-time service. The real-time service will enable one of them for this victim, while keeping the others dormant. The one enabled will then receive work instructions to perform the required requests. If the enabled client dies for whatever reason, such as a closed tab, one of the rest of the clients will be woken up to take over the attack.

The client is a Javascript program written using harmony / ECMAScript 6 and compiled using babel and webpack.

```

doWork(work) {
  const {url, amount, alignmentalphabet} = work;

  if (typeof url == 'undefined') {
    this.noWork();
    return;
  }
  console.log('Got work: ', work);

  const reportCompletion = (success) => {
    if (success) {
      console.log('Reporting work-completed to server');
    }
    else {
      console.log('Reporting work-completed FAILURE to server');
    }

    this._socket.emit('work-completed', {
      work: work,
      success: success,
      host: window.location.host
    });
  }

  req.Collection.create(
    url,
    {amount: amount, alignmentalphabet: alignmentalphabet},
    function() {},
    reportCompletion.bind(this, true),
    reportCompletion.bind(this, false)
  );
};

```

Figure 3.4: Client code

### 3.3.4 Real-time

The real-time service is a service which waits for work requests by clients. It can handle multiple connections. It receives command-and-control connections from various clients which can live on different networks, orchestrates them, and tells them which ones will remain dormant and which ones will receive work, enabling one client per victim.

The real-time service is developed in Node.js.

The real-time service maintains open web socket command-and-control connections with clients and connects to the backend service, facilitating the communication between the two.

The real-time server forwards work requests and responses between the client and the backend. It communicates with the client in a bi-directional way using web sockets.

This also facilitates the ability to detect that a client has gone away, which is registered as a failure to do work. This can happen for example due to network errors if the victim disconnects from the network, closes their tab or browser, and so on. It is imperative that incomplete work is marked as failed as soon as possible so that the attack can continue by recollecting the failed samples.

```
const createNewWork = () => {
  const getWorkOptions = {
    host: BACKEND_HOST,
    port: BACKEND_PORT,
    path: '/breach/get_work/' + victimId
  };

  winston.debug('Forwarding get_work request to backend URL ' + getWorkOptions.path);

  const getWorkRequest = http.request(getWorkOptions, (response) => {
    let responseData = '';
    response.on('data', (chunk) => {
      responseData += chunk;
    });
    response.on('end', () => {
      try {
        client.emit('do-work', JSON.parse(responseData));
        winston.info('Got (get-work) response from backend: ' + responseData);
      }
      catch (e) {
        winston.error('Got invalid (get-work) response from backend');
        doNoWork();
      }
    });
  });
  getWorkRequest.on('error', (err) => {
    winston.error('Caught getWorkRequest error: ' + err);
    doNoWork();
  });
  getWorkRequest.end();
};
```

Figure 3.5: Real-time code

The web socket API exposed by the real-time service is explained below.

## Client ↔ Real-time protocol

The client / real-time protocol is implemented using socket.io websockets.

### client-hello / server-hello

When the client initially connects to the real-time server, it sends the message *clienthello* with its *victim\_id* to the real-time server. The server responds with a *serverhello* message. After these handshake messages are exchanged, the client and server can exchange further messages.

### get-work / do-work

When the client is ready to perform work, it emits the message *get-work* requesting work to be performed from the real-time server. The real-time server responds with a *do-work* message, passing a *work* object that is structured as defined below:

```
typedef work
  amount: int
  url: string
  timeout: int (ms)
```

Figure 3.6: work object

If the real-time service is unable to retrieve work from the backend due to a communication error, real-time will return an empty work object indicating there is no available work to be performed at this time.

## work-completed

When the client has finished its work or has been interrupted due to network error, it emits a *work-completed* message, containing the following information:

```
work: work
success: bool
```

Figure 3.7: work-completed

*success* is *true* if all requests were performed correctly, otherwise it is *false*. *work* contains the work that was performed or failed to perform.

### 3.3.5 Backend

The backend is responsible for strategic decision taking, statistical analysis of samples collected, adaptively advancing the attack, and storing persistent data about the attacks in progress for future analysis.

The backend talks to the real-time service for pushing work out to clients. It also speaks to the sniffer for data collection.

It is implemented in Python using the Django framework.

The backend exposes a RESTful API via HTTP to which the real-time service makes requests for work. This API is explained below.

#### Real-time → Backend (HTTP)

The backend implements various API endpoints for communication with the real-time server.

**/get\_work/<victim>** is an HTTP GET endpoint. It requests work to be performed on behalf of a client. The argument passed is the *victim* - the id of the victim.

If there is work to be done, it returns an *HTTP 200* response with the JSON body containing the work structure. The samples associated with a particular work request and performed all together constitute a *sampleset*.



In case no work is available for the client, it returns an HTTP '404' response. Work can be unavailable in case a different client is already collecting data for the particular victim, and we do not wish to interfere with it.

**/work\_completed/<victim>** is an HTTP POST endpoint. It indicates on behalf of the client that some work was successfully or unsuccessfully completed. The arguments passed are the *work* and a boolean *success* parameter.

If *success* is *True*, this indicates that the series of indicated requests were performed by the victim correctly. Otherwise, the victim failed to perform the required requests due to a network error or a timeout and the work has to be redone.

## 4. BACKTRACKING

Backtracking is a method that allows the adversary to recover the desired target secret of an HTML page, whose known prefix can be encountered multiple times. This method is implemented in a different way than the other two (Serial, Divide & conquer), since it does not require searching for an optimal candidate each round. Instead, it adds every possible candidate on a pool and chooses the best one to be examined on the next round, based on a probabilistic model. Then, it advances repetitively until the whole secret is recovered.

In section 1 we describe in detail how Backtracking implementation is structured, as well as how the method works.

Section 2 explains the architecture of the new patch and analyzes the crucial components responsible for conducting a Backtracking attack.

In this chapter, we only compare Backtracking characteristics with those of the Serial one. Even though Divide & Conquer works in a different way than Serial during each round, it does reach to an optimal candidate in the same way the Serial does, making the comparison between the first two sufficient.

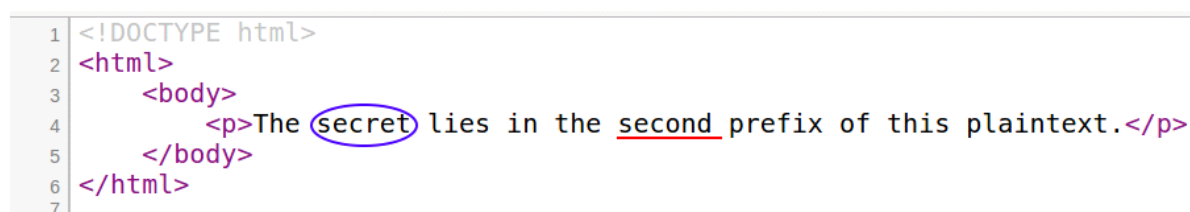
### 4.1 Method Specifications

When it comes to analyzing compressed ciphertexts, Backtracking deals with the problem of secret discovery in a more spherical way.

