FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study programme / specialisation:
Computer Science

The spring semester, 2022

Open / Confidential

Author:
Stefan Aasebø
William Kristoffersen

William Kristoffersen. Stefan Aasebø
(signature authors)

Course coordinator:
Arian Baloochestani

Supervisor(s):
Arian Baloochestani

Thesis title:
theeCB: A tool to programmatically circumvent caches

Credits (ECTS): 20

Keywords: Caching, Proxy, Web application, Information security, Penetration testing.

Pages: 70

Stavanger, 15.05.2022

date/year

W. KRISTOFFERSEN AND S.AASEBØ
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

theeCB: A tool to programmatically circumvent HTTP proxy caches

Bachelor's Thesis - Computer Science - May 2022

```
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
    if len(opts) < 1 || len(opts) > 2 {
        return nil, fmt.Errorf("wrong number of options: %d", len(opts))
    }
    c = &Configuration{}
    for _, opt := range opts {
        switch v := opt.(type) {
        case gorums.NodeListOption:
            c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
            if err != nil {
                return nil, err
            }
        case QuorumSpec:
            // Must be last since v may match QuorumSpec if it is interface{}
            c.qspec = v
        default:
            return nil, fmt.Errorf("unknown option type: %v", v)
        }
    }
    // return an error if the QuorumSpec interface is not empty and no implementati
    var test interface{} = struct{}{}
    if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
        return nil, fmt.Errorf("missing required QuorumSpec")
    }
    return c, nil
}
```

I, **W. Kristoffersen and S.Aasebø**, declare that this thesis titled, "theeCB: A tool to programmatically circumvent HTTP proxy caches" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Abstract

In recent years as the internet has evolved and there have been major advances in technology, demands on the web infrastructure has grown. An increasing growth of the internet without any performance enhancing measures would lead to overloading. Caching helps to resolve this issue by storing copies of certain data for later usages. This stored data can then be delivered much faster than to retrieve the data from the web application. While speeding up response times, insecure cache configuration can lead to vulnerabilities.

In this thesis, we have developed a cache detection tool, theeCB, that informs the user if a web application uses a form of caching mechanism. This tool can be useful for discovering whether a web application uses some cache mechanisms, and which caching rules are used. The goal of this tool is to bring useful information, that can be further used in penetration testing, or for debugging purposes.

Acknowledgements

We would like to thank our supervisor, Arian Baloochestani for his valuable guidance, expertise, and feedback throughout this thesis process.

Contents

Chapter 1

Introduction

In this thesis our goal is to create a tool to detect caching in HTTP proxies, as well as to gain information of how the cache is set up. The tool is a cache buster; therefore, we chose the name, ”thee Cache Buster”, in short ”theeCB”. This bachelor thesis builds upon the idea that it is possible to automate the current manual way of detecting whether a HTTP proxy uses caching. theeCB’s objective is to send multiple HTTP requests to determine how a normal response looks like, and use different methods to attempt to avoid the cache. theeCB will potentially return data regarding how it bypassed the cache. The output of theeCB aids a penetration tester to find caching related vulnerabilities further manually.

1.1 Background and motivation

HTTP proxy caches are complicated and there are numerous ways of setting them up. The multitude of ways of setting up a cache makes it very difficult to understand how the cache is set up during a penetration test.

Some proxies hand out a header called E-Tag; this header includes a unique identifier for the specific cached response. This header cannot be trusted blindly as the caching mechanism may be implemented in any way imaginable. The same way any cache related headers should not be trusted blindly either.

It can be difficult to properly test a web application that has a HTTP proxy in front, when the information given from the proxy is sometimes incorrect and difficult to interpret. This results in a major problem, penetration testers taking lightly on caching, and a hidden surface with critical vulnerabilities.

When conducting a penetration test, any cache flaw can significantly amplify any low impact vulnerabilities. If a vulnerability is found, and the cohesive payload is stored in the cache, the vulnerability suddenly affects everyone visiting the web application.

The process of identifying how the cache is set up is time consuming, especially if testing every single endpoint in a web application. Therefore, automating this process increases efficiency and precision during a penetration test in most cases.

1.2 Contribution and approach

The main idea behind our thesis originates from James Kettle. James Kettle is a researcher and developer at PortSwigger, looking at advanced hacking techniques, and shares his knowledge with the community [1]. James Kettle did research on caching and posted findings on PortSwigger's website [2]. This research includes the important information regarding proxies and caching. James Kettle refers to his work as manual techniques, we however, are going to automate some of these techniques.

Kettle's research presents basic information of how caching works, and how to effectively bypass and manipulate it. We used information from a research paper written by James Kettle named Web cache entanglement to create theeCB [2]. By further using the research done by Kettle, we can lessen the research we would have had to do ourselves.

When developing and designing theeCB, we are using some information from Kettle's research paper, as well as researching ourselves. We are using a tool created by PortSwigger called Burp Suite, this is an HTTP proxy designed to manipulate requests and responses. Burp Suite allows us to inspect traffic to verify that theeCB works as intended and allows us to learn more about web applications and proxies.

The main objective is to create a tool to make everyday penetration testing more efficient and reduce false negatives. Another objective is to create a unique, robust, easy, and effective tool, which provides the essential data to determine if a web application uses caching. We have a desire to make a clean code, which is easily extendable and keeping up to date. We also want to make the code modular, meaning that parts of the code can be re-used in later projects.

1.3 Choice of language

In order to create and design theeCB as useful as possible, a modern and highly developed programming language like Golang is chosen. Golang is a cross-platform language, meaning code written to this language is compiled to different platforms with ease [3]. There are many benefits of using a programming language like Golang, for instance Golang is built rather different from other languages and there is no need for third-party web frameworks. Golang also has an organized and clean library including a variety of different packages. Another considerable reason for choosing Golang is its rich support for concurrency. In Golang, go functions run independent of each other, making it simple to run a concurrent function.

Chapter 2

Background

2.1 User Agent

A user agent is an agent that performs a request on behalf of the user, for instance a web browser. A web browser is what most people use to communicate over the internet, however this is not the only way to do so. Besides browsers, it is also possible to use other programs to perform HTTP requests. In this thesis Golang is used to create an application which sends HTTP requests on behalf of the user

2.2 Open Systems Interconnection Model

The Open Systems Interconnection Model or more commonly known as the OSI model is a reference model for data communications [4]. The OSI model delivers a framework for sets of protocols, or in other words a language that should be used when different systems are communicating. The OSI model is divided into 7 layers, and each layers have their own responsibilities. The layers are as following:

- Application layer
- Presentation layer
- Session layer
- Transport layer
- Network layer

- Data link layer
- Physical Layer

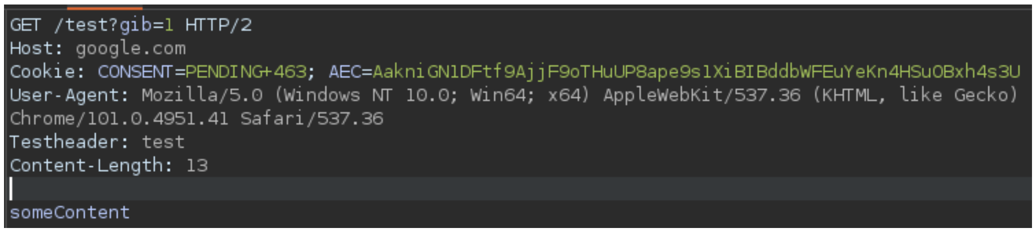
In this thesis we will be exclusively looking at the application layer, as that is where the HTTP protocol operates.

2.3 Hyper Text Transfer Protocol

The hypertext transfer protocol or more commonly known as HTTP is a hypermedia data transfer protocol found in the application layer in the OSI model [5]. This protocol is a central part of the internet and builds a foundation of how data communication works. The HTTP protocol creates and manages communications between the servers and clients, using requests and responses. For instance, a user agent initiates a request for data that are located on the web application, like an HTML document. This HTML document is generated by many other features like CSS, scripts, etc. [6]. Upon retrieval of the HTML document, the user agent typically makes multiple other requests for each of the features. On the other side of the request, we have the web applications. These servers are constantly listening for request and provides a response for each of the individual request received. As a result, the web application delivers the data or documents requested. The user agent then put these responses together to create the entire web page.

An HTTP request is defined by different elements, method, scheme, host, path, query string, headers, and body. The elements of an HTTP request are as following:

- The scheme specifies the protocol that is going to be used,for web applications the scheme are often http or https.
- The host specifies which host the request is sent to.
- Path defines which endpoint on the host the request is requesting.
- The query string is a list of parameters sent to the web application.
- Headers are also a list of parameters, and can be used in a similar way to the query string.



```
GET /test?gib=1 HTTP/2
Host: google.com
Cookie: CONSENT=PENDING+463; AEC=AakniGN1DFtf9AjjF9oTHuUP8ape9s1XiBIBddbWFEuYeKn4HSuOBxh4s3U
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/101.0.4951.41 Safari/537.36
Testheader: test
Content-Length: 13

someContent
```

Figure 2.1: Sample HTTP request.

- The body is plain text sent to the web application, this can be used in a similar way of the headers and query string, but is a lot more flexible.

- Request URI is another word used for a combination of the scheme, host, path and query string.

The figure 2.1 shows an example of how a request might look. The following list shows an example request with the elements of the request in figure 2.1.

- The method in figure 2.1 is ”GET”.

- The scheme in figure 2.1 is not specified, but it is ”https”.

- The host in figure 2.1 is ”google.com”.

- The path in figure 2.1 is ”/test”.

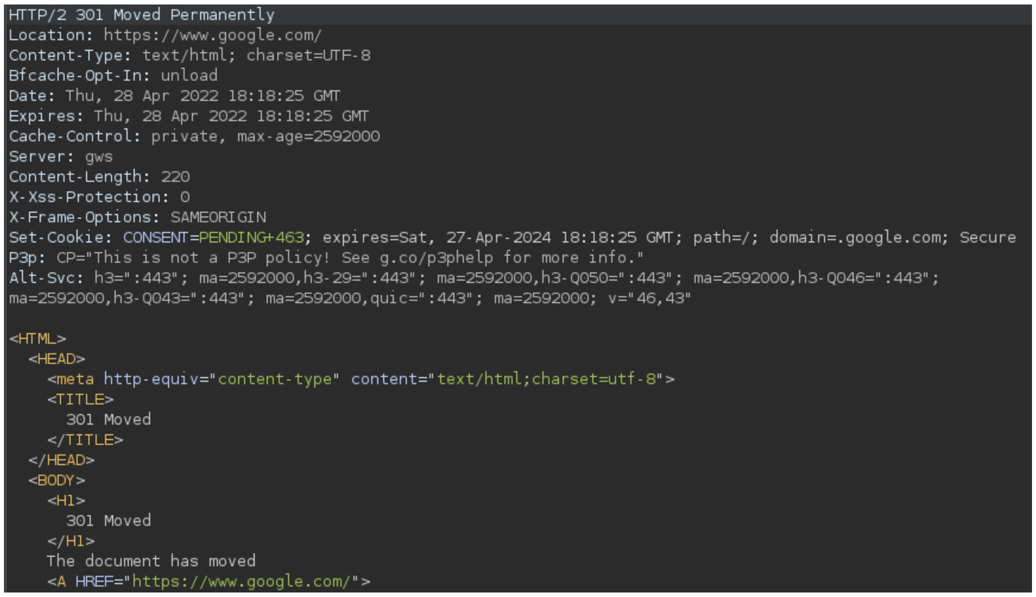
- The query string in figure 2.1 is ”gib=1”.

- The header names in figure 2.1 are ”Host”, ”Cookie”, ”User-Agent”, ”Test-Header” and ”Content-Length”.

- The body in figure 2.1 is ”someContent”.

- The request URI in figure 2.1 would be ”https://google.com/test?gib=1”.

The HTTP protocol messages is designed to be rather simple and easily understandable [5]. HTTP is also designed in a way that allows sessions by using HTTP cookies, resulting in request sharing the same state. Therefore, there does not need to be any connection between the current state and the current connection to the server. Furthermore, the HTTP protocol does not control the connection



```
HTTP/2 301 Moved Permanently
Location: https://www.google.com/
Content-Type: text/html; charset=UTF-8
Bfcache-Opt-In: unload
Date: Thu, 28 Apr 2022 18:18:25 GMT
Expires: Thu, 28 Apr 2022 18:18:25 GMT
Cache-Control: private, max-age=2592000
Server: gws
Content-Length: 220
X-Xss-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: CONSENT=PENDING+463; expires=Sat, 27-Apr-2024 18:18:25 GMT; path=/; domain=.google.com; Secure
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443";
ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"

<HTML>
  <HEAD>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <TITLE>
      301 Moved
    </TITLE>
  </HEAD>
  <BODY>
    <H1>
      301 Moved
    </H1>
    The document has moved
    <A HREF="https://www.google.com/">
```

Figure 2.2: Example HTTP Response

since this is determined upon the transportation layer in the OSI model. However, the HTTP protocol musts have a reliable connection so that packages are not misplaced, in addition it needs the ability to present an error if a situation like this occurs. Consequently, the HTTP protocol uses the TCP standard instead of the UDP standard. Due to this set of pre-determined procedures, a TCP connection must be established between web application server and the user agent when there is a need for communication [7]. Firstly, the user agent must open a connection or multiple connections or reuse a connection. Secondly, the user agent sends a HTTP message. Figure 2.1 illustrates an example.

Then the web application sends a message back, which may look like the response in figure 2.2.

2.4 Cross-Site Scripting

In short Cross-Site Scripting (XSS) is a commonly known vulnerability where the adversary gets control over the victim's browser by code injection [8]. This control can potentially be used to hijack sessions or perform unwanted actions on behalf of the victim. A common payload for finding XSS vulnerabilities is

”<script>alert(1)</script>”, to simplify; if this payload is reflected in the response you have found a vulnerability. This is a proof-of-concept payload which simply shows that you have found a vulnerability, and this is the payload we will be using when explaining cache-related vulnerabilities.

