BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DEMONSTRATION AND ANALYSIS OF ATTACKS ON PROTOCOL HTTPS

DEMONSTRACE A ANALÝZA ÚTOKŮ NA PROTOKOL HTTPS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	**ADAM MURGAŠ**
SUPERVISOR VEDOUCÍ PRÁCE	**doc. Ing. ONDŘEJ RYŠAVÝ, Ph.D.**

BRNO 2022

Zadání bakalářské práce

148217

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Murgaš Adam**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Demonstrace a analýza útoků na protokol HTTPS**
Kategorie:	Bezpečnost
Akademický rok:	2022/23

Zadání:

1. Seznamte se s protokoly HTTP/1.1, SPDY a HTTP/2.
2. Nastudujte principy útoků BEAST a CRIME.
3. Vytvořte si vhodné virtuální prostředí pro další experimenty.
4. Demonstrujte útoky pro několik scénářů a zaznamenejte komunikaci z těchto útoků ve formě anotovaných datových sad.
5. Analyzujte komunikaci z útoků a navrhněte jednoduchou metodu pro jejich detekci.
6. Zhodnoťte provedené experimenty a diskutujte možné způsoby obrany proti těmto útokům.

Literatura:

- O. Ivanov, V. Ruzhentsev & R. Oliynykov, "Comparison of Modern Network Attacks on TLS Protocol," *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)*, 2018, pp. 565-570
- Duong & J. Rizzo, "Here come the Ninjas", *Ekoparty*, 2011.
- T. Duong & J. Rizzo, "The CRIME attack", *Ekoparty*, 2012.
- Y. Gluck, N. Harris & Angelo Prado, "BREACH: Reviving The CRIME Attack" in Blackhat, USA, 2013.

Při obhajobě semestrální části projektu je požadováno:
Minimálně body 1-3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Ryšavý Ondřej, doc. Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	28.10.2022

Abstract

The objective of this report is to analyze two attacks against the HTTPS protocol, namely the BEAST and CRIME attacks. The main point is to see whether or not they are still possible with the technologies of today, as well as demonstrate how they work and how to prevent or detect similar attacks in the future. This report describes the theoretical foundation behind these attacks and addresses possible solutions for detection or prevention. Following the theoretical foundation and the prevention and detection methods, this report also provides a demonstration of the principles behind these attacks as well as a dataset focused on certain metrics regarding the attacks, in order for readers to gain better understanding of their principles, as similar attacks might be discovered in the future.

Abstrakt

Cieľom tejto práce je analyzovať dva útoky na protokol HTTPS, najmä útoky BEAST a CRIME. Hlavnou pointou práce je zistiť, či sú tieto útoky stále možné s dnešnými technológiami, a zároveň demonštrovať ako tieto útoky funguju a ako sa podobným útokom vyhnúť alebo ako ich včas detekovať. Táto práca opisuje teoretický základ týchto útokov a taktiež opisuje možné riešenia pre detekciu a prevenciu. Po teoretickom základe a spôsoboch prevencie, táto práca taktiež poskytuje demonštráciu princípov týchto útokov a taktiež dataset, ktorý sa zameriava na určité metriky útoku, aby mali čitatelia lepšiu znalosť o princípoch za týmito útokmi, pretože podobné útoky by mohli byť objavené v budúcnosti.

Keywords

`HTTPS`, `SPDY`, `BEAST`, `CRIME`, `TLS/SSL`, **C**ipher **B**lock **C**haining (`CBC`), **I**nitialization **V**ector (`IV`), cipher suites, block ciphers, `TLS` compression, Python

Klíčová slova

`HTTPS`, `SPDY`, `BEAST`, `CRIME`, `TLS/SSL`, **C**ipher **B**lock **C**haining (`CBC`), **I**nicializační **V**ektor (`IV`), soubor šifer, blokové šifry, `TLS` komprese, Python

Reference

MURGAŠ, Adam. *Demonstration and analysis of attacks on protocol HTTPS.* Brno, 2022. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Ondřej Ryšavý, Ph.D.

Demonstration and analysis of attacks on protocol HTTPS

Declaration

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána doc. Ing. Ondřeja Ryšavého Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

. . . .
Adam Murgaš
May 11, 2023

Contents

Chapter 1

Introduction

Nowadays, as people spend more and more time on the internet, more and more day to day activities are also shifting into the virtual online world. People do all sorts of activities online, from chatting with friends or ordering food, to playing video games or even managing bank accounts. As it happens, some of those activities are bound to be private or confidential and therefore need to be secured. Researchers and developers are constantly putting in massive amounts of effort into improving existing security mechanisms and inventing new ones, in order to keep all of our online activities safe. Despite their best efforts however, possibilities of new clever exploits and attacks are constantly being discovered and exploited, which fortunately also serves to improve online security. Because of this, it makes more sense than ever for everyone involved in the online world, to be aware of how to behave on the internet, so that they keep their personal details and activities as safe as possible.

A vast majority of the internet in today's age uses the HTTPS protocol, in order to encrypt online communication and thus keeps it safe and private. However even this very secure protocol is not completely safe and throughout the years many vulnerabilities have been discovered. The focus of this thesis is to describe two kinds of attacks on the HTTPS protocol, which are the BEAST attack and the CRIME attack. As described in the later chapters, these attacks are no longer possible with modern technologies. However, they still provide a great example of a handful of principles on how attackers might be able to exploit certain vulnerabilities and break encryption mechanisms. The goal of this thesis is to provide a theoretical basis and the preconditions of these attacks, as well as a demonstration of their principles and methods of how we can possibly prevent or detect similar attacks in the future. With the understanding of these topics, attacks similar to the BEAST and CRIME attacks, which might be discovered in the future, will be easier to deal with.

Chapter 2

BEAST attack

The **B**rowser **E**xploit **A**gainst **S**SL/**T**LS (BEAST) attack is an attack on the HTTPS protocol, which aims to exploit a vulnerability in TLS version 1.0 or any older SSL protocol[12]. Specifically, it exploits a vulnerability in the **C**ipher **B**lock **C**haining (CBC) encryption mode of TLS. If an attacker can exploit this vulnerability successfully, they will have the ability to decrypt HTTPS secured communication between a client and a server without ever needing to obtain the decryption key and thus being able to perform session hijacking. The BEAST attack as a whole, is a combination of multiple kinds of attacks and techniques, such as record splitting, a chosen boundary attack and a **M**an-**i**n-**t**he-**M**iddle (MitM) attack.

The origins of this attack date back to 2002, when Phillip Rogaway, a professor of computer science and cryptography at the University of California, highlighted a predictability in the cipher block chaining mode of TLS. Later, in 2011, two security researchers Juliano Rizzo and Thai Duong have further exploited this vulnerability and formed the BEAST attack, as it is known today.

In order for the BEAST attack to be possible, several preconditions need to be met. These preconditions, in combination with the fact that the attack can only read very short pieces of information in limited time as well as several security countermeasures having been developed since its discovery, make the BEAST attack very impractical and therefore also very unlikely. However, even though the BEAST attack is no longer considered very effective, it still displays how it is possible to combine multiple principles and exploits to form an effective attack. Because of its effectiveness at the time of discovery, the attack was considered threatening enough for the vulnerability in cipher block chaining to be fixed in version 1.1 of TLS and the following versions. Most modern browsers and servers use TLS version 1.1 or higher and launching a BEAST attack is only possible, if they are using TLS 1.0 or an older SSL protocol. Another option would be to use the BEAST attack in combination with a different attack, which forces a server to revert to older versions of TLS[16].

2.1 Cipher block chaining vulnerability

The entire BEAST attack is based on a vulnerability in cipher block chaining. It is therefore important to understand how cipher block chaining works and how this vulnerability becomes relevant, so that it can be exploited efficiently in the BEAST attack.

When using TLS during internet communication, the browser and the server will first use an asymmetrical encryption mechanism during the negotiation phase of the communication. During this phase the client verifies the server's identity using its SSL certificate authority's digital signature. After that, the client and the server will negotiate several encryption details, which will be used during the communication. This is called a `TLS handshake` [10]. After the negotiation process, the communication between the client and the server will be encrypted symmetrically, using the encryption key which was previously negotiated. Symmetrical encryption means that both participants in the communication will use the same encryption key to encrypt and decrypt messages.

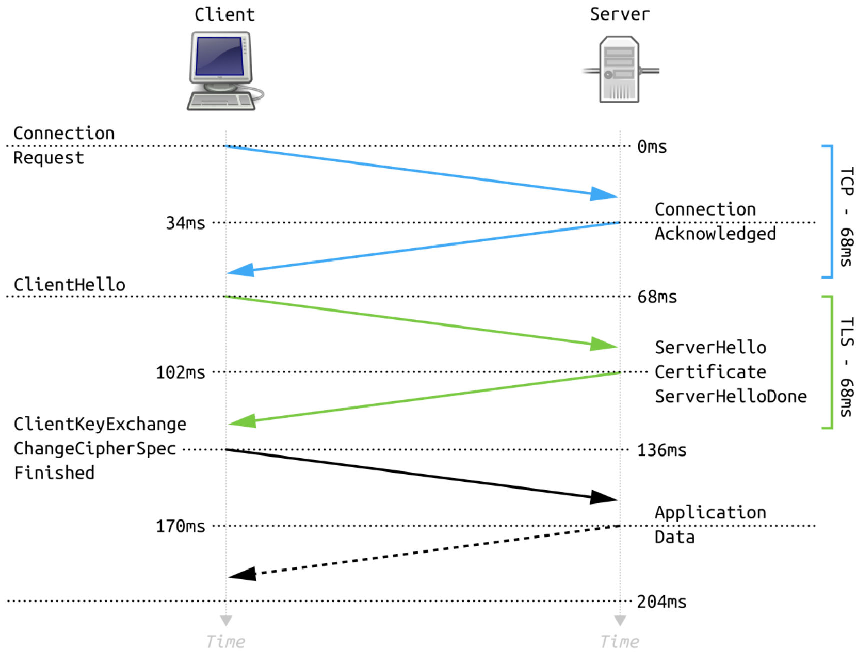


Figure 2.1: TLS handshake[2].

TLS uses block ciphers during encryption of the communication. This means that the data gets divided into blocks, which have a fixed length. Each block of data then gets encrypted separately, before it is sent[3]. However, when encrypting identical blocks of data with the same encryption key, the resulting encrypted ciphertexts will also be identical, which creates possible vulnerabilities. In order to counter this, TLS uses what is called `initialization vectors` (IVs). Initialization vectors are arbitrary numbers, which are of the same length, as the data blocks. They serve to prevent identical data blocks being encrypted into identical ciphertexts. Before encrypting a data block, TLS first performs a logical `XOR` operation between the data block and the initialization vector. Only the result of this logical operation then gets encrypted with the negotiated encryption key, which finally gives us the final ciphertext. Because of the data blocks getting logically `XOR`ed with an arbitrary initialization vector, even if the raw data blocks are identical, the resulting ciphertext will be different.