In contrast with the Serial, this method does not produce unexpected results when the target secret's prefix can be found multiple times on the HTML page.

Serial method creates multiple requests for every possible character, produced by a given alphabet. During each round it decides an optimal candidate by analyzing the size of the compressed cipher responses. The response with the smallest size, is marked as the optimal candidate and is added to the known prefix in order to proceed to the next round.

But what happens if there are two (or possibly more) words with the same prefix on the HTML page we examine. Here is an example where `sec` is the known prefix, `secret` is the target secret word and `second` is also part of the HTML page:



```

1 <!DOCTYPE html>
2 <html>
3   <body>
4     <p>The secret lies in the second prefix of this plaintext.</p>
5   </body>
6 </html>
7

```

Figure 4.1: Common prefix plaintext.

As the above image shows, both the 'r' character as well as 'o' character will give the same size of compressed responses, since both can be compress well with the prefix

`sec`. Although there is no violation of any kind, based on the implementation of the Serial method, there is a chance the 'o' character will be chosen instead of the 'r' character, giving the wrong outcome. During the next round, the updated knownsecret will be the string 'seco', leading to the discovery of word `second` instead of our target secret.

On the other hand, Backtracking works in a different way than the previous method. Instead of deciding the optimal candidate on each round, Backtracking assigns relative probability values on each candidate based on the compression they produce. The lower the compression size is, the higher the value goes. So, instead of working in a linear structure, where each round we create is the one to be chosen on the next iteration, Backtracking creates multiple rounds for each candidate and stores them in a tree structure for future analysis. Here is how backtracking works in a step by step analysis:

As the Rupture architecture states, every round starts with a given knownsecret. During the first iteration of the attack, the knownsecret is equal to the initial known prefix of the target secret. Every node represents a round and holds a knownsecret. It is easy to assume that the top node of the tree is the first round, where the initial known prefix is stored.

Based on that, the framework starts to make requests on a targeted website, in order to measure its responses. When we receive those responses, we analyze them by assigning relative probabilities to each one of them. This assignment aims to translate the compression size of a candidate into a probability. After this process is completed, we store each one of the candidates into the tree structure. This is done by creating new rounds with the permutation of knownsecret and the appropriate candidate. This technique is executed repetitively until the target secret is recovered.

It is quite clear, that if we pick a random node and follow its path, starting from the top, we can observe every step of the analysis process for the given knownsecret. The key aspect though, behind each node is that it doesn't only store a string with a knownsecret, but it also holds an accumulated probability number which represents the certainty of the knownsecret being a part of the target secret. This value is calculated by combining the relative probability of each candidate at a given round and the depth of the current node. Accumulated probability is a very valuable variable, since we use it in order to decide the next round to be analyzed. The one with the highest value is the one to be chosen.

This technique give us the ability to explore more than one known prefixes at a time, ensuring that if a conflict arises in a round (as stated above, two words having the same prefix) we can solve it by exploring all possible secrets until we find the desired one.

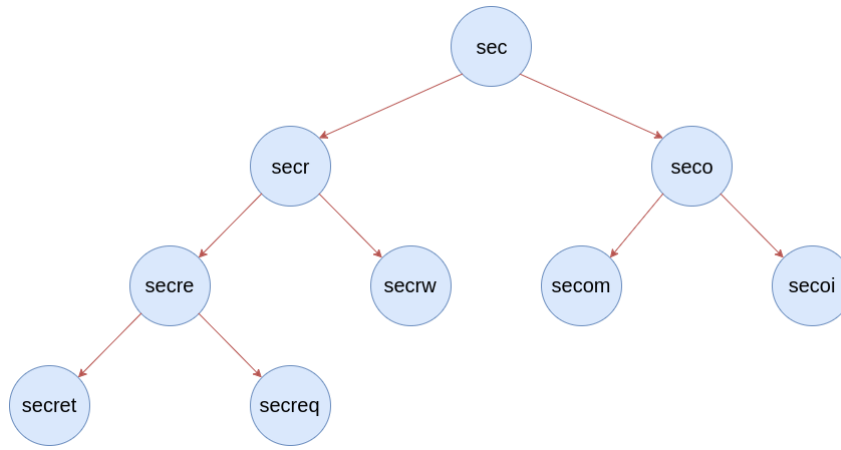


Figure 4.2: Backtracking tree implementation.

## 4.2 Architecture

### 4.2.1 Round Model

In order to support the Backtracking tree structure, there was a need for the Round model to undergo a few changes. Since Serial method works in a linear way, Round's old form simply starts every time we want to explore our knownsecret and ends when we gather a proper amount of data, in order to extract a decision.

However, when we need to explore multiple paths of the same prefix, this approach doesn't work. We need to create Rounds with various knownsecrets and store them in the database for future use. This is done by the fields `started` and `completed` on Round model. These two fields not only allow us to store multiple possible candidates in the form of a Round, but also help us to distinguish which Round we are currently working on, which one is already completed and which one is about to be explored.

Another important part for the Backtracking tree structure is the *accumulated\_probability* field. This field holds the amount of certainty of each Round's knownsecret being in the final secret.

### 4.2.2 Backtracking Analyzer

Backtracking Analyzer is responsible for three major tasks. Firstly, to calculate relative probabilities for each character from a given knownalphabet, secondly calculate accumulated probabilities for those characters and finally to return all these values and states.

#### ❖ Relative Probabilities

The function *get\_accumulated\_probabilities* takes up the first two tasks. It is given a dictionary of sorted candidate alphabets and it calculates the relative probabilities of each candidate being in the target secret based on their associated accumulative lengths. Since the length significance of the better compressed candidates is more important than those with the least compression, the function connecting each candidate's probability cannot be linear. This is why we use an exponential function.

The formula that calculates the relative probability of the  $i^{\text{th}}$  candidate is:

$$\frac{b^{i-min}}{\sum_{k=1}^{k=S} b^{k-min}}$$

**Figure 4.3: Relative probability formula.**

- $b$ : compression function factor
- $i$ :  $i^{th}$  candidate compression length
- $min$ : candidate with the least compression length
- $S$ : alphabet' s size

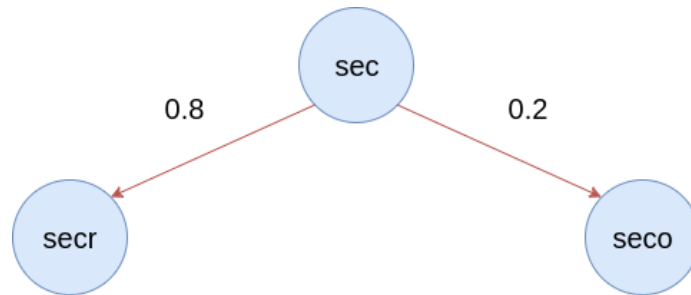
Base  $b$  represents the efficiency of the compression function. A good compression function indicates that the total length we collect is rather accurate, increasing the significance of the compression difference between two candidates.

