Trusting the Big Friendly Giants: large-scale evaluation of dependencies on Finnish websites

Master's thesis
University of Turku
Department of Future Technologies
Computer Science
2018
Joonas Salovaara

Supervisors:
Ville Leppänen
Jukka Ruohonen

UNIVERSITY OF TURKU
Department of Information Technology

JOONAS SALOVAARA: Trusting the Big Friendly Giants: large-scale evaluation of dependencies on Finnish websites

Master's thesis, 59 p., 3 app. p.
Computer Science
March 2018

Software development companies compete with each other in cost effectiveness, quality and speed of delivery like any other businesses operating on the free market. To keep up with the competition companies reuse code and implement common features with third-party tools and libraries.

Using third-party code can can help in staying ahead of competition but it can also increase the attack surface of your application, cause loss of privacy and control and increase the likelihood of information leaks.

In this thesis we define a new term (cross-domain) that is better suited for dependency control analysis and develop a dependency checker tool that can find dependencies and the entities behind them on web pages. We also perform and empirical study where we use this tool for a corpus of ~370,000 Finnish websites and analyze the results.

In the study we find that about half of the dependencies on Finnish websites are cross-domain and that almost 73% of the dependencies are controlled by entities registered to United States. We also find that the cross-domain dependency landscape in Finland is dominated by the "Big Friendly Giants" Google and Facebook and that this has a negative impact on privacy and security of Finnish websites.

In the end of the thesis we present possible countermeasures that can alleviate the risks caused by third-party dependencies and note that these dependencies should be better understood, monitored and their powers limited.

Keywords: web page, website, dependency, cross-origin, cross-domain, third-party

Acknowledgements

Thanks to my supervisors Ville and Jukka and all of my friends and family for conversations that have helped me with this work so much.

> “The matter with human beans,” the BFG went on, ”is that they is absolutely refusing to believe in anything unless they is actually seeing it right in front of their own schnozzles.”
>
> (Roald Dahl — The BFG)

Contents

Chapter 1

Introduction

In development programmers have always reused code whether it was language specific libraries or community shared functions and tools. Netscape's Navigator was one of the first web browsers. In January 1998 Netscape decided to release the source code of Navigator that created Mozilla and kicked into movement the open-source software movement as we know it. These days it is estimated that 51% of the world's population has internet access. With World Wide Web's growth browsers have gotten immensely more capable and complex. This huge explosion of a new medium has created a vast web software industry and with it the use of dependencies in web related code has changed.

In today's world software companies compete with each other in cost efficiency, quality and speed. All of these requirements together with a booming open source community has lead to a lot of web code reuse across the stack. From the browsers perspective we as a community have built large amounts of CSS and JavaScript frameworks and libraries that can be used directly in the browser. Introducing these kind of third-party dependencies might propagate trust and increase the attack vector of your web related software.

In this thesis we are focusing on three separate research questions (RQs). The questions are as follows:

RQ_1: *How to technically measure cross-domain dependencies on web pages?*

RQ_2: *How does the dependency landscape on Finnish web pages look like on a large scale and on why does it look like it does?*

RQ_3: *What kind of threats the current situation has created?*

When talking about dependencies and their security implications the focus of this thesis is on JavaScript due to its wide client side capabilities in modern web browsers though most of the conclusions and notes on some level apply for all dependency types.

In the second chapter we talk about trust relationships that should be considered when introducing third-party dependencies and go through a few imaginary risk scenarios that illustrate the problems and security implications that third-party dependencies can introduce.

It the third chapter we talk about different dependency types on web pages and propose a way for categorizing them. We also inspect and define the terms same-origin and same-domain and explain why these both terms are required in the context of dependencies.

In the beginning of the fourth chapter we plan and build an open source tool that can be used to run a dependency check for any web page. We talk about technology selections and go deeper into what is technically required to measure dependencies on web pages in general. We also scrape ~370,000 Finnish domain names and run the dependency checker tool against them and analyze the results.

In chapter five we go through a non-exhaustive list of possible countermeasures for security implications third-party cross-domain dependencies can cause.

Chapter six makes conclusions of the whole thesis. We will revise the findings on the theory and case study and what implications they have. At the end of the chapter are thoughts on further studies on this matter.

Chapter 2

Risks of Third-Party Dependencies

In the fast phased world of web, software companies compete with each other in cost effectiveness, quality and speed of delivery like almost any other businesses that operate on the free market. To keep up with the competition companies reuse code and integrate/implement common features to their applications with third-party tools and libraries. Reuse of third-party code can have a lot of benefits but it also creates risks. In this chapter we discuss about the possible risk categories and scenarios. This forms a base for our **RQ3** and directs the development and design of the dependency checker tool discussed in more detail in chapter 4.

Most code reuse creates a some sort of dependency. Installing third party utilities can speed up development significantly but it also easily leads to a lot of third-party dependencies of which you are not directly in control of. Most developers do not have the resources to read through all source code of the dependencies they are forced to use to keep up with business requirements and project deadlines. We would argue that in these cases part of the control of how the application works moves under the control of the controller of the dependency. The effect of the lost control materializes if and only if the dependency is even partly used as a ”black box” which usually end up being a requirement for speeding up development and thus makes the case quite common.

Reusing open source code has a lot of benefits and as the open source movement has

gained traction over the years the quality of open source tools and libraries has improved. Also the amount of freely usable open source code has exploded and thus using someone else's code is easier than ever. Code reuse can also have negative effects like loosing control or trust to third-parties that in most cases are not as well know as developers or companies might want to think they are.

In this thesis we focus on linked ("live") dependencies (see section 3.2) on web pages putting weight specifically on JavaScript dependencies due to their dynamic possibilities. Prime examples of these kind of dependencies are for example client-side scripting libraries like jQuery or user tracking libraries like Google's analytics.js when downloaded from third-party controlled content delivery networks (CDN).

2.1 Organizational trust and control

Like mentioned above using third-party dependencies almost always moves some of the responsibility and control of the intended end result (web page or web site or web application) to the owners of the dependency. With packaged dependencies there is a possibility to inspect the dependencies and make an informed decision about the dependency, what it does and can do within the application and what the risks that come from it are when it is added. With linked dependencies that you do not have a direct control over, this is not the case as the same process does not guarantee the same level of safety. Linked dependencies (like packaged ones) can be inspected when the dependency is added but knowing when the contents of the dependency change after adding it is not trivial and it can not be checked in the same process where packaged dependencies are inspected (if such step even exists in the current process) as you have no direct control over the third-party dependency and when its contents change. Nothing guarantees that the contents of the linked third-party dependencies change only when you want them to change or that they are what they seem to be when you inspect them.

When adding a third-party linked dependency you are trusting the organization in control of that dependency. When it comes to allowing a third-party to load and execute a linked JavaScript file from their servers you are giving them a possibility to take control over the whole user experience of the pages the dependency is downloaded and executed on – you just trust them not to do it. This is possible due to the native APIs that any script run on a page can take advantage of (like DOM manipulation) and the fact that there is really no standard way to limit for example what element of a page certain scripts can manipulate or what data the script can collect of the page and send back home. Third-party scripts are pretty much first-class citizens like you own scripts.

Are the organizations that companies trust their digital businesses to worth the trust and how is that trust formed? How much do organizations and developers use linked third-party dependencies and how well the risks are know that adding these kind of dependencies exposes them to? What can be done to the current situation?

On a more philosophical level if we think about trust from human perspective it can be defined as follows: ”Trust is the personal believe in correctness of something. It is the deep conviction of truth and rightness and can not be enforced. If you gain someones trust you have established an interpersonal relationship, based on communication, shared values and experiences. Trust always depends on mutuality.” [1].

In the context of technology trust can form on the human level for example between organizations and individuals but it can also be formed on a technological level. The biggest difference between these two levels being that within the technological level trust can be usually automatically verified and limits of agreements enforced. In most cases this kind of automation does not exist for third-party linked dependencies.

2.2 Risk scenarios

Propagating trust for example via third-party linked dependencies exposes the organization and the products the dependencies are in for different types of risks. We will go over a few different types of risk categories and preset possible example risk scenarios.

2.2.1 Acquisitions

Scenario 1

Lets imagine a situation where company A agrees with company B to use company B's analytics script on its website. The trust is formed between individual actors inside the companies A and B. An agreement is formed between the companies based on the trust of the actors on both sides. Now the companies have formed a mutual agreement and can be considered to trust each other.

Now in these companies employees come and go and the individual actors between whom the trust initially was formed are gone but the agreement is still in place and A still uses B's analytics script without any problems. Behind the scenes company C makes an offer to buy company B's analytics product and the owners of B decide to sell. C has ensured that nothing in the B's analytics product breaks for the existing users. Though company B releases a news post about selling its analytics product the information about the acquisition never reaches the company A.

At this point company A has an agreement with a company C who does not employ the people who personally were part of forming the agreement with them and the whole company is not the same anymore. As A has tens of these kind of deals it has a very low possibility of noticing these kind of changes in the trust relationship without an internal process that periodically checks all its agreements for these kind of issues.

The company C who acquired company B's analytics product is malicious and changes how the analytic script it now owns (and that is already used on multiple high traffic sites)

behaves. It keeps the normal business running as is to look as nothing would have changed but addition to that it starts mining cryptocurrencies with A's users using the analytics script they have full control over.

The user experience of the users of A's website gets worse but in such a way that A's users end up blaming their own devices for the light slowdown. This kind of user feedback never reaches company A in a way that would trigger an inspection. To keep even more under cover the company C alters their analytics script so that when the company A's employees use their own website a version of the script is loaded that does not have the cryptocurrency mining functionality. The company C is able to do this as part of their agreement the company A reports all their internal IP addresses to C so that internal traffic can be excluded from the user analytics.

Finally after half a year of exploitation company A finds out what is going on after launching an investigation related to the soaring usage of their website. A immediately removes C's analytics script from its website and takes legal action against B and C but the damage has already been done.

Conclusion

As we can see from the scenario 1 above company and product acquisitions can pose a significant risk if the trust relationships are not periodically followed and the agreements between the companies enforced and tested to hold as has been agreed upon. These kind of attacks are called supply chain attacks [2] and they take advantage of common trust relationships in software supply chains. One good example of this kind of an attack was when an unnamed company bought a popular WordPress captcha plugin from another company and embedded a backdoor to the plugin. This affected more than 300,000 websites [3].

2.2.2 Information leaks

Scenario 2

Company A is a newly formed bank that is working on its online banking platform. Company B is an analytics company like in the scenario 1 (see section 2.2.1) but not malicious.

Company A wants to know how its customers are using its public website and how they move from there to the online banking platform. They also want to know how their customers use the online banking platform to guide development to the right direction in the future.

To achieve this the developers in the company A are given a feature request from management. Their public website already uses B's analytics script. To measure conversions and see how users move between these two sites the same analytics script is installed to the online banking platform. Analytics data starts flowing from the online banking platform to the systems of B and management in company A is happy to be able to look at the new data in their dashboards.

In a few weeks A's social media team alerts their own developer team about a Twitter conversation where a technologically minded customer is asking why his bank account data is sent to the analytics company (B) when he uses A's online banking platform. A's social media team tells transparently that they collect usage data to be able to improve the product in the future but does not understand the underlying issue.

The Twitter discussion becomes a national social media storm and the development team is pulled in to investigate the issue. The developers find that the analytics script is addition to tracking what pages the user visits sending a huge amount of information back to the analytics system including partial page content. A's development team realizes that the analytics script has collected peoples private banking information about accounts, balances and transactions of hundreds of thousands of their customers. The management is notified and the B's analytics script is removed from the online banking platform.

After the incident is uncovered and the issue fixed A starts an internal investiga-

tion about the issue. In the investigation they find that the marketing department has shared read access to the data collected in B's systems for three advertising and marketing companies they work with. The read access is immediately removed, the analytics data cleaned and the partner companies asked to delete all data but in reality A has no possibility to know where the data might have already leaked.

Conclusion

As we can see from the scenario 2 when the inner workings of third-party dependencies are not fully understood by the users (in this case company A) they can pose a risk for unintended information leaks that can easily leave the business in a vulnerable state.

In the scenario above we can also see how the data not only leaked to B's systems where it should have never been sent in the first place but it leaked forward to subcontractors. This was not intentional but it is easy to see that this kind of case is quite realistic. Company A had never wanted to trust their customers private banking data to marketing related subcontractors but did want to trust them with the user flow and action data to improve marketing impact. The third-party dependency and not understanding its power ended up forcibly widening the trust between A and its marketing related subcontractors and thus loosing control of the data.

The above scenario is totally fictional but a relatively similar case happened in Finland with S-Pankki that was using Google Analytics on their online banking platform [4].

