Nidan

A Security Search Engine for The World Wide Web

Master's Thesis - 10th Semester Software Engineering

ds103f19

Cassiopeia
Department of Computer Science
Aalborg University

Written in LaTeXfrom a template made by Jesper Kjær Nielsen.

Software Engineering
Aalborg University
https://www.cs.aau.dk/

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Nidan: A Security Search Engine for The World Wide Web

Theme:
Security in distributed systems

Project Period:
Spring Semester 2019

Project Group:
ds103f19

Participant(s):
Jesper Windelborg Nielsen
Mathias Jørgen Bjørnum Leding

Supervisor(s):
René Rydhof Hansen
Thomas Panum

Copies: 4

Page Numbers: 37

Date of Completion:
June 2, 2019

Abstract:

In this report, the development and usage of Nidan and KNAS are described. Nidan is a systematic webcrawler which collects all loaded JavaScript, cookies, and related metadata and stores it in a well-strutured relational database. KNAS is a data-processing tool that detects vulnerabilities connected to each visted website. These include vulnerabilities in the implemented JavaScript libraries, CMSs, and server software. Nidan and KNAS has been tested on around 2 % of the entire .dk zone file. This test showed that KNAS detected vulnerable software on 40.47 % of the websites. 92.49 % of the vulnerable websites have vulnerabilities from last year or older, meaning that the vast majority of vulnerable sites rarely update their software. From the data collected by Nidan, it is also possible to analyze the cookies. Since Nidan makes no interaction with the websites other than visiting, all tracking cookies sat break the GDPR and EU's cookie law.

Aalborg University, Januar, 2019

Jesper Windelborg Nielsen
<jwni14@student.aau.dk>

Mathias Jørgen Bjørnum Leding
<mledin14@student.aau.dk>

Contents

Preface

This report is a documentation of the project work of group ds103f19 from Aalborg University.

We, ds103f19, would like to thank our supervisor René Rydhof Hansen and assistant supervisor Thomas Panum for their supervision throughout the project period. This project has resulted in two programs — Nidan and KNAS — which will both be uploaded along with the project and can be seen on GitHub at jwindelborg/aau-security after 14-05-2019.

In this semester report, we use the Chrome DevTools Protocol (CDP) to run Chrome instances. For the purpose of this project, the use of the proprietary Chrome or the free version Chromium is of no importance and we refer to them interchangeably.

Chapter 1

Introduction

The internet is one of the most central parts of most people's lives. Nearly every Danish citizen today is online. With the internet comes a range of possibilities but also related threats. The amount of internet-related attacks towards individuals and organizations are on the rise[17][7]. These attacks can try to extort money from a victim, either from ransomware[13] or from utilizing leaked data such as user credentials from a vulnerable site[10]. One of the ways these attacks can be made possible is by exploiting vulnerabilities in a non-malicious website. For example, ransomware can be installed on a victim's computer if an attacker finds a severe Cross-Site Scipting (XSS) vulnerability that lets him inject malicious JavaScript. Another common way of exploiting users in cyberattacks is phishing which can often exploit XSS vulnerabilities to make the phishing attack more believable.

Similarly, internet users are being tracked more than ever before. Advertisement companies, analytic companies, and the rise of big data have made spying on users a lucrative business. The concern about users' privacy has lead to bills passed by the EU such as the EU directive 2002/58/EC also known as the EU Cookie Law and EU's General Data Protection Regulation (GDPR).

At the time of writing, the GDPR has been in effect for more than a year and the EU cookie law for even longer. This means that no website visited should track a user before their consent is given. It would be interesting to see whether any of the visited sites do not comply with the law in this regard.

It would be interesting to see how the situation is for visitors of Danish websites. How many websites are either vulnerable to an extend that puts the user at risk and how many are not complying with the laws protecting the users' privacy?

Another way a website can put its users at risk is by being vulnerable to server-side flaws; for example in their TLS/SSL server. A flaw, such as The Heartbleed Bug, could expose user credentials or even user credit card information[3].

Chapter 2

Preliminaries

In the previous semester project report[9], it was concluded that there is a need for a software solution to identify security related problems on the entire Danish web. In the report, the subject was narrowed down to focusing on detecting malicious websites and vulnerabilities in JavaScript on websites as both of these are common and can be tested without performing any non-consensual penetration tests on the website. From the research, it was also concluded that CDP is one of the only tools for running a Chrome instance headless. This lead to the choice of building Nidan around CDP. This leads into the following problem statement.

2.1 Problem statement

What are the key factors when detecting and identifying web-related security issues across all Danish websites?

To answer this problem, the following sub-problems are presented:

- How do we crawl every Danish website for JavaScript?
- How do we reliably, and legally, identify and verify security-related problems on Danish websites?
- How can the collected data be queryable with information about the security-related problems and their context?

Chapter 3

Design

To visit every .dk top-level domain (TLD), it is needed to know which domains exist. This information is commonly stored in what is known as a zone file. As of writing, the entire .dk zone file contains 905399 unique domains. Visiting this many websites, while saving relevant data, is a hard task and requires a complex program. The task about visiting each site needs to be handled in a way where multiple sites can be visited at the same time, with a distinction between the data collected from different sites. For a robust solution, it is also needed to make sure identical data is not stored multiple times. A JavaScript library might be used on hundreds or even thousands of sites. The system should be able to handle many instances writing and reading data at the same time and the system should be able to handle any sort of crash or error a website could produce. For this reason, a solid design is needed.

3.1 Architecture for Nidan

The main CDP manager is named Nidan. Its task is to visit every website in a table consisting of the desired domains and gather all needed data for later analytics. This process is the slowest but also most important process in the system. Since Nidan is meant to be used on large inputs, it needs to be fast. To make Nidan as fast as possible, it is designed to be parallelized both on multiple machines and with multiple running instances on each machine. To ensure multiple running instances do not work on the same domain at the same time, each instance adds the domains they are going to work on to a table of locked domains. This table has information about which instance performed the locking. When a set of domains is locked, the program instance starts a headless Chrome instance listening on a user-specified port. The next step is to process the domain queue one domain at a time. For each domain, it clears all browser data before navigating to the domain. It then saves all JavaScript loaded on the site, along with a reference to the site. It also saves all browser cookies. Simultaneously, it keeps Chrome running by a thread that stops and restarts Chrome every time it is not responding on its debugging port. After a domain is processed, it is added to a table of visited domains. Every time an instance is assigned domains, every domain previously locked by the instance is removed from the table of locked domains in favor of a new batch of domains. This way, the instance does not have to waste resources on removing every locked domain individually and it also ensures that locked domains, which were never visited due to a crashed instance, are released back to the pool of free domains.

Nidan is separated into several modules handling specific parts of the application's tasks. A main module is responsible for running the program. It starts the argument parser module and

decides what to turn on and run based on the user input. Since Nidan heavily relies on CDP, the main module also works as the Chrome handler and is responsible for starting Chrome and keep Chrome running throughout the scanning session. The argument parser module handles user input supplied as a command line argument when running the main application. A server module handles the server listening for information supplied by Privacy Badger. A database module provides access to the database. The separation of the code into smaller modules allows for parts to be easily swapped in favour of new parts. For example, changing the database engine would only require a new database module to be built and replace the old. Similarly, adding new modules is also trivial as keeping the code modular makes it possible to avoid side effects interfering with the existing functionality.

To make Nidan run stable and be resistant to all kinds of errors associated with the Chrome browser crashing or stalling or the database failing a transaction, the primary Nidan instance is developed to fail early. When the Nidan instance is made to fail early, it prevents instances from getting stuck on websites and it ensures that no work is performed if, for example, the connection to the database is lost. To counter the Nidan instances terminating on errors, Nidan is run through a runner application. The runner application starts the amount of desired Nidan instances and ensures that each instance has its own unique Chrome debugging port and a unique instance name. It also ensures that Nidan is always running the desired number of instances. If an instance crashes, a new instance is immediately created.

