Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Toni Huttunen

Browser cross-origin timing attacks

Master's Thesis
Espoo, November 18, 2016

Supervisor: D.Sc.(Tech.) Vesa Hirvisalo
Advisor: M.Sc.(Tech.) Jussi Hanhirova

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Toni Huttunen

Title:
Browser cross-origin timing attacks

Date: November 18, 2016 **Pages:** 75

Major: Software Technology **Code:**

Supervisor: D.Sc.(Tech.) Vesa Hirvisalo
Advisor: M.Sc.(Tech.) Jussi Hanhirova

Web browsers are an integral part of people's everyday life. The browser is required to manage high privacy tasks, including bank account management, examination of patient records, and all the small and large aspects of life in the form of email and social media. Web page environment develops in a continuously more diverse path, which broadens the browser attack surface and makes protecting browser from security vulnerabilities more challenging.

One challenging feature is that a page is allowed to include third-party documents into the page content. A web browser prevents this page from reading the actual content of those third-party inclusions. However, cross-origin inclusion support induces security issues relating to hidden channels. One family of those issues is formed by cross-origin timing attacks.

This master's thesis describes an attack method where a malicious page steals sensitive secrets from third-party pages. The attack abuses search functionality by triggering browser to perform multiple third-party loads to a targeted search engine, which in specific situations leaks sensitive cross-domain information via timings. We perform a practical attack, where a page controlled by the attacker manages to reveal email content from a simulated web mail service. In addition to this, we introduce a measurement tool, which is built to simplify the process of revealing and fixing those vulnerable web services.

Furthermore, we bring out a security issue affecting Mozilla Firefox browser, which I found during writing this thesis. This vulnerability CVE-2016-2830 is a high level security finding approved by Mozilla. The vulnerability enables a web page to perform new requests even after an attacking page has been closed. This makes it possible to perform Cross-Site Request Forgery (*CSRF*) attacks for a much longer time from the closed page.

Keywords: security, browser, page load, timing attacks, exploiting cross-site search page, attack method

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Toni Huttunen

Työn nimi:
Selaimen alisivulatausten aikahyökkäykset

Päiväys:	18. marraskuuta 2016	**Sivumäärä:**	75
Pääaine:	Ohjelmistotekniikka	**Koodi:**	
Valvoja:	Tekniikan tohtori Vesa Hirvisalo		
Ohjaaja:	Diplomi-insinööri Jussi Hanhirova		

Verkkoselaimella käsitellään yhä arkaluontoisempia ja luottamuksellisempia tietoja. Selaimet suorittavat käsittelemäänsä sisältöä ja ovat siten tekemisissä henkilöiden arkipäiväisissä tai tietosuojaa edellyttävissä asioissa, kuten pankkitilin hallinnoinnissa, potilastietojen tarkastelussa ja vaikkapa sähköpostin ja sosiaalisten sivustojen kautta koko elämän pienissä yksityiskohdissa. Verkkosivujen sisältö ja rakenne kehittyvät yhä entistä monipuolisemmiksi, mikä laajentaa selaimeen kohdistuvaa hyökkäyspinta-alaa, ja tekee suojautumisesta haastavampaa.

Eräs haasteita aiheuttava ominaisuus on selaimen mahdollisuus liittää sivustoon kolmannen osapuolen sisältöä. Selain estää teknisesti liittävää sivustoa lukemasta liittämäänsä sisältöä suoraan, mutta tämä mahdollisuus sisällyttää kolmannen osapuolen sisältöä sivuille aiheuttaa haasteita tietoturvassa syntyvien piilokanavahaavoittuvuuksien muodossa. Tähän hyökkäyskategoriaan kuuluu tutkimani selaimen sivulatauksia hyödyntävät aikahyökkäykset.

Tämä työ esittelee hyökkäystavan, jolla pahantahtoinen verkkosivu voi sisältöä lataavalla aikahyökkäyksellä tietyissä olosuhteissa paljastaa kokonaisia mielivaltaisia merkkijonoja kohdesivustolta hyödyntäen sivuston hakutoiminnallisuutta näkemättä hakutuloksia. Tässä esiteltävässä käytännön kokeessa hyökkäävä sivusto onnistuu varastamaan simuloidusta sähköpostista viestin sisältöä. Työssä on tuotettu mittaustyökalu, jolla sivustot, jotka ovat alttiita kyseiselle hyökkäykselle, ovat helpommin testattavissa ja siten korjattavissa.

Tässä työssä julkaistaan myös Mozilla Firefox -selaimesta työn aikana löytämäni haavoittuvuus CVE-2016-2830, jonka Mozilla luokitteli korkean tietoturvatason haavoittuvuudeksi. Haavoittuvuus mahdollistaa sivuston luoda uusia verkkoyhteyksiä mielivaltaisiin verkkoresursseihin, vaikka selain olisi jo hyökkäävältä sivustolta aikaa sitten poistunut. Tämä mahdollistaa siten *CSRF*-hyökkäyksen (Cross-Site Request Forgery) suljetulta sivustolta.

Asiasanat:	tietoturva, selain, sivulataus, aikahyökkäykset, kolmannen osapuolen hakusivuston väärinkäyttö, hyökkäysmenetelmä
Kieli:	Englanti

Acknowledgements

I would like to thank my parents Jarkko Huttunen and Pirjo Huttunen, siblings and friends for their support during the writing. I thank Mika Huttunen and Lari Sinisalo for insightful comments and suggestions. I wish to thank my supervisor Vesa Hirvisalo and my instructor Jussi Hanhirova for their help in creating this thesis. I would also like to express my gratitude to F-Secure colleagues of many fascinating years. Driven by the Challenge.

Espoo, November 18, 2016

Toni Huttunen

Abbreviations and Acronyms

Ajax	Asynchronous JavaScript and XML
Apache	World's most used web server software
API	Application programming interface
BFS	Breadth-first search
CSP	Content Security Policy
CRIME	Compression Ratio Info-leak Made Easy
CORS	Cross-origin resource sharing
CSRF	Cross-site request forgery
CSS	Cascading Style Sheets
CVE	Common Vulnerabilities and Exposures
DFS	Depth-first search
DNS	Domain Name System
DOM	Document Object Model
Gmail	A free and popular mail service provided by Google
HEIST	HTTP Encrypted Information can be Stolen through TCP-windows
HSTS	HTTP Strict Transport Security
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IFrame	HTML Inline Frame Element
OWASP	Open Web Application Security Project
RTT	Round-trip time
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
Token	A piece of unguessable data
URL	Uniform Resource Identifier
VPN	Virtual Private Network
W3C	World Wide Web Consortium

Contents

Chapter 1

Introduction

The Internet is a massive networking infrastructure that allows millions of computers to communicate and share resources with each other. World Wide Web (WWW) is a familiar part of the Internet, which is an information-sharing model over the Internet. In this model, information is transmitted and presented in a standardized way. The Web consists of multitude of information sources which are scattered over various servers in the form of web sites. This creates the basis for the modern web services, where different computers can communicate and share resources. The Internet has made access to information far easier than before and the Internet affects nearly every aspect of modern life from education and healthcare to business and government.

Web browser is a crucial part of the modern Internet, where security is only as strong as its weakest link. End-user web browsers are targeted by a growing number of attacks. These attacks may tempt attackers to penetrate through popular websites, not to steal and leak secrets from single server, but to have a possibility of controlling browsers of the all visitors of the victim page. This approach makes popular services valuable targets, even if they themselves do not contain anything valuable as such.

Users have permitted browsers to handle their web usage, including authentication, session handling and page history. In order to make user's life simpler, the browser may even collect service passwords and fill them automatically. Those details are valuable information to any hacker that trade those information dumps for cash in black markets. Details are used for various activities, including blackmailing, scamming, targeted advertising or simply using reputation of specific user accounts to order goodies.

After an end-user is lured to a page controlled by an attacker, the attacker may perform multiple attacks that could lead to total compromise of the end-user machine. Even if the browser could not be compromised, it is

common that a single browser instance has active sessions over multiple web pages simultaneously. Attacker may try to utilize browser to generate actions over those sites and perform functions on the victim's behalf. This thesis is relating to those attacks where web page targets other pages using existing session of the browser.

One branch of Information Security is web application security that focuses security of websites, web applications and services. Open Web Application Security Project (OWASP) community raises awareness about application security and they have published a familiar list of TOP 10 most important web application security weaknesses. [1]

In addition to this, they maintain a large comprehensive best practice recommendation cheat sheets of issues that must be taken into account when securing a web page. They point out that lists are updated regularly because new flaws are discovered and attack methods sophisticate. They state that the most cost-effective solution to find and eliminate security weaknesses is human experts armed with good tools. [2]

This work presents a new cross-origin timing attack method where a malicious page steals sensitive secrets from third-party pages via abusing search functionality timings of the targeted site. In our demonstrative experiment, we simulated a mailbox and managed to steal forgot password link from the mailbox only by luring mailbox victim to follow a site controlled by an attacker.

We also developed a tool that is used to reveal those timing issues in practice web applications. We benchmark the tool by revealing how the html content embedders time distributions behave based on processed payloads.

1.1 Purpose of the study

In order to understand the current situation of web timing attacks, this study has two major purposes. One to investigate published web timing attacks. The other is to demonstrate an attack method based on embedded content retrieving timings.

1.2 Research questions

The research questions of this thesis mainly focus on cross-origin timings attacks. This thesis seeks answers following research questions:

- How to measure cross-origin timing vulnerabilities in browsers?

- Depending on the cross-origin target context, which content retrieving elements are recommendable for more accurate results?
- How a targeted document context affects to a browser load and processing time?
- How a timing vulnerability can be advantaged retrieving cross-origin secrets?
- How altering response time effects the impact of cross-origin search attack?

1.3 Contribution

This thesis describes general timing attacks and especially cross-site timing attacks in a browser context. We propose a novel attack method to steal cross-origin tokens by targeting content embeddees' timings. We implemented two applications to study this issue in more detail.

The first application is made in order to study cross-origin timing issues from practical web applications. We introduce this measurement system that compares any two cross-origin page resources and helps to reveal whether they leak timings.

In our demonstrative experiment, we use this tool to reveal how the browser and application timings operate with a different resource contexts. We identify vectors that are affecting measured timing differences and reveal how the different HTML content embedders are seen in terms of time distribution.

The second application is a implementation of this presented novel attack method. The application executes this attack in practical environment. In our demonstrative experiment this application manages to steal arbitrary token from implemented vulnerable web application.

During carrying out this thesis, we found socket handling security issue in Mozilla Firefox browser. We build a proof of concept application to make it more easier for browser developers to locate, test and repair. We reported this and confirmed that the latest patch resolves the issue. We decided to include brief details to this thesis and present occurring problem and its security impact.

1.4 Structure

The paper is organized as follows. In the following chapter 2 we introduce background of the web browser environment and describe in what kind of environment this thesis settles. Chapter 3 describes what timing attacks are, what different types of timing attacks there are and what relevant research is published in this area relating to this thesis.

The main part of this work is inside chapter 4. We describe our built implementations that include measurement tool, search extractor attack tool and a description of the vulnerability affecting Mozilla Firefox browser.

In chapter 5 we are experimenting our tools. In short, we figure out how HTML element timings react when targeted resources are altered. We are also experimenting how cross-site search extractor tool works in a measurement environment.

Chapter 6 covers evaluation of the performed measurements and their results. In the final chapter we draw our conclusion about the implemented work, and suggest how the research should be continued.

Chapter 2

Web browser

In order to understand problem domain we introduce a web browser environment. In particular, we introduce session handling mechanisms and how the data flows between browser and server. All following chapters are associated more or less in session management, which is vital to understand following attacks.

2.1 Web browser

Web browser is a main gateway to access information and capabilities through web interfaces in a context of online environments. The web browser presents a chosen web resource by requesting it from the server and displaying it in the browser window.

The current web is filled up with a huge number of different types of context and the browser is assumed to translate this information into a presentable format. Thus, the browser manages over multiple external media parsers and handles different types of images, videos and for example *pdf* formats to embedded them in browser context. [3] This exposes a large attack surface and the browser vendors have to make huge compromises between security and usability.

The browser main components are user interface, browser and rendering engine, networking, user interface backend, JavaScript interpreter and Data storage. Safari, Chrome and Opera use Webkit based rendering engine, Internet Explorer uses Trident, and Firefox uses Gecko. The basic rendering engine main flow is similar between Gecko and Webkit. [3]

Major browsers are following W3C (World Wide Web Consortium) standards, which involve standards for building and rendering web pages, including technologies such as *HTML*, *CSS*, *SVG* and *Ajax* [4]. Many security

mechanisms are result of practical experimentations and standardised after adoption increases [5]. In past, the browsers were not following standards so thoroughly which caused serious compatibility issues for web authors [3].

ECMAscript, widely known as JavaScript is currently the facto client-side scripting language. Sites utilize a basic scripting interface called Document Object Model (*DOM*). The *DOM* allows scripts to dynamically access and update the content, structure and style of documents. In other words, *DOM* is object presentation model of the HTML document and interface for the JavaScript. [3] The modifications to the *DOM*, by the user and by scripts trigger events that developers utilize to build rich user interfaces.

The security of client side web application relies on browser security features. Web browser security primitives are document object model (*DOM*), frames, cookies and localStorage, where principals are origins and where interactions are protected using mandatory access control. [5] The resource location is specified by using Uniform Resource Identifier (*URI*). Browser uses HTTP protocol to get required document, and the HTTP transfers requests over *TCP*, which finally delivers segments over small *IP* diagrams.