The key concept behind this formula is the effort to quantify how much a given candidate differs from the minimum candidate and the average difference. This is all translated with a probability value. Here is an example of the usage of that formula:

$r$  compression length: 45

$o$  compression length: 47

$b$ : 1.2



**Figure 4.4 Relative probability example.**

As the above image shows, if the alphabet for the first round consists of characters  $r$  and  $o$ , given their compression length their relative probability will be 0.8 and 0.2 respectively. This example demonstrates accurately the importance of the compression function factor in the calculation of relative probabilities, as we can configure the constant according to the nature of the problem (compression function algorithm) and ultimately create more accurate results.

### ❖ Accumulated Probabilities

Once the relative values are calculated then *get\_accumulated\_probabilities* associates them with the probability of the parent Round and calculates the final accumulated probability.

In order to get more accurate results it is essential that each candidate that is analyzed should have the appropriate weight of importance. So what happens if we want to choose the next round, but two or more candidates do not have the same depth on the tree structure? Is it enough to compare only the relative values?

This is where accumulated probability comes in. The calculation of this value is done by the following formula:

$$AP_i = AF * RP_i * AP_{i-1}$$

- $AF$ : amplification factor
- $RP_i$ : relative probability of current candidates
- $AP_{i-1}$ : parent's accumulated probability

The Amplification factor is the most crucial part in this formula. This constant states that the deeper a candidate is on the Backtracking tree structure, the more important it gets. This can also be verified intuitively. As long as we make “optimal” candidate choices and we still get “optimal” compression length, then it is more likely that the next choice we are about to make, will be “optimal” too. Here is an example of a Backtracking tree, with both relative and accumulated probabilities:

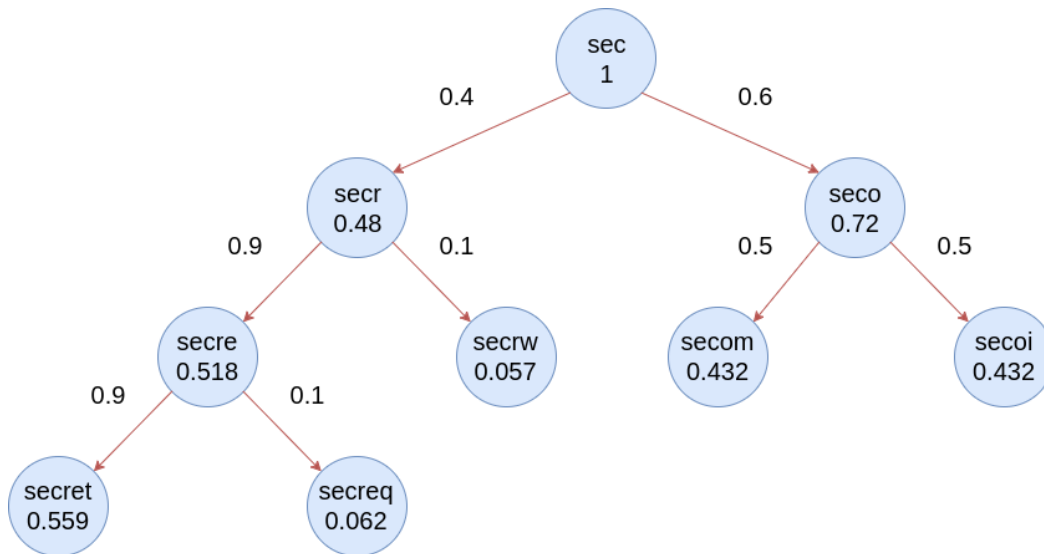


Figure 4.5: Complete Backtracking graph with all types of probabilities

### 4.3 Backtracking experimental results

The Backtracking method was tested on lab environment attacks. More specifically, we created a test end-point containing two words with the same prefix and tried to recover both of them, based on the theoretical architecture. The first one is the word `secret` and the other one is the word `second`. Our goal was to confirm our hypothesis:

At first, Rupture will identify both of the secrets by appearing two candidates with significant accumulated probability, compared to the other ones. This can be shown in the following figure:

```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba... dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:488] Length: 63968
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:04:48] "POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:216] Found 1 unstated samplesets
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:04:48] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:04:48] "GET /breach/get_work/22 HTTP/1.1" 200 194
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:488] Length: 63958
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:290] {'probability': 0.7211538461538461, 'knownsecret': 'u'seco', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:290] {'probability': 0.15137019230769233, 'knownsecret': 'u'secr', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:290] {'probability': 0.03173076923076923, 'knownsecret': 'u'secj', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:290] {'probability': 0.03173076923076923, 'knownsecret': 'u'secs', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:415] Created new round:
[22/Jul/2017 15:04:49] DEBUG [breach.strategy:416] Known secret: seco

```

Figure 4.6: Backtracking execution step 1.

Both `seco` as well as `secr` have a distinguishable probability value compared to the other candidates (we only show two of them for simplicity purposes). This hints that characters `o` and `r` could possibly be part of two different secrets. As the attack continues, Backtracking analyzer chooses the candidate with the maximum value, in our case `o` and continues the analysis:

```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba... dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture dimitris@alleria: ~/Documents/rupture
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:488] Length: 64056
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:05:17] "POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:216] Found 1 unstated samplesets
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:05:17] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:05:17] "GET /breach/get_work/22 HTTP/1.1" 200 195
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:488] Length: 64056
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:290] {'probability': 0.16039558949809113, 'knownsecret': 'u'secon', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:290] {'probability': 0.06664436743645687, 'knownsecret': 'u'secoc', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:290] {'probability': 0.040916914880963044, 'knownsecret': 'u'secom', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:290] {'probability': 0.040916914880963044, 'knownsecret': 'u'secol', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:05:18] DEBUG [breach.strategy:415] Created new round:

```

Figure 4.7: Backtracking execution step 2.

Given the prefix `secon` as knownsecret Backtracking continues by extending it until the whole secret is recovered.



```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba...  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:488] Length: 64126
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:05:43] POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:216] Found 1 unstarted samplesets
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:05:43] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:05:43] GET /breach/get_work/22 HTTP/1.1" 200 196
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:488] Length: 64126
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:290] {'probability': 0.160993565598807, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:290] {'probability': 0.0008210671845543237, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:290] {'probability': 0.0005151794099164384, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:290] {'probability': 0.0005151794099164384, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:05:44] DEBUG [breach.strategy:415] Created new round:

```

Figure 4.8: Backtracking execution step 3.

