DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2021

Empirical Study of HTTP Request Smuggling in Open-Source Servers and Proxies

MATTIAS GRENFELDT

ASTA OLOFSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Empirical Study of HTTP Request Smuggling in Open-Source Servers and Proxies

MATTIAS GRENFELDT AND ASTA OLOFSSON

Degree project, first cycle (15 hp)
Date: June 23, 2021
Supervisor: Robert Lagerström
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: Empirisk undersökning av HTTP-förfrågningssmuggling i servrar och proxys med öppen källkod

Abstract

During the last couple of decades cybersecurity has become increasingly important for society. As more and more of our lives turn digital, the security of the web becomes more important to its everyday users. HTTP Request Smuggling (HRS) is a vulnerability which arises when web servers and proxies interpret the length of a single HTTP request differently. In this study empirical testing was used to find parsing behaviours which could lead to HRS in six popular proxies and six servers. A literature study was conducted to compile a corpus containing requests adopting all known HRS techniques and different variations of them. A test harness was built to enable automatic sending of requests and recording of responses. The responses were then manually analysed to identify behaviours vulnerable to HRS. In total 17 vulnerable behaviours were found and by combining the proxies with the servers two almost full and three full attacks could be performed. At least one behaviour which went against the HTTP specification was found in every system tested, however, not all of these behaviours enabled HRS. In conclusion most of the proxies had strict parsing and did not accept requests which could lead to HRS. The servers however were not so strict.

Sammanfattning

Under de senaste årtiondena har cybersäkerhet blivit alltmer viktigt för samhället. Allt eftersom en större del av våra liv blir digitaliserade blir webbens säkerhet en viktigare aspekt för dess vardagliga användare. HTTP-förfrågningssmuggling (HFS) är en sårbarhet som uppstår när webbservrar och -proxys tolkar längden på en och samma förfrågning olika. I denna studie användes empirisk testning för att hitta tolkningsbeteenden som skulle kunna leda till HFS i sex populära proxys och sex servrar. En litteraturstudie genomfördes för att sammanställa ett korpus innehållande förfrågningar med alla kända HFS tekniker och olika versioner av dem. Ett testskelett byggdes för att möjliggöra att skicka förfrågningar automatiskt och dokumentera svaren. Svaren analyserades sen manuellt för att identifiera beteenden som var sårbara för HFS. Totalt hittades 17 olika sårbara beteenden och genom att kombinera proxyna med servrarna hittades två nästan fulla och tre fulla attacker som kunde genomföras. Minst ett beteende som går emot HTTP-specifikationen hittades i varje system. Däremot kunde inte alla dessa beteenden användas till HFS. De flesta proxyna som testades var strikta när de tolkade förfrågningar och accepterade inte förfrågningar som kunde leda till HFS. Servrarna var dock inte lika strikta.

Contents

Chapter 1

Introduction

During the last couple of decades, cybersecurity has become increasingly important for society. As more and more of our lives turn digital, the security of the web becomes more important to its everyday users.

There are many kinds of vulnerabilities that can be found on the web. The Open Web Application Security Project (OWASP)[1] has a project tracking the "Top 10 Web Application Security Risks"[2]. The list includes things like SQL injection and cross-site scripting (XSS). A lot of both defensive and offensive research has been conducted in these areas [1][2]. But there are also many lesser known vulnerabilities out there. One of which is HTTP Request Smuggling (HRS).

HRS arises when multiple web servers and proxies interpret a single HTTP request differently. More specifically, it happens when the different parties interpret the length of the request differently. This is caused by ambiguities in the HTTP protocol specification and differences in implementations.

Because HRS often requires two bugs, one in a server and another one in a proxy, one might think HRS is not that likely to occur. However, the impacts of HRS can be critical. Depending on how the systems are set up HRS can be used to cause cache poisoning, bypass security layers and steal cookies [3].

HRS was first discovered over 15 years ago, in 2005, by Linhart et al. [4] after which it gained little to no attention. But two years ago, in 2019, it increased in popularity as it turned out there were a lot of HRS vulnerabilities still out there despite how long ago it was discovered. Therefore this study aims to investigate how well a selection of a few open source servers and proxies have adapted to this vulnerability. This was done by empirical testing of the

[1] https://owasp.org/ Fetched: 2021-05-20

[2] https://owasp.org/www-project-top-ten/ Fetched: 2021-05-20

selected servers and proxies.

1.1 Research Question

Out of a selection of open source servers and proxies, which parsing behaviours can lead to HRS vulnerabilities when combined with other systems?

1.2 Goals

- Empirically test all servers and proxies individually and reason about their behaviour.
- Classify all strange behaviour that could potentially lead to HRS.
- Reason about how another party would have to behave for the classified behaviours to cause HRS.

1.3 Scope

The proxies were chosen based on popularity. When picking servers, the most popular servers from a few different programming languages were chosen. As of publishing this thesis, not all issues have been fixed in the systems. The systems will therefore appear with pseudonyms in this report. In total, six servers (S1-S6) and six proxies (P1-P6) were tested. Once all issues have been fixed or the responsible disclosure deadline has passed, the systems which were tested will be published[3]. This study will look at HTTP versions 0.9, 1.0 and 1.1, but the focus will mainly be on version 1.1.

[3]Which systems were tested will be published here:
https://github.com/mattiasgrenfeldt/bachelors-thesis-http-request-smuggling

Chapter 2

Background

2.1 HTTP

HTTP stands for Hypertext Transfer Protocol and is a text based ASCII-encoded protocol (at least up until version 1.1). It is the basis for the World Wide Web. HTTP is sent over TCP. TCP stands for Transmission Control Protocol and is a connection based protocol. HTTP is used for communication between clients and servers. The client sends requests and the server responds to them.

In this thesis a server is defined as a system which receives requests and responds to them. A proxy is defined as a system which receives requests, forwards them to another system, waits for their response and finally responds to the original request. A system could be either a proxy or a server.

There are five versions of HTTP: 0.9, 1.0, 1.1, 2.0 and version 3.0 is in the works. In this report the main focus is going to be on 1.1, but older versions will be mentioned. This is because 1.1 is still used to a large degree.

The HTTP protocol is standardized in so-called RFCs. RFC stands for Request For Comments. It is a standard which usually processes internet and web technologies. The first HTTP 1.1 specification is called RFC 2616 and was released in 1999 by Fielding et al. [5].

In 2007, the HTTP Working Group (HTTPWG)[1] began to rewrite the old HTTP 1.1 specification from RFC 2616. Seven years later, in 2014 this work resulted in the RFCs 7230-7235. In RFC 7230 by Fielding and Reschke [6] the syntax of HTTP 1.1 requests is specified. This document was among other things updated to address some ambiguities related to HRS. Request smuggling is also mentioned in section 9.5 of the document as a security consider-

[1]HTTP Working Group charter: https://datatracker.ietf.org/wg/httpbis/about/ Fetched: 2021-05-20

ation for implementers of the protocol.

2.2 HTTP Request Syntax

The syntax of HTTP 1.1 requests is specified in RFC 7230 [6]. A request consists of a request-line, followed by header fields, followed by an empty line that marks the end of the header fields and an optional message body. Below follows an example of a request:

```
GET /about.html HTTP/1.1    <---- Request-line
Host: example.com           <-\
Connection: close           <--+- Header fields
Content-Length: 5           <-/
                            <---- End of header fields
hello                       <---- Body
```

Each line except the body ends with a carriage return (CR) and a line feed (LF) character. When used right after each other, like in this case, the abbreviation CRLF is used. This will be the case in all our examples unless otherwise specified. It is worth noting that according to RFC 7230 [6, section 3.5], "[...] a recipient MAY recognize a single LF as a line terminator [...]".

The `Host` header contains which host the request is directed at and the middle part of the first line, also known as the request target (see Section 2.2.1) contains which resource on the host is targeted. The above request is therefore directed at `http://example.com/about.html`.

