technische universität
dortmund

Bachelorarbeit

Automated Scanning for Web Cache Poisoning Vulnerabilities

Maximilian Hildebrand
September 2021

Gutachter:
JProf. Dr.-Ing. Ben Hermann
Prof. Dr.-Ing. Juraj Somorovsky

Technische Universität Dortmund
Fakultät für Informatik
Programmiersysteme (LS-5)
https://sse.cs.tu-dortmund.de

In Kooperation mit:
Hackmanit GmbH
https://hackmanit.de

Acknowledgments

I would like to thank my supervisors Ben Hermann and Juraj Somorovsky for their guidance and advice. Many thanks to my father Robert Hildebrand and my advisor Karsten Meyer zu Selhausen for proofreading comprehensively and handing out advice. I am extremely thankful to my parents Robert Hildebrand and Sabine Hildebrand for supporting me at all times. Last but not least, this thesis would not have been possible without the love and support of my wife Natalie Hildebrand and my daughter Kira Hildebrand.

Abstract

Due to the ever-growing amount of discovered attack vectors and techniques to exploit them, it is important to test for vulnerabilities in an efficient way. Automated scanners help to cover the consistently growing attack surface. In this thesis, a self-developed scanner, which tests websites for a vulnerability called "web cache poisoning", is introduced. This scanner is then used to test 51 of the world's most frequently visited websites and several ten thousand of their subdomains. The results showed that 11 of the 73620 tested URLs are vulnerable to web cache poisoning. Interestingly, they all were vulnerable to the same technique called "unkeyed header poisoning". In addition to the 11 weaknesses, 85577 false positives were identified during the tests. The causes for this large number of false positives could be identified and a mitigation is presented.

Contents

Chapter 1

Introduction

New attack vectors and techniques to exploit them are discovered constantly. Hence, it is important to test for them effectively in order to cover the ever-growing attack surface. Scanners help security professionals by covering a wide variety of attack techniques and automating repetitive tasks which are needed in order to test these attack techniques. Thus, they speed up and enhance testing procedures in order to ensure the security of systems in question. Web cache poisoning is a vulnerability where an attacker can impact HTTP responses which are served to users after requesting a certain resource, such as a website. This type of vulnerability was discovered in 2004 [1]. Since then many new techniques for this vulnerability have emerged. In particular, in the last three years several whitepapers and elaborate blog posts disclosed new web cache poisoning techniques [2][3][4][5][6][7]. While there are already some specific scanners for identifying web cache poisoning, they have shortcomings which weaken their effectiveness or even their overall usefulness. The two most outstanding shortcomings are the lack of automation possibilities and the missing support for a great variety of techniques. These shortcomings increase the effort of time necessary due to manual testing, while leaving many web cache poisoning techniques untested. This situation led to the development of a new web cache poisoning scanner called Web Cache Vulnerability Scanner (WCVS). In march 2021, the development of WCVS was started as a student project at the IT security company Hackmanit. While the scanner's core functionalities were implemented prior to this thesis, the scanner was enhanced by new techniques.

The main goals of this thesis were to implement five new web poisoning techniques, validate the reliability of WCVS and to prove its effectiveness to find web cache poisoning. This was done by testing 51 of the world's most frequently visited websites and several ten thousand of their subdomains. Afterwards, the results were analyzed with focus on the identified vulnerabilities and the occurring false positives. The structure of this thesis is the following. Chapter 1 is this introduction. Chapter 2 covers the fundamentals of the topics needed throughout this thesis. The different web cache poisoning techniques are

explained and correlated on the basis of their impact and their preconditions. Chapter 3 introduces WCVS by presenting its structure and implementation of web cache poisoning techniques. Also the strategy of WCVS to analyze caching behavior in order to find so-called “cachebusters”, which are mandatory to test for web cache poisoning, is described. Chapter 4 deals with the preparations to test websites for web cache poisoning and with the evaluation of the results. Chapter 5 finishes this thesis with a conclusion.

Chapter 2

Foundations

This chapter will strengthen the fundamentals needed throughout this thesis in four sections. These definitions will be needed in the later chapters: The first section describes the function of web caches, why they are important and what types of caches exist. The second section introduces three types of web vulnerabilities that are used during examples in this thesis. The third section specifies two classes of web cache vulnerabilities: web cache poisoning and web cache deception. The fourth section succinctly describes a tool called "Burp Suite" and why this tool is relevant in the following chapters.

2.1 Web Caches

A web cache is located between a client and a web server. Its function is to relieve the webserver from repetitive identical requests. Unknown requests are passed through to the webserver and the answer from the web server is stored for later use. If an already known request is detected, the web cache delivers the stored content back to the client without contacting the web server. This functionality can be limited to certain requests, for example static resources such as CSS or JavaScript files, by configuring the web cache. To identify an already known request a so-called "cache key" is used. This cache key consists of specified parts of the HTTP request, for instance the domain, path and headers. Parts of the HTTP request which are included in the cache key are called "keyed", while those parts which are not included in the cache key are called "unkeyed". When a client issues an HTTP request the web cache generates the corresponding cache key and checks whether it has a copy stored for this cache key. If that is the case, the web cache directly responds with the stored copy. Otherwise, the web cache forwards the request to the web server. The web cache stores the response of the web server and forwards it to the client. The web cache can be configured, which parts of the request are included in the cache key and which are not. Also it can be specified how long a copy is stored by the web cache, for example for thirty seconds. The attacker needs to either predict the moment that the cache gets

cleared, send as many requests as possible to raise his chances or be lucky to send the first request to an empty cache. Throughout this thesis it is assumed that the attacker manages to get his requests cached either way. Web caches are widely used because of their great benefits for websites. Their advantages include reducing network traffic, network latencies and the load on web servers and their databases [8][9]. There are multiple types of web caches that can be classified into shared and private ones [10][11]. Private web caches only store copies for one user, whereas shared web caches store copies for multiple users. Reverse proxy servers, forward proxy servers, web servers, web frameworks and templates may contain a shared web cache. Content delivery networks (CDN) are a group of servers which can be spread around the world to provide fast access to content. Clients send their requests to "edge servers" which often use a shared web cache [12]. More than 74% of the alexa top 1k websites were found to use a CDN [13]. Those shared web caches can often be modified to store copies per user via the cache-control directive [14, Section 5.2.2.6], in order to use them as private caches. Throughout this thesis the term web cache will refer to a reverse proxy containing a shared web cache for unification.

2.2 Web Site Vulnerabilities

Websites can have many different kinds of vulnerabilities. Some of them can be more harmful when they are combined with web cache poisoning, which will be explained in the following subchapter. Three vulnerabilities which are used for examples throughout this thesis are the following:

Cross site scripting (XSS) [15]: A website which is vulnerable to XSS enables an attacker to inject malicious javascript code, called XSS payload, to it. When a web server returns in its response a part of the unsanitized user input which contains the XSS payload it is called reflected XSS. An attacker has to trick a user into issuing a malicious HTTP request, for example by sending the user a link where the XSS payload is embedded into the query string. If a XSS payload is stored on the server, such as in the database, it is called stored XSS. In that case no user interaction is needed. To prevent XSS vulnerabilities user input needs to be "sanitized". This means that the website ensures that the user input doesn't contain any valid XSS payload.

Denial of service (DOS) [16]: The goal of a DOS attack is to make a website unavailable. This can be done in multiple ways, for instance a web server can be overflood with packets until the web server's capacity is overloaded. If an attacker tricks a web server into issuing an error response after requesting a certain web page and the web cache stores this error response, this would also be considered a DOS. As users, which want to request the earlier requested web page, will receive the error response

Impact	Technique								
	1	2	3	4	5	6	7	8	9
Malicious Content Injection	(x)	(x)	(x)	(x)	(x)	x			
Open Redirect	(x)	(x)	(x)	(x)	(x)				
Denial of Service	(x)	(x)	(x)	(x)	x		x	x	x

Table 2.1: Impact of the different web cache poisoning techniques.

instead of the web page. Another way is to cache a redirect, as explained in the following vulnerability.

Open redirect [17]: In a so-called "open redirect" an attacker is able to trick a web server into redirecting a user to an arbitrary, potentially malicious, website. The redirection can either be an intended feature or an oversight, if for example the web framework uses a specific header to issue a redirect instead of its otherwise normal response. An open redirect can also be considered as a denial of service if it gets cached and all users are prevented from accessing a website because they get redirected to another website or to a not available server.

2.3 Web Cache Vulnerabilities

Despite their advantages, web caches also have disadvantages, as they allow new attack vectors such as web cache poisoning and web cache deception. The difference between the both attack vectors is the following. During web cache poisoning an attacker aims to store a malicious response in the web cache which gets served to other users. During web cache deception an attacker tricks a victim to use a modified link which tricks the web cache to store a web page which would not be cached otherwise. Afterwards the attacker can use the same link to receive the cached web page. The focus of this thesis lies on web cache poisoning and not on web cache deception. But as web cache deception can be accidentally found when testing for web cache poisoning, it will be explained shortly at the end of this subchapter.

2.3.1 Web Cache Poisoning Impact

Table 2.1 illustrates the typical impact of web cache poisoning, which are malicious content injection, redirection and denial of service. The numbers one to nine reference different web cache poisoning techniques, which will be clarified in Section 2.3.3. A `x` indicates that a successful exploitation of this technique leads to the corresponding impact. Whereas a `(x)` indicates that this impact may be possible but depends on the website already being vulnerable to this kind of exploit. The impact of those vulnerabilities is often increased

when they are used in combination with web cache poisoning. Further details on the improved impact will be mentioned during the clarification of each technique. `Malicious content injection` means that an attacker can either inject a malicious payload, such as XSS, in the body of the cached response or that the attacker even has complete control over the cached response. The latter is only the case for technique 6 namely HTTP Response Splitting. HTTP Response Splitting is also the only technique where this is a guaranteed impact. For technique 1-5 it is only potential and depends on the website having a content injection vulnerability. `Open Redirect` is, when it is a possible impact, only potential. It always requires the website to have a redirect ability which can be triggered with either headers or parameters. As technique 6 HTTP Response Splitting enables the attacker to have full control over the cached response, they could also generate a redirect response. However, this would lessen the impact. The attacker can generate an arbitrary response which will be issued by the targeted web server. While a direct redirect could be noticed by a victim. `Denial of Service` can be accomplished with three different techniques by the web cache poisoning techniques. Technique 7-9 trick the web server to return an error page instead of the requested web page. If the web cache stores this response, users who try to access the web page will receive the error page instead. Technique 6 namely HTTP Request Splitting entangles the web cache which response of the web server belongs to which request. This can lead to users trying to access web page X receiving always the cached response for web page Y instead; Thus Denying the access to web page X. Technique 1-4 fully depend on the website and may differ from every website. The impact of Denial of Service is only potential for them and will be further discussed when the techniques are clarified. Four techniques rely solely on the website having already one of these three vulnerabilities. If the website does not have any of these vulnerabilities any of these four web cache poisoning will have no impact, even if they are successful. In case that a website has two or more of these three vulnerabilities, it is up to the attacker which one of these they want to utilize. Four other techniques only have one possible impact, however they do not rely on the website having another vulnerability. If these four techniques are successful the impact is guaranteed. One technique has DOS as guaranteed impact and the other two impacts are potential, depending on if the website is already vulnerable to a redirect or malicious content injection.

2.3.2 Web Cache Poisoning Preconditions

Table 2.2 illustrates the preconditions which need to be met in order for the web cache poisoning techniques to be possible. Just like Table 2.1, the numbers one to nine reference different web cache poisoning techniques, which will be clarified in Section 2.3.3. `Unharmonized configuration` indicates that the web server and the web cache are configured differently and that this difference can be exploited to achieve web cache poison-

Precondition	Technique								
	1	2	3	4	5	6	7	8	9
Unharmonized configuration			x	x			x		
Non-compliant with RFC					x	x	x	x	
Header with impact	x			(x)		x1			x
Parameter with impact		x	x	x		x2			
Unkeyed header	(x)			(x)		(x1)	x	x	(x)
Unkeyed parameter		(x)	x			(x2)			

Table 2.2: Preconditions of the different web cache poisoning techniques.

ing. `Non-compliant with RFC` indicates that either the web server or the web cache needs to be non-compliant with a certain RFC. `Header with impact` has two different indications. For technique 4 and 9 it is indicating that a particular header which overrides the HTTP method, such as X-HTTP-Method-Override, needs to be supported by the web server. As this is only needed for 1 of 3 variants of technique 4 a `(x)` was used instead of a `x`. For technique 1 and 4 it is indicating that an arbitrary header, whose name or value has an impact on the response, is needed. In both cases the particular header has to be unkeyed. `Parameter with impact` is indicating that an arbitrary parameter, which has an impact on the response, is needed. This parameter has to be unkeyed for technique 2 and technique 6. Technique 3 and 4 can be used to bypass that the impactful parameter is keyed. In exchange they have more preconditions than technique 2. The impact that `Header with impact` and `Parameter with impact` is referring to depends solely on the website. If a header or parameter is reflected in the body this could lead for example to malicious content injection, such as XSS. If the header or parameter triggers a redirect, this would be an Open Redirect. In case that for example an error page or empty page is returned, this would be a DOS. There might be further impacts depending on the functions of the website. `x1` and `x2` at the column of technique 6 indicate that either one of them has to be true. `Unkeyed header` indicates that an unkeyed header is needed for this technique to be successful. A `x` means that this can be any arbitrary header, as long as it is an unkeyed header. A `(x)` means that the header with impact has to be unkeyed. `Unkeyed parameter` indicates the same as unkeyed header but for a parameter. A `x` means that any arbitrary header is possible, as long as it is unkeyed. A `(x)` means that the parameter with impact has to be unkeyed. All techniques aside from technique 5 HTTP Request Smuggling need either an unkeyed header or parameter. 6 of the 8 techniques require an unharmonized configuration of the web server and web cache or a non-compliance with a specific RFC of the web server or web cache.