At this point, since the target secret `second` is recovered, we expect that on the next iteration Backtracking is going to get insignificant probability value for every single candidate, since none of them belongs to the actual secret.

```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba...  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:488] Length: 64018
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:06:12] POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:216] Found 1 unstarted samplesets
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:06:12] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:06:12] GET /breach/get_work/22 HTTP/1.1" 200 197
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:488] Length: 64038
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:290] {'probability': 0.027372321174738305, 'knownsecret': 'u'secondf', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:290] {'probability': 0.016803867969171844, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:290] {'probability': 0.016803867969171844, 'knownsecret': 'u'second', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:290] {'probability': 0.016316627850758868, 'knownsecret': 'u'secondq', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:06:13] DEBUG [breach.strategy:415] Created new round:

```

Figure 4.9: Backtracking execution step 4.

Since our hypothesis is confirmed by the probability values, Backtracking chooses as the next round to analyze the node with the prefix `secr`, which was stored in the tree structure, at the beginning of the attack. This is due to the fact that the node with the prefix `secr` was the one with the highest accumulated probability value, among every single not started Round models, saved in the tree.



```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba...  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@al
[22/Jul/2017 15:06:38] "POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:06:38] DEBUG [breach.strategy:216] Found 2 unstarted samplesets
[22/Jul/2017 15:06:38] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:06:38] DEBUG [breach.strategy:227] Candidate: y
[22/Jul/2017 15:06:38] "GET /breach/get_work/22 HTTP/1.1" 200 195
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:488] Length: 64030
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:06:39] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:06:40] "POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:06:40] DEBUG [breach.strategy:216] Found 1 unstarted samplesets
[22/Jul/2017 15:06:40] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:06:40] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:06:40] "GET /breach/get_work/22 HTTP/1.1" 200 195
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:488] Length: 64030
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:290] {'probability': 0.15784954009641175, 'knownsecret': 'u'secre', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:290] {'probability': 0.00012627963207712938, 'knownsecret': 'u'secrh', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:290] {'probability': 7.892477004820587e-05, 'knownsecret': 'u'secrq', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:290] {'probability': 4.735486202892352e-05, 'knownsecret': 'u'secrb', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:06:41] DEBUG [breach.strategy:415] Created new round:

```

Figure 4.10: Backtracking execution step 5.

This way Backtracking recovered both secrets, with high amount of certainty.

```

dimitris@alleria: ~/Documents/rupture
dimitris@alleria: ~/Documents/rupture/ba...  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@alleria: ~/Documents/rupture  dimitris@al
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:488] Length: 64160
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:267] candidatealphabet: y
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:07:21] "POST /breach/work_completed/22 HTTP/1.1" 200 18
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:216] Found 1 unstarted samplesets
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:226] Giving work:
[22/Jul/2017 15:07:21] DEBUG [breach.strategy:227] Candidate: z
[22/Jul/2017 15:07:21] "GET /breach/get_work/22 HTTP/1.1" 200 196
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:487] Work completed:
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:488] Length: 64140
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:489] Records: 64
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:266] Marking sampleset as completed:
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:267] candidatealphabet: z
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:268] roundknownalphabet: abcdefghijklmnopqrstuvwxyz
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:282] #####
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:288] Optimal Candidates:
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:290] {'probability': 0.16555990120990144, 'knownsecret': 'u'secret', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:290] {'probability': 1.6555990120990144e-05, 'knownsecret': 'u'secrea', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:290] {'probability': 1.6555990120990144e-05, 'knownsecret': 'u'secre', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:290] {'probability': 1.6555990120990144e-05, 'knownsecret': 'u'secrek', 'knownalphabet': 'u'abcdefghijklmnopqrstuvwxyz'}
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:298] #####
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:349] Checking max reflection length...
[22/Jul/2017 15:07:23] DEBUG [breach.strategy:415] Created new round:

```

Figure 4.11: Backtracking execution step 6.

## 5. IPV6

As stated before, the Rupture attack is based on controlling the victim's browser, while he visits websites that do not provide encrypted communication (HTTP connections). This situation creates the opportunity for an adversary to send arbitrary requests to HTTPS websites and as a result of that to allow him to measure the compressed response.

This technique requires from an adversary to perform a Man in the Middle (MitM) attack in order to inject javascript code to the victim's browser and ultimately send crafted requests to targeted endpoints. This is performed by the usage of the BetterCap framework, a Ruby gem that enables various MitM attacks, along with PacketFu [14], a mid-level packet manipulation library for Ruby.

However current versions of BetterCap as well as PacketFu, only support the IPv4 protocol. That being said, Rupture cannot target popular IPv6 endpoints such as CNN or IMDb making the extension of BetterCap for IPv6 protocol mandatory.

In this chapter we present both our patches for BetterCap, as well as PacketFu, that made the Rupture attack possible for IPv6 endpoints.

Section 1 is a brief introduction to the IPv6 theoretical background which is an essential part of the IPv6 attack.

In Section 2 we describe all the important changes of the IPv6 patch in BetterCap, along with the algorithm of the MitM injection attack.

### 5.1 Theoretical Background

#### 5.1.1 IPv6 Addressing Modes

##### ❖ Unicast

In unicast mode of addressing, an IPv6 interface (host) is uniquely identified in a network segment. The IPv6 packet contains both source and destination IP addresses. A host interface is equipped with an IP address which is unique in that network segment. When a network switch or router receives a unicast IP packet, destined to a single host, sends out that packet to one of its outgoing interface which connects to that particular host.

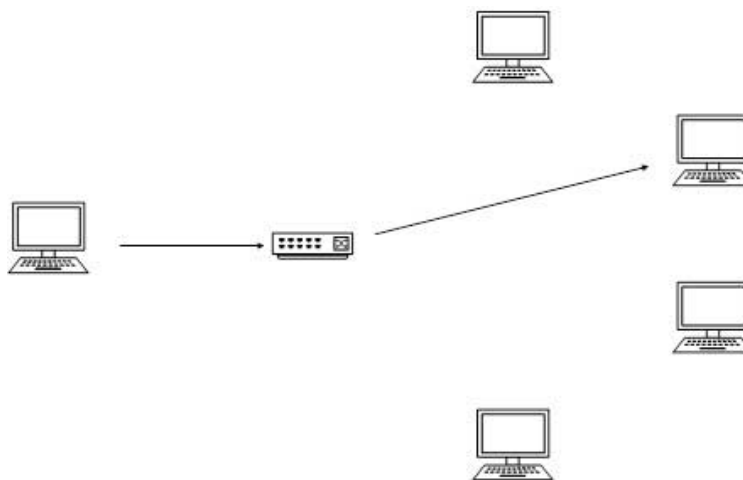


Figure 5.1: Unicast messaging.