Instead of always using a random initialization vectors to encrypt separate blocks of data, older versions of TLS (namely version 1.0 or any older SSL protocol) will use the resulting ciphertext of the previous data block as the initialization vector for the encryption of the subsequent data block. This is called `Cipher Block Chaining`[15]. Here is exactly where the vulnerability, which makes the BEAST attack possible, exists. If an attacker has the ability to monitor the HTTPS communication between the server and the client, they has access to the encrypted ciphertexts. They may not be able to read the raw data, which is encrypted in these ciphertexts, but if cipher block chaining is in use, they can abuse the fact that the ciphertexts they have access to, will be used as initialization vectors for the following data blocks. Exactly how this can be abused will be described in the following sections.

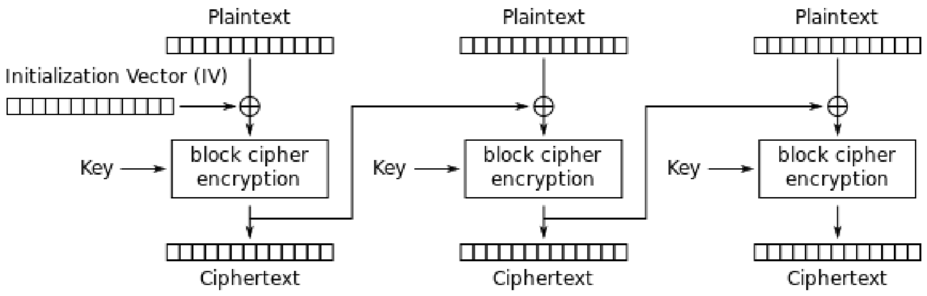


Figure 2.2: Cipher block chaining[4].

This specific vulnerability was highlighted by Phillip Rogaway in 2002. Later in 2011 he also published a very comprehensive evaluation of different block cipher modes[14]. As mentioned before, Phillip Rogaway uncovered this vulnerability in the cipher block chaining mode of TLS by highlighting the predictability of the initialization vector used for every subsequent message after the first one. Assuming the attacker has access to the encrypted ciphertexts, they would then be able to know, that a given ciphertext would be used for the following message. Using this information, the attacker can then attempt to brute-force (guess) the contents of the plaintext they want to decrypt. However, at the time it seemed like it was only possible to decrypt the encrypted data, by correctly guessing the entire block of plaintext. The usual sizes of cipher blocks are 8 bytes (64 bits), 16 bytes (128 bits) or 32 bytes (256 bits). Correctly guessing the entire block of any of these sizes is astronomically unlikely and practically impossible. Because of that, the attack was only considered to be a theoretical threat.

In the year 2011, two security researchers Juliano Rizzo and Thai Duong discovered a new approach to the process of guessing the encrypted data[12]. They discovered, that by performing a `chosen boundary` attack, it is possible to isolate just one single byte of the plaintext, which the attacker wants to decrypt. After the attacker guesses this isolated byte correctly, they can then shift the cipher block boundaries to isolate the next unknown byte and then repeat the process. This way, the attacker can guess one single byte at a time, instead of having to guess an entire block of 8, 16 or 32 bytes at once, which is significantly more manageable and makes this attack a lot more feasible. While the BEAST attack was always considered an unlikely, theoretical one, Duong and Rizzo's discovery prompted many servers and browsers to upgrade to newer versions of TLS, which provide invulnerability against the BEAST attack.

2.2 Preconditions

As mentioned before, the BEAST attack has a few preconditions, which need to be met in order for an attacker to be able to launch it[12].

These preconditions are:

- TLS version 1.0 or an older SSL protocol must be used for encrypting the communication, or the attacker must have a way to enforce this by performing a downgrade attack[13] - this is because BEAST aims to exploit the predictability of the initialization vectors used in cipher block chaining, which is fixed in newer versions of TLS.
- The attacker must be able to monitor the ongoing encrypted communication between the browser and the server - this is because the attack utilizes the encrypted ciphertexts, by comparing them with the ciphertexts generated with the guessed plaintext.
- The attacker must be able to inject plaintext data blocks into the communication, to observe the generated ciphertext output - this used to be possible by performing a **M**an-**i**n-**t**he-**M**iddle (MitM) attack[6], a JavaScript injection, or other methods.

2.3 How BEAST works

Now that we know what exactly the BEAST attack is trying to exploit and we know the preconditions that we need to meet before launching this attack, let's look at how it works in more detail.

Let's assume that a client, who will be the victim of the BEAST attack, has logged into a website using their private credentials and that they are communicating with the server using TLS 1.0 (or any other older SSL protocol with cipher block chaining). Part of this communication will inevitably be some sensitive data, such as a password, a session ID, or anything of this sort. The attacker's objective is going to be to decrypt this sensitive data, without using the encryption key.

2.3.1 Initialization vector canceling

When using TLS for encrypting communication, the raw data of the communication gets divided into blocks of a fixed length. All of these blocks are then individually logically `XOR`ed with the current initialization vector (IV). The result of this logical operation then gets encrypted with the negotiated encryption key, which results in a ciphertext. This ciphertext is then sent, but also stored, so that it can be used as the initialization vector for the next plaintext block. For reference, see figure 2.2.

Suppose that there is a block of plaintext `X`, which contains information, that we (the attacker) want to obtain. This block `X` gets logically `XOR`ed with an initialization vector, which will be the ciphertext of the previous message `Y` and then encrypted with the encryption key, resulting in ciphertext `Z`. Suppose also that at a later point in the communication, there is a block of plaintext `A`, which we, as the attacker, have control over. This block `A` gets logically `XOR`ed with an initializaiton vector, which will be the ciphertext of the previous message `B` and then also gets encrypted with the same encryption key, resulting in ciphertext `C`.

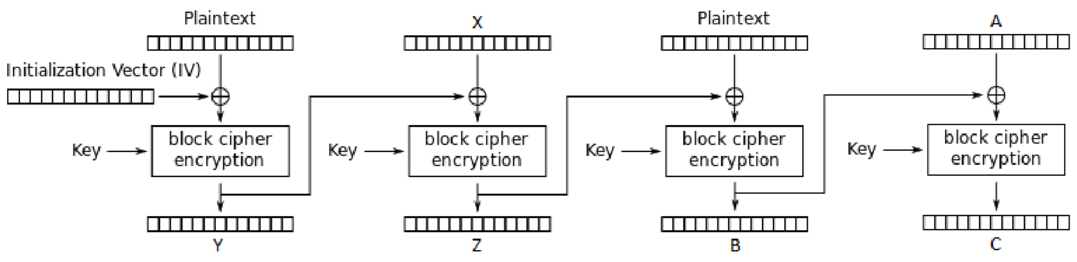


Figure 2.3: Scenario visualization.

We can set the value of plaintext `A` as such:

$$A = B$$

What this essentially means, is that we will be using the ciphertext of the previous message as the value of the plaintext, that we are in control of. This value then gets logically `XOR`ed with the previous message's ciphertext, which has the same value. Logically `XOR`ing two blocks with identifal values results in a block of zeros. We have therefore effectively canceled out the initialization vector for the message that we are in control of.

We can then further abuse this by setting the value of plaintext `A` to:

$$A = B \oplus Y \oplus X'$$

Here we are `XOR`ing three values and setting the result as the value of our block of plaintext `A`. The three values here are:

- `B` - the ciphertext of the previous message
- `Y` - the ciphertext which was used to encrypt the block of plaintext we want to decrypt
- `X'` - our guess of the contents of plaintext `X`, which we want to decrypt. The goal is to try to guess `X'` and match it with `X`.

If the block of plaintext `A` is set as such, the value which is going to get encrypted will be $B \oplus B \oplus Y \oplus X'$. The two identical values `B` will cancel each other out, leaving us with $Y \oplus X'$. Encrypting the `XOR` of these two values then results into ciphertext `C` and the goal is to match ciphertexts `C` and `Z`. The important thing to note here, is that we essentially canceled out the initialization vector `B` for our constructed block of plaintext `A` and replaced it with the initialization vector `Y`, which was used for encrypting the plaintext, that we want to decrypt. Therefore both blocks of plaintext `X` and `A` will use the same initialization vector, as well as the same encryption key. This means, that in order to match ciphertexts `C` and `Z`, the only thing we need to do is to match `X` and `X'`.

2.3.2 Guessing the plaintext block

When communicating over the internet using HTTPS, part of the communication are also multiple headers containing various pieces of information, such as the language, browser version on the client side, character encoding or many other pieces of metadata. More importantly, there are some headers, which contain sensitive information, such as cookies, which hold the session ID, passwords, or anything of the sort. All of these headers are arranged in the HTTPS communication in a predetermined manner. Therefore when the client makes the same request to the server multiple times, the headers containing the metadata of the communication will be arranged in the same way, resulting in the same HTTPS request structure each time. That is to say, the contents of the HTTPS packets are predictable and we (the attacker) can tell where exactly the sensitive information is located. Using this knowledge, we can make specially crafted HTTPS requests, in order to manipulate the position of the sensitive information in such a way, so that a data block contains only one unknown byte of the sensitive information. We can then attempt to guess this single byte and once successful, we can manipulate the position of the sensitive information again, to expose the next unknown byte. This is known as the chosen boundary attack[9].

For example, suppose that during an ongoing encrypted communication, a client will try to access a file `index.html`. An HTTPS request will be assembled with the use of the mentioned headers, in a predetermined way. The final HTTP request might look like this:

```
GET /index.html HTTP/1.1
Host: google.com
Cookie: Session=21047948
Accept-Encoding: text/html
Accept-Charset: utf-8
```

In this example, let's also assume that the size of the block cipher will be 8 bytes. At 1 byte per character, the HTTPS request will get divided into blocks of 8 bytes. Since the headers are organized in a predictable way, we can accurately tell the position of the session ID.

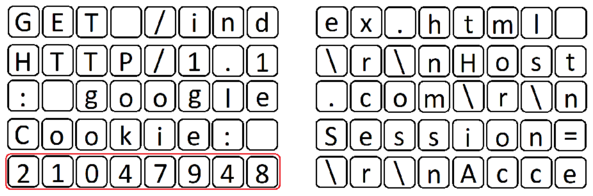


Figure 2.4: Separation of HTTP request into data blocks.