2.5 What is a proxy?

A proxy is a like a postman forwarding the message to some destination [9]. A proxy can be used for multiple purposes, either for performance or security reasons. In this thesis we are explicitly discussing HTTP proxies. There are multiple other forms of proxies, such as SOCKS proxies, FTP proxies, SSL proxies, but those are not relevant for this thesis. For this thesis we will be mostly be looking at reverse proxies, although forward proxies are also highly relevant. A forward proxy is a proxy used on the client side; the proxy forwards the messages to the destination decided by the client [10]. This is a proxy the client can decide to use or not, it also decides how to utilize it. A forward proxy is very useful for giving others access to something on an internal network, without giving access to everyone else, similar to a VPN [9]. It is also useful for inspecting traffic, for examples to block certain content, such as malicious websites, on the local network. It is also useful for modifying requests on the fly, which is a necessity of penetration testers. However, we are not going to discuss forward proxies further in this thesis, as it would act in the exact same manner as the reverse proxy would. Attacking a forward proxy is also a much more targeted attack, meaning it would be more specific and it wouldn’t affect as many people.

A reverse proxy is a proxy set up on the server side [10]. This is a proxy the end user seldom can choose to use or not. A reverse proxy is often used in front of a web application and end users are seldom talking directly to the web application. The reverse proxy then takes the information given to the web application. When implementing a reverse proxy in front of a web application, it gives the possibility of implementing caching, load balancing, security mechanisms and more. The biggest difference between a forward and a reverse proxy, is that the destination of the reverse proxy is not control-able by the end user. Everything that can be done with a proxy is not in scope for this thesis, but the important thing is that the proxy relays information to the web application, and controls everything caching related.

2.6 What is caching?

In later years as the internet has evolved and there have been major advances in the computer technology, the demands on the web infrastructure have grown [11]. An increasing growth of the internet without any performance enhancing measures, would lead to server overloading, in the form of terrible service response times and a lack of efficiency. Caching resolves this issue, as it enhances the performance by storing a copy of certain data for later usages [12]. For instance, a copy of a web page from a web application. When the clients send an HTTP request for data for the first time, the web application provides an HTTP response. The web application generates a response and sends it to the client. And as a result, the requested data in form of images, CSS, files, HTML documents are provided to the client. The second time the client requests data from the web application, the proxy's cache intercepts and returns a copy of the collected data without informing the web application. This saves turnaround time, bandwidth and reduces the load on the server significantly since the client don't have to re download data directly from the web application. Often the web applications don't see the request, since the web cache already has the data stored. The proxy decides with help of a cache key, if the request is the same and if the cache already has a response stored for it. When a client is requesting a web page multiple times, it may be a waste of resources to create a new response from the web application every time. The solution to this is caching.

A major portion of a web application can be stored in the cache for a longer period. Some data may include sensitive information such as credentials, or personal information. Caching coherent data could have devastating effects on users. It is very important to review the set-up of the caching mechanism to make sure only non-sensitive data is cached. Scripts and other content that are modified frequently should not be cached either, as the cache may take a while to update. Caching is used in various locations and have different effects. We are only looking at caching in HTTP proxies in this thesis, other caching mechanisms such as local storage and DNS caching, is out of scope.

In figure 2.3 the first request sent from a client goes to the web application, which creates and returns a response. The cache recognizes the following request and knows what the web application will return. The proxy then just returns the response instead of involving the web application.

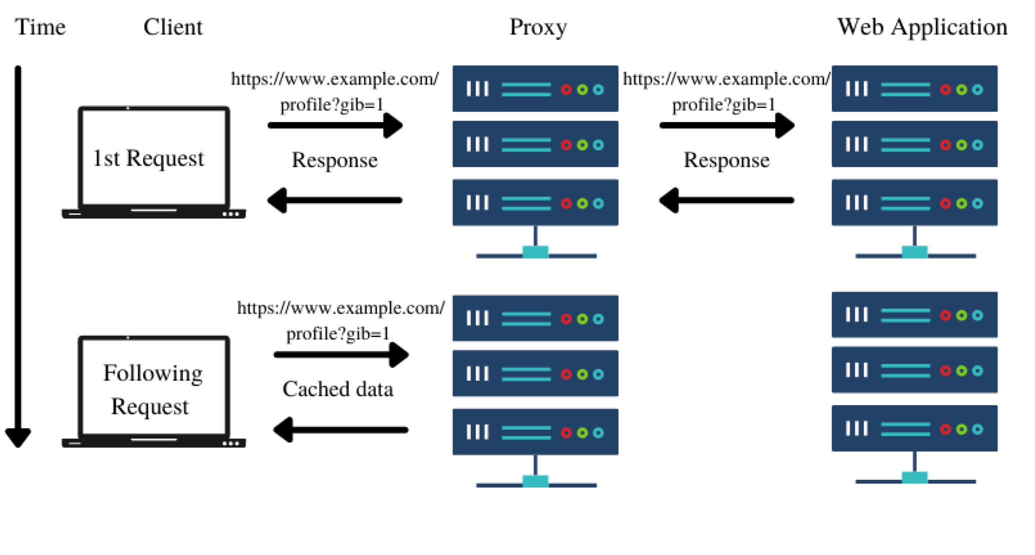

Time
Client
Proxy
Web Application
https://www.example.com/profile?gib=1
Response
1st Request
https://www.example.com/profile?gib=1
Response
https://www.example.com/profile?gib=1
Following Request
Cached data

Figure 2.3: Sample traffic through proxy with caching enabled.

2.7 Proxy setup

The proxy cannot simply cache everything, this would overload the storage, and in turn make everything slower [13]. As a compromise the developer must create some rules for the caching, this often involves using some keys. Take logos for example, these does not need to be sent from the web application every time, these can be stored in the cache. If the logo is different from a desktop computer and mobile phone, the same logo can't be used. The logo from the desktop version will most likely be too big for the phone and will result in a bad looking website. The obvious solution here is to cache based on the User-Agent header, i.e. using the User-Agent header in the cache key.

Normally a proxy uses the scheme (e.g., https), host header, path, and query string to form a key for the cache. Most caches are then capable being bypassed by changing the query string, which seldom has an impact on the functionality of the web application itself.

As an example, a cache key can look like this: ”GET|https|www.example.com|/profile?gib=1”, where the cache key must match the new request for the cached response to be returned. The cache key may include any element of the request to form numerous different cache keys, all depending on the requirements of the web application.

Everything not included in the cache key is so-called unkeyed parameters, these are very interesting for finding vulnerabilities. The server may or may not display whether the response is cached or not, it may also display how long the cached response is valid. Displaying these headers is commonly the default configuration for multiple types of proxies, hiding these headers would be a good idea for security reasons. Some proxies may for example display an E-tag header, this is a header displaying a unique hash, this hash is calculated from the response which is cached. If this header value changes, the cached response has most likely expired. Note that you may see multiple proxies in a chain where there may be multiple caches set up with different rules.

Proxies may also display some Cache-Hit header or some variation of it, this simply tells the end user whether the response was cached or not. It will return a value of either ”HIT” or ”MISS”. The Last-modified header is a header returning the time of which the response was last updated in the cache [5]. There are also a lot of variations and other caching related headers, these may include different and sensitive information of how the cache works.

The information given by the proxy may be incorrect simply to mislead security professionals. The proxy may also be set up quite complex with multiple proxies behind it, it is most likely not possible to distinguish the proxies form each other. One proxy may be returning caching-related headers, while the others may or may not. This makes finding how to bypass the cache difficult and tedious.

Figure 2.4 shows how multiple proxies can be used behind each other, with different caching rules. Proxy 1 is set up with this cache key: ”$method$scheme$host$request$request_path$query_string”, Proxy 2 is set up with a similar key, but omits the query string: ”$method$scheme$host$request_path”. As an exmaple a key may look like ”GETHTTPSwww.google.com/testgib=1” in Proxy 1. For the same request Proxy 1 would have a key like this: ”GETHTTPSwww.google.com/test”. The difference between these keys is that Proxy 1 uses the query string, Proxy 2 does not. A valid cache key in Proxy 1, might not be valid in Proxy 2. This means that a request may miss the cache in proxy 1, while hitting the cache in proxy 2 if only the query string changes.

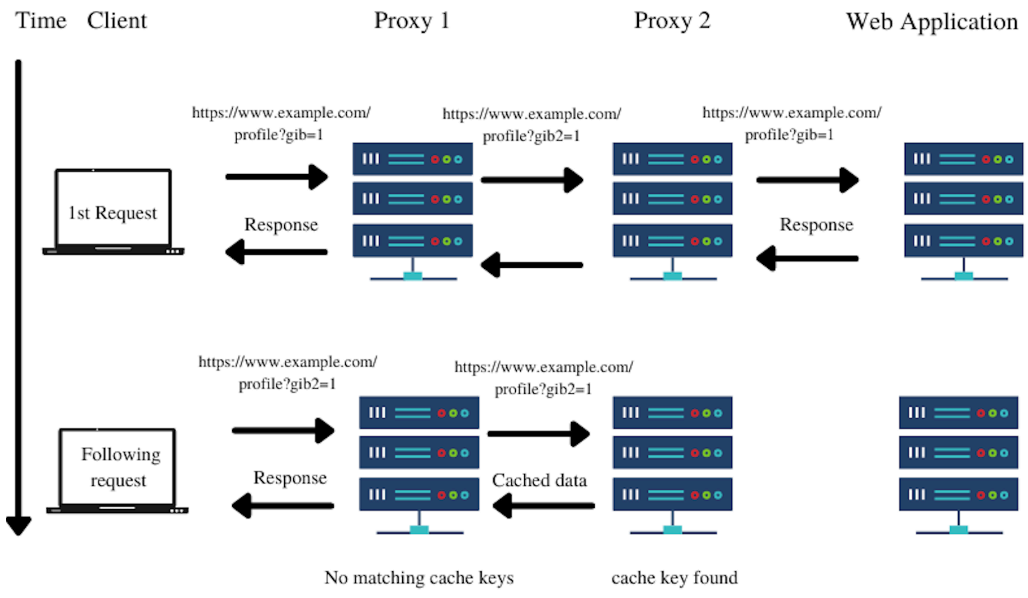

Time
Client
Proxy 1
Proxy 2
Web Application
https://www.example.com/ profile?gib=1
https://www.example.com/ profile?gib2=1
https://www.example.com/ profile?gib=1
1st Request
Response
Response
https://www.example.com/ profile?gib2=1
https://www.example.com/ profile?gib2=1
Following request
Response
Cached data
No matching cache keys
cache key found

Figure 2.4: Traffic going through multiple proxies with different cache keys.

2.8 Cache poisoning

The use of caching provides the end users a responsive, reliable, and fast web application. However, this performance enhancing technique opens for a new security threat called cache poisoning. Cache poisoning ends up in users being served malicious HTTP responses [14]. The objective for an adversary in this case, is to force the proxy to cache a malicious response, which is then served to other users. The adversary may achieve multiple things, all depending on the vulnerability found and the proxy configuration. This may vary from stealing sensitive information, to escalating other vulnerabilities.

To be able to manipulate the cache, knowledge about the cache key is essential. When looking closer on user inputs, the application may use part of the input to form a cache key. Keyed parameters are used by the cache to determine if there is an appropriate stored response for the request, or if it's necessary to forward the request to the web application. If the request is forwarded to the web application, the web application would need to create a new response.

For instance, if the adversary tricks the cache to cache a malicious response, the response may be served to other users. An adversary can then exploit this, where the attack can impact numerous users. The malicious cached response will be served to unknowing victims until the response expires.

However, this type of attack is rather difficult to execute, and there are multiple criteria that needs to be met. To be able to successfully exploit cache misconfigurations, information of how the proxy is set up is essential. Information regard cache keys and when the cached data expires is highly valuable. When combining the observed information with some critical functionality, or a vulnerability, the adversary may begin exploring exploitation of the cache. If everything aligns, the adversary can injection some form of payload into the cache, which negatively impacts other users.

Figure 2.5 shows how an adversary might go about poisoning the cache. The adversary sends a request containing an XSS payload in an unkeyed parameter. It can be assumed in this case that the payload is simply reflected and has an XSS vulnerability.

The vulnerability is now cached, meaning when User x visits the same page, using the same cache key, User x receives the payload from the proxy's cache. User y also receives the same payload from the cache when using the same cache key.

Practically poisoning the cache usually is rather difficult because the cache configuration is unknown. The proxy may use date or number of requests in multiple ways to determine which responses should be cached and how long the cache responses should be valid. In addition, the most frequently used cache keys could be used to determine which endpoints should be cached. Therefore, the adversary must observe the different behaviors of the request on the different target endpoints.

To successfully force the proxy to choose a tampered malicious request, the request must be sent after the previous response has expired, and before any other request is sent with the same cache key. This can be done by sending numerous requests, consequently ”flushing” the proxy for old data [14]. If the proxy hands out how long the cache is valid, it is also possible to simply time the request. Best case scenario is to only need two requests to poison the cache.