### ❖ Multicast

The IPv6 multicast mode is the same as that of IPv4. The packet destined to multiple hosts is sent on a special multicast address. All hosts interested in that multicast information, need to join that multicast group first. All interfaces which have joined the group receive the multicast packet and process it, while other hosts not interested in multicast packets ignore the multicast information.

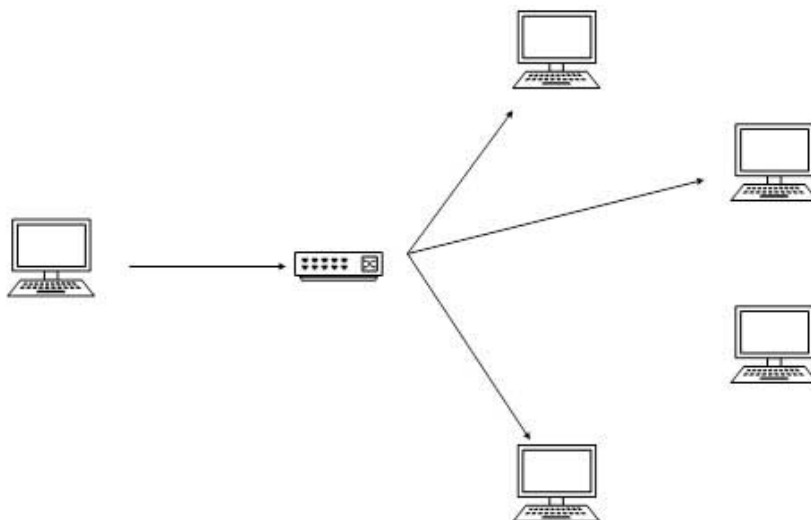


Figure 5.2: Multicast messaging.

### ❖ Anycast

IPv6 has introduced a new type of addressing, which is called Anycast addressing. In this addressing mode, multiple interfaces (hosts) are assigned the same Anycast IP address. When a host wishes to communicate with a host equipped with an Anycast IP address, it send a Unicast message. With the help of a complex routing mechanism, that Unicast message is delivered to the host closest to the sender, in terms of Routing cost.

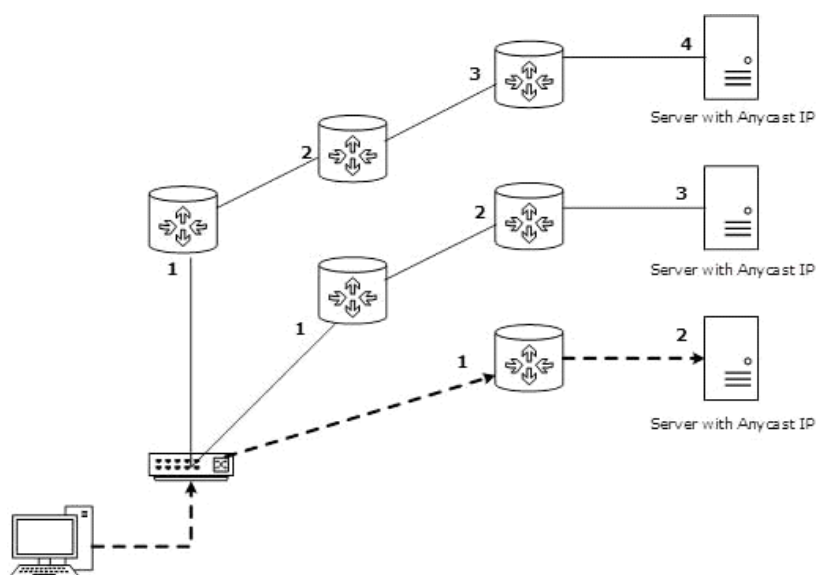


Figure 5.3: Multicast messaging.

### 5.1.2 IPv6 Addresses

An IPv6 address is represented as eight groups of four hexadecimal digits, each group representing 16 bits (two octets). The groups are separated by colons (:). An example of an IPv6 address is:

*2001:0db8:85a3:0000:0000:8a2e*

One or more consecutive groups of zero values may be replaced with a single empty group using two consecutive colons (::).

*2001:0db8:85a3::8a2e*

The first two types are mainly used for end-to-end IPv6 routing. On the other hand the last one has a limited use in private networks.

\* **Global Unicast Addresses** are similar to IPv4 public addresses and they are used for one to one communication by exchanging unicast messages. This mode dictates that the sender's outgoing packets are destined to a single host. Global Addresses can be acquired with three different ways, either by DHCPv6, SLAAC or Static Configuration. [DHCPv6](#) is the equivalent protocol of IPv4, DHCP, dynamically configuring hosts with IPv6 addresses, IP prefixes and other configuration data required to operate on the network. [Stateless Address Autoconfiguration](#) (SLAAC) protocol gives the ability to hosts to configure themselves automatically when connected to an IPv6 network. SLAAC is the most preferable method in IPv6 configuration, unless an application imposes otherwise. Static Configuration is executed manually in order to assign a host with a static IPv6 Global Address that cannot be changed by DHCPv6 or SLAAC protocols.

\* **Link-local Address** is a network address that is valid only for communication within the network segment that the host is connected to. They are not guaranteed to be unique beyond a single network segment. Therefore, routers do not forward packets with Link-local addresses outside a LAN. This type of address consists of two main components. The first part is generated by adding the prefix 0xFE80 to a sequence of 48-bits of 0. The second part is called Interface ID and it is created by splitting the interface's MAC address in half and adding the 16-bit Hex value 0xFFFE between the two halves. An example of Link-local Address is:

*fe80::ca78:abff:fevc:12df*

Link-local addresses can be used in multicast messaging mode, meaning that a packet can be destined to multiple hosts. All hosts interested in that multicast information, need to join that multicast group first. All interfaces which have joined the group receive the multicast packet and process it, while other hosts not interested in that type of packet ignore the information.

\* **Unique Local Addresses** is globally unique, but it should be used in local communication, limiting their scope to an organization's boundary.

### 5.1.3 Neighbor Discovery

The Neighbor Discovery Protocol [15] (NDP) is used with the IPv6 protocol and it operates in the Data Link Layer of [OSI model](#). It is responsible for address autoconfiguration, discovery of other nodes on the link, duplicate address detection, finding available routers and maintaining reachability information. The protocol defines five different ICMPv6 packet types to perform functions for IPv6 similar to the Address Resolution Protocol (ARP), such as Router Discovery as well as Router Redirection:

- **ROUTER SOLICITATION**

Hosts inquire with Router Solicitation messages to locate router on an attached link. Routers which receive the solicited packets will generate Router Advertisements immediately upon receipt of this message rather than at their next scheduled time.