On a more light scale loading a linked dependency from a third-party server always gives some data to that server in the request like for example cookies set for that domain. This can and should also be seen as telling something of your users to a third party.

2.2.3 Hacking

Scenario 3

Company A is a media company that runs a big online news website. Company B provides CDN services as part of its service portfolio. To deliver content to its users faster A uses B's content delivery network (CDN) for all its static resources on its news website.

In company A a secretary at the financial department needs monthly receipts for book keeping of the payments charged by B from A from using its CDN. A developer at the IT department of company A who has access to the CDN web interface where the receipts can be downloaded is frustrated by this mundane task and wants the secretary to be able to download the receipts directly by himself. As he can not add any new user accounts to the CDN web interface as it would rise the monthly fee that is partially based on the amount of user accounts he emails his credentials to the secretary and tells him not to change the password and keep the account log in information only to himself.

On a weekend trip to Prague the secretary's phone gets stolen. The thieves are able to access the secretary's company email via the phone and they get hold of the CDN web interface credentials.

The secretary tells about the stolen phone to the company when he gets back to work on Monday. The standard procedure takes place. All the secretary's account passwords are reset and the phone automatically remotely wiped if it ever is seen by the remote mobile device management system of the company. The CDN account password is not reset as it does not belong to the secretary and the IT department employees taking care of the resets are not aware of the shared account credentials. The developer at the IT department who gave his credentials to the secretary is not aware that he has ever lost his phone.

The CDN credentials of the well known media company are sold by the thieves on black market and bought for couple of thousand dollars by a hacker group. The hacker group uses the credentials to log in to the CDN control panel and infects all JavaScript

files they can find with a cryptocurrency miner script.

The script runs successfully and unnoticed for couple of weeks until the breach is noticed when updated to the site are being deployed. All account passwords related to the CDN are reset and an investigation about the incident started but the damage has already been done.

Conclusion

Neither of the first two scenarios above had anything to do with hacking. Both of the issues were related to organizations, trust and not fully understanding the tools in use. Hacking by definition entails intruding to systems, circumventing protections and finding edge cases to trigger systems to behave in unintended ways. This means that there is an unimaginable number of ways to hack something in a way that makes it possible to compromise a file that acts as a third party dependency for someone else.

The point in mentioning hacking here is that it is one of the risk scenarios you are exposed to when using third-party dependencies. Instead of worrying just about your own servers getting hacked with third-party dependencies you will have to also worry about the third-party servers getting hacked thus increasing the attack surface of your website.

In case of a large news site using a CDN is close to a must and commercially available options are probably safer and more robust that building your own. We would argue though that in smaller cases like linking jQuery to your site from a public CDN for a possibility of a small speed gain is a case where the advantages do not overcome the disadvantages.

Chapter 3

Dependencies on web pages

3.1 Introduction

Web pages are what make up the world wide web. They're usually written in Hypertext Markup Language (HTML) [5] and consumed with web browsers [6].

Web pages consist of HTML, Cascading Style Sheets (CSS) and JavaScript. HTML is used to create a semantic structure for inline and linked content, CSS is used to style this content and JavaScript is used to provide enhanced interactivity and user experience. Web pages can house all kinds of formats of digital content the most common and widely supported formats being text, image, video and audio.

Web pages are usually accessed with web browsers via internet. Most of these pages are hosted on servers which simply put are just internet connected computers running a program that allows access to predefined HTML documents on that computer.

The HTML documents accessed with a web browser and served to you by the server might be static or dynamically created. Dynamic in this case means that the HTML document was written by a program instead of for example a human being.

When web browsers ask servers for resources they use a protocol called Hypertext Transfer Protocol (HTTP) [7] for communication. It is specifically designed for this use and forms the base for data communication for the World Wide Web.

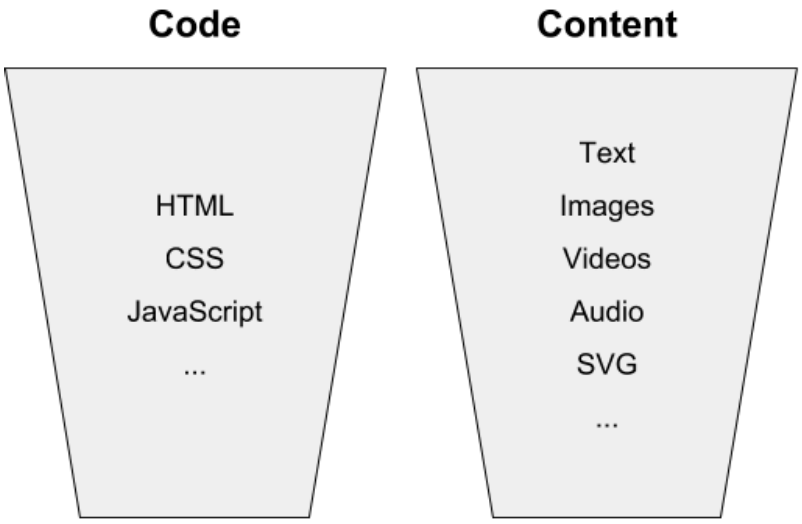

Figure 3.1: Code and content division

Understanding the basics of ”how the web works” is a prerequisite for answering our **RQ1**, **RQ2** and researching dependency landscape on the web in general.

3.2 Dependency types

Dependencies on web pages can be looked from many angles and thus what is seen as a dependency depends on the point of view. Lets go through a few examples of what can be seen as dependencies.

Lets divide a web page to two different ”buckets” called code and content. The code bucket has all markup, styles and JavaScript – everything that technically makes a web page. The content bucket has all actual content of the page like text, images, videos and audio but no code. This kind of division between code and content allow us to look at dependencies from two different perspectives (see 3.1).

Dependencies from a code point of view are more complex than from a content point of view. Lets say that a script on the page relies on a native feature of a web browser. Depending on what the script wants to achieve the native feature can be seen as a dependency that is missing if the web browser doesn’t support it. This is a very common case

as native APIs supported by different versions of different web browsers vary greatly [8]. This leads to a conclusion that native browser APIs and features can be seen as third-party dependencies. The version of the platform and available features are something we build on but in the end we have no control over. This is something that needs to be taken into account when building and testing web pages.

When it comes to code and especially modern JavaScript packaged dependencies are extremely common and are known to cause horrifying problems like the infamous left-pad incident [9] [10]. All packaged dependencies are dependencies but they're only dependencies from the point of view of the build process at development time. From the point of view of the web browser packaged dependencies are something it can not do anything about if it encounters an error at runtime. Outdated packaged dependencies can cause serious security problems but there is little the browser can do to alleviate this. The outdated packaged dependencies need to be updated by the developer and a new version of the web page needs to be deployed to fix problems related to packaged dependencies.

Dependencies the web browser has the possibility to do something about are the linked ("live") unpackaged dependencies of a web page. The HTML document might include multiple links to external scripts and/or style sheets across the document. In some cases the order of these dependencies in the document is relevant for correct execution. Failed loading of even one linked dependency might render the web page in a state that does not reflect the intended end result from the perspective of the developer of the page. Thus all external scripts, styles, fonts etc. meant to be loaded by the browser can be seen as dependencies if they are required to achieve the intended end result.

A web page can download resources that the rendering and functionality of the current page does not need. This kind of pre-emptive loading of resources can be related to for example the web page preparing for a user triggered navigation to another page on the same website. In this case pre-emptive loading of resources can significantly shorten then time it takes for the other page to be shown to the user if the user decides to navigate to it. We

would argue that these kind of ”soft” dependencies can also be seen as dependencies from the perspective of the developer. Implementing this kind pre-emptive resource loading is an intentional choice and thus falls under the umbrella term ”intended end result”.

From the point of view of an intended end result for the rendering of a web page content also plays a big role. If all content fails to load there is not much on the site for humans to consume. Its directly consumable informational and functional value is close to zero. Though linked content is not usually seen as a dependency I would argue the opposite. It may not be a direct dependency related to code but it is still crucial for achieving the intended end result.

This leads us to note that in terms of this thesis the most beneficial way to look at dependencies is to think them as something the browser loads whether it is a script or an image or something in between. The natural inspection point is thus the network traffic that happens in the web browser when a web page is loaded.

When inspecting HTTP requests made by the browser when loading a web page the requested resources can be categorized by type. At the time of writing there seems to be no written standard or specification for how browsers should categorize requests though most of them do provide an API for inspecting how they do it internally.

The need for providing this kind of classification that is accessible via a public APIs has risen from the development of web browser developer tools over the past years. Based on an acknowledgement on an article at Mozilla Developer Network (MDN) the current documented version of the `webRequest` JavaScript API that allows developers to access the internal categorization types of resources is based on Chromium's `webRequest` API [11].

MDN currently lists 20 different resource types[1]. The browser compatibility table in the article [11] lists only 12 of them[2] and the `web_request.json` file [12] the article

[1]Them being: beacon, csp_report, font, image, imageset, main_frame, media, object, object_subrequest, ping, script, stylesheet, sub_frame, web_manifest, websocket, xbl, xml_dtd, xmlhttprequest, xslt and other

[2]Missing ones being: image, main_frame, object, script, stylesheet, sub_frame, xmlhttprequest and other

says the documentation is derived from lists only 13 resource types. The article lists 7 more resource types than the file it tells it is base on but doesn't really tell where they come from. Digging around Chromium's source you can find an internal enumerated type (enum) definition [13] that lists 19.

As we can see from the inconsistencies and lack of proper specification above the web is a place where specifications live their own life and so do browser vendors implementations of the specifications. It is a bit of a game of who happens to implement it first. Thus any of the lists mentioned above can not be considered as exhaustive, static or standard but an image of the current state at the time of writing. What comes to the backwards compatible nature of web it is more probable that more things get added than that something gets dropped in the future.

In this thesis we are most interested in the "more classical" resource types like `script`, `stylesheet` and `image` and specifically of the origins these types of resources are downloaded from.

3.3 Origins

Origin is a term defined in the RFC 6454 [14] of IETF. Roughly speaking origin consists of scheme, host, and port. The definition of origin forms the base for categorizing resources downloaded by browsers. As a term it is mainly security related and makes deriving terms like same-origin and cross-origin possible.

3.3.1 Same-origin and cross-origin

As origin consists of scheme, host and port for having two different URIs belong to the same origin their scheme, host and port must match. Let's look at a few simple examples that can be found from the origin specification.

All of the following URIs have the same origin:

```
1 http://example.com/
2 http://example.com:80/
3 http://example.com/path/file
```

Each of the following URIs have different origins:

```
1 http://example.com/
2 http://example.com:8080/
3 http://www.example.com/
4 https://example.com:80/
5 https://example.com/
6 http://example.org/
7 http://ietf.org/
```

Cross-origin is a counter term for same-origin and means that the origin is different from hypothetical origin we are comparing the current origin to. So for two imaginary URIs either one or more of the three required components that make an origin do not match. This means that if either one or more of scheme, host or port differ between two URIs they are considered cross-origin.

From the cross-origin example URIs above we can see for example that different subdomains of the domain are considered cross-domain though the domain matches.

3.3.2 Same-domain and cross-domain

The definition of the terms same-domain and cross-domain are not standard and are used quite freely and even wrongly in many cases as the concept of domain is not properly understood. This thesis defines and explains the terms same-domain and cross-domain.

In the origin specification the host part of an origin can be seen as a fully qualified domain name (FQDN) without the full stop (period) character at the end. Thus they are almost interchangeable. A FQDN consists of a top level domain (TLD), domain and possible sub-domains [15]. This means that host defined in the origin specification can be further split to these three different components. Lets look at a FQDN example.

```
1 www.example.com
```

In the above example the ”com” part is the TLD. The ”example” part is the domain and the ”www” part is the sub-domain. Understanding this allows us to define the terms same-domain and cross-domain relatively strictly.

With the term same-domain this thesis refers to a much relaxed interpretation of same-origin where two origins are considered same-domain if and only if their TLD and domain match. The sub-domain part of the FQDN is not considered and neither are the protocol or the port in the context of the origin.

The following URI examples are all considered as same-domain when compared to each other.

```
1 http://example.com/
2 http://example.com:8080/
3 http://www.example.com/
4 https://example.com:80/
5 https://example.com/
```

As with cross-origin the term cross-domain is a counter term for same-domain. In this thesis cross-domain refers to a much relaxed interpretation of cross-origin where two origins are considered cross-domain if and only if their TLD and/or domain differ. The sub-domain part of the FQDN is not considered and neither are the protocol or the port in the context of the origin. The following URI examples are all considered as cross-domain when compared to each other.

```
1 http://example.com/
2 http://example.org/
3 http://ietf.org/
```

Simply put if the TLD and domain parts of the FQDN part of an origin match with another origin the origins are considered same-domain. Otherwise they are considered cross-domain.