2.2.1 Request-Line

The request-line starts with a method, followed by a request target and ends with the HTTP version. They are separated by space (SP) and the line is terminated by CRLF. Below follows an example where the method is GET, the request target is /hello.html and the HTTP version is HTTP/1.1.

```
GET /hello.html HTTP/1.1
```

2.2.2 Header Fields

The header fields consist of key-value pairs separated by CRLFs. On each line, the key and the value are separated by a colon. There are many headers, but we are mainly going to focus on the Content-Length and the Transfer-Encoding headers since they determine the length of the request body.

Content-Length

The Content-Length (CL) header signifies the length of the body in bytes. If a request contains more than one CL header it should be rejected, unless they all have the same value. Example:

```
Content-Length: 5
```

Transfer-Encoding

The Transfer-Encoding (TE) header specifies the encoding used on the body of the request. There are four values this header can have according to RFC 7230 [6]. These four values are: `chunked`, `compress`, `deflate` and `gzip`. A request may contain multiple transfer encodings as a comma separated list or multiple TE headers with different values. If the TE header is present in a request, `chunked` should always be present and it should always be the last value. If a TE header contains an unknown value, the request should be rejected. Here follows two valid ways of writing TE headers. They are both equivalent.

```
Transfer-Encoding: gzip, chunked
-----------------------------------------
Transfer-Encoding: gzip
Transfer-Encoding: chunked
```

In RFC 2616 [5] another value, `identity` was also valid. When a request contains the value `identity` in a TE header, it should be treated as if the header was not included in the request. The value was however removed in RFC 7230 [6].

Combining CL and TE

According to RFC 7230 [6], if a request has both a CL and a TE header, the request could either be rejected, or interpreted with the TE header taking precedence over the CL header. Furthermore, a proxy should never forward a request containing both a CL and a TE header.

2.2.3 Message Body

The body of a request contains information to be sent to the web server. If the CL header is specified then the message body should consist of the number of bytes specified in the header. If there is a TE header with the value `chunked` then the message body is `chunked`-encoded and consists of one or more chunks. If neither the CL nor the TE header is specified the body length is assumed to be zero according to RFC 7230 [6].

`chunked` body

A `chunked` body consists of one or more chunks. A chunk consists of the chunk size specified as a hexadecimal number on the first line followed by the chunk data beginning on the next line. The chunk size is measured in bytes. The chunk data is terminated by a CRLF, not included in the chunk size. The `chunked` body ends with a final chunk with size `0`.

Below is an example of a request with a `chunked` body. It consists of two chunks with the content `hello`. The final terminating chunk consists of a `0` followed by an empty line.

```
GET / HTTP/1.1
Host: example.com
Transfer-Encoding: chunked

3             <--- Size of chunk
hel           <--- Chunk data
2             <--- Size of chunk
lo            <--- Chunk data
0             <--- Terminating chunk
              <--- Chunk data of terminating chunk
```

2.3 Persistent Connections

In HTTP 1.0 each request and response pair would by default correspond to a single TCP connection. The normal flow for a client wanting to make a request to a server can be seen in Figure 2.1. What happens is that the client opens a TCP connection to the server, sends the request, waits for the server's response and then closes the TCP connection.

In HTTP 1.1 the default behaviour was changed to use persistent connections. When a persistent connection is used, the same TCP connection is

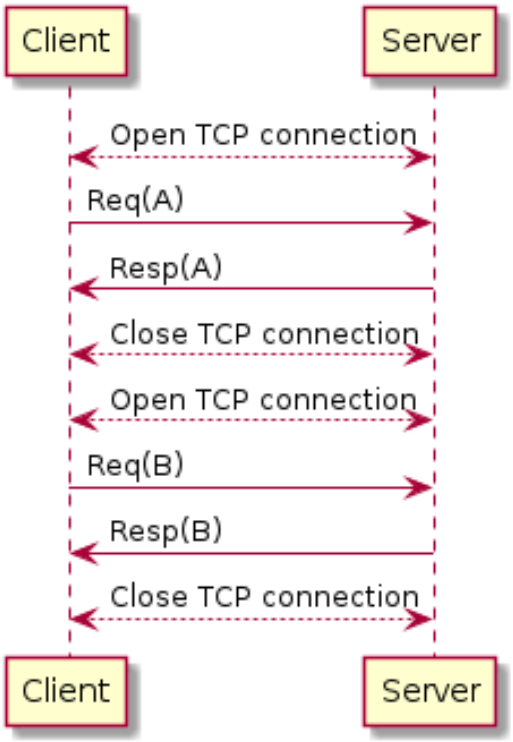

Figure 2.1: An example of a non-persistent connection.

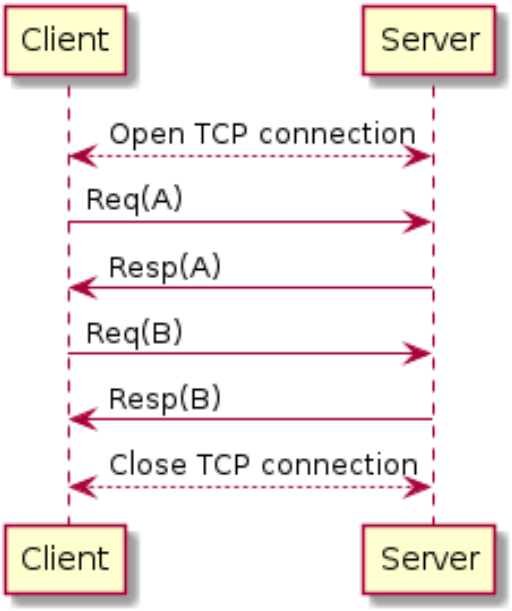

Figure 2.2: An example of a persistent connection.

reused for multiple request and response pairs. An example flow can be seen in Figure 2.2. What happens is that a client opens a TCP connection, sends request `A`, receives response `A`, sends request `B`, receives response `B` and closes the TCP connection. When a connection should be closed the party should send a message containing `Connection: close`. Persistent connections are a prerequisite for HRS to occur.

2.4 Pipelining

In HTTP 1.1 pipelining was introduced. Persistent connections are a prerequisite for pipelining. If pipelining is used, a client may send multiple requests to the server before having received a response. The server processes the requests and sends back the same number of responses in the corresponding order. An example of this can be seen in Figure 2.3. If the server does not support pipelining it will drop the extra requests and it is the client's responsibility to resend them.

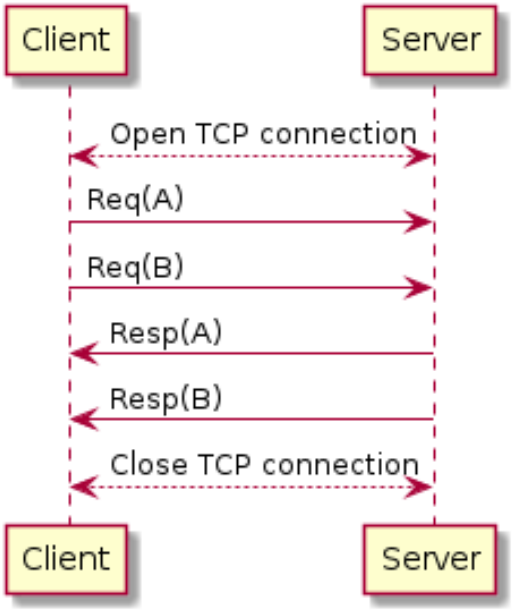

Figure 2.3: An example of a pipelined connection.

2.5 HTTP Request Smuggling

HRS is a vulnerability which arises when multiple servers and proxies interpret the same request differently. Specifically, they must interpret the body length of the request differently. For a HRS vulnerability to occur there has to be two parts which both fail in some way to interpret the request correctly, and they have to be put together. In some cases there only has to be one. Those cases are very severe because the vulnerability is always present, regardless of which other party the system is paired with. If there is a proxy and a server which together cause a HRS vulnerability, they will be called a "matching pair" in this report. If there is a proxy or a server which has a behaviour that could cause a HRS vulnerability, their "matching behaviour" is the behaviour they would have to be combined with for a HRS vulnerability to occur. Note that if a proxy forwards requests to another proxy, HRS could occur between them.