Cache poisoning can be used to retrieve sensitive information from other users in some cases [15]. If the proxy is setup in an insecure manner, it may be possible to trick the proxy to cache personal information on an endpoint available without

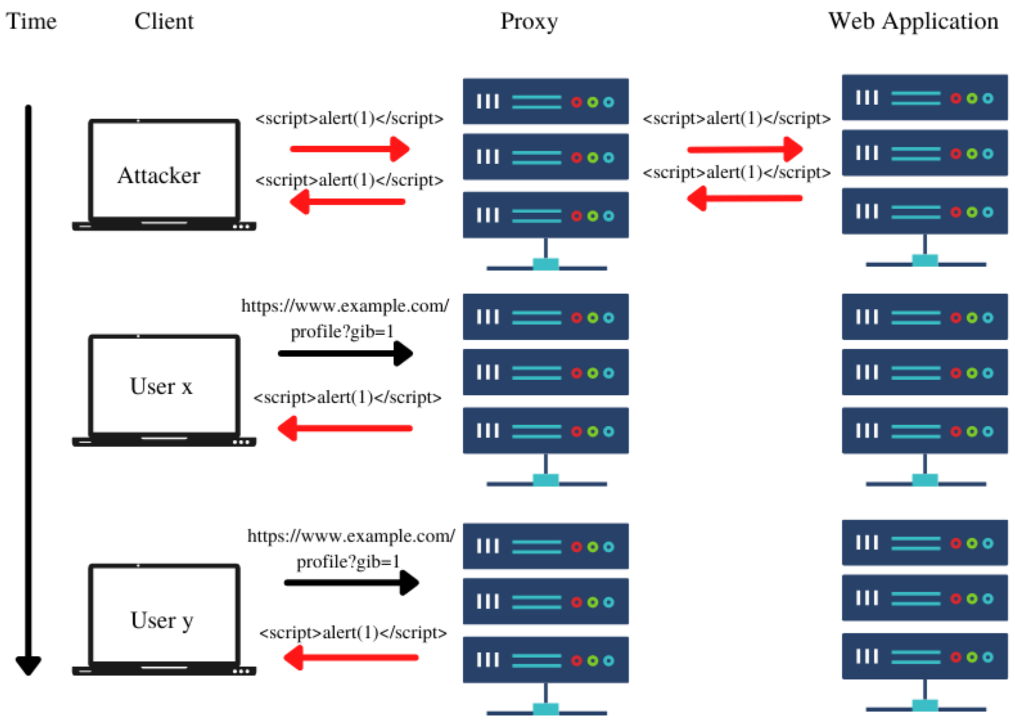

Time
Client
Proxy
Web Application
Attacker
<script>alert(1)</script>
<script>alert(1)</script>
<script>alert(1)</script>
<script>alert(1)</script>
User x
https://www.example.com/profile?gib=1
<script>alert(1)</script>
User y
https://www.example.com/profile?gib=1
<script>alert(1)</script>

Figure 2.5: Adversary manipulating cache to attack arbitrary users.

authorization.

In some cases, the proxy can be set up to cache everything that is static, this includes JavaScript, HTML and CSS files. If the proxy simply caches everything that ends in ".css", it may be possible to make a request to a sensitive endpoint while making the proxy cache it as a CSS file [16].

As an example, if making a request to "https://www.example.com/profile/test.css" the web application sees "/profile", displays the profile page. The proxy sees ".css", therefore caching the response. The sensitive profile page is now cached under a CSS file available for everyone to see. This is very simple to exploit, but very difficult to discover and is probably harder to exploit in the wild. An attack like this is called cache deception, where an adversary may trick the proxy to cache sensitive information from other users.

2.9 Pros and cons of caching

Caching responses has a huge advantage, namely performance. The cache will deliver responses faster than if the web application had to create it from scratch every time. But that is not the only advantage, the cache will give the web application less strain, meaning that the web application has lower hardware requirements. As a result, there is a reduction in cost of running the web application by using a proxy with caching enabled in front. If the cache is set up in a sophisticated manner, the cache may also hide vulnerabilities. In some cases, it is not advisable to use a custom configuration of proxies, as this is difficult to set up correctly and should be handled with care. On the contrary in some cases custom configurations are required for optimal security.

The disadvantages of using a cache are possible cache poisoning vulnerabilities and setup complexity. Whenever setting up caching, there is a risk of configuring the proxy incorrectly, leading to severe risks. The proxy is difficult to set up correctly, and even more so if the web application is complicated.

There are extra risks to consider when setting up caching, especially endpoints that include sensitive information. Harmless looking vulnerabilities pose a bigger threat when using a cache enabled proxy. All dynamic content should be reviewed before enabling caching. Harmless vulnerabilities such as self-XSS (a vulnerability only affecting the adversary) are suddenly critical vulnerabilities affecting everyone. Caching can be seen as a booster of vulnerabilities; any vulnerability is usually a lot worse whenever combined with cache poisoning. Another disadvantage of using caching is that the if the content of the application were to change, the cache would need time to update its cached contents. Depending on the configuration, this could take second, minutes, hours or even months. If the cache takes too long to refresh, the cache must be manually cleared.

2.10 PortSwigger

PortSwigger is an web security organization, who mainly focuses on developing security tools like Burp Suite, which is a suite of tools in one program [17]. PortSwigger also host educational content to improve peoples awareness when it comes to web application security.

2.11 Penetration testing

Penetration testing could simply be explained like a simulation of a real-world attack, where the customer gets a report of the vulnerabilities found, instead of the vulnerabilities being exploited for malicious purposes [18]. This method allows for a modernized way of checking for vulnerabilities and deficient source code. Penetration testing is supposed to show all the flaws made during development, and how to optimally improve them. For this purpose, penetration testers must be familiar with, and master advanced hacking methods. A penetration tester also must be very methodical, as any oversight could be disastrous. For an adversary, any vulnerability is considered a win, for a penetration tester any vulnerability is considered a failure. This means that penetration testers must succeed 100% of the time, while adversaries only need to succeed once.

2.12 Bug bounty

Bug bounty programs are very similar to penetration testing, the difference is that these programs only pay for what researchers find [19]. On the other hand, penetration testers are paid the same amount regardless of the number of issues. Bug bounty programs is a way to allow researchers to freely test an application. Researchers are then expected to deliver any security issues to the respective company. If the issue is deemed critical enough, the researcher will be given some kind of reward for their efforts. Bug bounty is great for people that want to find security issues, either as a hobby or as a freelancer. These programs allow skilled people to use their skills in a legal manner, instead of conducting illegal acts.

Chapter 3

Demonstration

3.1 theeCB demonstration

In this section, we will give a demonstration of the functionality of theeCB. The demonstration is presented in this section by explanations and screenshots.

As a demo we will be testing a self-made web application for a Cross Site-Scripting vulnerability. This is an intentionally vulnerable application. The end-point on the application we will be testing has the source code shown in code snippet3.1

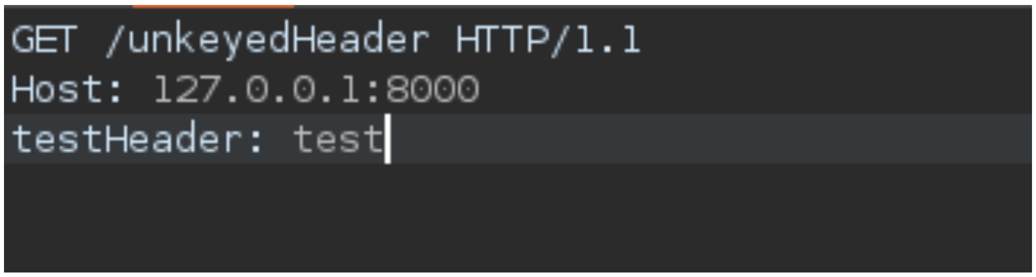
```python
@app.route("/unkeyedHeader")
def unkeyedHeader():
    return request.headers.get("testHeader")
```

Code snippet 3.1: Simple python web application endpoint.

The source code is quite simple, it returns the value of the header ”testHeader” back in the response. An obvious flaw is that the application doesn't validate or sanitize the input from the user before returning the data. An attacker can then run arbitrary JavaScript code due to the nature of the code. The only issue for the attacker is that he can't set the testHeader value for other users, he would have to have another vulnerability or make other users change it manually. That's not much of a dangerous vulnerability on its own. Let's now first look at how a vulnerability scanner would handle this.

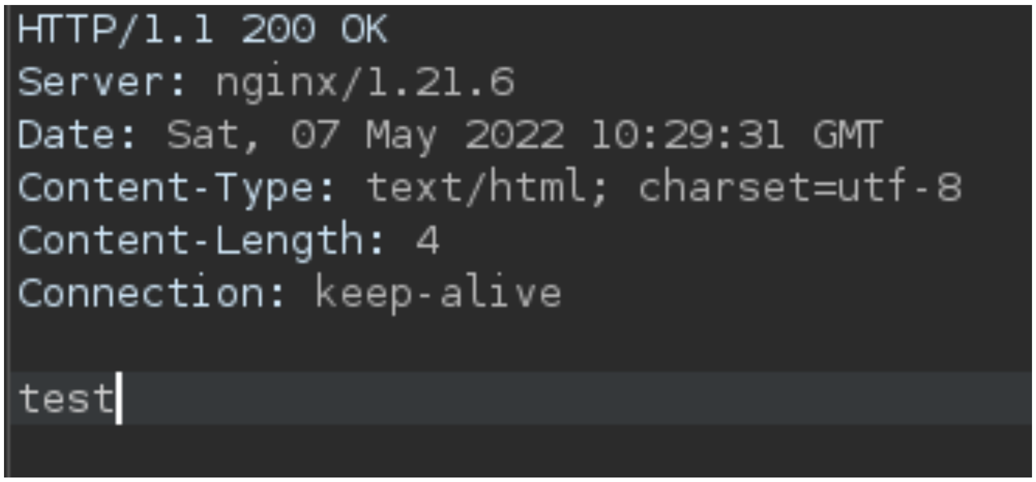
In figure 3.1 and 3.2 you can see a normal request to the test application, and its default response.

In theeCB, we modified the request. As shown in figure 3.3 the ”testheader”



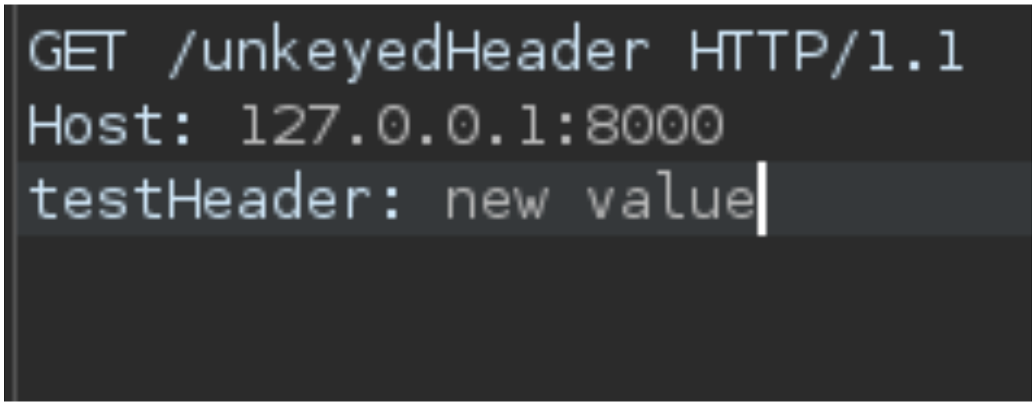
```
GET /unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader: test
```

Figure 3.1: Default request to the web application endpoint.



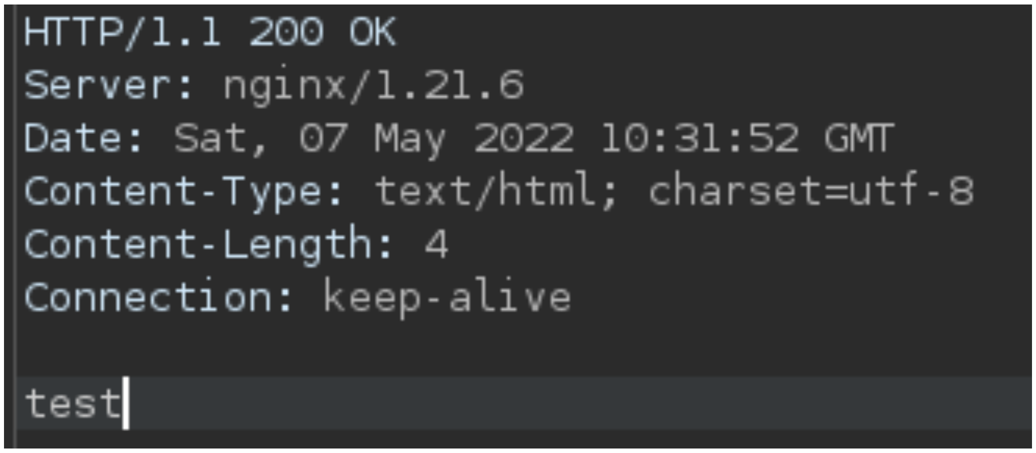
```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:29:31 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 4
Connection: keep-alive

test
```

Figure 3.2: Default response from the web application.



```
GET /unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader: new value
```

Figure 3.3: Changing the testHeader header value.



```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:31:52 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 4
Connection: keep-alive

test
```

Figure 3.4: The response is cached, and have therefore not changed.

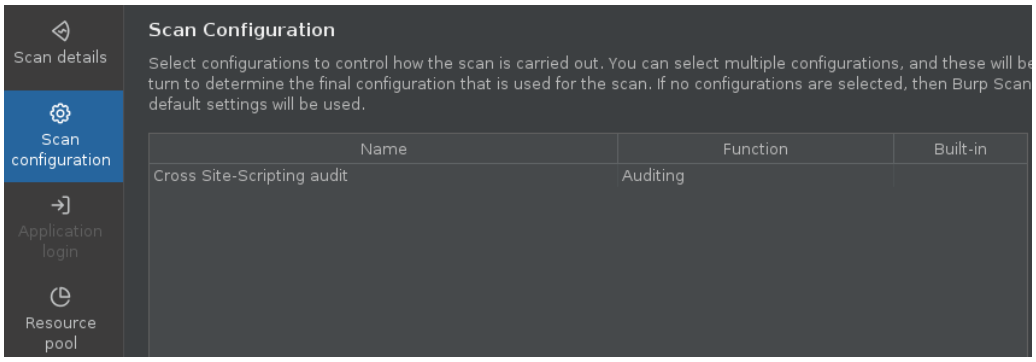
header value has been changed for a new value.