Firefox is one of the major browsers. Current web browser usage trends measured by W3Counter (8/2016) reveals that the most used browsers are Chrome, Safari, Firefox, Internet Explorer & Edge and Opera. Chrome has the largest market share (58.1%), while the second popular browser is Safari (12.7%) and Firefox (12.4%), which is fighting for second place. The fourth is Internet Explorer & Edge with 9.6% market share, and Opera which has 2.8%. [6]

2.2 Client-server protocol

Web browser and server models a client-server computing model. Web browser builds a requests and the server responds to them based on the requested resource. The protocol what the both parties are using for communicating is Hypertext Transfer Protocol (HTTP), which runs over application layer protocol in *OSI* model. Protocol packages are transmitted over Transmission Control Protocol (*TCP*), which provides reliable, ordered and error-checked delivery stream from end-to-end. [7]

A simple request demonstrates a data flow scheme of a request from browser to a web site www.aalto.fi when performing a search query to a page as logged-in user:

```
GET /fi/search/?site_search=all&q=thesis\%20research HTTP/1.1
Host: www.aalto.fi
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0)
Gecko/20100101 Firefox/51.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: visid_incap_577306=saHAebChRnSWh1wLNJQ6jB4H2VcAAAAAQUIP
AAAAAAAjYkMVp/719TTuXqICCOn7;incap_ses_108_577306=05ffewKkCX+vX
u7xfbJ/AR4H2VcAAAAAqMR6qz782+z+a/m1xyZ9bA==;incap_ses_276_57730
6=CRm1ZnHWx2oymIvO34zUA2IH2VcAAAAAI4T9gA+GP1PTWlex5LDfvA==;_hjI
ncludedInSample=1;_ga=GA1.2.9074173.1473840993;_dc_gtm_UA-76242
031-1=1;_gat_UA-12300388-1=1; _gat_UA-52497828-1=1; _gat_UA-524
77555-1=1; _gat_UA-52497828-8=1; _gat_UA-52477555-2=1
Connection: keep-alive
Cache-Control: max-age=0
If-Modified-Since: Tue, 13 Sep 2016 12:32:17 GMT
If-None-Match: 2dede8a862b84a53d71ca5322ac8ccfa62293085dcf9fc5d
d96e6dd69999bf66
```

The request is targeted to the www.aalto.fi server, resolved by the according *DNS* record. The browser includes a request line, and tells that it is interested about /fi/search/ resource path. In this case, the Uniform Resource Identifier (*URI*) also provides query section, which carries additional attributes to the performing server. In this case, the section includes q parameter with a value 'thesis research'. The request includes a cookie, which for example provides session specific identifiers, so the server may act according to the user's access right. The targeted server receives the given headers and acts.

The response could be for example:

```
HTTP/1.1 200 OK
Date: Wed, 14 Sep 2016 09:01:42 GMT
Server: Apache/2.4.18
Last-Modified: Tue, 22 Dec 2015 11:52:16 GMT
ETag: '1aea-5277b3cb5cd82'
Accept-Ranges: bytes
Content-Length: 6890
Vary: Accept-Encoding
Content-Type: text/html; charset=UTF-8

<html><head>Search</head><body>... [rest bytes are truncated]
```

The response follows the same pattern than the request. The first line indicates that the response status code is *200 OK*, which means the server found targeted resource. The response contains a content-length header, which indicates how long the response body is, so the browser knows when the whole response is received. The response body follows headers and is separated by a single empty new line. The response body has the actual document of the response, the browser has requested. The other headers offer details about the server, and for example details about the response encoding and media type. [7]

The headers are passing caching information to make efficient updates of cached information with a minimum amount of transaction overhead. Should be noted, that all documents are not cacheable, including explicitly marked ones, and ones that are dynamically generated. [8]

The browser includes a If-Modified-Since and If-None-Match headers. Those headers tells server, that the browser is already aware of a document from this resource, but it wants to ensure it is up to date. The server compares the modification time and entity tag to a potential response it has, and if the content is not changed, the server responds shortly with a status code *304 (Not modified)*. [8]

2.3 Session handling

In order to understand cross-site request forgery and cross-origin timing attacks the basic knowledge about the browser session handling is required. The browser supports multiple sessions over various web sites and it is required to keep those sessions separated and still offer smooth browsing experiment. The following chapter introduces basic concepts of browser's session, session isolations and cross-origin policy.

2.3.1 Session

Session is a term used to represent time user spends in a web service. Session is required for many basic interactions between web services. In order to associate a single request to an user the service asks credentials and in exchange offers a unique session token, which client includes in every request sent to that specific service. This session token is implemented as cookie header in HTTP protocol. [2]

The HTTP is stateless protocol, which means that the state must be maintained somehow between requests to identify user from other users on the service. For example, when you log in to the web page you don't have

to repeat log-in action every time you follow a link to the same service. A common method to maintain session is a cookie which is included in every request header that browser is performing. [2]

When the service wants to identify user, it includes a cookie header to the page response. The browser reads the header and includes it to all subsequent requests to that service. For this reason, all you have to do is open URL and cookie is automatically included in every new page requests and filled form sent as long as cookie stays valid. The cookie session identifier value is typically strong random character string selected by the server, which is long and random enough no other user may have a chance to guess it. If they could guess or predict the token they could easily act out as an original token user. [2]

The Internet is global network containing millions of sites where single node could be owned by anyone. Thus, the services may not trust each other and there must be an method to separate services and avoid exposing cookie to a malicious service. Acknowledged solution is to isolate service cookies.

2.3.2 Cross-origin policy

The same-origin policy is a security mechanism to isolate documents from each other by restricting how a document or script loaded from one origin can interact with a resource from another origin, such as HTTP cookies for session authentication. The origin is a set of the protocol, port number and host. If they all match, then the origin is same, and the pages can interact without origin restrictions. [9] When a document executes an external script, the browser performs cross-origin request, loads code from third-party page and executes it using permissions of the caller site.

The interactions are placed in three categories, which are cross-origin writes, embeddings and reads. Writes are cross-domain interactions including links, redirects and form submissions, which are typically allowed. Embedding is allowed, including external JavaScripts, images and iframes. On the other hand, cross-origin reads are not directly allowed, which means that in browser context you can not make a page that directly reads a source code of some cross-origin page using active browser session, without target page explicitly allowing it. [9]

There are multiple mitigations to that statement. A targeted web service may relax same-origin policy and selectively permit sources to be shared. Also, for example, Mozilla developer article states that when you load an image content, it is possible to read image height and width of the embedded image. Furthermore, resource status code of the HTTP response is accessible, thus there are situations where server reveals the existence of the resource.

[9]

2.3.3 Session isolation

The cookie is tied to the origin of the domain and is included in every request within the domain-scope. It should be noted, that the service port number and protocol scheme are not part of the scope. This has a strong security impact, which does not necessarily come into mind at first. The cookie mechanism does not respect same-origin policy similarly like when web browsers isolate content retrieved via different points. [10]

The set-up causes a potential security issue, because despite that the cookie is transferred over secure encrypted communication channel HTTPS, the browser also willingly includes it to any same-domain request. The session confidentiality is doomed when the request is sent over insecure channel. [10] Thus, if an attacker has an ability to observe and modify victim's web traffic, they can trick browser to perform resource request to HTTP content and receive the secret token. Also, the request can be initialized from cross-origin site, for example using a image element targeting insecure resource.

To mitigate attacks like this, the HTTPS server may narrow down the cookie scope by introducing attribute named as 'secure' tied to the session token. The 'secure' attribute defines that in addition to previous restrictions the browser may send the specific cookie only over secure connection. [10] Now, because an attacker can't do the man-in-the-middle hijacking of the HTTPS connection, the cookie stays safe.

There is also an another cookie hardening attribute called as 'httponly', which instructs web browser not to expose that specific cookie to the scripts. Thus, same-origin JavaScript has only write, but no read-access to the cookie value. This way, the site's potential same-origin cross-site scripting vulnerability has no direct access to the session and it can't pass it over to the attacker. In addition, there are cookie attributes to restrict cookie to specific path or issue cover domain to expand cookie scope to specific subdomains. [10] Those hardenings are beyond this thesis.

For example, by default, if page `https://www.example.com` sets a cookie, the value is attached to the following pages also `http://example.com`, `http://example.com:8080`, `https://example.com:8080`. The 'secure' attribute would deny the cookie to leak to any of those http sites.

2.3.4 Cross-site request forgery

We stated at the previous chapters that the cross-origin writes are allowed. There are situations where this privilege should be prevented. There are

sites, that do not recognize whether the write is performed by the site itself, or by the another cross-origin page. [11]

For example, a web shop has a shopping basket and a basic form flow to carry out an buy order. Now, it is important, that no cross-origin page may exploit the trust that a web shop has between user's browser. Otherwise, a third-party site could impersonate a user action, which adds an item to the shopping basket and after launch a buy order in victim's name.

Cyber-criminals are targeting legitimate websites to inject malicious scripts that compromise the security of the visitors of such websites. Later, they perform actions using the visitor's browser without their permission. Those actions could be cross-origin writes targeting vulnerable services. In that case, browser, vulnerable server or even network firewall does not detect that the request is not made according to the user's own will and thus, the requests are routed to the target service, even when the attacker has no direct access to the service.

The cross-origin writes are prevented by including a secret token to all functionalities, which are meant to be protected against cross-site request forgery (*CSRF*) attacks. The token is similar to the session token, but unlike the session token, it is included in the URL parameter or as part of the HTTP POST body parameter, while the session token is transmitted over cookie value. This way, the token must be included every time to the forms and requests by the same-origin script, and the receiver server verifies that the CSRF token is the same original one the server has given before. The token is random secret token generated by the server and tied to the unique user session. Third-party pages do not know the token, and thus the cross-origin writes are denied [11]. Furthermore, the third-party page has no access to the cookie either but the relating cookie is included automatically to the requests.

Chapter 3

Timing attacks

In this chapter, we describe what the timing attacks are in the context of a web browser. In particular, we introduce what cross-site timing attacks are, and how they differ in the context of other timing attacks. We also note why building a reliable attack is challenging.

Furthermore, we mention how browser user could mitigate risks and how service provider should protect against those attacks. After, we expose existing research about browser web timing attacks. Knowledge of presented linear comparison attack and resource load timings are later in this thesis used to build cross-origin search extractor implementation.

3.1 Timing attacks

A timing attack uses statistical analysis of how long it takes your application to do something in order to learn something about the data it's operating on [12]. Timing vulnerabilities are the issues that may still exist even when proper application hardening is done successfully. Vendors have a difficulties in finding timing vulnerabilities because they are hard to identify from a code review perspective and may be found only through testing and experimentation [13]. Majority of the timing attacks exists because of the algorithmic optimizations, such as ending operation immediately when result is known. Those timing issues are hard to identify, but they are also hard to fix. Eliminating a timing issue typically makes software perform less efficiently. In other words, fixing a timing vulnerability is a trade-off between privacy and performance. [14]

Timing attacks are the most well-known type of side channel attacks. Another side channel attacks monitor variables like visuals, power consumption and size of network packets. Timing attack extracts private information by

issuing certain operations with a purpose of measure time took to perform those operations [15]. Typically, timing attack give a small bits of information, which may be eventually leveraged to a more significant attack. The timing attack is a different than common invasive attacks, like buffer overflows, injections and cross-site scripting attacks that the active attack does not change the original control flow of the application [16].

Laboratory environment where the timing vulnerabilities are studied should be thought thoroughly. When a real world environment is simulated in a simplified testing environment, there is a chance that the final measurements are biased and the results are not transferable to other environments. The cross-origin attacks might seem to work very well in the laboratory, but in practice, it could be common that there are larger jitter or for example more simultaneous browser pages open, which affect to the measurements enough to mitigate, or at least make the actual attack more challenging. As within any timing attack, the attack reliability is highly dependent on the quality of the gathered timing data.

3.2 Web timing attacks

In a network timing attack, the attacker is targeting web applications and learning their internal states by launching requests over network. Timing attacks are effective when targeting local internal application [17], but they work also over local area network and Internet, even though the additional network hops increase noise of the measurements.

A random noise variable, known as jitter, absorbs randomness from the measurement. The measured time consists of a constant clock skew, constant processing time, propagationTime and jitter. The jitter is eliminated by using statistical methods and repeated samples. Time resolution an attacker can observe is a function between how many samples attacker takes and how large parts of jitter can be efficiently filtered out [18].

In a classic way, web timing attacks are divided into two types, direct timing attacks and cross-site timing attacks [19]. In a study of Timing Attacks in the modern web they introduce a third attack class, which consist of internal browser timing.

3.2.1 Direct timing attacks

In a direct timing attack the attacker learns private information from the server state by performing various actions while inspecting the response time differences. The attacker has direct connection to the targeted server

A simple direct timing attack is performed for example when an attacker tries to resolve whether a specific user exists in a target system using timings. The potential timing difference occurs when attacker logs in as arbitrary user using arbitrary password. Login implementation verifies that the user exists, and if and only if the user exists, verifies that the account is not locked or expired and that the password hash matches to the known one.

In case of invalid user, the server is only required to perform database query to discover that the user does not exist. In other case, the user exists and later checks are required. Thus, work amount and thereby processing time depends on the validity of the tested user name. In practical, database caching might cause issues if there is a hit and request is queried more than once.

One of the most famous direct timing attack is introduced by Brumley and Boneh. They introduced a practical remote timing attack against *OpenSSL* that is wide use in web servers to secure communications against eavesdropping. Their practical attack managed to extract 1024-bit *RSA* private key over local area network stored on the Apache web server. The attack execution took about two hours and a million queries. In a result, they concluded that 1 millisecond variance is measurable over local area network. Besides the Apache, they found similar timing issues from the *libgcrypt* library used in *GNUTLS* and *GPG* security libraries. [18]

Later, study of Crosby focused to examine time resolution more closely. In their simulated case, the attacker using some knowledge about the measured environment, they were able to reliably distinguish a processing time differences as low as 200 nanoseconds over local network with false negative and false positive rates under five percent. Over Internet, they managed to measure 30 microsecond difference within 1000 measurements. [18]

For local measurements and hardware-near measurements central limit theorem applies, which says that after you measure enough times a number of independent random values, then the resulting dataset will be normally distributed. Crosby study underline that the network timing measurements are not normally distributed, and for accurate measurements, the median, average and minimum values should not be used for getting most reliable data. They suggest to filter non-gaussian distributions using low-percentile filters, which exhibit significantly less noise. Study also suggest that when comparing whether two sets of messages had the same response time those filters should be used [18].

Mona-timing-lib [20] is a simple python library to measure time taken from request to response. Furthermore, mona-timing-report [21] application generates statistical model from taken measurements.

3.2.2 Cross-site timing attacks

In a cross-site timing attack we have no direct access to the measured application. The attack is indirectly performed in a victim's browser against a targeted web service. In this way, we abuse a trust between a victim's browser and the targeted service. Requests sent by the victim are authenticated and actions are executed in a state the victim has in the targeted website. [14]

Cross-site timing attack imitates a cross-site request forgery (*CSRF*), where victim's browser is triggered to perform cross-site requests over third-party-service without the victim's knowledge. However, cross-site timing attack is not targeting cross-origin page writes. It is targeting cross-origin write timings and is thus a bit more complex. [14] In a classic way, the time measurement includes data handshakes, transmission, and processing time in both server and client side, and rendering time in the browser side. It would grow up the attack surface when all those timing factors could be measured individually but in many cases, the browser does not offer direct interfaces to give those details. The leaked information can be relayed back to the attacker. Like in any timing attack, caching and other optimizations in the processed paths affects to the browsing performance and thus most likely are leaking timing information about the processed tasks, especially when speaking about effective page caching mechanisms.

In a related research chapter we introduce some known attacks, but in short, timing attacks poses an imminent threat to a online privacy of the users. To name a few, they provide methods to discover browser owner's partial browsing history, personal information, including age, location and interests of a user [15].