Below follows the archetypal example of a HRS attack, a double CL attack. In this scenario the proxy interprets only the first CL header and ignores any of the following. The proxy also forwards all CL headers as-is. The server instead interprets only the last CL header and ignores all preceding ones. Both of these behaviours are wrong since all requests with more than one CL header should be rejected (See section 2.2.2).

The attack can be seen below. Let the blue request be called A, the red one B and the orange one C. In Figure 2.4 a diagram of this example can also be seen.

```
GET / HTTP/1.1
Host: example.com
Content-Length: 66
Content-Length: 0

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

A malicious user sends a request A+B. The proxy interprets the first CL header which has the value 66 and thinks the request has a body of length 66. The proxy forwards the request, but the server interprets the second CL header which has the value 0 and sees only the A part. The server sends back a response to what it has read. The server then starts reading the next request, which will start with the smuggled part B. But there is not enough data in B to finish the request, so the server will wait until more data arrives. The malicious user or a normal user now sends a normal request C to the proxy which is forwarded to the server. The server now has enough data and can send a response to the B+C request. The proxy tries to read the response for the user's request, but instead it reads the response for the smuggled request. We managed to send a request which the proxy did not see.

2.5.1 Brief History

HRS was first discovered by Linhart et al. [4] in 2005. In their whitepaper they described the basic concept of HRS, presented several techniques and described the potential impacts of HRS.

In 2015 a security researcher called Régis "Regilero" Leroy started publishing research on HRS in various proxies and servers.[2] In 2016 he presented his research at DEF CON 24 in the talk "Hiding Wookiees in HTTP" [7]. He

[2]Regilero's blog: https://regilero.github.io/ Fetched: 2021-05-20

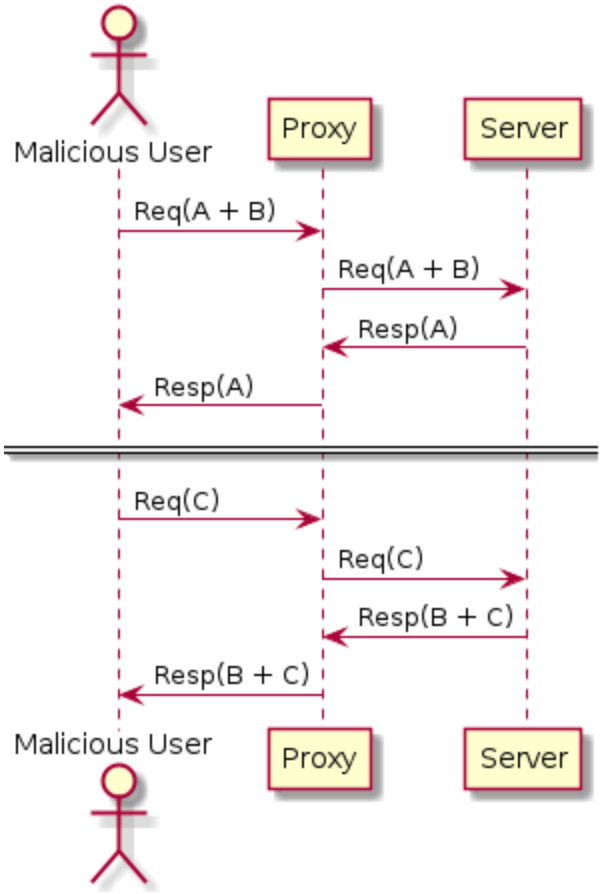

Figure 2.4: The flow of a basic HRS attack.

also published a tool for testing servers and proxies for HRS vulnerabilities called HTTPWookiee.[3] Since then he has also published more research on the topic on his blog.

In 2019 a security researcher called James Kettle gave a talk at Blackhat USA called "HTTP Desync Attacks: Smashing into the Cell Next Door" [3]. In the talk he presents his research on HRS, which he calls "HTTP Desync Attacks". Using techniques already discovered earlier and some new ones he was able to exploit many real world sites using HRS. He looked at sites that had bug bounty programs and was able to show a major impact on many of them using HRS. This talk and research garnered a lot of attention and popularized HRS.

Amit Klein was one of the researchers who worked on the original "HTTP Request Smuggling" research back in 2005. In 2020 he made a new presentation at Blackhat USA [8] explaining new HRS bugs he had found in various servers and proxies. He concluded his talk with that even though HRS has been known for 15 years there is still much uncharted territory in the area.

2.5.2 Impact

The impact of HRS depends very much on the web application and how it is structured. HRS can be used to bypass proxies as explained in Section 2.5. If the proxy acts as an extra security layer, it can be bypassed.

Another impact of HRS is cache poisoning. This can happen if the proxy has an internal cache that caches responses of frequently received requests. As a result of HRS the proxy gets the wrong response for a certain request. See how the proxy receives the response `B+C` for the request `C` in Figure 2.4. If we manage to cache the wrong response in the proxy, the proxy will start responding to everyone who sends a request the wrong way. This could for example be a request to the front page of a website. This is what is called cache poisoning.

HRS can also be used to steal someone's cookies or form data. If there is a website which supports editing and storing any kind of data it can be exploited by prefixing another users request with our own, then make the application save their request and show it to us. The same technique can also be used to leak internal headers which could be used for further exploits. [3]

[3] HTTPWookiee https://github.com/regilero/HTTPWookiee Fetched: 2021-05-20

2.6 Forward and Backward HRS

HRS attacks can be divided into two broad categories, Forward HRS and Backward HRS. The terms were first introduced by Linhart et al. [4]. The distinction comes from which party interprets the second request sent. If it is the server it is called Forward HRS and if it is the proxy it is called Backward HRS. Figure 2.4 shows an example of Forward HRS where request B is smuggled past the proxy and is interpreted by the server. This happens when the proxy can be made to see a larger body than the server.

Below follows an example of Backward HRS where the proxy forwards and ignores the meaningless `Content_Length` header but the server instead interprets it as a CL header. In this case, only the server is misbehaving by interpreting the header. The attack can also be seen in Figure 2.5. Note that the smuggling occurs in the third request and not the second one as in Forward HRS.

```
GET / HTTP/1.1
Host: example.com
Content_Length: 57

GET / HTTP/1.1
Host: example.com
Content-Length: 46

GET /forbidden HTTP/1.1
Host: example.com
```

Backward HRS only works with pipelining (See Section 2.4) enabled between the proxy and the server. In Backward HRS the server sees the request as having a larger body than what the proxy sees. When the server receives request `A` from the proxy, it will wait for the rest of the body to arrive. If pipelining is not used then the proxy will wait for a response before sending request `B` and `C`. Therefore both the proxy and server will be stuck waiting. If the proxy instead sends requests using pipelining, and the server supports it, request `B+C` will be sent right after request `A`. In that case the server will continue reading part `B` as the body of request `A` and generate a response for that. After which the server will interpret the smuggled request `C`, which was never seen by the proxy.

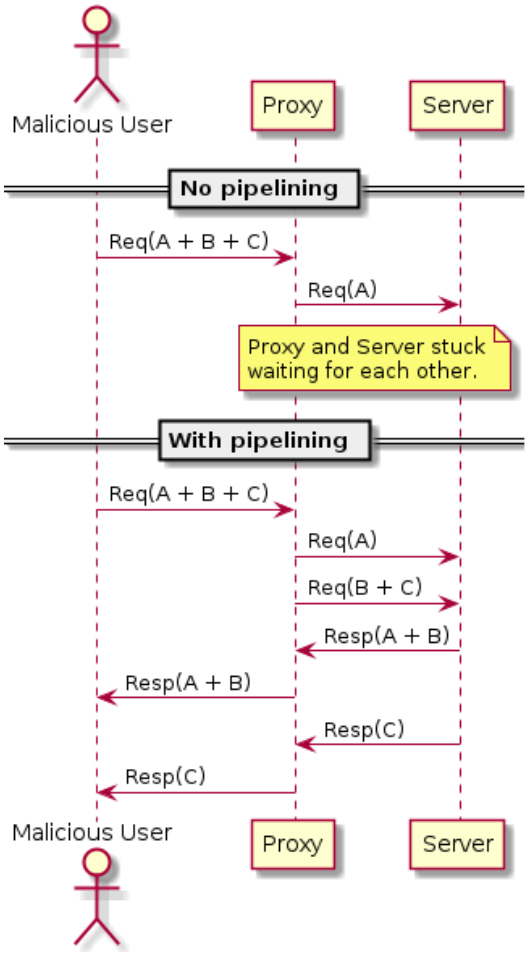

Figure 2.5: Two examples of Backward HRS. In the top one there is no pipelining between the proxy and server. In the bottom one there is pipelining.

2.7 HTTP Request Smuggling Techniques

2.7.1 Double Content Length

As explained in Section 2.2.2, it is fine to have multiple CL headers as long as they all have the same value. But the request should be rejected if they have different values. One of the first HRS vulnerabilities found by Linhart et al. [4] was a technique using two CL headers with different values. The proxy interpreted one of the headers and the server interpreted the other one. An example of this can be seen in Section 2.5.

2.7.2 Content-Length and Transfer-Encoding

As previously described in Section 2.2.2, if a request is received with both CL and TE, then TE should be prioritized. Proxies that receive such requests should never forward them with both CL and TE in the request. Linhart et al. [4] discovered that this was not always the case. Some servers did not prioritize the TE header which caused a HRS vulnerability to occur.

Here follows an example where the proxy incorrectly prioritizes the CL header over the TE header and forwards both headers. This is a Forward HRS attack.

`/forbidden` is a page only accessible to admins. The proxy enforces this. This will be the case in all of the following examples.