2.3.3 Web Cache Poisoning Techniques

Listing 2.1: Request with cache key "example.com/index.php"

```
1 GET /index.php?user=max HTTP/1.1
2 Host: example.com
3 Cookie: uuid=0bd0a3a4-02f8-4971-afbf-7fe00421337e
```

Listing 2.2: Response to previous request

```
1 200 OK
2 [...]
3 <p>Welcome back max</p>
4 [...]
```

Listing 2.3: Malicious request with cache key "example.com/index.php"

```
1 GET /index.php?user=<script src=att.ac/ker.js></script> HTTP/
  1.1
2 Host: example.com
3 Cookie: uuid=1ba0d3f4-03f8-4961-afbc-7fe00421337e
```

Listing 2.4: Response to previous malicious request

```
1 200 OK
2 [...]
3 <p>Welcome back <script src=att.ac/ker.js></p>
4 [...]
```

Common Flow for Most Techniques Unkeyed input is every part of an HTTP request that does not influence the cache key, whereas keyed input is every part which is used to generate the cache key. In Listing 2.1 and 2.3 keyed input is visualized in bold. While both requests have the same cache key "example.com/index.php" they differ in their responses, which are illustrated in Listing 2.2 and 2.4. The body of the second response contains XSS which was injected by the attacker using the `user` parameter. If the cache currently does not have a cached response for the cache key "example.com/index.php" and the attacker is the first to send the request, the response containing the XSS would get cached. All users requesting "example.com/index.php" now receive the malicious response from the web cache. All techniques, besides technique 5 HTTP Request Smuggling, utilize such a factor that there is a possibility to trigger a different, potentially malicious, response without changing the cache key.

1. Unkeyed Header Poisoning

- Preconditions: Unkeyed header poisoning has two preconditions. First, there has to be a header which has an impact on the response of the website. Second, this header has to be unkeyed. An impact is for example, that the name or value of the header is reflected in the header or the body of the response or that the header leads to a redirect.
- Flow: If the preconditions are met an attacker can send a request which contains this impactful header. The response which is returned by the web server now contains for example a redirect to the value of the impactful header. This response gets cached by the server. Other users requesting the same web page will receive the cached response, as the impactful header is unkeyed and does not change the cache key.
- Impact: The impact of unkeyed header poisoning relies solely on the impact of the unkeyed impactful header. This can be for example malicious content injection, such as XSS, an Open Redirect or a DOS. If the website does not have any unkeyed impactful header this technique cannot be conducted.

A few examples for headers which might have, depending on the website, an impact are the following (found in [2],[4]): The X-Forwarded-Host header could, sometimes in combination with the X-Forwarded- Scheme header, lead to an Open Redirect or XSS. The X-Forwarded-Port header might change the port the request is forwarded to and therefore lead to DOS. The X-Original-URL or X-Rewrite-URL could change the path of the request and therefore lead to an Open-Redirect. The Transfer-Encoding set to a not valid value can trigger a 501 Not implemented response which might be stored by a web cache and result in DOS. The Max-Forwards header set to 0 could result in the web server not processing the request and only returning the request as the response body leading to a DOS.

2. Unkeyed Parameter Poisoning The preconditions, flow and impact are similar to unkeyed header poisoning. The only difference is, that instead of an unkeyed impactful header an unkeyed impactful parameter is used. If no unkeyed impactful parameter can be found, still a keyed impactful parameter can be used by technique 3. Parameter Cloaking and technique 4. Fat GET.

3. Parameter Cloaking

- Preconditions: There are three different preconditions that all must be met. The first precondition is that there has to be an keyed impactful parameter (in contrast to unkeyed parameter poisoning). The second precondition is that there has to be an arbitrary unkeyed parameter. The third precondition is that the web cache and web server need to be configured unharmonized in respect to the parameter delimiters.

Listing 2.5: Fat GET request

```
1 GET /index.html?lang=de HTTP/1.1[CRLF]r]
2 Host: target.com
3 Content-Length: 7
4
5 lang=pl
```

While RFC 3986 [18, Section 2.2] allows `&` and `;` as query parameter delimiters, many web caches or web frameworks don't treat `;` as a delimiter.

- Flow [5]: Let the parameter `imp1` be an impactful parameter and the parameter `unk2` an unkeyed parameter. An attacker can create a malicious URL by exploiting an unharmonized configuration of a web server and web cache. This malicious URL has the query string "imp1=foo&unk2=bar;imp1=malicious". Ruby on Rail interprets this query string as three query parameters: "imp1=foo", "unk2=bar" and "imp1=malicious". The last occurence of `imp1` would override the previous one, so that `imp1` has the value "malicious". A web cache which does not honor `;` as a query parameter delimiter interprets the query string as only two query parameters: "imp1=foo" and "unk2=bar;imp1=malicious". As `unk2` is unkeyed - which is often the case for tracking and marketing parameters - the cache only includes "imp1=foo" in the cache key even if Ruby on Rails uses "imp1=malicious". Therefore the malicious value of imp1 was cloaked by the unkeyed query parameter unk2.

- Impact: The impact of parameter cloaking solely depends on the impactful query parameter, just like unkeyed parameter poisoning and fat GET does.

4. Fat GET

- Preconditions: There are two to four preconditions, depending on the three variants. All variants need an keyed impactful parameter, as first precondition, and an unharmonized configuration, as second precondition. Although RFC 7230 [19, Section 3.3] allows the use of a request body during a GET request it is very uncommon. Hence the configuration of a web cache might not include the request body in the cache key, while the website processes the request's body. If the website does not process the request's body the first variant is not possible and the other variants, which require more preconditions can be tried. They might trick the web server into processing the body. The third precondition for the second variant is that the web server supports the POST HTTP method. The third precondition for the third variant is that a

Listing 2.6: HTTP request smuggling example [21]

```
1  POST /index.html HTTP/1.1[CRLF]
2  Host: target.com[CRLF]
3  Content-Length: 0[CRLF]
4  Content-Length: 62[CRLF]
5  [CRLF]
6  GET /poison.html HTTP/1.1[CRLF]
7  Host: target.com[CRLF]
8  HeaderWithSpace: [Only a space and no CRLF]
9  GET page_to_poison.html HTTP/1.1[CRLF]
10 Host: target.com[CRLF]
11 [CRLF]
```

header such as "X-HTTP-Method-Override: POST" is supported and unkeyed. This might change the HTTP method to POST.

- Flow [5]: The fat GET technique enables an attacker to overwrite the value of a keyed parameter, without changing the cache key. In contrast to parameter cloaking, the URL remains unaffected. Instead, as Listing 2.5 shows, the query parameter, which shall be poisoned, is written into the body with the malicious value. Some frameworks, such as Ruby on Rails, read the body of GET requests as well and prefer body parameters over query parameters. According to the RFC 7231 [20, Section 4.3.1] it is an allowed but also uncommon behavior. If a framework ignores the body of GET requests it might be possible to use `X-HTTP-Method-Override: POST` or similar headers to change the method to POST or even immediately send a POST instead of a GET request. For this technique to work the body, HTTP method, and - if used - the `X-HTTP-Method-Override` header must not be part of the cache key.

- Impact: The impact of fat GET solely depends on the impact of the impactful query parameter, just like unkeyed parameter poisoning and parameter cloaking.

5. HTTP Request Smuggling (The explanation of this method has been basically described in [21] and was enhanced by content of the related RFCs)

- Precondition: HTTP Request Smuggling is the only technique which does not require any unkeyed header or parameter. Its only requirement is non-compliance with RFC 2616 [22, Section 4.4]. There are two ways to specify the content length. First the `Content-Length` header, which indicates how many bytes long the content

Listing 2.7: Common request to HTTP response splitting vulnerable website and its response [1]

```
1 http://www.example.com/redirect=http://www.foo.bar/

1 HTTP/1.1 301 Moved Permanently
2 Location: http://www.foo.bar/
```

is. Second the `Transfer-Encoding: Chunked` header, which allows it to send and receive chunks of the content independently from each other. If the content is divided into multiple chunks, a hexadecimal number in front of each chunk specifies how long the corresponding chunk is. The last chunk doesn't contain any content and is announced with the number 0. RFC 2616 [22, Section 4.4] declares that if both the `Content-Length` and the `Transfer-Encoding` header is used, the `Content-Length` header must be ignored. However, not every web cache or web server complies to this RFC. The web cache might utilize the `Content-Length` header, while the web server might utilize the `Transfer-Encoding` header, or vice versa. Likewise, an attacker could send two `Content-Length` headers and the web cache and web server might utilize the respective other one. If one of those prerequisites is met, an attacker could try to exploit HTTP request smuggling.

- Flow: Listing 2.6 illustrates such an attack. In this example the web cache is utilizing the second `Content-Length` header, while the web server utilizes the first one. As the web cache assumes that the content has a length of 62, it considers line 6-8 as body of the first request. The lines 9-11 are interpreted as a second request because the previous request was already completed. The web server on the other hand assumes that the content length is 0, because it utilizes the first `Content-Length` header. Hence the web server interpretes the lines 1-5 as the first request and the lines 6-11 as the second request. The line 9 will be considered as the value of the `HeaderWithSpace` header at line 8, because the header is only followed by a space and no `CRLF`. If the second request gets cached, every user who tries to access `http://target.com/page_to_poison.html` will instead receive a copy of the response of `http://target.com/poison.html`.

- Impact: A guaranteed impact is that the access to a web page can be denied, because the web cache returns the cached response from another web page of the web server. If this other web page contains any vulnerability, such as XSS, this vulnerability is also served to users who try to access the original web page.

6. HTTP Response Splitting

Listing 2.8: HTTP response splitting request [1]

```
1 http://www.example.com/redirect=http://foo.bar/%0d%0a%0d%0aHT
  TP/1.1%20200%20OK%0d%0aContentType:%20text/html%0d%0aContent-
  Length:%2022%0d%0a%0d%0a<html>malicious</html>
```

Listing 2.9: HTTP response splitting response [1]

```
1 HTTP/1.1 301 Moved Permanently
2 Location: http://www.foo.bar/
3
4 HTTP/1.1 200 OK
5 Content-Type: text/html
6 Content-Length: 22
7
8 <html>malicious</html>
```

- Precondition: There are two preconditions. The first one is that there is either an unkeyed header or an unkeyed parameter which is reflected in a header of the HTTP response. The second one is a non-compliance with RFC 3986 [18, Section 2.2], which prohibits meta characters in a URL, and RFC 7230 [19, Appendix A.2], which prohibits meta characters in a header.

- Flow [1]: If input from an HTTP request is reflected unsanitized in a header of the HTTP response, HTTP response splitting might be possible. Types of headers which most commonly reflect parts of an HTTP request are the Location header of a redirection response or either the name or value of a cookie. To exploit a vulnerable web cache an attacker sends two requests. The first request tries to manipulate the web server to generate a response that is interpreted as two responses by the web cache. The second request is a common request which targets the web page the attacker wants to poison. The web cache matches the second response of the first malicious request to the innocent second request. Listing 2.7 shows a website which redirects to any website by using the `redirect` parameter. An attacker could now insert two `CRLFs` to generate a second response with arbitrary content. The web cache might not distinguish if that second response was generated by the web server or injected by an attacker. Listing 2.8 illustrates a request which exploits HTTP response splitting. The value of the query parameter `redirect` is URL encoded. The corresponding response can be seen in Listing 2.9. The web server URL decodes the value of `redirect` and writes it, without checking on malicious content, as value

of the `Location` header at lines 2-8. The web cache interprets the 301 response at lines 1-3 as response for the first request and the 200 response at lines 4-8 as response for a second innocent request sent by the attacker. This innocent request might be something simple as `http://example.com/page_to_poison.html`. If users try to access `http://example.com/page_to_poison.html`, they will receive a copy of the 200 response with arbitrary content generated by the attacker.

- Impact: This technique has the highest impact of all these web cache poisoning techniques as an attacker can create an arbitrary response.

7. HTTP Header Oversize (HHO) (The basic explanation of this method has been found in [6] and was enhanced by additional content from related RFCs and header size limits)

- Precondition: There are three preconditions. First, there has to be an arbitrary unkeyed header. Second, there has to be an unharmonized configuration between the web cache and the web server. The web cache needs to have an either slightly lower, equal or arbitrary higher header size limit than the web server. Last, the web cache needs to be non-compliant to the RFC 7231 [20, Section 6.1] which states that only `404 Not Found`, `405 Method Not Allowed`, `410 Gone` and `501 Not Implemented` are allowed to be cached.

- Flow: Most web caches and web servers have a size limit for HTTP request headers. The default limit for the web servers Apache and NGINX is for example 8190 bytes[1,2]. CDNs such as Akamai and Cloudflare on the other hand accept by default requests with a header size of up to 16000 bytes or even 32000 bytes [23][6]. If the size limit is exceeded a "413 Entity too large", as specified by RFC 2616 [22, Section 10.4.14] or falsely a "400 Bad request" answer is returned by the web server. If the web cache has a higher size limit than the web server, the web cache might forward a request which exceeds the size limit of the web server. The web server then returns an error which might get stored by the web cache and served to other users with the same cache key. Current research [7] shows that an attacker could also exploit HHO if the web cache and web server have the same size limit or the size limit of the web cache is even a few bytes lower. This is possible if the web cache normalizes headers or adds new headers, for instance `X-Forwarded-For`.

- Impact: The only impact of this technique is a DOS. A web page cannot be accessed as the web caches returns a cached error response.

[1] `https://httpd.apache.org/docs/current/mod/core.html#limitrequestfieldsize`

[2] `https://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers`

8. HTTP Meta Character (HMC) (The basic explanation of this method has been found in [6] and was enhanced by additional content from related RFCs)

- Precondition: There are two preconditions. First, it is required that the web cache is non-compliant to the RFC 7231 [20, Section 6.1] which states that only `404 Not Found`, `405 Method Not Allowed`, `410 Gone` and `501 Not Implemented` are allowed to be cached. Second, the web cache needs to ignore meta characters in headers, which are stated as illegal in RFC 7230 [19, Appendix A.2], while the web server correctly issues an error response.
- Flow: A potentially harmful meta character such as `\n`, `\r`, `\a` or `\0` can be included in any header. If the cache forwards this request and the web server responds with "400 Bad Request" the response might be stored by the web cache.
- Impact: Similar to HHO, the only impact of this technique is a DOS. A web page cannot be accessed as the web caches returns a cached error response.

9. HTTP Method Override (HMO) (The basic explanation of this method has been found in [6] and was enhanced by additional content from related RFCs)

- Precondition: This technique has two preconditions. First, the web server has to support a header which changes the HTTP Method, such as `X-HTTP-Method-Override`, `X-HTTP-Method` or `X-Method-Override`. Second, this header has to be unkeyed.
- Flow: The headers `X-HTTP-Method-Override`, `X-HTTP-Method` or `X-Method-Override` can override the method of a request. Using `POST`, `PUT`, `DELETE` or something arbitrary as value might result in the server returning "404 Not Found" or "501 Not Implemented" if the method is not supported or "405 Method Not Allowed" if the method is not allowed. All those 3 error responses are cacheable according to the RFC 7231 [20, Section 6.1]. If those headers are unkeyed the same error response might be served from the web cache to other users, leading to a DOS.
- Impact: Similar to HHO and HMC, the only impact of this technique is a DOS. A web page cannot be accessed as the web caches returns a cached error response.

