Bachelor Degree Project

Evaluation of Security of Service Worker and Related APIs

Author: Maxim Kravchenko
Supervisor: Jesper Andersson
Examiner: Dr. Johan Hagelbäck
Semester: VT 2018
Subject: Computer Science

Abstract

The Service Worker is a programmable proxy that allows the clients to keep offline parts of websites or even the whole domains, receive push notifications, have background synchronization and other features. All of these features are available to the user without having to install an application - the user only visits a website. The service worker has gained popularity due to being a key component in the Progressive Web Applications (PWAs). PWAs have already proven to drastically increase the number of visits and the duration of browsing for websites such as Forbes [1], Twitter [2], and many others. The Service Worker is a powerful tool, yet it is hard for clients to understand the security implications of it. Therefore, all modern browsers install the service workers without asking the client. While this offers many conveniences to the user, this powerful technology introduces new security risks. This thesis takes a closer look at the structure of the service worker and focuses on the vulnerabilities of its components. After the literature analysis and some testing using the demonstrator developed during this project, the vulnerabilities of the service worker components are classified and presented in the form of the vulnerability matrix; the mitigations to the vulnerabilities are then outlined, and the two are summarized in the form of security guidelines.

Keywords: Service Worker API, Push API, Cache API, Application Cache, security, Progressive Web Apps, HTTPS

Contents

List of Figures

List of Tables

Listings

1 Introduction

The Service Worker is a web technology that allows the clients to browse websites comfortably while on a poor internet connection or even entirely offline, receive push notifications from the websites, and have more native app-like functionality without having to install a native application form the Application Store. Service worker is still a new web technology, and while many new technologies disappear quite quickly, the service worker is set to be a keystone in the future of the web platform.

While the Service Worker is an incredible technology, it can be quite challenging to explain to a client and check if they would like to allow it since most of the clients "do not generally have sufficient context to understand permission requests." [8] Therefore, the browsers do not ask the clients if they would like to allow a service worker from a particular website to be installed, they install it without the client's awareness. Unauthorized installation raises an issue of security - can it be harmful to the client to have a service worker installed on their machine?

1.1 Background

The Service Worker has an application programming interface (API) and makes use of multiple other APIs. Unlike some other web technologies - such as Application Cache - the service worker is a collection of software functions and procedures interacting with other such collections: Promise, Fetch, Cache, Push, and Notification. Furthermore, the service worker uses HTTPS for secure communication. This abundance of programming interfaces and technologies increases the complexity and flexibility of the service worker significantly. Thus, it is important to make sure that the service worker is implemented and used safely, as well as understand how it could potentially be used in an attack against the client or the server.

1.2 Related work

There has been some research done regarding the security of more traditional technologies used by the service worker. For example, this report covers security of HTTPS, and in that area, much extensive research has been done. Some of the HTTPS-related works are used in this project, for instance, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice" is written on the security of common Diffie-Hellman implementations [9] and "DROWN: Breaking TLS with SSLv2" which focuses on DROWN attack on SSLv2, but also touches upon attacks on older versions of SSL and TLS. [10] Furthermore, some of the vulnerabilities are known to the developers of the service worker. An example of that is a service worker exploit utilizing the cross-site scripting attack via JSONP, which is covered in "Service Worker Security FAQ" by Jake Archibald, one of the creators of the service worker. [8] However, there is little to no research regarding the security and potential abuse of some newer parts of the service worker, for example, Push and Notification APIs.

1.3 Problem formulation

The Service Workers are installed on the clients' machines without the clients' awareness. The most that the client can do is to block the notifications from the website, yet the service worker will still be installed and run. The service worker is a programmable proxy, it is a piece of code which is sneaked into clients' computers and which can be triggered remotely by the server. Taking into account all the aforementioned APIs, the service worker as technology is fairly complex; it is also a new and developing technology. The combination of all these factors makes the service worker an additional attack surface, a target for existing attacks, and a tool for new attacks.

1.4 Motivation

The Service Worker is a powerful tool aimed towards improving web performance and client experience, especially on mobile devices. The service worker is one of the crucial parts of a larger technology called Progressive Web Apps, which brings a native-like experience for mobile users in their browsers when they enter a website. With it, the high-friction step of strongly encouraging - or in some cases even forcing - the clients to go to the Application Store and download an application to use a website's service comfortably can be avoided entirely. Furthermore, the clients can enjoy the offline experience, which is extremely potent in situations when there is no connection at all, for example being on a plane. Alternatively, the connection might be so weak that there is no connection in effect, yet the device still tries to reach the server, for example when the client is on the underground. In addition to that, the service workers offer other benefits, such as background synchronization and push notifications.