3.2.3 Browser-based timing attacks

In addition to the attacks targeting to the server side timings, browser-based timing attacks are targeting how long the browser process resources internally. Timing starts instantly after resource has been downloaded, and stops after the resource has been processed, which may include caching, parsing and rendering operations. The attack targets internal browser mechanisms that are not relying on the unstable networking. Also, it is much faster to perform multiple internal measures than launch hundreds of requests to an external service [15].

3.2.4 Threat model

General cross-site timing attack threat model has an attacker that controls a web server and hosts a malicious site. The attacker lures victim to his page by using social engineering or by compromising some third party site used by the victim. The attacker run javascript code in the victim's browser, but can't bypass the same-origin policy directly. The timing attack is utilized and the attack takes as long as the victim keeps the malicious site open. Otherwise, the browser is expected to disconnect any sockets related to the session and close any scripts running in the background.

3.2.5 Timing challenges

The cross-site timing attack is performed in the victim's browser against known third-party service. Victim's environment has multiple factors that are making reliable attack challenging. Environmental variables changes, such as, operating system, the browser product and version, browser extensions, network performance, cache aggregates, round-trip time (RTT) and various network connection features, including latency, jitter, network congestion and packet dropping stability. There are also challenges that are familiar from the direct timing attacks, including server error factors like constantly changing server loads and internal background tasks that consume capacity and delay the response. [15]

An attacker may manage to overcome various timing challenges by launching multiple measurements and apply statistical methods. However, the attack time frame is often short and redoing measurements cause loss of time available and may undermine the success of the attack. Furthermore, constantly repeated requests may trigger server defence mechanisms and connection attempts end up as blocked state. [15]

In addition to timing errors, a targeted feature is often not directly measurable by provided browser API and attacker may have to find circuitous paths to make required measurements. For example, the server processing time leaks information about the complexity of the asked resource. Obviously, we have no direct API to measure server delay. We can't even measure latency to the server like ping command does.

Fortunately for attackers, many features could be measured in a different manner, for example to measure ping we could check current time, build a request that requires server to use minimum amount of resources to send back a short message and later check time again and calculate timing difference. This measurement is actually known as HTTP latency that may be used to eliminate jitter from the transmissions. Furthermore, if the HTTP latency

is minor or stable enough we can measure server processing time by issuing various operations to the server and measure how long the operations took. In practical part of this thesis, we introduce an attack utilizing this kind of leakage.

At least Firefox has Navigation Timing API that has large number of measurable events including domain lookup, request loading and DOM parsing. However, the interface respects same origin policy which can be broadened by including time-allow-origin header in response. [22] Attacker has no ability to set that header, and thus has to use different methods to perform measurements. One way is to start a timer, include cross-origin resource to the DOM, wait until browser loads content and rises *onload* callback to stop the timer.

3.2.6 Mitigate timing attacks

Web user has many defense methods available for mitigating against cross-origin attacks. At first, most simple way is to mitigate risks is creating separate containers for different kind of sites. This is achieved by running multiple browsers, using private windows, or multiple profiles side-by-side, that are using different completely separated cookie jars. For example, Firefox has containers feature, which enables users to use multiple sessions on same site simultaneously. This gives ability to segregate site data for improved privacy and security. Furthermore, we recommend terminating web page session after it is not required anymore.

Secondly, browser third-party cookies should be disabled. This would drop cookies from the requests that are sent to the cross-origin targets. Those requests would be performed as unauthenticated user and thus resulting timings would not reveal targeted user web application state. Disabling third-party cookies would also prevent other cross-site attacks for very similar reason. Major browsers, including Firefox supports disabling third-party cookies but they are not disabled as default, because of potential compatibility issues within existing sites.

Thirdly, page cache should respect same-origin policy. However, blocking cross-origin cache usage would issue more than 50 % performance overhead among Alexa Top 100 websites [23]. No mainstream browser has adopted defenses against cache issues.

Web service providers should not trust that every user protects their sessions. Fortunately, many server side solutions against cross-origin timing attacks has been proposed. At first, service provider should utilize basic *CSRF* protection that prevents third-party pages from generating requests to their content. The *CSRF* token is introduced in more detail in session

handling chapter. The protection has restraints, but we recommend this option. We return to this protection in the Limitations chapter of the Cross-Site Search Extractor implementation section.

Secondly, Sebastian Schinzel introduced an effective defence mechanism against direct login timing attacks where the main concept is that the given input, for example user name is salted, hashed, and later casted to bounded integer. The response is then delayed by the amount of that integer. This way, all queries, regardless of their actual status have unique delay and timing comparison could not be done. [24] It should be noted, that simple random delays without any logics are neither effective, nor an efficient mitigation for timing attacks. The random delay brings only a little more noise, which are efficiently filtered out using statistical methods. [25] It should be noted, that the same statistical methods works when some of the user input does not alter the performed action time but changes the hash and delay.

Thirdly, a mitigation method is to ensure that all inputs given to the implementation takes the same amount of time. The execution time is measured and remaining fixed time is added as delay at end of the execution. [14] However, this is ineffective and in most implementations impractical.

3.3 Related attacks

In this section we discuss some related research about the effects of timing attacks on browser security. We introduce what kind of cross-site and internal browser timing vulnerabilities there have been and what kind of attacks they provides. Also, we introduce a linear comparison vulnerability method that is applied and exploited in the cross-site search extractor implementation.

3.3.1 Traditional cache-based timing attacks

Browser cache-based timing-attacks take advantage of time difference required to load external resource. Caching purpose is to make access to visited files faster. Due to the cache properties cache hit is available a lot faster than cache miss where resource is loaded from the net. This side-channel exposes whether victim has ever loaded URL before. The attack could target any URL that offer static file that browser is willingly to cache and measure whether the are already in cache. [15]

It is possible to load static content as an iframe and measure the loading time. The same measurement can be accomplished using external stylesheets, scripts, images and any other html entity, which perform cross-domain request while leaking the response time. The caching content may be exposed

by loading targeted resource twice, and if the response times differ less than threshold, the content was part of the cache. The issue has been known more than 15 years but the fix is not easy because the caching is part of the browser design. [15]

In the end of 2015 Yan Zhu released Sniffly attack tool that reveals browser history by loading bunch of http services, which support HTTP Strict Transport Security (*HSTS*) and time how long it takes for browser to be redirected from HTTP to HTTPS. Based on that, Sniffly decides whether the browser has seen the domain before and is thus previously known HSTS domain. Content Security Policy (*CSP*) directive trick filters targeted favicon images to HTTP content, which would in context of HSTS otherwise redirected automatically to relevant HTTPS site. [26]

The Sniffly listens how long the load process took, and if the amount is in order of a millisecond the network had no chance to perform request and receive response so fast, which means it was cached *HSTS* redirect. The cache hit indicates that the user has visited a page before. Because of shared *HSTS* cache, the attack works over incognito session and even reveals a subset of sites user has visited in non-incognito session. For successful attack the vulnerable pages must have *HSTS* security header enabled. [26]

3.3.2 Rendering timing attacks

In the past, it was possible to perform simple JavaScript check whether user has visited an URL by calling Window.getComputedStyle [27] method, which gives values of all applied stylesheet properties and basic computations it may contain. The method reveals directly whether the link had visited or unvisited styles. This was a huge privacy issue because any website could ask if exact URL is in the user's browsing history. Curious site could scan a large number of pages and resolve part of the browsing history. By experimenting, it became clear that a common browser could perform more than 200k URL checks in a minute. [28]

At the beginning, the browser developers did not see threat a sufficiently enough and for example, Mozilla had 10 year old bug record discussion about the issue, left unfixed. However, afterwards the impact was seen more evident. The original attack scenario of the history stealing was to figure out your online personal banking site and perform corresponding phishing site. The real impact become clear later, when history stealing attacks were combined with a knowledge of social network web applications. [29]

Research managed to deanonymize user identities based on their social network habits. In attack, a malicious site enumerates social media profile pages with different id values. Attack targets social media pages, that are

very likely visited only by the profile page owner. For example, resource editprofile?id=1338, and when the visited hit occurs it is pretty clear that the user identity has been revealed. It would be very unlikely that the other users would have any browsing hits to a edit feature of a profile page other than his own. Their public experiment de-anonymized 10k targeted volunteers and correctly identified 1.2k users, meaning 12% overall success rate, and they found traces about 3.7k users. [29]

As their presentation reveals, the practical attack requires optimizations, because social media sites are having too large user space. The attack is optimized by enumerating social groups rather than users. Finding a visited groups in the history indicates a membership, and by listing group members and figuring out intersected group users it was effective to test user candidates and reveal victim. [29] The vulnerable method were fixed in 2010 by reducing access to style properties, and by ensuring that Javascript API calls behave always like a link is unvisited [13].

3.3.2.1 History stealing via render timings

In July 2013, Paul Stone published a Pixel Perfect Timing Attacks with HTML5 writeup. The research found that a timing attacks could be used efficiently for sniffing browser history. The rendering timing based history sniffer was demonstrated to perform 13 to 20 URL checks per second. [13]

The pixel perfect timing attack brings a single URL one by one to a site and searches for a browser redraw event. The Firefox performs an asynchronous database lookup from history database for each link and decorates the style to match a 'visited' or 'unvisited' state. The unvisited style is used as default and later when lookup finishes as positive, the style is redrawn to visited state. [13]

Detecting a redrawn event is seen by using window.requestAnimationFrame API [30] that introduces a browser to call a specified function to update an animation instantly before a new repaint occurs. Using this method, they managed to measure time difference between callbacks and detect repaints. However, rendering state for a visited style is a short operation, which could not be measured reliably. [13]

To make those rendering repaints more detectable they managed to slow down process by introducing complex styles to the visited fonts and include for example expensive text-shadow effects with large blur-radius. To make attack more efficient, the the rendering delay should be observable but as small as possible. [13]

Target browsers have inconsistent performances, thus the research suggest to run calibration stage and find optimal values before running actual

measurements. They also bring up the idea to improve performance by introducing multiple different URLs at the same time to the browser, and if the detection happens then that item could be found using binary search. [13]

3.3.2.2 Pixel Perfect Timing Attack

The same Pixel Perfect write up published an another timing issue that relates to SVG filters. The research found an issue that the complex visual effects can be applied to the cross-domain content. Those effects could be done so challenging that rendering time could be made to leak information about single pixels they are operating. Issue can be applied to any cross-domain content including images and HTML content, where single pixels are measurable. This is browser based issue, because same-origin policy should prevent any read access to cross-domain content. [13]

Furthermore, they tricked browser to show source code of cross-origin page and managed to target view to a single source line using fragment identifier with specific line number. Also, they managed to perform optical character recognition (OCR) over the source code. In general, when the font is known, a text from a character set of size N can be read a character by testing only $\log_2 N$ pixels, which even more accelerated reading speed. For example, if there are 16 characters, then only four pixels are required for each letter. As a result, by reading pixelated cross-origin document sources, they managed to steal generic user identifiers and sensitive session tokens. [13]

3.3.3 Geo-inference cache timing attacks

Another study reveals that malicious site could sniff location-sensitive resources left by the location-oriented sites. The location resources are not exposed directly but they are static map media files that are included in browser caching and can be exposed similar way as browser cache sniffing. [31]

Researchers managed to localize volunteers using time-based cache sniffing against location-oriented sites. No any other details, including GPS or IP address were used. They noticed, that many sites are geo-targeted. Their study reveals that 62% of Alexa Top 100 websites contain location-sensitive resources. For example, at country level, Google has 191 geography-specific domains and user is redirected to local one. [31]

They managed to expose victim's country by enumerating all Google's logo images from their different domains. After knowing the country, the

exact city was found advantaging popular sites offering city-specific pages, for example service Craigslist. It is used to sell and buy local items. [31]

Furthermore, they extend study to online map services. They discover, that Google Maps has tile based map engine, and the map is divided into a huge number of small static image tiles. In addition, at default, Google Maps zooms into the user's current location. Thus, browser cache sniffing reveals recently visited areas. Therefore, finding visited tiles most likely reveals neighborhood of the user. [31]

3.3.4 HTTP Encrypted Information Stolen through TCP-windows

A research exploited combined flaws in browser and in underlying network protocols. They managed to reveal exact byte count of any cross-origin response and use this leakage to steal cross-site session tokens. This HTTP Encrypted Information Stolen through TCP-windows (HEIST) brings network-level attacks to the browser. [32]

They advantaged a new browser feature, Service Workers. Especially its interface Fetch API, which performs arbitrary network requests similarly like more commonly known *XMLHttpRequest* interface. Fetch is native implementation for performing *XMLHttpRequests* against a server. In this timing research context, the most important difference is that fetch launches a callback event instantly after first byte of the response is received and in addition, a next after whole response is received. Thus, while other APIs could measure only a moment after the whole response is received. This fetch expands measurable cross-origin time scale. By timing first and last byte, they learn that the difference exposes whether the bytes were encapsulated in the same TCP segment, which in most systems is approximately 14kB. [32]

They figured out, that if the web service has simple feature that echoes controlled parameter back to the user. It is possible to grow that parameter in a controlled manner, and with knowledge about TCP flow and TCP congestion window, they managed to recognize TCP segment split moment, and thus succeed to find out original byte count. [32]

They also suggest that *HEIST* could be used to take advantage of *CRIME* attack, which leverages compression rate of HTTPS responses to extrapolate secret messages from the responses. The original *CRIME* attack requires an active man-in-the-middle node, while this *HEIST* could be performed only by luring victim to malicious site. [32]

Now, when the service compress the response, the attacker sees the amount of bytes transmitted, in a compressed form. By testing different parameter

payloads, the attacker learns how efficiently each payload is compressed. Better compression ratio indicates that the iterated payload is included in the response more than once. By this assumption, they could abuse cross-origin restrictions and find out cross-origin session secrets. [32]

3.3.5 Linear comparison vulnerability

A common comparison timing vulnerability exists in many major programming languages. When the string is compared to another, a simple default compare function utilizes break-on-inequality algorithm. Thus, a byte array that shares no immediate prefix terminates instantly. Array that has only the first 15 bytes of the common prefix breaks after 16th comparison operation, instantly after the first unequal byte. Each comparison consumes time, short, but measurable time. Thus, there is a timing attack on the string comparison [12].

A linear comparison vulnerability in password authentication is a fatal security issue in any local and low-response network. The reason is, that the attacker may first guess deterministically following passwords: aaaaaaaa, baaaaaaa, caaaaaaa, daaaaaaa, and so on and chose the password candidate that took longest time to process. The attacker then knows the first character of the right password, and moves to the second character. For example, if the correct password begins with letter d, and it has been correctly recognized, the following guesses would be daaaaaaa,dbaaaaaa,dcaaaaaa, and so on. Step by step, and letter by letter the password leaks to the attacker and she may pass the authentication state and continue to the service.