2.3.4 Web Cache Poisoning Countermeasures

To preclude web cache poisoning website operators need to make sure that the preconditions are prevented. As can be seen from the Table 2.2, 4 of the 9 web cache poisoning techniques would not be possible if both the web server and the web cache would fully comply with certain RFCs. However, research shows that web caches [10] and web servers [6] often do not or only partially comply with these RFCs. While vendors improve their RFC

compliance to prevent web cache poisoning vulnerabilities [6] website operators need to wait until these are prevented, decide on a web server and web cache which is compliant to the needed RFCs or configure the web cache to behave correctly. For example if a web cache caches error responses which should not be cached according to the RFC 7231 [20, Section 6.1], such as "400 Bad Request", is cached by default, website operators should configure the cache to not cache such an error response. Also it is important for website operators to make sure that the web server and web cache are harmonized. Otherwise, an attacker might exploit any discrepancies in the configuration, such as different limits for the header size, as shown with technique 7 HTTP Method Override. Table 2.2 also illustrates that 6 out of the 9 techniques can be prevented if every header and parameter with impact is keyed. Further 2 techniques can be prevented if all headers and parameters are keyed. Technique 5 HTTP Request Smuggling is the only technique which does not depend on unkeyed input and therefore is also the only technique which cannot be prevented this way. It is also the only technique which has only one precondition. The only countermeasure for HTTP Request Smuggling is to check and enforce the RFC compliance. Also keying every header has a negative impact on the performance as the cache keys will differ more. For example the `User-Agent` header depends on the used browser, the browser's version and the OS's version. Thus, 10 different users with different user-agents would all have a different cache key. Hence, if each of the 10 users requested the same web page, each of them would receive a response from the web server and not a cached copy. For this reason, website operators need to gauge if keying every header is a good decision. While keying every header and parameter prevents 8 out of the 9 techniques, preventing the four preconditions "Unharmonized configuration", "Non-compliant with RFC", "Header with impact" and "Parameter with impact" defeats every technique and is also better for the performance, which is the primary demand for a web cache.

2.3.5 Web Cache Deception

While web cache poisoning was known since 2004 [1], web cache deception wasn't known until 2017 [24]. It is, besides web cache poisoning, another attack, which is only possible due to web caches. A web cache is tricked into storing a response which shouldn't be cached, such as a web page with personal information or authentication tokens. Afterwards, the attacker can receive the cached copy by issuing a request with an identical cache key and steal the user's information. One way to trick the cache into storing a response is to append a nonexistent css or image file to the end of a URL. For example, `https://www.example.com/sensitiveinformation.php/nonexistent.jpg` might be interpreted as a simple `https://www.example.com/sensitiveinformation.php` by the web server. The web cache instead might interpret it as a jpeg file and, as image files are most often cached, will cache the response. Web cache deception depends in contrast

to web cache poisoning on user interaction, as the user needs to be tricked into using a malicious link. 340 of the Alexa Top 5k websites were tested for web cache deception in 2020 and 10,8% of them were found to be vulnerable.

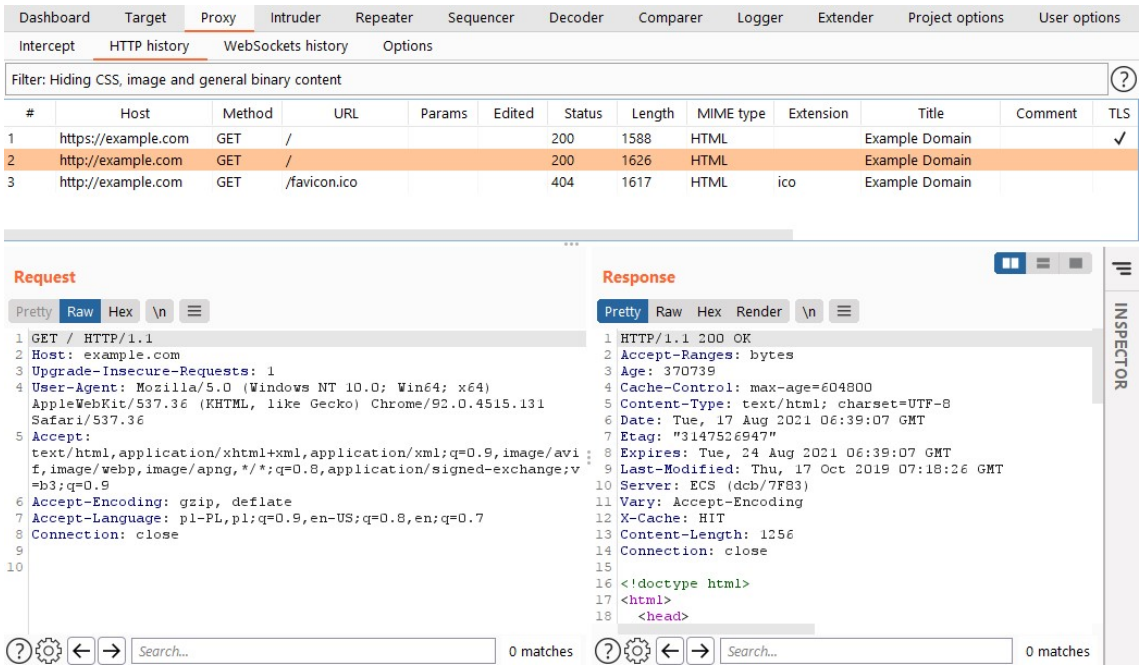

Figure 2.1: The HTTP proxy feature of Burp Suite

2.4 Burp Suite

Burp Suite[3] is a web penetration testing framework which combines many features to test websites and will be mentioned throughout the following 2 chapters. The most important features for manual testing, which will be of importance in Section 4.4, are the HTTP proxy - shown in Listing 2.1 - and the repeater - shown in Listing 2.2. The HTTP history shows every request, which was routed through the Burp Suite proxy, including its response. The history can be sorted, filtered and searched on to look for specific requests or responses. Requests can also be sent from the history to the repeater feature where a request can be sent multiple times. Every part of a request can be easily modified at the repeater for example to change cookies, headers or the path of a request. Furthermore, Burp Suite's features can be extended with extensions. One example of such an extension is "param-miner"[4] which can automatically scan for web cache poisoning and will be described in Section 3.2.

[3] https://portswigger.net/burp
[4] https://github.com/PortSwigger/param-miner

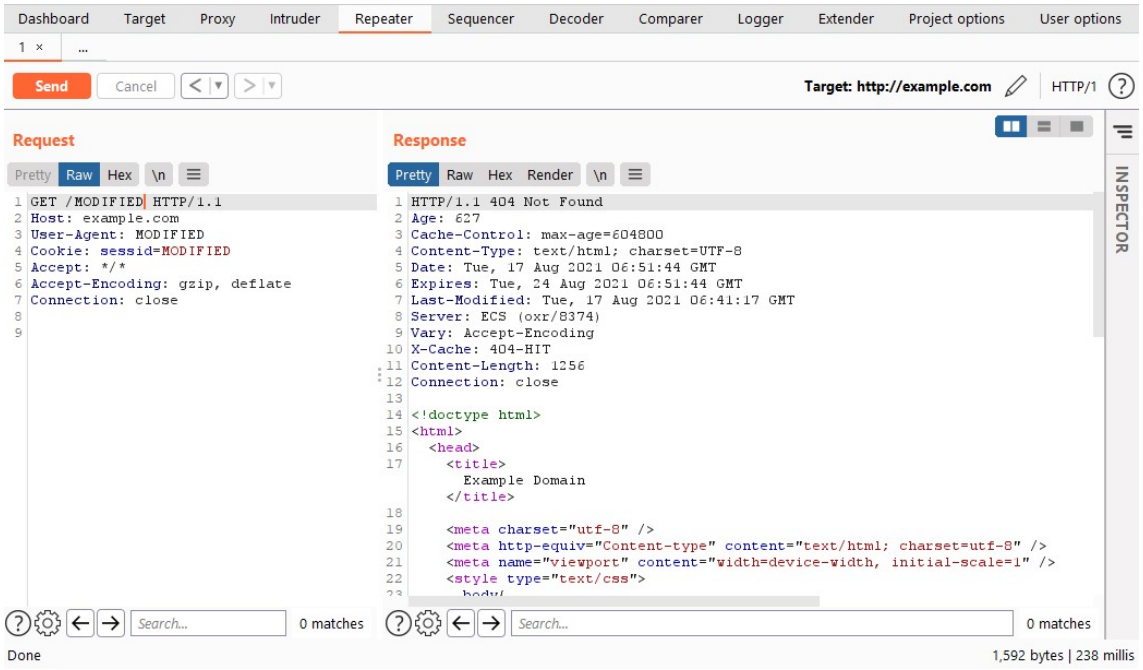

Figure 2.2: The repeater feature of Burp Suite

2.5 Bug Bounties

The following terminology is needed for the following chapters:

Bug bounty: A so-called bug bounty is a reward a company declares for hackers to search for vulnerabilities and disclose them to the company. This reward can either be monetary, material, such as merchandise, or immaterial, such as public acknowledgment.

Bug bounty hunter: A so-called bug bounty hunter is a loose term for anyone searching for vulnerabilities for bug bounties.

Bug bounty program: A bug bounty program basically defines what is legal and allowed to be tested and which reward is declared for this. It states for example which domains and subdomains are allowed to be tested and how these tests are allowed to be conducted. If bug bounty hunters follow these regulations, they cannot be called to account legally.

Bug bounty platform: A bug bounty platform is a website where companies can announce and manage their bug bounty programs. These platforms alleviate these processes by offering templates or to incur the whole management of the program. Bug bounty hunters have a wide collection of programs to choose from when using a bug bounty platform. Due to the templates used for the programs it is easy to glance over the regulations.

Chapter 3

Web Cache Vulnerability Scanner

This chapter presents the Web Cache Vulnerability Scanner (WCVS) that was developed to implement the techniques described in chapter 2. Starting from use cases for different types of users, several existing tools are evaluated to identify the important features and still unmet customer needs. This leads to the list of key features that are the basis for WCVS. The program flow with its main elements is described together with the most relevant implementation concepts. Afterwards, the implementation of each supported web cache poisoning technique is illustrated. Lastly, the setup and usage is described.

3.1 Use Cases

Two main user groups are relevant for WCVS, penetration tester and bug bounty hunter. Although both types of users try to discover vulnerabilities, their use case can be quite different. Table 3.1 summarizes these differences on the basis of a paradigmatic penetration tester and a paradigmatic bug bounty hunter.

The paradigmatic `penetration tester` is a representative for all users that are responsible for a special website, e.g., website programmers, administrators or similar persons responsible for this site. This relevant website will be tested for many different

Characteristics	Penetration tester	Bug bounty hunter
Websites to test	1	Countless
Crawling	Thorough	Minimal
Timelimit	Few days	None
Throttle requests	No	Yes
Use of a proxy	Yes	No
Degree of automation	Some	High

Table 3.1: Differences between a paradigmatic penetration tester and a paradigmatic bug bounty hunter.

Scanner	No installation	CLI	Report	Techniques	Crawler	Rate limiting
WCVS	x	x	JSON file	9	x	x
Param-miner			GUI	6	extra plugin	extra plugin
Fockcache	x	x	Terminal	1		
web_cache_poison.sh	x	x	Terminal	1		
Nuclei	x	x	Terminal	1		

Table 3.2: Comparison of web cache poisoning scanners.

vulnerabilities to secure the overall security. The tests are thorough to ensure that the website is not vulnerable to known vulnerabilities. However, the time to conduct tests is quite limited and averages out at a few days. Most likely the tests are conducted on a non-productive system, which does not have a web application firewall enabled. Thus, automated scans can be run at full speed to maximize results in the limited time. The traffic of scanners will be, if possible, routed through a proxy, such as Burp Suite. Thereby, the HTTP responses can be easily analyzed further if something striking was noticed. There is some degree of automation, as tools are started automatically to speed up the testing process. Though, each positive finding has to be validated conscientiously. The penetration tester might create a custom wordlist tailored for the relevant website which can be used by scanners.

The paradigmatic `bug bounty hunter` on the other hand is not interested in a special website but to find something in a very large number of sites. Bug bounty programs typically provide thousands of websites, in terms of multiple domains with wildcard subdomains, to test. The bug bounty hunter specializes in a few vulnerability types which will be inspected for as many websites as possible. Thus, they do not test the entire website, but rather up to a few different pages for each website. As the tests are almost every time on a productive system, a web application firewall is most likely in use. This web application firewall might block IP addresses if too many requests per second are issued from them. So the bug bounty hunter has to assure that they do not send too many requests in a specific time frame. Another reason to throttle requests are the regulations of the corresponding bug bounty program, as they often define an allowed maximum of requests per second. The bug bounty hunter does not need a proxy in between, as they do not intend to fully analyze a specific website, but rather to cover a large amount of websites. A high degree of automation is used in order to start multiple scanners and analyze their results.

3.2 Contestants

There are already several web cache poisoning scanners available. An overview of the Web Cache Vulnerability Scanner and its contestants is illustrated in the Table 3.2.

Detectify[1], Appcheck-ng[2] and Acunetix[3] are comprehensive website security solutions which claim to (also) scan for web cache poisoning. All three are on-premise security solutions for companies and aim to replace or supplement penetration tests. As they are on-premise solutions they are not eligible for penetration testers and bug bounty hunters. This ranges over the licensing models and the inability to scan many different domains.

Fockcache[4] and web_cache_poison.sh[5] are both open source web cache poisoning scanners. Fockcache tests only the two X-Forwarded-Host and X-Forwarded-Scheme headers for web cache poisoning and therefore misses most relevant attacks. web_cache_poison.sh uses the wordlist[6] of the shortly mentioned param-miner which contains more than 1000 headers. It tests if any value of those headers is reflected in the body of the HTTP response. Both scanners have only a bare minimum functionality and have not been maintained since almost two years.

Nuclei[7] is an open source community powered vulnerability scanner. It offers the possibility to use community created templates or create user specific templates. There are two templates available which test for web cache poisoning[8,9]. However, both test only for a couple of headers, similar to fockcache. Nuclei is well suited to run many different templates against one or multiple targets. Nonetheless some web cache poisoning techniques (like A or B) cannot be transformed into such a template, as the available functions within templates are quite limited.