- **ROUTER ADVERTISEMENT**

Routers advertise their presence together with various link parameters either periodically, or in response to a Router Solicitation message.

- **NEIGHBOR SOLICITATION**

Neighbor solicitations are used by nodes to determine the link layer address of a neighbor, or to verify that a neighbor is still reachable via a cached link layer address.

- **NEIGHBOR ADVERTISEMENT**

Neighbor advertisements are used by nodes to respond to a Neighbor Solicitation message.

- **REDIRECT**

Router may inform hosts of a better first hop router for a destination.

IPv6 hosts can configure themselves automatically (SLAAC) when connected to an IPv6 network following the Neighbor Discovery Protocol, by using Router Discovery messages. When first connected to a network, a host has to send a Router Solicitation multicast request for its configuration parameters. The message is received by all nodes but only routers are the ones to process it. When the default gateway receives the Solicitation message, it has to respond immediately with a Router Advertisement informing the sender about its Link-local address, MAC address as well as Internet Layer configuration parameters. Routers present a special case of requirements for



address configuration, as they often are sources of autoconfiguration information, such as router and prefix advertisements. This leads to the host configuring its own IPv6 addresses, as well as discovering the IPv6 and MAC address of the gateway.

Similar to default gateway address resolution, the MAC address of a local host can be discovered by sending Neighbor Solicitation messages requiring the MAC address of the receiver, for the given IPv6 address. The response is sent in the form of a Neighbor Advertisement message, containing all the appropriate information about the inquired host.

#### **5.1.4 Neighbor Cache States**

Every pair of IPv6/MAC addresses, originated from a neighbor, is saved locally as a new entry in the neighbor discovery cache. When communication between two nodes is required, hosts will inquire the appropriate origin of the table, about their IPv6/MAC pairs, in order to maintain an up to date discovery cache. The reachability of a neighbor node is determined by monitoring the state of the neighboring node's entry in the cache. [RFC 2461](#) defines the following states:

##### **INCOMPLETE**

IPv6 address resolution, which is using a solicited-node multicast Neighbor Solicitation message, is in progress. The INCOMPLETE state is entered when a neighbor cache entry is created but does not yet have the node's corresponding Link-layer address.

##### **REACHABLE**

Reachability has been confirmed by receipt of a solicited unicast Neighbor Advertisement message. The neighbor cache entry stays in the REACHABLE state until the number of milliseconds indicated in the Reachable Time field in the Router Advertisement message elapses.

##### **STALE**

Reachable time (the duration since the last reachability confirmation was received) has elapsed. The neighbor cache entry enters the STALE state after the number of milliseconds in the Reachable Time field in the Router Advertisement message (or a host default value) elapses, and the entry remains in this state until a packet is sent to the neighbor. The entry also enters the STALE state when the host receives an unsolicited Neighbor Advertisement message that is advertising the Link-layer address.

##### **DELAY**

To allow time for upper layer protocols to provide reachability confirmation before sending Neighbor Solicitation messages, the neighbor cache entry enters the DELAY state and waits a configurable period of time after sending a packet. If reachability is not confirmed by the delay time, then the entry enters the PROBE state, and a unicast Neighbor Solicitation message is sent.

##### **PROBE**

Reachability confirmation is in progress for a neighbor cache entry that was in the STALE or DELAY state. Unicast Neighbor Solicitation messages are sent at intervals corresponding to a retransmission timer field in the Router Advertisement message that

this host received. A configurable variable determines the number of Neighbor Solicitation messages sent before the reachability detection process is abandoned and the neighbor cache entry is removed.

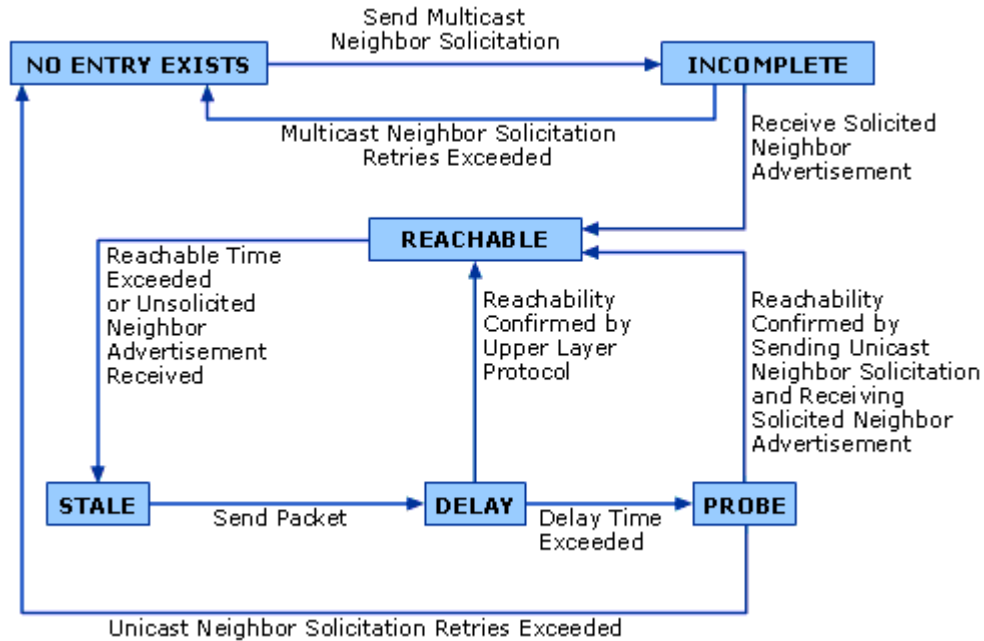


Figure 5.4: Cache state algorithm

## 5.2 Ipv6 Patch Architecture

This thesis presents important network improvements that make the Rupture attack more practical by implementing two new patches: one in the BetterCap framework and one in the underlying ruby PacketFu library. With these two patches IPv6 attacks are now possible in the Rupture. The patches have successfully been merged in upstream BetterCap, allowing BetterCap to perform IPv6 NDP attacks from the CLI, a contribution which is of independent interest as this class of attacks had not been automated previously.

Below are described the key aspects of this patch's architecture.

### 5.2.1 IPv6 Parser

The first part of the attack involves user input parsing. One of the most important changes in the BetterCap patch is to enable the gem to manage IPv6 addresses. IPv6 protocol uses a different address scheme, making address manipulation challenging. As a result of that, the extension needs to validate IPv6 addresses and exclude those with the wrong format. This gap is filled by the IPv6 validator with the appropriate regex matching patterns.