All these benefits improve the user experience, which in turn encourages clients to continue using a service or browsing a website. For example, when Forbes redesigned their website as a Progressive Web App, only about 25% of users could get the experience of the PWA right away for various reasons. Nonetheless, Forbes' website "has seen a 20% increase in impressions per page ... 12% increase in the number of users that get to the site ... 6x increase in the number of readers completing articles." [1] The success of PWAs is seen more and more on the web; a website called *PWAstats.com* showcases an abundance of statistics from various websites who redesigned their websites as progressive web apps. One of the statistics featured on this website is about Twitter Lite PWA, which "became the default mobile web experience for all users globally in April 2017" [2] and helped "65% increase in pages per session, 75% increase in Tweets sent, 20% decrease in bounce rate" [2]. A considerable part of this success is attributed to the Service Worker which allows for "enabling users to view and create Tweets as quickly as possible." [2]

1.5 Objectives

In this section the objectives of this thesis work are presented. "SW" in the table below refers to Service Worker.

O1	Understand why an API like SW is desirable from users' and business' points of view
O2	Understand successes and failures of SW's predecessors and their influence on SW
O3	Understand security threats related to the technologies used by SW
O4	Make a vulnerability matrix that outlines existing SW vulnerabilities
O5	Make a list of mitigations for discovered vulnerabilities
O6	Make a list of security guidelines

1.6 Scope and Limitation

This project takes into consideration two scenarios. The first scenario is the server installing a malicious service worker on the client's computer aiming to either obtain some information from the client or compromise their device. The second scenario is the man-in-the-middle attacks - a third party trying to compromise the client.

This project does not take into consideration the situation when the client attempts to attack the server or gain unauthorized access to some resources. The exclusion is mostly due to the fact that this level of attack would require either an unrealistically oversimplified implementation of the server side or decompilation and alteration of an installed service worker.

1.7 Target group

The target group of this project is computer science bachelor students, who have some interest in web development and security of the APIs and technologies covered in this paper. Basic understanding of JavaScript and web communication is required to understand the result and discussion of this project report.

1.8 Outline

The following section covers methodology used in this project to achieve the aforementioned objectives. After that, The Service Worker section covers in-depth Service Worker API and other technologies that it utilizes. Subsequently, Developing Vulnerability Matrix covers the implementation of the artifacts of this project: the service worker vulnerability matrix, list of mitigations, and security guidelines for using the parts of the service worker. In Results, the vulnerabilities are classified according to the criteria outlined in Developing Vulnerability Matrix. Consequently, the artifacts are presented. In Discussion the vulnerabilities, their classification, and mitigations are discussed further. Conclusion and Future Work presents the final overview of the paper and the overall security of the service worker, based on the findings made during this project; suggestions are made regarding further research.

2 Method

This project sets out to point out which parts of the service worker are the weakest, what their vulnerabilities are, and exactly how dangerous they are. This task is achieved by analyzing relevant literature and showcasing the vulnerabilities by using a demonstrator developed during this project. After discovering, discussing, and in some cases demonstrating the vulnerabilities, they are categorized based on the severity, exploitability, and complexity of mitigation. The result of this categorization is a vulnerability matrix, which can be used to keep track of vulnerability management of the service worker in a particular setting. Furthermore, some mitigations to the vulnerabilities are presented. Finally, the technologies and mitigations are mapped to each other in the Security Guidelines.

2.1 Method Description

Firstly, basic information was gathered about the technologies covered in this report and their impact on business and security. After confirming the current and potential utility of the service worker and related technologies, the search for implementation guidelines has been conducted, with the focus on implementation presented by Mozilla and Google, since those guides go into great depth of implementation. Furthermore, these guidelines outline potential bugs and security issues that developers might come across; these security issues served as the starting point in the search for vulnerabilities. Subsequently, the drafts of the related standards have been consulted to obtain inspiration for additional attack vectors. These attack vectors have been confirmed either by showing them through a demonstrator developed during this project or by referring to related academic papers. Consequently, the vulnerabilities are gathered and classified in a vulnerability matrix, in style similar to that of OWASP Top 10; the mitigations to these vulnerabilities are then presented.