Since a caching mechanism is in use the response has not been changed, as you can see in figure 3.4.

Furthermore, a vulnerability scan within Burp Suite was set up and used. This scan was set up to detect everything related to XSS and caching related vulnerabilities as you can see in figure 3.5.

In figure 3.6 you can see the vulnerability scanner searching through the test application with various payloads.

As a result of the scan, Burp Suite's issue activity logger did not find any vulnerabilities, as shown in figure 3.7.



Scan Configuration

Select configurations to control how the scan is carried out. You can select multiple configurations, and these will be
turn to determine the final configuration that is used for the scan. If no configurations are selected, then Burp Scan
default settings will be used.

Name	Function	Built-in
Cross Site-Scripting audit	Auditing	

Figure 3.5: Setting up Burp Suite to do a vulnerability scan.

When running theeCB, it found two ways of bypassing the cache on this particular endpoint, shown in figure 3.8 . However, in this demonstration we will be using the second way, by appending a slash (/) after the first slash in the path.

In figure 3.9 you can see a modified request, modified according to the results of theeCB, for the purposes of bypassing the cache. Each path can only be used once to bypass the cache, as the cache will cache the new response with the current path within the cache key. The cache detection tool uses up the path using two slashes, therefore in this request there are three slashes in the path.

Furthermore, the cache was indeed bypassed as seen in figure 3.10, the new ”testHeader” header value is reflected in the response, showing that the old cached response was not used by the proxy.

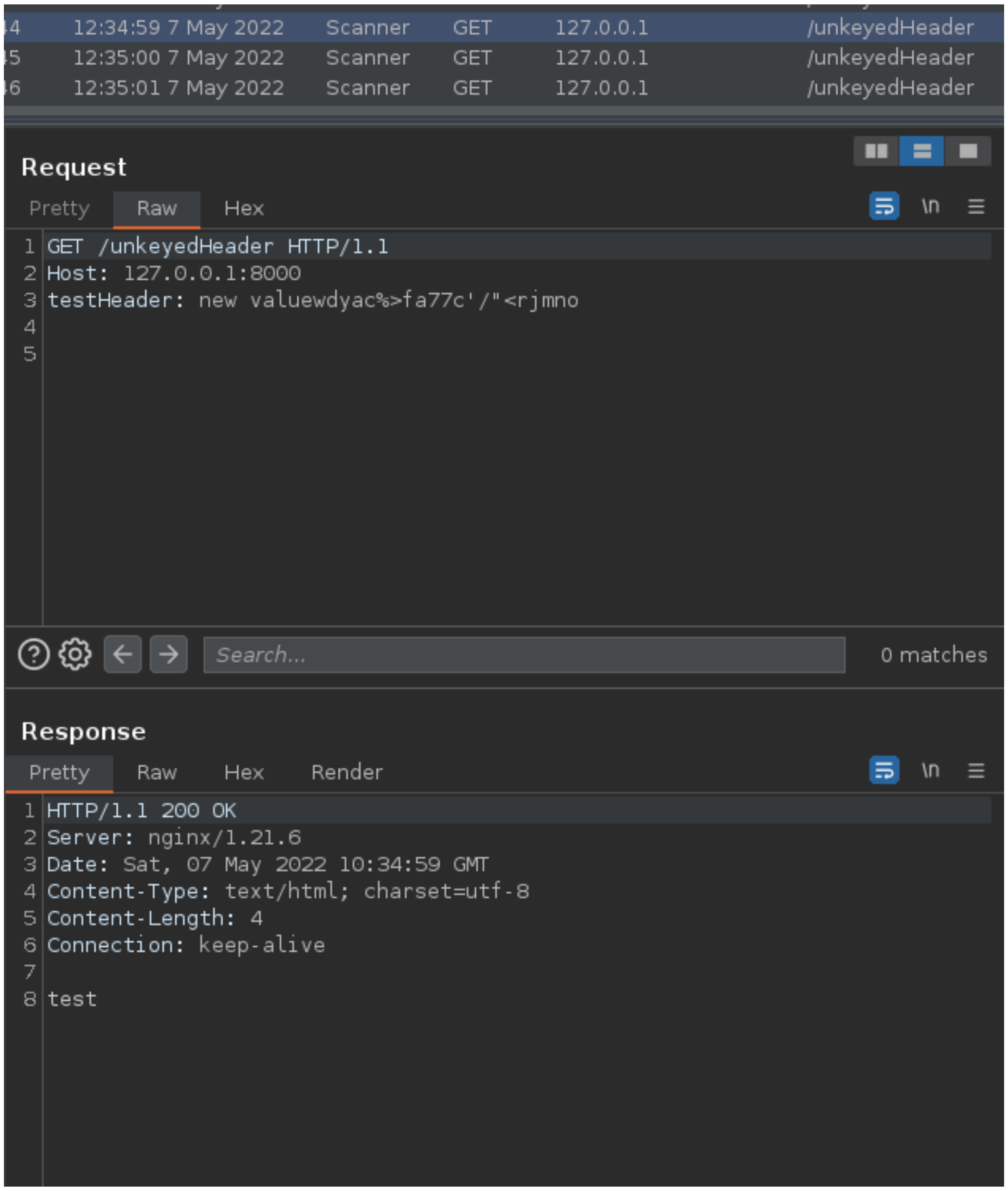
In Figure 3.11 you can see a malicious payload being inserted into the ”testHeader” header. Another slash is also added to the path to again bypass the cache.

Nextly, in figure 3.12 shows that the malicious payload is reflected in the response, once again bypassing the cache. There was also proved to be a XSS vulnerability, it is possible to make the web application return arbitrary JavaScript code.

There was also a new request with the ”testHeader” header removed, shown in figure 3.13. This is to check if the malicious payload is still cached if the header is removed.

When the payload is now cached by the proxy, it will affect every user going to the same endpoint, this is shown in figure 3.14.

As an example we made the web application store a malicious payload in the ”/unkeyedHeader” path, using the same path as in 3.1. This results in a normal



```
4   12:34:59 7 May 2022   Scanner   GET   127.0.0.1          /unkeyedHeader
5   12:35:00 7 May 2022   Scanner   GET   127.0.0.1          /unkeyedHeader
6   12:35:01 7 May 2022   Scanner   GET   127.0.0.1          /unkeyedHeader
```

Request

```
GET /unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader: new valuewdyac%>fa77c'/"<rjmno


```

Response

```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:34:59 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 4
Connection: keep-alive

test
```

Figure 3.6: A screenshot of the vulnerability scanner running.

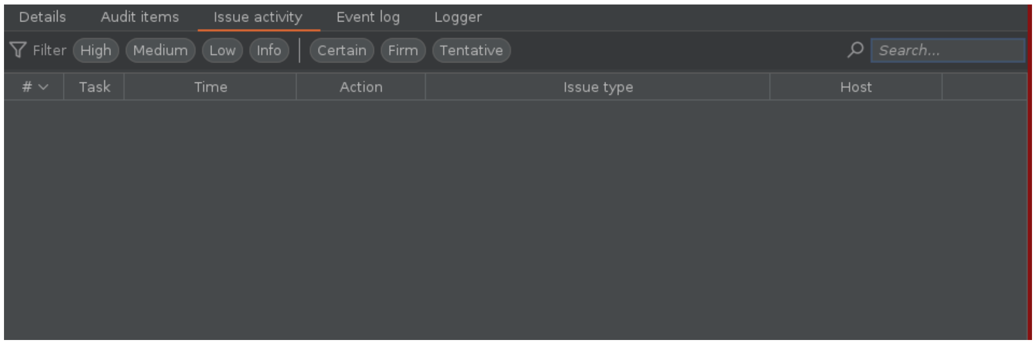
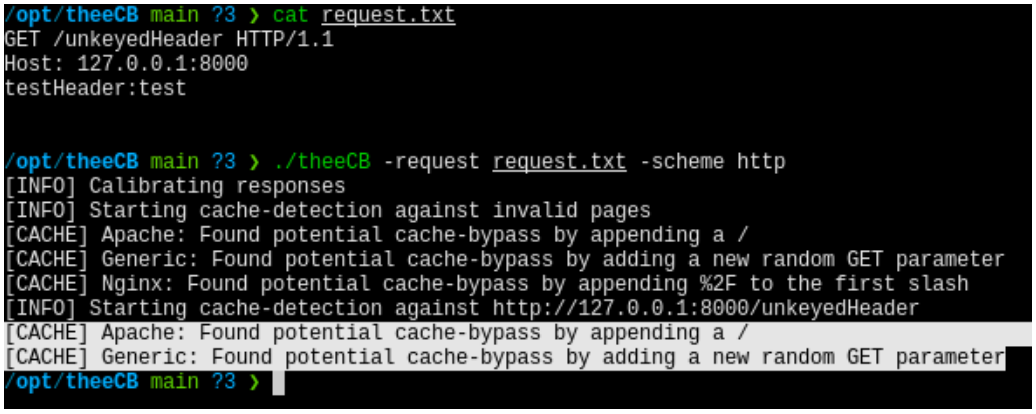
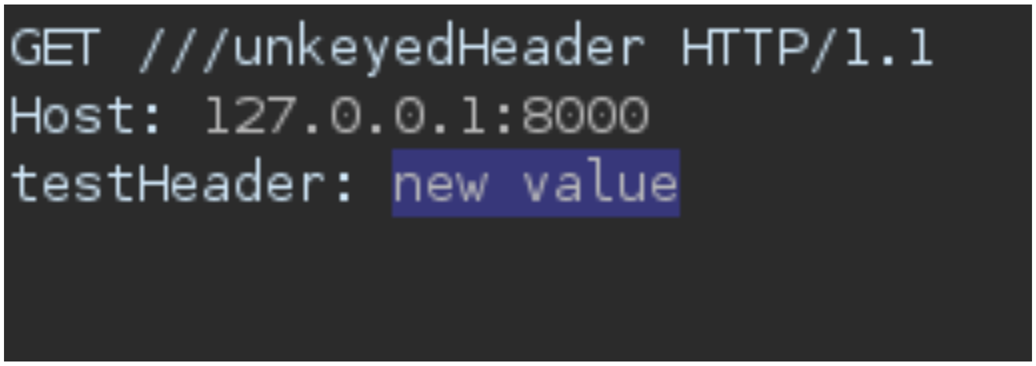


Figure 3.7: Burp Suite's issue activity is empty, meaning no issues found.



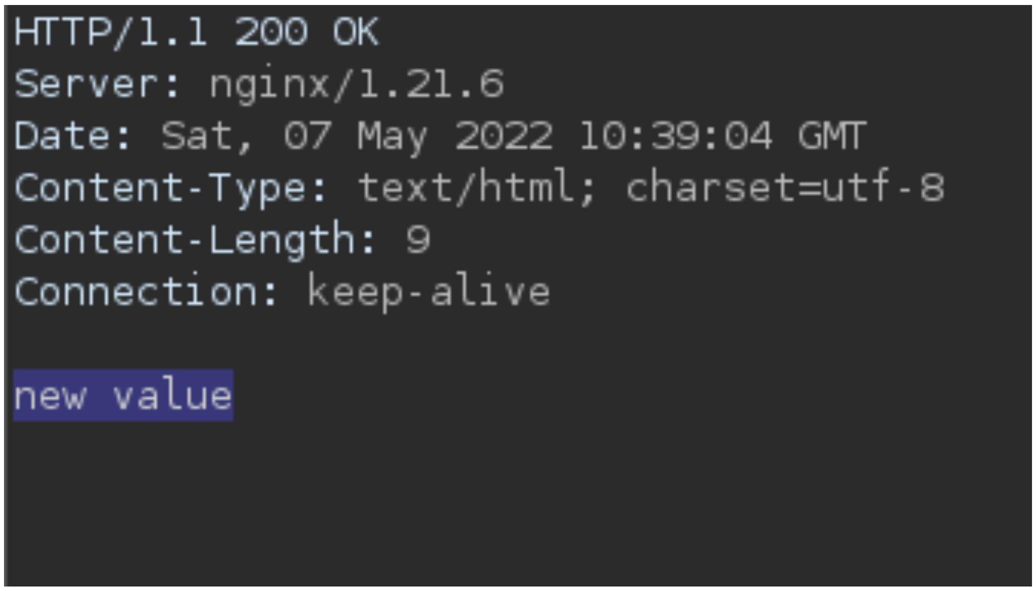
```
/opt/theeCB main ?3 ❯ cat request.txt
GET /unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader:test


/opt/theeCB main ?3 ❯ ./theeCB -request request.txt -scheme http
[INFO] Calibrating responses
[INFO] Starting cache-detection against invalid pages
[CACHE] Apache: Found potential cache-bypass by appending a /
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
[CACHE] Nginx: Found potential cache-bypass by appending %2F to the first slash
[INFO] Starting cache-detection against http://127.0.0.1:8000/unkeyedHeader
[CACHE] Apache: Found potential cache-bypass by appending a /
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
/opt/theeCB main ?3 ❯
```

Figure 3.8: The cache detection tool finding ways to bypass the cache mechanism.



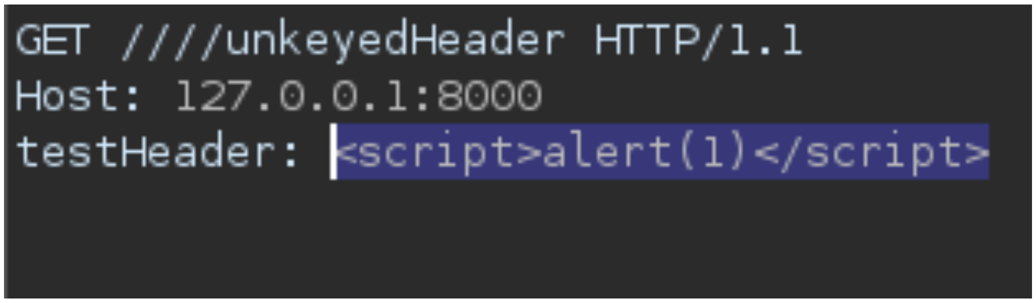
```
GET ///unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader: new value
```

Figure 3.9: Adding two slashes(/) to the path, as our tool suggested.