3.2 KNAS

To process and analyze the collected data, a tool named KNAS was created. KNAS combines all the tasks for the analytics and identification of systems, servers, and technologies, and performs these actions in a parallel, fast, and uniform fashion. KNAS has a separate repository file which maps identified strings that to technologies that normally use these. For example, if the string `drupal` is identified in a URL, there is a very big chance that the website is built upon Drupal as this is the name of a default Drupal folder. KNAS is also responsible for handling the parsers and for running the collected data through other tools.

KNAS uses a few different matching techniques depending on which type of data it is working on. One job is to analyze and tag all the JavaScript code by attempting to identify known libraries and potential vulnerabilities associated with these. This is done using the tool Retire.js which matches the scripts using a few different techniques as seen in Section 3.2.1.

Another job is to identify the purpose of cookies stored during the scan. These purposes could, according to the aforementioned EU laws, be illegal, for example tracking for advertisements or tracking in general, or legal purposes such as usability and authentication.

A cookie with a unique user ID as its value could be used both for authentication, tracking, or advertisement. Identifying the purpose of cookies can be challenging. However, some tools can help. One approach for determining a specific cookie is the website Cookiepedia which is a database of cookies, associated with their names and domain. The website also attempts to tag the cookies in regard to their purpose[16]. Another tool is Privacy Badger which is a browser extension by Electronic Frontier Foundation (EFF) that learns while browsing how cookies appear across domains and attempts to determine if the site and its cookies are tracking the user. Of the two approaches, it was chosen to focus on Privacy Badger rather than Cookiepedia since it is a breach of their terms of use and the fact that many cookies were not found on the site. It is also a huge advantage the way Privacy Badger gets its information. By training using the cookies present on all the input websites, there is no risk of cookies being unknown. Either

they are determined to be for tracking purposes or they are not.

Some other jobs could have yielded interesting results but are not feasible to perform for ethical or legal reasons. As an example, it would be trivial to automate the use of testssl.sh [18] and save results associated with the TLS/SSL certificate and server setup. However, such tests can hog resources on the server being scanned while also potentially being illegal due to it sending malicious TLS/SSL payloads to the server. Furthermore, sending such payloads leans towards being active penetration testing of a non-consenting website which is illegal.

3.2.1 The tool Retire.js

To identify known JavaScript libraries, a few tools exist. For example, the commercial tool Netsparker can fingerprint more than 20 of the most popular JavaScript frameworks and provide information regarding known vulnerabilities in these libraries[12]. Another example is the tool Retire.js which can also detect known JavaScript libraries and associated vulnerabilities with the added benefit of being Free and Open Source Software (FOSS), making it possible to tweak the output and detection mechanisms to be perfectly tailored to any desired purpose.

The tool Retire.js takes a JavaScript file as input and attempts to identify it as a library[15]. This is done with several different approaches. The first attempt is to look at the filename for any hints about the library. If this fails, it looks for specific comments that exist in specific versions of libraries. If these are not present, the MD5 checksum of the JavaScript file is compared to the checksum of each entry in the database of known library versions[14]. As a last resort, the JavaScript is executed in a Chrome sandbox to observe its behavior. If the library name and version are identified, Retire.js performs a lookup in its database to see if the library with the specific version has any known vulnerabilities. If any vulnerabilities are detected, the library version and its corresponding vulnerabilities are received as output[15]. In Nidan, Retire.js is only used with two of its identification methods. It was chosen to not use the filename in the identification process because of how easy it is to either on accident or purposefully having a misleading filename. It is also possible that people choose to update the content of a JavaScript file without updating the file name. It was also chosen not to include the dynamic analysis of the JavaScript since that would require Retire.js to be run in a Chrome instance which would significantly increase the resource load of the post-processing phase. All detected libraries and vulnerabilities are stored in their own tables as described in Section 3.4.

3.2.2 Identification of server software

When Nidan runs, it collects HTTP headers. Some HTTP headers contain clues or references to the software running on the server. As an example, if an HTTP request has the header `X-Aspnetmvc-Version`, it is a strong indicator that the website is powered by Microsoft's MVC framework. Similarly, many sites include a `Server` header containing information such as `Apache/2.4.6 OpenSSL/1.0.2k-fips` which indicates the web server is run using Apache 2.4.6 and that the TLS/SSL server is OpenSSL 1.0.2k. There is no guarantee that any of these values are correct the server decides what to write in any sent HTTP header. For example, in Node.js, the server could spoof its `X-Powered-By` by overwriting it as seen below.

```
response.setHeader('X-Powered-By', 'ASP.NET');
```

The above code example would indicate that the website is run using ASP.NET even though it would actually be run by Node.js. However, in many instances, people tend to use the default values sat by the server software which reports honest values. In the event of website using

modified headers, humorous values have been used such as `X-Powered-By: Nerds`. Such values are stored as a HTTP header by KNAS but since `Nerds` is not a known X-Powered-By value, the value is not used in the analysis of the website. This way, the data does not get tainted by the untrue values.

3.2.3 Identification of content management system (CMS)

To identify possible CMSs used, two techniques are utilized. The first technique is to tag it from the header. As described in Section 3.2.2, a website can include an HTTP header, such as `X-Aspnetmvc-Version`, which indicates the use of Microsoft's MVC framework. Similarly, Express.js sets its `X-Powered-By` header to `expressjs`. The second method is to look at folders used on the site. As an example, WordPress stores most of its content in a folder named `/wp-content/` out of the box. Therefore, KNAS searches through all the URLs of the collected JavaScript for each particular site. If any of these URLs references a folder associated with a specific CMS or framework, KNAS tags the website as using that CMS.

3.2.4 The tool cve-search

To link vulnerabilities to specific versions of server software, it was chosen to use the tool cve-search[2] which performs lookups in a locally stored copy of MITRE's CVE database. It is used in two places. The first is to acquire the CVEs associated with a specific version of a server software. The second is to acquire information about a specific CVE. This includes the description of the CVE and the Common Vulnerability Scoring System (CVSS) value. While the tool has a web API, it can not be used to acquire the desired data. Therefore, a new API was created.

3.2.5 Analysis of cookies

For the analysis of cookies, it was chosen to modify the tool Privacy Badger.

Privacy Badger is a browser extension from EFF that identifies and blocks websites from tracking users. It trains itself to learn which third parties are tracking by grouping websites into three categories; green, yellow, and red. Green websites have not yet been caught trying to track the user across different websites. Yellow websites track but completely banning them on third party sites would result in lost functionality. Therefore, only cookies are blocked. Websites in the red category are blocked all together[5].

By modifying the tool to, instead of blocking the domains from setting cookies, only track these domains and cookies and store the data in the database, it is possible to categorise the crawled cookies.

Normally with CDP, one would use a Chrome instance launched with the `--headless` flag. However running Chrome headless disables support for extensions[1]. Furthermore, there are also no supported way to extract data from a browser extension utilizing an API. A solution to this limitation is to run Chrome in a nonheadless instance. Running a nonheadless instance of Chrome on a headless server is not directly supported. However, using X virtual framebuffer, which is a part of the X Window System, makes it possible to run a non-headless application with a virtual framebuffer. With this workaround, it was possible to run Chrome in a headless environment but with support for extensions.