Knowing the current position of the session ID, we can create a specially crafted HTTPS request in such a way, so that the session ID is shifted to a position where only one byte of it is exposed. In our example, this could be done by changing the accessed document from `index.html` to `index.htm` for example. By doing this, we can shorten the data before the session ID by one byte, resulting in one byte of the session ID being shifted into the previous data block. The final HTTPS request would then look as follows:

```
GET /index.htm HTTP/1.1
Host: google.com
Cookie: Session=21047948
Accept-Encoding: text/html
Accept-Charset: utf-8
```

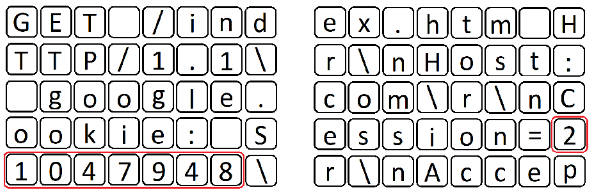


Figure 2.5: Separation of crafted HTTP request into data blocks.

We first make the victim send an HTTPS request for `index.htm` and we let all of the headers get appended automatically, with the knowledge of the session ID's position. Then we make the victim send another request for `index.htm`, but this time we will be appending the headers manually. After doing this, we can observe the two encrypted ciphertexts of these HTTPS requests, while trying possible options for the given byte of the session ID that we are trying to guess. If the ciphertext of the request with automatically added headers matches the ciphertext of the request with our guessed byte, we have guessed the byte successfully. We can then move on to the next byte by shifting the session ID position again, in this case by making the next series of requests for `index.ht` and we repeat this process until the entire session ID is decrypted.

2.4 Prevention against BEAST

The BEAST attack has prompted many web browser developers and server administrators to try to mitigate the possibility of the attack. The simplest solution by far is to enforce the use of newer versions of TLS, such as version 1.1 or higher, because these newer versions address the underlying cipher block chaining vulnerability. However, using newer versions of TLS is not always possible. Because of this, other methods and workarounds have been explored.[9].

At first, it was recommended to switch to a stream cipher, as opposed to a block cipher. The vulnerability was only present in block ciphers, but older versions of TLS also supported the `RC4` stream cipher. However, it was later discovered, that the `RC4` stream cipher was theoretically unsafe and as more and more flaws have been highlighted, the use of the `RC4` stream cipher has eventually been prohibited. Therefore, other methods of mitigating this attack had to have been implemented.

2.4.1 Randomized padding

Randomized padding is a technique in cryptography, used to prevent attacks which exploit messages having a known structure or length. The purpose of randomized padding is to add extra randomized data into a message before it gets encrypted. This ensures that encrypting the same data multiple times will produce unique ciphertexts. Thanks to that, the BEAST attack cannot rely on the predictability of the structure of requests and it will become considerably harder for the attacker to analyze encrypted data, identify patterns and extract sensitive information.

2.4.2 Use of packet pattern recognition

The BEAST attack involves the victim sending a large number of requests in a row, while only slightly altering the contents of the message. Knowing about this pattern, we can then prevent the BEAST attack with the use of:

- rate-limiting techniques, with which we can limit the amount of requests from clients to the server. By doing this, we can considerably lengthen the time required for such an attack, or make it completely unviable.
- **W**eb **A**pplication **F**irewalls (WAF). These firewalls are placed between the server and the clients and their purpose is to monitor traffic and compare it to its rules and policies. If the traffic matches a pattern of known attacks, it can be blocked, or an alert can be triggered, for server administrators to take action.[7]

2.4.3 0/n split

Some web browser developers and server administrators have implemented a so called `0/n split`, in order to mitigate the BEAST attack. The idea behind this is to first send an empty data block with a payload length of `0` before sending actual data blocks with the size of `n` (hence the name `0/n split`). Blocks, which are not fully filled with data, get padded with randomly generated data. Therefore sending an empty data block results in a data block full of randomly generated padding. This block then gets encrypted and used as the initialization vector for the first data block of the actual message, which restores randomness of the encryption. However, as this was only a quick work around, it was not officially documented in the TLS documentation, which caused it to create many compatibility issues. Some web browsers also do not support sending empty messages, which caused further issues with the `0/n split`.

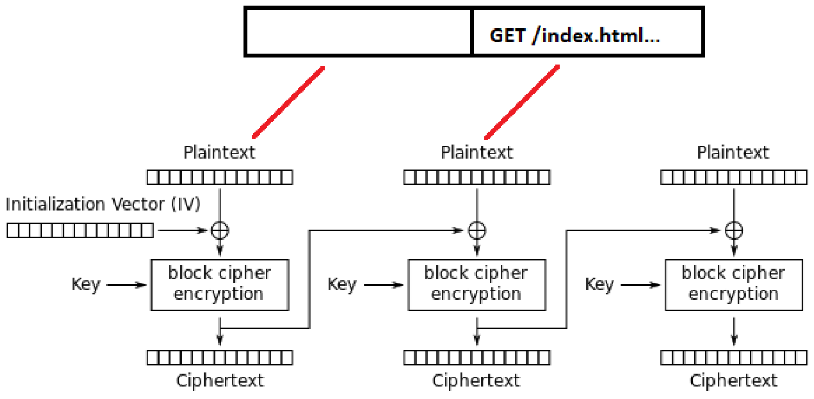


Figure 2.6: 0/n split.

2.4.4 1/n-1 split

An upgraded version of the `0/n split` which solves the empty message incompatibility is the `1/n-1 split`. In this version of the solution, instead of sending an empty message before the actual data, the first byte of the actual data is used. The first message will contain the first byte of the data and the rest of the message gets padded with randomly generated data. The result then gets encrypted and used as the initialization vector for the following block of data. This solution also restores the randomness of the encryption, however since it doesn't use empty data blocks, there are no compatibility issues, compared to the `0/n split`.

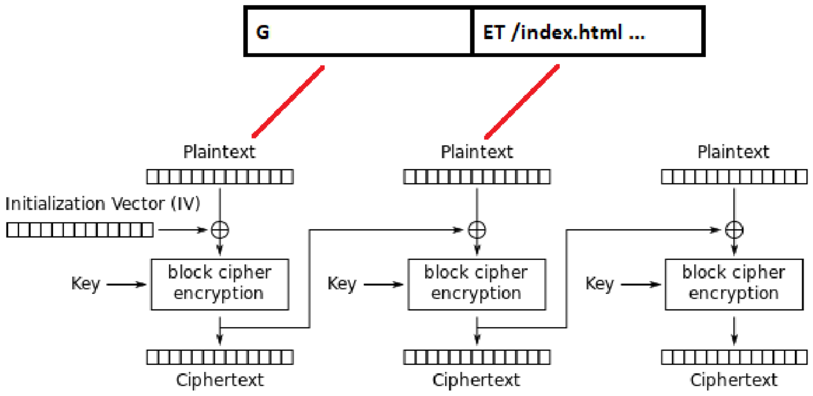


Figure 2.7: 1/n-1 split.

Chapter 3

CRIME attack

The **C**ompression **R**atio **I**nfo-leak **M**ade **E**asy (CRIME) attack, is an attack which is very similar to the BEAST attack in many aspects. Same as for BEAST, the CRIME attack aims to exploit a vulnerability in protocols used for encrypted communication over the internet and its objective is also to decrypt sensitive parts of this communication, without the use of the negotiated encryption key. However, instead of the predictability in the cipher block chaining mode of the older versions of TLS, the CRIME attack exploits a vulnerability, which is caused by compression. This vulnerability is present both in HTTP and in SPDY, which is Google's HTTP-like protocol[11]. CRIME also requires the attacker to perform a **M**an-**i**n-**t**he-**M**iddle (MitM) attack, in order to force the victim to make cross-site requests, similarly to BEAST.

The compression vulnerability was first highlighted in the SPDY protocol in 2011 by Adam Langley, Google's software engineer. He described the possibility of being able to deduce contents of encrypted SPDY packets, based on observing their length after compression. This concept was also demonstrated in the form of the CRIME attack by two security researchers Juliano Rizzo and Thai Duong, during the Ekoparty security conference in Argentina in 2012[5].

The attack itself is a combination of a chosen plaintext attack and unintentional information leakage through data compression. Similarly to BEAST, the CRIME attack also has a handful of preconditions, which need to be met in order for an attacker to be able to execute it. It is also only able to decrypt short strings of sensitive information in limited time, while using a relatively large amount of HTTP requests. This makes the CRIME attack fairly impractical and very unlikely to happen, however with a wide range of websites having been prone to these kinds of attacks at the time, developers and researchers have since looked for solutions, or even stopped HTTP header compression altogether. This in turn made the CRIME attack a lot less likely to happen today.

3.1 Information leakage by compression

As mentioned before, the basis of the CRIME attack lies in a vulnerability caused by data compression. When compressing data in an encrypted communication, the compression might leave behind clues, which can help deduce the encrypted content. It is therefore important to understand what exactly the vulnerability is and how we can exploit the clues, in order to figure out sensitive data. During encrypted communication over the internet, the server and the client use multiple metadata headers. As these headers become more numerous, they also make up more data and therefore more bandwidth on the network. In order to make the communication more efficient, compression was implemented into the HTTP and SPDY protocols (as well as others). Compression was introduced into these protocols in the form of compression modes, which could be disabled. For HTTP, the `TLS DEFLATE` compression scheme in particular was found to be vulnerable to CRIME[11].

When using compression during communication over the internet, the compression will locate duplicate occurrences of strings and it will replace them with smaller tokens, which point to their first instances, in order to get rid of redundant data. However, if an attacker can monitor the ongoing communication, as well as inject data into it, they can alter the client's requests, with the goal of trying to insert duplicate data into them. If the data injected into the request headers by the attacker is common with the original data, it will get compressed. The attacker will therefore be able to observe a decrease in the encrypted packet's size. Exactly how this can be done, will be explained in the following sections.

3.2 Preconditions

In order for an attacker to be able to execute the CRIME attack, the following preconditions need to be met:

- The attacker must be able to monitor the ongoing encrypted communication between the browser and the server - this is because the attack will be observing the changes in length of the HTTP packets.
- The attacker must be able to inject data into the clients HTTPS requests - this used to be possible by performing a **M**an-**i**n-**t**he-**M**iddle (MitM) attack[6], a JavaScript injection, or other methods.
- Both the client's browser and the server must support the SDPY protocol, or any version of TLS with compression enabled - this is because the attacker will be abusing the compression mechanism by injecting redundant data into the HTTP headers.

3.3 How CRIME works

With the theoretical basis for the CRIME attack explained and assuming all of the preconditions have been met, let's look into the principles of CRIME, so that we understand how it works in more detail.

Suppose the following scenario. A client, who will be the victim of the CRIME attack, has logged into the website `example.com` using their private credentials. Both the server and the client's browser have to either be using the SPDY protocol, or TLS with header compression enabled. In order for the server to be able to identify the client, it has stored the session ID `21047948` into a cookie in the client's browser. This session ID will be used in any subsequent requests for `example.com` from our victim. Similarly to the BEAST attack, our objective will be to decrypt the session ID without the negotiated encryption key. If we can obtain the session ID of the connection between the victim's browser and the server, we can then hijack the victim's session and impersonate them, therefore being able to send requests on their behalf.