Need for same-domain and cross-domain terms

The need for the terms same-domain and its counter pole cross-domain arises from investigating organizational trust relationships on web pages. In many cases a lot of resources a web page downloads are from the same origin but with larger organizations where web pages are built on top of custom CDNs and micro services a lot of resources are downloaded from sub-domains owned by the same organization. In the context of organizational trust resources tightly controlled by the organization under inspection are not as interesting as resources controlled by third-party organizations.

The definition of origin is strict for good reasons and works very well when talking about technical trust but to better cater for real life situations where most organizations can be assumed to have control over of their main domain, its sub-domains, available protocols and open ports this thesis introduces the terms same-domain and cross-domain. The definitions of these terms are much more relaxed than the definition of origin and is thus more suitable for inspecting the real world organizational trust relationships via resource downloaded by web pages. Applying this more relaxed definition when comparing origins between resources downloaded by a browser allows us to filter away most of the resources that in real life are controlled by the inspected organization and allows to focus more on the more interesting third-party controlled resources.

This thesis applies the cross-domain definition in practice when doing resource origin comparisons in the case study chapter when scraping over a large amount of front pages of .fi TLDs.

Chapter 4

Case study

4.1 Introduction to case study

This case study spans a time from February 2018 to March 2018. In this case study we focus on researching cross-origin JavaScript dependencies on home pages of Finnish websites by developing a dependency checker tool, running that tool with a specific configuration against around 260,000 websites that use the internet country code top-level domain (ccTLD) of Finland (.fi) [16] and analyzing the collected data. The development of this tool is our answer to **RQ1** and it makes possible the aggregation of a corpus that allows us to answer **RQ2** and **RQ3**.

The developed Dependency Checker tool has been published in GitHub [17] with MIT license for anyone to use. Though the focus of this thesis is in JavaScript dependencies the Dependency Checker tool is more general and can be used to inspect other dependencies like for example external Cascading Style Sheets [18], fonts and media.

4.2 Dependency checker tool

To be able to measure dependencies of web pages from different domains in large volumes and to collect the measured data I decided to create a command line tool that when given

a web page URL [19] would be able to gather the data and count some basic statistics of loaded dependencies.

4.2.1 Goals

The first goal for the dependency tool was for it to be able to keep track of all dependencies loaded by a web page whether the dependencies are loaded statically based on the initial HTML document delivered to the browser or loaded dynamically by some of the initially loaded scripts on the page. Lazy loading resources is a common design pattern used on the web to improve speed or perceived speed of web pages. Lazy loading on web pages is usually done after the `DOMContentLoaded` [20] event of the `document` or after the `load` [21] event of the window.

The second goal for the tool was for it to be able to categorize the found dependencies in a meaningful way. So that you could easily inspect or focus on selected dependency type instead of having to wade through them all.

The third goal was for the tool to be able to distinguish between cross-origin, cross-domain and subdomain dependencies. Cross-domain inspection is the strictest option as f.ex. if a page hosted at `https://www.example.com` loads resource from `http://example.com` the request is considered as cross-origin as the schemes of the origins do not match. A strict interpretation like that is necessary for security reasons at the browsers end but in this tool it might not be beneficial. That is why a goal was set so that you could tell the tool f.ex. to only consider the `hostname` [22] or even only the root domain when analyzing dependencies.

The fourth goal for the tool was for it to be able to recognize the organizations or individuals behind the loaded cross-origin dependencies. This would allow the user to easily see who is in control and of what dependencies.

4.2.2 Technical design

To meet to set goals a technical design to support them was done before starting the implementation of the tool. The first problem was how to solve detecting the dependencies. Static dependencies could be gathered from the loaded HTML document but it would not be able to catch the dynamically loaded dependencies. Thus a simple static analysis of HTML documents would not be enough. To be able to observe the dynamically loaded dependencies the web page would need to rendered in an environment that would be as close to a normal browser environment as possible so a Chrome based headless browser called Puppeteer was selected.

By their own words Puppeteer is a Node library which provides a high-level API to control headless Chrome or Chromium over the DevTools Protocol [23]. We evaluated the API provided by Puppeteer and decided that it would serve as a good base for the tool. As Puppeteer was meant to be used with Node the language of choice for writing the tool was selected to be JavaScript on Node. Also asynchronous features like `async` and `await` supported by latest Node versions were a strong argument for using Node when working in an inherently asynchronous environment like web.

Puppeteer API provided for example methods for opening a new page in the headless browser, waiting for the page `load` [24] event to fire and attach different kinds of observers to the page before it gets navigated to. As all cross-origin dependencies would need to be loaded over the network by the browser listening for requests made by the browser was selected as the point for gathering data. Puppeteer also provided APIs for measuring used CSS and JavaScript coverage of pages but I decided not to use it as the focus of this study wasn't in the efficiency between loaded and actually used resources but just in the loaded ones. Thus taking advantage of the `requestInterception` [25] method of the Puppeteer API (see 4.1).

The second bigger problem to solve was how to find the organizations or individuals behind the loaded resources. In the context of a web page resources can be downloaded

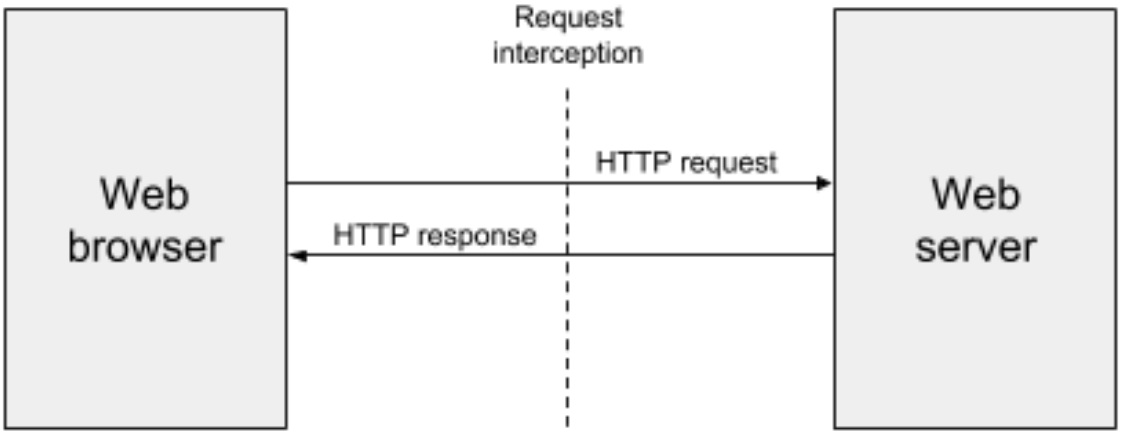

Figure 4.1: Request Interception in Puppeteer

from a wide variety of origins. As an origin always consists of scheme, hostname and port and the hostname is the only part that actually tells where the resource is instead of just defining how to access it is the only part that has potential for identification.

There is a query and response based protocol called WHOIS [26] that is intended for fetching information about internet resources like domain names and IP addresses in a human readable format. It is also the only currently available free and widely used way to acquire information about domain name ownership. What makes WHOIS a difficult protocol to deal with is that the response format isn't strictly defined and thus the content and the format of WHOIS responses vary wildly between different WHOIS servers. This makes it harder for computers to further process the data.

I decided to use WHOIS for querying all possible data for each different root domain found from the dependency origins and then try to parse owner information out of the different WHOIS responses.

4.2.3 Development

4.2.4 Usage

I will go shortly over the usage of the tool in this section but complete and always up to date instructions on how to install and use the tool can be found from the GitHub repository is README.md file.

Performing a basic test is as simple as running the following command in the root directory of the tool.

```
1 yarn start --url=https://example.com/
```

This kind of command will print out status information about the execution of the test while it runs and finally print out the results in to the console. In the command above it is good to note that if the server responding to URL `https://example.com` would for example perform a redirect to URL `https://www.example.com` the tool would consider all resources loaded from URL `https://www.example.com` to be cross-origin as the URL you asked for it to test was `https://example.com` and thus the origin differs.

So by default the URL you give will be used in the cross-origin or cross-domain comparisons as is. By setting the `--follow-redirects` flag the comparison URL will be automatically updated to the URL the initial document was downloaded from. A command that would tell the tool to automatically follow redirects would look like the example below.

```
1 yarn start --url=https://example.com/ --follow-redirects
```

Like mentioned in the Technical design subsection dependencies might get lazy loaded or dynamically loaded after the `load` event of the `window` object. To better capture dy-

namically or lazily loaded dependencies in the test we can tell the tool to wait a certain amount of milliseconds after the `load` event. A command example of waiting 5 seconds after the page has been loaded and thus giving time for dynamic dependencies to load can be seen below.

```
1 yarn start --url=https://example.com/ --wait=5000
```

By default the tool doesn't fetch the organizational information for the dependencies and doesn't print all the found dependencies to the console but only shows found dependency types and counts for same-origin and cross-origin dependencies. It also counts a cross-origin percentage for each of the found dependency types (see 3.2) and for the page in total. Possible dependency type categories are document, stylesheet, image, media, font, script, texttrack, xhr, fetch, eventsource, websocket, manifest and other [27].

To trigger the tool to print out all cross-origin dependencies and fetch owner data for them with WHOIS you'll need to use the `-l` handle that stands for "long output". Running commands with the `-l` handle take noticeably longer as the tool needs to perform separate WHOIS queries for each unique root domain and IP address among the found dependencies. For larger sites this might mean performing hundreds of WHOIS queries to be able to aggregate the results.

An example of a command that would list detailed data of all found cross-origin dependencies is shown below. It is good to note that if the site doesn't have any cross-origin dependencies none would of course be listed and the output would not differ from the output of the same command without the `-l` handle.

```
1 yarn start --url=https://example.com/ -l
```

As large sites can have huge amount of dependencies when media is counted it makes analyzing the data reported by the tool harder and finding the meaningful dependencies

becomes harder. As an example let is imagine that a page at `https://www.example.com` loads all of it is images from a content delivery network hosted in a subdomain `cdn.example.com`. For our tool this means that all of the images loaded by the page from that subdomain are cross-domain dependencies. The user of the tool might not be interested of the dependencies they load from their own content delivery network or from their own subdomains. It is possible to filter away all dependencies loaded from the subdomains of the root domain we're running the test for. To tell the tool to consider all dependencies loaded from subdomains of the root domain of the given `--url` handle we can use the `--ignore-subdomains` handle. Example of such a command can be seen below. By using this handle the comparison becomes cross-root-domain instead of cross-origin.

```
1 yarn start −−url=https://example.com/ −−ignore−subdomains
```

In some cases organizations might have subsidiaries etc. that host their own services in totally different root domains than the main organization. The case might be that some of the services provided and maintained by the subsidiaries are integrated f.ex. to the main organizations website. In this case it can be relevant to be able to filter out dependencies that are loaded from different root domains or subdomains owned by the same organization. This can be achieved by using the `--consider-trusted` handle. It basically allows you to list other domains that the tool will consider "same origin" when it is doing its comparisons. Multiple trusted domains can be inputted by repeating the handle. An example of a command that adds two trusted root domains that differ from the root domain the test is run for can be seen below.

```
1 yarn start −−url=https://example.com/ −−consider−trusted=www.other−domain.com
      −−consider−trusted=another−domain.net
```

We've now gone through how the tool can be used to analyze single web pages by hand and how we can affect its internal concept of what is considered trusted to be able to filter the most interesting results out of all the found ones. The tool also supports a couple of other handles that aren't very useful when doing by hand analysis from the command line but are valuable when the tool is called by another program.

The tool supports a `--silent` handle for disabling all progress logging and `--output` handle for selecting an output format for the data. At his point the tool supports only outputting the data in human readable format to the console and JSON output meant to be read by computers. The JSON output contains more detailed data of the dependencies than the human readable version enabled by default. Using these two handles together it is possible to make the program run silently and only output JSON when it is done. Below is an example of a command like that.

```
1 yarn --silent start --url=https://example.com/ -l --silent --output=json
```

Notice that in the above example we also add a `--silent` handle for yarn that we're using to invoke the start command defined in the package.json file of the project. We'll need to do this as yarn prints some output of it is own and in the case where we want the result only to contain valid JSON we can't let yarn progress logging to pollute that.

For an example of a pretty printed JSON output of the command like above run for page `https://www.vero.fi/henkiloasiakkaat/` and stripped so that only one request example is show per each found resource type see appendix A.1.