Param-miner[10] is an open source extension for Burp Suite. It is maintained by James Kettle, who identified many of the web cache poisoning techniques in Section 2.3 in his papers and blog posts.. Param-miner's main purpose is to identify headers and parameters supported by a website. It is also able to find many of the in Section 2.3 described web cache poisoning vulnerabilities. While param-miner is currently the best alternative to WCVS, its biggest downside is that it is only usable through Burp Suite. For this reason it is not possible to use it in an automated way, such as starting scans and processing the findings. Furthermore, it may require some effort to get used to Burp Suite. For functionalities like a crawler or rate limiter other add ons need to be used. Besides that, Param-miner does not support the following techniques: HHO, HMO, HMC, HTTP Request Smuggling and HTTP Response Splitting.

[1] https://detectify.com/
[2] https://appcheck-ng.com/
[3] https://www.acunetix.com/
[4] https://github.com/tismayil/fockcache
[5] https://github.com/fngoo/web_cache_poison
[6] https://github.com/PortSwigger/param-miner/blob/master/resources/headers
[7] https://github.com/projectdiscovery/nuclei
[8] https://blog.melbadry9.xyz/fuzzing/nuclei-cache-poisoning
[9] https://gitbook.seguranca-informatica.pt/fuzzing-and-web/cache-poisoning-using-nuclei
[10] https://github.com/PortSwigger/param-miner

3.3 Key Features

As there was no completely satisfying web cache poisoning scanner for penetration testers and bug bounty hunters available yet, as shown in table 3.2, the Web Cache Vulnerability Scanner (WCVS) was developed. The following key features were in mind:

- Standalone CLI tool: A standalone CLI tool is easy to set up and can easily be executed by scripts or other automation frameworks. WCVS is provided as a standalone binary file to satisfy these benefits.

- Transparency: Transparency is important to gain insight into the scanner's actions and into its deliverables. The WCVS produces four different outputs. It writes different kinds of information to the console, such as:

  - Program flow: What is the scanner currently doing? Which techniques is it executing?
  - Errors: Errors get catched and a verbose error message is created.
  - Oddities: If any response has oddities they are reported.
  - Cache behaviour: The analyzed cache behaviour is reported.
  - Vulnerabilities: What vulnerability could be found? Which technique was used? How was it confirmed as a vulnerability? Which URL was affected?

  The same and a few more pieces of information are also written to the log file. A JSON report file is created which contains this information and a little more, such as which settings were used to start the scanner. Also a list with completed URLs is created, which can be used to resume a test at a specific point by skipping every URL in this list. Thus, the report, log and list with tested URLs can be further processed in an automated fashion.

- Crawler: A crawler collects URLs from HTTP responses. This helps to identify and scan multiple - if not all - URLs of a domain. WCVS is able to collect all URLs from responses, add them to a queue and scan them before continuing with the regular URL queue. By default the crawler only adds URLs with the same domain as the response derives from to the queue. The scanner can be configured to also add URLs from other specified domains. An allowlist with strings, which have to be included in the URL, can also be provided for example to only add certain file endings such as ".css" and ".js". Furthermore, a limit can be specified how deep the crawler shall recursively crawl and how many URLs shall be added on each recursivity level.

- Web application firewall compatibility: When testing a productive system, a web application firewall might block an IP address if it is spotting unusual behaviour. An

allowed maximum of requests per second can be specified so that WCVS does not get blocked by web application firewalls for sending too many requests in a certain amount of time. The user-agent can also be set to an up-to-date chrome browser in case that a web application firewall does not allow uncommon user-agents.

- Ease of use: Security is important for everyone and not only for experts. Thus, it is also important that a scanner is easy to use and does not scare off potentially interested users by requiring complex preliminaries. The more tests for web cache poisoning are done, the harder it will be for vulnerabilities to remain undetected. Therefore, WCVS offers a default setup that only requires a single command line flag. This flag specifies a single URL or a file containing at least one URL. Every other command line flag is optional. The default settings of WCVS were chosen in a way that no or only minimal further configuration should be needed for most use cases. Appendix A.1 shows the help output of the scanner which explains all 34 possible command-line flags.

- Customizability: Customizability ensures that a tool can satisfy individual requirements and that it can be optimized for varying conditions. There are a total of 34 command line flags which give the possibility to tweak the scanner to one's need. For example, if on the one hand a strict web application firewall is in place the requests per second can be limited. On the other hand, the performance, and thereby also the requests per seconds, can be increased in case that no web application firewall or other limiting conditions are present.

- State of the art tests: A scanner should utilize thorough testing techniques in order to be able to conduct meaningful tests. WCVS supports nine different web cache poisoning techniques, which are based on current whitepapers and blog posts from IT security professionals.

3.4 Program Flow

The program flow can be simplified into 3 nested loops, as illustrated in Figure 3.1.

1. The inner loop from line 5 to line 10 is sending and analyzing HTTP requests in order to check for web cache poisoning. The loop runs for every variant of a web cache poisoning technique. Variants are small alterations of a technique. The following four methods are called sequentially in order to test a variant of a technique.
   `attackerRequest()` is the request which attempts to poison the web cache. It issues a modified request to the given url and returns specific values, such as the response's body, status code and headers needed later on to check for successful poisoning indicators. It can take optional parameters to modify the request in order

Listing 3.1: Program flow of WCVS in pseudeocode

```
 1 while URL_Queue not empty {
 2   dequeueFirst()
 3   analyzeFirst()
 4   for each technique {
 5     for each variant {
 6       attackerRequest()
 7       victimRequest()
 8       checkPoisoningIndicators()
 9       testResponseSplitting()
10     }
11   }
12   crawlAndEnqueueURLs()
13 }
```

to test for the different web cache poisoning techniques. Most techniques only need one of the following optional modifications:

- change the value of existing headers or add additional headers with values
- change the value of existing cookies or add additional cookies with values
- change the value of existing parameters or add additional parameters with values
- change the value of request's body
- change the HTTP method

`victimRequest()` is a common request which checks if the prior request got cached. It uses the same cachebuster as `attackerRequest()` in order to generate the same cache key but does not have any further modifications. Also, it returns the same kind of variables as `attackerRequest()`, which will be analyzed by checkPoisoningIndicators().
`checkPoisoningIndicators()` checks if the variant of the corresponding web cache poisoning technique was successful. The returned values from `attackerRequest()` and `victimRequest()` are checked for the four successful poisoning indicators, described in Section 3.7. If the result of any of these checks is true, the reason why the web cache was found to be vulnerable is saved and written to the console. If a report shall be generated, this is also written to the report file. Lastly, the method returns a boolean which indicates if it was possible to inject content into a header of the response and is needed for the following step.
`testResponseSplitting()` tests if HTTP Response Splitting is possible. It is

Listing 3.2: Inner loop for HTTP Request Smuggling in pseudeocode

```
5 for each variant {
6   for x = 0 to 3 {
7     attackerRequest()
8   }
9 }
```

only executed if the boolean returned by `checkPoisoningIndicators()` is true, because injecting content into a header of the response is a precondition for HTTP Response Splitting. HTTP Response Splitting has the highest impact and therefore would always increase the impact of the tested variant. All of the web cache poisoning techniques are implemented this way, besides HTTP Request Smuggling. The inner loop of the HTTP Request Smuggling technique is illustrated in Figure 3.2. The attack variants of this technique try to trigger a timeout in order to confirm HTTP Request Smuggling. Further information on how this works is provided in Section 3.8.

After `attackerRequest()` with the parameters of the corresponding variant has finished executing, `checkRequestSmugglingIndicators()` is executed. This method checks whether a timeout occurred. This is repeated three times, to lower the likelihood of accidental timeouts. If all three requests time out the variant is considered as most likely successful. Similar to `checkPoisoningIndicators()` the finding with the reason of success is written to the console and - if configured - to the report.

2. The middle for loop is from line 4 to 11 and very simple. Every technique, which the scanner supports, is executed one after another. By default every technique is executed, but the scanner can also be configured to only execute specific techniques. Because of the inner loop, every variant of a technique is executed, before continuing with the next technique.

3. The outer loop from line 1 to line 13 ensures that the web cache poisoning techniques are tested for every eligible URL. It is supplied with URLs to be tested by the URL_Queue. The initial URL_Queue consists of all URLs specified via the -u/-url command line flag.

   `dequeueFirst()` takes the topmost URL of the URL_Queue.

   `analyzeCache()` analyzes the caching behaviour of the URL, in particular if a hit or miss indicator and a cachebuster can be found. More details on hit or miss indicators and cachebusters is given in the following two subchapters. If no cachebuster can be found, the URL will not be tested, as a cachebuster is a necessary condition

to test for web cache poisoning.

Otherwise, the middle loop is run. `crawlAndEnqueueURLs()` is executed after all desired techniques are tested. It crawls the source code of the current URL for additional URLs which are neither in the queue of this loop nor have been already tested. The newfound URLs are then added to the top of the URL queue. By this technique all crawled URLs are tested before continuing with the next URL from the -u/-url command line flag.

3.5 Hit or Miss Indicatros

Web caches generally add headers to HTTP responses which indicate whether a cached or uncached response has been sent. Most of those headers contain the value `hit` if a cached response and `miss` if an uncached response has been sent. As a preparation for the planned tests the responses of a few websites, which will be tested in the following chapter, were analyzed and the following headers which use `hit` or `miss` as an indication were identified: `X-Cache`, `Cf-Cache-Status`, `X-Drupal-Cache`, `X-Varnish-Cache`, `Akamai-Cache-Status`, `X-NC`, `Server-Timing`, `X-Hs-Cf-Cache-Status`, `X-Proxy-Cache`. Two other headers which can be used as indicators to determine whether a response was cached or uncached are the `X-Proxy-Cache` and `Age` headers. The header `X-Proxy-Cache` states how often the cached resource was requested. A value of 0 indicates that the response has not been requested before. Therefore, the response was not cached and was issued by the web server directly. A number greater than `0` indicates that a cached response was served. The `Age` header indicates for how long a resource was already cached. If its value is greater than `0` the response was cached. This indicates a `hit`. The value `0` indicates a `miss`.

Web caches can also be configured to not add these headers. If that is the case, as a last effort, the time until the response is received can be measured. This allows it to differentiate between cached and uncached responses, because cached responses tend to be served faster than uncached responses. However, this approach is prone to both false positives and false negatives. For this reason specific statistics are gathered in order to check the reliability of the time measurement as an indicator. More specifics on how WCVS implements the time measurement is given in the next section. WCVS utilizes all these mentioned hit or miss indicators to not miss a valid target.

3.6 Cachebusters

A cachebuster is any arbitrary part of a request which influences the cache key. Hence, when a header, parameter or cookie is keyed and used with a random value it should result in a cache miss. This is important to test for web cache poisoning as the first request which tries to poison the cache has to receive a cache miss. Otherwise the response of another

request is returned. At first when testing for cachebusters, two identical requests are sent. If both requests receive a miss the cache seems to be configured not to cache and no further tests will be conducted for this URL. Elsewise, the scanner tests for different cachebusters in the following order:

1. Query parameters: A query parameter, whose default name is `cb`, is used with a random value and appended to the URL.

2. Cookie values: Every cookie value is, one after another, replaced with a random value.

3. Header values: The values of the following headers are, one after another, appended or replaced with a random value: `Origin`, `Accept`, `Accept-Encoding`, `Cookies`.

4. HTTP methods: `PURGE` and `FASTLYPURGE` are used separately as HTTP method. Both HTTP methods are used to clear the cache and therefore force a cache miss for the request

If a hit or miss indicator was found in the previous step the tests are simple. However, the tests for the fourth kind of cachebusters differs a bit from the first three kinds. In the following the tests for query parameters, cookie values and header values are illustrated. Lastly, the differences of the tests for HTTP methods are described. For each of the cachebusters two requests are sent with different random values. If both requests receive a `miss` the cachebuster is regarded as suitable and will be used for the tests. In case that a `hit` is received the cachebuster is viewed as ineffective, as the likelihood of not having an unique cache key, when using a 12 character long random value as cachebuster, is minimal.

If no hit or miss indicator could be found in the previous step, the timed approach will be used. For every potential cachebuster 10 requests are sent. The requests 1,3,5,7,9 generate a new random value for the cachebuster, while the requests 2,4,6,8,10 use the same value as the previous request. If a cachebuster is working, every odd-numbered request should be a `miss` and every even-numbered request a `hit`. The time for each response is measured. If every odd-numbered request is faster than the following even-numbered request, it can be assumed that the cachebuster is working. This method is not as reliable as the other hit or miss indicators because the time could be influenced by various factors other than if the response was cached or not. Therefore, 10 requests are made to increase the significance of the time measurements. In addition, a default threshold was set. Only if the odd-numbered responses were at least 30ms faster than the even-numbered ones the time measurements were used as a valid hit or miss indicator. This threshold appeared as appropriate during test runs.

Afterwards the fourth cachebuster method is tested. The process is similar to the other three cachebuster methods. Instead of a random value, the HTTP method is set to `PURGE`

and instead of using the same value the HTTP method is resetted to its initial value. The same process is repeated with `FASTLYPURGE`.

3.7 Successful Poisoning Indicators

WCVS uses four different indicators for successful web cache poisoning attacks.

1. Reflection in the body: The poison value included in the HTTP request is reflected in the body of the HTTP response. This could lead to, among other consequences, stored XSS or defacement of the website.

2. Reflection in a header: The poison value included in the HTTP request is reflected in a HTTP response header. This could be for instance the cookie header, possibly leading to session fixation, or the Location header leading to an open redirect.

3. Status code: The status codes of the attacker and victim responses are not identical to the initial response. For instance, the initial status code is `200 OK`, but for the attacker and victim responses the status code is `302 Found` thus indicating an open redirect.

4. Content-Length difference: The value of the `Content-Length` header of the attacker and victim responses are not identical to the respective value of the initial response. For example, the request shall contain the `User-Agent` header of an outdated Internet Explorer browser. The target site might send a `200 OK` response, similar to the initial response. However, its content is not the expected website, but a notification that the browser is not supported and shall be updated. The poison value is not reflected in the header or the body and the status code is the same. However, the `Content-Length` is different, as the initial response has the `Content-Length` of the expected webpage and the attacker and victim responses have the `Content-Length` of the "Update your browser"-webpage. This would be considered a DOS attack as other users trying to access the same page would receive the same notification page.

For all four poisoning indicators false positives are possible. While false positives are improbable for the first and second indicator, the third and fourth indicators are more prone to false positives.

1. Reflection in the body: WCVS uses random 12 digit numbers as poison value. While it is unlikely that this random number was already present in the response's body it is not impossible.

2. Reflection in the header: Just like the first indicator the reliability of this indicator depends on the uniqueness of the randomly generated poison value.