Nowadays, most services are not saving passwords as plaintext anymore, and each of them are at least converted to a result of a strong cryptographic hash function and to authenticate a user, the password given by the user is hashed and compared to the stored hashed version. This way, if the server is in some day compromised, the password has a bit more protection, because the secure one-way hash function has no ability to convert password back to plain text. The attacker must attempt to hash every password combination and generate their hash values to figure out a matching password. [12]

In general, hash functions have no data-dependent timing issues. However, if the hash values are compared using linear comparison as before, the outcome is vulnerable, again. The attacker must guess the hash prefix step by step instead of the password prefix. The strong cryptographic hash function makes the attack to take longer, but still, the online guessing work can be transformed to a magnitude faster offline attack. To reduce the amount of offline load, the attacker may filter a password candidates from a large word lists and test only the most promising ones instead of generating all

character permutations.

In the middle of the 2013, the pseudonym aj-code published a tool TimingIntrusionTool5000. He states that with a certain compare functions a more correct password takes longer to compare than a less correct password. In his measurements, the single letter difference is in the order of 5 to 100 ns. By taking multiple measurements and filtering out jitter outliers by the 10th percentile rule he manages to reveal password of simple Python Socket Server that reads a password from the network, compares it to the hard coded password and responds with true or false. In a less than 10 minutes, he manages to resolve eight character plaintext password using 673506 measurements in total over 100Mbit local network. [33]

The published implementation does not manage to perform attack over remote web app, but the author believes that the concept outperforms a simple brute force attack over Internet and that the issue exists even after the real world exploiting it is hard. [33] Real world attacks are mitigated by following security recommendations to use account specific unique password salts and prevent login attempts after enough guesses.

3.3.6 Cross-site search attack

The study advantaged search timing side-channel attack. The attack abuses search functionality, which leaks sensitive cross-domain information via timings. They abused cross-origin search features and found out that many sites leak sensitive information about current site state, including user's contact names, email contents, relationships, search history and other personal structured information. [34]

They succeed to perform boolean queries against Gmail and Bing accounts and managed to reveal whether victim's web mail has specific phrases in email messages and whether email relates to particular sender. Also, they managed to reveal victim's first and last name, with 90% success rate. The attack utilized advanced search features and measured the differences in response times depending on the hits of the result. [34]

Sites usually protect sensitive state-changing cross-origin requests by verifying *CSRF* token validity. However, it is common that requests that do not change state are not cross-origin protected, which is true for at least with search functions. In this cross-site search attack, called as *XS-search*, attacker's goal is to detect whether a specific search word has results or not in victim's browser targeted to a specific web application. The attack iterates over studied wordlist and reveals words that are included in the targeted context. [34]

They encounter issues including noise, small sample size and inaccurate

measurements. Large scale of tools were built to deal with the challenges. They evaluated several *XS-search* optimized statistical tests, response-inflate and compute-inflate mechanisms, and divide and conquer algorithms. [34]

3.3.7 Resource load timings

In a study researchers experimented how the different html element embedders are affecting to the measured load time of arbitrary content, when their actual outcome is ignored. They figured out, that cross-origin response sizes are measurable. [15]

In a most classic way, the browser is instructed to embed cross-origin image resource and measure the amount the load operation takes. The measurement works against any content, even when the target resource is not expected image, because the file is completely loaded before the parser notices the incompatible format. The research states that less than 15kB difference is not measurable in a classic way. [15]

They found out, that the HTML5 introduces <audio>and <video>element embedders, which issue an event instantly after actual content is downloaded, and afterwards when the content is parsed. They noticed that in video element, a larger document takes more time to parse, and the parse timings are less distributed (have a smaller standard deviation) than in a case of resource download time. Using this knowledge, the network connection and server jitter could be left out from the measurement time and thus perform more accurate timings. They found, that especially for resources with a small difference the browser-based attacks offer much accurate timings than network based measurements. [15]

Another finding they made, was that attacker can force an external resource to be cached in a context of his attack site. They figured out that writing and reading offline cache is measurable and results depend on the content size. By performing those operations multiple times, they could estimate the size of the accessed file. In their results, time required to perform AppCache-based timing attack with 95% accuracy to detect 25kB difference took less than 2.5 seconds in various environments. [15]

As proof of concept, they published an online browser-based timing attack page, which determines your login status to Facebook, your Facebook gender and political preference based on Twitter followers. They used described observations to perform those tricks. For example, to reveal gender, they published two pages in Facebook, where one was limited to male readers, and another to females. Both messages were huge, which caused that based on the gender, the response sizes were either 110 kB or 40 kB. Observing the timing difference they managed to deduce the gender. [15]

Chapter 4

Tools for timing attacks

In this chapter, we introduce built tools in more detail. At first, we are going through measurement tool, which is used to find cross-origin timing vulnerabilities. The measurement tool is a web page which is run in a browser like any other web page. The page generates requests to the targeted service which later responds and leaks timings. Those timing leakages are based on the combination of vulnerabilities in both the browser and the target application.

The second implemented tool is also a web page and starts exactly where the first one left. The tool advantages timing leakages and performs an actual attack targeting vulnerable cross-origin search page. We present how the implementation works and especially how it recovers from time to time occurring faulty measurements. We briefly remark some issues, which rise up during testing and building the implementation.

Later in this chapter we introduce previously unknown security bug in Firefox's implementation, relating how the browser fails to handle specific connection sockets properly, and how it affects as an security impact. We expose how the vulnerability could be exploited.

4.1 Load time measurements

We made a tool to find and inspect cross-origin timing attack to circumvent same-origin policy. The key factor is to run Javascript on victim's browser and using Javascript to perform cross-origin requests and recognize whether two cross-domain payloaded requests differs in context. This knowledge could then be exploited further as is shown in the following search extractor experiment.

We built a web page measurement tool that is used to reveal whether

two targeted cross-origin page contents are similar to each other. Browsers do have a same-origin policy security feature, which should prevent this kind of information leaks. However, the tool abuses request timing side channels, which in specific scenarios seem to reveal more than enough information for a practical attack.

There are many logical reasons why timings are leaking information in a general application. For example, a private mailbox web service could have a search form to query content from user's private mails. When a browser issues a query, the server might take a bit longer when the search finds content. That is, because server might run a different code path and perform multiple extra database queries to provide more detailed answer. Furthermore, when a hit is found, the response size extends and transmission time takes longer. Also, in that case, the browser must render a bit more complex page, including potential related hyperlinks, thumbnails and other context.

The introduced measurement tool enables us, as security auditors, to understand request timing issues in practical context and expose what are the indicative content boundaries that are at least vulnerable. At this moment, we are not clear how small content differences are measurable. Also, we are not aware that the industry has any public tools to measure those cross-origin timings and very likely there are web pages vulnerable to this issue. This implementation helps us to reveal those issues and keeps us more protected in the future.

In order to measure cross-origin response similarities we are comparing two pages in a same application. Timing variables are highly dependent that the targeted application is same. Thus, we can point out whether two pages differ in the same application, but we cannot measure if two different application pages are similar. In this context, the similarity is measured upon the difference of content bytes, difference in the amount of inner dependencies (subrequests) and in the time the server took to build the response.

Browser processes payloads differently based on the *html* element, which issues the cross-origin request. For example, timing leakage may be exposed by using *iframe* tag, but while using *script* tag the issue does not exist. Thus, the chosen element affects to the result and to make comprehensive tests there are a multiple elements to study. Based on W3.org [35], the following *html* elements embeds content: *img*, *iframe*, *embed*, *object*, *param*, *video*, *audio*, *source*, *track*, *map* and *area*.

In general, we are interested in any accessible details; variables and features that specific element offers and thus provides additional surface for comparing payload responses. This includes event handlers, progress events [36], attributes, timings and functions. Many features are behind the same-origin

policy and could not be touched behind the cross-origin request, including progress event that has trigger for *loadstart*, *progress*, *error*, *abort*, *load* and *loadend* events. In this thesis, we focus here on timing issues related to element callbacks.

In attacker's viewpoint, we cannot alter the server code, or configure cache settings of the browser. However, we may avoid some browser caching mechanisms by changing a requested address a bit, so that the browser and server thinks a request is a fresh one. Instead, the targeted resource is exactly same, except a randomized parameter at end of the URL that does not affect to the server response but removes potential page caching. Every requested address is unique, thus the browser cache makes a miss.

In our measurement tool, we are measuring payloads one at a time, and comparing two payloads in turns. When measuring two different payload sets, the naive timing measurement approach would run two payload sets separately and compare the sets of timing afterwards. However, we parallel measurements to perform them in turn and run compared payloads close to each other, which leads to a better results. That is, because the momentary network conditions are more similar to each other when those measurements are run between a short time frame. For example, the jitter is approximately same and thus, the results are more precise [16]. Also, the first few dozen measurements should be skipped to avoid TCP negotiation and cache level warm-up outliers. In the following measurements, we are not filtering warm-up outliers out. We assume that the sample size is high enough to filter out any outliers.

4.1.1 Implementation

We developed a system that inspects cross-origin request load timings in a browser context. Our tool measures embedded content elements timing behaviour when they imports another cross-origin resources into the document. Those elements are used to include cross-origin documents and measure how long the operation took. Each element aims to different purpose in browser context, but we are interested how they adapt to our timing research.

Our measurement tool gathers two target address from the user and then utilizes same basic concept the W3C Candidate has introduced, which is a simple attempt [37] to measure time it takes to fetch a single resource. When we perform a measurement, we check current time, introduce a new cross-origin element and assign *onload* callback to measure time difference. The browser scheduler issues request almost instantly, which enables measuring accurate processing time.

We prefer to perform compared requests close to each other to adapt

Element	Description
img	Represents an image
iframe	Represents a nested browsing context
embed	Provides an integration point for an external non-HTML application
object	Represents an external resource based on its type
video	Used for playing videos or movies, and audio files with captions
audio	Represents a sound or audio stream
script	Allows authors to include dynamic script and data blocks in their documents

Table 4.1: Element embeddors

better against environmental changes. If a single measurement performs slower than before, it is likely that the downwarding impact affects to the second measurement also. For example, a Windows update could issue heavy packet transmissions that constrict the available bandwidth capacity and influences to the measurements.

The measurements are performed using element callbacks, thus the main loop of the measurement tool is implemented so that the callback initializes a new measurement from the shuffled work queue. In every row, the implementation shuffles execution order of the elements to avoid potential strange caching errors, which may otherwise recur in every iteration. In our implementation, the *performance.now()* method is used instead of a bit more inaccurate *new Date().getTime()* method, which is introduced by W3C. We chose seven elements for our basis. The chosen elements are listed in Table 4.1. We introduce a couple of those mentioned elements in more detail because of their special properties.

Iframe [35] provides a way to embed another cross-domain site within the current *HTML* document. The outer document does not have access to inner document DOM, but *iframe* offers *onerror* and *onload* triggers to give feedback about the success of the load. *Onload* of the *iframe* fires after all *iframe* content and sub content are done loading. This includes, that the server has generated response, transmitted it to the browser, and browser has constructed a new DOM tree and loaded all inner dependence elements and rendered response to the screen [35]. Thus, *iframe* loading time changes significantly depending on the content that may have a large set of external dependencies.

Script and *img* element requests should be a lot faster to process than *iframe*, because there is only one element that needs to be retrieved. Therefore, this is a lot more robust way to measure server processing time than *iframe*. The same thing affects to the *fetch*, which should also trigger instantly. Also, a study states that the video element processing time depends

Iframe				**Img ...**
URL1	URL2	onerror	onnonload	URL1 ...
a	b	a-b		...
c	d		c-d	
e	f	e	-f	

Table 4.2: Matrix representation

about the content amount, even when the content is not in expected media format [15].

In addition to mentioned elements, we measure parsing time using fetch content downloader and it is included in the results like it would be an element. It measures how long the browser parses. This is possible, because *fetch* triggers a callback as soon as the first byte is received from the server. Later, polling *performance.getEntries()* it is possible to measure when the last byte is received. The same trick is used as part of the the HEIST attack.

Furthermore, it should be mentioned that W3C introduce *performanceResourceTiming* interface that allows Javascript mechanisms to provide complete client side measurements, but that is only accessible on the same-origin requests and thus is not relevant to our study.

4.1.2 Representation

We perform measurements by launching a single request to both URLs iterating over every individual *html* embedder. The response timings are written to the matrix view, where first column has the total time of the request for that specific element for *URL1*, and the second column same thing for *URL2*. The third and fourth columns shows the times separated based on the issued callback. *URL1* time value is put to the raised callback column and same thing with the *URL2*, but here the value is treated as negative value. When callbacks collide the first *URL1* time is subtracted by time of the *URL2*. If only a *URL2* time is written to a specific column, it is negative. Next columns repeats the introduced scheme for the rest of the element types. Each row represent a new measurement.

In the end, this results a matrix view respecting the scheme shown in Table 4.2. Using this representation, every callback column represents a measured feature, and if any those callback columns have a large percentage of positive, or negative numbers, then the response is likely vulnerable. That happens, when tested element is calling different callbacks, or the actual time measurement reveals issue. We know, that this kind of representation does not provide a strong statistical view and more development should be done to

Event handler	Description
onerror	Triggered if an error occurs while loading an external file
onload	Occurs when an object has been loaded

Table 4.3: Callbacks

analyze results more thoroughly. However, current version reveals certainly vulnerable issues and indicates about potential cases.

To summarize, we measure timings and search for a clue that the measured payload affects to the timings. Afterwards, when the issue is found, we may build actual attack and abuse the leakage. We believe, that there are a large number of pages that are vulnerable to this timing attack.

4.1.3 Leaking callbacks

In that measurement representation model, in addition to looking timings, we compare triggered callbacks. We catch callback element calls and try to identify responses from each other without potential unstable time measurements. We are interested about *onerror*, and *onload* events and in this context, the callback is either *onload* or *onerror*. Those events are listed in Table 4.3.

In future, there might be more callbacks we are interested. W3C state that every HTML element must support *onblur*, *onerror*, *onfocus*, *onload*, *onresize* and *onscroll* event handlers, and their specific event handler event types [38].

In general, the *onload* [39] event triggers when an external document is loaded, parsed and presented properly. Otherwise, *onerror* event is triggered. *Onerror* is triggered when the content is not reachable (indicated with 404 - not found status code), or in a supported format, or is malformed. The W3C specification does not cover precisely, which are the exact scenarios when the *onload* and *onerror* should be raised and thus the browsers are not working consistent with this case [40]. For example, when loading external document, should an empty response be classified as *onload* or *onerror* event?