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=706008

Since extensions are not officially supported in a headless environment or as a part of CDP, it is not possible to directly communicate with extensions from outside. Therefore, to gather the data from Privacy Badger, a custom fork was built where `webrequest.js`, and `heuristicblocking.js` have been modified to send an HTTP request to a server running on localhost with information regarding the action that Privacy Badger would apply. The server is then run from within the core Go program, thereby providing contextual information towards which site is being visited. Since Privacy Badger is trained by the user itself rather than getting its data from a central repository, a training run is performed on all the domains before the real scan. A snippet illustrating the sending of Privacy Badger data is seen in the following code:

```
utils.xhrRequest("http://127.0.0.1:9000/TrackingCookie/" + origin,
↪  function(err, response) {
  if (err) { console.error('Problem calling listener'); }
});
```

The above code is executed in `heuristicblocking.js` in Privacy Badger if the function `hasCookieTracking()` returns `true`. To not interfere with the data collected, Privacy Badger is also modified to not alter any HTTP content during the scan. This means nothing is actually blocked. It also means that the Do Not Track (DNT) request is not sent to the server. This goes against EFF's normal intention of blocking trackers not honoring the DNT request. The reasoning behind choosing not to send DNT requests is that we want to know exactly how websites react when all the user does is connecting to their site. Also, because of EU's cookie law, tracking of EU citizens without their consent is not legal regardless of whether they send DNT requests or not. Therefore, by not sending DNT requests, more websites breaking the law can be identified. It is also worth noting that the vast majority of users on the internet do not send these requests which gives a more realistic image of the websites[6].

At the moment, the above code sends the affected domain as an input parameter on port 9000. Then, the listener is a server which has the context and knows the domain being visited when the action occurs. Since it only sends the affected domain, it can not run multiple instances at the same time.

3.3 Fingerprinting known systems and libraries

Websites are often built upon existing systems such as WordPress, Joomla, Microsoft MVC5, and other prebuilt solutions. These systems are often complex and contain numerous bugs across different versions.

Similarly, sites are often built using ready-made libraries for web development such as ExpressJS, Django, and similar.

To identify CMSs, a few strategies can be utilized. Some CMSs include their name in the `X-Powered-By` header. Others have folders only associated with a specific CMS. For example, having a folder named `wp-context` heavily indicates the use of the WordPress framework. By utilizing the collected data, such as HTTP headers and URLs for JavaScript libraries, many CMSs can be correctly identified.

When a site is identified to be a known system, such as WordPress, it is possible to fingerprint installed extensions, themes, and WordPress version and see if these are associated with any vulnerabilities.

3.3.1 WPScan

If a website has been identified as likely to be a WordPress site, further information can be gathered by using the tool WPScan. The WPScan tool can be used for free for any non commercial purposes and the source code is publicly accessible. WordPress is the most widely used CMS on the web and has a huge amount of plugins. The codebase for WordPress is large which, combined with the huge amount of plugins, means that WordPress has a high amount of vulnerabilities in outdated versions and in plugins and themes. WPScan uses different techniques to find the WordPress version along with installed plugins, themes and other related information. Included in WPScan is a database containing known vulnerabilities which is then reported. WPScan can run with three different detection modes: Mixed, passive and aggressive. The mixed and aggressive modes engage with the WordPress site to gather more information. Some options detect registered user accounts and can even attempt to brute-force login credentials. To avoid ethical and legal risks, KNAS uses WPScan in passive mode which opens the site and identifies versions, plugins, and themes using only passive data parsing.

3.4 Data storage strategy and architecture

The collected data is stored in a MySQL database, visualised in Appendix A, consisting of tables with the different important information from each visited website. These tables are linked with associative tables to create a many-to-many relationship between them.

3.4.1 Table for domains

The table `domains` stores all the domains to be crawled during the Nidan scan. The table consists of the domain of the website, a unique ID, and the time of when the domain was inserted into the table. The unique ID is used across all tables referring to a specific domain.

3.4.2 Table for crawl history

The table `cdp_visit_history` is used to keep track of which domains have already been visited by a specific worker at a specific time with a specific scan label. Scan labels are used to circumvent the need for separate databases for each scan. When a scan is initiated, it is given a scan label such that all the different workers working on the same scan are grouped together. This allows different scans performed at different times to be stored in the same database while still being distinguishable from each other, even though any one scan might be performed over a long period of time.

3.4.3 Table for JavaScript files

The table `javascripts` contains all the JavaScript collected from each domain. It has a unique hash such that the same JavaScript sample is not collected more than once. To link JavaScript files to where they were discovered, the table `domain_has_javascripts` links the JavaScript checksums to all domains where the JavaScript has been discovered. It also contains information about whether it was inline embedded on the page, or if it was fetched from a separate URL.

3.4.4 Table for libraries

All detected libraries used on the different websites are stored in `libraries`. Each row consists of the name of the library, the version number, a unique ID which is a checksum of the two, and a `created_at` column that indicates when the library was added to the database. The reason for using an ID is to prevent the same versions of the same libraries to be stored more than once. The table `libraries` links to `javascripts` with the table `javascript_is_library` which consists of the JavaScript checksum, the library ID and the time of which the relationship was discovered.

3.4.5 Table for vulnerabilities

In Nidan, there are three different categories of vulnerabilities. The first type of vulnerabilities is client-side vulnerabilities in the form of JavaScript. These vulnerabilities are often in a front-end framework such as jQuery or Twitter Bootstrap. The second type of vulnerabilities is the ones found in server software such as Apache or OpenSSL. The last type of vulnerabilities are the ones detected in the CMS used by the website. The most common CMSs are WordPress, Joomla, and Drupal[11].

The need for the distinction between the three kinds of vulnerabilities lies in the nature of where they exist, how severe they can be, and how they can be verified. Vulnerabilities in the front-end are often more a threat to the end-user than to the site owner itself. Similarly, vulnerabilities in the front-end can often be tested and verified without legal risk. On the contrary, vulnerabilities in server software can lead to Denial of Service (DoS), compromise of the entire application or system, and are in general hard to verify without legal problems. Vulnerabilities in the framework is a mixture between the two since they can be either client-side or server-side. These can lead to compromise of the entire framework and, in extreme cases, the entire server. In other instances, the vulnerability is only limited to an XSS vulnerability. Another reason why it is required to store the vulnerabilities in separate tables is the fact that they each have their own way of measuring severity. The front-end vulnerabilities are added with Retire.js which scores the vulnerabilities from 0-4. The server software vulnerabilities are added with the tool cve-search which searches through Mitre's CVE database and uses the official CVE severity scale of 0.0-10.0. Lastly, the CMS vulnerabilities are added via WPScan which does not add a severity score. While it would be trivial to simply look up the CVE for the vulnerabilities not in the server software vulnerability table and grab the CVSS value, around half of all vulnerabilities do not have a CVE which would require them to acquire CVSS values by other means. Technically, it would be possible to convert Retire.js's five step scoring system to CVE's 101 step scoring system. However, that would cause severe inaccuracies. Furthermore, even this would not be possible with the CMS vulnerabilities, as the ones without CVEs give no indication of their severity.

All detected vulnerabilities in each visited website are stored in `javascript_vulnerabilities`, `cms_vulnerabilities`, or `server_vulnerabilities`. Each vulnerability has an entry for the vulnerability itself, its severity, unless it is a CMS vulnerability, and an ID. The ID is either its CVE name or a checksum of the two other entries if it does not have a CVE name. This is to avoid duplicate vulnerabilities in the database. To link vulnerabilities to JavaScript, libraries, or domains, three junction tables exist providing a many-to-many relationship between vulnerabilities and vulnerable entities.