3. Status code: The status code could also differ, when an error, such as a time out occurs, for example. This may happen in case a gateway, proxy or web server is temporarily overloaded.

4. Content-Length difference: The `Content-Length` can be different for multiple reasons, for example when a file is updated. While it is important that the threshold for the `Content-Length` difference is not set too low as this might increase the amount of false positives, it is also important that the threshold is not set too high as the scanner might miss a web cache poisoning vulnerability. The `Content-Length` indicator is disabled as default, because of its unreliability. However, a command-line flag can be used to specify a threshold for the `Content-Length` difference. The optimal threshold depends on the tested website.

3.8 Implementation of Web Cache Poisoning Techniques

Given that the development of WCVS was started a few months prior to this thesis, techniques 1 to 4 - namely unkeyed header poisoning, unkeyed parameter poisoning, parameter cloaking and fat GET - were already implemented. During this thesis the techniques 5 to 9 - namely HTTP request smuggling, HTTP response splitting, HTTP header oversize, HTTP meta character and HTTP method override - were developed to further complete the list of web cache poisoning techniques WCVS covers.

1. Unkeyed Header Poisoning For unkeyed header poisoning all cookies which are set by the website are tested at first. Every cookie represents a variant. For each variant the corresponding cookie value will be changed to a different random poison value. Then, a few headers are tested with preset values. For example, to test if the port can be changed to achieve a DOS ":31337" is appended to the `Host` header and in another request the "X-Forwarded-Port: 31337" header is added. Each of these examples represents a variant of header poisoning. Afterwards, a wordlist with header names is used. Each header name of this wordlist represents a variant. The only parameter which is handed over to the firstRequest() method is the header name of the corresponding variant and a random poison value. If the default request already contained a header with this name, such as `Origin`, its value is overwritten. Otherwise a new header with the passed name and value is added.

2. Unkeyed Parameter Poisoning Similar to the last part of unkeyed header poisoning a wordlist with parameter names is used. Every parameter name which can be extracted from the given URL and every parameter name from the wordlist represents a variant. If the parameter name of the variant is already present in the URL its value is overwritten with a random poison value. Otherwise a new parameter with the parameter

name and a random poison value is appended. Parameters which alter the response but are keyed, cannot be utilized for this technique. However, they can be useful for the fat GET and parameter cloaking techniques, where the fact that the parameter is keyed can be circumvented by exploiting an unharmonized configuration of the web cache and web server. Therefore, those parameters will be added to a list which can be utilized for the fat GET or parameter cloaking techniques.

3. Parameter Cloaking Parameter cloaking uses the parameter list which was created by the unkeyed parameter poisoning technique, as well as a short list of well-known parameters. At first, it will test if any of the following well-known parameters are unkeyed: `utm_source`, `utm_medium`, `utm_campaign`, `utm_content` and `utm_term`. This is often the case for marketing and tracking purposes. If none of them is unkeyed the test for parameter cloaking is terminated. Afterwards, each parameter of the first mentioned list is tested one after another. The current parameter and a poison value is appended with a `;` to every of the unkeyed utm parameters. Thus, creating for example a query string like `?keyed_parameter=foo&utm_parameter;keyed_parameter=poison_value`. Each of the created query parameters represent a variant of this technique.

4. Fat GET Just like parameter cloaking, the list which was created during the unkeyed parameter poisoning technique is used. Three different variants of fat GET are evaluated with every parameter of this list:

1. A GET request with the current parameter and a random poison value in the body.
2. Three GET requests with the current parameter and a random poison value in the body and each with one of the following three headers set: `X-HTTP-Method-Override: POST`,`X-HTTP-Method: POST` and `X-Method-Override: POST`.
3. A POST request with the current parameter and a random poison value in the body.

5. HTTP Request Smuggling Four different HTTP request smuggling variants are used which were described in the paper `HTTP Desync Attacks: Request Smuggling Reborn` [25]. While other variants require two requests to be sent and that no other requests were received between them, those four variants only require one request and are therefore more reliable. The goal is to provoke a timeout by making the web cache not forward the complete request while the web server is waiting for the remaining part of the request. Listing 3.3 illustrates one of these variants. When the web cache utilizes the `Transfer-Encoding` header, while the web server utilizes the `Content-Length` header a timeout will arise. The web cache will only forward the blue part of the request as the `0` indicates that the body contains no further content and that the request is finished

Listing 3.3: HTTP Request Smuggling variant

```
1 POST /about HTTP/1.1[CRLF]
2 Host: example.com[CRLF]
3 Transfer-Encoding: chunked[CRLF]
4 Content-Length: 6[CRLF]
5 [CRLF]
6 0[CRLF]
7 [CRLF]
8 X
```

at this point. The web server on the other hand waits for a body with a length of `6` while the web cache only forwarded a body with the length of `1`. The result is either a timeout, if the web server keeps waiting, or an error response, such as `500 Internal Server Error`, if the web server stops waiting. Both indicate that the combination of web cache and web server is vulnerable to HTTP request smuggling. If no further servers than the web cache proxy server and the web server are involved it is sufficient to only send the blue part of the request. However, if another server is located before the web cache and passes through the request to the web cache it is important that the blue part of the request is also present. If the server in front of the web cache obeys RFC 2616 [22, Section 4.4] and therefore utilizes the `Content-Length` header instead of the `Transfer-Encoding` header, a timeout would occur if the blue part of the request is missing. This would then result in a false positive since a timeout is considered as success for this technique variant.

6. HTTP Response Splitting The precondition of HTTP response splitting is that a part of the request is reflected unencoded in a header of the response. If any of the techniques of WCVS results in the poison value being reflected in a header of the response, another similar request with a different cachebuster value is issued. This time `\r\nWeb_Cache: Vulnerability_Scanner` is appended to the poison value. If the backslashes are not encoded by the web server, the resulting response will contain the new header `Web_Cache: Vulnerability_Scanner`. In that case the HTTP response splitting attack was successful.

7. HTTP Header Oversize (HHO) HHO has three variants, each testing for a different size limit by adding multiple headers with a size of 80 bytes each. The different variants add headers with a size of in total 4000, 8000 and 16000 bytes to the request.

The different values for the total size are selected by the header size limit of common web servers, such as Apache[11] and NGINX[12].

8. HTTP Meta Character (HMC) HMC has 10 variants where each adds one of the following meta characters to a header: `\n`, `\r`, `\a`, `\0`, `\b`, `\e`, `\v`, `\f`, `\u0000`. These characters are forbidden in HTTP headers according to RFC 7230 [19, Appendix A.2] as already mentioned in Section 2.3.3.8.

9. HTTP Method Override (HMO) HMO has a total of 12 variants. The three headers `X-HTTP-Method-Override`, `X-HTTP-Method` and `X-Method-Override` are each combined with the four different methods `GET`, `POST`, `DELETE` and `NONSENSE`.

3.9 Setup and Usage

WCVS can be executed without installation as it was developed in Go. Standalone binaries for windows/amd64, linux/amd64 and darwin/amd64 can be downloaded from the releases page[13] of the github repository. The release page also provides a header and a parameter wordlist, which are recommended as suitable default values unless scenario specific wordlists are needed. A header wordlist is required for the technique "unkeyed header poisoning". A parameter wordlist is required for the techniques "unkeyed parameter poisoning", "fat GET" and "parameter cloaking".

Building from Sourcecode It is possible to build a binary for other operating systems by using a Go installation. The source code of the web cache vulnerability scanner can be downloaded from Github[14], as it is open source under Apache 2.0. Using the command "go build web-cache-vulnerability-scanner.go" a binary for the current operating system will be built. It is also possible to build binaries for other operating systems[15] by setting the GOOS and GOARCH environment variables.

Execution The user executing the binary needs three permissions. The first permission is to execute the binary, the second to read the files which the scanner may need, such as the two wordlists, and the third to write to the directory the scanner writes output to - which is the directory the scanner was started from as default. The scanner can be invoked

[11] https://httpd.apache.org/docs/current/mod/core.html#limitrequestfieldsize

[12] https://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers

[13] https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner/releases

[14] https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner/archive/refs/heads/master.zip

[15] https://www.digitalocean.com/community/tutorials/building-go-applications-for-different-operating-systems-and-architectures

by command-line. The only command-line flag which is required is the -url (short: -u) flag. This flag indicates which URLs shall be tested. Its value can either be a single URL or a file with multiple URLs. There are 33 other command-line flags which are all optional and can be used to customize and tweak the scanner. Appendix 1 shows the help output of the scanner which explains all 34 possible command-line flags. The 11 command-line flags which were used during the test and will be referred to in the following chapter "Test Approach" are the following:

1. **url (short: u) [string]** As already mentioned, this flag is the only mandatory one. It takes a string as value. This string can either be a single URL or the path to a file which contains multiple URLs. In case that a path to a file is used, the string "file:" has to be added as a prefix. For example: "file: C:/path/to/urls.txt".

2. **generatecompleted (short: gc)** A file, which contains every URL for which testing has been completed. The file is updated each time a new URL has been tested. The file can be used to continue a test at the same point in case the scanner crashed or had to be terminated for whatever reason.

3. **generatereport (short: gr)** A report will be generated and updated after a URL was completely tested. The report is in JSON format and contains - among other things - the following information:

   - Which settings were used
   - Which errors occurred
   - Which vulnerabilities were found including the request, the response and the reason why it was flagged as vulnerability

4. **generatepath (short: gp) [string]** This path will be used to write all files to. This takes a string which should represent a valid directory path, for example "C:/path/to/directory". A log file is always created and contains more verbose information than the console output. The report and completed-URL file will also be written to this path, if their flag was used.

5. **recursivity (short: r) [integer]** This flag defines how deep the crawler shall crawl recursively. It takes an integer as value. A value of "0" means that the URLs should not be crawled.

6. **reclimit (short: rl) [integer]** This flag limits how many URLs shall be added by the crawler for each recursivity depth. It takes an integer as value. A value of "0" means that there is no limit.

7. **headerwordlist (short: hw) [string]** This flag defines the path to the header wordlist which will be used for the "unkeyed header poisoning" technique. It takes a string with a valid path to a file, such as "C:/path/to/wordlist.txt".

8. **parameterwordlist (short: pw) [string]** This flag defines the path to the parameter wordlist which will be used for the "unkeyed parameter poisoning", "Fat GET" and "parameter cloaking" techniques. It takes a string with a valid path to a file, such as "C:/path/to/wordlist.txt".

9. **useragentchrome (short: uac)** The default user-agent of WCVS is "WebCacheVulnerabilityScanner v" followed by the version number. Some web applications firewalls may block requests which do not use a browser as the user-agent. Therefore, this flag changes the user-agent to the user-agent of a Google Chrome browser.

10. **reqrate (short: rr) [float]** This flag limits the maximum allowed amount of requests per second. It takes a float as value. For example, a value of "1" would mean that no more than one request will be sent every second, while a value of "0.5" would mean that no more than one request will be sent per every two seconds. While some bug bounty programs require to throttle the amount of requests per second, the throttling is also important to not get blocked by some web application firewalls or to not overload a busy or weak server.

11. **contentlengthdifference (short: cldiff) [int]** This flag enables the scanner to use the content-length differences as successful web cache poisoning indicators. As this indicator is the one most prone to false positives, it is disabled by default. The flag takes an integer as value. A value of "0" means that the content-length difference shall be ignored, while an integer greater than "0" defines the threshold for how many bytes the content-length may differ.

Chapter 4

Testing for Web Cache Poisoning

This chapter deals with the conducted tests for web cache poisoning. First, the selection of eligible websites is described. Second, the approach of the tests using the Web Cache Vulnerability Scanner is explained. Afterwards, the results of the tests are presented. That includes - among others - the identified hit or miss indicators, cachebusters, web cache poisoning vulnerabilities and false positives. Lastly, two countermeasures which reduce the number of false positives are described.

4.1 Selection of Websites

As it can be legally troublesome to test foreign sites for vulnerabilities only websites with a bug bounty program were chosen. The procedure for this selection was the following. Bug bounty platforms were used to collect websites that allow it to be tested for vulnerabilities. These platforms have a large collection of testable websites and provide legal safety. In particular Hackerone[1], Bugcrowd[2], Intigriti[3] and YesWeHack[4] were used as they are the most popular bug bounty platforms based in Europe or the USA. OpenBugBounty[5] - another popular bug bounty platform - was not used because they prohibit the use of automated scanners entirely. Using the four platforms mentioned before more than 1000 suitable websites were collected.
Afterwards, the Tranco list[6] (generated on 01 June 2021) was used to sort the websites depending on their rank on the tranco list. The Tranco list is a ranking of the most popular websites which applies specific countermeasures against manipulation [26]. The selection of the websites was then shrinked, as it was not possible to read through over 1000 bug bounty programmes, each containing a minimum of 4 pages, in the limited time of this thesis. Only

[1] https://www.hackerone.com/
[2] https://www.bugcrowd.com/
[3] https://www.intigriti.com/
[4] https://www.yeswehack.com/
[5] https://www.openbugbounty.org/
[6] https://tranco-list.eu/list/QXW4

websites within the top 1000 of the Tranco list were considered for further inspection. The

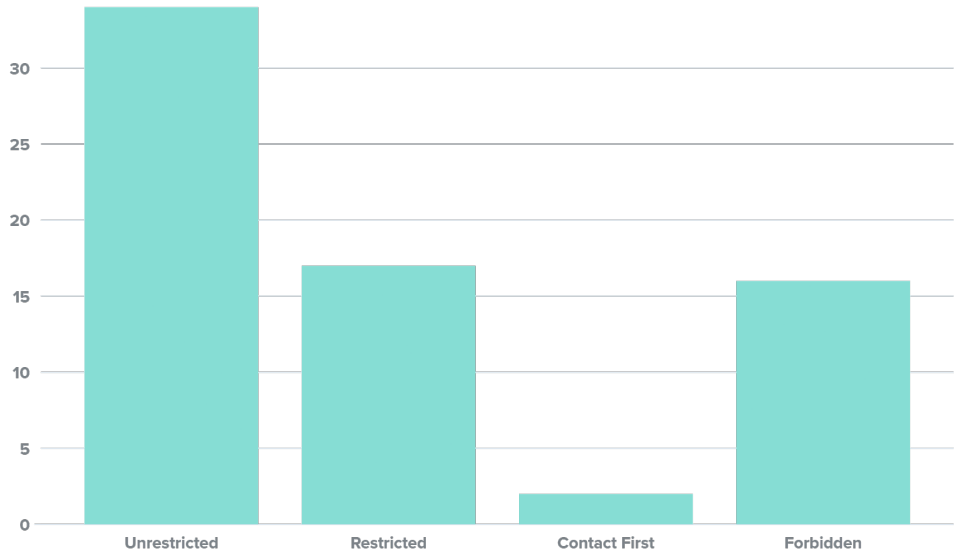

Figure 4.1: Bug bounty program rules regarding the use of automated scanners

bug bounty programs of the remaining 69 websites contained different regulations regarding the use of automated scanners. Listing 4.1 illustrates these differences. While 34 bug bounty programs allowed the use of automated scanners, 16 strictly disallowed their use. 17 bug bounty programs contained limitations or requirements for automated scanners. Some of those were:

- The User-Agent header has to include a specific string.
- A specific header has to be set.
- Only a certain amount of requests per second is allowed, e. g., one request per second.
- Commercial scanners are disallowed, while noncommercial scanners are allowed.