A social network service is vulnerable and leaks user login status when profile image is served normally for logged-in user and otherwise not found error is raised. Advertising page could perform a simple check and bring like-button ads more visible when user is known to have active Facebook session.

That status code leakage is server side implementation or configuration issue and should be fixed. The status code catcher is robust, takes a small effort to implement and there are no similar challenges that occurs with

timings. However, there are a lot of pages which do not have status code leakage issues, but are affected to timing attacks.

4.2 Cross-site search extractor

In previous section, we introduced a tool that digs timing differences from targeted web applications. In this section, we introduce an another tool, which is made as a proof of concept to take over found timing leakages and make them to a practical attack. The purpose of this tool is to extract data from targeted web application in a way that the author of the page and/or browser has not been taken into account.

We introduce a cross-site search extractor attack, where secret cross-origin sentences are stolen by studying resource load timings. The concept is to take advantage of web page search functionality and by issuing boolean queries character by character construct a cross-origin secret tokens. The attack reminds cross-site search attack, which reveals whether some chosen exact keyword exists in a vulnerable service. This is introduced more specifically on the related research chapter

Our attack is not limited to a wordlist, the attack is able to construct words and sentences without previous knowledge about their actual context. We construct those secrets by using similar concept a linear comparison attack utilizes. Thus, the known cross-site search attack is made more lethal by combining these known attacks together. We call this attack as cross-site search extractor, to avoid confusion to the simpler wordlist based attack.

It should be noted, that in this case, significant time difference does not occur because of the optimized compare function that uses before mentioned vulnerable break-after first inequality method. The timing difference is significant, because the payloads are for example running different code paths and issuing different amount of database queries. This is described in more detail in load time measurements chapter.

The following fictitious example puts our attack in a practical context. A victim has a web mailbox which includes a powerful search engine. Victim receives email containing link to a site that promises cute premium rating cat pictures. The victim follows unique link to the attacker's page and surfs those photos. In the meanwhile, attacker recognizes that the unique token is loaded and resolves the victim's email address. After the email is known, the attacker launches a forgot password query to all popular sites using the victim's details.

The mailbox fills up with the password reset URLs including the secret authentication tokens. The attacker continues the attack as long as the user

stays on the page, or until the attack success. For example, victim receives following email:

```
Dear user, someone recently requested a password change
    for your account. If you don't want to change your
    password or didn't request this, just ignore and
    delete this message.

If this was you, you can set a new password here:
https://example.com/reset?token=d7a8fbb307d7809469ca9a
    bcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592

To keep your account secure, please don't forward this
    email to anyone. See our Help Center for more
    security tips. Thanks!
```

The attacker attempts to resolve those secret tokens and thereby steal valuable user accounts. The attacker has no direct access to the secret token. However, he is able to run javascript code on victim's browser that is known to have active session to the targeted webmail.

Attacker exploits cross-site search attack against search functionality of the mailbox and finds all services that are triggered to perform password request change. The search could be limited to recent mails with a known targeted senders.

After a while, the attacker finds password recovery mails and builds a search query pointing to single recovery mail, while including a keyword to the known prefix of the secret authentication token. In our example, the prefix is 'token='. After, the attacker grows payload character by character based on the measurements and finally retrieves the secret and launches the password reset successfully.

4.2.1 Breadth-first search algorithm

For a environments having nearly minimal jitter, a simple heap algorithm works well. The algorithm flow is represented in Image 4.1.

In every iteration, the largest time candidate is popped from the heap. The candidate URL is measured and for each character available a new candidate is built using same URL prefix where each character is appended one at a time. All those candidates are copied back to the heap including a fresh timed value. After, a new iteration begins. In this way, this breadth first search (BFS) alike algorithm navigates promising paths. The algorithm

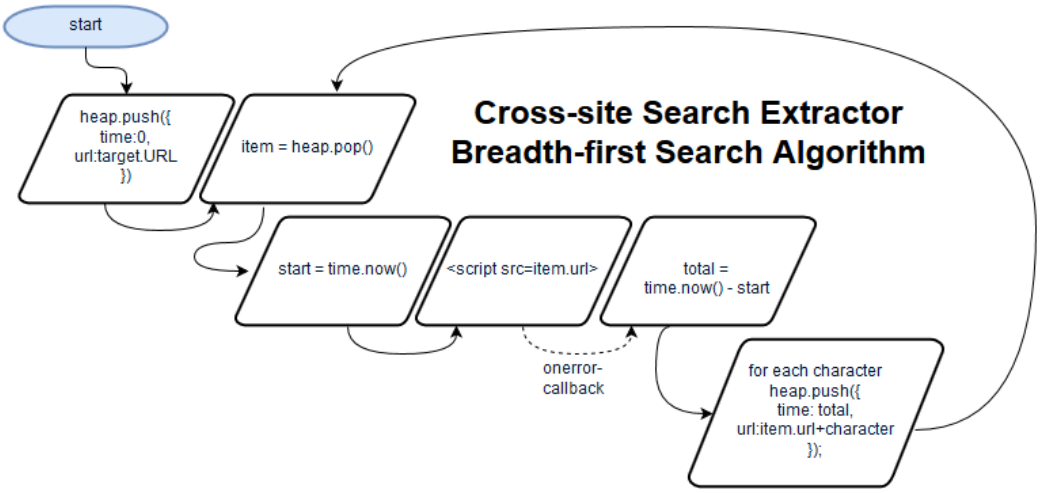

Figure 4.1: Cross-site Search Extractor:Breadth-first Search Algorithm

has no trigger to detect when the iterating should be terminated and thus continues forever.

Faulty measurements on the correct track cause issues when the measured value settles lower than some incorrect path measurement, that is because it is unlikely that the algorithm will come back to test that branch and thus the rest of time the search will spend building junk. In addition, faulty measurements from incorrect paths are not an issue. When the algorithm takes a wrong path, it iterates over whole alphabet but finds no large measurements and jumps back to the previous track.

The mentioned algorithm has no ability to recover specific faulty measurements. When measuring over various environments and especially over Internet, it is likely that after hundreds of requests a fault happens. The special constraints of the varying network conditions demands a more appropriate solutions for targeting browsers over Internet.

4.2.2 Depth-first search algorithm

At finally, we build a bit different approach that implements depth-first search algorithm and deals better over varying network conditions. The implementation verifies a chosen suffix before accepting, and furthermore is able to survive from any single faulty measurement by observing the overall situation. The algorithm flow is represented in Image 4.2.

In every iteration we are growing prefix with a character that took the longest time of the round. Depth-first search (DFS) algorithm takes the

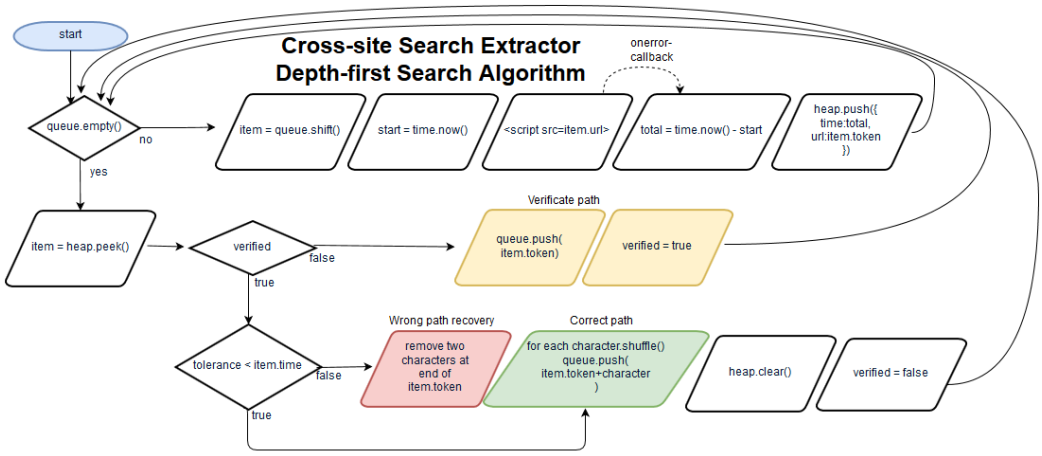

Figure 4.2: Cross-site Search Extractor:Depth-first Search Algorithm

character from the heap, just like breadth first search. The difference is that the heap is cleared after every time a character is inserted or removed from the chosen prefix path, so actually, a heap structure could be replaced with a max-function. In a single iteration, the payload prefix continues a character forward or backward. The backward step happens when algorithm thinks that it has took the wrong path.

The wrong path is recognized in a verification step, which is done instantly after a best character candidate is chosen from the heap. The chosen prefix including best candidate character must take longer time than a fresh measurement with an empty reference payload. The path is wrong, if the precondition is not met in a couple of measurement attempts. In this verification step, the algorithm could be done to performance better over short delays by ensuring the verification step using stronger statistical methods before proceeding.

The empty payload works as reference time, and it should be refreshed regularly to react network changes. This could be done, for example, in correct path stage by including an empty payload to the queue, and afterwards just update the tolerance value based on that.

The reference payload should be similar order of magnitude than the (correct) path we are validating, and thus the algorithm reverses pointlessly. However, the unnecessary backward step probability is decreased by increasing verification reload attempts and by tolerating a bit larger variation, in example, accept measurements that are 20 milliseconds faster.

The algorithm is optimized to ignore rest of potential characters instantly after a measurement triggers a time that is larger than previous reference

time. In that case, the verification should be done normally. This optimization terminates a search on average after $alphabet\_size/2$ character measurements. The expected total average measurement count drops to $(alphabet\_size/2) * token\_length$. In order to maintain a clear basic flow of the algorithm, the optimization is not included in the presented flow image.

To simplify the main points of the implemented DFS Cross-site search extractor algorithm:

- In every iteration, algorithm tries all charset characters one by one, and before a next iteration continues by adding slowest character at end of the current prefix. However, before character is accepted, it has to exceed measured reference point during two verification attempts
- If reference point test fails the correct path is lost, and a character is removed one at a time, until reference point test succeed
- Reference point is time taken from a static empty query that is known to be correct and which it is updated regularly to react fast changing environmental delays
- Algorithm moves to the verification step instantly after measured character exceed reference time (optimization)
- No two measurements are performed at the same time. No a new resource load until previous has finished or errored out

The practical implementation is a bit more complex, because the javascript content timing requires event based approach, thus the main loop is actually an event-loop, where a set of event-triggers are calling each others.

4.2.3 Setbacks

During developing and experimenting it became clear that measurements do have many errors factors and proper error handling is required to survival from the faulty measurements. In every iteration we prefer character that took longest time of the round, and because of the network congestion, browser scheduling and other delays it is likely that in time to time a wrong character hangs a bit longer and a wrong prefix path is chosen. We succeed overcoming those issues by implementing verification and recovery state to the DFS algorithm.

The implementation generates hundreds of elements to the DOM. Single element is removed instantly after single measurement is done, but anyway,

the browser slows down little by little. Thus, to avoid extra jitter between experiments, the browser should be restarted every time. Furthermore, the developer mode, including debugger and network activity window causes a significant jitter that makes the attack fail.

The ideal implementation should adapt in various environments and work over different connections and browsers. A larger jitter could be filtered by performing more measurements, but it is a trade-off between amount of required measurements for unique payloads versus amount of time required to measure a single payload when limited total time is available. In future, we could make algorithm to dynamically adapt for those conditions by issuing more or less requests during verification state.

4.2.4 Limitations

Our implementation assumes multiple things about the environment. At first, underlying *html* elements operate over *HTTP GET* requests and for this reason, our attack targets search implementations that supports *GET* requests.

It is required, that the search functionality works over partial search words and that the search query length has no significant effect to the timing. In addition, active browser session and stable enough network connection is required between browser and targeted web application. Also, the search must have a measurable timing difference between results versus no results.

Another limitation is that the attack is mitigated, when the search functionality is protected with *CSRF* token. More details about mitigating this issue is represented in chapter 3.2.6. In many situations, the service provider would like giving user an opportunity to add custom search keyword page to a browser favorites, but this expiring *CSRF* token would make it fail.

Alternative server mitigation implementation, is to prevent cross-origin embedding and implement query field in an URL fragment section that is located after hash(#) character. *Iframe* timing attacks are one of the multiple reasons to disable cross-origin *iframes* using server security headers. Furthermore, because the javascript builds the actual query request, it is simple to add *CSRF* token to the actual API request.

When the query is in the fragment section, the query is available only for same-origin javascript, and the server will not receive it, nor process it. When using content embedders like *script* and *img-tag*, they are not executing javascript which is mixed inside *html* content, thus there are no place for leaking timing differences.

4.3 Mozilla Firefox socket timeout issue

During this thesis write up, we managed to find serious security vulnerability from the Mozilla Firefox browser product. We go through thinking process and list how a minor socket timeout issue were eventually evaluated to a serious vulnerability when exploited server response delay. We also introduce actions the vulnerability process management included.

The presented issue is not a timing attack, but it has many properties that are discussed in this thesis. For example, a related research chapter includes clever ways to escalate security issues, the concept of sessions and their relation to the cross-site request forgeries are already explained and a concept of manipulating operations for performing slower and thus achieving a better measurements is clear. Thus, we decided to introduce the issue in this thesis, immediately after practical timing tools are dealt with.

4.3.1 Favicon socket handling

We found out that the Firefox launches *favicon* request which will not timeout or close when related browser window is closed. The *favicon* is a small website icon associated to a specific website, which is shown in the browser's address bar. At first, the server could spy how long the browser stays open after page close by delaying initialized transmission indefinitely and wait until there is a moment when connectivity problem occurs.

The issue sounds like a minor issue but it does have a bigger security impact. One possible exploitable scenario exist in the Tor Browser product which is based on the Mozilla Firefox and thus inherits same issues. Tor Browser anonymizes user browsing and offers multiple security hardenings to hide any privacy leakages that could help third-party server owner to classify unauthenticated browser sessions together.

Tor Browser has a new identity feature which cleans up any active session cookies, temporary environmental values, forces a new ip address and even makes browser to do soft reboot which closes all active windows. As you can guess, there is a fault that the *favicon* connection is not terminated after soft browser reboot which gives server owner ability to deduce that the browser is still open and user may have just tried to hide browsing history by initializing a new identity.

For example, if the user continues surfing on the curious server after soft reboot and after a while closes the browser. The browser closes which in turn closes the original *favicon* connection and the current connection exactly at the same time. Thus, the server owner could easily indicate that the sessions

are ran by the same browser instance. More research should be done to figure out how the leaked online time could be utilized and is there other practical harm to the end user that could be done.