Testing

The testing of the dependency checker tool and it is different running configuration options was made by hand running the tool against different kinds of URLs. The results reported by the tool were then checked by hand to be correct. The tool doesn't have any automated test at this point but with more time such tests could be implemented.

4.3 Collecting data

4.3.1 Step one: collecting .fi domains

The first required part for running the planned test for a large amount of .fi TLD web pages was to gather a list of .fi TDL domain names. Finnish Communications Regulatory Authority (FICORA) provided an open OData API [28] that provides information about domain names registered by organizations and associations. This API does not include information about domain names registered by private individuals if they have not specifically selected that they want to publish their data when registering a domain. This is the reason our corpus does contain a small amount of domains that are associated with a private person. Despite the OData API not containing most of the domains registered by private individuals FICORA states that the data provided via this API contains roughly 80% of all registered .fi domains.

To be able to locally process all the data the API holds of the domains as part of this thesis a tool was written [29] that is capable of extracting all the data stored in the API in ~22 minutes. The tool supports saving the extracted data in two formats: a single JavaScript Object Notation (JSON) file and single comma-separated values (CSV) file[1]. This tool has also been published in GitHub as open source under the MIT license [29].

For the purpose of collecting data for this thesis the tool was installed on a virtual private server[2] and ran with the following parameters:

```
1 npm run start -- --no-csv
```

The duration of the scrape in the setup described above was 21 minutes and 52 seconds and resulted in a single JSON file with a size of 235 megabytes. The file included data of 371,331 unique .fi domains. In the gathered corpus we can directly see that ~91% of the

[1]The schema of the files can be checked from the GitHub repository of the tool

[2]Technical specifications of the VPS: 4 GB RAM, 2 CPU Cores, 48 GB SSD Storage, 40 Gbps Network In, 1000 Mbps Network Out, Ubuntu 17.10

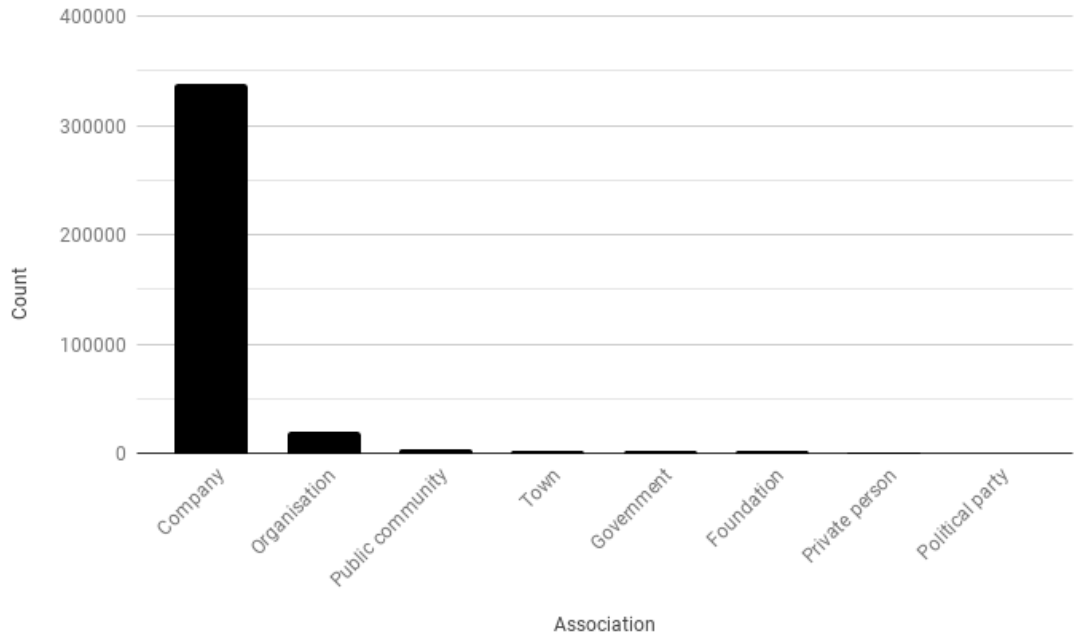

Figure 4.2: Domains per associated entity type

domains are owned by companies (see 4.2) and registered to Finland (see 4.3). Top 20 registrars control ~55% of the domains (see 4.4). The 8th largest registrar with name −−− in the graph is not really a registrar but is a default value Ficora uses to fill the registrar name field in the data if the domain does not have a registrar. This is the case for 2.57% of the domains.

4.3.2 Step two: gathering HTTP status codes for filtering

In the first step a large set of .fi domains was gathered but blindly running the dependency checker tool (see 4.2) for all of them would have taken quite a lot of time. The estimated execution time for running the dependency tool with the planned configuration (see 4.3.3) against the home pages of 371331 unique domains single threaded on the VPS was close to 43 days [3]. To take into account only domains we could gather data most likely from the list of domains gathered in the first step was decided to be filtered based on whether

[3](371331 domains x 10 s) / (60 s * 60 m * 24 h) = ~ 42.98 days

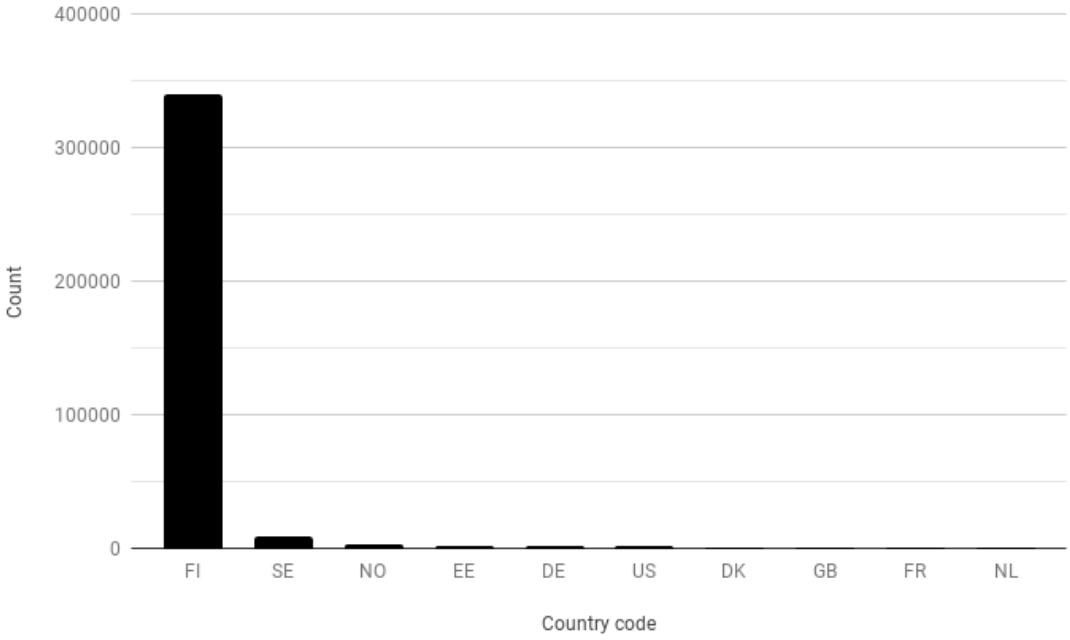

Figure 4.3: Top 10 home countries of domain users

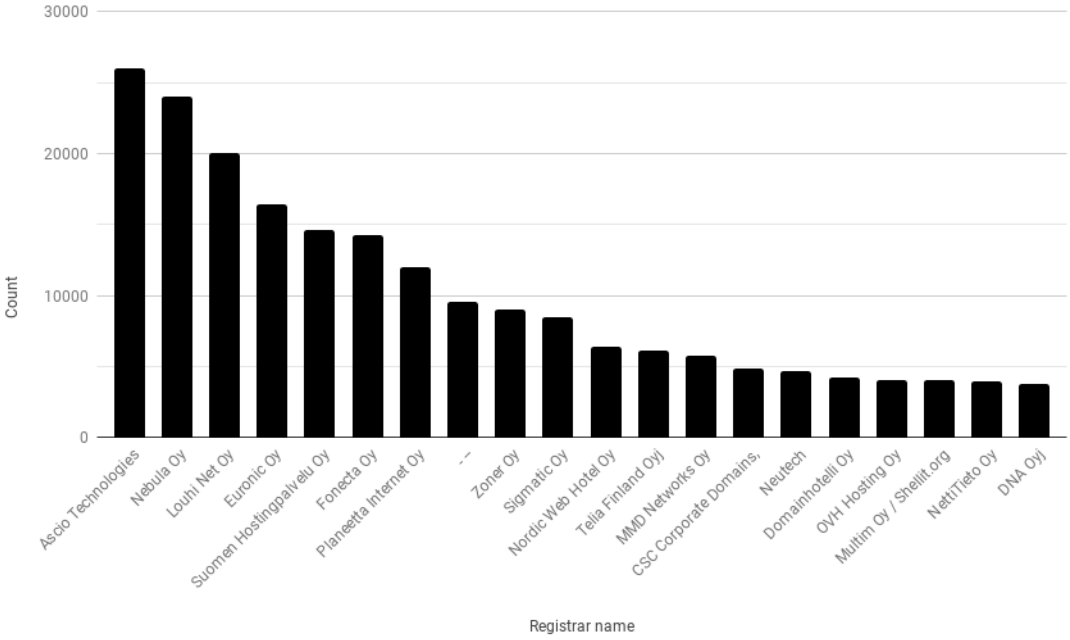

Figure 4.4: Top 20 registrars

a server would answer properly with a `200 OK` HTTP response code to a HEAD request sent by our VPS.

To prepare for the HTTP status code check the data was imported form the JSON file generated in the step one to a relational SQLite database. The database schema was designed to match the schema of the JSON file. SQLite was selected over other options for its easy portability and lightness. Importing all of the data asynchronously to a clean SQLite database with a Node.js[4] script took ~2 hours and 27 minutes.

After the import a second Node.js script was written the purpose of which was to iterate over all the domains imported to the SQLite database, send a HTTP HEAD request to the domain and save the response code to the same database for that domain.

For making the HEAD request and parsing the HTTP status code the servers cURL tool was used in the following form:

```
1 curl −s −o /dev/null −I −L −w ”%{http_code}” −−connect−timeout 60 −−max−
    time 120 <domain>.fi
```

In the bash command string above curl is the name of the tool and the rest are handles that control how it behaves. The `-s` handle makes curl silent so that it does not show progress meter or error messages. The `-o` handle and its value `/dev/null` directs the output of the command to `/dev/null` which means basically throwing it away. The `-I` handle instructs curl to fetch only the headers of the document using the HEAD command that most HTTP-servers support. The `-L` handle makes curl follow redirects if the server reports that the requested page has moved to a different location. The `--connect-timeout` handle and its value `60` tell curl that the maximum time it can wait for the connection to the server to open. The `--max-time` handle and its value `120` set a hard execution time limit for the whole command. The value for both of these time limiting commands is set in seconds. Finally the `-w` handle with the value `"%http_code"` makes curl to write only the numerical response code that was found in

[4]Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine

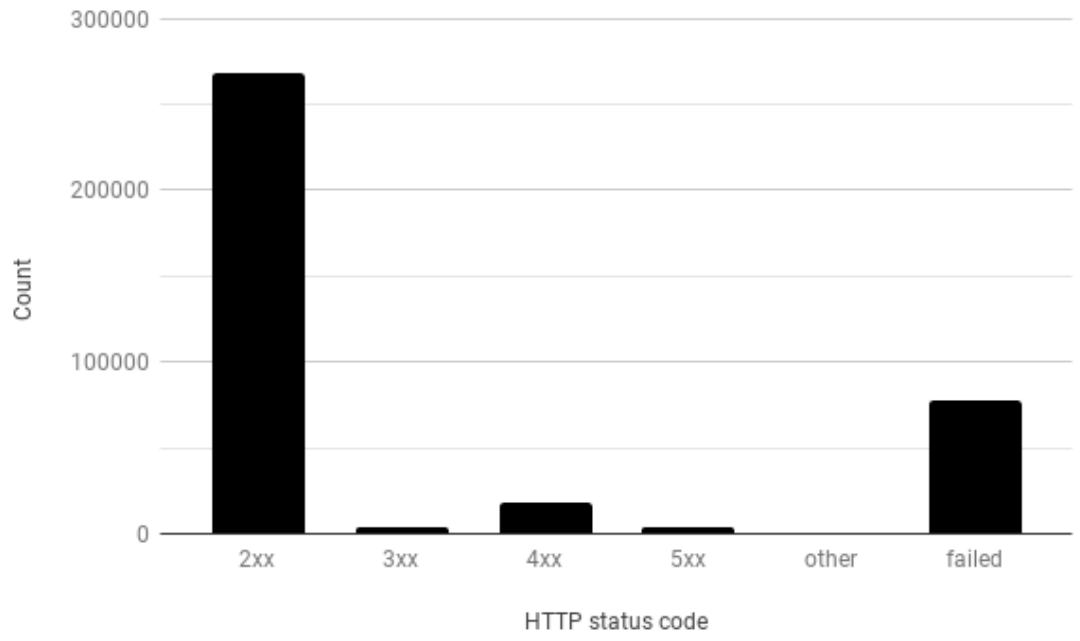

Figure 4.5: Counts of HTTP status codes ranges in the corpus

the last retrieved HTTP(S) transfer to stdout. This is what we want our script to read and save.