WCVS has options to set custom headers and to throttle the requests per second. Also it is not a commercial scanner. Therefore, all these requirements can be fulfilled. The two remaining bug bounty programs required to contact their customer service before conducting any tests. Both customer services were contacted, but no response was received within two months prior to the beginning of the tests. Therefore, those two domains were excluded from the tests. In the end 51 domains were found eligible for the tests. While 11 bug bounty programs listed all subdomains of the website which are allowed to be tested, the other 42 allowed the testing of any subdomain one can find. For these 42 bug bounty programs, the open-source tool subfinder was used to generate a list of subdomains of the website. Subfinder gathers subdomains for a given domain by utilizing multiple

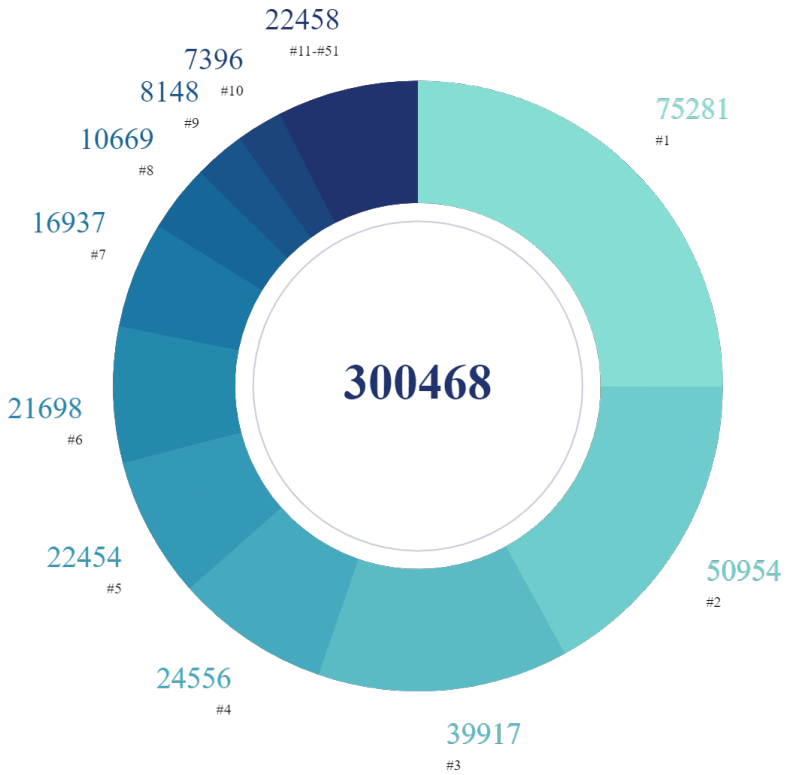

Figure 4.2: Amount of discovered subdomains per subdomain

search engines and data of services which collect subdomains of websites. Listing 4.2 illustrates the amount of found subdomains split amongst the ten websites with the most subdomains and the rest. In total 300468 subdomains were found and around 92,5% belong to the ten websites with the most subdomains. As subfinder uses passive methods to gather subdomains, the data gathered from search engines and other databases might be outdated and some of these subdomains are no longer available.

4.2 Test Approach

For each of the 51 websites a folder was created which contains a file called “domains” with a list of all given or passively found subdomains. Then a script was used to start an instance of WCVS for each list of subdomains. The following 11 flags were used for all scanners:

1. **url file:C:/path/to/domain/subdomains.txt** This is the path to the file which contains all the found subdomains of the corresponding domain.

2. **generatecompleted** This was done as a precaution to be able to continue a scan, if a scanner would crash or be terminated for whatever reason.

3. **generatereport** The report is important for the analysis as it contains all findings, errors and anomalies, which occurred during the scans.

4. **generatepath C:/path/to/domain/** The list with completed URLs, the report and the log file shall be written to this path.

5. **recursivity 5** The crawler was set to a recursivity depth of 5 in order to test multiple web pages of a subdomain, if they are linked in the response.

6. **reclimit 5** A maximum of 5 URLs will be crawled for each depth. This combined with the recursivity depth of 5 results in a maximum of 25 additional URLs which will be tested for each subdomain.

7. **headerwordlist C:/path/wordlists/headers** The used wordlist contains 1118 headers and is a purified version of the param-miner header wordlist[7] as duplicates were removed.

8. **parameterwordlist C:/path/wordlists/parameters** The used wordlist contains 6453 parameters and is a purified version of the param-miner parameter wordlist[8] as duplicates were removed

9. **useragentchrome** The user-agent was set to an up-to-date Google Chrome browser in order to have less problems with web application firewalls, as they might block unusual user-agents.

10. **reqrate 1** The scanner was throttled to send no more than one request per second. This was done because some bug bounty programs required this limitation and also because first tests showed that the web application firewalls of programs without limitations were blocking the IP address if too many requests per second were sent. Hence the limit was set to such a low amount.

11. **contentlengthdifference 2000** A deviation of the content-length on the one hand may indicate that the used web cache poisoning technique triggered a different response. On the other hand it may just be an intended or random change of the content-length. As this technique is highly vulnerable to false positives a content-length difference of 2000 bytes was chosen.

Five bug bounty programs required the use of a specific header in order to identify the traffic generated by bug bounty hunters. Thus, five scanners were started with an additional flag,

[7] https://github.com/PortSwigger/param-miner/blob/master/resources/headers
[8] https://github.com/PortSwigger/param-miner/blob/master/resources/params

specifically the "setheader" flag, to set this specified header.
To sum up, an instance of WCVS was started for each website. The scanners scanned every subdomain for the website sequentially. The version 0.4.36[9] of WCVS was used for these tests.

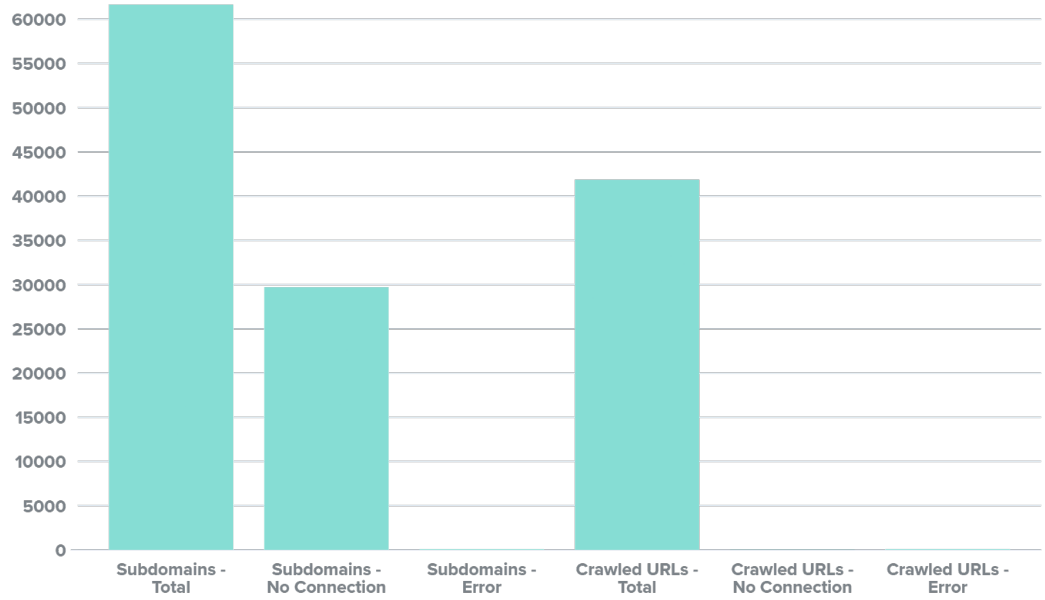

Figure 4.3: Amount of subdomains and URLs with no connection or an error during the cache analysis

4.3 Statistics

The scans ran for almost 12 days. Within this time frame 35 of the 51 scanners had finished testing every subdomain of the website in question. The remaining scanners were stopped, as enough data was collected. Overall, 61650 of the 300468 subdomains were scanned completely. Listing 4.3 illustrates selected statistics for those tested subdomains. To almost half of these subdomains no connection could be established. This is most likely due to the passive collection approach, as the passively collected subdomains may be out of date and non-existent anymore. During the cache analysis of 76 subdomains an error occurred wherefore these subdomains were skipped. 41857 additional URLs were crawled from the eligible subdomains. A total of 73620 URLs, consisting of the reachable subdomains and their crawled URLs, were completed by WCVS. For 50626 URLs a hit or miss indicator could be found. 114 of these were either not reachable or an error occurred during the cache analysis. To sum up, a total of 73620 URLs were tested. Listing 4.4 shows the identified hit or miss indicators. While in about 31% of the cases no hit or

[9] https://doi.org/10.5281/zenodo.5539478

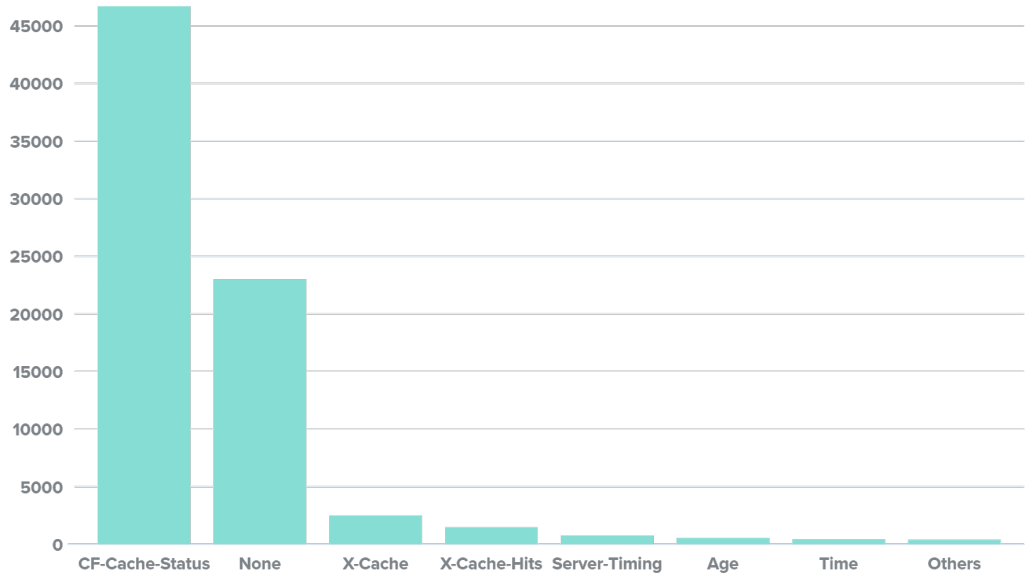

Figure 4.4: Amount of identified hit or miss indicators

miss indicator could be found, in almost 64% the CF-Cache-Status header was found to be a valid hit or miss indicator. This header is created by the Cloudflare CDN, indicating that most of the tested URLs use Cloudflare services for caching. If no explicit hit or miss indicator is found, WCVS measures the time in order to check if a cachebuster exists. This was successful in 438 cases.

During the tests a bug in the version 0.4.36[10] of the WCVS resulted in flawed cachebuster statistics. Around 10% of the confirmed cachebusters were in fact not eligible. Therefore a second test run was conducted using the version 0.4.39[11] which fixed the statistics bug. This time only the caches were analyzed and no web cache poisoning techniques were executed. Listing 4.5 illustrates the identified cachebusters of the second test with the correct statistics. Out of the 50445 URLs with a found hit or miss indicator, 49149 web caches were configured to not cache the content of the URL. This could be verified by sending two equal requests and receiving a cache miss both times. Only for 946 URLs a cachebuster could be identified. Most of the time the query parameter with the name `cb` was successful. The second most successful cachebuster was the `Origin` header; the `PURGE` and `FASTLYPURGE` HTTP methods were the third and fourth most successful cachebusters. For safety reasons `PURGE` and `FASTLYPURGE` should not be usable by users. These two HTTP methods force the web cache to clear every cached response of a URL. This makes it easy for an attacker to store a malicious response, if they found a way to

[10] https://doi.org/10.5281/zenodo.5539478
[11] https://doi.org/10.5281/zenodo.5540751

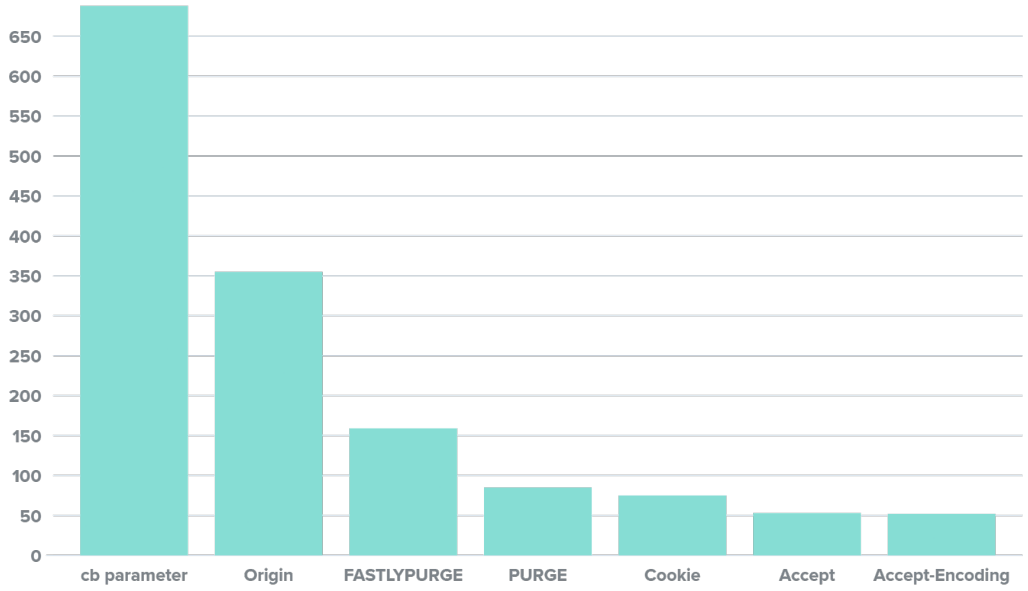

Figure 4.5: Amount of identified cachebusters

create oneThey can clear the cache for the affected URL and send the malicious request immediately afterwards to ensure it is cached.