3.4.6 Table for cookies

All the collected cookies that are automatically set when visiting each site are stored in the table `cookies`. The table consists of the domain ID of the site it was detected on, its name, its value, the domain of the creator of the cookie, its expiration time as a Unix timestamp, and flags that determine whether the cookie is secure and whether it is HTTP only. These two flags can be used to determine the security of the cookies. The `is_secure` flag means that the cookie can only be sent over an HTTPS connection. Since HTTPS requests, unlike HTTP requests, are not sent in plaintext, it removes the possibility for an attacker to eavesdrop on a connection between a user and the server, hijacking the user's cookie, and stealing their identity this way. However, another way of stealing cookies is to exploit a XSS vulnerability to steal the authentication cookie. This is where the flag `http_only` is useful as it removes the possibility of using JavaScript to acquire the authentication cookie.

3.4.7 Table for Privacy Badger data

For every website visited by Nidan, a modified Privacy Badger is run. It logs every action that Privacy Badger would have performed. These actions can be to block cookies from a third party or to block all resources from a third party. These actions is denoted by Privacy Badger using a label of green, yellow and red, where green is no action, and therefore not stored, yellow is blocking cookies but not all content, and red is when all content from a third party is blocked. The table consists of the visited domain, the action, the creator of the cookie, and information about time and scan label.

3.4.8 Table for HTTP headers

During the scan, HTTP headers are collected, which can contain information about things like HTTP Strict Transport Security (HSTS) and server software. HTTP headers give a good suggestion on what is installed and what is used to run the site. However, the headers can not always be trusted in the data samples. Headers, such as `X-Powered-By: Crazy Poulz On Caffeine, Monster And Pizza`, have been received during the scans. On the other hand, some HTTP headers come with some interesting warnings such as `PHP/5.5.37-1+deprecated+dontuse+deb.sury.org~precise+1`, indicating that an old, deprecated, and vulnerable version of PHP is installed.

3.5 Design choices

When designing the software and the database, it is important to make the correct design choices. In this section, the different possibilities are explored and choices are made based on this exploration.

3.5.1 System architecture

It was chosen to split the system up in two main programs: Nidan and KNAS. The reason behind this is that data collection and analytics are very different areas, each with different best suited tools and languages. For example, the language Go is currently the de facto solution for running CDP which is used to crawl the websites. Python, on the other hand, is the number one language for data science in terms of popularity[8]. Furthermore, there exists countless

FOSS tools that can be modified to the desired purposes. Having separate programs allows for both being used independently of each other. It is also worth noting that Nidan is designed to be executed in many instances at the same time, while this is not, in its current state, fully possible with KNAS.

3.5.2 Data storage technique/engine

There is a wide variety of ways the collected data can be stored. A way to store the data could be as objects in a document-based database such as MongoDB. document-based databases is good at handling semi-structured data. A more traditional option is to use a relational database such as MySQL or MariaDB. Relational databases is good at handling relationships between data with concepts such as foreign keys. Relational databases is fast at storing data with a clear defined data-structure and makes it accessible based on structured queries. The data collected is well structured; a cookie always has the same fields with properties and a domain always has a set of JavaScripts loaded. At the same time, the data has some very important relationships. For instance, multiple sites can have the exact same JavaScript in common and multiple JavaScripts can share specific vulnerabilities. Such relationships provide important information. For this reason, it was chosen to use a relational database.

3.5.3 Historical changes

When visiting the same page multiple times, it is likely that some things change between each visit. For example, different advertisements can be loaded each time. Therefore, it is necessary to consider how to handle updated data; is it preferable to simply replace the old data with the data from a new scan or to keep a snapshot of web sites as they were at a certain time? Keeping each scan as a snapshot that provides historical data can provide information as to how frequently web sites update their JavaScript libraries. It can also be used to show time-based statistics, for example, whether web security improves or diminishes. On the other hand, storing data as a snapshot requires more storage and a more complex database structure. Similarly, it might leave data in a more fragmented and inconsistent state. This is because the scripts may be changed and improved over time, or the amount of domains being processed might change, altering which and how much data is collected. For example, the first scans did not include Privacy Badger like the later scan do. It is also possible that the different scans were performed with different scan options. For example, there are options to not save any data and Privacy Badger can be turn off. This can potentially hinder historical comparisons. It was chosen to keep historical snapshots by using a combination between timestamps and scan labels. Each scan requires a label which groups all data from that specific scan together. The timestamps can be used to compare two scans in terms of historical changes.

3.5.4 Choice of checksum

While storing JavaScript for every domain, there was discovered a need for a unique ID which can be generated on the fly. Many JavaScript files have their own URLs which can be used as a unique identifier. However, embedded JavaScript does not provide such a unique and simple ID. Similarly, the exact same JavaScript file can be used on multiple different sites with different URLs. Therefore, it was decided to use a checksum of the JavaScript for each unique script while saving the relationship between the script and the site using it together with the URL from which the site loads the particular script.

It was chosen to use SHA-256 which is a heavy but cryptographically secure hashing algorithm. The reason for this is that any collision between two different JavaScript scripts would invalidate some of the data since the database would be completely unable to tell these scripts apart and only save one of these scripts. Therefore, avoiding any collisions between checksums is preferred. The difference in performance between a heavy and a light hashing algorithm is not going to make any substantial execution time since, in this context, the biggest bottleneck is waiting for websites to respond to requests or to perform all required database actions.

Chapter 4

Results

As a part of testing Nidan, a sample of around 2 % of the entire .dk zone file was crawled and analysed. In this chapter, some of the extremes are explored such as sites with most cookies or highest number of vulnerabilities. The goal is to provide insight into what Nidan has to offer.

4.1 Deep dive into the test crawl

After the crawl and analysis on the sample size of 18129 Danish TLDs, it was chosen to verify the accuracy of the analysis. The scanning and analysis was performed on the 27th of March 2019.

When verifying the results, there are three focal points. The first is evaluating whether the detection of the vulnerability is a true or false positive. The second is whether the vulnerability is exploitable. The last is whether there are any false negatives in the automatic scan. This is done through a manual audit of the sites.

4.1.1 Railgun.dk

Railgun.dk sets the highest amount of cookies of all websites in the test query. Querying Railgun.dk results in 306 cookies being set in ten seconds without any user input. The cookies are put into four categories; tracking, advertisement, self-set, and Cloudflare protection. The tracking category is every cookie that is used to log some sort of data from the user and use this data for analytics, whether that is for targeted advertisements or otherwise. The advertisement category covers cookies placed for the purpose of showing advertisements. Self-set refers to cookies created by the website itself. Lastly, Cloudflare protection cookies are used to determine whether the user is a real person or a bot. This is to, among others, protect the website against DDoS attacks. Among the tracking cookies are several alarming cookies described as "This cookie collects unidentifiable data; this data is sent to an unknown receiver. The identity of this receiver is kept classified by Perfect Privacy LLC". The categorization of all the cookies can be found in Appendix B. From an analysis of all 306 cookies, it shows that cookies are set from 102 different companies. Of these companies, 74 were identified as tracking the user. Most of these track the behaviour and preferences of the user depending on which articles and advertisements they click on. Others were used to track the user across different websites and track their location and IP address.

Library	Vulnerability	Exploitable
AngularJS v. 1.4.4	usemap can avoid sanitizing	no, functionality not used
	CSP bypass via add-on in Firefox	no, only affects Firefox extensions
	DoS in $sanitize if malicious HTML	no, only if a bad actor can supply HTML input
	XSS in $sanitize in Safari/Firefox	no, functionality not used.
Bootstrap v. 3.3.7	CVE-2019-8331	no, functionality not used.
	CVE-2018-14041	no, functionality not used.
	CVE-2018-14040	no, functionality not used.
	CVE-2018-14042	no, functionality not used.

4.1.2 etiskraad.dk

Etiskraad.dk is on a shared second place in terms of highest amount of unique vulnerabilities. Etiskraad.dk uses an outdated version of AngularJS, Twitter Bootstrap, and jQuery which presents a number of vulnerabilities.