```
GET / HTTP/1.1
Host: example.com
Content-Length: 71
Transfer-Encoding: chunked

0

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

The proxy interprets the first and second request, blue and red, as one request, while the server behind it sees two separate requests. When the next

request, the orange one, is sent to the proxy the server answers with the response to the `/forbidden` page. The proxy has been bypassed.

2.7.3 Lax Integer Parsing

A problem that can cause HRS bugs is if integers are parsed lazily. Amit Klein [8] found that some systems accepted $+1234$ as a value in the CL header. Here follows an example of how this can cause a HRS vulnerability if a proxy interprets the number $+66$ in the CL header as 66, and the server simply ignores the CL header because it contains a $+$ in the value. This is a Forward HRS attack.

```
GET / HTTP/1.1
Host: example.com
Content-Length: +66

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

Another lax integer parsing technique is integer truncation. Integer truncation takes place when only the prefix or suffix of a string which is to be parsed as an integer is interpreted. For example if the eight byte suffix of the following string `"100000015"` were to be interpreted as an integer, it would have the value 15 not 100000015. Regilero found a suffix integer truncation bug[4] in the chunk size parsing in Go's standard library net/http package.

2.7.4 Lax TE Parsing

As mentioned previously the `chunked` value has to be present if the TE header is part of the header fields. So if a parser encounters a TE header a simple approach would be to search anywhere for the substring `chunked` in the header value, however this is not the right way to do it. For example `xchunked` could get accepted and interpreted as the value `chunked`, which

[4]https://go-review.googlesource.com/c/go/+/18871 Fetched: 2021-05-20

is not correct. Kettle [3] found HRS vulnerabilities of this sort where systems accepted `xchunked` as `chunked`.

Here follows an example of how HRS can be conducted if a proxy incorrectly prioritizes CL over TE, ignores unknown TE values and forwards both headers. The server has the behaviour described above. This is a Forward HRS attack.

```
GET / HTTP/1.1
Host: example.com
Content-Length: 71
Transfer-Encoding: xchunked

0

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

The proxy sees the first two requests, blue and red, as one and forwards it to the server behind it. The server however sees them as two separate requests because it interprets `xchunked` as `chunked`.

2.7.5 Whitespaces and Other Illegal Characters

If systems interpret line endings differently it can lead to HRS. Regilero found a bug where if a NULL-byte was encountered in the header fields the parser would continue parsing on the next line[5] and Kettle [3] found several HRS vulnerabilities involving whitespaces, such as LF being interpreted as either a line ending or part of a header value.

Here follows an example of how HRS can be conducted if a proxy interprets LF as a line ending and forwards it. The server however does not see the LF as a line ending. Instead it incorrectly sees it as part of the header value. All rows end with CRLF except for the third one. It ends with a single LF (`\n`) as seen below. This is a Forward HRS attack.

[5] https://regilero.github.io/english/security/2018/07/03/security_pound_http_smuggling/#toc6 Fetched: 2021-05-20

```
GET / HTTP/1.1
Host: example.com
Dummy: hello\n
Content-Length: 66

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

The proxy sees the first two requests, blue and red, as one. The server however sees two requests because it sees the header `Dummy` with the value `hello\nContent-Length: 66`.

2.7.6 Wrong Version

In the request-line the HTTP version is specified. The TE header was first introduced in HTTP version 1.1 [5]. So a request with version 1.0 and a TE header should be accepted, but the header should be ignored. It is just an arbitrary meaningless header. But this is not always the case. While researching response smuggling in browsers, firewalls and Intrusion Detection Systems, Steffen Ullrich found that some parties interpreted a 1.0 request with TE `chunked` as a chunked request, which is incorrect[6]. This could potentially cause HRS as well.

Another way things can go wrong is when a request is downgraded to a lower version (or upgraded to a higher version). For example if a request is interpreted as a 1.1 request by one party and a 1.0 request by another and it has a TE header with value `chunked` as well as a CL header. The server which interpreted the request as a 1.1 request will think its body is chunked while the other server which interpreted the request as a 1.0 request will read the length of the body from the CL header.

Steffen Ullrich also found that HTTP/1.010 and HTTP/01.1 in responses got treated as HTTP/1.1 or HTTP/1.0 depending on which browser was used[7].

[6] https://noxxi.de/research/dubious-http.html#hdr1.1.1 Fetched: 2021-05-20

[7] https://noxxi.de/research/http-evader-explained-10-lazy-browsers.html Fetched: 2021-05-20

This is not HRS, however the same techniques can be used. Here follows an example where the proxy interprets the version HTTP/1.010 as HTTP/1.0 and the server interprets it as HTTP/1.1. This is a Forward HRS attack.

```
GET / HTTP/1.010
Host: example.com
Content-Length: 71
Transfer-Encoding: chunked

0

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

The proxy sees the first two requests, blue and red, as one because it thinks the version of the request is 1.0, which does not support the TE header and therefore it gets ignored. The server however sees it as two separate requests because it thinks the version of the request is 1.1 which supports the TE header.

2.7.7 obs-fold

A headers value typically resides on one line. Like this:

```
header-name: abc def xyz
```

But if an obs-fold is used, the value can be spread across several lines. Like this:

```
header-name: abc
 def
 xyz
```

An obs-fold consists of a CRLF followed by one or more SPs or horizontal tabs (HTAB). According to RFC 7230 [6] one or more obs-folds in the header values should be accepted, but requests should not be sent containing it. It is for backwards compatibility. This feature is obsolete, hence the name obs-fold.

When a receiving system is to interpret an obs-folded header, it should replace all CR and LF in the header value with SP before interpreting it.