3.3.1 Observing the length of the original packet

As mentioned before, in order to be able to execute the CRIME attack, we must be acting as the man-in-the-middle. This is because we need to be able to force the client into sending multiple requests, and then observe them. Therefore for this example, assume that the victim visits a website `malicious.com`, which is under our control and contains malicious code. Using this website, we can force the victim to make a request to the website `example.com`, which he was accessing beforehand, with his private credentials. The request will be built in a predetermined way using several headers and it might look like this:

```
GET /index.html HTTP/1.1
Host: example.com
Cookie: Session=21047948
Accept-Charset: utf-8
```

After the request is created, compression will remove duplicate bits and then the request will be encrypted. We can then observe the encrypted packet's size. In this example, let's assume the size is 90 bytes. We will consider this the packet's base size.

3.3.2 Injecting redundant data

As the next step, we are going to be injecting data into the headers, with the intention of creating redundancies and minimizing the packet's size by compression. For example, we could inject the string „`Cookie: Session=`". The final request will then look like this:

```
GET /index.html HTTP/1.1
Host: example.com
Cookie: Session=21047948
Accept-Charset: utf-8
Cookie: Session=
```

The string „`Cookie: Session=`" is now present in the request twice. This will get recognized by the compression mechanism and the injected data will get compressed, meaning the length of the packet will remain the same size (90 bytes). Now we can start guessing the first byte of the session ID. The way to guess if, is to add a likely value of the first byte after our injected string and then observe the packet's length. If the packet's length remains the same, we have guessed the value correctly and we can proceed to the next byte.

For example, let's assume that our guess of the first byte of the session ID is „1". The string we would be injecting into the request's headers would in this case be „`Cookie: Session=1`". Since the next byte we added is not correct and therefore not common with the actual session ID, the packet's length would be greater by 1 byte. If we instead guess the value „2", the request will look like this.

```
GET /index.html HTTP/1.1
Host: example.com
Cookie: Session=21047948
Accept-Charset: utf-8
Cookie: Session=2
```

Now our entire injected string is common with the cookie containing the session ID. Because of this, it will get compressed, resulting in the same packet size of 90 bytes as the original request. When we observe the length of the packet not changing, we move onto the next byte and repeat the process, until the entire session ID is decrypted.

3.4 Prevention against CRIME

Even though the CRIME attack may be nowadays be considered impractical and outdated, the vulnerability created by header compression posed a threat to a big number of servers and browsers. Because of that, researchers and developers have since looked into developing countermeasures against the CRIME attack and the header compression data leakage exploit in general. The simplest and most effective solution is to disable header compression, or using newer versions of TLS, such as TLS 1.2 or higher. Newer versions of TLS do not support header compression, while also providing better security, compared to TLS 1.0. There are also other options for detecting and preventing CRIME.

3.4.1 Randomized padding

Similarly to the BEAST attack, we can use randomized padding to prevent the CRIME attack. The CRIME attack exploits requests having a known length by repeatedly sending requests with small changes and observing their length. The purpose of randomized padding is to add extra randomized data into a request before it gets encrypted. This will unpredictably alter the requests length, thus preventing the attacker from proceeding with the CRIME attack.

3.4.2 Use of packet pattern recognition

The CRIME attack also involves the victim sending a large amount of requests in a row, while only slightly changing the contents of the requests. Knowing about this pattern, we can then prevent the CRIME attack by the use of:

- rate-limiting techniques, with which we can limit the amount of requests from clients to the server. By limiting the amount of requests in a given timeframe, we can considerably lengthen the time required for the CRIME attack, or make it completely unviable.
- **W**eb **A**pplication **F**irewalls (WAF). We can place a special firewall in between the server and the clients with the purpose of monitoring traffic and comparing it to its rules and policies. If the traffic matches a pattern of known attacks, it can be blocked, or an alert can be triggered, for server administrators to take action.[7]

Chapter 4

Security countermeasures

As mentioned before, both the BEAST and the CRIME attack were considered threatening enough at the time of their discovery, to prompt the implementation of a series of security countermeasures on all fronts, such as client-side browser security enhancements and server-side security enhancements. With these security countermeasures in place, it has become extremely difficult and impractical to execute these attacks in today's age. In this chapter we go over the outdated technologies, which were vulnerable to BEAST and CRIME, as well as newly developed countermeasures designed to protect against these kinds of attacks.

4.1 Outdated and unsupported technologies

After the discovery of these attacks, servers and web browsers started using updated versions of their respective technologies, which address certain vulnerabilities, which made these attacks possible. A few examples of the technologies, which are outdated, or no longer supported nowadays are:

- **TLS 1.0** - in order for us to be able to execute the attacks, both the victim's browser and the server must support TLS version 1.0. This is because the attacks make use of vulnerabilities in a handful of older cipher suites, which are no longer used in the newer versions of TLS. It is still possible to enable TLS version 1.0 on both the server and the victim's browser, however it is not allowed by default and enabling it is generally heavily discouraged.
- **CBC and DEFLATE** - as mentioned before, the attacks make use of older cipher suites. The BEAST attack requires the use of any cipher suite, which uses the cipher block chaining mode. A few examples of cipher suites which use CBC are:
  - `TLS_RSA_WITH_DES_CBC_SHA`
  - `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
  - `TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA`

  The CRIME attack on the other hand would also work on cipher suites, which use CBC, however ideally the `RC4` stream cipher would be used. Same as with TLS version 1.0, it is still possible to enable these cipher suites on the web server. Unfortunately on the client side becomes be an issue. All up-to-date browsers have completely dropped all cipher suites, which are viable for these attacks due to security concerns.

Additionally, the CRIME attack also requires the `TLS DEFLATE` compression method to be used. This compression method has also been dropped from all modern browsers. With these restrictions in mind, here is a rough estimation of the versions of browsers which the client would have to be using, for these attacks to be possible:

– Internet Explorer versions between 6 to 9
– Safari versions between 5.0.1 to 6.0
– Firefox versions between 2 to 15
– Chrome versions between 4 to 25

It is worth mentioning that the `TLS DEFLATE` compression method was also dropped on the server side, due to the same security concerns. In order for the web server to support this compression method, older versions of SSL libraries would have to be used. Once again, here is a rough estimation of the versions of SSL libraries which would have to be used, for the attacks to be possible:

– OpenSSL - versions prior to 1.1.0.
– BoringSSL - versions prior to 5.0.0
– wolfSSL - versions prior to 4.0.0
– GnuTLS - versions prior to 3.0.0

4.2 Security policies

The discovery of the BEAST and CRIME attacks, as well as many others, has also encouraged web browser developers to implement a series of security policies and mechanisms. Their purpose varies depending on the individual mechanism or policy, however generally their goal is to prevent or limit all kinds of different web browser functions, which could potentially be abused by attackers. A few examples of the newly developed security policies and mechanisms are as follows:

- **SOP** - the Same-Origin Policy is a security countermeasure which is implemented by all modern browsers. Its purpose is to prevent cross-site scripting (XSS) and other vulnerabilities, which are exploitable by running malicious scripts in the victim's browser. The Same-Origin Policy allows access to the browser's resources only if the request for these resources comes from the same origin, as the origin of the resources. An origin is defined as the combination of the domain, the protocol and the port of a given web page. If any of these components do not match, the request is considered to be coming from a different origin and thus access is not allowed. For these attacks specifically, this is an issue, because part of the attacks is to send modified requests from the browser through a java applet or pure javascript. When these are executed in the victim's browser, they will attempt to send a request to the server. Here the browser will recognize that these requests would be from a different domain, and it will first send an `OPTIONS` request to the server, in order to figure out whether or not the cross-domain request is allowed. Unless the web server specifically allows access from the attacker's domain, it will not allow its resources to be shared with the scripts and any following requests coming from the scripts to the server would be denied. The Same-Origin Policy can therefore be considered a good countermeasure against the BEAST and the CRIME attacks.

- **CORS** - the Same-Origin Policy generally does not allow any requests coming from different domains, unless the server is configured to allow them. However, there are legitimate use cases, where cross-domain access is required. Because of this, the **C**ross-**O**rigin **R**esource **S**haring (CORS) was introduced. The CORS mechanism allows us to create exceptions for the Same-Origin Policy through a series of HTTP headers, which specify whether or not a cross-domain request is allowed. Unfortunately for the BEAST and the CRIME attacks, even though there may be real web servers nowadays, which might still be using CORS exceptions for the Same-Origin Policy, the attacker's domain will still not be allowed to make requests, if the server is configured correctly and securely.

- **Modifying HTTP stream (BEAST specific)** - in order to perform the BEAST attack, the attacker must have a way to directly influence the raw data going into the HTTP stream, while also having the ability to read the ciphertexts of the previous parts of the message, which were encrypted by TLS. In the original demonstration of the attack, this was done by a java applet browser exploit. However, it was recognized as a major security issue and has long been addressed. This exploit is therefore no longer possible. One way to possibly work around this issue, would be for the attacker to use a proxy. This proxy would be intercepting encrypted ciphertexts coming from the victim and then sending these ciphertexts back to the attacker's scripts in the victim's browser via a websocket connection. The scripts would then be able to make use of these ciphertext when crafting the following request blocks, however here the attacker would encounter yet another issue. For the BEAST attack to work, the attacker needs to be able to only send the beginning blocks of the request, then intercept them with his proxy, send information back to the script and then send the following updated blocks.

- **chunked transfer** - a possible solution to overcome the previous issue of modifying the HTTP stream is to make use of `chunked transfer encoding`. Chunked transfer encoding is a special type of encoding included in HTTP 1.1. Its purpose is to divide data into smaller blocks and sending them independently over a single HTTPS connection. It is nowadays mostly used for cases where the length of the data isn't known in advance, such as video streaming applications. This encoding can be specified with the `Transfer-Encoding` HTTP header[8].

Chapter 5

Demonstration of attacks

The demonstration of the BEAST and the CRIME attacks would generally include three participants:

- the server, which is serving content over HTTPS and provides a secret cookie
- the victim's machine, which is running a browser containing the secret cookie from the server. The attacker also runs his malicious code on the victim's machine in order to decrypt the secret cookie.
- the attacker's machine, which will be hosting a website with malicious code, as well as intercepting traffic from the victim to the server.