The vulnerabilities detected, based on identified libraries, are, however, not necessarily exploitable since the exploitability depends on whether the specific library functionality is in use. The following table shows the identified vulnerabilities and an estimate of whether they are exploitable or not.

As seen in the table, the particular detected vulnerabilities are not be exploitable in etiskraad.dk. The reason behind this is that KNAS does not only detect vulnerabilities in used functionality but in the implemented libraries as a whole. It is still important to look at since the developers of the site could choose to use the vulnerable functionality in the future. Furthermore, the fact that a vulnerable library is used is an indicator of the site not regularly being kept up-to-date which significantly increases the risk of other vulnerabilities being present.

The automatic process of discovering potential vulnerabilities is not guaranteed to discover every potential vulnerability. For this reason, to determine if any obvious easily discoverable vulnerabilities are left out, a light manual audit of the website is performed.

In the manual audit, it was discovered that the site uses a CMS called 'Sitecore.NET 8.0 (rev. 150427)'. This version of Sitecore has, among others, the following vulnerabilities.

- In remote provider configuration, Web API can be called without authentication.
- JavaScript or HTML may be imported via csv, representing a security risk. Csv files should only contain text contact fields (25209).
- SC2017-001-170504
- Vulnerabilities in Sitecore's use of PowerShell
- Vulnerabilities enabling code and command injections

Since these vulnerabilities are all server-side vulnerabilities, it is not possible to verify whether they are present on etiskraad.dk because that would require an active scan.

4.2 Deep dive into Nidan scan 1

After leaving the testing phase of Nidan, a new mass scan was performed on the 26th of April 2019. The same 18129 were scanned and analysed with the improved version of Nidan that included additional post-processing tools. From the results of the new scan, it was evident that

the same most vulnerable sites from the first scan were still the most vulnerable sites in the new scan.

4.2.1 Highest amount of set cookies

In the time period between the test scan and this scan, Railgun.dk has implemented a cookie consent dialog box and now only sets three cookies without the user's consent. Furthermore, if the user declines the box, no ads load on the site. However, a new type of websites now dominates the list of most set cookies. The first nine websites, in addition to a lot of websites further down the upper end of the list, are all blogs from Bloggers Delight. Bloggers Delight is a WordPress-based hosting platform for blogging that offers a WYSIWYG blog creator. The website with the highest amount of cookies is the blog lifebymette.dk with 231 cookies set within 10 seconds of visiting the site without any user input nor consent. The vast majority of the cookie values are unique IDs which are clear indications of tracking across sites. Most of the cookies are from the same companies as the ones found in the analysis of Railgun.dk, meaning that several different advertisement companies set cookies to track which advertisements the user clicks on to remember their preferences across websites. Most of the remaining cookies are set by the companies from the advertisements themselves.

4.2.2 The state of the Danish web

From the collected data, queries have been performed to determine the state of the Danish web. Some of the results of these queries are shown in the table below.

Name	Value
Total amount of domains	18129
Active domains	14085 (77.69 %)
Sites with vulnerabilities	5700 (40.47 %)
Sites with identifiable CVEs	5527
Sites with old vulnerabilities	5112 (92.49 %)
Amount of vulnerabilities with CVEs	16540
Amount of old vulnerabilities	10340 (62.52 %)
Sites without SSL	11567 (63.80 %)

From the queries, it shows that 63.80 % of all the websites do not have any form of TLS/SSL. It is also evident that 40.47 % of the active websites had some kind of vulnerability, whether that is the usage of outdated server-side frameworks or outdated client-side libraries. Another interesting discovery is that 82.38 % of the vulnerable websites have vulnerabilities from last year or older which indicates that the majority of the vulnerable websites have not updated their used libraries and frameworks this year. In addition to this, 62.52 % of the detected vulnerabilities are also from last year or older[1].

[1]These calculation only includes vulnerabilities with an identifiable CVE, since the age of a vulnerability is determined from the CVE name.

Chapter 5

Evaluation

Now that the results of Nidan and KNAS have been discussed, the different aspects of the tools themselves need to be evaluated.

5.1 The module tool Retire.js

From using Retire.js, it shows that, while a lot of vulnerabilities are correctly identified, Retire.js still has some flaws. First of all, the possibility of a detected vulnerability being a false positive is very high. For example, a lot of websites include the library jQuery. This library contains thousands of functionalities. If just one of these has a vulnerability, the library with the version number is flagged as vulnerable which means that the website is flagged as vulnerable even though the chance of the site using that specific functionality is small. As previously argued, the results from Retire.js still provide a strong indication of how up-to-date the website is and how likely it is that other vulnerabilities are present.

Another problem with Retire.js is the fact that its database of vulnerabilities is rather smaller compared to all known web vulnerabilities since vulnerabilities are manually added by contributors. This could potentially lead to some false negatives if the correct vulnerabilities are not added to the database yet. However, Retire.js has an active community so the risk is fairly low. Furthermore, the fact that the list is small compared to all know web vulnerabilities is due to the fact that it only contains vulnerabilities relevant to JavaScript.

5.2 The post-processing tool KNAS

When data is gathered, KNAS has been used to run different modules. As described in Section 5.1, it ran a Retire.js job on all the collected JavaScript discovering 58 unique vulnerabilities across 215 unique vulnerable libraries affecting 5700 domains from the sample of 18129 Danish domains. KNAS similarly performs a scan on the HTTP headers finding information such as `X-Powered-By` and `Server`. It then uses a heuristic technique to identify known CMSs such as WordPress and Drupal. When well-known CMSs are identified, these systems can be passively scanned for further clues about versions and vulnerabilities. KNAS currently supports doing this for WordPress leading to the identification of 585 unique WordPress vulnerabilities affecting 1084 sites with a total of 7892 detected WordPress vulnerabilities.

5.3 SSL certificate findings

To audit the state of TLS/SSL, it is interesting to look at the `Server` field in an HTTP header. This field can bring information about the installed SSL software. As an example, a search revealed that three sites report the usage of the software OpenSSL with the version 1.0.1f which is vulnerable to the Heartbleed Bug.

The header is acquired through Go's `net/http` library. All detected server software is saved for each website in the table `server_software`, together with a hash for the software configuration. Then, the tool cve-search is used to perform lookups in Mitre's CVE database for each detected server software which returns all known vulnerabilities for the server software at a specific version. These are stored in the table `server_vulnerabilities`.

From the results in Section 4.2.2, it was revealed that 63.80 % of the websites do not use any form of TLS/SSL certificates. This is a worrying discovery since it can be catastrophic if the websites handle any sensitive data such as usernames and passwords. This is due to the fact that the communication between the user and the server is not encrypted, allowing for third parties to intercept the data. It can also be a problem without website handling any sensitive data since it allows third parties to inject code into the communication between user and website if a successful Man-in-the-middle (MITM) attack is performed.

5.4 Privacy Badger

Since the tool Privacy Badger is not built for academic research, but to block and learn as the user browses, it presents some important challenges that need to be addressed. In academic research, reproducibility is important. However, since Privacy Badger uses machine learning and learns as it sees more websites, it is inconsistent in nature. Since it updates its data by detecting sites not honoring the DNT request at all times, sites, at different stages in the scan or on different machines, might yield different results. A way around this is to pre-train Privacy Badger in a scan where every site is visited once, but no real scan data is saved. After the pre-train scan is finished, the data can be exported and used as the default state before every page visit preventing training from occurring during the real scan. A similar issue is how the web is constantly changing; two different visits to the same site might result in completely different results. A site might dynamically decide which advertisement platform to load based on market prices or collected user data. Furthermore, some advertisements themselves set cookies, meaning the same advertisement platforms can also give different results.