The Node.js script that was in charge of executing the above command for each domain in the corpus and saving the status code returned by the curl command to the database was written so that it was able to run 400 separate instances of the above command asynchronously and concurrently. This allowed gathering and saving the HTTP status codes in one run for all of the domains in ~4 hours and 3 minutes.

As we can see in the figure 4.5 72.30% of the domains in our corpus answer with the HTTP status code starting with number two (`2xx`), 1.04% with number three (`3xx`), 4.89% with number four (`4xx`), 0.91% with number five (`5xx`), 0.01% with a non standard status code and 21.16% of the queries failed. The fact that 20.86% of the checks resulted in to a status code `000` that in the figure is labeled as failed is surprising. The `000` is not a valid HTTP status code but a code that cURL returns if the Domain Name System (DNS) resolution fails, the connection refuses or times out or if the server for

some reason invalidly returns the status code `000` [30]. From this we can deduce that about 20% of registered .fi domains are not in any use at all.

4.3.3 Step three: scraping results

In the second step we gathered HTTP status codes for all domains in our corpus. This allowed us to pick only domains for the dependency check that looked healthy. All domains with a HTTP status code `200` were considered healthy and selected from the corpus for the dependency test. The exact number of domains were selected for further analysis based on the HTTP status code was 268,482. That is 72.30% of all the domains in our corpus.

The dependency data was gathered with the dependency checker tool described in this thesis (see 4.2). A small Node.js helper script was also written for this data gathering phase. The scripts purpose was to iterate over all domains in the corpus where the saved HTTP response code for the domain was `200` and skip all other domains. The script would then run the dependency checker tool with the following configuration for all the qualified domains in the corpus and save the JSON output returned by the dependency check to the SQLite database for later analysis.

The dependency checker tool was run with the following configuration:

```
1 yarn --silent start -l --follow-redirects --wait=5000 --output=json --silent
      --url=http://<domain>.fi/
```

The handles in the above command are explained in detail in the chapter that describes the dependency checker tool (see 4.2). In short the above command will perform the dependency test on the domain passed for it and when testing it will observe the network traffic triggered by the page for 5 seconds after the page `load` event has triggered [31]. This is to allow lazily loaded or triggered scripts to execute and possibly cause more HTTP requests that will then get recorded by the dependency checker tool. It is good to

note that by default the dependency checker tool will emulate iPad 10 when performing the test so that it would look more like a normal visitor to the server and web page it is testing. This makes it harder for servers and web pages to modify their behavior based on the default User Agent [32] of Puppeteer [23].

After running the dependency test for the 268482 unique domains we could inspect the results and saw that the dependency test succeeded for 265177 domains and failed for 3305 domains. All tests marked as failed were timeouts. The script in charge of running the dependency check for the selected domains had a 120 second maximum execution time set for individual tests. This means that if the dependency test took more than 120 seconds to complete for one domain the test was killed and reported as timed out. Only domains where the dependency test succeeded were selected for further analysis.

In our corpus based on the dependency check results 27 sites had initiated zero same domain requests. As the download of the initial document is counted as a request this kind of a situation should not be possible but there was a logical explanation for this. The dependency checker tool when run with the `--follow-redirects` and `--wait` handles picks up the domain it will use for the origin comparisons after the wait. If the server did not issue a redirect but returned a page with status code `200 OK` that used the meta refresh tag [33] or JavaScript to perform a client-side redirect to a page that fails to load[5] the dependency checker tool compares all made requests to an internal error page domain of Chromium. This leads to a situation where all requests made were considered cross-domain. These 27 domains with erroneous results where also filtered out when running further analysis for the data. The size of the domains in this filtered corpus was 265,150.

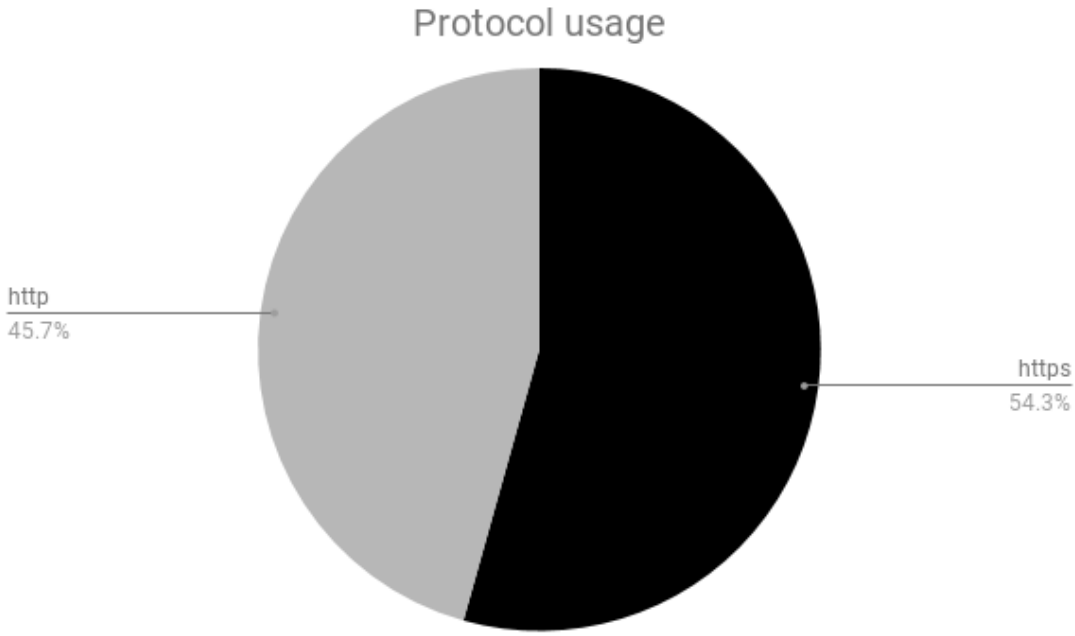

Figure 4.6: Protocol usage in request data

4.4 Analysis

After the dependency data collection our corpus of 265,150 .fi domains includes data of 12,754,376 separate requests. Of all those requests 54.3% were downloaded over `https` and 45.7% over `http` (see figure 4.6). Of the requests, 49.8% were same-domain and 50.2% cross-domain.

Looking at all of the requests in the corpus we can see that the largest amount of requests a single web page made was 1,435 and the smallest 1. On average a site makes roughly 48 requests. It is good to note that the median of 37 probably better reflects the ”real world average” (see table 4.1).

When we look at the same-domain requests only we can see that the largest amount of requests a page made was 1432 and the smallest 1. On average a single web page makes roughly 24 same-domain requests but again the median of 17 is probably better reflects the ”real world average” (see table 4.2).

[5]For example if the DNS resolution for the domain fails or the server refuses the connection.

Table 4.1: All requests

Property	Requests
Max	1435
Min	1
Average	48.10
Median	37
Standard deviation	46.72

Table 4.2: Same-domain requests

Property	Requests
Max	1432
Min	1
Average	23.94
Median	17
Standard deviation	27.00

Lastly when we only look at the cross-domain requests of our corpus we can see that the largest amount of cross-domain requests a site made was a staggering 1377 and the smallest amount was zero. Doing zero cross-domain requests is possible but doing zero same-domain requests is not as the initial HTML document needs to be downloaded from the same-domain. That sets the possible minimum of same-domain requests to one. On average a web page makes ~24 cross-domain requests but in our corpus the median is only 11 requests and again probably better reflects the ”real world average” (see table 4.3).

It is good to note that in these all three tables (4.1, 4.2 and 4.3) the standard deviation is relatively large which means that the amount of requests a web page makes varies a lot between the web pages in our corpus.

The figure 4.7 shows a stacked bar chart that on the x-axis shows request dispersion

Table 4.3: Cross-domain requests

Property	Requests
Max	1377
Min	0
Average	24.17
Median	11
Standard deviation	34.64

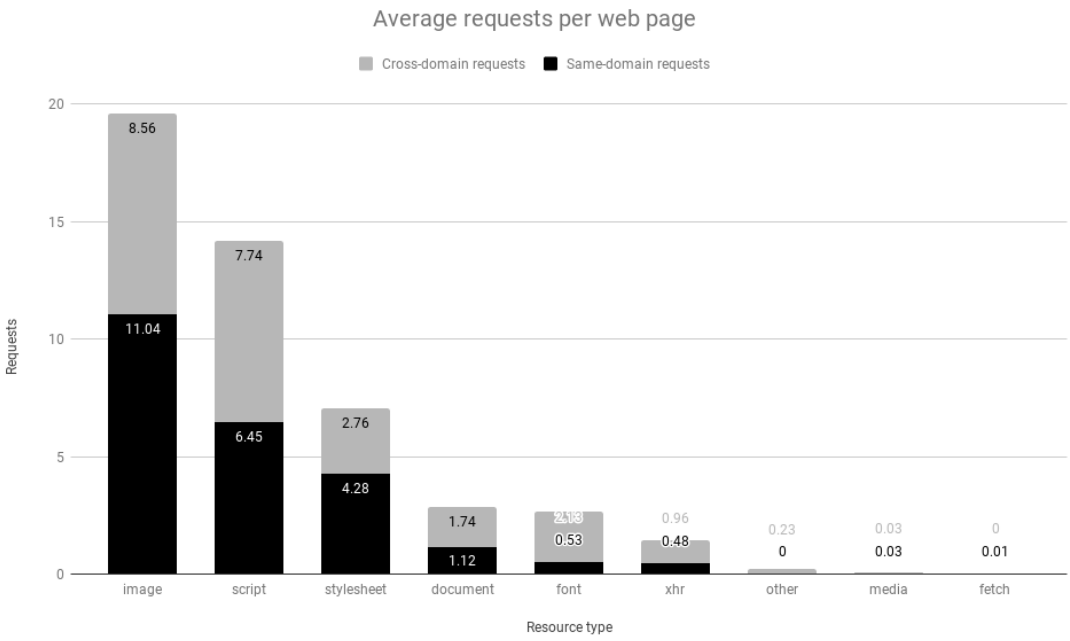

Figure 4.7: Same-domain vs. cross-domain per resource type

between different resource types. On the y-axis an average request count is shown for that particular resource type and how many of the requests on average are cross-domain and how many on average are same-domain.

From the graph we can see that the most downloaded resource type is image which after text is the most common media type on web pages. After that comes scripts and style sheets. The amount of on average downloaded scripts being relatively close to the amount of downloaded images supports our initial intuition of scripts being used a lot. What comes to the same-domain vs. cross-domain we can see that it is roughly half and half for almost all resource types. Images are downloaded a bit more from same-domain than cross-domain and scripts a bit less. This points to the direction that using third party scripts is a bit more common than using your own.

In figure 4.8 we can see how cross-domain requests are divided between the country of the registrant. This means that we have gone through all cross-domain requests, tried to find those requests a registrant with country information and then counted the results

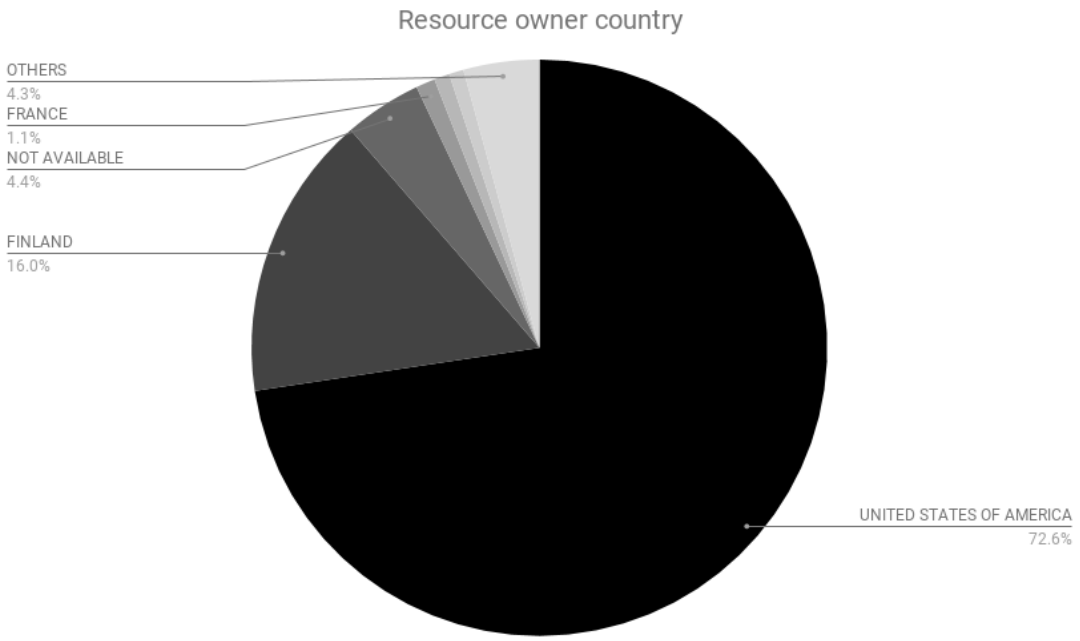

Figure 4.8: Requests by country of the registrant

per country.