The best way to demonstrate the BEAST and the CRIME attacks in practice would be to actually perform these attacks in a controlled testing environment with a test server, a victim and an attacker. However, as mentioned in chapter 4, with several security mechanisms and policies, as well as updated versions of technologies in place, the ideal conditions for these attacks have become very difficult and unlikely to reach. Therefore a different approach for demonstrating the principles of these attacks will be used. Instead of executing these attacks directly, they will instead be demonstrated with the use of Python scripts, which closely follow the events of what would happen during the actual attacks. The following diagram describes how the attacker interacts with the victim and his communication with the server.

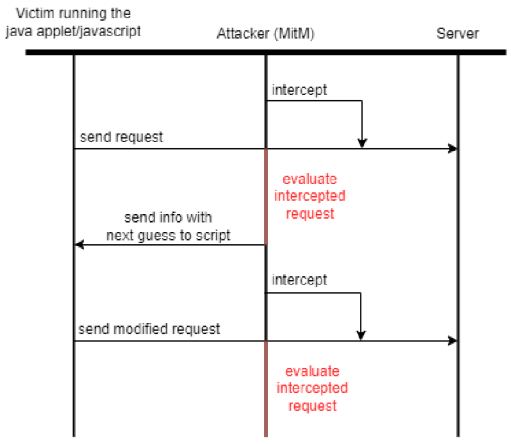

Victim running the java applet/javascript
Attacker (MitM)
Server
intercept
send request
evaluate intercepted request
send info with next guess to script
intercept
send modified request
evaluate intercepted request

Figure 5.1: Man-in-the-middle communication diagram.

The principles of the BEAST and the CRIME attack are going to be demonstrated with the use of Python proof-of-concept scripts, which follow the events of what would happen during the actual attacks as closely as possible, while also providing a clear explanation of the principles of these attacks. The version of Python used for creating these proof-of-concept scripts is `3.10.2`. In the following sections, the attacks are going to be demonstrated by running the script one time and then explaining in great detail what happens during the execution of the code.

5.1 Used libraries

Before diving into the process of demonstrating the attacks, here is a quick overview of all of the libraries, used to implement the Python proof-of-concept scripts:

- **`csv`** - used for logging data into a dataset
- **`time`** - used for tracking the time of execution of the script
- **`random`** - used for generating a random cookie
- **`os`** - used for generating encryption keys, which would under real circumstances be used for encrypting the communication between the victim and the server.
- **`sys`** - used for handling system input/output, such as specifying a custom cookie, which the attacker will be trying to decrypt.
- **`time`** - used for simulating the delay between requests, to give a rough idea of the amount of time needed for the attacks, as well as visiblity over the progress of the attack.
- **`string`** - standard Python library for working with strings.
- **`binascii`** - used for converting data from raw bytes into hexadecimal values.
- **`cryptography`** - this specialized Python library provides access to ciphers, encryption algorithms and their modes, such as CBC, which is necessary for the BEAST attack.
- **`zlib`** - this Python library provides compression and decompression of data using the **`zlib`** format. This library is used to simulate the `TLS DEFLATE` compression mechanism, which is necessary for the CRIME attack.

5.2 Demonstration of BEAST

Before the BEAST attack can start, a couple of things need to happen. Suppose that a client, who will be our victim attempts to access a web page `bank.com/index.html` using HTTPS, which is hosted on a secure server. Suppose also that both the server and the client have TLS version `1.0` enabled, the server allows encryption mechanisms which use the cipher block chaining mode and the client's browser also supports these encryption mechanisms. Since the server is secure and the client is trying to access the web page using HTTPS, the `TLS handshake` must first occur in order to establish a secure connection, before any actual application data can be sent.

The client first sends a `ClientHello` message to the server over the `TLSv1` protocol. In this message, the client lists all of the encryption mechanisms, which are supported by the browser. Suppose that one of these encryption mechanisms uses the `AES` cipher with support for the CBC mode, where the cipher block size is `16 bytes`. The server reads through all of the supported encryption mechanisms and then selects one of them, based on its configured settings. Suppose that the server also selects the same encryption mechanism, which uses the `AES` cipher with support for the CBC mode, where the cipher block size is `16 bytes`. The server then sends a `ServerHello` response to the client, in which it specifies which encryption mechanism was chosen (among other things). The client will then take note of the chosen encryption mechanism and this mechanism will then be used for the remainder of the communication between the client and the server. During this process, an encryption key is also agreed upon between the client and the server. This encryption key will be used by the `AES` cipher in order to encrypt the following communication. In the Python proof-of-concept demonstration script, the process of generating a random key for the communication is simulated by the following snippet of code:

```
AES_block_size = 16
key = os.urandom(AES_block_size)
```

For this specific instance of the communication between the client and the server, suppose that the randomly generated key has the following hexadecimal value:

```
key = 5f43d9be6f8be3529d9bbef19d6de8e6
```

After being generated, this key will now be used by both the client and the server for the purpose of encrypting messages between each other.

5.2.1 Storing the cookie

Upon the successful `TLS handshake`, the client finally requests the `bank.com/index.html` file, which he was attempting to access. The server then sends a response containing the web page. Inside of the response will also be an HTTP header for setting a cookie into the client's browser. This cookie will then be used for authentication of the client in any subsequent requests. Suppose that in this instance, the HTTP header looks like this:

```
Set-Cookie: SESSIONID=a5Ecr37c0Ok1E
```

Upon receiving the response from the server, the client's browser reads the response, takes the value of the cookie provided by the server, and stores the cookie as:

```
Cookie: SESSIONID=a5Ecr37c0Ok1E
```

With the cookie stored in the client's browser, the BEAST attack may now begin.

5.2.2 Man-in-the-Middle

Suppose that after visiting the web page `bank.com/index.html`, hosted by the secure server, the client now visits a second web page `malicious.com/index.html`, which is hosted on the our's machine (we play the role of the attacker in this demonstration). Inside of the code of this web page, we can place a java applet or some javascript, which will be executed in the client's browser. One of the capabilities of these scripts is to send HTTP requests on behalf of the client's browser. Generally speaking, we would now only have very limited control over the requests we can make with the use of javascript or java applets. This is mainly because of the implementation of the **S**ame-**O**rigin **P**olicy (SOP) as mentioned in chapter 4.2. However, for the purpose of this demonstration, let us assume that the Same-Origin Policy is either bypassed, disabled or that the server has allowed cross-domain resource sharing (CORS) for requests coming from any domain, which would implicitly include our domain. If this is the case, then we now has the ability to send requests from the client's browser, while also having control over them. Therefore we became the „Man-in-the-Middle".

5.2.3 Crafting requests

With control over the browser's requests, we can now specially craft our own requests. The goal is to decrypt the following cookie:

```
Cookie: SESSIONID=a5Ecr37cOOk1E
```

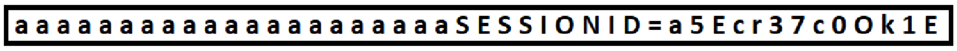
In order for us to craft a request, which can help us achieve this, we can use the following two pieces of information:

- the block size of the encryption mechanism used for encrypting the communication between the client and the browser is `16 bytes.`
- we know that the cookie begins with the string „`SESSIONID="`.

Using this information, we can artificially increase the number of characters before the cookie, in order to get the secret cookie into such a position, where the first character of the unknown part of the cookie is the last character in a block. In the real attack, this can be achieved by inflating the request path with arbitrary characters. An example of an arbitrarily inflated request path can be `POST /aaaaaaaaa`. Another thing that we need to keep in mind however, is that we do not want the block containing the first unknown character of the cookie to be the first block. This is because we need to make use of the ciphertext of the previous block. In the real attack, this is ensured naturally, due to the HTTP request containing other headers before the Cookie header. In the Python proof-of-concept script we assure this by simply another block of arbitrary characters in front of the block with the first unknown character. All of this is achieved with the following snippet of code:

```
front_padding = AES_block_size - len(known) - 1
add_bytes = AES_block_size
target = cookie
front_padded_message = "a" * (add_bytes + front_padding) + target
```

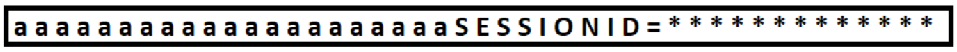
First we calculate how much front padding is needed in order for the first unknown character to be at the end of the block. We calculate this by subtracting the length of the string we know (in this case „SESSIONID=", which is 10 characters long) from the block size and then subtracting an extra 1 for the first unknown character. After that we append the letter „a" at the front of the cookie however many times necessary, so that the first unknown character will be isolated, giving us the result `front_padded_message`, which looks like this:



aaaaaaaaaaaaaaaaaaaaaSESSIONID=a5Ecr37c0Ok1E

Figure 5.2: Crafted request.

Of course, we as the attacker do not know the contents of the cookie, so from our perspective, we would only be able to know this (* marks an unknown character):



aaaaaaaaaaaaaaaaaaaaaSESSIONID=*************

Figure 5.3: Crafted request from the attacker's point of view.

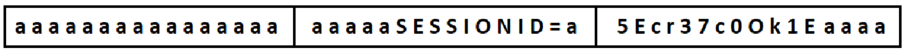
At this point, in the real attack we would send our specially crafted request, which would be processed by the encryption algorithm. During this processing, the request gets divided into blocks of `16 bytes` and in case the last block is not fully occupied, the encryption mechanism adds padding onto the end of the request to fill the gap. In the Python proof-of-concept script, this is simulated with the following snippet of code:

```
raw_message =
front_padded_message + (16 - len(front_padded_message) % 16) * "a"
```

In our case, the last block is missing 4 bytes at the end, so the character „a" is appended 4 times. After adding the padding onto the end we can now force the browser to send our crafted request. Before the browser sends it to the server, it is first encrypted. This is simulated as such:

```
def encrypt(plaintext, init_vector=0):
    if init_vector == 0:
        init_vector = os.urandom(AES_block_size)
    cipher = Cipher(algorithms.AES(key), modes.CBC(init_vector))
    encryptor = cipher.encryptor()
    return encryptor.update(plaintext)
. . .
. . .
encrypted_message = encrypt(raw_message.encode())
```

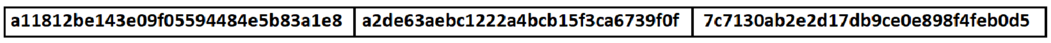
Here, the encryption mechanism first takes the raw message and then splits it into blocks of `16 bytes`, giving us the following blocks of data (we can also observe that we successfully shifted the first unknown character of the cookie into the back of the block):



aaaaaaaaaaaaaaaa	aaaaaSESSIONID=a	5Ecr37c0Ok1Eaaaa

Figure 5.4: Data split into blocks.