Kettle [3] found HRS vulnerabilities involving incorrect parsing of obs-folds. Steffen Ullrich also found in his research about response smuggling in browsers that obs-fold caused different interpretation of headers[8]. This could be used for HRS as well. Here follows an example of this where the proxy supports obs-fold, and the server does not support obs-fold, it simply ignores the obs-folded headers. This is a Forward HRS attack.

```
GET / HTTP/1.1
Host: example.com
Content-Length:
 66

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

2.8 Related Work

Empirical analysis, which is used in this study, is a common method in security research. In a study by Bui et al. [9] empirical testing was used to investigate how common XSS vulnerabilities are in cloud-application add-ons. They also used manual analysis, which is done in this study as well. Another study by Bui et al. [10] also used empirical testing to research the security of commercial VPN services.

In this study the CL and TE headers are the main focus. However there are many other headers that can easily be interpreted incorrectly, which can lead to security problems. One of these headers is the `Host` header. Chen et al. [11] investigated the parsing of the `Host` header in different HTTP implementations and discovered inconsistencies which could cause two different implementations to interpret the header differently.

[8] Line Folding
https://noxxi.de/research/http-evader-explained-6-whitespace.html Fetched: 2021-05-20

In a study by Tyson et al. [12] data was collected from all across the world in order to understand how networks intercept HTTP headers and found that 25% of all the measured autonomous systems modify HTTP headers. In poorly connected regions they observed a high number of cache headers being modified, indicating that caching systems were being employed. It is highly likely that the caching systems are HTTP proxies, a potential target for HRS. This means that even if a company does not itself use a proxy, their site could still get cache poisoned through the intermediary proxies in the wider network.

Fuzzing as a method of discovering vulnerabilities has risen in popularity over the last couple of years. Fuzzing involves automatically generating and mutating parser inputs with the goal of finding bugs. Using an approach called grammar-based fuzzing, highly structured input formats can be fuzzed [13]. By using the ABNF of HTTP, this approach could potentially be used to generate HRS requests.

Another example of fuzzing HTTP is AspFuzz by Kitagawa, Hanaoka, and Kono [14] which is a fuzzer for application level protocols. To demonstrate the capabilities of the fuzzer it was used on HTTP servers. It detected 25 previously reported vulnerabilities.

In a study by Gruber et al. [15] they proposed an approach which makes it possible to generate test cases automatically for protocols defined as ABNF. These test cases could then be mutated to form potential HRS requests. The requests in our study were created manually. However, using fuzzing to generate them instead is an area for future research.

One way to mitigate the risk of HRS is to use a verified ABNF parser [16]. This would prevent any syntax parsing errors that might occur otherwise. However, this does not solve semantic parsing errors, such as double CL. One reason that this is not currently used in practice might be for performance reasons.

Postel's Robustness Principle, published in RFC 793 Transmission Control Protocol, states that you should "be conservative in what you do, be liberal in what you accept from others" [17]. This principle has affected the development of many network protocols since, among others, HTTP. In a study by Sassaman, Patterson, and Bratus [18] called "A patch for Postel's robustness principle", they argue that Postel's Robustness Principle has lead to insecurity which stems from among other things, ambiguity, which is one of the causes of HRS. The patch that they propose is being more definite in what you accept, which could prevent vulnerabilities such as HRS.

Chapter 3

Method

3.1 Methodology

In Figure 3.1 an overview of our methodology can be seen. The diagram describes the workflow which was an iterative process of discovering strange behaviour, analyzing it to try and understand it and finally exploiting it. By having an iterative workflow more systems and techniques could easily be added to the analysis. In the following sections the methodology is described in detail.

3.2 Techniques and Requests

A literature study was conducted to collect different HRS techniques. All major previous research in the area was studied, the techniques were gathered, sorted and generalized into the categories that can be seen in Section 2.7. Requests were generated semi-automatically using scripts based on the techniques that were found. Existing techniques were copied straight off and some new variations were conceived. In total over 130 requests were generated and used for testing[1].

[1]The full list of requests can be found at: https://github.com/mattiasgrenfeldt/bachelors-thesis-http-request-smuggling

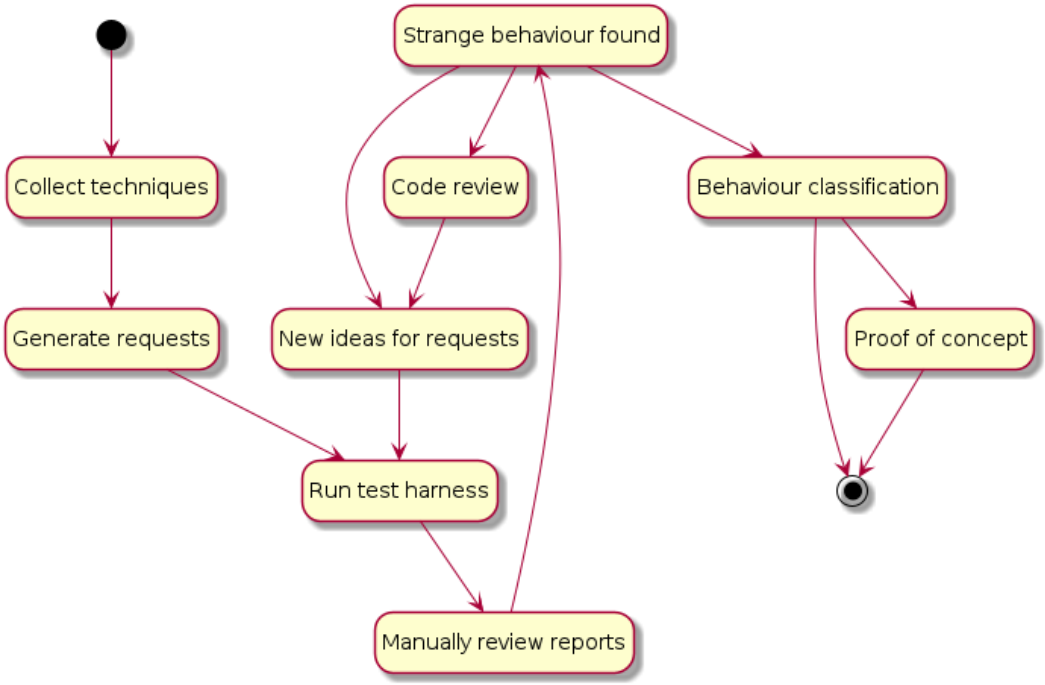

Figure 3.1: Methodology diagram.

3.3 Test Harness

A test harness was built to facilitate automatic testing[2]. The harness sends all requests and records the responses for later analysis. For each system a Docker[3] container is created. This way the systems can be managed easily and new systems can be added without modifying the harness. The harness takes a list of requests and systems and produces a report showing how all of the systems behaved when they received the requests. The testing process is different for servers and proxies, this is described below.

3.3.1 Servers

Each server was configured to respond with the size and content of the body when receiving a request. For each request that was to be tested on a server, the following process was performed:

[2]The full code for the harness can be found at:
https://github.com/mattiasgrenfeldt/bachelors-thesis-http-request-smuggling

[3]https://www.docker.com/ Fetched: 2021-05-20

1. The server was started
2. The request was sent
3. The response was received and recorded
4. The server was shut down

This way, none of the requests could interfere with each other during testing. The responses were collected and included in the resulting report.

3.3.2 Proxies

To be able to test how proxies forward the incoming request to back-ends a dummy server is used by the harness. This dummy server does not try to parse the incoming request but simply records everything that it receives and responds with a predetermined response. All proxies are configured with their default configuration with the only addition to proxy requests as a reverse-proxy to the dummy server.

For each request that was to be tested on a proxy the following process was performed:

1. The dummy server and the proxy were started
2. The harness sent the request to the proxy
3. The proxy received the request and forwarded it to the server
4. The server received the request, recorded it and responded with the predetermined response
5. The proxy received the response and sent it back to the harness
6. The harness received the response and recorded it
7. The server and the proxy were shut down

Thus, for each request, two points of data were generated. The modified request that the proxy forwarded to the server and the response that the harness received from the proxy. Since both the dummy server and the proxy were restarted for each request, no requests could interfere with each other during the testing.

3.3.3 Reports

The harness produces reports which contain the gathered data in a table. The table is organized with one column for each request and one row for each system. At the top of each column the name and content of each request can be seen. Each cell of the table contains the response of the system to the given request.

3.4 Manual Analysis

A manual analysis was conducted by looking through each report and judging, request by request, if the behaviour was according to the specification or not. When a behaviour that was not according to the specification was found, depending on our understanding of the behaviour, either new requests were written, a code review was conducted or the behaviour was classified.

If the behaviour was not understood very well, new requests were generated to narrow down what might be happening. The harness was then rerun with the new requests.

If, after several new requests had been tested but the behaviour was still not understood, a review of the source code was made. This was to identify what caused the strange behaviour and to understand it. After the review, more requests were created to verify the new assumptions about the behaviour.

When a behaviour was found that could be understood without further testing the behaviour was classified. It was classified based on whether it could cause HRS or not. A matching behaviour was then devised.

When a matching behaviour was among the already classified behaviours a matching pair had been discovered. The proxy and the server in the matching pair were set up together for further manual testing. A Proof of Concept attack was devised and tested. If it was successful a full HRS attack had been discovered.

Irrespective of whether a matching behaviour was found among the investigated systems or not the classified behaviour was reported to the respective open source project.

Chapter 4

Results

4.1 Summary

Overall, the systems tested were secure against HRS. While only 2/6 of the proxies and 1/6 of the servers had no issues whatsoever, most of the issues were only minor. The systems which had the most severe issues were the proxy P3 and the servers S1, S2 and S4. In total 17 different vulnerable behaviours were found. Two almost full and three full attacks were found.

Table 4.1 shows vulnerable forwarding behaviours in proxies. Table 4.2 shows vulnerable behaviours when receiving requests in both servers and proxies.

BEHAVIOURS / PROXIES	P1	P2	P3	P4	P5	P6
Forward and Ignore Plus in CL			x			
Forward LF as a Line Ending			x			
32-bit Integer Truncation in Chunk Size Forward			x			
Bad Chunked Body Parsing Forwarding			x			
Forward and Ignore TE Values Not Supported			x	x		
TE `"chunked"`			x			

Table 4.1: This table shows vulnerable forwarding behaviours in proxies. More details about the behaviours can be read in Section 4.2.

4.2 Vulnerable Behaviours

Here follows a description of behaviours which are vulnerable to HRS. All of them need another matching part for a HRS vulnerability to occur, if no

BEHAVIOURS / SYSTEMS	S1	S2	S3	S4	S5	S6	P1	P2	P3	P4	P5	P6
Plus Sign In CL	x	x		x	x							
LF in Header Value	x			x								
CR in Header Value	x											
Multiple CL		x		x								
64-bit Integer Truncation Chunk Size		x			x							
HTTP Version 1.0 TE `chunked`	x	x	x	x			x	x	x	x		
Ignore Rows Starting with SP								x				
Bad Chunked Body Parsing				x					x			
Ignore TE Values Not Supported	x	x	x	x	x				x	x		
0xN in Chunk Size	x			x								
Bad TE Header parsing				x								

Table 4.2: This table shows vulnerable behaviours when receiving requests in both servers and proxies. More details about the behaviours can be read in Section 4.2.

matching part was found a description of what a matching behaviour could look like is given. If a matching part was found the corresponding section is referred to.

4.2.1 Interpreting Plus Sign in CL Header

When receiving a request containing a plus sign in front of the number in the CL header the value was interpreted. This behaviour was identified in: S1, S2, S4 and S5.

A matching behaviour was found. It is described in Section 4.2.2.

4.2.2 Forwarding and Ignoring Plus Sign in CL Header

When receiving a request containing a plus sign in front of the number in the CL header, the header was forwarded as-is and no body was forwarded. This behaviour was identified in: P3.

A matching behaviour was found. It is described in Section 4.2.1.

4.2.3 Illegal Character LF in Header Value

LF was accepted as a legal character in header values by the following servers: S1 and S4.

A matching behaviour was found. It is described in Section 4.2.4.

4.2.4 LF as a Line Ending Forwarded

LF was interpreted as a line ending and forwarded in a request with LF as a line ending on a single row (all other rows were terminated by CRLF) by the following proxy: P3.

As mentioned earlier in Section 2.2 a single LF may be recognized as a line ending. However, it is unclear in RFC 7230 whether forwarding requests containing LF as a line ending is correct. Therefore it is unclear if P3 has done anything wrong here. A matching behaviour was found. It is described in Section 4.2.3.

4.2.5 Illegal Character CR in Header Value

CR was accepted as a legal character in the header values by the following server: S1.

A matching behaviour would be a proxy which interprets CR as a valid line ending and forwards it.

4.2.6 Multiple CL Headers

A request containing multiple CL headers with different values was accepted and only one of the headers was interpreted. This was found in the following servers: S2 and S4.

The matching behaviour would be if a proxy forwarded multiple CL headers and interpreted some other one than the server. This is described in greater detail in Section 2.5.

4.2.7 64-bit Integer Truncation In Chunk Size

Suffix integer truncation as explained in Section 2.7.3 was found in the parsing of the chunk size in the following servers: S2 and S5.

In both of these servers a 64-bit integer suffix truncation bug was found. For HRS to occur there would in theory have to be no matching behaviour since every proxy should parse the chunk size correctly. However in practice no proxy would be able to parse a chunk greater than 2^{64} bytes. A more realistic matching behaviour which would cause HRS would be for a proxy to have a prefix integer truncation bug and to forward the chunk size as-is. Prefix truncation existed in Apache HTTP Server in versions before 2.4.14. (CVE-2015-3183, found by Regilero.)

4.2.8 32-bit Integer Truncation In Chunk Size and Forwarding

A 32-bit integer suffix truncation bug found in P3. The proxy also forwarded the chunk size unmodified.

In theory no matching behaviour would be needed for this to cause a HRS vulnerability, but no server would be able to parse a chunk of more than 2^{32} bytes. A more realistic matching behaviour would be if a server had a prefix integer truncation bug.

4.2.9 HTTP Version 1.0 with TE `chunked`

A request with HTTP version 1.0 containing a TE header with the value `chunked` should be accepted and the TE header should be ignored. This is described further in Section 2.7.6. However, the following servers and proxies interpreted the TE header despite the version being 1.0: S1, S2, S3, S4, P1, P2, P3 and P4.

For a HRS vulnerability to occur there would have to be a proxy which correctly parses HTTP version 1.0 requests and forwards the TE header. No proxy with this behaviour was found.

4.2.10 Ignoring Rows Beginning with SP

If a request contained rows starting with SP, these rows were ignored. This behaviour was discovered in the following proxy: P2.

When P2 receives a request with rows starting with SP it ignores them and does not forward them. Therefore, there is only one way for a HRS vulnerability to occur and that is if P2 is on the receiving end. For HRS to occur when P2 is on the receiving end there would have to be a proxy which interprets a header with an SP in front as if the SP was not there and forwards the header without removing the SP.

Here follows an example where P2 acts as a server behind a proxy which wrongly reads headers with an SP in front of them and forwards them as-is. This is a Forward HRS attack.