As we can see a staggering 72.6% of the resources at the end of these cross-domain requests are controlled by entities in USA. Only 16% of the resources are controlled by entities in Finland. France accounts for 1.1% and all other 152 countries for 4.3% (marked as OTHERS). For 4.4% we were unable to automatically determine the country of the registrant (marked as NOT AVAILABLE).

The fact that 72.6% of the resources are controlled by entities in USA correlates with the previous tables about popular URLs we have looked at as companies like Google, Facebook and Twitter are registered in USA.

4.4.1 Popular hostnames

We have listed top 20 hostnames to which the web pages in our corpus made the most requests (see table 4.4). The top 20 hostnames gather 3,823,779 requests of the total 12,754,376 requests. It means that around 30% of all the requests go to these 20 host-

names.

If we look at the organizations behind these top 20 hostnames we can see that Google controls roughly 37% of them, Facebook around 13% and Fonecta Oy 10% – Fonecta being the most significant Finnish company in the list. Some of the hostnames allow us to deduce something of the nature of the services behind them. By looking at the top 20 hostnames from this perspective we can see that Google fonts, maps, analytics and video (YouTube) are popular products on the Finnish market. It also seems to be quite popular to connect your page to Facebook in one way or another. Embedding Instagram photos and Tweets also seems to be popular. The RCMS is a CMS (content management system) product owned by Valve Branding Oy and is used by Fonecta in their own systems. The other identifiable closed source CMS's in the top 20 domains are Kotisivukone and Wix.

4.4.2 Popular URLs

In the table 4.5 we have listed top 30 URLs. The list was aggregated from all the requests in the corpus. A URL was considered unique based on the combination of origin and pathname. Query strings were not considered when aggregating the list. From the list we can naturally see the same kind of corporate landscape than we already saw on the top 20 hostnames list. Google and Facebook dominate the list while Wix is the only visible third player on it. Fonts, analytics, maps and ads account for almost all of the top 30 URLs.

4.4.3 Popular JavaScript and CSS resources

In the table 4.6 we list the top 20 JavaScript URLs. Like with popular URLs (see table 4.5) the list was aggregated from all the requests in the corpus and a URL was considered unique based on the combination of origin and pathname. Addition to this the URLs were filtered based on their pathname. If the pathname would end to a character sequence like `.js` it was included. Otherwise it was discarded. The same applies for the table 4.7 with the exception that the matched character sequence for the end of the pathname was `.css`

Table 4.4: Top 20 hostnames

Hostname	Request count
www.facebook.com	703,062
fonts.gstatic.com	422,628
www.google-analytics.com	384,459
maps.googleapis.com	259,060
fonts.googleapis.com	229,909
scontent-frx5-1.xx.fbcdn.net	192,888
static.parastorage.com	162,361
varattu.domainkeskus.com	149,245
www.google.com	111,942
connect.facebook.net	96,403
www.youtube.com	93,411
maps.gstatic.com	90,751
use.typekit.net	88,318
rcms-f-production.s3.amazonaws.com	69,148
ajax.googleapis.com	53,959
staticxx.facebook.com	53,626
stats.g.doubleclick.net	53,382
scontent.cdninstagram.com	52,986
pbs.twimg.com	52,479
cdn.kotisivukone.fi	51,630
service.giosg.com	51,395
maps.google.com	48,236
insight.fonecta.fi	47,553
www.planeetta.net	46,795
beacon.krxd.net	44,704
googleads.g.doubleclick.net	44,578
cdn.krxd.net	44,521
asiakas.kotisivukone.com	44,255
frog.wix.com	41,031
s7.addthis.com	39,047

Table 4.5: Top 30 URLs

URL (origin + pathname)	Request count
http://fonts.googleapis.com/css	117,222
https://fonts.googleapis.com/css	111,282
https://www.google-analytics.com/analytics.js	92,726
https://www.google-analytics.com/r/collect	84,088
https://maps.googleapis.com/maps/vt	60,050
https://stats.g.doubleclick.net/r/collect	46,554
https://www.facebook.com/tr/	44,032
http://www.google-analytics.com/analytics.js	32,857
https://staticxx.facebook.com/connect/xd_arbiter/r/Ms1VZf1Vg1J.js	32,724
https://www.google-analytics.com/collect	31,592
https://beacon.krxd.net/optout_check	29,211
http://www.google-analytics.com/r/collect	27,362
https://www.google.com/maps/vt	26,345
https://googleads.g.doubleclick.net/pagead/id	26,231
https://frog.wix.com/bt	25,182
https://fonts.gstatic.com/s/roboto/v18/KFOmCnqEu92Fr1Mu4mxMKTU1Kg.woff	24,471
https://www.google.com/ads/ga-audiences	22,660
http://maps.googleapis.com/maps/vt	22,241
https://www.facebook.com/rsrc.php/v3/yI/l/0,cross/0Xlu5eqfuG_.css	21,653
https://www.facebook.com/rsrc.php/v3/yQ/l/0,cross/bQHxYl9iQJ4.css	21,653
https://www.facebook.com/rsrc.php/v3/y0/r/NzGftzPVAEh.js	21,361
https://www.facebook.com/rsrc.php/v3/yU/r/WVTlt39y1Y8.js	21,342
https://www.facebook.com/rsrc.php/v3/yv/r/77Cj57ioOEE.js	20,993
https://www.facebook.com/rsrc.php/v3/yS/l/0,cross/kJpSEF3CePB.css	20,852
https://www.google-analytics.com/ga.js	20,488
https://www.facebook.com/rsrc.php/v3/y-/r/KsG3u3GrOUd.png	20,358
http://www.google-analytics.com/ga.js	20,337
http://staticxx.facebook.com/connect/xd_arbiter/r/Ms1VZf1Vg1J.js	20,266
https://www.google-analytics.com/r/__utm.gif	18,002
http://www.google-analytics.com/r/__utm.gif	18,002

instead of `.js`.

It is good to note that though here we are filtering JavaScript and CSS files from the requests based on their filename extensions it is possible for a path without the extension to return CSS or JavaScript. This means that the way of filtering we have used is naive but it is naive in purpose. By using only the filename extension for filtering we increase the likelihood of catching URLs related to third-party libraries like for example Bootstrap[6]

As in the previous two tables (see 4.4 and 4.5) here too Google and Facebook dominate the scene. They are the only two visible players on the top 20 JavaScript URLs table. The services behind these URLs are mostly related to analytics, ads and embedded YouTube videos. No popular open source libraries or frameworks are visible among the top 20 JavaScript URLs.

The case with the top 20 CSS URLs is somewhat similar (see table 4.7) but in the table we can see a few identifiable open source libraries. Among these URLs Google, Facebook and Twitter again control most of the scene but we can see that a CSS file related to Font Awesome icon library [34] is quite popular. The library is loaded from `maxcdn.bootstrapcdn.com` that is related to Bootstrap mentioned before. Thus we can deduce that Bootstrap is relatively popular on Finnish websites.

Other CSS URLs that catch the eye are downloaded from `cdn.kotisivukone.fi` and `rcms-f-production.s3.amazonaws.com`. Kotisivukone seems to use a custom version of jQuery UI [35]. Some of their own CSS files named `common.css` and `common_responsive.css` are also quite popular. It is probable that these files are downloaded by default on almost all pages created with Kotisivukone.

The `rcms-f-production.s3.amazonaws.com` hostname is related to the RCMS that was mentioned earlier. They seem to be loading some common/shared theme files as well and use Fancybox that is a jQuery plugin [36]. It is also probable in this case that the files are loaded out of the box on most of the sites created with RCMS.

[6]Bootstrap refers to Twitter Bootstrap. More information of it can be found here: https://getbootstrap.com/

Table 4.6: Top 20 JavaScript URLs

URL	Request count
https://www.google-analytics.com/analytics.js	92,726
http://www.google-analytics.com/analytics.js	32,857
https://staticxx.facebook.com/connect/xd_arbiter/r/Ms1VZf1Vg1J.js	32,724
https://www.facebook.com/rsrc.php/v3/y0/r/NzGftzPVAEh.js	21,361
https://www.facebook.com/rsrc.php/v3/yU/r/WVTlt39y1Y8.js	21,342
https://www.facebook.com/rsrc.php/v3/yv/r/77Cj57ioOEE.js	20,993
https://www.google-analytics.com/ga.js	20,485
http://www.google-analytics.com/ga.js	20,334
http://staticxx.facebook.com/connect/xd_arbiter/r/Ms1VZf1Vg1J.js	20,266
https://connect.facebook.net/en_US/fbevents.js	17,910
https://www.googletagmanager.com/gtm.js	17,412
https://www.facebook.com/rsrc.php/v3/yp/r/TBQDewS0gKn.js	15,733
https://static.doubleclick.net/instream/ad_status.js	15,336
https://www.google.com/js/bg/lSvH2GMDHdWiQ5txKk8DBwe8KHVpOosizyQXSe1BYYE.js	14,963
https://www.facebook.com/rsrc.php/v3/y2/r/tQ_sLmrosus.js	14,680
https://connect.facebook.net/fi_FI/sdk.js	14,595
https://www.facebook.com/rsrc.php/v3/yU/r/AM6qVQ7gqTd.js	14,083
https://www.youtube.com/yts/jsbin/www-embed-player-vflSh-IWH/www-embed-player.js	13,418
https://www.youtube.com/yts/jsbin/player-vflMfSEyN/en_US/base.js	13,214
https://maps.googleapis.com/maps-api-v3/api/js/32/6/util.js	12,286

Table 4.7: Top 20 CSS URLs

URL	Request count
https://www.facebook.com/rsrc.php/v3/yI/l/0,cross/0Xlu5eqfuG_.css	21,653
https://www.facebook.com/rsrc.php/v3/yQ/l/0,cross/bQHxYl9iQJ4.css	21,653
https://www.facebook.com/rsrc.php/v3/yS/l/0,cross/kJpSEF3CePB.css	20,852
https://www.youtube.com/yts/cssbin/www-player-sprite-mode-vflb71xJh.css	16,989
https://www.facebook.com/rsrc.php/v3/yX/l/0,cross/1L0gwr0mSGQ.css	12,693
https://www.facebook.com/rsrc.php/v3/y3/l/0,cross/uqUo2rrLKdz.css	11,861
https://www.facebook.com/rsrc.php/v3/yJ/l/0,cross/26V-iCD0O0M.css	9,378
https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css	5,470
https://cdn.kotisivukone.fi/libs/jquery/ui/css/jquery-ui-1.8.20.custom.min.css	5,467
https://cdn.kotisivukone.fi/r201/b2481/clients/css/common.css	5,466
https://platform.twitter.com/css/timeline.529166ecfeb05abf3ee5afe0a8c349a4.light.ltr.css	5,340
https://www.facebook.com/rsrc.php/v3/yQ/l/0,cross/s-0cTMMtN2G.css	4,790
https://www.facebook.com/rsrc.php/v3/yt/l/0,cross/L8mrIXnxuCy.css	4,784
https://ton.twimg.com/tfw/css/syndication_bundle_v1_2801d83f2f75998762a22055f578875d6e10fd1d.css	4,513
http://rcms-f-production.s3.amazonaws.com/themes/_default/hide.css	4,295
http://rcms-f-production.s3.amazonaws.com/js/release/jquery/fancybox/jquery.fancybox.css	4,295
http://rcms-f-production.s3.amazonaws.com/themes/_default/cookie-policy.css	4,293
http://rcms-f-production.s3.amazonaws.com/themes/fonecta/fonectaframework/framework-v2.css	4,151
http://rcms-f-production.s3.amazonaws.com/themes/fonecta/fonectaframework/base-v2.css	4,151
https://cdn.kotisivukone.fi/r201/b2481/clients/css/responsive/common_responsive.css	4,145

4.4.4 Landscape and threats

As we have seen in the results presented in chapter 4.4 in the landscape of Finnish websites about half of all resources are downloaded from somewhere (and in most cases someone) else than the website at hand. We have also seen that the cross-domain resource landscape regarding JavaScript and CSS is mostly dominated by a couple of big corporations the two biggest ones being Google and Facebook.