For the first block, a random initialization vector is generated. In this case the generated vector is `bf976b06c69d1fad5a6cab7102a3fb9c`. This vector then gets `XOR`-ed with the first block and the result then gets encrypted using the encryption key, resulting in a ciphertext. This cipher text is then used as the initialization vector for the next block and so on, according to the principles of CBC. After the encryption is complete, the encrypted blocks would get sent over to the server. On the way there, we as the attacker would be monitoring this communication and we would be able to intercept the following three encrypted ciphertext blocks:



a11812be143e09f05594484e5b83a1e8	a2de63aebc1222a4bcb15f3ca6739f0f	7c7130ab2e2d17db9ce0e898f4feb0d5

Figure 5.5: Intercepted ciphertext blocks.

In the demonstration Python script, these blocks are saved for inspection like this:

```
hexlified = binascii.hexlify(encrypted_message)
blocks = [hexlified[i:i+32] for i in range(0, len(hexlified), 32)]
```

After intercepting the encrypted messages, we can establish the three following pieces of information:

- the first intercepted block is the ciphertext block, which was used to encrypt the next block, which contains the first unknown character.
- the last intercepted block is the ciphertext block, which will be used to encrypt the next message.
- the second intercepted block is the result of a plaintext block which contained the string `aaaaaSESSIONID=*`, where `*` marks the unknown character.

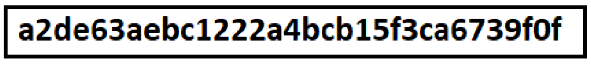
The next step for us is to replace the unknown character `*` with our guess for the first character. In the demonstration Python script we are testing for all lowercase characters, all uppercase characters and all digits. After we replace the unknown character with out guess, we `XOR` the three following values (the logic behind this `XOR` is explained in chapter 2.3):

- the first intercepted ciphertext
- the last intercepted ciphertext
- our guess of the plaintext of the second intercepted ciphertext

In the script, this is done with the following snippet of code:

```
last_iv = encrypted_message[-AES_block_size:]
iv_for_target = encrypted_message[0:AES_block_size]
guess = (known + i).encode('utf-8')
xored =
bytes([_a^_b^_c for _a,_b,_c in zip(last_iv,iv_for_target,guess)])
```

After we calculated the `XOR` value of these three items combined, we then force the client's browser to send a second request which contains the `xored` value. This request would once again go through the same encryption process as the first request and then it would be sent. We can then intercept the encrypted message again and we can find the following ciphertext:



```
a2de63aebc1222a4bcb15f3ca6739f0f
```

Figure 5.6: Intercepted ciphertext of the second request.

If our guess of the unknown character was correct, the ciphertext of the second request and the second ciphertext of the first request will be equal, therefore we found a match and decrypted the first character of the cookie. If this is the case, we shift the block boundaries again, to expose the new character and start the next iteration of steps described in chapter 5.2.3. If our guess was incorrect, then we simply follow the steps described in chapter 5.2.3 with the next character guess, but without shifting the block boundaries. We repeat this process until the entire cookie is decrypted.

5.2.4 How to use the script

The Python proof-of-concept script for the BEAST attack is submitted alongside this report. Therefore for the purpose of providing better understanding of the demonstration for the readers, this short section is dedicated to providing instructions on how to operate the script.

In order to launch the script, use the following command:

```
py beastPoC.py [cookie] [-nonalphanum] [-debug]
```

The script has three optional parameters:

- `cookie` - for specifying the cookie which the script will be decrypting.
- `nonalphanum` - for including nonalphanumerical characters in the cookie.
- `debug` - for enabling debug mode. The purpose of this mode is to provide extra detailed console output throughout the duration of the attack, in order to better highlight how the attack is proceeding.

Without debug mode enabled, the script will first display important metadata about the attack, such as the target cookie, the known prefix, the encryption key and the encryption mechanism details. Following the metadata, the script will output the current plaintext block guess, as well as the ciphertext blocks, which the script is trying to match. If the correct character is found, it will be displayed and the script continues with the next character. After the whole cookie is decrypted, the script outputs the cookie, as well as the number of requests it took for decryption.

With debug mode, additional information such as raw messages, initialization vectors and ciphertext blocks are displayed.

5.3 Demonstration of CRIME

Similarly to BEAST, before the CRIME attack can start, a few events have to happen beforehand. Once again let us suppose that there is a client, who will be our victim and he attempts to access a web page `bank.com/index.html` using HTTPS, which is hosted on a secure server. Suppose also that both the server and the client have TLS version `1.0` enabled, the server allows encryption mechanisms which use the `RC4` stream cipher and the client's browser also supports these encryption mechanisms (once again it is worth noting that the CRIME attack is not limited to the `RC4` stream cipher, but the attack's principles are best demonstrated on this stream cipher). Since the server is secure and the client is trying to access the web page using HTTPS, the `TLS handshake` must first occur in order to establish a secure connection, before any actual application data can be sent. The client once again first sends a `ClientHello` message to the server over the `TLSv1` protocol. In this message, the client lists all of the browser supported encryption mechanisms. Suppose that one of these mechanisms uses the `RC4` stream cipher. Suppose also that the browser supports the `TLS DEFLATE` compression method. This will also be communicated to the server in the `ClientHello` message. The server reads through all of the supported encryption and compression mechanisms and then selects one of each, based on its configured settings. Suppose that the server also selects the same encryption mechanism, which uses the `RC4` stream cipher, as well as the `TLS DEFLATE` compression method. After selecting, the server sends a `ServerHello` response to the client, in which it specifies the selected encryption mechanism and compression method. The client's browser will take note of these mechanisms and they will be used for the remainder of the communication between them. Same as before, an encryption key for the `RC4` stream cipher is also agreed upon between the client and the server. In the Python proof-of-concept demonstration script for the CRIME attack, the process of generating a random key is simulated by the following code:

```
key = os.urandom(AES_block_size)
```

For this specific instance of the communication between the client and the server, suppose that the randomly generated key has the following hexadecimal value:

```
key = 003314749205ba2e77d5f9204f423925
```

After being generated, this key will now be used by both the client and the server for the purpose of encrypting messages between one another.

5.3.1 Storing the cookie

Upon the successful `TLS handshake`, the client requests the `bank.com/index.html` file, which he was attempting to access. The server then sends a response containing the web page. Inside of the response will also be an HTTP header for setting a cookie into the client's browser. This cookie will then be used for authentication of the client in any subsequent requests. Suppose that in this instance, the HTTP header looks like this:

```
Set-Cookie: Cookie: SESSIONID=aV3rY5Ecr37c0Ok1E
```

Upon receiving the response from the server, the client's browser reads the response, takes the value of the cookie provided by the server, and stores the cookie as:

```
Cookie: SESSIONID=aV3rY5Ecr37c0Ok1E
```

With the cookie stored in the client's browser, the BEAST attack may now begin.

5.3.2 Man-in-the-Middle

We will once again be playing the role of the attacker for the duration of this demonstration. The first step of the CRIME attack for us is to become the „Man-in-the-Middle". Suppose that after visiting the web page `bank.com/index.html`, hosted by the secure server, the client now visits a second web page `malicious.com/index.html`, which is hosted on our machine. Inside of the code of this web page, we can place a java applet or some javascript, which will be executed in the client's browser. Similarly to the BEAST attack, we can send HTTP requests on behalf of the client's browser, with the use of these scripts. Because of the Same-Origin Policy, we would once again only have very limited control over the requests we can make using our scripts. However, for the purpose of this demonstration, let us assume that the Same-Origin Policy is either bypassed, disabled or that the server has allowed cross-domain resource sharing (CORS) for requests coming from any domain, which would implicitly include our domain. If this is the case, then we now have the ability to send requests from the client's browser, while also having control over them. Therefore we successfully became the „Man-in-the-Middle".

5.3.3 Initial request

With control over the victim's browser requests, we can now specially craft our own requests. The goal is to decrypt the following cookie:

```
Cookie: SESSIONID=aV3rY5Ecr37c0Ok1E
```

We know that the known prefix of the cookie is „`Cookie: SESSIONID=`". In order for us to decrypt this cookie, we will first force the victim to send a crafted request, where we inject the known prefix into it, in order to create a duplicate substring, which will then be compressed by `TLS DEFLATE`. In our demonstration the crafted request will look like this:

```
GET /index.html HTTP/1.1
Host: www.bank.com
Cookie: SESSIONID=aV3rY5Ecr37c0Ok1E
Cookie: SESSIONID=
```

In the Python script this is represented with the following code:

```
known = "Cookie: SESSIONID="
base_request = """GET /index.html HTTP/1.1
Host: www.bank.com""" + "\n" + cookie
request = base_request + "\n" + known
```

Afterwards, using our javascript or java applet, which is running in the victim's browser, we can force him to send our crafted request. This request would get compressed using `TLS DEFLATE` and then encrypted using the `RC4` encryption mechanism, before finally being sent.

The process of encryption and compression is simulated by the following code:

```
def encryption(raw):
    algorithm = algorithms.ARC4(key)
    cipher = Cipher(algorithm, mode=None)
    encryptor = cipher.encryptor()
    return encryptor.update(zlib.compress(raw.encode()))
. . .
. . .
encrypted_request = encryption(request)
```

We can then intercept this message on its way to the server and observe its length. Here is the code from the demonstration script, which simulates these events:

```
encrypted_request = encryption(request)
current_length = len(encrypted_request)
```

We will be able to observe, that the length of the message is `89 bytes` and we will take note of this length.

5.3.4 Inflating initial request

After taking note of the length of the initial request, we will now force the victim's browser to send more crafted requests with additional injected plaintext, using our javascript or java applet. In these requests, we will be appending an extra character after the known prefix „`Cookie: SESSIONID=`". This character will be our guess for the first unknown character of the actual cookie. We will therefore further inflate the initial request as such:

```
GET /index.html HTTP/1.1
Host: www.example.com
Cookie: SESSIONID=aV3rY5Ecr37cOOk1E
Cookie: SESSIONID=a
```

After crafting the request, we once again force the victim's browser to send this request and then intercept it on its way to the server and we observe its length. In the demonstration script this is done with the following snippet of code:

```
request = base_request + "\n" + known + x
encrypted_request = encryption(request)
```

Here, we will be able to observe, that even though we added an extra byte into the request, the size of the message is still `89 bytes`, same as before. This is because the `TLS DEFLATE` compression mechanism has compressed our additional character, meaning it was the same as the one in the original cookie, therefore we have found a match. If this is the case, we can communicate to our javascript or java applet, that we have found a match, so that it appends the given character to our known substring, so we can start guessing the next character.

If the length of the intercepted message would be `90 bytes`, that would mean that our injected character was incorrect. We would therefore repeat the process with the next character, instead of appending it to the known substring. Now we simply repeat the process starting from chapter 5.3.4, until we decrypt the entire cookie.

5.3.5 How to use the script

The Python proof-of-concept script for the CRIME attack is submitted alongside this report. Therefore for the purpose of providing better understanding of the demonstration for the readers, this short section is dedicated to providing instructions on how to operate the script.