```
GET / HTTP/1.1
Host: example.com
 Content-Length: 66

GET /forbidden HTTP/1.1
```

```
Host: example.com
Content-Length: 37

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

4.2.11 Bad chunked Body Parsing

Some systems parsed the chunked body format, as explained in Section 2.2.3, very poorly. The exact parsing behaviour differed slightly between the affected systems. They however both had in common how they parsed the chunk size. Both systems parsed a prefix of the line as the chunk size if it contained hexadecimal characters and ignored the rest of the line. We have not seen any previous research use this kind of behaviour as part of HRS. The following systems had this behaviour: S4 and P3.

With these systems acting as servers, a matching behaviour would be if a proxy with pipelining enabled would forward a request to the server which gets interpreted as `chunked` by the server, but not by the proxy.

4.2.12 Bad chunked Body Parsing and Forwarding

The following proxy parsed and forwarded a bad chunked request, as explained in Section 4.2.11, unmodified: P3.

A matching behaviour for this would be a server which interprets the body of the request as not being chunked. That way a request can be hidden in the chunked body so that P3 does not see it, but the server behind it sees it.

4.2.13 Ignoring TE Values Not Supported

A request containing a TE header value which is not supported was ignored by the following systems: S1, S2, S3, S4, S5, P3 and P4.

A matching behaviour would be a proxy which only interprets the first Transfer-Encoding header, supports the deprecated transfer coding `identity` header and forwards all TE headers unmodified.

This proxy behaviour almost existed in proxies based on Go version 1.14.15 and below. Go only looked at the first TE header, interpreted `identity`, but did not forward the TE headers unmodified.

4.2.14 Ignoring TE Values Not Supported and Forwarding

A request containing a TE header which is not supported was accepted and forwarded by the following proxies: P3 and P4.

One possible matching behaviour would be for a server to support the transfer coding `identity` and when presented with multiple TE headers only choose to interpret the first one. Go version 1.14.15 and earlier had this exact behaviour. However, this would only enable to smuggle a chunked body as a request to Go. So unless a very invalid chunked body is forwarded by the proxy (See Section 4.2.12) or the request parsing in the server is very broken, this can not be exploited.