Another challenge associated with using Privacy Badger is how the tool is built to block either cookies or entire requests. This would naturally alter the collected results and remove information otherwise saved. To solve this challenge, a fork of Privacy Badger replacing all request-altering actions with a logging feature was necessary and missing any offending functionality could taint the results, not only for the Privacy Badger part of the data, but for all the data gathered in a scan session.

In the Nidan fork of Privacy Badger, `webrequest.js` and `heuristicblocking.js` have been altered, removing all actions interfering with the scan. At the same time, these actions are sent to a HTTP listener describing the action and complementary information and stores the data in the database. This solution, combined with pre-training, heavily improves the non-reproducible nature of Privacy Badger.

5.5 Why Nidan?

There already exists some tools allowing for the search of or analysis of some of the things searchable by Nidan. Shodan makes it possible to search for the Internet of Things (IoT) based on banner grabbing, and also attaches some related information such as vulnerabilities. Cookiepedia makes it possible to search for websites and see which cookies are stored from visiting that side, and to compare this result with other sites and see an attempted categorization of the cookies. However, many of the detected cookies are not found in Cookiepedia. Similarly, Libscore allows users to search for the usage of some JavaScript libraries and their popularity based on Alexa's top 1 million[1]. However, its only use is to scan Alexa's top 1 million sites rather than any desired set of domains like Nidan and KNAS can. Furthermore, it is only used to rank libraries after popularity.

Nidan collects and indexes all JavaScript files found from a list of domains and stores a lot of different information about these files such as all known present vulnerabilities. Nidan also stores all cookies and detects their purpose, whether that is for tracking the user across websites or simply storing the user's preferred language settings. All this data is stored in a way that allows for easy analytics in the form of SQL queries. Shodan offers some of the same services, such as scanning and providing which known vulnerabilities are present in a queryable fashion. However, Shodan only gains its information from the banner of the IP and does not consider other things like JavaScript vulnerabilities or cookies since Shodan focuses on IoT rather than websites.

A unique feature of Nidan is also the possibility of searching all websites based on very specific JavaScript features. For example, to the best of our knowledge, no other tools make it possible to search and find every site which includes a piece of JavaScript or cookie having specific keywords. For example, if the website includes a JavaScript file with the word "adblock", it indicates that the websites most likely uses an anti-adblock feature. If a JavaScript file includes the string "miner.start();", it means that the site most likely has a crytominer. This specific string is found on, among others, CoinHive-using sites such as The Pirate Bay. A positive effect of this is that the discovery of new JavaScript exploits or functionality being used in the wild is immediately queryable. Nidan hereby makes it possible to find the usage of cryptomining libraries not tagged as a library in Libscore.

Chapter 6

Conclusion

To crawl every .dk TLD, the tool Nidan was created which is built upon CDP. While Nidan was not tested on the entire .dk zone file, it is still proven that this is possible within a reasonable time frame since Nidan is easily scalable. With 40 instances running at the same time, which is what was used during the Nidan scan 1, roughly 15000 domains are visited an hour. This means that the entire zone file would be finished in 2.5 days. With even more instances, this crawl duration can be reduced even further.

To identify the security-related problems on every crawled website, the post-processing tool KNAS was created. KNAS was created to identify vulnerabilities in the data from Nidan. The module that identifies vulnerabilities in the implemented JavaScript files, Retire.js, accurately identifies whether the particular site uses vulnerable libraries. This does not necessarily mean that the site is vulnerable as this does not guarantee that the site uses the vulnerable functionality from the library. This, however, is still dangerous as the vulnerable functionality could be used in the future. It also means that the website is not regularly updated, giving it a higher risk of having other vulnerabilities now and in the future.

By storing all the data in a relational database in clear defined structure, it is easy to perform queries for analytical purposes. This enables answering questions related to the state of the Danish websites. Such questions could answer how compliant with GDPR Danish websites is as a whole and similar interesting inquiries.

The research demonstrates that finding security related issues at a large scale is possible. Fortunately, the research also suggests that many easily identified vulnerabilities are harder to exploit than to find. The use of large JavaScript frameworks and common CMSs make one security flaw in a single system affect thousands of instances across the web if the websites are not regularly updated. From the results, it is clear that the vast majority of the vulnerable sites have not updated their implemented JavaScript in the current year. It was estimated that 92.49 % of all vulnerable websites has vulnerabilities from 2018 or older. This means that the vast majority of vulnerable sites are heavily outdated. To add to this, it was estimated that 62.52 % of all detected vulnerabilities are from 2018 or older.

6.1 Future Work

While the Nidan/KNAS system is successful at gathering security related data and analyze these, there is still room for some improvement. These improvements include more data, better accuracy, and faster data collection and analytics.

6.1.1 KNAS

The tool KNAS is responsible the data processing happening in the Nidan/KNAS system. For that reason, KNAS needs to be as fast as possible. KNAS has been constructed to run in parallel. However, it currently only runs the different jobs in parallel; the jobs themselves are not parallelized. The way KNAS is built makes it simple to enable threading on the jobs. However, a more high-level thread handler needs to be constructed.

For future work, it would be beneficial to make every job capable of running threads, thereby speeding KNAS up massively for larger amounts of domains. For a small number of domains, such as 18129, it generally takes little enough time to perform every job that the benefits of more threading are not that important. If Nidan was to index all .dk TLDs, it would have a much greater effect.

6.1.2 Privacy Badger

To receive data from Privacy Badger, the tool is forked and communicates through GET requests to a web API on localhost. A limitation of this is that the Privacy Badger extension is a static fork which means it will always send the data to one specific predefined port. It has not currently been possible to get context about which site is visited through Privacy Badger. Instead, Nidan listens on a local port and connects the data from Privacy Badger together with the site it is currently visiting. This limits the number of Nidan instances that could be run on the same machine if Privacy Badger data is to be collected because more than one instance of Privacy Badger on the same machine would cause Nidan to combine the data from both instances into one domain.

To solve this, a future version of Privacy Badger should either be able to tell which site it is currently visiting, and therefore only need one server for each machine running Nidan, or Privacy Badger should be built for each Chrome instance running, allowing each instance to communicate with different ports.

A necessary improvement of the Privacy Badger module is to have the data being reproducible by having a static set of training data which remains the same on every page visited. This ensures the same treatment of third party trackers. This functionality, however, is currently not implemented.

6.1.3 Retire.js

One of the jobs enabled by KNAS is to run every collected script through Retire.js. It currently only matches JavaScript through the content of the JavaScript file using a checksum or a search for library names and versions. However, Retire.js is built to also check for clues in the filename and to, if run in Chrome, try the script in a sandbox. These features are not utilized in KNAS leading to a potential limitation in the amount of identified libraries.

For a future version of KNAS, Retire.js should be better utilized. Retire.js should also use the filename in its identification process but in some way tell how certain it is in the identification. For example, if only the filename has been used to identify, the identification should be marked as "not certain". The filenames are already stored in Nidan's database allowing for such analytics.

The other unused identification method, having the JavaScript file executed and observed in a Chrome instance, is, on the other hand, a more difficult process. There are, as described in Section 3.2.5, no official supported ways for Chrome to be run headless with extensions. Since Retire.js is also FOSS, the same technique that we already used for Privacy Badger could be

applied here. A limitation of this approach is the increasing cost of running a Chrome instance to perform KNAS jobs with comparably minimal gain.

6.1.4 Nidan

Nidan has the longest and most important job. It is able to run in parallel both on multiple machines working together and further as multiple parallel instances on the same machine. Unfortunately, Chrome has a tendency to sometimes hang and never respond again until it has been killed and allowed to restart. This provides great gaps of time where Nidan is just in a wait state, never continuing without user interaction, which leads to serious delays in the scan. This difficulty is attempted mitigated by looking for signs of Chrome's responsiveness and killing the Chrome thread if it is not responding. Another problem with Nidan is that it seems to not always close database connections in an orderly fashion, leading to the process sometimes being locked out due to too many continuous connections. This would have to be fixed in a future version.