In order to launch the script, use the following command:

```
py beastPoC.py [cookie] [-nonalphanum] [-debug]
```

The script has three optional parameters:

- `cookie` - for specifying the cookie which the script will be decrypting.
- `nonalphanum` - for including nonalphanumerical characters in the cookie.
- `debug` - for enabling debug mode. The purpose of this mode is to provide extra detailed console output throughout the duration of the attack, in order to better highlight how the attack is proceeding.

Without debug mode enabled, the script will first display important metadata about the attack, such as the target cookie, the known prefix, the encryption key and the encryption mechanism details. Following the metadata, the script will output the current substring of the request which we are injecting. If the correct character is found, it will be displayed and the script continues with the next character. After the whole cookie is decrypted, the script outputs the cookie, as well as the number of requests it took for decryption.

With debug mode, additional information such as the initial request's length, lengths of messages after compression and lengths of messages intercepted by the attacker are displayed.

Chapter 6

Data analysis

After having covered the theoretical basis behind these attacks, as well as demonstrating how each of them works, we can take a look at some other aspects. Both of the attacks have certain time requirements for decrypting secret cookies. If these attacks were to be executed in the real world, the attacker might not always have the time that's needed. It is therefore worth taking a look at just how fast these attacks can be. Another important metric of the attacks is the amount of requests. As mentioned before, both of the attacks force the victim's browser to send many requests, which then get intercepted by the attacker for the purpose of decrypting cookies. It is therefore also worth investigating how many requests these attacks take. When combining this information, we can gain good understanding of the time and request complexity of these attacks, which might help us come up with effective methods of detecting and preventing these attacks in advance.

6.1 Simulation constraints

In order to gain the understanding mentioned before, a dataset was created by observing certain metrics during the repeated execution of the Python proof-of-concept scripts. However, it is important to note, that even though the proof-of-concept scripts are made to closely follow the events of these attacks, they are still only simulations. This naturally comes with certain limitations and abstractions, which we have to take into account. One such important abstraction is time. In a real execution of these attacks, the victim first has to obtain the cookie from a legitimate server, then visit the attacker's website and load the malicious scripts. These scripts then also have to establish a connection with the attacker. All of these actions take some time, however the proof-of-concept scripts only focus on the decryption part. Therefore these actions are not considered in the following datasets.

The decryption stage of both of the attacks consists of the victim repeatedly sending requests to the server. Naturally there is always going to be some delay between each request, however in the real world, this delay is not always the same. In order to account for this, the proof-of-concept script uses an artificial delay of `10ms` between each request.

The value of `10ms` is an estimated average of delay between requests based on the following components:

- time between sending the request and the attacker intercepting it - in an ideal situation for the attacker, they would be in the same local network as the victim. Generally speaking the latency from the client to the default gateway falls within the range of `1-5ms`. It is therefore reasonable to assume, that the time frame for the attacker to intercept packets from the victim would also fall into a similar range.
- time of processing the intercepted request on the attacker's machine - the operations on the intercepted requests are not very complex nor time consuming. The time of processing requests will generally never be more than `1-2ms`.
- time of sending information from the attacker's machine to the script in the victim's browser - once again, since both the attacker and the victim would ideally be on the same local network, the time frame of `1-5ms` is considered.

With these components in mind and based on observations of delays in modern browsers, the arbitrary value of `10ms` has been chosen.

6.2 Target cookie length range

Another important aspect of decrypting cookies is of course their length. It goes without saying that the longer the cookie is, the longer it will take for the attacks to decrypt it. The length of cookies is determined by the configuration on the server side. A web performance researcher, Paul Calvano, conducted research on the length of cookies in all web pages tracked in the HTTP archive[1] in 2020. According to his research, the median value of the length of cookies is 36 characters, but it can range anywhere from just 1 character, up to a maximum of 29735 characters, while the 99th percentile is 287 characters. The created dataset is therefore targeted at the cookie length range from 20 to 56 characters with increments of 4. This provides us with sufficient understanding for the time and request requirements of the attacks for most common lengths of cookies.

6.3 BEAST attack data analysis

As mentioned before, the dataset for the BEAST attack targets the cookie length range from 20 to 52 characters with 4 character intervals in between. The Python proof-of-concept script for the BEAST attack was launched repeatedly for each interval. The dataset then describes the average of all script execution results for each interval. The main focus of the data set is to measure how big of an impact the length of the cookie has on the time and request requirements of the attack. We are also differentiating between cookies, which only use alphanumerical characters and cookies, which use non-alphanumerical characters as well, in order to see how big of an impact the complexity of the cookie has on the time and request requirements. The repeated execution of the Python proof-of-concept script produced data displayed on the following page.

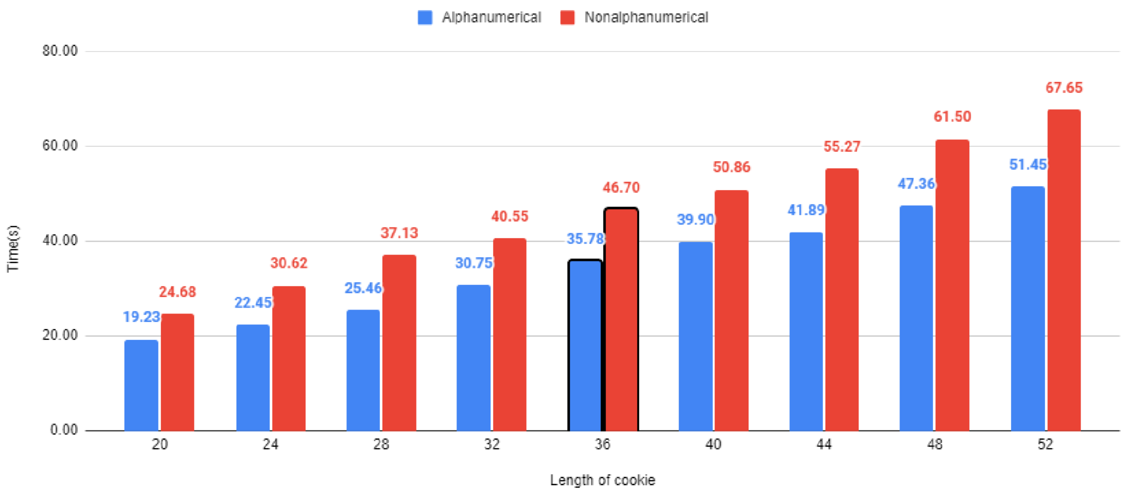


Figure 6.1: Graph of average time by length for BEAST.

Firstly, we may observe that for the range of 20 to 52 characters the decryption process of the BEAST attack takes between `19.23` and `51.45` seconds. From this we can deduce that the attack needs about 1 second on average to decrypt a single alphanumerical character. It is however worth noting again, that this time is purely an estimation, due to the simulation restrictions described in section 6.1. If we take into account the use of non-alphanumerical characters as well, the dataset shows an average increase of about `33%` in the amount of time required for decryption, giving us an average of about 1.34 seconds for a single character. Similar ratios can be observed on the following graph, which depicts how many requests on average it takes to decrypt cookies of different lengths. For alphanumerical characters only, the average amount of requests ranges from 1233 up to 3306, giving us an average of about 65 requests per character. Similarly to the time metric, we can also observe an increase of roughly `33%` in the number of requests, when taking into account non-alphanumerical characters.

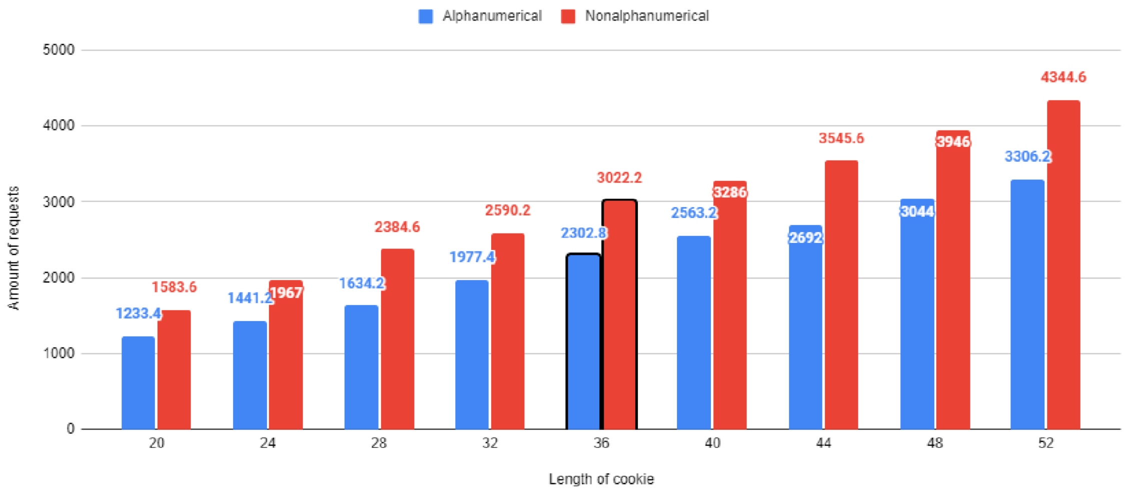


Figure 6.2: Graph of average number of requests by length for BEAST.

6.4 CRIME attack data analysis

The CRIME attack dataset targets the cookie length range from 20 to 52 characters with 4 character intervals in between. Likewise, the Python proof-of-concept script was launched repeatedly for each interval. The dataset output describes the average results for each interval, with the main focus of the dataset once again being to measure how big of an impact the length of the cookie has on the time and request requirements of the attack. The repeated execution of the Python proof-of-concept script has produced the following results:

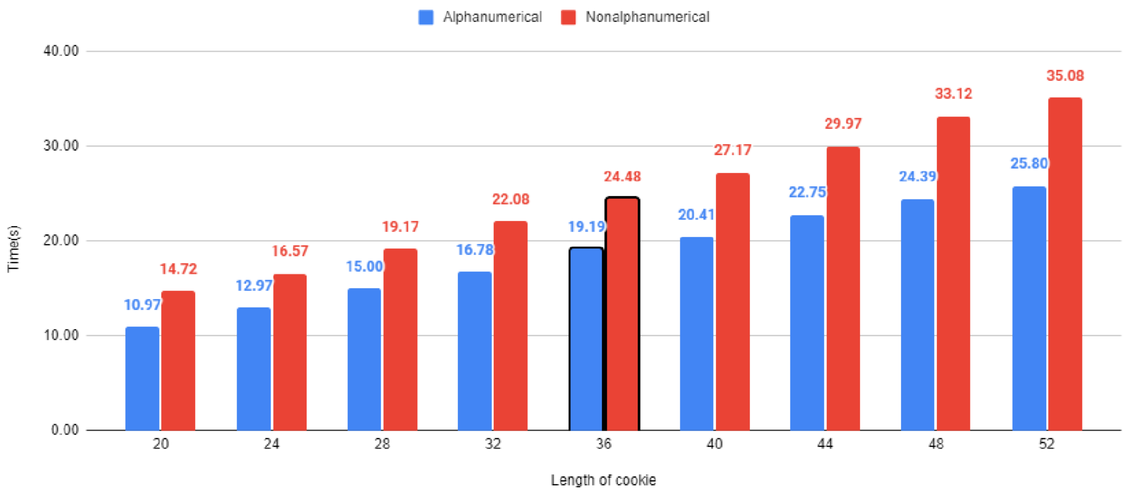


Figure 6.3: Graph of average time by length for CRIME.