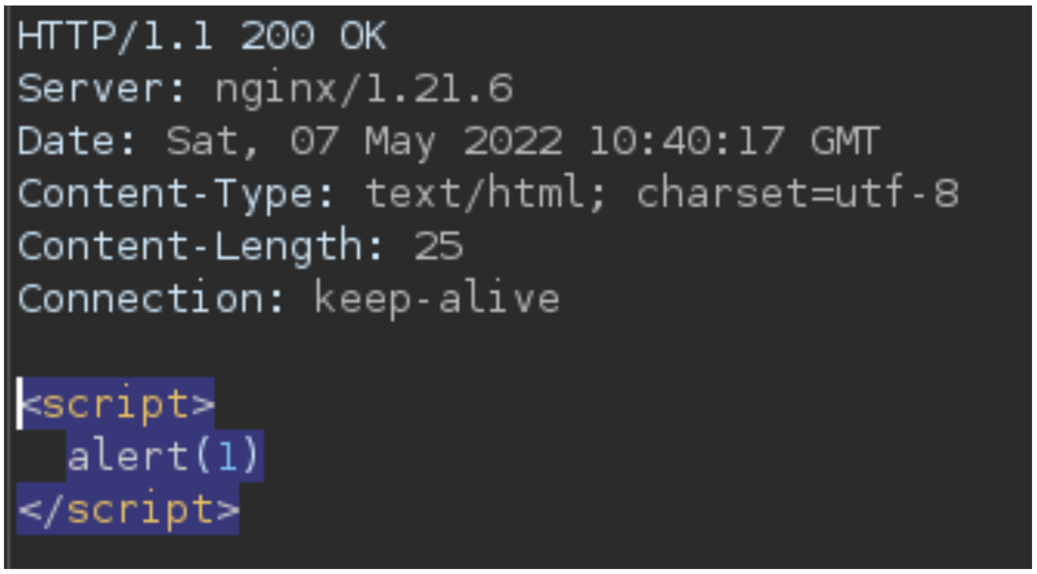
```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:39:04 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 9
Connection: keep-alive

new value
```

Figure 3.10: The modified value is reflected, the cache was bypassed.



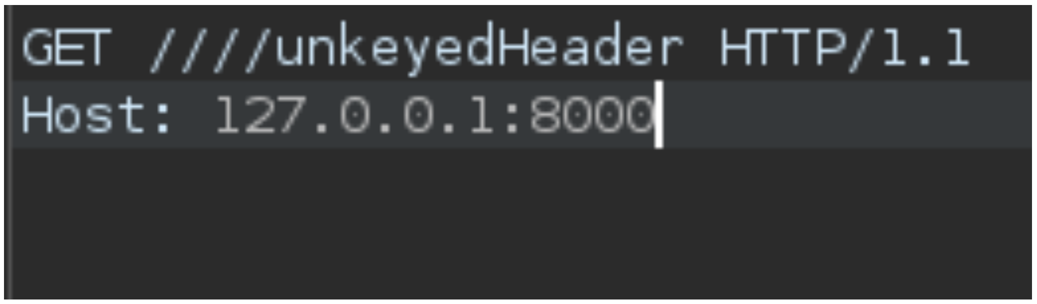
```
GET ////unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
testHeader: <script>alert(1)</script>
```

Figure 3.11: Adding a malicious payload to the request.



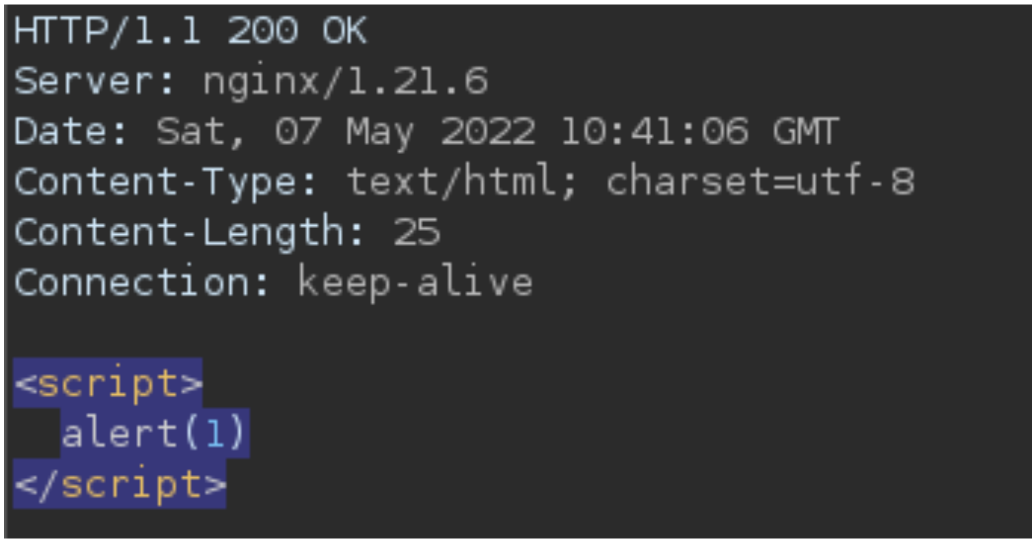
```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:40:17 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 25
Connection: keep-alive

<script>
  alert(1)
</script>
```

Figure 3.12: The payload was reflected.



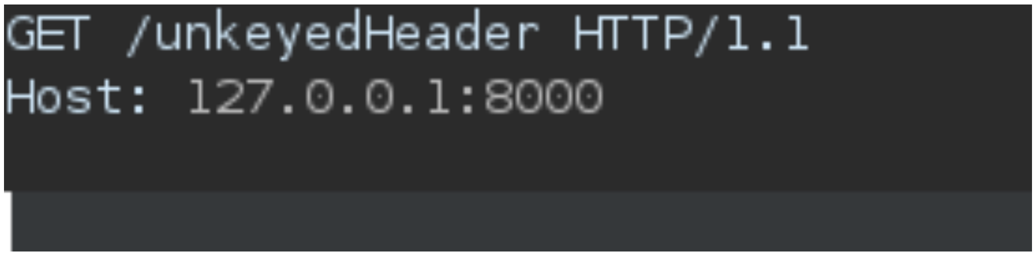
```
GET ////unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
```

Figure 3.13: Remove the payload to check if the vulnerability is persistent.



```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:41:06 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 25
Connection: keep-alive

<script>
  alert(1)
</script>
```

Figure 3.14: The malicious payload is still reflected, meaning the payload is stored in the cache.



```
GET /unkeyedHeader HTTP/1.1
Host: 127.0.0.1:8000
```

Figure 3.15: After a successful attack, an end-user goes to the test application like normal.

cache key being used, and this attack would affect every user going to this end-point.

Figure 3.15 shows a new and normal request being sent to the application.

The victim is then served with a malicious payload, without performing any unusual behavior, shown in figure 3.16.

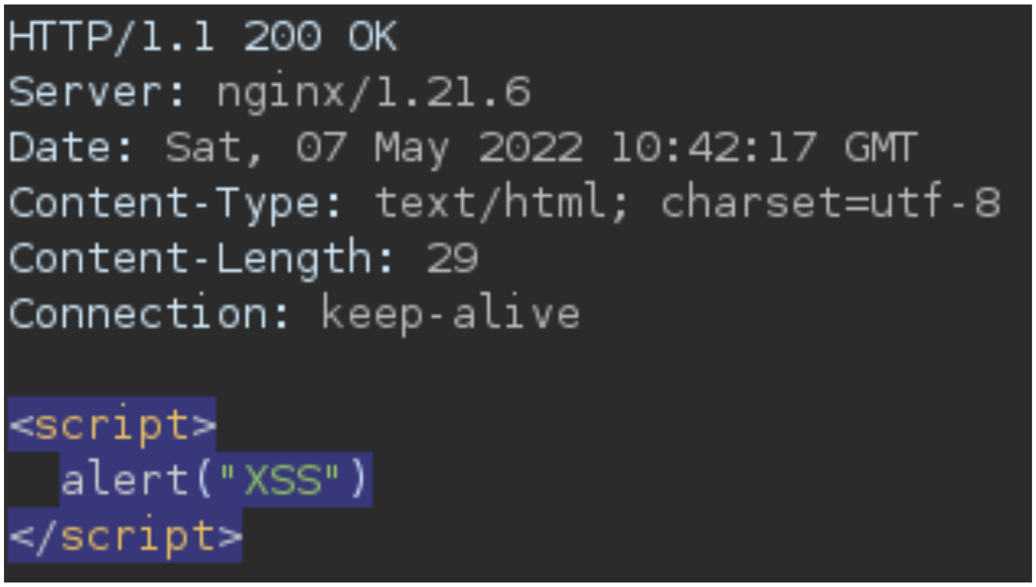

```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Sat, 07 May 2022 10:42:17 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 29
Connection: keep-alive

<script>
  alert("XSS")
</script>
```

Figure 3.16: The user receives the malicious payload.

Chapter 4

Model

4.1 Fingerprinter

Our code is divided into multiple sections to make the code more modular, making it easier to extend or re-use code. The fingerprinter measures differences and similarities in responses. Therefore, the fingerprinter must first find a baseline. It does this by sending multiple requests and filtering out dynamic content. By ignoring any dynamic content, it is easier to find essential differences in the responses.

4.2 Overview

theeCB is highly dependent on the fingerprinter module we developed and designed. The fingerprinter's most significant feature is determining the expected response time for uncached content and comparing it to new responses. The fingerprinter will detect whether the new response is cached based on the response time.

If the response headers are similar to the previously seen headers, the fingerprinter determines that the response is cached. Lastly, the fingerprinter also parses XML/HTML data. HTML is very commonly used in HTTP responses, and the fingerprinter will detect if the HTML document is changed in various ways. The fingerprinter is only looking at the responses and determining differences or similarities. We also developed a cacheProxies module. This module is responsible for creating the various requests that are sent. In the cacheProxies, it

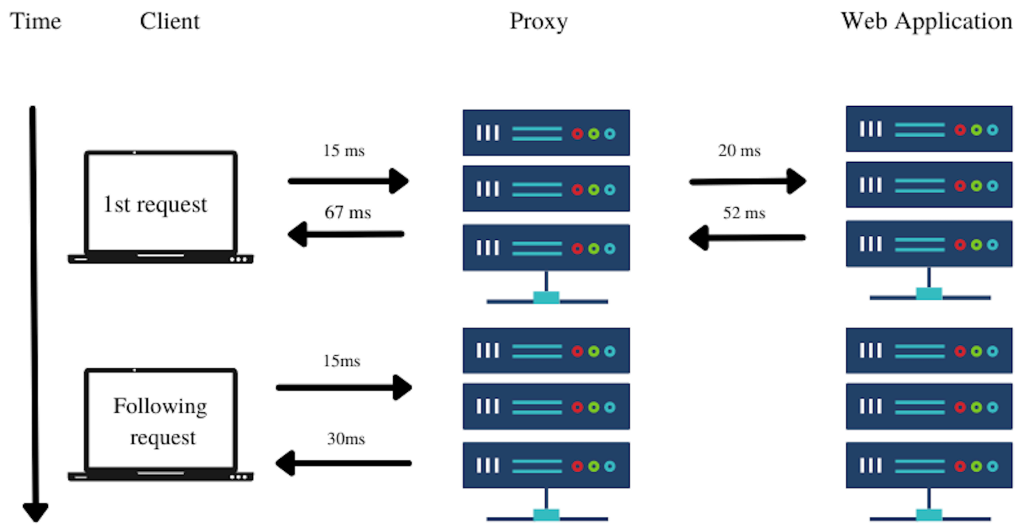


Figure 4.1: Figure showing the efficiency increase by using a cache.

is possible to create payloads for specific proxies. In theeCB, we implemented a few proxies, such as Generic, Apache, Nginx, and Ruby. Each of these implementations has a list of payloads sent sequentially.

4.3 Timing

When navigating through websites in the browser, the browser sends some requests and receives some responses. To provide a smooth browsing experience, the website needs to send responses as quickly as possible after a request is sent. The response time depends on multiple factors, such as the travel length, hardware on the server which runs the web application. An easy way to reduce response times is to set up a proxy which caches frequently used data. After implementing caching, the web application will respond a lot quicker when pages are visited multiple times. Response times are therefore essential for theeCB.

The fact that the developers try to make the response times as short as possible, makes it easier to create a cache detection tool. Consequently, it gives us the possibility to detect whether a response is cached or not, purely based on the response times. theeCB is created with this in mind, therefore the fingerprinter module first performs a calibration.

Response times are not reliable as there are many factors and ways for the data to be delayed. This means that a single reading may be entirely off, this is easily detectable by a human, but not by a program. Therefore, a range of the seen response times will be stored during the calibration phase. This range can be for example 1.2 seconds to 5.3 seconds. If theeCB detects a response within the calibrated scope, the response is considered uncached. For instance, if a response is delivered much earlier than the previous lowest value, the response is most likely cached. We currently don't have any better solutions to this problem, and theeCB is relying on consistent results. This might result in some false positives; however, this can be solved by running theeCB multiple times.

Measuring the response times of the request till the response is the most efficient way of determining whether the response is cached or not, everything else added is added to minimize false negatives.

Humans can easily detect spikes in response times, and remove the invalid results, this then makes for a highly efficient way of determining how the proxy's cache is set up. However, this is very difficult to automate since the program can't tell which response times are clearly invalid.