### 5.2.2 Neighbor IPv6/MAC Address discovery

The IPv6 extension on BetterCap has a specific algorithm for discovering matches between IP and MAC addresses (including the gateway's MAC in case of an error at the lookup). The NDP discovery agent is responsible for activating this procedure. First, it performs a lookup into the neighbor discovery cache and if no entry is found matching the specific address, it then sends a Neighbor Solicitation message on the wire, seeking for a proper response. The message consists of a multicast IPV6 address. This address

is created by the prefix ff02::1:ff plus the 24 least significant bits of the destination address. An example is:

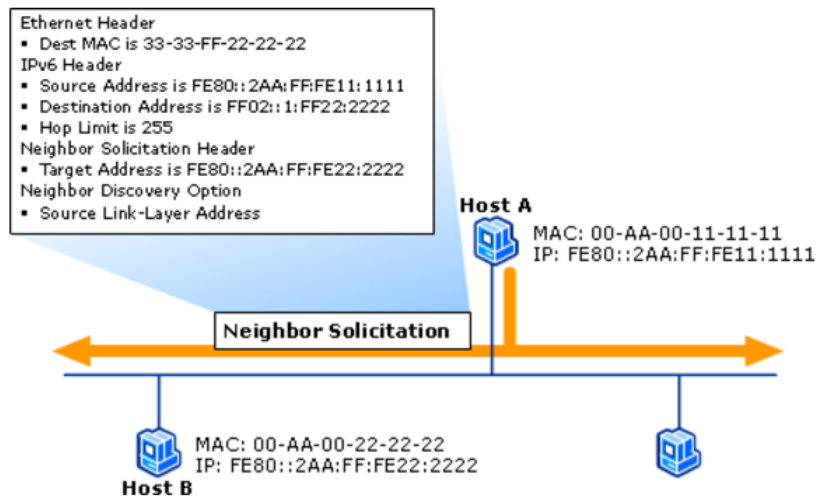


Figure 5.5: Multicast Neighbor Solicitation Message for address resolution.

When the targeted host receives the message, it responds to the sender so he can update his cache entry with the valid data:

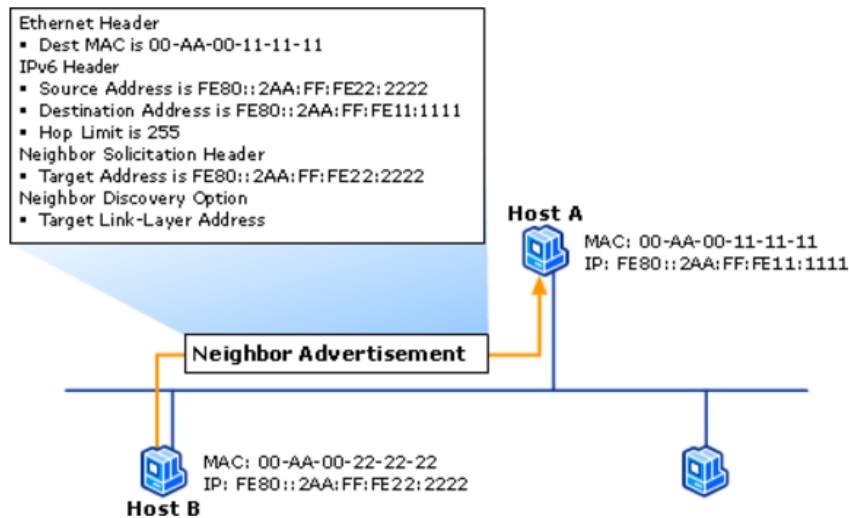


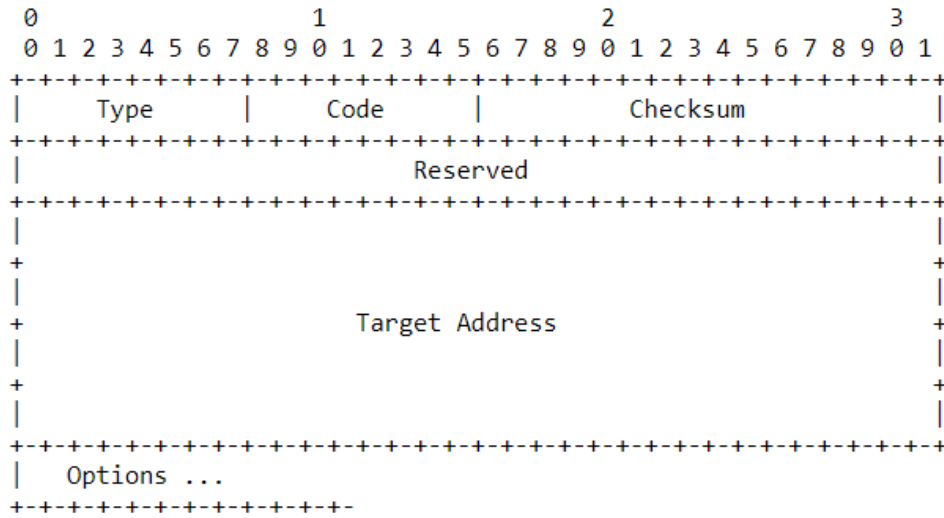
Figure 5.6: Unicast Neighbor Advertisement Message for address resolution.

### 5.2.3 IPv6 Packet Manipulation ( PacketFu extension patch)

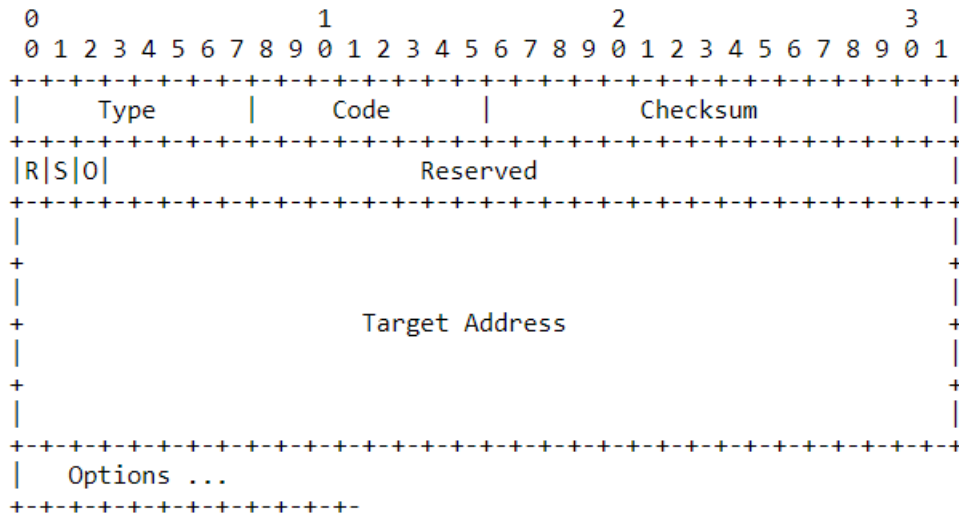
The current Ruby implementations on low level packet crafting (with PacketFu library being the most famous among them and the one BetterCap is using) do not provide the ability to create Neighbor Discovery packets. PacketFu's ICMPv6 old packet formats only provide error reporting and network diagnostics. However NDP extended those capabilities on [RFC 4861](#) by adding multiple header fields serving the protocol's purposes. So in order to extend BetterCap capabilities, we created a [PacketFu patch](#) supporting NDP packet manipulation.