4.4 Found Web Cache Poisoning Vulnerabilities

Four subdomains were found to be vulnerable to web cache poisoning. In all four cases there is unkeyed input which has an impact on the response. This is dangerous as an attacker can manipulate the response without affecting the cache key. However, during the limited time of this thesis no possibility was found to generate a malicious response using the unkeyed input. In the following the requests and responses which revealed the web cache poisoning vulnerabilities are explained. The first finding enabled it to change the value of the `Access-Control-Allow-Headers` header, while the other three enabled it to influence the response's body at certain locations. All four findings were achieved with the technique "unkeyed header poisoning" and a total of 11 of the tested URLs were affected. The vulnerabilities were reported to the corresponding bug bounty programs. For these reports a web cache poisoning report template was created. The template can be found in Appendix A.2.

1. Finding The first finding allows it to change the value of the `Access-Control-Allow-Headers` header of the response. The affected URL is a permanent redirection to the new location of a specific file. Such a request is illustrated in Listing 4.1 and the corresponding response in Listing 4.2. The two modifications WCVS made to the request can be seen in the first

Listing 4.1: Shortened request of the first finding

```
1 GET /scripts/optimizely.js?cb=548676702587 HTTP/2
2 Host: [Redacted]
3 Access-Control-Request-Headers: 492320315892
```

Listing 4.2: Shortened response of the first finding

```
 1 HTTP/2 301 Moved Permanently
 2 Date: Mon, 06 Sep 2021 13:30:33 GMT
 3 Location: https://cdn.optimizely.com/js/[Redacted].js?cb=54867
   6702587
 4 Access-Control-Allow-Origin: *
 5 Cf-Cache-Status: MISS
 6 Access-Control-Allow-Headers: 492320315892
 7 Access-Control-Allow-Headers: *
 8 Access-Control-Allow-Methods: GET, HEAD
 9 Access-Control-Expose-Headers: x-amz-meta-revision
10 Access-Control-Max-Age: 86400
```

listing. First, WCVS added the `cb` query parameter with a randomly generated 12 digit value as cachebuster. Second, WCVS added the `Access-Control-Request-Headers` header with another randomly generated 12 digits as poison value. The `Access-Control-Allow-Headers` value and the query string are highlighted in both listings, as they are both reflected in a header of the response. The query string is keyed and therefore was not applicable to unkeyed header poisoning. Also parameter cloaking and fat GET were not successful, because the query string was copied and appended to the value of the location header without any interpretation. Furthermore, the query string did not seem to have any impact on the response of the redirected URL. The `Access-Control-Allow-Headers` header is used both for HTTP requests and HTTP responses. It is usually used in preflight requests to find out which headers are allowed for requests and in responses to these preflight requests to show the allowed headers[12]. The `Access-Control-Allow-Headers: *` header in line 7 of the second listing, indicates that already every header is allowed. RFC 2616 [22, Section 4.2] defines that the values of headers with the same name are merged together. Therefore, it should not have any impact on the behavior of the browser if there are multiple `Access-Control-Allow-Headers` headers as long as one has a wildcard as value and if RFC 2616 is implemented correctly. Meta characters in both the query string and

[12] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Headers`

Listing 4.3: Shortened request of the second finding

```
1 GET /page/2?cb=634598798798 HTTP/2
2 Host: [Redacted]
3 Referer: 477242609399
```

Listing 4.4: Shortened response of the second finding (part 1)

```
1 <script type="text/javascript" nonce="[...]">
2   !function(s){
3      s.src='https://px.srvcs.tumblr.com/impixu?T=[...]&J=[...]&
       U=[...]&K=[...]&R=477242609399'.replace(/&R=[^&$]*/,'').
       concat('&R='+escape(document.referrer)).slice(0,2000).
       replace(/%.?.?$/,'');
4   }(new Image());
5 </script>
```

Listing 4.5: Shortened response of the second finding (part 2)

```
1 <noscript>
2   <img style="[...]" src="https://px.srvcs.tumblr.com/impixu?
    T=[...]&J=[...]=&U=[...]&K=[...]&R=477242609399">
3 </noscript>
```

the value of the `Access-Control-Allow-Headers` header were sanitized so that they were not interpreted by the web server. Thus, HTTP response splitting was not possible.

2. Finding The second finding allows it to add a string to the response's body at four certain places. Listing 4.3 illustrates a request which utilizes this behavior. Again, the query parameter `cb` was used as cachebuster. The header `Referer` was given a random poison value. Listing 4.4 shows the first part of the response where the poison value is reflected. Javascript is used to generate an image. The source of the image is set to an external URL with many query parameters. The last parameter called `R` contains the value of the `Referer` header. If the script is executed it does four changes to the URL, before creating the image. First, it removes the `R` parameter and its value; thereby removing the poison value. Second, it takes the `document.referrer` property, URL encodes it and uses this as the value of a new `R` query parameter which is appended to the URL. Third, if the URL is longer than 2000 characters, everything after the 2000th character is removed. Fourth, it removes the last three characters if they represent any URL encoded

Listing 4.6: Shortened request of the third finding

```
1 GET /?cb=553972046741 HTTP/2
2 Host: [Redacted]
3 Referer: 387921069845
```

Listing 4.7: Shortened request of the fourth finding

```
1 GET /?cb=121337194684 HTTP/1.1
2 Host: [Redacted]
3 User-Agent: 499136016762
```

character. The `document.referrer` property is read during runtime and therefore does not get influenced by caching. The poison value is URL encoded before it is added to the URL. Hence, it is not possible to break out of the string or to append additional query parameters to the URL. Listing 4.5 shows the second part of the response where the poison value is reflected. If Javascript is disabled an image is created with the same URL as the one in Listing 4.4. However, this time the URL is not modified by Javascript and thus the poison value is not removed. The poison value gets URL encoded before it is added to the URL. Hence, it is again not possible to break out of the string or to append additional query parameters to the URL. The third part of the response where the poison value is reflected is similar to the first part. Only some values of the other query parameters of the URL are different. This, however, does not make any difference for the behavior. In the same manner the fourth part is similar to the second part.

These four HTML code snippets seem to be common snippets for websites generated with Tumblr[13]. Internet searches with these snippets reveal a huge amount of other Tumblr websites with identical snippets.

3. Finding The third finding is similar to the second one. Once again a website generated by Tumblr allows it to add the value of the `Referer` header to the URL of the four image sources. The header is again unkeyed and the value gets URL encoded before the insertion. A total of five web pages of this subdomain were found to be affected by this behavior.

4. Finding The fourth finding allows it to add a string to the response's body at one place. Listing 4.7 illustrates a request which utilizes this behavior. Similarly to the other findings, the `cb` query parameter is used as cachebuster. The poison value was inserted as the value of the `User-Agent` header. Listing 4.8 shows a part of the corresponding

[13] Tumblr is a service to create microblogs (`https://www.tumblr.com/`)

Listing 4.8: Shortened response of the fourth finding

```
1 <div id="client_ua" style="display:none">
2   499136016762
3 </div>
```

response. The poison value is reflected inside of a `div` element. As the style attribute of the `div` element is set to `display:none` the div element including its content is invisible. Thus, the poison value is only visible in the page source. Any string which gets injected by the `User-Agent` header gets HTML-encoded which prevents XSS attacks and injection of HTML code in general. A total of four web pages of this subdomain were found to be affected by this behavior.

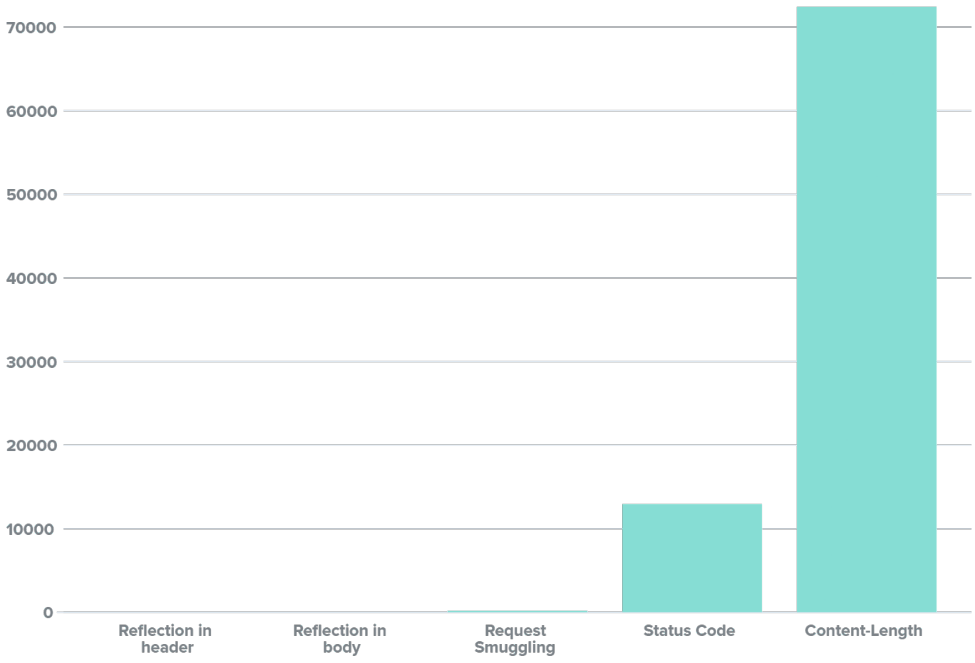

Figure 4.6: Amount of false positives

4.5 False Positives

While WCVS reported a total of 85577 potential vulnerabilities, only 11 could be confirmed as correct. The remaining reports seem to be false positives. Due to the huge amount of reports only samples could be checked for correctness. These samples were chosen wisely in order to suggest the same result for other findings with similar characteristics. All findings

were grouped by their successful poisoning indicator, which were introduced in Section 3.7, or if the used technique was HTTP request smuggling. Because the two successful poisoning indicators for HTTP request smuggling, timeout and 501 error code, differ from the successful poisoning indicators of the other techniques, HTTP request smuggling was grouped separately from the other techniques. Listing 4.6 illustrates the distribution of the amount of the findings on each group of the five groups. It can be clearly seen that the `status code` and `content-length` successful poisoning indicators issued almost every false positive. WCVS tested about 8600 variants for each URL which has a cachebuster, whereas 8500 variants arose because of the length of both used wordlists. To test 8600 variants with a rate limit of one request per second takes about five hours. If the `status code` or `content-length` of a web page is changed a few minutes or hours into the test, the remaining variants will each report that they were successful. Hence, leading to such a high amount of false positives. A more thorough breakdown of the five false positives groups is given in the following.

Reflection in Header Only for one finding the successful poisoning indicator was that its poison value was reflected in the header. This one is the first found web cache poisoning vulnerability. Thus, this successful poisoning indicator did not lead to any false positives.

Reflection in Body A total of 11 findings have been identified with the successful poisoning indicator that the poison value was reflected in the body. Ten of them were identified as correct, while one was identified as false positive. This false positive occured, because the `X-Forwarded-Port` header was tested with the predefined poison value `31337`. This was done because a random 12 digit value, which is normally used for poison values, is not a valid port number. However, this predefined port value is not that unique and the target website contained an URL with `31337` in it.

Request Smuggling 184 findings were identified with the HTTP Request Smuggling technique. These findings were checked manually and also with the Burp Suite extension "HTTP Request Smuggler"[14]. This extension was developed by the already mentioned security researcher James Kettle and has the ability to detect and exploit HTTP Request Smuggling. The HTTP Request Smuggling findings can be split into two different successful poisoning indicators. For 51 the successful poisoning indicator was that the status code changed to `500`. However, HTTP Request Smuggling could not be confirmed. Hence, those findings are categorized as false positives. For the other 133 the successful poisoning indicator was that three consequent requests all resulted in a timeout when testing for HTTP Request Smuggling. However, it was not possible to exploit this behavior. Furthermore, while the time out variant is said to generate only a few false positives, this is

[14] `https://github.com/portswigger/http-request-smuggler`

not completely certain and needs to be confirmed [25]. The "HTTP Request Smuggler" also reported the findings as most likely vulnerable to HTTP Request Smuggling due to the timeouts. Nonetheless, it also could not exploit the vulnerabilities or confirm them with certainty. As the possibility that these findings are false positives could not be erased completely, they were all categorized as false positives.

Status Code 12944 findings were reported with the successful poisoning indicator that the status code changed. The amount of false positives for this successful poisoning indicator is so high, because the status code of websites changed independently of the web cache poisoning technique used. There were four status code changes in particular which triggered the false positives.

1. **503 Service Unavailable** The web server was currently not available.

2. **504 Gateway Timeout** The web cache or another proxy server did not receive a response from the web server.

3. **403 Forbidden** The IP address got blocked, because the scanner sent too many requests.

4. **429 Too many requests** The same as 403 Forbidden; the IP address got blocked, because too many requests were sent.

If the IP address got blocked or the web cache or the web server was unavailable the status code received by the WCVS changed. Every technique received this same status code which was different from the initial status code. Thus, every technique and variant was reported as successful after the change of the status code happened.

Content-Length For 72437 findings the successful poisoning indicator is a change of the content-length, whereas 68995 came from only two of the 51 tested domains. These two domains in particular have multiple subdomains where some files are updated every few minutes. WCVS treated a change of the content-length as a successful poisoning indicator for every tested technique and variant afterwards, if the length differed more than 2000 bytes as specified by one of the command-line flags. Four domains had only one finding with this successful poisoning indicator, however the reported behavior could not be replicated. The remaining domains with findings with the content-length as successful poisoning indicator also had periodic or sudden file changes which were independent of the tests.

4.6 False Positives Countermeasures

Two types of false positives could be identified. The first type, which only occurred one time, is that the targeted website contains a string in its headers or body which is randomly used by WCVS as poison value. This is especially likely if a short poison value, for example a valid port number, is used. The second type, which was accountable for almost every false positive, was that the response of the server changed independently from the scan.