If the response time for a response is delayed for some reason during the calibration, the results of theeCB is most likely useless. On the other hand, if the response time for a response is delayed after the calibration and during the scan, a singular false positive may arise. An easy solution to this problem is to display the calibration data to the user. The user then must make a sensible decision whether to trust the result or not.

Another side of timing is the delay between the requests. If the ten requests in the calibration phase were to be sent too quickly, the web application might not have been given the time to change dynamic content. Certain headers may be changed in intervals, such as a Date header. The Date header may for example be changed each minute. Therefore, the calibration phase must last longer than one minute to remove the Date header. In this case, if the Date header doesn't update before the calibration phase is done, all the results will be incorrect. The solution to this is to implement a delay between each request, this is an optional argument to theeCB with the default value of zero seconds.

Chapter 5

Implementation

5.1 Interfaces

A graphical user interface, more commonly known as a GUI, is an interface where a user interacts with visual elements such as icons and menus. Most people are using graphical user interfaces daily. A GUI allows for intuitive navigation in programs and is universally designed. GUI programs are often more memory and CPU intensive.

Command line interface, or CLI, is an interface for use in the terminal. CLI is designed to be used with the keyboard, with minimal mouse interaction. Tools using CLI are usually using less memory and CPU power. The output of CLI tools can easily be used as an input to other tools, meaning it is possible to create a chain of tools programmatically.

While both the CLI and GUI have their advantages and disadvantages, the choice of which to choose is only preference. We chose to use a command line interface for theeCB. There are many CLI tools that returns URLs as their output, by building a CLI tool, it is often easier to use the output of these tools as the input to theeCB. For someone who knows how to use a terminal, CLI based tools can be a lot more efficient to use. CLI tools allows to chain multiple tools, using the output of tool 1 as the input to tool 2. This means that we can use some other tool to discover URLs and send these URLs to theeCB without much hassle.

We chose a CLI because it has less CPU and memory consumption, and because of the tool chaining possibilities. The tool chaining possibilities are practical, as many CLI tools output in the form of URLs.

5.2 CacheProxy

When developing theeCB, we wanted to be able to discover methods to bypass the cache, therefore we implemented the CacheProxy interface giving the ability to implement various payloads. This results in a more diverse and feature rich tool. The CacheProxy interface specifies the functions a struct needs to implement to be compliant with the interface, shown in code snippet 5.1.

Init is the function that needs to be run before the scanner starts, this function adds the appropriate functions to a list in the struct. The Name, CurrentPayload and PayloadLength functions will return a simple value, the name of the proxy, the current payload it is at and the total length of payloads respectively. The Reset function simply resets the current payload variable, which makes it ready to be re-used. NextRequest is the function that executes the payloads, it will modify the base request, send it, and compare the new response to the baseline Response struct.

```go
type CacheProxy interface {
    NextRequest(req *http.Request) bool
    Name() string
    Init() error
    CurrentPayload() int
    PayloadLength() int
    Reset()
}
```

Code snippet 5.1: The CacheProxy interface.

As an example for the implementation of the CacheProxy interface, the code for the Apache struct is shown in code snippet 5.2. The payloads variable is a slice of functions, these are the payloads that will modify the requests. The currentPayload variable specifies the current payload and is used as an index in the payload's variable.

```go
type Apache struct {
    currentPayload int
    payloads       []func(req *http.Request) error
    Opt            *Options
}
```

Code snippet 5.2: Apache struct

Code snippet 5.3 shows a simple payload that will be used with the Apache struct. This function clones the original request, appends a slash after the first slash in the path. The request is then sent, and then the response is compared to the calibrated Response struct. Adding a slash in this manner is a technique to try to avoid the cache.

```go
func (p *Apache) addSlash(req *http.Request) error {
    reqClone := req.Clone(p.Opt.Ctx)
    path := reqClone.URL.Path
    path = path[:1] + "/" + path[1:]
    reqClone.URL.Path = path
    resp, duration, err := p.Opt.sendRequest(reqClone)
    if err != nil {
        return err
    }
    p.Opt.checkResponse(resp, duration, p.Name(), "appending a /")
    return nil
}
```

Code snippet 5.3: Sample payload used for the Apache struct.

In code snippet 5.4 the payloads variable is populated. In this case only the addSlash function is added.

```go
func (p *Apache) Init() error {
    p.payloads = []func(req *http.Request) error{}
    p.payloads = append(p.payloads, p.addSlash)
    return nil
}
```

Code snippet 5.4: Init function.

Code snippet 5.5 shows the NextRequest function, this function is called before every time a new payload is sent. This function finds the current payload index, increases the index, and executes the function at the current payload index within the payloads slice.

```go
func (p *Apache) NextRequest(req *http.Request) bool {
    currentPayload := p.CurrentPayload()
    p.incCurrentPayload()
    if currentPayload >= len(p.payloads) {
        return true
    }
    p.payloads[currentPayload](req)
    return false
}
```

Code snippet 5.5: NextRequest function.

5.3 Execution flow

theeCB is set up to check for caching both on invalid and specific pages. An invalid page is an endpoint that is assumably non-existent on the web application. A specific page is an endpoint that includes some functionality that can't be found on invalid pages. The easiest way to check for caching is to check against invalid pages. By changing the path on each request, almost all web applications give an uncached response. The results then become more consistent. If invalid pages are set up not to cache any responses, this step is useless and won't return any useful information. Another disadvantage is that some specific pages may include custom cache configurations, which the tool won't pick up on when scanning against invalid pages.

Checking against specific pages gives more information about caching rules set up for particular endpoints. These caching rules are harder to discover and may uncover vulnerabilities. theeCB relies on bypassing the cache during the calibration; however, it can't be certain that it managed to avoid the cache. The scan against invalid pages then gives reliable information, and the scan against any specific page may or may not work as intended. The scans against specific pages works for most web applications and results in better information than the scan against invalid pages.

5.4 Fingerprinter

The fingerprinter must first find a baseline. It does this by sending multiple requests and filtering out dynamic content. For the calibration phase, a Response struct is used. This struct holds all calibrated data and information about what data in the responses are considered static.

```
type Response struct {
        disabledModules *disabledModules
        responseCode    int
        pageTitle       string
        headers         http.Header
        tags            []*tag
        regex           map[*regexp.Regexp]int
        responseTime    []time.Duration
        verbose         bool
}
```

Code snippet 5.6: Response struct

Verbose is used to specify if verbose messages should be returned, shown in code snippet 5.7 Next regex is used to specify a regex string to search for, and to count the instances of the results. Additionally, disabledModules determines here which modules (e.g., headers) should be disabled. If the responseCode boolean is true, theeCB will overlook any changes in the response codes. pageTitle can be disabled in the same way, as well as any specific header, tag, or regex. The disabledModules struct is defined in code snippet 5.7.

```
type disabledModules struct {
    responseCode bool
    pageTitle    bool
    headers      map[string]bool
    tags         map[string]bool
    regex        map[*regexp.Regexp]bool
}
```

Code snippet 5.7: disabledModules struct

If any of the boolean values in the disabledModules is true, theeCB will overlook the results, in any case. In the CreateResponse function, theeCB receives the needed information about the response, and creates an appropriate Response struct. Most of the features in the fingerprinter is redundant in most cases. Looking at response times is commonly more than enough. However, when the response times can be accurately differentiated, response times can't be used. For instance, multiple proxies may be chained together, where the last proxy (from the end user) may be very close to the web application. The difference in cached and uncached responses from the last proxy, may therefore be negligible. To figure out which responses are cached, the fingerprinter must look at other elements of the response. Because of this uncertainty we created a fingerprinter which will record multiple responses and remove all dynamic content, it will compare response codes, headers, HTML tags and response times. The idea behind the fingerprinter is that it will detect the smallest of differences which not even a human can do reliably. Although the fingerprinter does not look at the entire response, it will detect the most of changes.

The calibration phase is critical to create a proper baseline. If any changes are detected during the calibration phase, the coherent module (with appropriate values) will be disabled. For instance, the checkResponseCodes function will compare the current calibrated response code with the new response code. If the new response code is different, the responseCode in the disabledModules is set to true. The checkResponse code is shown in code snippet 5.8.

```go
func (resp *Response) checkResponseCodes(rCode2 int) {
    if resp.responseCode != rCode2 && !resp.disabledModules.
    responseCode {
        resp.disabledModules.responseCode = true
        if resp.verbose {
            log.Println("Not using response code.")
        }
    }
}
```

Code snippet 5.8: Function checkResponseCodes

Most of the ”check” functions are written in a similar manner, some are more complicated because the values are not only integers. For instance, checkHeaders requires all the header names and values to be checked. checkHeaders then becomes larger than checkResponseCodes, and then looks like the code shown in code snippet 5.9.

```go
func (resp *Response) checkHeaders(h2 http.Header) {
    for k, v := range resp.headers {
        if v2, ok := resp.disabledModules.headers[k]; ok && v2 {
            continue
        }
        v2, ok := h2[k]
        if !ok {
            resp.disabledModules.headers[k] = true
            if resp.verbose {
                log.Println("Not using header: ", k)
            }
        } else {
            for i := 0; i < len(v); i++ {
                v3 := v[i]
                if len(v2) <= i {
                    resp.disabledModules.headers[k] = true
                    if resp.verbose {
                        log.Println("Not using header: ", k)
                    }
                    continue
                }
                v4 := v2[i]
                if v3 != v4 {
                    resp.disabledModules.headers[k] = true
                    if resp.verbose {
                        log.Println("Not using header: ", k)
                    }
                }
            }
        }
    }
}
```

Code snippet 5.9: Function checkHeaders

The nested loops are required, because Golang stores the values of duplicate header names in a list. When comparing XML data, theeCB must be able to parse XML (HTML). For this purpose, a feature to parse XML was added. The most important part of the parser is the parseXML function, shown in code snippet 5.10.

```
func (resp *Response) parseXML(bodyString string) bool {

    // variable initialization

    for i := 0; i < len(bodyString); i++ {
        if escaped {
            escaped = false
            continue
        }
        char = strings.TrimSpace(string(bodyString[i]))
        if i == 0 {
            if char != "<" {
                return false
            }
        }
        if char == "\\" {
            escaped = true
            continue
        }
        insideString=checkInsideString(char, insideString)
        if insideString != "" {
            continue
        }
        name, started, nameFinished = resp.checkCharString(char, name,
    started, nameFinished)
    }
    if started && name != "" {
        resp.addTag(name, 0.5)
    }
    return true
}
```

Code snippet 5.10: Function parseXML

This function will go through each character in the XML document and determine if the current character will open a new string, start a new tag, or escape the next character. The checkInsideString will check if the current character is a quote and update the insideString variable to an appropriate value, either an empty string or a single quote or a double quote. checkCharString is responsible for checking if a new tag has opened or closed, as well to determine if the tag name is a valid name. In the end the parser will find every tag in the document and count the number of each tag found. This parser will not store the values, as this may be very large.

The last part of the fingerprinter is the comparing of the baseline responses and a new response. These functions under ”compare.go”, which are responsible for comparing new values against the calibrated values, are similar to the calibration part. compareResponseCodes does the same thing as checkResponseCodes, except that it does not update the baseline Response struct if differences occur. compareResponseCodes is shown in code snippet 5.11.

```
func (resp *Response) compareResponseCodes(rCode2 int) bool {
    if resp.responseCode != rCode2 && !resp.disabledModules.
    responseCode {
        if resp.verbose {
            log.Println(fmt.Sprintf("Previous response code: %v, new
    response code: %v", resp.responseCode, rCode2))
        }
        return false
    }
    return true
}
```

Code snippet 5.11: Function compareResponseCodes

After the calibration phase is finished, the fingerprinter is looking for similar response times to the calibration phase. The fingerprinter should look for similar response times since the fingerprinter tried to avoid the cache in the calibration phase. If the response time is significantly faster than in the calibration it means that the response is probably cached, and the payload failed.

However, when looking over other differences (e.g. headers), the fingerprinter should look for differences. If any changes are detected, it probably means that this is a new response, and not the old cached one.

Chapter 6

Tests

6.1 Docker

Docker is a simple way to create a semi-isolated machine that is running for one specific purpose, similar to a virtual machine [20]. It is easy to separate projects, which may require different dependencies. Creating a docker image makes it possible to separate and re-use projects without library conflicts. We created a docker image for each component in our test application, making it very easy to deploy the application on any computer.

6.2 Digital Ocean

Digital Ocean provides droplets that can be rented for a low fee [21]. A droplet is a virtual machine that can be adjusted by the individual's needs. This fee is payed hourly and can be cancelled at any time. Therefore, Digital Ocean is perfect for our scenario, where we want to quickly make some tests against a public test application. We used a droplet from Digital Ocean to host our test application and ran various tests to compare against theeCB.

6.3 Google

Google has a bug bounty program allowing researchers to freely test some of their applications, including but not limited to ”google.com” and ”youtube.com” [22]. In our testing we have used these to websites to quickly test theeCB and to check if any new changes perform as wanted. We also looked on the results and considered if they were reasonable and consistent. This has been a great way to double check that everything works as it was planned.