4.2.15 TE "chunked"

A request containing a TE header with the value `"chunked"` was interpreted as `chunked` and the header was forwarded as-is. This behaviour was found in the following proxy: P3.

For HRS to occur there not only needs to be a matching behaviour in a server there also needs to be an additional issue in the proxy. If the behaviour described is matched with the behaviour from Section 4.2.13 and the proxy also suffers from the issue described in Section 4.2.12 then HRS arises.

4.2.16 Bad TE Header Parsing

A request containing a TE header with the value `FFchunked,` or `VTchunked,` where FF is the form feed character and VT is the vertical tab character, was interpreted as `chunked`. This behaviour was found in S4.

A matching behaviour would be a proxy which forwards and ignores the invalid value of the TE header, see Section 4.2.14. However, this would not be enough to cause HRS, as the two bugs only enable a chunked body to be smuggled as a request. For HRS to be possible the server would have to have an additional bug, like the one described in Section 4.2.11.

4.2.17 0xN in Chunk Size

In requests with chunked bodies where the chunk size was formatted as `0xN` it was instead interpreted as `N`. This behaviour was found in the following servers: S1 and S4.

A matching behaviour can be found in Section 4.2.12.

4.3 Almost Full Attacks

4.3.1 P3 and S4: Bad Chunked Body

This attack combines the behaviour from Section 4.2.14 in P3 with the behaviours from Sections 4.2.16 and 4.2.11 in S4. That is, the proxy ignores and forwards unknown TE values and the server interprets `FFchunked,` and `VTchunked,` as valid chunked values and also parses the chunked body poorly. This is a Backward HRS attack where a valid DELETE request sent past P3 is interpreted as a chunked body by S4. Here follows the attack:

```
GET / HTTP/1.1
Host: example.com
Transfer-Encoding: FFchunked,

DELETE / HTTP/1.1
Host: example.com
Content-Length: 46
Padding: AAAAAA...[174 times]...AAAAAA
0: x

GET /forbidden HTTP/1.1
Host: example.com
```

The first two letters of DELETE are interpreted as the chunk size of the first chunk by S4. By using a padding header, the end of the header section of the DELETE request is lined up with the terminating chunk of the first GET request.

But, since this is a Backward HRS vulnerability and P3 does not support pipelining, this attack does not work.

4.3.2 P3 with S1 or S4: 0xN in Chunk Size

This attack combines the behaviour from Section 4.2.12 in P3 with the behaviour from Section 4.2.17 in S1 or S4. That is, the proxy parses chunked bodies poorly and forwards them as-is and the server interprets chunk sizes on the format `0xN` as `N`. It is a Backward HRS vulnerability. Here follows the attack:

```
GET / HTTP/1.1
Host: example.com
Transfer-Encoding: chunked

0x57

GET / HTTP/1.1
Host: example.com
Content-Length: 51

0

GET /forbidden HTTP/1.1
Host: example.com
```

The first chunk size `0x57` would be interpreted as 0 by P3 and as 57_{16} by S1 and S4. Unfortunately this attack does not work due to an unknown bug in P3. When P3 receives a chunked request the persistent connection hangs and any subsequent requests do not yield a response, but the connection remains open. Therefore, the blue part is received and forwarded by P3, but then the attack stops.

4.4 Full Attacks

4.4.1 P3 and S1: Plus Sign in CL

As mentioned earlier in Section 4.2.2, P3 forwards requests with a plus sign in the CL header and does not forward the body. S1, S2, S4 and S5 however interpret the CL header value as if the plus sign was not there, which was mentioned in Section 4.2.1. If we put these two together we get a Backward HRS vulnerability. However, P3 does not support pipelining. This means that the HRS cannot be exploited between P3 and a server with the matching behaviour, unless there is some way to go around that.

As it turned out, a bug was discovered in S1 which causes it to send the response before reading the body of the corresponding request. This only occurs if the request handler invoked by S1 never reads any part of the body.

Here follows an example of how this could be exploited. P3 acts as a proxy in front of the S1 server. The request handler for `/` in S1 does not read the body

in this case.

```
GET / HTTP/1.1
Host: example.com
Content-Length: +23

GET / HTTP/1.1
Dummy: GET /forbidden HTTP/1.1
Host: example.com
```

P3 forwards the blue request to S1. Since the handler for `/` in S1 does not read the body it returns a response to S1 which forwards the response back to P3. S1 then starts reading the body of the request. P3 forwards the red and orange part as one request to S1. S1 however reads the red part as the body of the last request and starts reading the orange part as the second request. The sender will receive the response for `/forbidden`. The proxy was bypassed.

4.4.2 P3 with S1 or S4: LF as Line Terminator or in Header Value

As mentioned earlier in Section 4.2.4, P3 treats a single LF as a line terminator and forwards LF line endings as-is. This in combination with the behaviour described in Section 4.2.3 causes HRS between P3 and S1 or S4. Since S1 and S4 both interpret LF as being part of the header value and not the line ending. This same attack is also described in Section 2.7.5. This is a typical Forward HRS attack.

```
GET / HTTP/1.1
Host: example.com
Dummy: x\nContent-Length: 32

GET /forbidden HTTP/1.1
Dummy: GET / HTTP/1.1
Host: example.com
```

4.4.3 P3 and S1: "chunked"

This attack combines the behaviours from Sections 4.2.12 and 4.2.15 in P3 and from Section 4.2.13 in S1. That is, the proxy parses chunked bodies poorly

and forwards them as-is and it interprets `"chunked"` as `chunked`. The server ignores unsupported TE values. This is a Forward HRS attack where the chunked body of the first request is interpreted as a new request by the server.

```
GET / HTTP/1.1
Host: example.com
Transfer-Encoding: "chunked"

DELETE / HTTP/1.1
Host: example.com
Content-Length: 57
Padding: AAAAAA...[174 times]...AAAAAA
0: x

GET / HTTP/1.1
Host: example.com
Content-Length: 66

GET /forbidden HTTP/1.1
Host: example.com
Content-Length: 37

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

The first two letters of DELETE are interpreted as the chunk size of the first chunk. By using a padding header, the end of the header section of the DELETE request is lined up with the terminating chunk of the first GET request.

This attack however fails for the same reason that the attack in Section 4.3.2 fails. After sending a chunked request to P3, the persistent connection hangs. This means that the first request in the attack above is the only one that is received and forwarded by P3. Using only this one request, a Forward HRS attack can be performed where it is only possible to smuggle DELETE requests and the response can not be seen. Here follows such an attack:

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

```
Transfer-Encoding: "chunked"