4.3.2 Delayed CSRF attack

We studied the *favicon* timeout issue in practice and found another consequence of the bug, which we are calling as delayed *CSRF* attack from a closed page. The attack server includes a http redirection header to a vulnerable *favicon* retriever and delays the response until desired amount of time. *Favicon* request respects redirection headers and the *favicon* service attach a fresh cookie header to the request. Thus, because of the redirection, we are also able to perform new connections after page close. The attack is finalized by redirecting user to the vulnerable service.

Since, the target service is vulnerable to the *CSRF* the attacker could have perform the attack instantly. However, the point is, that the victim may not have active session to the target service when victim has opened the attacker's page. Usually, attacker could wait and try to exploit *CSRF* issue as long as the attacker's page is open, but our finding leverages the attack surface to the moment when the user has closed the suspicious page. We measured that the attack succeed even after 30 min from the page close.

Since, Mozilla Firefox accepts multiple ongoing *favicon* requests to the same site we can leverage the chance to perform a payload request at a right time period where the user has logged in. In more precisely, we trigger the *favicon* request to load several times and delay the response timings. We delay those requests and release one at a time in a controlled manner, which leads us to perform *CSRF* attack at regular intervals after page close.

4.3.3 Manage over connection drops

An attacker may force browser to perform redirection to arbitrary address even after the browser page is closed and network connection is disconnected for a short moment. The *favicon* service supports multiple ongoing *favicon* requests to the same service, where only dozen of those are requested at the same time. Rest connection attempts are queued in order. We are not familiar how the exact queueing system operates, but the queueing is a impact of browser's max connection limit targeted to specific host.

This makes the attack more interesting, because of that feature our set-up supports short a couple of minute connection disconnects during the attack. When the connection drops, the *favicon* service attempts a reconnect and after a while gives up and moves to a next malicious request. A large queue

performs longer and defeats the network drop better. Thus, when we disconnect a single connection, the *favicon* service launches a new connection from the queue. The queue can hold thousands of requests without a significant performance loss.

There are multiple situations where disconnection may happen during an attack. For example, an victim could carelessly surf on the cafe guest network. Later, when she swaps current connection to a trusted *VPN* connection or respectively walks back to the office the *CSRF* payload is launched and the targeted vulnerable internal service is compromised.

In addition to the method to perform delayed *CSRF* attack to the internal network it is violation of the user privacy because we may track the user across multiple IP addresses as the user changes networks. The fresh IP address leaks to the *favicon* server because when the user swaps the network, the living *favicon* TCP connection drops and *favicon* service launches a new from the queue.

4.3.4 Theoretical follow-up attack

In a previous main chapter, we introduced an attack which extracts data from target page using timing leaks. Later, we mentioned how to launch a page requests from a closed page. Now, we may have a chance to compound the attacks, and perform cross-site search extractor attack behind the delayed *CSRF* attack. The following described attack is theoretical, and not tested in action, thus there is a large chance that it will not operate as expected.

The search extractor attack required Javascript to perform requests, measure timings and to determine how the attack should proceed next steps based on the obtained knowledge. We may have chance to perform this attack without any client side logistic and ability to perform Javascript. The attack would be based on the assumption, that the browser supports a large *favicon* queue and operates new connections deterministic enough.

We introduce a large static page, which consists thousands of *favicon* elements that are targeted to the attacker's server. The attacker has ability to perform page redirection to the vulnerable target server, and control the exact moment when the redirection should be issued. The Firefox browser has a upper limit of parallel connections the whole browser process has globally. Thus, after filling the global connection limit, we may issue a redirect request to a vulnerable server. The current connection disconnects and a new prioritized connection to the vulnerable server is issued. The new connection disconnects instantly after target server responds. The connection is closed, which releases one global connection reservation, which is instantly used to perform a new *favicon* load. After following chain of events, we mea-

sure how long it takes that the browser issues the redirect to the state where the browser issues a new *favicon* connection. The measurement contains the server processing time we are interested.

The attacker server controls browser to create requests to the vulnerable server, has ability to measure how long the request took and build new requests dynamically based on the learned knowledge. Thus, in this threat, we could perform search extractor attack from the closed browser page which enlarge the timeframe available to perform proper attack. The attack is theoretical idea, which should be handled as one. This theoretical attack relies the browser mechanisms and assume the browser behaves as described.

4.3.5 Vulnerability process management

We reported two bugs to the Mozilla's Bugzilla bug tracker. The first reported bug is 1255270 [41] which relates to an issue that *Favicon* request doesn't timeout, or close when related window is closed. The second bug 1255267 [42] notes that it is possible to perform delayed *CSRF* attack from the closed page. Both bugs were reported on March 9, 2016.

We marked them as security bugs which reduced potential observers heavily and made issue hidden from the public during the processing and repairing stage. Eventually, the developers saw second bug as duplicate of the first and handled bugs like an single issue. We had close access to follow whole repair process and participate.

Firefox has a lot of software components, and it took time from developers to find which component was responsible for related *favicon* timeouts. The issue was that there was no original implementation relating to the *favicon* timeouts. Developers had to design and decide where the connection monitoring system should be done. For example, *Mozilla2:ImageLib* [43] is optimised for loading images, decoding and displaying but it doesn't initialize network request. Networking component on the other hand executes requests but has no knowledge about the front-end window navigation triggers, and thus can't decide when the request should be cancelled. The final patch modified *setAndFetchFaviconForPage* network API to make it possible to cancel request afterwards, and added proper *dom-window-destroyed* observer notification to trigger cancel function when the inner window is closed.

Afterwards, fixing the issue turned out to be a bit more challenging than expected. A new bug 1279208 [44] was submitted to tracker to approach bug from a new viewpoint on June 9, 2016. However, we pointed out that even the new implemented patch did not cover all corner cases and the browser was still vulnerable. Two bugs, 1283067 [45] and 1285196 [46] were issued on June 29 and July 7, 2016.

After those, we could finally verify that the patch fixes issues correctly. The bugs were closed, and the vulnerability was announced at the public [47] on August 2, 2016, with a code name *CVE-2016-2830* [48]. The bug was approved to the Mozilla's client bug bounty program [49], and was awarded as an high level vulnerability.

Chapter 5

Demonstrative experiment

In this chapter we are experimenting how HTML element timings are reacting where targeted resource amount of bytes, processing time and document dependencies are altered. We compare how different elements measure this same action. Furthermore, we are also experimenting how presented cross-origin search extractor attack works against simulated web mail and how the server delay affects to the attack. The chapter begins with overview of the measurement environment and proceeds describing features we are measuring.

5.1 Experiment setup

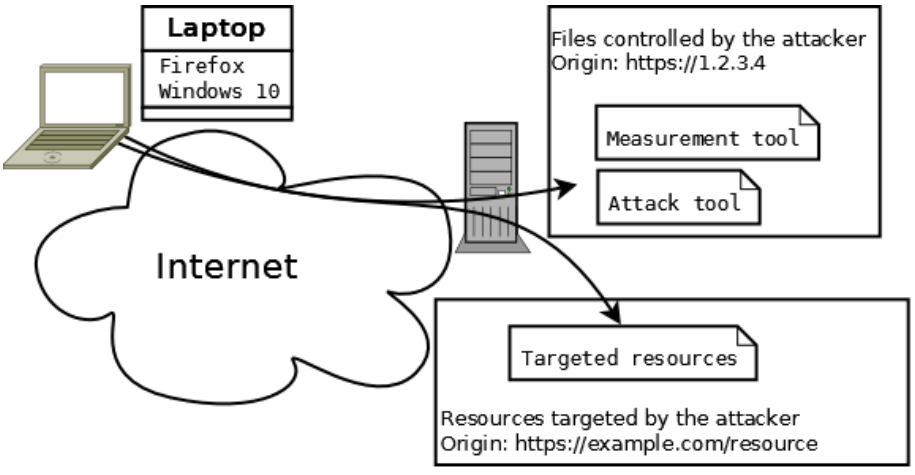

Figure 5.1: Measurement environment

The measurement environment is represented in figure 5.1. It consists of a laptop in office network that has regular Internet connection to a web server located in our home. In this measurement setup, the server serves measurement tool, search extractor tool, all cross-origin test files and mailbox simulation service.

In a regular attack, the attack tool and targeted application are not served under same server. However, we saw no issue that would endanger results because of the chosen arrangements. The measurement and attack tool could have been executed locally on the client machine, but in the end it were simpler to keep them in the same place where the rest of the files are.

Furthermore, because we focus on cross-origin attacks, it is important that the measurement tool has no same-origin access to targeted files. Otherwise, the browser control flow might differ and affect the timing results making them in practice worthless. Thus, we ensure cross-origin behavior by launching tools over direct ip address, and loading all the rest test files over a domain. This way, the cross-origin requirement applies, even the files are served from the same place.

The all measurements were done on a up-to-date Windows 10 on the end of the August. The computer is Dell Latitude E7450 laptop with 16 GB of RAM. The measurements were done on the Ruoholahti office over a wifi network that was connected to a wired Internet. The latency from client to the server was 8 ms RTT, and short test indicates a 0% packet loss. Bandwidth to the server was measured to be 6 MB/s download, and 4 MB/s upload.

Tested web browser is Mozilla Firefox Browser version 45.2.0 that was released in March 8, 2016. We rely on JavaScript events and their timings, which are standardised APIs and widely supported over various browsers [3]. Thus, it is very likely that the following measurements could be performed on any browser, but in order to save extra work, we will specify measurements to a single web browser. The Mozilla Firefox is chosen, because we have most personal experience about it, and in addition, a delayed CSRF attack which is introduced later in this thesis relays closely on its faulty implementation.

The server locates in Otaniemi, which has 8 km distance from the Ruoholahti. It hosts the measurement tool and the targeted measurable resources. The server has Ubuntu Linux 16.04.1 Long Term Support operating system, which serves the files over the Apache/2.4.18 web server. The Apache configs are left as original. The processor is Intel Core 2 Duo E6750 and there are 4 GB of ram memory. The server is connected to the FUNET network [50] and has a gigabit network connection over the Internet.

5.2 Load time measurements

We could go wild and measure how real world applications are handling timings. However, to simplificate measurement scenarios and to avoid launching thousands of requests to third party sites, we decided to make a three measurement scenarios to our server that are requested over Internet in our introduced experimentation setup.

We found three potential features that are at least affecting to the timing. We build a specific measurement scenarios to handle all of them individually. The exact measurement scenarios are in the appendix section of thesis but briefly they are the following:

- A first measurement scenario measures how the response size affects to the response time. It contains a document that has character A 30k times in a row. The document is compared to a document that has 100 A characters in a row.

- Second measurement scenario measures how browser handles subdocument loading and rendering. It contains a HTML document that embeds six internal image elements versus a page with only one of those dependencies.

- Third measurement scenario measures how the server process time effects to the response time. It demonstrates a simplified search engine, which delays the answer 25 milliseconds if the hit occurs. We compare this hit to a query keyword that causes no delay.

In general, time differences are irrelevant and we are more interested in measuring the differences in latency across measured events. Timing results are shown as box and whisker plot, which is ideal for comparing distributions of measurements. The measurements are intended to give a good compact overview about the distributions. Specially, we are interested about median, time range and skews in the data. We are comparing distributions graphically, which adds inaccuracy. However, main conclusions are based on the features that are clearly visible, so we are not dealing them in greater statistical depth. Measurement application provides measurements as comma separated values that are then imported to Microsoft Excel to build graphical plots.

We are performing those mentioned measurements multiple times so it is inevitable that caching occurs in multiple levels. A hypothesis is, that a caching may confuse the results. However, as long as a there is a clear difference in measurement results, the application is vulnerable, it makes no

difference whatever or not a specific cache was used, as long as it is repeatable. However, so that we can make the attack more accurate, we must eventually understand the reason. If the caching could be tampered on purpose, the attack surface grows and handles both scenarios.

Every element has 1000 measurements to both targets, which means that a single graph represents 12000 requests overall. Overall measurements for comparing two pages in application took approximately 15 minutes that is a bit more than 13 request / second. We made three measurement scenarios. We introduce them in the same order mentioned before.

5.3 Cross-site search extractor

We introduced a Cross-site Search Extractor attack that manages to steal secret tokens from specific cross-origin pages using timing differences. In laboratory environment, we may alter the delay difference in arbitrarily manner. In practical attack to a real service, we have no direct method to control timing delays. It is sure that the attack fails when the delay measurements between different responses are small enough. In this experiment, we study how the delay affects to the attack.

We build a vulnerable server simulating a simple webmail search functionality. The service takes a search word as input, and responses whether the server has content about that word. In this experiment, the perceivable delay is added manually to the implementation using sleep command. Now, because we control this value, we can observe how the attack implementation behaves under different delays.

Our load time measurement experimentation points out that there are at least three features that are measurable by timings. In this implementation, we rely on the server delay changes. Our approach is based on the assumption that the server process search query longer when a hit occurs and finishes faster without hits.

It is particularly interesting to find a smallest delay difference where the attack still works, because it implies the magnitude of delay difference required to perform attack. Later, in security assessments, we could just measure delays and with this knowledge, conclude the implementation as vulnerable. The tests are done using a single network connection setup with stable jitter. Thus, it is likely, that the attack may fail even with larger delays when the jitter dominates.

The vulnerable server code is following:

```
<?php
```

```
   $delay = 38*1000; // milliseconds
   $message = "Dear user, Someone recently requested a password
       change for your account. If you don't want to change your
       password or didn't request this, just ignore and delete this
       message.

   If this was you, you can set a new password here:
       https://example.com/reset?token=d7a8fbb307d7809469ca9abcb008
       2e4f8d5651e46d3cdb762d02d0bf37c9e592.

   To keep your account secure, please don't forward this email to
       anyone. See our Help Center for more security tips. Thanks!";

   $u = urldecode($_GET['query']);
   if (strpos($message,$u) !== false) {
      usleep($delay);
      $output = 'Content found!'
   } else {
      $output = 'Sorry, no content!';
   }
?>
<html><head></head><body>Email Inbox - Search engine
    <br><br>Search query: <?php echo htmlentities($u); ?> <br>
<textarea rows='4' cols='100'>
<?php echo htmlentities($output); ?>
</textarea>
</body></html>
```

The vulnerable server has a secret token d7a8fbb307d7809469ca9abcb0082 e4f8d5651e46d3cdb762d02d0bf37c9e592, which is sha256 hash of text 'The quick brown fox jumps over the lazy dog'. The hash is 64 characters long, and the keyspace is a-z0-9. We initialize our attacks by choosing search prefix and reference timing point as keyword 'token='. We measure how long it takes to steal this secret token by varying delay.

5.4 Measurement results

In this chapter, we show how the measurement tool performs recognizing cross-origin timings. We will find out how HTML embedder timings are performing against each other depending on the measured context.