The first type is simple yet effective to solve. When WCVS generates a poison value it needs to check this value before using it for a request. For the stored website response must be checked whether this poison value is already present. In this case a new poison value must be generated, checked for its existence again. This process is repeated until a unique value could be found or until a specified amount of tries was not successful. If no unique value could be found, an error will be thrown.

The second type can also be solved simply yet effectively. When WCVS spots the status code or content-length indicator it reports a finding. Instead it should issue another victim request with another cachebuster value, thus receiving a response from the web server and not from the web cache. If the response differs from the expected and stored default response, the default response is replaced by the new received one. Afterwards, the check for successful poisoning indicators is repeated.

Chapter 5

Conclusion

In this thesis, the known web cache poisoning techniques were correlated on the basis of their impact and their preconditions. During the thesis Web Cache Vulnerability Scanner (WCVS) was enhanced with five more techniques, so that it is able to identify all the mentioned web cache poisoning techniques. Also WCVS's ability to find cachebusters, which are mandatory to test for web cache poisoning, was improved. Afterwards, websites were chosen to test for web cache poisoning using WCVS. Only websites which had a bug bounty program and allowed the use of automatic scanners were considered in order to avoid any legal troubles. Additionally, only websites which are within the top 1000 of the world's most frequently visited websites were chosen. All in all 73620 URLs were tested belonging to 51 domains and their subdomains. The results revealed that for 31% of the URLs no hit or miss indicator could be found. Therefore, for these URLs there was probably no web cache in use. For further 67% of the URLs the web cache was configured to not cache the responses, thus always passing requests to the web server. Only for 946 URLs a cachebuster could be identified and thus only these URLs could be tested for web cache poisoning. Out of these URLs 11 were found to be vulnerable to web cache poisoning. Interestingly, all affected URLs were vulnerable to the same web cache poisoning technique "unkeyed header poisoning".

Those tests proved the effectiveness of WCVS. The starts of the tests could be automated in order to start multiple tests with different command line flags. WCVS crawled successfully for further URLs to test and excluded URLs from the test which did not get cached or did not have a cachebuster in order to not waste time and other resources. This left just under 1000 URLs to test. WCVS found web cache poisoning in 11 of these URLs. The results were analyzed with automated scripts to quickly locate findings. Two not anticipated circumstances resulted in a vast amount of false positives. These two circumstances were quickly identified and simple yet effective solutions for both were presented. These solutions will be implemented into WCVS soon.

The tested websites and therefore the results of the tests are not representative for the rest of the Internet, because of two reasons. The first reason is that the number of scanned websites may simply not be big enough. The second reason is that the selection of websites was based on very specific criteria. On the one hand only websites of the 1000 most frequently visited ones worldwide were chosen. Therefore, the selection does not include amateur, small-sized or medium-sized websites. On the other hand only websites with a bug bounty program were chosen. Almost every bug bounty program was active for many months, if not years. During this time many web cache poisoning vulnerabilities may have already been reported and fixed. However, the selection of websites had to be this specific in order to be legally safe and to get a first insight into the spread of web cache poisoning. Future work has to show more representative insight into the spread and impact of web cache poisoning for the whole Internet. WCVS proved itself to be suitable for automated scanning for web cache poisoning and therefore is an appropriate candidate for further automated tests.

Appendix A

Additional Information

A.1 WCVS Help Output

```
❯ ./wcvs -h
https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner
version 0.4.39

Usage: Web-Cache-Vulnerability-Scanner(.exe) [options]

General Options:
--help                          -h      Show this help and quit
--verbosity                     -v      Set verbosity. 0 = quiet, 1 = normal, 2 = verbose
--reqrate                       -rr     Requests per second. Default value is 0 (= unlimited)
--threads                       -t      Threads to use. Default value is 20
--timeout                       -to     Seconds until timeout. Default value is 15
--onlytest                      -ot     Choose which tests to run. Use the , seperator to specify multiple
ones. Example: -onlytest 'cookies,css,forwarding,smuggling,dos,headers,parameters,fatget,cloaking,splitting
'
--skiptest                      -st     Choose which tests to not run. Use the , seperator to specify multi
ple ones. Example: -skiptest 'cookies,css,forwarding,smuggling,dos,headers,parameters,fatget,cloaking,split
ting'
--proxycertpath                 -ppath  Path to the cert of the proxy you want to use. The cert has to have
 the PEM Format. Burp e.g. is in the DER Format. Use the following command to convert it: openssl x509 -inf
orm DER -outform PEM -text -in cacert.der -out certificate.pem
--proxyurl                      -purl   Url for the proxy. Default value is http://127.0.0.1:8080
--force                         -f      Perform the tests no matter if there is a cache or even the cachebu
ster works or not
--contentlengthdifference       -cldiff Threshold for reporting possible Finding, when 'poisoned' response
differs more from the original length. Default is 0 (don't check).

Generate Options:
--generatepath          -gp     Path all files (log, report, completed) will be written to. Example: -gp '/
p/a/t/h/'. Default is './'
--generatereport        -gr     Do you want a report to be generated?
--escapejson            -ej     Do you want HTML special chars to be encoded in the report?
--generatecompleted     -gc     Do you want a list with completed URLs to be generated?

Request Options:
--url                   -u      Url to scan. Has to start with http:// or https://. Otherwise use file: to
specify a file with (multiple) urls. E.g. -u https://www.example.com or -u file:templates/url_list
--usehttp               -http   Use http instead of https for URLs, which doesn't specify either one
--declineCookies        -dc     Do you don't want to use cookies, which are received in the response of the
 first request?
--cachebuster           -cb     Specify the cachebuster to use. The default value is cachebuster
--setcookies            -sc     Set a Cookie. Otherwise use file: to specify a file with urls. E.g. -sc uid
=123 or -sc file:templates/cookie_list
--setheaders            -sh     Set a Header. Otherwise use file: to specify a file with urls. E.g. -sh 'Us
er-Agent: Safari/1.1' or -sh file:templates/header_list
--setparameters         -sp     Set a Query Parameter. Otherwise use file: to specify a file with urls. E.g
. -sp user=admin or -sp file:templates/parameter_list
--setbody               -sb     Set the requests' body. Otherwise use file: to specify a file with urls. E.
g. -sb 'admin=true' or -sh file:templates/body_file
--post                  -post   Do a POST request instead of a GET request
--contenttype           -ct     Set the contenttype for a POST Request. Default is application/x-www-form-u
rlencoded. If you don't want a content-type to be used at all use -ct ''
--statuscode            -status Expected status code of the responses. If not specified it takes the status
 code of the first response
--parameterseperator    -ps     Specify the seperator for parameters. The default value is &
--useragentchrome       -uac    Set chrome as User-Agent

Crawl Options:
--recursivity   -r      Put (via href or src specified) urls at the end of the queue if the domain is the s
ame. Specify how deep the recursivity shall go. Default value is 0 (no recursivity)
--reclimit      -rl     Define a limit, how many files shall be checked recursively. Default is 0 (unlimite
d)
--recinclude    -rin    Choose which links should be included. Seperate with a space. E.g: -rin '.js .css'
--recexclude    -rex    Use -cp (-completedpath) or -gc (-generatecompleted) to generate a list of already
completed URLs. Use -rex path/to/file so the already completed URLs won't be tested again recursively.
--recdomains    -red    Define a additional domain which is allowed to be added recursively. Otherwise use
file: to specify a file with urls. E.g. -sh 'api.example.com' or -sh file:templates/recdomains_list

Wordlist Options:
--headerwordlist        -hw     Wordlist for headers to test. Default path is 'wordlists/top-headers'
--parameterwordlist     -pw     Wordlist for query parameters to test. Default path is 'wordlists/top-param
eters'
```

A.2 Report Template

Summary

The website is vulnerable to web cache poisoning

Affected URLs

- x
- y
- z

Description

The header {Header} influences the HTTP response of the web server. It is possible to write arbitrary content into the body/{Header2} header of the HTTP response. However, this header does not get included in the cache key. Hence, if a HTTP response which was influenced by the {Header} header gets cached this HTTP response will be served to other users. The consequences can be devastating if the attacker is able to exploit another vulnerability.

Steps to Reproduce

Two requests are sent. The first one utilizes the {Header Name} header to generate an influenced HTTP response. The second one does not utilize the {Header} header, however receives the same influenced HTTP response, as the {Header} header is not included into the cache key. In order for the first response to be cached a cachebuster is used. In this case the cachebuster is a query parameter called “cb” with a long number as value.

Request 1:
GET /?cb=123456789 HTTP/1.1
Host: example.com
{Header}: foobar

Request 2:
GET /?cb=123456789 HTTP/1.1
Host: example.com

Recommendation

Every part of a HTTP request which has any influence on the HTTP response should be included into the cache key.

Further Information

For further information on web cache poisoning check out
https://hackmanit.de/de/blog/142-is-your-application-vulnerable-to-web-cache-poisoning

Listings

List of Figures

List of Tables

Bibliography

[1] Amit Klein. *"Divide and Conquer": HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics*. 2004. URL: https://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf.

[2] James Kettle. *Practical Web Cache Poisoning: Redefining 'Unexploitable'*. 2018. URL: https://portswigger.net/kb/papers/7q1e9u9a/web-cache-poisoning.pdf.

[3] James Kettle. *Bypassing Web Cache Poisoning Countermeasures*. 2018. URL: https://portswigger.net/research/bypassing-web-cache-poisoning-countermeasures.

[4] James Kettle. *Responsible denial of service with web cache poisoning*. 2019. URL: https://portswigger.net/research/responsible-denial-of-service-with-web-cache-poisoning.

[5] James Kettle. *Web Cache Entanglement: Novel Pathways to Poisoning*. 2020. URL: https://portswigger.net/kb/papers/c3wwniai/web-cache-entanglement.pdf.

[6] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. "Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack". In: *CCS'19. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, United Kingdom, November 11-15, 2019*. CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security (London United Kingdom). Ed. by Lorenzo Cavallaro et al. New York, NY: Association for Computing Machinery, 2019, pp. 1915–1936. ISBN: 9781450367479. DOI: 10.1145/3319535.3354215.

[7] Aleksei Tiurin. *Cache poisoning denial-of-service attack techniques*. 2021. URL: https://www.acunetix.com/blog/web-security-zone/cache-poisoning-dos-attack-techniques/.

[8] Ramón Cáceres et al. "Web proxy caching". In: *ACM SIGMETRICS Performance Evaluation Review* 26.3 (1998), pp. 11–15. ISSN: 0163-5999. DOI: 10.1145/306225.306230.

[9] G. Barish and K. Obraczke. "World Wide Web caching: trends and techniques". In: *IEEE Communications Magazine* 38.5 (2000), pp. 178–184. ISSN: 0163-6804. DOI: 10.1109/35.841844.

[10] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. "Mind the cache". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19: The 34th ACM/SIGAPP Symposium on Applied Computing (Limassol Cyprus). Ed. by Chih-Cheng Hung and George A. Papadopoulos. New York, NY, USA: ACM, 4082019, pp. 2497–2506. ISBN: 9781450359337. DOI: 10.1145/3297280.3297526.

[11] Seyed Ali Mirheidari. *Confused by Path: Analysis of Path Confusion Based Attacks*. 2020. URL: https://iris.unitn.it/retrieve/handle/11572/280512/382175/rpo/.

[12] Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali, eds. *Content delivery networks*. eng. Vol. 9. Lecture notes in electrical engineering. Berlin: Springer, 2010. 417 pp. ISBN: 3642096700.

[13] Run Guo et al. "Abusing CDNs for Fun and Profit: Security Issues in CDNs' Origin Validation". In: *2018 IEEE 37th International Symposium on Reliable Distributed Systems. SRDS 2018*. Piscataway, NJ: IEEE, 2018. ISBN: 9781538683019. DOI: 10.1109/srds.2018.00011.

[14] R. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234*. IETF. 2014. URL: http://tools.ietf.org/rfc/rfc7234.txt.

[15] Ankit Shrivastava, Santosh Choudhary, and Ashish Kumar. "XSS vulnerability assessment and prevention in web application". In: *Proceedings on 2016 2nd International Conference on Next Generation Computing Technologies (NGCT). October 14th-16th, 2016, Center for Information Technology, University of Petroleum and Energy Studies, Dehradun*. 2016 2nd International Conference on Next Generation Computing Technologies (NGCT) (Dehradun, India). Ed. by Amit Agarwal. NGCT et al. Piscataway, NJ: IEEE, 2016, pp. 850–853. ISBN: 978-1-5090-3257-0. DOI: 10.1109/NGCT.2016.7877529.

[16] Roger M. Needham. "Denial of service". In: *Proceedings of the 1st ACM conference on Computer and communications security*. the 1st ACM conference (Fairfax, Virginia, United States). Ed. by Dorothy Denning. ACM Special Interest Group on Security, Audit, and Control. New York, NY: ACM, 1993, pp. 151–153. ISBN: 0897916298. DOI: 10.1145/168588.168607.

[17] Craig A. Shue, Andrew J. Kalafut, and Minaxi Gupta. *Exploitable Redirects on the Web: Identification, Prevalence, and Defense*. 2008. URL: https://www.usenix.org/legacy/event/woot08/tech/full_papers/shue/shue.pdf.

[18] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax. RFC 3986.* IETF. 2005. URL: https://datatracker.ietf.org/doc/html/rfc3986.

[19] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230.* IETF. 2014. URL: https://datatracker.ietf.org/doc/html/rfc7230.

[20] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231.* IETF. 2014. URL: https://datatracker.ietf.org/doc/html/rfc7231.

[21] Chaim Linhart et al. *HTTP REQUEST SMUGGLING.* 2005. URL: https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[22] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.* IETF. 1999. URL: http://tools.ietf.org/rfc/rfc2616.txt.

[23] Haixin Duan. *Forwarding-Loop Attacks in Content Delivery Networks.* 2016. URL: https://www.researchgate.net/profile/haixin_duan/publication/307578711_forwarding-loop_attacks_in_content_delivery_networks.

[24] Omer Gil. *Web cache deception attack.* 2017. URL: https://www.blackhat.com/docs/us-17/wednesday/us-17-gil-web-cache-deception-attack-wp.pdf.

[25] James Kettle. *HTTP Desync Attacks. Request Smuggling Reborn.* 2019. URL: https://portswigger.net/kb/papers/z7ow0oy8/http-desync-attacks.pdf.

[26] Victor Le Pochat et al. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: *Proceedings 2019 Network and Distributed System Security Symposium.* Network and Distributed System Security Symposium (San Diego, CA). Ed. by Alina Oprea and Dongyan Xu. Reston, VA: Internet Society, February 24-27, 2019. ISBN: 1-891562-55-X. DOI: 10.14722/ndss.2019.23386.