Of the web pages in our corpus 69.84% opened a connection to an origin that contained at least one of the following substrings `google`, `youtube`, `adsense` or `adwords`. These strings are commonly know to be used by Google on its domain names and thus are a good fit for approximating connections to Google services when looking at a request origin. Based on this naive but effective test we can say that **roughly 70% of Finnish websites give data about their users to Google**.

We performed the same kind of test for Facebook and we can see that 21.81% of the web pages in our corpus opened a connection to an origin that contained at least one of the following substrings `facebook`, `fbcdn`, `fbsbx` or `instagram`. Based on this we can say that **roughly 22% of Finnish websites give data about their users to Facebook**.

From a technical point of view extensive usage of third-party dependencies has the possibility to greatly expand the attack surface of websites. In case of large operators like Google and Facebook the resources provided to third parties are a cornerstone of their business and well taken care of. Having large economic resources and interest in taking care of these technical resources greatly reduces the risk of them being hacked, maliciously used or even sold. This means that using these kind of dependencies provided by large companies is relatively safe and the real effect that using these dependencies has to your attack surface or loose of control is relatively small. What adding third-party resources will always do though is weaken the privacy of your site.

Like mentioned earlier in chapter 4 if we look at the dependencies related to Google and Facebook we can identify some of the products behind these dependencies and de-

duce possible reasons why these dependencies might have formed addition to Google and Facebook just being strong brands.

From Google's products Google Fonts, Google Analytics, DoubleClick, Google Ads, Google Maps, Google Tag Manager, reCAPTCHA and YouTube are widely used based on the tables 4.4, 4.5, 4.6 and 4.7. Based on the same data we can see that from Facebook the Facebook Pixel is Facebook's most used service. The other popular Facebook services are "web plugins" and custom built integrations with Facebook Web SDKs. Of the plugins Comments, Facepile and Like Button seem to be the most popular ones.

If we look at the different widely used Google services we can identify a user needs that they fulfill. Website owners like to use custom fonts on their websites. Custom fonts are a great way to personalize your site from a visual and brand perspective. Google Fonts service provides a wide variety of web fonts and all its fonts are free to use. Addition to that they provide good instructions on how to use the fonts on your website and the overall quality of the service is high. All of these make a pretty strong case for using their fonts. It is also good to note that most content management systems provide plugins that allow the end users to customize the fonts on their sites and some of them provide easy access to all fonts in the Google Fonts service [37] [38].

As with Google Fonts we would argue that the case with Google Analytics, Google Maps, YouTube and so forth is similar. They are all free and high quality products that are easy to use. This coupled with Google's strong brand and huge user volumes they are able to push their own services to their users if they so wish and why would not they wish that as more users using their services means more data of the users and their behavior for them. The saying "If You're Not Paying for It; You're the Product" might already sound like a cliche but it is sadly true.

What comes to the trending products from Facebook like the Pixel and different web plugins they tell a similar story. Pixel provides entities advertising through Facebook the possibility to create a loop-back link from their own website back to Facebook that again

allows them to more efficiently target advertising etc. With the Facebook web plugins directed towards website owners Facebook allows users to pull some real time data from their accounts or pages and display that data on their own site. From a website owner or developer perspective this is a very easy way to integrate two different services. From Facebook's perspective it is good customer service but they also conveniently get a beacon that can report back who visited that site creating an information loop like the Pixel.

If we think about the products Google and Facebook provide they share quite a few similar properties. They are all free (if you do not count the money you need to pay for advertising), high quality and easy to use. As a website owner or a developer there is a lot to gain by using them. For example implementing a sophisticated user tracking software like Google Analytics could easily take years and cost millions but with Google Analytics it can be done in matter of minutes and it costs nothing. With Facebook the media you have created and the communities you have built can be reused on your own systems only via the options they provide (like web plugins). Luring in users with free quality software and locking them in to your system are different strategies but both of these strategies lead to a similar situation where data of your users flows back to these entities. By using free Google or Facebook services we expand their ”sensor network” around the web and facilitate their core business that is selling advertising. The more they know about us the more they can sell ads and make money.

We would argue that the usage of third-party cross-domain dependencies can create at least the following threats:

- Increased attack surface
- Loss of privacy and information leaks
- Loss of control

As a whole the chapter 4 together with chapters 5 and 6 answers our **RQ2** and **RQ3**.

Chapter 5

Threats and possible countermeasures

In this chapter we go through a non-exhaustive list of possible treats related to third-party cross-domain dependencies and countermeasures that can help in working more safely with (third-party) dependencies. This chapter will not go technically deep into the different possible solutions but it will introduce them and provide some citations for learning more on the subject. This chapter also partially answers **RQ3**. In this chapter we have divided the countermeasures under different threat categories and introduced concrete methods under them but it is good to note that some of the solutions overlap multiple categories.

5.1 Increased attack surface

Like mentioned already in a theoretical risk scenario in section 2.2.3 cross-domain third-party dependencies will increase the attack surface of your application compared to not having these dependencies at all. How much these cross-domain third-party dependencies will increase the attack surface of your site is debatable. Dependencies controlled by large organizations are usually better taken care of due to bigger available resources for taking care of the provided resources. The third-party resources having large value for the core business of the owner of the dependency will make it more likely for the dependency to

be taken care of and more resources allocated for that purpose.

5.1.1 Local copies

Third-party dependencies can be added to web pages using for example public content delivery networks like jQuery CDN or Google Hosted Libraries. The advantage of loading a dependency from a CDN is mostly speed but every time you use a CDN that you do not control you are giving away some information about your site and your users. One way to improve privacy with these kind of dependencies is to create local copies of them. By hosting these dependencies yourself you will loose the speed gain that your would have achieved by using a third party CDN but what you have lost in speed you have gained in privacy and control.

5.1.2 SSL

Loading cross-domain third-party dependencies over HTTP exposes you to a larger risk of man in the middle attacks. This means that in theory someone could be actively eavesdropping and replacing the contents of the third-party dependencies without you knowing. Using HTTPS protocol instead of the plain HTTP protocol you will make a man in the middle attack harder to execute.

In the world wide web today it is preferable to serve everything over HTTPS instead of HTTP. Google for example has announced that its search algorithm takes HTTPS into account when ranking search results [39]. In other words it could be said that Google penalizes sites that do not use the HTTPS protocol.

5.1.3 Security Audits

Like with all software performing recurring security audits can help in spotting security problems early. In terms of security audits for third-party cross-domain dependencies at

least all security aspects listed in this thesis should be taken into consideration. Security audits are best performed by a third-party due to objectivity reasons. A wide variety of security related consultancies offer security auditing related services and testing.

Addition to consultancies a variety of different kinds of semi-automatic test suites and tools exist [40]. These kinds of vulnerability scanning tools can be used to run tests for existing infrastructure and software to get a quick overview of the robustness of the software against known threats and attacks.

5.2 Loss of privacy and information leaks

Like noted in section 4.4.4 and theorized in section 2.2.2 using cross-domain third-party dependencies will always cause a certain level of loss of privacy and when used negligently can lead to unintended information leaks. The level of privacy lost depends on many things on the server and client side. We will not go deep into the technical details but rather point out that this is a thing that should be taken into consideration when adding dependencies. Every request a web browser makes carries a certain amount of information with it. This amount of information can and should be limited to the minimum required amount required by the web page to work and stay secure to minimize the potential risk of loss of privacy and information leaks.

A simple practical way of testing what kind of data your web page sends when loading a dependency is to capture the network request and inspect its contents and headers. This will allow you to evaluate what kind of information you are giving away just by downloading the dependency.

5.2.1 Proxying

One way to prevent information leaking is to proxy requests through a server that you control. This will help in anonymizing user requests and gives you the power to decide

exactly what information you send with the requests. Depending on the situation proxying can be either an easy task or an unfit solution. For anonymizing requests to static public resources (like for example Google Fonts) is relatively easy but implementing anonymization for resources that for example require authentication can be a more difficult task.

Proxying can have all the same effects as local copies (see subsection 5.1.1) but depending on the proxy implementation you can allow the dependencies to change dynamically that which in case of local copies require manual updating of the dependencies or setting up a system that takes care of automatically updating the local copies of the dependencies. This kind of a system closes the purpose of a proxy and is probably best implemented as a proxy.

5.3 Loss of control

We would argue that one of the biggest issues with third-party cross-domain dependencies is the lost control. Lets say that you add a simple Google Analytics script to your site. That script is downloaded every time someone accesses your site and the caching time Google sets for that script is 2 hours. Lets say that you inspected the Google Analytics script file when you added the dependency. Do you know how often Google changes the contents of the script file? You can probably find that out but most of us do not do that. We add third-party dependencies to our sites and then just forget them.

When adding cross-domain third-party script dependencies on web pages you are giving these dependencies addition to a vast access to your page and user a possibility to change the contents of the loaded scripts and thus to change what these scripts do. These script could for example load more scripts from totally different servers, redirect your users to a totally different site or send all visible data on the page forward including data requiring logging in.

5.3.1 Subresource Integrity

One way to limit the threats that emerge form the loss of control using cross-domain third-party scripts cause is to take advantage of a pretty well supported modern browser feature called subresource integrity. This feature simply enables browsers to verify that a file is delivered without unexpected manipulation. It is based on a new `integrity` attribute supported by the HTML `script` and `link` tags. The integrity attribute contains a cryptographic hash of the representation of the resource the author expects to load [41].

In simple terms it means that when you add a dependency to your web page you can ensure that the contents of the dependency do not change without you updating the hash in the `integrity` attribute of the script tag that invokes the download of the resource. This feature is a rather simple but very powerful way of ensuring that third-party scripts are what you think they are. As a security feature it is also relatively easy to use.

5.3.2 Content Security Policy

Another way to limit the powers of possibly unknown contents of third-party scripts is to use a content security policy (CSP). Content security policy can be triggered in supported browsers by setting a `Content-Security-Policy` HTTP header or a specifically formed HTML `meta` tag [42].

With CSP the browser can be instructed for example to disable all inline scripts or even disable scripting totally. It is also possible to enforce the browser to limit downloaded resources only to certain domains and this can be even done per resource type if necessary. It is also possible to enforce the use of HTTPS protocol on all connections and so fort. The specification is large and its third version is still very much a work in progress though the two previous versions are quite well supported by the major browsers.

Chapter 6

Conclusions

In this thesis we have researched how to technically measure dependencies on web pages (**RQ1**), how does the dependency landscape on Finnish web pages look like on a large scale and on why does it look like it does (**RQ2**) and what kind of threats the current situation has created (**RQ3**).

Measuring cross-domain dependencies on web pages is possible. A good way to do so is to intercept and analyze requests made by browsers when loading a web page. Collecting data of the requests and responses allows analysis on many levels. During the case study (see chapter 4) of this thesis a tool for this purpose was created and the tool tested with a large corpus of Finnish websites.

In this thesis we have focused on analyzing the origins and controllers of the origins of the requested resources, the type division of requests between different resources types recognized by browsers and protocols used for accessing these resources. Our corpus shows that about half of the requests made are cross-domain. We have seen that about half of the requests are made over HTTP and the other other half over HTTPS. We know that almost 73% of the resources downloaded by the web pages in our corpus are controlled by entities registered in United States of America and only 16% by entities registered in Finland. We also know that in average only the count of downloaded images exceeds the amount of downloaded scripts.

We have also established that the cross-domain dependency scene on Finnish websites is dominated by Google and Facebook and determined that they have achieved this market dominance by providing ”free to use” quality products that match user needs. The motivation for providing these services is that they support the core business of these companies – selling ads – by aggregating data of the users of these services that can be then used to target adds better.

The kind of business model that the ”Big Friendly Giants” of the web (mainly Google and Facebook) run has had an effect on the Finnish web scene and in terms of privacy it is not positive. In general we can say that by using Finnish websites some information of you as a user is more often than not sent to Google, Facebook or both. Having a small set of dependencies shared by a large volume of websites and controlled by only a couple of entities creates a ”single point of failure” that in theory can have security implications of catastrophic scale.