5.4.1 Load time measurements

In this section, we evaluate what kind of results load time measurement tool collects. We test how timings are behaving in three measurement scenarios covering different features that are all leaking timings.

5.4.1.1 Results

In the following scenarios we have six different elements and we measure how their box and whisker distributions differ. Boxes are not describing taken time directly. They are describing timing difference of two requests of the same element that targets two different pages in the site we are trying compare. The JavaScript program runs on the browser and measures the time spent between before and after loading a resource. Subtracting the two measured times yields the spent time difference, which is shown as an distribution.

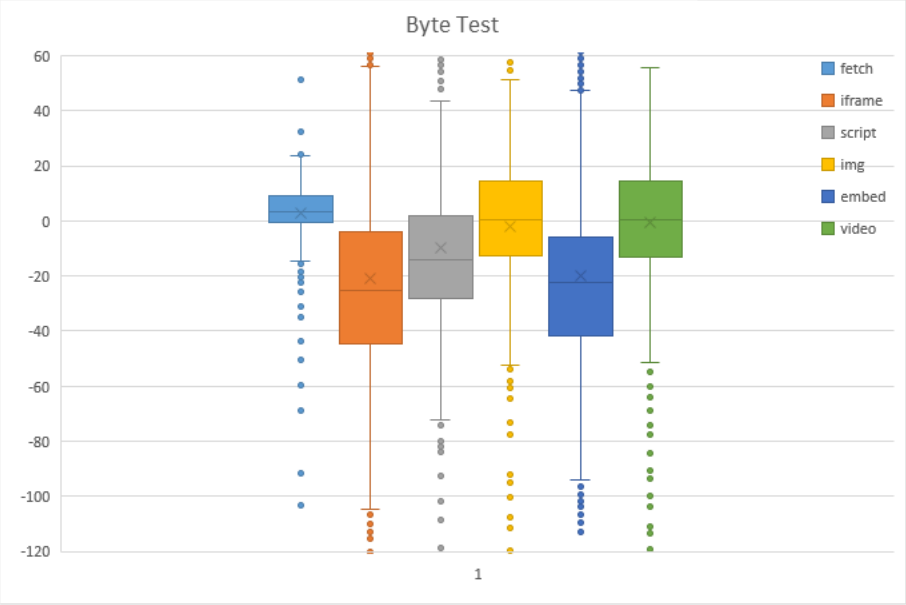

Figure 5.2: Measuring timings of 100 byte payload compared to 30k bytes

In Figure 5.2 we have a measurement scenario altering response size. Figure shows the distribution of loading time difference measured using a JavaScript Web page embedding document with a size of 100 bytes versus 30000 bytes.

At first, the measurements have long error bars and a large number of outliers. Outliers are points that are 1.5*IQR beyond the quartiles. Fetch distribution is packed closely around the median. Median is higher than zero level, which is surprisingly when all other components are very close to zero or lower. The fetch is not that far away, but when taking into a small packed distribution size there could be something strange happening with it also.

Zero median level indicates that in our test run the measured targets had no measurable difference. The further away the median is from zero level, the more likely there is some relational behaviours to object examined. When closer to zero, the measurement is more difficult to be utilized in practical attack and there is a larger chance that the measurement is inside the error margin, especially with a large distributions.

Iframe and embed has their second quartile including rest of quartiles are below the zero. This means that byte test that had a larger amount of bytes took longer to operate. It is a bit unexpected that iframe and embed are leaking timings more than others, because the same amount of traffic is supposed to be transmitted. The reason might be, that they both are successfully rendering those contents to the screen, where the rest script, img and video elements quickly recognize the wrong media content and interrupt transmission and execution. On the other hand entities might include Accept-header that restricts response to specific content on the server side, which makes the server build an empty response. Different entities might also have more aggressive caching mechanisms that drops data transmission time to zero.

In Figure 5.3 we have a measurement scenario altering image resource count. We compare loading a page containing a six images versus a single one. As a result, all boxes are around zero. The most significant observation is that with iframe and embed medians are clearly lower than rest of the elements. All other elements have median closer zero. This is expected, because there is no large amount of difference in bytes. Furthermore, zero medians indicates a good accuracy of the results.

Iframe and embed are the only elements that are parsing context as HTML, and which are reacting to its resource dependencies. They fetch those resources, which produces a couple of HTTP requests. In our measurement scenario, the dependencies are served dynamically, which prevents the use of browser cache.

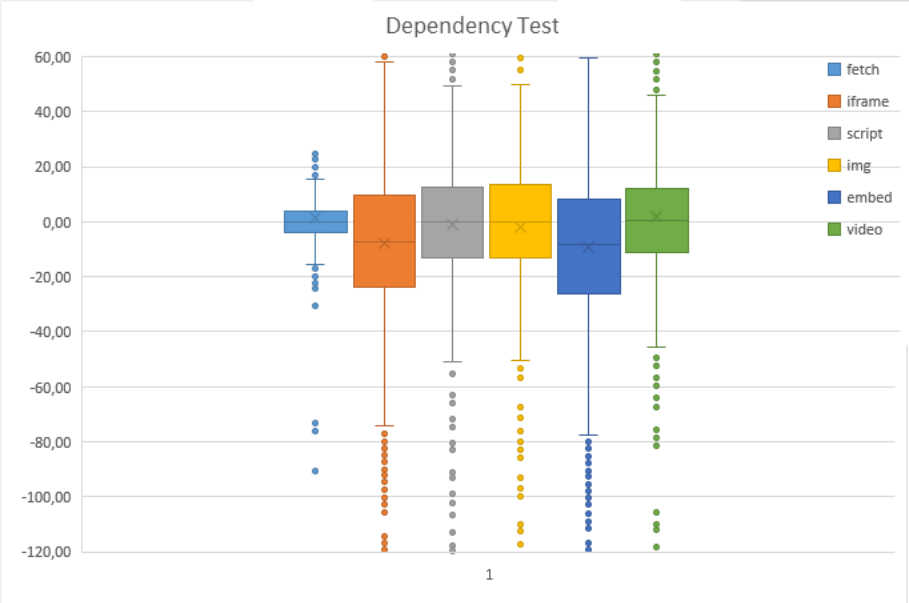

Figure 5.3: Measuring timings of loading six non-cached image resources

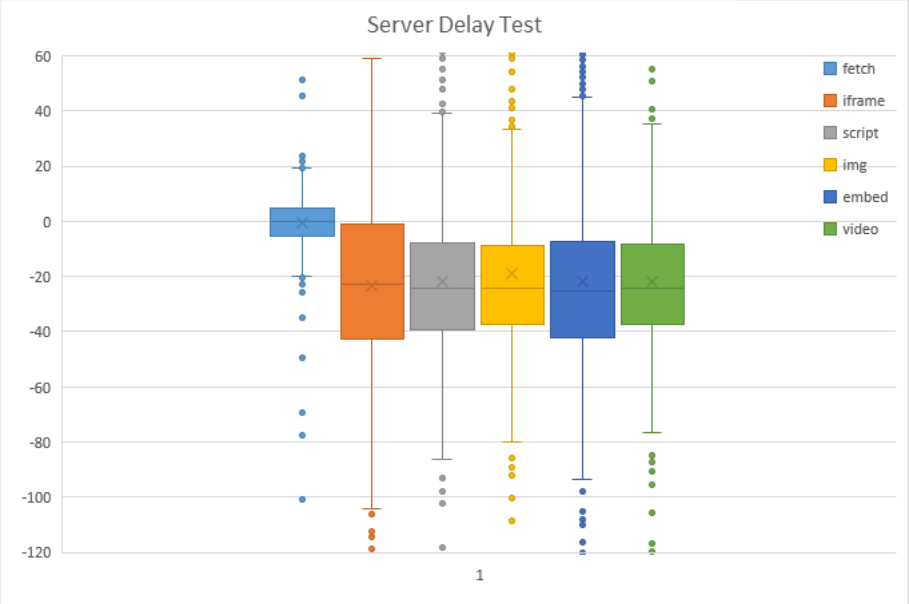

Figure 5.4: Measuring timings of server delaying response 25 milliseconds

In Figure 5.4 measurement scenario we are altering server response delay.

First page targeted has no extra delay, and the second has 25 ms per request. As a result, the fetch measurement has entirely different results than all others. The others are reacting server delay increase with a larger operating time, which is extremely understandable. The medians are a bit less than -20 milliseconds. Precisely, the in-order millisecond medians are -0.17, -22.98, -24,40, -24.51, -25.40 and -24.24.

The fetch measures the process time after the response has arrived until it is processed, which clarifies the situation that there is no related dependences to server time. Furthermore, its median is close to the zero that indicates a good quality of the measurements.

5.4.1.2 Analysis

We measured three different scenarios that affects to the timings. We found out, that iframe and embed leaks timings in all cases. We also found, that those have a bit larger distribution than others. This might be a bit problematic for an attacker, because comparing a larger distribution requires a bit more measurements to make assurance level high enough. Furthermore, script element performed well in the byte and server delay test. For some reason img and video elements failed to measure response content length difference.

Based on the results, when targeting server delay, we suggest using very commonly used script element. Otherwise, embed element seems to have a bit smaller distribution than iframe. Thus, we recommend using embed when targeting resources using altering dependencies and varying content lengths.

We conclude that over stable Internet, size difference of 30 kilobytes payload against 100 bytes is detectable. We found that 25 millisecond processing time difference is detectable and exploitable. We also noted, that a page that renders six dynamic images against one is detectable. We compared the results graphically, thus the numbers are indicative and are intended to show the magnitude of the difference required to be detectable easily.

In this experiment, we managed to reveal distribution differences in a form of box and whisker plot. In real world attack, we would still have to figure out how many measurements is required to achieve sufficient level of assurance to perform more far-reaching decisions.

5.4.2 Cross-site search extractor

In table 5.1 measurements indicate about how long it takes to steal secret token from our cross-origin site using search extractor attack:

5.4.2.1 Results

Delay(ms)	Requests	Backwards	Time(s)
500	2226	3	238.874
400	2137	3	191.79
300	2187	1	154.155
200	1910	0	94.254
100	2091	1	73.396
75	2241	5	74.837
60	2560	11	81.405
50	2396	6	61.307
40	2531	11	70.218
35	2634	12	65.697
30	4575	45	123.258
25	5587	58	132.152
23	19094	255	433.825
22	43989	717	1201.644
20	33684	302	-

Table 5.1: Server delay difference affecting to cross-origin search extractor attack

The graph 5.5 represents same table values in a graph.

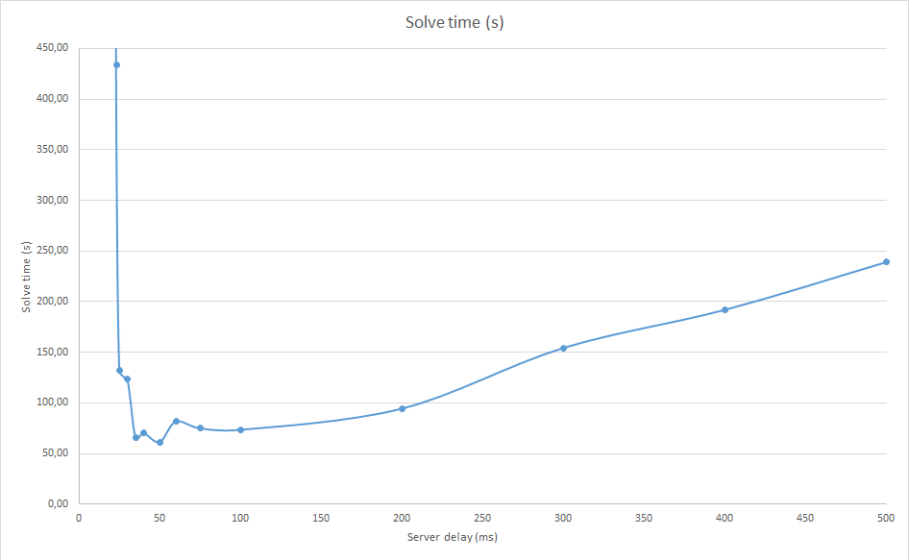

Figure 5.5: Server delay difference affecting to cross-origin search extractor attack

There is certain correlation between delay and solve time despite the fact that the amount of measurement points is low. Performing multiple measurements manually took a lot of time, which reduced the total measurement count as low as 15. Largest data point is left out of the graph to ensure better visibility over the more interesting points.

5.4.2.2 Analysis

The secret token is built using alphabet of characters a-z0-9. In a basic implementation without measurement errors, the required measurements total are expected to be count of $alphabet\_size * token\_length$. Our token is 64 character long, and it has 36 character alphabet. Thus, the required measurements are size of 2304. However, there are six measurements out of 15 below 2300 measurements. It is possible, because our implementation included a optimization where we are skipping rest of character candidates when a promising one is already found. Also, the total amount varies because of measurement errors and backward-steps.

It seems that after 100 ms server delay, the solve time grows linearly based on the delay. The linearity is logical, because a large delay causes longer character candidate checks and thus large solving time even when check measure reliability is reinforced . That is because those time taking measurements have to be performed hundreds or thousands of times. After

the required reliable level have been reached, the spare delay just slow down the attack. Furthermore, larger delay values could still not protect all outlier measurements that are magnitude larger. This can be seen from the introduced table, which has backward steps even though the delay is as large as 500 ms.

The amount of measurements is too low to approximate how the delay function behaves exactly within short delays. However, small enough delay increases solve time. Jitter makes single measurements unstable and the algorithm consumes time by continuous getting lost from the correct path and causes time-consuming backward steps. At some point when decreasing the delay enough, the jitter overwhelms the current implementation and the attack fails.

The attack works reliable in measurement environment when the delay is larger than 40 ms. Solving 64 bytes secret takes roughly one to three minute, which is long time for user to stay on the attacker's page. However, when the token is reduced to eight characters, the attack takes five seconds on 40 ms delay, and 500 ms delay consumes 22 seconds. Furthermore, the attack implementation could be optimized further, for example instead of using serial requests make them parallel. When making multiple connections parallel, you must pay attention to browser global and host specific connection limits.

Chapter 6

Discussion

The user interface of the measurement tool is presented as matrix form allowing adding a new measurement features as part of the others. The implementation was made for long-term usage, and it bends over a new future features that are left out of the scope of this assessment. At current representation, the timing measurement tool separates triggered callbacks, which are valuable information when studying the attack surface [51].

In a demonstrative experiment we introduced three measurement scenarios that are leaking timings. We are not aware how popular those scenarios occurs in practice but it is likely that there are services which do not expect that the cross-origin page could find out the response difference of submitted payloads [52]. Besides, the measurement tool is made exactly for investigating those issues in practical environments.