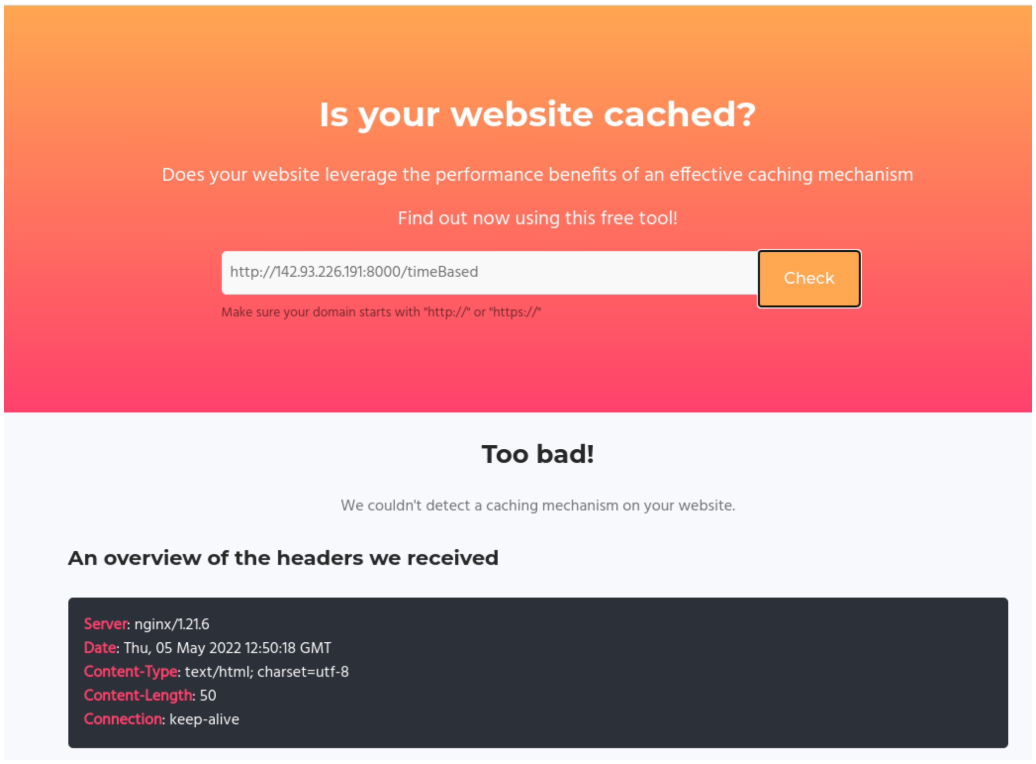
6.4 Performance and benchmarks

Execution time is not of essence when checking for caching-related data. Since speed will likely invalidate the results, as adding more speed has unknown consequences. The results of speeding up theeCB will vary depending on how each individual proxy is configured. As mentioned previously, some headers are updated in intervals, e.g., each minute. Therefore, the calibration phase must either be longer than one minute, or theeCB must finish before one minute. The response times may also vary because of an increase in requests if we were to speed up theeCB. The results would then be very hard to interpret, which is another strong argument for slowing down instead of speeding up. Finishing theeCB in one minute is not reasonable, therefore speed tests are irrelevant when benchmarking. However, we compared theeCB to the results of Burp Suite's caching related tests and other services checking for caching in web applications. To test the progress, we made a simple test application to test theeCB against. This test application has a few endpoints which has some differences in behavior

6.5 Benchmarking with test application

We also uploaded our test application to Digital Ocean for a consistent way of testing theeCB. This test application gives us the opportunity to test theeCB up against other scanners. As a result, we could see how each of the different tools preformed, and which tool that performed the finest. This was very beneficial for further developing theeCB. For instance, the web application cache-checker.com couldn't find a form for caching mechanism in our test application. Result presented in the figure 6.1.

Additionally, Burp Suite's cache tests could not find any kind of caching mechanism either. In this case Burp Suite's scanner was additionally configured to look at XSS and caching related vulnerabilities, however Burp Suite returned no results. This is presented in the issue activity log, shown in figure 6.2.


Is your website cached?
Does your website leverage the performance benefits of an effective caching mechanism
Find out now using this free tool!
http://142.93.226.191:8000/timeBased
Check
Make sure your domain starts with "http://" or "https://"

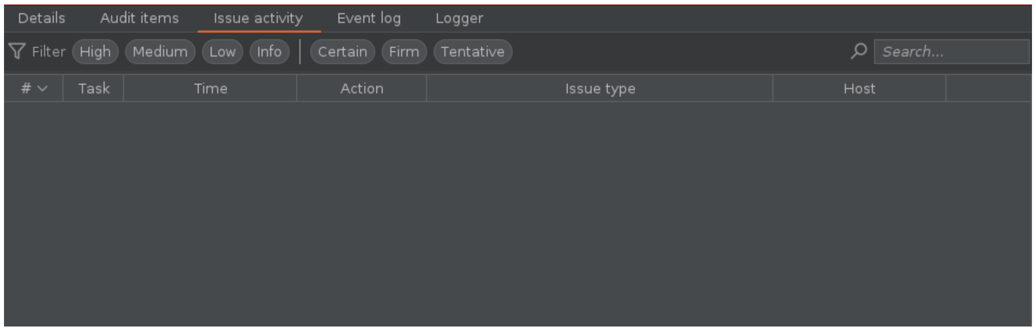
Too bad!

We couldn't detect a caching mechanism on your website.

An overview of the headers we received

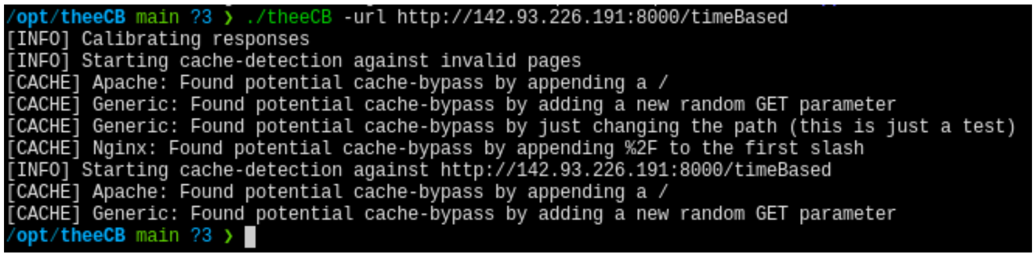
```
Server: nginx/1.21.6
Date: Thu, 05 May 2022 12:50:18 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 50
Connection: keep-alive
```

Figure 6.1: Test Application scanner for caching with Cache Checker



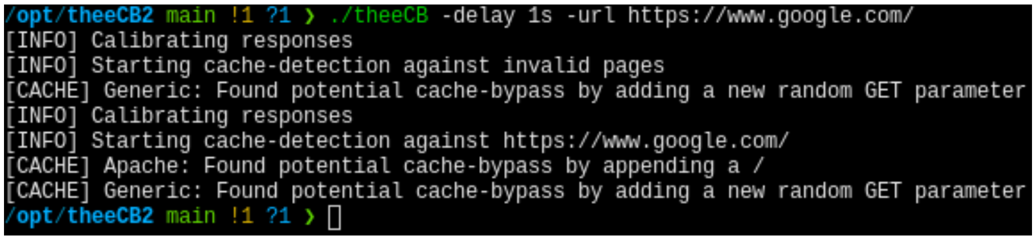
#	Task	Time	Action	Issue type	Host

Figure 6.2: Burp's Issue Activity Scanner did not find any caching mechanisms



```
/opt/theeCB main ?3 ❯ ./theeCB -url http://142.93.226.191:8000/timeBased
[INFO] Calibrating responses
[INFO] Starting cache-detection against invalid pages
[CACHE] Apache: Found potential cache-bypass by appending a /
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
[CACHE] Generic: Found potential cache-bypass by just changing the path (this is just a test)
[CACHE] Nginx: Found potential cache-bypass by appending %2F to the first slash
[INFO] Starting cache-detection against http://142.93.226.191:8000/timeBased
[CACHE] Apache: Found potential cache-bypass by appending a /
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
/opt/theeCB main ?3 ❯
```

Figure 6.3: theeCB detected caching mechanisms and potential bypasses.



```
/opt/theeCB2 main !1 ?1 ❯ ./theeCB -delay 1s -url https://www.google.com/
[INFO] Calibrating responses
[INFO] Starting cache-detection against invalid pages
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
[INFO] Calibrating responses
[INFO] Starting cache-detection against https://www.google.com/
[CACHE] Apache: Found potential cache-bypass by appending a /
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
/opt/theeCB2 main !1 ?1 ❯
```

Figure 6.4: theeCB scanning for caching mechanisms against www.google.com

In comparison theeCB detected the use of caching, and how to circumvent it. Shown in 6.3.

6.6 Benchmarking with web applications

During development we ran theeCB against google to check that theeCB ran as intended. We also manually tested the results to determine the performance of theeCB. The goal was to get the same results from theeCB as manual testing, without any false positives, or false negatives.

Figure 6.4 shows the results of theeCB running against google.com, theeCB found two ways of bypassing the cache at the root path (/). Figure 6.5 shows the results of cache-checker against the same endpoint, and revealed that google indeed was using caching, a similar result as to theeCB.

Figure 6.6 shows the results of theeCB running against youtube.com, theeCB found two ways of bypassing the cache at the root path (/). Figure 6.7 shows the results of cache-checker against the same endpoint, and revealed that Youtube indeed was using caching, a similar result as to theeCB.

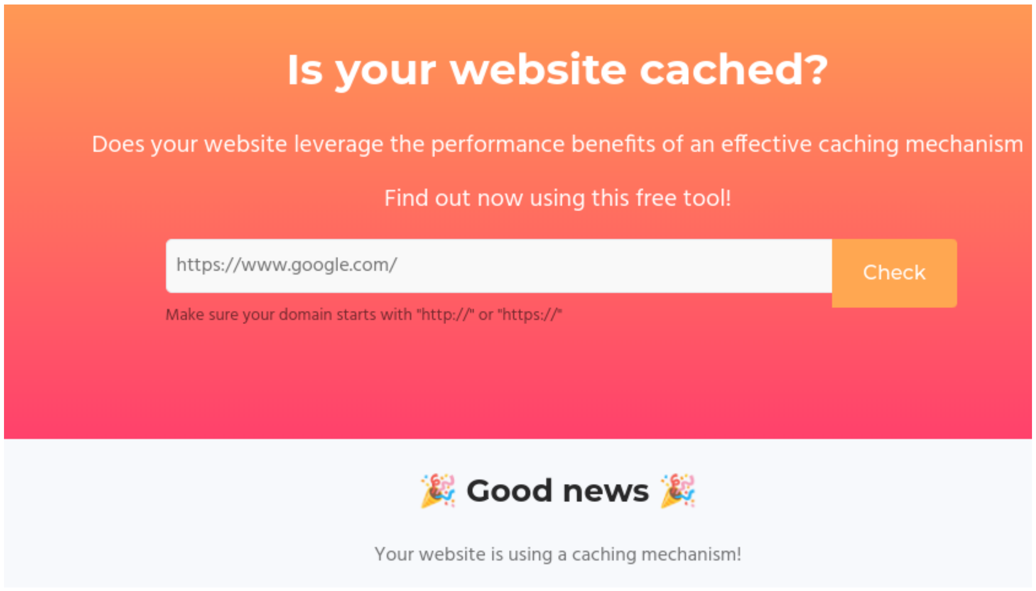
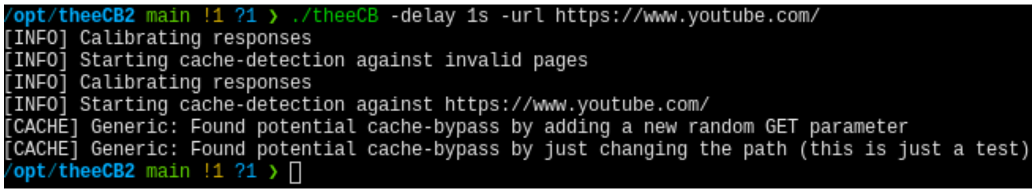
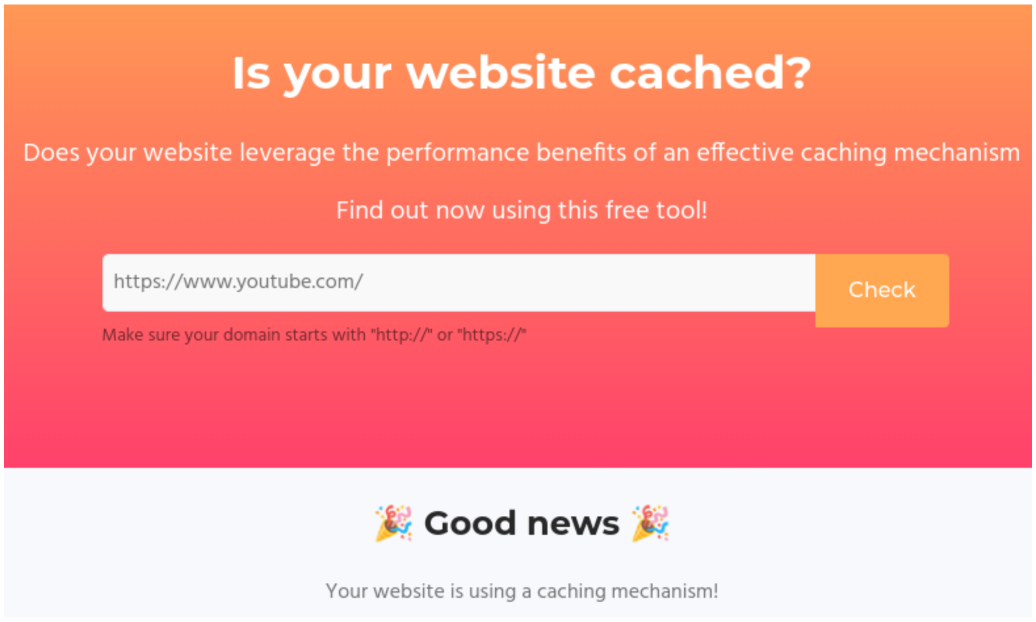

Is your website cached?
Does your website leverage the performance benefits of an effective caching mechanism
Find out now using this free tool!
https://www.google.com/
Check
Make sure your domain starts with "http://" or "https://"
🎉 Good news 🎉
Your website is using a caching mechanism!