We advice that the users of third-party dependencies should deeply understand what the dependencies they use really do, hold the controllers of these dependencies accountable for what they do with them and preferably automatically monitor changes in dependencies and enforce limitation on when the contents and functionality of these dependencies can change.

During the case study performed in chapter 4 data of the responses to the requests performed were not collected. It would be interesting to perform a study that collects data of the responses too. This would allow determining for example things like average page weight in terms of data downloaded ”over the wire” and average page render time that can not be deduced from the corpus collected in this thesis.

References

[1] TCPA - Trusted Computing Platform Alliance - YouTube. https://www.youtube.com/watch?v=mLoIcdIu_Kk. Accessed: 2018-05-6.

[2] Windows Defender ATP thwarts Operation WilySupply software supply chain cyberattack – Microsoft Secure. https://cloudblogs.microsoft.com/microsoftsecure/2017/05/04/windows-defender-atp-thwarts-operation-wilysupply-software-supply-chain-cyberattack/. Accessed: 2018-03-25.

[3] Backdoor in Captcha Plugin Affects 300K WordPress Sites. https://www.wordfence.com/blog/2017/12/backdoor-captcha-plugin/. Accessed: 2018-04-02.

[4] Tietosuojavaltuutettu selvittää S-Pankin Google Analytics -käyttöä - Tivi. https://www.tivi.fi/Kaikki_uutiset/2015-02-26/Tietosuojavaltuutettu-selvitt%C3%A4%C3%A4-S-Pankin-Google-Analytics--k%C3%A4ytt%C3%B6%C3%A4-3216398.html. Accessed: 2018-03-26.

[5] HTML Standard. https://html.spec.whatwg.org/multipage/. Accessed: 2018-03-20.

[6] Browser - Glossary — MDN. https://developer.mozilla.org/en-US/docs/Glossary/Browser. Accessed: 2018-03-20.

[7] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616. Accessed: 2018-03-20.

[8] Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/. Accessed: 2018-03-20.

[9] A discussion about the breaking of the Internet – Mike Roberts – Medium. https://medium.com/@mproberts/a-discussion-about-the-breaking-of-the-internet-3d4d2a83aa4d. Accessed: 2018-03-20.

[10] I've Just Liberated My Modules - Azer Koçulu's Journal. http://azer.bike/journal/i-ve-just-liberated-my-modules/. Accessed: 2018-03-20.

[11] webRequest.ResourceType - Mozilla — MDN. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest/ResourceType. Accessed: 2018-03-21.

[12] extensions/common/api/web_request.json - chromium/src - Git at Google. https://chromium.googlesource.com/chromium/src/+/master/extensions/common/api/web_request.json. Accessed: 2018-03-20.

[13] chromium/resource_type_struct_traits.cc at master · chromium/chromium. https://github.com/chromium/chromium/blob/master/content/public/common/resource_type_struct_traits.cc. Accessed: 2018-03-21.

[14] RFC 6454 - The Web Origin Concept. https://tools.ietf.org/html/rfc6454. Accessed: 2018-03-19.

[15] Domain - Glossary — MDN. https://developer.mozilla.org/en-US/docs/Glossary/Domain. Accessed: 2018-03-19.

[16] Finnish Communications Regulatory Authority. https://www.viestintavirasto.fi/en/fidomain.html. Accessed: 2018-02-28.

[17] sarukuku/dependency-checker: A command line tool that checks your web page's dependencies, what percentage of them are cross-origin or cross-domain and who's in control of them. https://github.com/sarukuku/dependency-checker. Accessed: 2018-03-20.

[18] Cascading Style Sheets. https://www.w3.org/Style/CSS/. Accessed: 2018-02-28.

[19] W3C, Infrastructure, URLs. http://w3c.github.io/html/infrastructure.html#infrastructure-urls. Accessed: 2018-02-28.

[20] DOMContentLoaded - Event reference — MDN. https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded. Accessed: 2018-03-01.

[21] load - Event reference — MDN. https://developer.mozilla.org/en-US/docs/Web/Events/load. Accessed: 2018-03-01.

[22] Origin - HTTP — MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin. Accessed: 2018-03-01.

[23] GoogleChrome/puppeteer: Headless Chrome Node API. https://github.com/GoogleChrome/puppeteer. Accessed: 2018-03-01.

[24] puppeteer/api.md at v1.1.1 · GoogleChrome/puppeteer. https://github.com/GoogleChrome/puppeteer/blob/v1.1.1/docs/api.md#pagegotourl-options. Accessed: 2018-03-01.

[25] page.requestInterception() · puppeteer/api.md at v1.1.1 · GoogleChrome/puppeteer. https://github.com/GoogleChrome/puppeteer/blob/v1.1.1/docs/api.md#pagesetrequestinterceptionvalue. Accessed: 2018-03-02.

[26] RFC 3912 - WHOIS Protocol Specification. https://tools.ietf.org/html/rfc3912. Accessed: 2018-03-01.

[27] request.resourceType() · puppeteer/api.md at v1.1.1 · GoogleChrome/puppeteer. https://github.com/GoogleChrome/puppeteer/blob/v1.1.1/docs/api.md#requestresourcetype. Accessed: 2018-03-02.

[28] Viestintävirasto - Technical interfaces. https://www.viestintavirasto.fi/en/fidomain/informationforregistrars/technicalinterfaces.html. Accessed: 2018-03-21.

[29] sarukuku/fi-tld-scraper: A command line tool that scrapes data of .fi domains from an open api provided by the The Finnish Communications Regulatory Authority. https://github.com/sarukuku/fi-tld-scraper. Accessed: 2018-03-21.

[30] curl - How To Use. https://curl.haxx.se/docs/manpage.html. Accessed: 2018-03-22.

[31] UI Events. https://www.w3.org/TR/DOM-Level-3-Events/#event-type-load. Accessed: 2018-03-22.

[32] RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231#section-5.5.3. Accessed: 2018-03-22.

[33] H76: Using meta refresh to create an instant client-side redirect — Techniques for WCAG 2.0. https://www.w3.org/TR/WCAG20-TECHS/H76.html. Accessed: 2018-03-31.

[34] Font Awesome. https://fontawesome.com/. Accessed: 2018-04-02.

[35] jQuery UI. https://jqueryui.com/. Accessed: 2018-04-02.

[36] Fancybox. http://fancybox.net/. Accessed: 2018-04-02.

[37] Search Results for "google fonts" — WordPress Plugins. https://wordpress.org/plugins/search/google+fonts/. Accessed: 2018-04-22.

[38] Module project — Drupal.org. https://www.drupal.org/project/project_module?text=google+fonts&op=Search. Accessed: 2018-04-22.

[39] Official Google Webmaster Central Blog: HTTPS as a ranking signal. https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html. Accessed: 2018-04-27.

[40] Category:Vulnerability Scanning Tools - OWASP. https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools. Accessed: 2018-04-27.

[41] Subresource Integrity. https://www.w3.org/TR/SRI/. Accessed: 2018-04-27.

[42] Content Security Policy Level 3. https://www.w3.org/TR/CSP3/. Accessed: 2018-04-27.

Appendix A

Attachments

A.1 Dependency tool output example (partial)


```
{
  "resources": {
    "document": {
      "requests": [
        {
          "url": "https:\/\/www.vero.fi\/henkiloasiakkaat\/",
          "method": "GET",
          "headers": {
            "upgrade-insecure-requests": "1",
            "user-agent": "Mozilla\/5.0 (iPad; CPU OS 9_1 like
                Mac OS X) AppleWebKit\/601.1.46 (KHTML, like
               Gecko) Version\/9.0 Mobile\/13B143 Safari\/
               601.1",
            "x-devtools-emulate-network-conditions-client-id":
                "B3FC45C669DD4C96F30F784176E7D58C",
            "accept": "text\/html,application\/xhtml+xml,
               application\/xml;q=0.9,image\/webp,image\/apng
               ,*\/*;q=0.8"
          },
          "crossOrigin": false
        }
      ],
      "totalCount": 1,
      "sameOriginCount": 1,
```

```
    "crossOriginCount": 0,
    "crossOriginPercentage": 0
  },
  "stylesheet": {
    "requests": [
      {
        "url": "https:\/\/www.vero.fi\/static\/bundle\/
          styles-983873274d.css",
        "method": "GET",
        "headers": {
          "user-agent": "Mozilla\/5.0 (iPad; CPU OS 9_1 like
              Mac OS X) AppleWebKit\/601.1.46 (KHTML, like
            Gecko) Version\/9.0 Mobile\/13B143 Safari\/
            601.1",
          "x-devtools-emulate-network-conditions-client-id":
              "B3FC45C669DD4C96F30F784176E7D58C",
          "accept": "text\/css,*\/*;q=0.1",
          "referer": "https:\/\/www.vero.fi\/
            henkiloasiakkaat\/"
        },
        "crossOrigin": false
      }
    ],
    "totalCount": 1,
    "sameOriginCount": 1,
    "crossOriginCount": 0,
    "crossOriginPercentage": 0
  },
  "image": {
    "requests": [
      {
        "url": "https:\/\/stat.vero.fi\/piwik.php?
          action_name=Henkil%C3%B6asiakkaat%20-%
          20Verohallinto&idsite=2&rec=1&r=255281&h=15&m=3&
          s=54&url=https%3A%2F%2Fwww.vero.fi%
          2Fhenkiloasiakkaat%2F&_id=acee47dc549594d4&_idts
          =1519995834&_idvc=1&_idn=0&_refts=0&_viewts=
          1519995834&send_image=1&cookie=1&res=1366x1024&
          gt_ms=203",
        "method": "GET",
        "headers": {
          "user-agent": "Mozilla\/5.0 (iPad; CPU OS 9_1 like
              Mac OS X) AppleWebKit\/601.1.46 (KHTML, like
            Gecko) Version\/9.0 Mobile\/13B143 Safari\/
            601.1",
```

```
          "x-devtools-emulate-network-conditions-client-id":
              "B3FC45C669DD4C96F30F784176E7D58C",
          "accept": "image\/webp,image\/apng,image\/*,*\/*;q
            =0.8",
          "referer": "https:\/\/www.vero.fi\/
            henkiloasiakkaat\/"
        },
        "crossOrigin": true,
        "whoisData": {
          "domain": "vero.fi",
          "registrantName": "Verohallinto",
          "registrantOrganization": "",
          "registrantCountry": "Finland"
        }
      }
    ],
    "totalCount": 9,
    "sameOriginCount": 8,
    "crossOriginCount": 1,
    "crossOriginPercentage": 11.11
  },
  "script": {
    "requests": [
      {
        "url": "https:\/\/stat.vero.fi\/piwik.js",
        "method": "GET",
        "headers": {
          "user-agent": "Mozilla\/5.0 (iPad; CPU OS 9_1 like
              Mac OS X) AppleWebKit\/601.1.46 (KHTML, like
            Gecko) Version\/9.0 Mobile\/13B143 Safari\/
            601.1",
          "x-devtools-emulate-network-conditions-client-id":
              "B3FC45C669DD4C96F30F784176E7D58C",
          "accept": "*\/*",
          "referer": "https:\/\/www.vero.fi\/
            henkiloasiakkaat\/"
        },
        "crossOrigin": true,
        "whoisData": {
          "domain": "vero.fi",
          "registrantName": "Verohallinto",
          "registrantOrganization": "",
          "registrantCountry": "Finland"
        }
      }
```

```
    ],
    "totalCount": 2,
    "sameOriginCount": 1,
    "crossOriginCount": 1,
    "crossOriginPercentage": 50
  },
  "xhr": {
    "requests": [
      {
        "url": "https:\/\/www.vero.fi\/api\/tweets\/",
        "method": "GET",
        "headers": {
          "accept": "application\/json, text\/plain, *\/*",
          "x-devtools-emulate-network-conditions-client-id":
              "B3FC45C669DD4C96F30F784176E7D58C",
          "user-agent": "Mozilla\/5.0 (iPad; CPU OS 9_1 like
              Mac OS X) AppleWebKit\/601.1.46 (KHTML, like
            Gecko) Version\/9.0 Mobile\/13B143 Safari\/
            601.1",
          "referer": "https:\/\/www.vero.fi\/
            henkiloasiakkaat\/"
        },
        "crossOrigin": false
      }
    ],
    "totalCount": 1,
    "sameOriginCount": 1,
    "crossOriginCount": 0,
    "crossOriginPercentage": 0
  }
 },
 "totalRequests": 14,
 "totalSameOriginRequests": 12,
 "totalCrossOriginRequests": 2,
 "crossOriginPercentage": 14.29
}
```