Prior research analyses covering cross-origin resource sizes and differences have been published before [32]. Goethem, Vanhoef, and Piesses found out that files with a difference in size of less than 15kB or larger than 140kB could not be differentiated when using classical content loading methods [15]. We did not benchmark our measurement tool accuracy but at least in demonstrative experiment we found out that 30kB difference is noticeable. However, we noticed that even tiny 256 bytes difference in resources is catchable when target is having nested dependencies. Furthermore, we found out that server processing time difference larger than 25 ms is measurable.

We utilized box and whisker distributions for showing that the implemented measurement tool works. This experiment also reveals difference between HTML element performing the measurement. Based on performed measurement scenarios we managed to recognize the distribution difference between different HTML elements and succeed recommend specific elements for specific situations.

The performed measurement experiment proves that the measurement

tool performed the measurements successfully and operates within a precision enough to identify some potential timing issues. Graphical box and whisker comparison work well for this demonstrative purpose but for future measurements we should find a more accurate way to compare those distributions.

Introduced Cross-site search extractor method improves previously known *XS-search* attack [34]. We introduce how this *XS-search* attack could steal any previously unknown token. Our attack constructs secret token character by character in contrast to before published method that identify terms from large dictionaries. Our attack makes the *XS-search* attack more serious but in contrast demands more from the targeted environment.

Demonstrative cross-site search extractor attack experiment were performed in Mozilla Firefox browser over stable Internet connection. The tool works surprisingly well and offers stable results as long as the target has observable timing leakage. The attack tool most likely works with different browsers and environments. For example, the preliminary experimenting shows that the attack works even over mobile network using Samsung Galaxy S5 mobile phone and Chrome 52.0.2743.98 browser.

This thesis considers timing attacks as a main attack vector for revealing resource sizes but there are many other side channel leakages [52] that could be utilized when exploiting presented extractor attack method.

Chapter 7

Conclusions

We investigated a method to measure external resource load timings. We then designed and implemented a tool to perform those measurements. The introduced measurement tool in itself is a valuable tool for security assessments, where security of the studied environment is under surveillance.

We experimented how HTML element timings are reacting where amount of bytes, processing time and document dependencies are altered. We found out, that all of those are leaking vulnerable timings and our measurement tool works. In order, to make those measurements more valuable form, we implemented a penetrative attack tool that takes over cross-origin search page timing leaks and exploits them to reveal cross-origin secrets.

In our measurement environment, we simulated mailbox and managed to steal forgot password link from simulated mailbox only by luring mailbox victim to follow a site controlled by an attacker. The attack exploits before published cross-site search attack further and manages to steal custom tokens instead of trying to iterate over a list of known words single word at a time to find out what of those make an actual hit.

The experiment was performed in specific environment and successful attack relays on many factors, which makes the real-word attack less common. Timing attacks are serious vulnerabilities and they should be treated as such. The timing attack touches the implementation as a whole and the timing mistakes are a sum of inner mistakes.

The third and last practical publication made it clear that Firefox implementation has a severe security vulnerability relating to favicon socket handling. The attack is not a timing attack, as an word actual context, however, the attack is still made by manipulating time. We leveraged the issue to a possibility to launch arbitrary page requests over victim's browser even after time has passed and the malicious page is closed. When publishing this thesis, the issue has been already reported to the Mozilla and security patch

has been already published.

Overall, as seen in this thesis, timing security issues have a lot of potential, even though they are more difficult to utilize than simpler security holes. Not enough attention has been paid to this class of vulnerabilities, which can mean worse attacks in the future especially if better automated tools become readily available. We hope that the this thesis raise awareness of timing attacks in web browser world and web application implementers and security testers take the raised issues into account.

7.1 Future work

During this study, it became clear, that there are no public tools available for measuring cross-origin timing issues. The practical impact of those attacks cannot be comprehended without proper security applications that are specialized to discover timing security issues. We would like to see more applications that search for timing issues automatically and exploit vulnerabilities like our search extractor is doing, but with a larger scale in features and a bigger budget. The created tool is a small attempt to find out its potential as a concept. The security industry should study these issues more.

In future measurements, we could measure how browser memory layout is affected by a site that uses another site as a cross-origin dependency. We could measure memory usage by filling available memory with a junk, measure amount of reservable bytes and free the memory. Afterwards, we could include a cross-origin element and measure how the memory space behaves. Also, we could perform measurements to study whether filling up browser connection limit, and/or performed high load affects the timing results. Our load time measurement tool should work well, maybe even better with those additions. Furthermore, the measurement accuracy could be improved by using Crosby's low-percentile filters when comparing timing results. They were left out of this implementation to avoid broadening the scope of this thesis too much.

Favicon timeout leakage reveals that socket timeouts are not very carefully tested. One reason could be that their testing environment does not support those tests easily. Thus, we suppose there are a lot of potential that the same mistakes are done later in other browser components. In general, the issues in browser socket handling timing outs should be inspected more specifically.

One simple way to reveal those weaknesses is to use an aggressive network bandwidth limiter and investigate how the TCP streams are reacting to a situation where a single browser window is closed. The data flows very slowly

and we can then detect if there is an mechanism to issue a disconnection when browser window is closed, otherwise the connection stays open and browser component might still be ready to operate, even long time after the user has left from the site. This may lead to a similar attacks that we demonstrated in our socket timeout vulnerability.

Bibliography

[1] OWASP Foundation. Top 10 2013-Introduction. webpage, 2013. https://www.owasp.org/index.php/Top_10_2013-Introduction. Accessed 10.10.2016.

[2] Martin Woschek. OWASP Cheat Sheets, 2015. https://www.owasp.org/images/9/9a/OWASP_Cheatsheets_Book.pdf. Accessed 10.10.2016.

[3] Tali Garsiel, Paul Irish. How Browsers Work: Behind the scenes of modern web browsers. webpage, 2011. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/. Accessed 10.10.2016.

[4] W3C Members. W3C standards. webpage. https://www.w3.org/standards/. Accessed 10.10.2016.

[5] Charlie Hothersall-Thomas. BrowserAudit - A web application that tests the security of browser implementations, 2014. http://www.doc.ic.ac.uk/teaching/distinguished-projects/2014/c.hothersall-thomas.pdf. Accessed 10.10.2016.

[6] Awio Web Services LLC. Web Browser Usage Trends. webpage, 2016. https://www.w3counter.com/trends. Accessed 10.10.2016.

[7] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[8] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Conditional requests. RFC 7232, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7232.txt.

[9] Mozilla Developer Network and individual contributors. Same-origin policy. web, 2016. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed 10.10.2016.

[10] A. Barth. Http state management mechanism. RFC 6265, RFC Editor, April 2011. http://www.rfc-editor.org/rfc/rfc6265.txt.

[11] OWASP Foundation. Cross-Site Request Forgery (CSRF)). web, 2016. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF). Accessed 10.10.2016.

[12] codahale. A Lesson In Timing Attacks (or, Don't use MessageDigest.isEquals). web, 2009. https://codahale.com/a-lesson-in-timing-attacks/. Accessed 10.10.2016.

[13] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1055–1062, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516712. URL http://doi.acm.org/10.1145/2508859.2516712.

[14] Yoshitaka Nagami, Daisuke Miyamoto, Hiroaki Hazeyama, and Youki Kadobayashi. An independent evaluation of web timing attack and its countermeasure. In *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, ARES '08, pages 1319–1324, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3102-1. doi: 10.1109/ARES.2008.111. URL http://dx.doi.org/10.1109/ARES.2008.111.

[15] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1382–1393, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813632. URL http://doi.acm.org/10.1145/2810103.2813632.

[16] Sebastian Schinzel. Time is on my Side: Exploiting Timing Side Channel Vulnerabilities on the Web). slides, 2011. https://events.ccc.de/congress/2011/Fahrplan/attachments/2021_Slides. Accessed 10.10.2016.

[17] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251353.1251354.

[18] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur.*, 12 (3):17:1–17:29, January 2009. ISSN 1094-9224. doi: 10.1145/1455526.1455530. URL `http://doi.acm.org/10.1145/1455526.1455530`.

[19] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 621–628, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242656. URL `http://doi.acm.org/10.1145/1242572.1242656`.

[20] Sebastian Schinzel. Tools for timing attacks: mona-timing-lib. open source program, 2015. `https://github.com/seecurity/mona-timing-lib`. Accessed 10.10.2016.

[21] Sebastian Schinzel. A tool for analyzing timing measurements. open source program, 2014. `https://github.com/seecurity/mona-timing-report`. Accessed 10.10.2016.

[22] W3C. Navigation Timing Level 2. web, 2016. `https://w3c.github.io/navigation-timing/#performancenavigationtiming`. Accessed 10.10.2016.

[23] Zhenkai Liang Yaoqi Jia, Xinshu Dong and Prateek Saxena. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing*, 19(1):44–53, 2015. ISSN 1089-7801. doi: doi.ieeecomputersociety.org/10.1109/MIC.2014.103.

[24] S. Schinzel. An efficient mitigation method for timing side channels on the web. In *Proceedings of the 2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2011)*, 2011.

[25] Sebastian Schinzel. Time is NOT on Your Side: Mitigating Timing Side Channels on the Web. web, 2016. `https://events.ccc.de/congress/2012/Fahrplan/attachments/2235_29c3-schinzel.pdf`. Accessed 10.10.2016.

[26] unknown. Sniffing browser history using HSTS + CSP. application, 2015. `https://github.com/diracdeltas/sniffly`. Accessed 10.10.2016.

[27] Mozilla Developer Network and individual contributors. Window getComputedStyle API. web, 2016. `https://developer.mozilla.org/en-US/docs/Web/API/Window/getComputedStyle`. Accessed 10.10.2016.

[28] Sai. CSS Fingerprint: preliminary data. web, 2010. `http://saizai.livejournal.com/960791.html`. Accessed 10.10.2016.

[29] Gilbert Wondracek, Thorsten Holz, Engin Kirda, and Christopher Kruegel. A practical attack to de-anonymize social network users. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 223–238, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.21. URL `http://dx.doi.org/10.1109/SP.2010.21`.

[30] Mozilla Developer Network and individual contributors. Window requestAnimationFrame API. web, 2016. `https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`. Accessed 10.10.2016.

[31] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing*, 19(1):44–53, 2015.

[32] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. 2016.

[33] Adrian Hayes. A tool for performing network timing attacks on plaintext and hashed password authentication. web, 2013. `https://github.com/aj-code/TimingIntrusionTool5000`. Accessed 10.10.2016.

[34] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1394–1405, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813688. URL `http://doi.acm.org/10.1145/2810103.2813688`.

[35] W3C. A vocabulary and associated APIs for HTML and XHTML. web, 2014. `https://www.w3.org/TR/html5/embedded-content-0.html`. Accessed 10.10.2016.

[36] W3C. Progress Events. web, 2014. `https://www.w3.org/TR/progress-events/`. Accessed 10.10.2016.

[37] Arvind Jain, Todd Reifsteck, W3C. Resource Timing Level 1. web, 2016. `https://www.w3.org/TR/resource-timing/`. Accessed 10.10.2016.

[38] W3C. A vocabulary and associated APIs for HTML and XHTML. web, 2014. `https://www.w3.org/TR/html5/webappapis.html#event-loop`. Accessed 10.10.2016.

[39] w3schools. HTML DOM Events. web. http://www.w3schools.com/jsref/dom_obj_event.asp. Accessed 10.10.2016.

[40] Quirksmode. Error events. web, 2016. http://www.quirksmode.org/dom/events/error.html. Accessed 10.10.2016.

[41] Mozilla. Bug 1255270 - (CVE-2016-2830) Favicon request doesn't timeout, or close when related window is closed. web, 2016. https://bugzilla.mozilla.org/show_bug.cgi?id=1255270. Accessed 10.10.2016.

[42] Mozilla. Bug 1255267 - delayed CSRF attack from the closed page. web, 2016. https://bugzilla.mozilla.org/show_bug.cgi?id=1255267. Accessed 10.10.2016.

[43] Mozilla Developer Network and individual contributors. Mozilla2:ImageLib. web, 2009. https://wiki.mozilla.org/Mozilla2:ImageLib. Accessed 10.10.2016.

[44] Mozilla. Bug 1279208 - Favicon request doesn't timeout, or close when related window is closed (1255270 is not fixed yet). web, 2016. https://bugzilla.mozilla.org/show_bug.cgi?id=1279208. Accessed 10.10.2016.

[45] Mozilla. Bug 1283067 - Favicon request doesn't timeout or close when related window is closed (1255270 is not fixed yet) on Windows due to WindowsPreviewPreTab.jsm. web, 2016. https://bugzilla.mozilla.org/show_bug.cgi?id=1283067. Accessed 10.10.2016.

[46] Mozilla. Bug 1285196 - WindowsPreviewPerTab should not do anything when the feature is disabled. web, 2016. https://bugzilla.mozilla.org/show_bug.cgi?id=1285196. Accessed 10.10.2016.

[47] Mozilla. Favicon network connection can persist when page is closed. web, 2016. https://www.mozilla.org/en-US/security/advisories/mfsa2016-63/. Accessed 10.10.2016.

[48] Mozilla. CVE-2016-2830. web, 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2830. Accessed 10.10.2016.

[49] Mozilla. Client Bug Bounty Program. web, 2016. https://www.mozilla.org/en-US/security/client-bug-bounty/. Accessed 10.10.2016.

[50] CSC. Funet Network Services. web. https://www.csc.fi/funet-network-services. Accessed 10.10.2016.

[51] Jeremiah Grossman. Login Detection, whose problem is it? web, 2008. `http://blog.jeremiahgrossman.com/2008/03/login-detection-whose-problem-is-it.html`. Accessed 10.10.2016.

[52] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and conquer: Exposing cross-origin resource size. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goethem`.

Appendix A

Measurement samples

The documents listed here were used for timing resource loads.

Listing A.1: byte_sample1

```
aaaaaaaaaaaaaaaaaaaaa...[100 characters]
```

Listing A.2: byte_sample2

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...[30000 characters]
```

Listing A.3: delay_sample1

```
<?php usleep(1); ?> usleep(1);
```

Listing A.4: delay_sample2

```
<?php usleep(25000); ?> usleep(25000);
```

Listing A.5: embed_sample1

```
<!DOCTYPE html>
<html>
<body>
<img src="image1.php" alt="View1">
</body>
</html>
```

Listing A.6: embed_sample2

```
<!DOCTYPE html>
<html>
```

```
<body>
<img src="image1.php" alt="View1">
<img src="image2.php" alt="View2">
<img src="image3.php" alt="View3">
<img src="image4.php" alt="View4">
<img src="image5.php" alt="View5">
<img src="image6.php" alt="View6">
</body>
</html>
```

Where all of those included images have a photo of sailing boat with 259x194 resolution within a size of 4.90 kB.