6.1.5 Test sample

In the future, it is necessary acquire the entire .dk zone file and perform the scans on this as this would give a more precise depiction of the state of all Danish websites. While the sample was a random subset of the entire zone file, there is no way of knowing whether the sample is an accurate representation of the entrie zone file.

6.1.6 Detection of malware

Like the detection of vulnerabilities, the implementation of a modified version of an existing tool can be used to identify implemented malicious JavaScript on each website. For this purpose, the most used tool is the closed-souce, proprietary tool Zozzle from Microsoft. However, lesser known free tools exist. These are unfortunately old and outdated and would require heavy modifications to be useful. A promising tool, JaSt[4], is an unfinished and a bit outdated JavaScript malware detector which, with heavy modifications, would fit nicely into KNAS's modular design.

Bibliography

[1] Alexa. file: alexa top 1 million. https://s3.amazonaws.com/alexa-static/top-1m.csv.zip, 2019. Accessed: 2019-05-28.

[2] Alexandre Dulaunoy. cve-search. https://cve-search.github.io/cve-search/, 2018. Accessed: 2019-06-02.

[3] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[4] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[5] Electronic Frontier Foundation. Privacy badger. https://www.eff.org/privacybadger/faq, 2018. Accessed: 2019-04-10.

[6] Alex Fowler. Mozilla's new do not track dashboard: Firefox users continue to seek out and enable dnt. https://blog.mozilla.org/netpolicy/2013/05/03/mozillas-new-do-not-track-dashboard-firefox-users-continue-to-seek-out-and-enable-dnt/, 2013. Accessed: 2019-06-02.

[7] H.L.M. Gambrell and C.S. Horbus. Cybersecurity alert cyberattacks on the rise - no. 1 challenge facing businesses today. https://www.lexology.com/library/detail.aspx?g=c6cd77e0-f2bb-45c7-b177-384f53b164, 2019. Accessed: 2019-05-28.

[8] Bob Hayes. Programming languages most used and recommended by data scientists. https://businessoverbroadway.com/2019/01/13/programming-languages-most-used-and-recommended-by-data-scientists/, 2019. Accessed: 2019-06-02.

[9] Mathias Leding and Jesper Windelborg. Web application security. https://projekter.aau.dk/projekter/da/studentthesis/web-application-security(b6fbf35e-c3a9-4408-b64d-76c9c13081b3).html, 2019. Requires login. Accessed: 2019-04-10.

[10] Kif Leswing. Email scam uses old passwords and fake threats about your porn habits to collect bitcoin. https://www.businessinsider.com/new-email-scam-uses-old-password-fake-porn-threats-webcam-video-bitcoin-2018-7, 2018. Accessed: 2019-04-10.

[11] BuiltWith® Pty Ltd. Cms usage distribution in denmark. https://trends.builtwith.com/cms/country/Denmark, 2019. Accessed: 2019-04-18.

[12] Netsparker Ltd. Fingerprinting libraries. https://www.netsparker.com/support/fingerprinting-libraries/, 2019. Accessed: 2019-04-18.

[13] Alfred Ng. Worldwide ransomware hack hits hospitals, phone companies. https://www.cnet.com/news/england-hospitals-hit-by-ransomware-attack-in-widespread-hack/, 2017. Accessed: 2019-04-10.

[14] Erlend Oftedal. retire.js/blogpost.md at master · retirejs/retire.js. https://github.com/RetireJS/retire.js/blob/master/docs/blogpost.md#detection, 2015. Accessed: 2019-04-10.

[15] Erlend Oftedal. Retire.js. https://retirejs.github.io/retire.js/, 2019. Accessed: 2019-04-10.

[16] Cookiepedia OneTrust. Cookiepedia. https://cookiepedia.co.uk/, 2019. Accessed: 2019-04-18.

[17] Rob Sobers. 60 must-know cybersecurity statistics for 2019. https://www.varonis.com/blog/cybersecurity-statistics, 2019. Accessed: 2019-05-28.

[18] Dirk Wetter. Testing tls/ssl encryption. https://testssl.sh/, 2019. Accessed: 2019-04-18.