Figure 6.5: Cache Checker checking for caching mechanisms against www.google.com



```
/opt/theeCB2 main !1 ?1 ❯ ./theeCB -delay 1s -url https://www.youtube.com/
[INFO] Calibrating responses
[INFO] Starting cache-detection against invalid pages
[INFO] Calibrating responses
[INFO] Starting cache-detection against https://www.youtube.com/
[CACHE] Generic: Found potential cache-bypass by adding a new random GET parameter
[CACHE] Generic: Found potential cache-bypass by just changing the path (this is just a test)
/opt/theeCB2 main !1 ?1 ❯
```

Figure 6.6: theeCB scanning for caching mechanisms against www.youtube.com


Is your website cached?

Does your website leverage the performance benefits of an effective caching mechanism

Find out now using this free tool!

https://www.youtube.com/

Make sure your domain starts with "http://" or "https://"

🎉 Good news 🎉

Your website is using a caching mechanism!

Figure 6.7: Cache Checker checking for caching mechanisms against www.youtube.com

Chapter 7

Discussion

7.1 Vulnerability scanners

Vulnerability scanners probes for different types of vulnerabilities, in this case within a web application. [23]. theeCB, that we created is not a vulnerability scanner, as it does not find vulnerabilities in any way. In comparison theeCB is a tool for detecting if a web application uses a form for caching mechanism.

However, vulnerability scanners commonly test for cache poisoning vulnerabilities. Burp Suite for instance, includes one of the most sophisticated web application vulnerability scanners. When running a vulnerability scan from Burp Suite against our simple vulnerable application, Burp Suite finds no issues, even though a XSS vulnerability was implemented. The problem is not that Burp Suite lacks in performance, the problem is that caching is rather complex. In this case it is not possible to find this type of vulnerability by changing one part of the request at a time, which the Burp Suite's scanner mainly does. It can therefore be beneficial to run theeCB before running any type of vulnerability scanner.

By figuring out how the caching works firstly, it is possible to bypass the cache on every request in the vulnerability scanner. This makes the vulnerability scanner minimize false negative results and can result in finding hidden and critical vulnerabilities. However, this is not the intended way of using the supposed vulnerability scanner, this means that cache poisoning related tests will probably fail. This can be mitigated by running the vulnerability scanner twice or running the cache poisoning tests separately.

7.2 Cache detection tools

There exists various websites for checking if a website uses a from for caching mechanism. These websites often run simple checks for determining if the specified website is using a cache. Websites like https://www.giftofspeed.com/cache-checker/ or https://cache-checker.com/ performs such checks [24] [25]. By setting up a web application and specifying a URL to these two websites, it is obvious that these two websites only check the headers to determine if caching is implemented. This is clear because the web application only receives one request. With one request it is not possible to determine differences in response times, therefore the only solution is to look at caching related headers. Similarly, vulnerability scanners mostly check for headers when determining if caching is enabled. Some exceptions exist in certain tools, where simple cachebusters are implemented during the entire scans. These cachebusters will bypass most configurations, but it is not possible to be certain that it worked without manually figuring out how the caching is set up. Therefore, it may or may not be false negatives in the results. The cache detection tool theeCB that we made, is made to prevent false negative results, as well as to gather information to uncover more vulnerabilities. However, as the fingerprinter part of theeCB is made to mitigate false negatives, it might include some false positives instead. When developing theeCB we looked at different ways of avoiding the cache, an found out that the main way of avoiding the cache mechanisms is to use a form of cachebuster. In some cases, it can be as simple as adding a random GET parameter to the end of the query string, whereas sometimes it could be a lot harder since the caches are set up with different configurations. Optionally it is possible to wait til the cached response is expired. When a cache is identified, it is also useful to know about the cache key. By knowing the cache key, it is possible to reliably bypass the cache, as well as finding ways to poison the cache. However, this could be rather tedious, and therefore it is more efficient to automate it.

7.3 Cache related headers

The proxy may return cache related headers in the response. These headers include information of whether the response was cached, and how long the cached response is valid. A common cache related header is the E-Tag header. The E-Tag header displays a unique hash per cached response. If the hash is updated, then the response is also updated in the cache.

Without knowing about the configuration of the proxy, the E-Tag header cannot be trusted blindly. The E-Tag header may be used for different purposes, not related to caching, or given to mislead security professionals. When the E-Tag header is returned it is not possible to know which proxy returned it, as there may be multiple proxies in front of the web application. The fingerprinter in theeCB will detect if the E-Tag changes and will detect the caching mechanism in normal cases. The difference in the approach of the fingerprinter, is that the fingerprinter will check if the E-Tag header changes only when an uncached response is returned. This method is generalized, and applicable to every header, as well as other elements of the response. It is not possible to determine which proxy returns the E-Tag header in most cases. In theeCB we are not simply relying on what information we get from the proxy; we deem that information untrustworthy.

Chapter 8

Related Work

8.1 Web Cache Entanglement

James Kettle, an employee at PortSwigger, did research on caching and posted findings on PortSwigger's website [2]. His research shows how to manually bypass caching mechanisms, as well as how to exploit misconfigurations. It also shows the devestating effects a misconfigured proxy can have.

8.2 Burp Suite

Burp Suite is a tool created by PortSwigger [26]. This is a tool with advanced vulnerability scanner compatibilities. Burp Suite's vulnerability scanner can scan for caching related and XSS vulnerabilities, among other types of vulnerabilities. Burp Suite offers a lot of advanced scans for a lot of different web application types.

8.3 Zed Attack Proxy

Zed Attack proxy, commonly referred to as ZAP, is a web application scanner that are widely used among penetration testers and developers [27]. This is a similar proxy to Burp Suite, but is completely free and open source. The proxy is developed and maintained by volunteers under the Open Web Application Security Project, in short OWASP. ZAP can in a similar way to Burp Suite, be used to scan for XSS and caching related vulnerabilities, among other things.

8.4 Cache Checker

Cache Checker at cache-checker.com is a tool for detecting caching mechanisms in web applications [25]. This tool is provided by the organization cache warmer, which is a web application performance enchanting provider. Cache Checker provides an easy way to quickly check if a web application is using caching. When entering an URL to Cache Checker, Cache Checker will send a request to the specified URL and check if the response contains caching related headers.

Chapter 9

Conclusion

9.1 Conclusion

In this thesis we have created a web application cache mechanism detection tool. theeCB performs well with consistence findings, it is also quite robust. If any errors occur, the error is displayed in the terminal and the program will quit. This is to prevent errors from going undetected, and for easier resolving bugs.

We have created a tool that can easily be extended with more cache bypassing methods. By creating a new function that modifies the request in any preferred way, and adding the function to the list, a new payload is implemented.

When developing this application, we used a manual way of detecting bugs. The tool has been actively used in penetration tests since around February. By using the tool, we found multiple bugs that were later resolved. We also added some tests regarding the fingerprinting phase, which also uncovered some minor issues. The tool was built as modular as possible, which allows for easy extendibility.

9.2 Further work

Even though the cache detection tool performs as intended, there is more features and work that could be added before we can call it completed. There are some features that would add improvements to the tool, but we did not get the time to implement them, or they were out of scope for this thesis. For instance, we could have added numerous more payloads. theeCB could have been converted to a vulnerability scanner, this was out of scope for the thesis, and is also a much larger project. The advantage theeCB would have over most other vulnerability scanners, is that it would consistently bypass the cache while scanning.

We could implement payloads that check for caching-related vulnerabilities, trying to do cache-poisoning.

The fingerprinter part of theeCB can also be implemented in other projects for all types of scanning. Additionally, the fingerprinter could potentially be made to be its own tool. Meaning the tool would inspect two or more similar responses and determine whether they are equal or not, this could be used as an API for other tools.

The code could have been split up in smaller sections, to make the code even more readable and reusable.

theecB could be using some concurrency, this is something we were planning on doing all along, but might be a bit difficult. By adding concurrency, you are also adding strain along the same route, this could in many ways add issues in performance and results. If the web applications performance is negatively impacted due to the extra requests sent, the response times may become unstable and unusable. Another thing to consider is web application firewalls, web application firewalls may stop accepting requests if seeing too many in a short amount of time. In the future it may be a good idea to implement an argument to allow for using multiple threads, as it may be a nice feature to have in certain cases.

During development we focused mainly on manual testing. Testing functions was an afterthought, meaning the tests was created after core functions was written. In the future it would be a good idea to create more tests.

More arguments should be implemented in the tool, giving more options and more flexibility while using it in a penetration test. As an example, the tests against the invalid pages and the specific path should be optional, both for using either one of them, or both. Currently it automatically does tests against both.

theeCB currently relies on being able to bypass the cache in the calibration phase. This is no problem when testing against invalid pages, as the pages requested are most likely uncached. The problem arises when checking against specific pages. By adding multiple cachebusters to the request to the specific page during calibration, theeCB assumes the cache was successfully bypassed. If the cache were not successfully bypassed, theeCB will not return any usable results. It is not easy to make theeCB work against every web application, but there is still a lot of unused potential. As an example, we could use the methods discovered against invalid pages in the calibration phase against specific pages.

The threshold for determining that the new response time was different is set to 10%. This should ideally be changed to reflect the stability of the web application. If a web application is consistently responding at exactly 2ms every time, the threshold should be lowered. On the other hand, more unstable response times, e.g. a range of 1ms to 5s, the threshold should be increased.

If any element consistently changes only on uncached responses, the element should not be disregarded as it currently does. In the test application, the ”/html” endpoint consistently changes the HTML document only on uncached responses.Currently the fingerprinter will disregard these changes, as they are considered dynamic content in the calibration phase. By tracing which elements only changes on uncached responses, it is possible to include the changes in the HTML document to theeCB's advantage.

Appendix A

Instructions to Compile and Run System

Appendix B

Linux/MacOS

B.1 theeCB

To run the tool on Linux or MacOS, use the following instructions.

```
cd theeCB
go build
./theeCB -h
```

B.2 testapp

To run the test application on Linux or MacOS, use the following instructions.

```
cd theeCB/testapp
docker-compose up
```

Appendix C

Windows

C.1 theeCB

To run the tool on Windows, use the following instructions.

```
cd theeCB
go build
theeCB.exe -h
```

C.2 testapp

To run the test application on Windows, use the following instructions.

```
cd theeCB/testapp
docker-compose up
```

Bibliography

[1] PortSwigger Research. James kettle, . URL https://portswigger.net/research/james-kettle.

[2] James Kettle. Web cache entanglement: Novel pathways to poisoning, Aug 2020. URL https://portswigger.net/research/web-cache-entanglement.

[3] Rob Pike. The go programming language. *Talk given at Google's Tech Talks*, 14, 2009.

[4] Sumit Kumar, Sumit Dalal, and Vivek Dixit. The osi model: Overview on the seven layers of computer networks. *International Journal of Computer Science and Information Technology Research*, 2(3):461–466, 2014.

[5] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[6] Chuck Musciano and Bill Kennedy. *HTML & XHTML: The Definitive Guide: The Definitive Guide*. " O'Reilly Media, Inc.", 2002.

[7] Jon Postel et al. Transmission control protocol. 1981.

[8] Upasana Sarmah, DK Bhattacharyya, and Jugal K Kalita. A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, 118:113–143, 2018.

[9] Martin Sysel and Ondřej Doležal. An educational http proxy server. *Procedia Engineering*, 69:128–132, 2014.

[10] The Apache Software Foundation. Apache module mod_proxy, 2022. URL `https://httpd.apache.org/docs/current/mod/mod_proxy.html`.

[11] Michael Rabinovich and Oliver Spatscheck. *Web caching and replication*, volume 67. Addison-Wesley Boston, USA, 2002.

[12] Duane Wessels. *Web caching.* " O'Reilly Media, Inc.", 2001.

[13] Xiaoni Chi, Bichuan Liu, Qi Niu, and Qiuxuan Wu. Web load balance and cache optimization design based nginx under high-concurrency environment. In *2012 Third International Conference on Digital Manufacturing & Automation*, pages 1029–1032. IEEE, 2012.

[14] James Kettle. Practical web cache poisoning, Aug 2020. URL `https://portswigger.net/research/practical-web-cache-poisoning`.

[15] Amid Klein. Divide and conquer. *HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics, Sanctum whitepaper*, 2004.

[16] Omer Gil. Web cache deception attack. *Black Hat USA*, 2017, 2017.

[17] PortSwigger Research. Portswigger is a web security company on a mission to enable the world to secure the web., . URL `https://portswigger.net/about`.

[18] Aileen G Bacudio, Xiaohong Yuan, Bei-Tseng Bill Chu, and Monique Jones. An overview of penetration testing. *International Journal of Network Security & Its Applications*, 3(6):19, 2011.

[19] Aron Laszka, Mingyi Zhao, Akash Malbari, and Jens Grossklags. The rules of engagement for bug bounty programs. In *International Conference on Financial Cryptography and Data Security*, pages 138–159. Springer, 2018.

[20] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[21] Inc. DigitalOcean. Droplets, June 2018. URL `https://docs.digitalocean.com/products/droplets/`.

[22] Google LLC. Google and alphabet vulnerability reward program (vrp) rules. URL https://bughunters.google.com/about/rules/6625378258649088/google-and-alphabet-vulnerability-reward-program-vrp-rules.

[23] Johan Nilsson and Vesa Virta. Vulnerability scanners. *Master of Science Thesis at Department of Computer and Systems Sciences, Royal Institute of Technology, Kista, Sweden*, 2006.

[24] Gift of speed, browser caching checker, Mai 2022. URL https://www.giftofspeed.com/cache-checker/.

[25] Is your website cached?, Mai 2022. URL https://cache-checker.com/.

[26] Akash Mahajan. *Burp Suite Essentials*. Packt Publishing Ltd, 2014.

[27] Simon Bennetts. Owasp zed attack proxy. *AppSec USA*, 2013.

University
of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Hein Meling