DELETE /forbidden HTTP/1.1
Host: example.com
Padding: AAAAAAAA...[194 times]...AAAAAAAA
0: x
```

More headers and a message body can be added to the DELETE request by adding more chunks to the GET request.

Chapter 5

Discussion

5.1 Results

Overall the proxies tested were more secure against HRS than the servers. This might be because previous research has mostly focused on finding HRS in proxies. Since there has to be two misbehaving parties for HRS to occur, if the proxies are safe then it does not matter if the servers are vulnerable.

The servers which had the most vulnerable behaviours were S1, S2 and S4. S2 is quite a new project, which might explain why it had so many problems. S1 and S4 however are both old projects. One reason why these issues have not been discovered before might again be the result of the focus on proxies in previous research.

We however did find a proxy which had many issues, P3. P3 does not seem to be the most popular proxy, however it is still used by major organisations, according to P3's website. This shows that even though the most popular proxies such as P1, P2 and P4 are secure against HRS, there are still many large organisations using insecure proxies.

Even though most proxies are safe it is important not to ignore the server side. They share half the responsibility when it comes to HRS. Suppose a server has no protection against HRS whatsoever, the proxy only has to let one bad request through for an attack to be possible. If instead both parties have been tested against HRS, but maybe not all problems have been discovered, it is still less likely that the right combination of bad behaviours exist. But there are also servers which are secure against HRS. S3 and S6 are examples of this.

If you are using one of the most popular proxies, together with a less popular server you are more secure than if you are using a less popular proxy and a more popular server. This is because some of the popular servers of today,

such as S1 and S4, still have HRS problems.

5.2 Unexploitable Behaviour

During the project, behaviours which violate the RFC but did not cause HRS were discovered. Here follows a list showing some of the non-RFC compliant behaviours we found:

- After sending `Connection: close` the persistent connection remained open. The header was not respected.
- The HTTP version was simply echoed as the version in the response, but everything was interpreted as 1.1 regardless.
- HTTP version +1.1 and 1.+1 was accepted as version 1.1.
- Requests without `Host` headers were accepted.

This list only displays a few of the behaviours which were found but did not cause HRS. We do not know whether these behaviours can be exploited in some way. The only thing we know is that they are not according to the specification.

5.3 Test Harness

When using the test harness to test the systems it did not always catch the response for some systems. P6 and P5 were two of the proxies which we could not see the response from for some requests. We do not know if that was because they did not send a response or if there was something wrong with the test harness, however we strongly believe the fault lies in the test harness.

We did not have anything in the report which indicated whether it was the systems we tested that closed the connection or if we were the ones doing it because of a timeout. By adding that to the report created by the test harness we would have been able to see if there was something wrong with the test harness or if there was something wrong with the systems we were testing. For example, not closing a connection after a 400 error response is wrong according to the specification and could potentially be used for HRS[1].

[1] https://regilero.github.io/english/security/2019/10/17/security_apache_traffic_server_http_smuggling/ Fetched: 2021-05-20

5.4 Restarting The Systems

In our methodology the servers and proxies were restarted between each request. This has both pros and cons. If the systems are not restarted in between each request, the requests can interfere with each other. That means if a request we send makes the system behave in a strange way the consequences of that might show up many requests later. That way it can be hard to identify which request caused the strange behaviour. Some things might even slip by unnoticed because of this. By restarting the systems this can be avoided and it is certain that no interference occurred. On the other hand, there are issues which can only be caused by interference and therefore cannot be detected if the systems are restarted after each request. Because of this it would be interesting to compare or use both of the two methods in a future study.

5.5 Manual vs. Automatic Analysis

Our methodology used manual analysis of the results of the test harness. Another way to perform analysis is to automate it. There are pros and cons with both approaches. When using automatic analysis, more testing can be done in a shorter amount of time. However, it might be harder to discover things that were not anticipated.

Since we were both relatively new to the subject of HRS and the HTTP specification, we decided to use manual analysis for the project. This way, no unknown unknowns would be missed and we would get a deeper understanding of HRS. However, there are cons with doing manual analysis as well, such as misreading the results.

It would be interesting to use both approaches in a future study. We believe that the best results would probably come from using both automatic and manual analysis.

5.6 HTTPWookiee

HTTPWookiee[2] is a tool for stress testing servers and reverse proxies for RFC 7230 compliance. It can be used to look for HRS vulnerabilities.

Some of the tests that HTTPWookiee perform requires several requests to be sent in succession. Therefore it does not restart the systems between

[2]HTTPWookiee https://github.com/regilero/HTTPWookiee Fetched: 2021-05-20

each request. HTTPWookiee also uses automatic analysis on the responses received.

Although using HTTPWookiee to test for HRS would have saved time, we still opted to build our own test harness. This is because we wanted a harness which restarted the systems in between each request and to be able to do manual analysis to get a better understanding of HRS.

5.7 Hiding Requests in a Chunked Body

We encountered behaviours which allowed chunked bodies to be smuggled past the proxy, however a chunked body cannot be interpreted as a request. For example P4 interprets a 1.0 request containing a TE header with the value `chunked` as having a chunked body and forwards a 1.0 request with the TE header and the body. This is incorrect since version 1.0 did not have the TE header and it should therefore be ignored.

S6 ignores the TE header in 1.0 requests. If we put these two together we can smuggle the contents of a chunked body past P4. However, when S6 tries to parse the chunked body as the next request, it responds with an error. This is because a request should start with a request-line (Section 2.2.1) and no valid request-line is a hexadecimal number that could act as the chunk size, making the attack impossible.

A similar, but successful attack where a chunked body is smuggled can be seen in Section 4.4.3. The reason that attack is successful is because the proxy P3 parses the incoming chunked body very poorly, allowing a valid request to be interpreted as a chunked body (See Section 4.2.12).

5.8 Future Work

During the project we have identified a few different areas which might be of interest for future work:

- Increase the scope to more servers and proxies.
- Expand the request corpus.
- Experiment with restarting and non-restarting test harnesses.
- Experiment with manual and automatic analysis.
- Look at other headers than TE and CL.

- Use fuzzing to generate requests.
- Investigate the parsing of the request-line.
- Investigate other protocols for vulnerabilities similar to HRS.

Chapter 6

Conclusions

In this study we tested six open-source proxies and six servers to see if they had any parsing behaviour which could lead to HRS vulnerabilities when they were combined with other systems.

Most of the proxies did not accept requests which could lead to HRS. However P3 was not so strict about what it interpreted and forwarded, making it vulnerable to several attacks. When it comes to the servers most of them were also quite secure, except for S1, S2 and S4.

The proxies were in general more strict in their parsing of the protocol, making them less likely to be vulnerable than the servers.

In all of the systems we found at least one behaviour which went against the specification. Some of these behaviours were unexploitable. In total 9/12 of the systems tested had at least one vulnerable behaviour.

Even though the most popular systems are safe against HRS, there are still many others which are not. HRS is still a relatively unexplored area and there are many undiscovered vulnerabilities to be found.

Thank you

We want to thank our supervisor Robert Lagerström for all the good advice and support. We also want to thank Regilero who took the time to meet with us in the beginning of the project and give some guidance in this, to us, new field.

Bibliography

[1] Gary Wassermann and Zhendong Su. "Static detection of cross-site scripting vulnerabilities". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE. 2008, pp. 171–180.

[2] Christoph Kern. "Securing the tangled web". In: *Communications of the ACM* 57.9 (2014), pp. 38–47.

[3] James Kettle. "HTTP Desync Attacks: Smashing into the Cell Next Door". In: Black Hat Briefings USA. Aug. 2019.

[4] Chaim Linhart et al. *HTTP Request Smuggling*. Watchfire Whitepaper. June 2005.

[5] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, June 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[6] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor, June 2014. URL: http://www.rfc-editor.org/rfc/rfc7230.txt.

[7] Régis Leroy. "Hiding Wookiees in HTTP". In: DEF CON 24. Aug. 2016.

[8] Amit Klein. "HTTP Request Smuggling in 2020 – New Variants, New Defenses and New Challenges". In: Black Hat Briefings USA. Aug. 2020.

[9] Thanh Bui et al. "XSS Vulnerabilities in Cloud-Application Add-Ons". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 610–621.

[10] Thanh Bui et al. "Client-Side Vulnerabilities in Commercial VPNs". In: *Nordic Conference on Secure IT Systems*. Springer. 2019, pp. 103–119.

[11] Jianjun Chen et al. “Host of troubles: Multiple host ambiguities in http implementations”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1516–1527.

[12] Gareth Tyson et al. “Exploring http header manipulation in-the-wild”. In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 451–458.

[13] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. “Grammarinator: A Grammar-Based Open Source Fuzzer”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 45–48. ISBN: 9781450360531. DOI: 10.1145/3278186.3278193. URL: https://doi.org/10.1145/3278186.3278193.

[14] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. “Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols”. In: *The IEEE symposium on Computers and Communications*. IEEE. 2010, pp. 202–208.

[15] Markus Gruber et al. “Extraction of abnf rules from rfcs to enable automated test data generation”. In: *IFIP International Information Security Conference*. Springer. 2013, pp. 111–124.

[16] Alessandro Coglio. “A Formalization of the ABNF Notation and a Verified Parser of ABNF Grammars”. In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2018, pp. 177–195.

[17] Jon Postel. *Transmission Control Protocol*. STD 7. http://www.rfc-editor.org/rfc/rfc793.txt. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[18] Len Sassaman, Meredith L Patterson, and Sergey Bratus. “A patch for Postel’s robustness principle”. In: *IEEE Security & Privacy* 10.2 (2012), pp. 87–91.

TRITA -EECS-EX-2021:449