Acronyms

API
Application Programming Interface. 10, 26

CDP
Chrome DevTools Protocol. vii, 5, 7, 10, 14, 25

CMS
content management system. iii, 10, 11, 13, 18, 21, 25

CVE
Common Vulnerabilities and Exposures. 10, 13, 19, 22

CVSS
Common Vulnerability Scoring System. 10, 13

DDoS
Distributed Denial of Service. 17

DNT
Do Not Track. 11, 22

DoS
Denial of Service. 13

EFF
Electronic Frontier Foundation. 8, 10, 11

EU
European Union. iii, 3, 8, 11

FOSS
Free and Open Source Software. 9, 14, 26

GDPR
General Data Protection Regulation. iii, 3, 25

HSTS
HTTP Strict Transport Security. 14

HTTP
Hypertext Transfer Protocol. 9, 10, 11, 13, 14, 21, 22

HTTPS
Hypertext Transfer Protocol Secure. 13

IoT
Internet of Things. 22, 23

IP
Internet Protocol. 17, 23

KNAS
KNAS's Not A Script!. iii, v, vii, 8, 9, 10, 11, 14, 18, 21, 22, 25, 26, 27

MITM
Man-in-the-middle. 22

PHP
PHP: Hypertext Preprocessor. 14

SHA
Secure Hash Algorithm. 15

SQL
Structured Query Language. 23

SSL
Secure Sockets Layer. 3, 9, 19, 21, 22

TLD
top-level domain. 7, 17, 25, 26

TLS
Transport Layer Security. 3, 9, 19, 21, 22

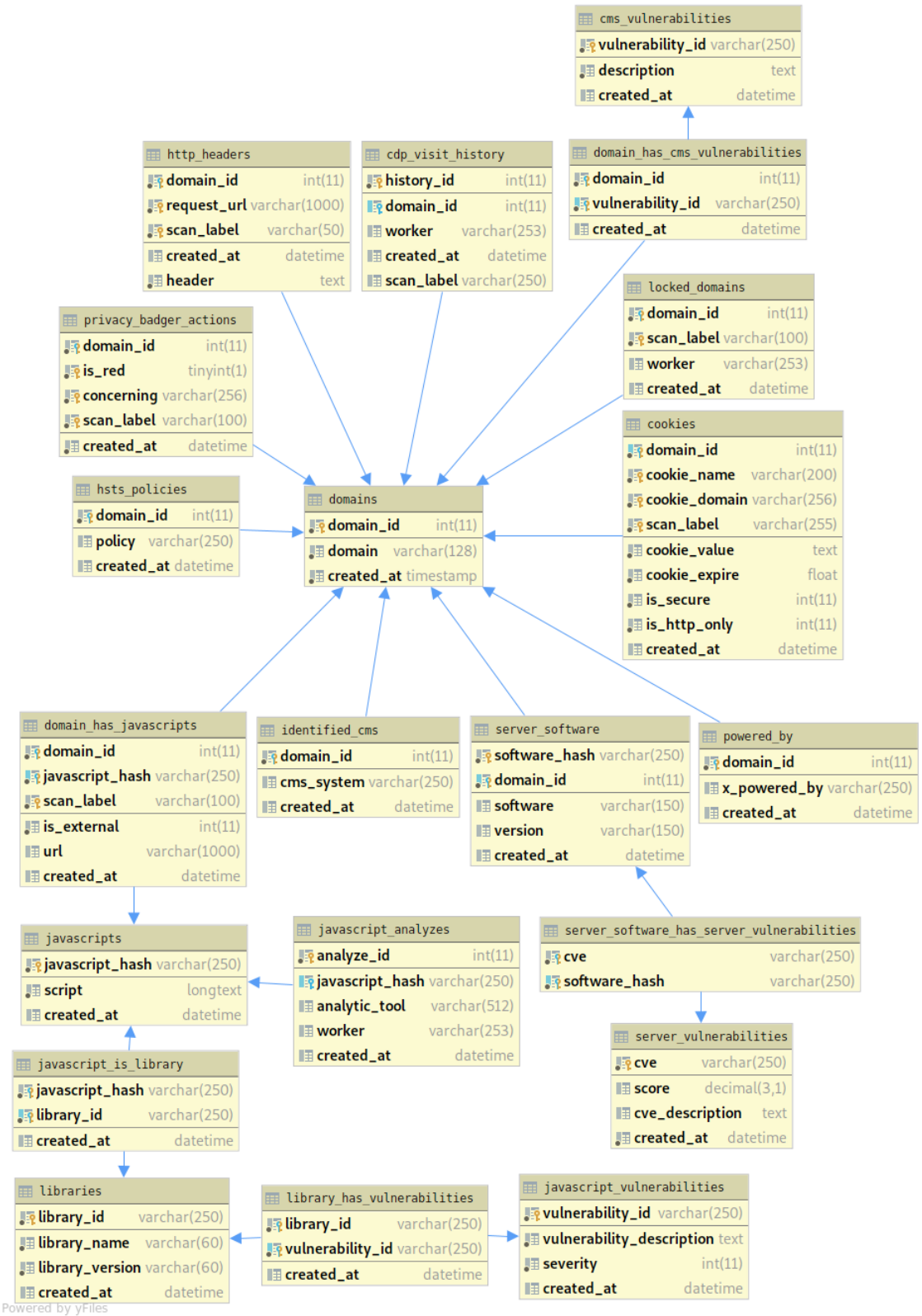
URL
Uniform Resource Locator. 8, 10, 11, 12, 15

WYSIWYG
What You See Is What You Get. 19

XSS
Cross-Site Scipting. 3, 13

Appendix A

Database Schema


cms_vulnerabilities
vulnerability_id varchar(250)
description text
created_at datetime
http_headers
domain_id int(11)
request_url varchar(1000)
scan_label varchar(50)
created_at datetime
header text
cdp_visit_history
history_id int(11)
domain_id int(11)
worker varchar(253)
created_at datetime
scan_label varchar(250)
domain_has_cms_vulnerabilities
domain_id int(11)
vulnerability_id varchar(250)
created_at datetime
locked_domains
domain_id int(11)
scan_label varchar(100)
worker varchar(253)
created_at datetime
privacy_badger_actions
domain_id int(11)
is_red tinyint(1)
concerning varchar(256)
scan_label varchar(100)
created_at datetime
cookies
domain_id int(11)
cookie_name varchar(200)
cookie_domain varchar(256)
scan_label varchar(255)
cookie_value text
cookie_expire float
is_secure int(11)
is_http_only int(11)
created_at datetime
hsts_policies
domain_id int(11)
policy varchar(250)
created_at datetime
domains
domain_id int(11)
domain varchar(128)
created_at timestamp
domain_has_javascripts
domain_id int(11)
javascript_hash varchar(250)
scan_label varchar(100)
is_external int(11)
url varchar(1000)
created_at datetime
identified_cms
domain_id int(11)
cms_system varchar(250)
created_at datetime
server_software
software_hash varchar(250)
domain_id int(11)
software varchar(150)
version varchar(150)
created_at datetime
powered_by
domain_id int(11)
x_powered_by varchar(250)
created_at datetime
javascripts
javascript_hash varchar(250)
script longtext
created_at datetime
javascript_analyzes
analyze_id int(11)
javascript_hash varchar(250)
analytic_tool varchar(512)
worker varchar(253)
created_at datetime
server_software_has_server_vulnerabilities
cve varchar(250)
software_hash varchar(250)
server_vulnerabilities
cve varchar(250)
score decimal(3,1)
cve_description text
created_at datetime
javascript_is_library
javascript_hash varchar(250)
library_id varchar(250)
created_at datetime
libraries
library_id varchar(250)
library_name varchar(60)
library_version varchar(60)
created_at datetime
library_has_vulnerabilities
library_id varchar(250)
vulnerability_id varchar(250)
created_at datetime
javascript_vulnerabilities
vulnerability_id varchar(250)
vulnerability_description text
severity int(11)
created_at datetime
Powered by yFiles

Appendix B

Cookies for railgun.dk

#	Name	Type
2	facebook	tracking
8	id5-sync	tracking
2	agkn.com	tracking
2	amazon-adsystem	ads
4	adtlgc	ads
13	railgun	selfset
1	alphonso.tv	tracking
1	admixer	ads
1	tribalfusion	ads
2	advertising	ads
1	spotxchange	ads
1	acuityplatform	tracking
2	justpremium	ads
2	playground	ads
1	omnitagjs	ads
1	blismedia	tracking
1	yahoo	tracking
2	bidr.io	tracking
2	bluekai(oracle)	tracking
1	rgtrk.eu	tracking
3	adhigh.net	ads
3	mfadsrvr	tracking
3	rtb.4finance	ads
3	bidswitch	tracking
1	adx.adform.net	ads
2	de17a.com	ads
1	deepintent	tracking
1	zorosrv	tracking
3	ctnsnet	tracking
3	yieldoptimizer.com	tracking
6	casalemedia.com	tracking
1	colpirio	tracking
2	onaudience	tracking

13	smartadserver	ads
1	ipredictive	tracking
2	quantserve	tracking
1	apxlv	tracking
34	pubmatic	ads
3	doubleclick(google)	ads
25	powerlinks	tracking
2	sxp.smartclip	tracking
2	rlcdn	tracking
2	turn.com	tracking
3	gwallet	tracking
2	rfihub	tracking
3	rundsp	tracking
2	stackadapt	ads
1	snapchat	tracking
8	stickyadstv	tracking
1	semasio	tracking
1	skimresources	tracking
2	contextweb	tracking
1	sharethrough	tracking
2	tidaltv	tracking
3	tapad	tracking
2	adsrvr	tracking
1	m6r.eu	tracking
1	collector.cint.com	tracking
1	3lift	tracking
4	truoptik	tracking
1	tracead	tracking
1	teads	tracking
2	254a	tracking
4	360yield	tracking
2	bidswitch	tracking
2	creative-serving	tracking
2	programattik	tracking
2	pool.admedo	tracking
2	r.scoota.co	tracking
3	services.experianmarketingservices	tracking
2	tag.clrstm	tracking
1	taboola	tracking
1	erne.co	tracking
1	mrpdata.net	tracking
1	exelator	tracking
1	adform.net	tracking
2	adotmob	tracking
2	scorecardresearch	tracking
1	simpli.fi	tracking
2	videohub.tv	ads
1	targeting.unrulymedia	tracking

7	flyrejser.dk	ads
1	adfarm1.adition	tracking
1	adscale.de	ads
2	mathtag	tracking
1	ads.avocet	tracking
1	ybrid.ai	ads
1	csync.loopme.me	tracking
1	gumgum	tracking
1	w55c	ads
1	zemanta	tracking
4	crwdcntrl	tracking
3	speedline.dk	ads
1	lijit	tracking
1	nrich.ai	tracking
1	1rx.io	tracking
1	addtoany	Cloudflare protection
1	adnami.io	tracking
1	adsymptotic.io	Cloudflare protection
1	glotgrx	Cloudflare protection
1	yabidos	Cloudflare protection
1	match.adsby.bidtheatre	tracking