This patch in particular, as an underlying library, can be used to develop new manipulation tools and techniques around Neighbor Discovery, giving the users the ability to create their own Neighbor Solicitation Packets, as well as Neighbor Advertisement Packets:



**Figure 5.7: Neighbor Solicitation message format.**



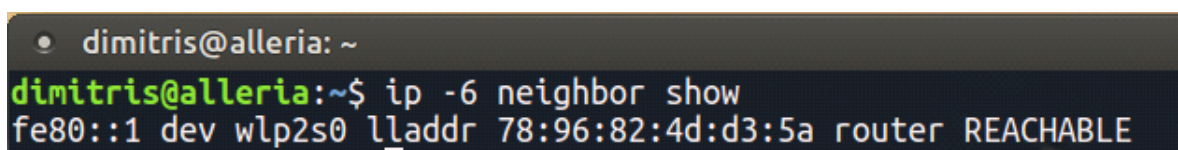
**Figure 5.8: Neighbor Advertisement message format.**

The three first fields of the above Packets are *Type*, *Code* and *Checksum*. As it states, type field determines the type of the ICMPv6 Packet. Its value determines the format of the remaining data. Number 135 refers to a Neighbor Solicitation packet, whereas 136 is for a Neighbor Advertisement. Code field should always be 0. The checksum field is used to detect data corruption in the ICMPv6 message and parts of the IPv6 header.

The PackteFu extension required the development of the rest of the fields that are introduced with Neighbor Discovery Protocol. Fields like *Reserved*, *Target Address*, *Options*, and most importantly R-S-O bits (which are used for NDP spoofing) are part of this thesis patch extension.

## 5.2.4 Neighbor Discovery Spoofer

The NDP spoofing algorithm is the most important part in IPv6 MitM attack. The adversary sends Neighbor Advertisement messages to the victim with a certain frequency, informing him that he is the default gateway. This is achieved by crafting a Neighbor Advertisement packet with gateway's Link-local address as the source, paired with the adversary's MAC address. In addition to this, Neighbor Advertisement packet contains three flag bits R,S,O. R-bit indicates that the sender is a router, S-bit indicates that the advertisement was sent in response to a Neighbor Solicitation from Destination address and O-bit indicates that the advertisement should override an existing cache entry. By enabling R/O-bits on the Advertisement message the victim updates his cache entry representing the gateway, with the adversary's MAC address. Since the S-bit is also enabled the victim doesn't send a revalidation Neighbor Solicitation message to verify the IPv6/MAC pair and marks its entry as REACHABLE. This results into the victim forwarding all of his IPv6 traffic to the adversary.

A terminal window with a dark background. The prompt is 'dimitris@alleria: ~'. The command 'ip -6 neighbor show' has been executed, resulting in the output 'fe80::1 dev wlp2s0 lladdr 78:96:82:4d:d3:5a router REACHABLE'.

```
dimitris@alleria: ~  
dimitris@alleria:~$ ip -6 neighbor show  
fe80::1 dev wlp2s0 lladdr 78:96:82:4d:d3:5a router REACHABLE
```

Figure 5.9: Spoofed neighbor cache entry.

## 5.2.5 ip6tables Firewall

As an adversary, being in the middle of a host and a router requires a proper grouping of the victim's traffic in order to inject the desired javascript code. The HTTP responses of the majority of the websites are directed to the receiver's port number 80. By using the [ip6tables library](#) we create a firewall redirecting the incoming packets with the victim's address source and port number 80, to a locally deployed proxy. The proxy's core role is to collect every packet sent by ip6tables rules and create readable HTML source code out of the HTTP responses. Then the proxy injects the javascript code at the top of the tag of the HTML page. As a result of that, the HTTP proxy sends an HTML response with injected javascript code to the victim, ready to be executed by the victim's browser.

## 6. CONCLUSION

The last few years we have witnessed a phenomenal growth in the security industry. The ever growing demands for safer communications over the Internet triggered researchers to come up with better and more secure practices. However, as this thesis states, while the technological industry develops new ways of improvement, the attackers can take advantage of this technology too.

Compression side-channel attacks, sophisticated as they are, can take place in everyday scenarios. It is important that developers should defend against this type of attacks, since various tools that have already been implemented for that purpose[16].

Also users should not rest by the fact that ARP is not used in IPv6 protocol. As already shown, IPv6 is also susceptible in MitM attacks. They should always follow the good practices, as concerns the MitM attacks, such as always check the domain name and provide sensitive data only on *https* connections.

Researchers should continue to develop tools that protect the community from every possible web attack, but also try to share their knowledge with everyone. This way the Security culture will be strengthen, lowering the risk of cyber attacks.

## REFERENCES

- [1] (online) url: <https://www.bettercap.org>.
- [2] (online) url: <http://www.secdev.org/projects/scapy/>.
- [3] D.Zindros D. Karakostas. Practical New Developments on BREACH, April 2016.
- [4] David A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IEEE, 40:1098–1101, September 1952.
- [5] Dimitris Karakostas. Probabilistic attacks against compressed encrypted protocols, January 2016.
- [6] Bodo Moller, Thai Duong, Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback, September 2014.
- [7] Jacob Ziv, Abraham Lempel. A universal algorithm for sequential data compression. Information Theory, IEEE Transactions, 23:337–343, May 1977.
- [8] (online) URL: [https://en.wikipedia.org/wiki/ARP\\_spoofing](https://en.wikipedia.org/wiki/ARP_spoofing).
- [9] (online) URL: <https://en.wikipedia.org/wiki/RC4>.
- [10] (online) URL: [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)).
- [11] David C. Plummer. An Ethernet Address Resolution Protocol, November 1982.
- [12] Andrei Popov. Prohibiting RC4 Cipher Suites, February 2015.
- [13] Yaron Sheffer Porticor, Ralph Holz, Peter Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS), February 2015.
- [14] (online) url: <https://github.com/packetfu/packetfu>
- [15] (online) url: [https://en.wikipedia.org/wiki/Neighbor\\_Discovery\\_Protocol](https://en.wikipedia.org/wiki/Neighbor_Discovery_Protocol)
- [16] (online) url: <https://ctxdefense.com/>
- [17] (online) url: [https://en.wikipedia.org/wiki/Huffman\\_coding](https://en.wikipedia.org/wiki/Huffman_coding)