From this dataset we can deduce that it takes about 0.5 seconds to decrypt a single alphanumerical character. If we consider non-alphanumerical characters as well, we can observe a 32-33% increase in time. The same ratios can also be observed on the average amount of requests by length here:

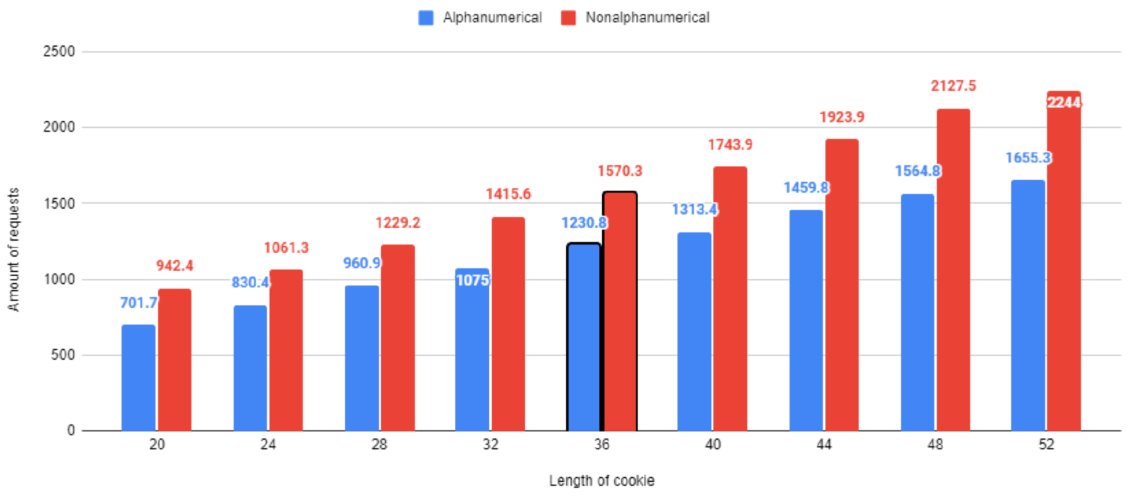


Figure 6.4: Graph of average amount of requests by length for CRIME.

6.5 Attack dataset comparison

In this section we take a quick look on how the time and request requirements of the attacks stack up to one another. This will help provide understanding on which attack is more effective, which is faster and by how much. Firstly the comparison of the attacks by the amount of time required is as follows:

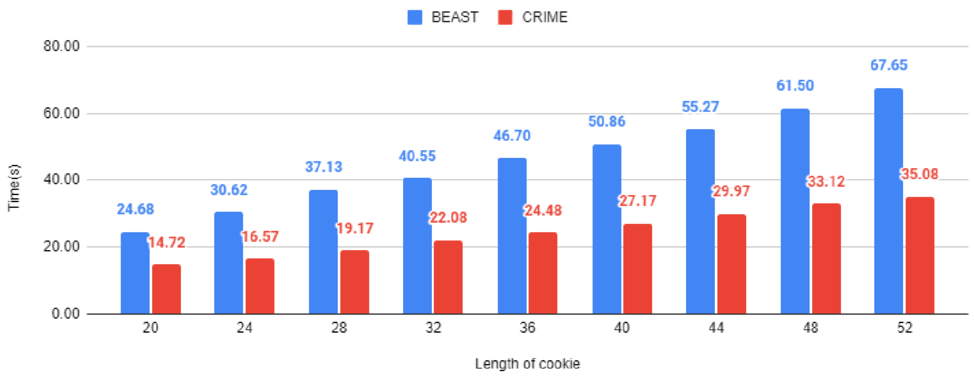


Figure 6.5: Amount of time required for BEAST vs CRIME.

Based on this data, we can see that the BEAST attack takes roughly twice as long as CRIME for cookies of the same length. This means that for shorter cookies both attacks can be viable, however with the growing length of the cookie, the BEAST attack becomes a lot less efficient in comparison to CRIME. The same can be observed on the amount of requests between the two attacks.

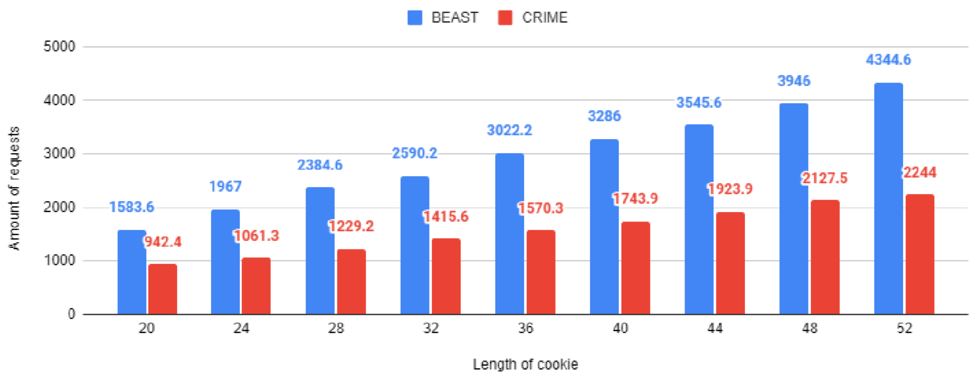


Figure 6.6: Amount of requests required for BEAST vs CRIME.

The difference in efficiency mainly comes down to the fact that the BEAST attack needs two requests for a single guess of a character, where as the CRIME attack only needs one. However, we can observe that with the use of cookies, which are at least 20 characters long, the attacks use upwards of 800 requests in just a couple of seconds. We can therefore use this information to detect the possibility of the attacks being in progress, while they are happening.

Chapter 7

Conclusion

In this report I have analyzed the BEAST and CRIME attacks against the HTTPS protocol. I have provided the theoretical basis behind them and their brief history, which is followed by an in-depth explanation of the principles of these attacks as well as ways to predict or detect them, or similar attacks which might be discovered in the future. Furthermore I have created Python proof-of-concept scripts for the purpose of demonstrating these attacks and by repeated execution of these scripts, I have obtained datasets, which describe how effective these attacks are and how they compare to each other.

The main objectives of this report are as follows:

- research the principles of the BEAST and CRIME attacks - the principles of the BEAST and CRIME attacks are described in chapters 2 and 3 respectively.
- create a suitable environment for demonstrating these attacks - the attacks were demonstrated with the use of Python proof-of-concept scripts, which are described in chapters 5.2 and 5.3 respectively.
- create annotated datasets from attack demonstrations - annotated datasets were created by repeated execution of the proof-of-concept scripts. They are described in chapter 6.
- analyze created datasets and research suitable attack detection and prevention methods - detection and prevention against the BEAST and CRIME attacks are described in chapters 2.4 and 3.4 as well as chapter 4.

This report could further be improved by demonstrating the attacks in an environment which resembles the real situation even closer, which can be done by separating the attack demonstrations into individual participants with legitimate TLS communication between them. This improvement could for example be achieved with the use of specialized Python libraries for simulating HTTPS servers and clients. Another way would be to create virtual machines and install all of the outdated versions of technologies necessary, as described in chapter 4, however some of the required technologies, such as `TLS Deflate` browser support, are very difficult to come by.

Bibliography

[1] *An analysis of cookie sizes on the web.* Paul Calvano, 2020 [cit. 5.5.2023]. Available at: https://paulcalvano.com/2020-07-13-an-analysis-of-cookie-sizes-on-the-web/.

[2] *Abbreviated TLS 1.2 Handshake.* Wikimedia, 2022 [cit. 15.4.2023]. Available at: https://commons.wikimedia.org/wiki/File:Abbreviated_TLS_1.2_Handshake.svg.

[3] *Block cipher.* Wikimedia Foundation, 2022 [cit. 25.11.2022]. Available at: https://en.wikipedia.org/wiki/Block_cipher.

[4] *Block cipher mode of operation.* Wikimedia Foundation, 2022 [cit. 27.11.2022]. Available at: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[5] *CRIME.* Wikimedia Foundation, 2022 [cit. 23.12.2022]. Available at: https://en.wikipedia.org/wiki/CRIME.

[6] *Man-in-the-middle attack.* Wikimedia Foundation, 2022 [cit. 28.11.2022]. Available at: https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[7] *What is a WAF? | Web Application Firewall explained | Cloudflare.* Cloudflare, Inc., 2023 [cit. 15.4.2023]. Available at: https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/.

[8] *What is HTTP chunked encoding? how is it used?* Bunny.net, 2023 [cit. 10.5.2023]. Available at: https://bunny.net/academy/http/what-is-chunked-encoding/.

[9] Banach, Z. *How the BEAST Attack Works.* 2020 [cit. 13.12.2022]. Available at: https://www.invicti.com/blog/web-security/how-the-beast-attack-works/.

[10] Cloudflare, I. *What happens in a TLS handshake?* 2022 [cit. 25.11.2022]. Available at: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/.

[11] Kiprin, B. *What Is the CRIME Attack and How Does It Work.* 2021 [cit. 23.12.2022]. Available at: https://crashtest-security.com/prevent-ssl-crime/.

[12] Kiprin, B. *What Is the SSL BEAST Attack and How Does It Work.* 2021 [cit. 24.11.2022]. Available at: https://crashtest-security.com/ssl-beast-attack-tls/.

[13] Kiprin, B. *What is a downgrade attack and how to prevent it.* 2022 [cit. 28.11.2022]. Available at: https://crashtest-security.com/downgrade-attack/.

[14] Phillip, R. *Evaluation of Some Blockcipher Modes of Operation* [online]. Security Evaluation. Davis, California, USA: University of California, Davis, february 2011 [cit. 27.11.2022]. Available at: https://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf.

[15] TechTarget. *Cipher block chaining (CBC)*. TechTarget, 2021 [cit. 25.11.2022]. Available at: `https://www.techtarget.com/searchsecurity/definition/cipher-block-chaining`.

[16] Willeke, J. *Beast*. 2015 [cit. 24.11.2022]. Available at: `https://ldapwiki.com/wiki/BEAST`.