2.2 Reliability and Validity

The web technologies are notorious for being ever-changing. While some of the web technologies covered in this project have been established decades ago, tested, and researched thoroughly - such as SSL/TLS - most of the APIs included in this report are in experimental state or rely on standards that are still in the draft state. Thus, the technologies analyzed throughout this project might not change much in the near future, but they have changed incredibly in the past few months and are likely to keep transforming at a steady pace for a few years. For this reason, the most contemporary resources available for the research has been used; additionally, the access dates to all the resources have been attached so that in future it is possible to look into the specific versions of the documents that are known for being dynamic, such as developer blogs by Mozilla and Google. Where possible, the versions of the technologies and third-party software are specified.

2.3 Ethical Considerations

The security testing mentioned in this report does not involve any participants hence it does not pose a threat to anybody directly. For the most part, the web server has been run on a local network behind a NAT, meaning that apart from the author of this paper nobody could have been affected by the web page. For the tests where the web server has been tunneled and made accessible on the internet using ngrok [11], a part of the URL of the web page has been randomized, making it less likely that people not aware of the project would connect to the web server. Furthermore, the web page had warnings in the title and the header so that accidental clients would not allow notifications on their devices. A tool called *One Signal* has been used in this project; it allowed for tracking which clients have installed the service worker, and it shows that the only clients that installed the service worker were the test machines. The files stored by the vulnerable service worker and the service worker itself have been erased from all the devices used in this research project.

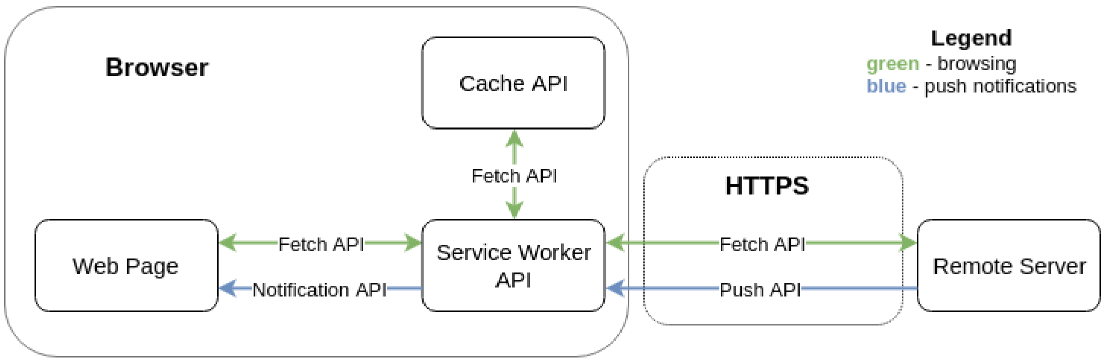


Figure 3.1: Service Worker Architecture

3 The Service Worker

This thesis project is concerned with the security of the Service Worker, so it is essential to have a solid understanding of what the Service Worker is. However, the Service Worker as technology makes use of several other technologies and Application Programming Interfaces (APIs): Promise API, Fetch API, Cache API, Push API, Notification API, HTTPS, and Service Worker API. Figure 3.1 is a visual representation of the service worker architecture. Note that Promise API is used in all parts (APIs) presented in the Figure 3.1.

3.1 Promise API

Traditionally, when a sequence of commands needs to be executed, the callback functions have been used:

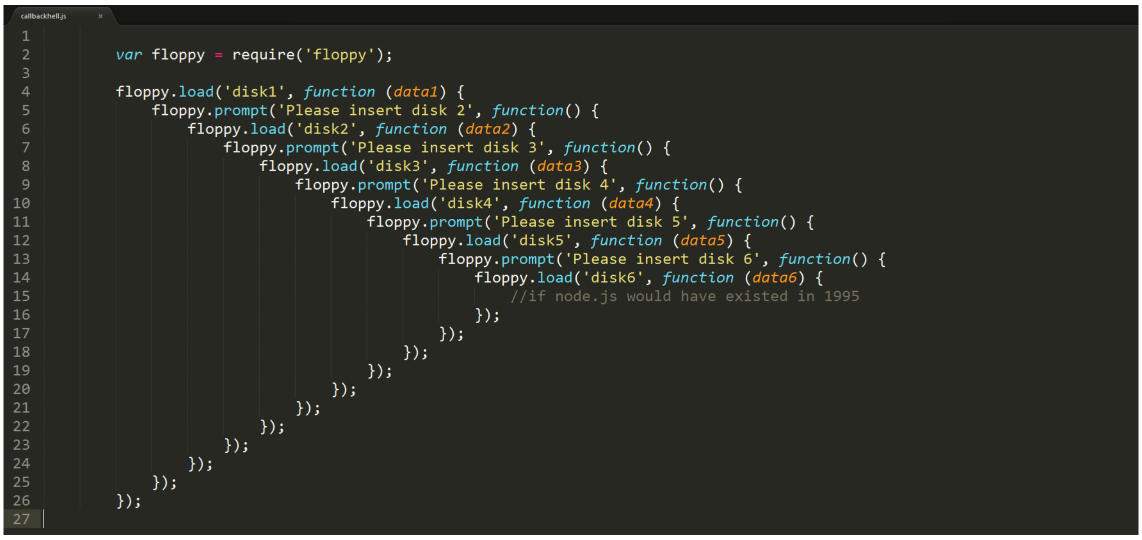
```
doSomething(successCallback, failureCallback);
function successCallback () {
    //Proceed execution
}
function failureCallback () {
    //Display error
}
```

Listing 1: Callback Example

Multiple callbacks can be used for one function. For example, if an HTTP package is received and the program must act differently depending on the status code of the package, the function *receiveHTTP* can have multiple callbacks corresponding to different status codes.

While this way of structuring execution of a program is useful in some scenarios, it has some downsides. For example, structuring code in this manner while trying to achieve asynchronous functionality is prone to becoming something that is known as "callback hell" or "pyramid of doom." Callback hell is a situation in which multiple callback functions are nested, which makes the code hard to read and debug. Figure 3.2 is a demonstration of callback hell.

One way to avoid this problem is using a different approach to asynchronicity - Promise API. "A Promise is an object representing the eventual completion or failure of an asynchronous operation." [12] At first, the difference between callback functions



```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', function() {
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in 1995
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

Figure 3.2: "Callback hell" [3]

and promises is inconsequential - instead of passing callbacks into a function, they are attached to an object. However, using an object allows for chaining - using the result of one promise as a trigger for starting the following operation. Passing the results of promises allows for avoiding the callback hell, hence for more readable and easier to debug code. Furthermore, Promise API guarantees error propagation and completion of the concurrent run. Error propagation means that once an exception is encountered in the chain of promises, the execution stops and looks down in the chain for the catch statement. Thus, error propagation eliminates the need for having multiple catch statements that are often present in callback hell. The guarantee of completion of the concurrent run means that the next promise will never be executed until the promise that is running at the moment has finished executing. This guarantee allows for promise chaining, where every action is taken based on the success or failure of the previous action. An example of that can be seen in Listing 2 from Google, which fetches an image, converts the response to a blob, works with this blob and returns the resulting array and a promise, meaning that the function can be a part of a promise chain as well. The beauty of this way of coding is that it is as easily read as synchronous code yet it is asynchronous which means that it is far more efficient.

```
// function for loading each image via fetch
function imgLoad (imgJSON) {
 // return a promise for an image loading
 return fetch(imgJSON.url)
 .then((response) => response.blob())
 .then(function (response) {
   //do stuff
   return arrayResponse
 }).catch(function (Error) {
   console.log(Error)
 })
}
```

Listing 2: Promise Chaining

Since all APIs that are used by the Service Worker utilize Promise API, it is key to the functionality of the Service Worker.

3.2 Fetch API

Fetch API is responsible for HTTP and HTTPS requests and responses. The key global method of this API is *fetch()*, which "provides an easy, logical way to fetch resources asynchronously across the network." [13]

While fetching in an asynchronous way can be achieved using XMLHttpRequest API, Fetch API is more modular, flexible, and easier to use, which gives it an edge over XML-HttpRequest. An example of the flexibility of Fetch is its support of Cross-Origin Resource Sharing (CORS) and RequestMode parameters for it: *same-origin*, *no-cors*, *cors*, *navigate*. Choosing between these modes allows for different degrees of exposure and modification of HTTP headers, which aids security and privacy. Another important benefit of using Fetch is that it can use Cache API to interact with cache: a developer can define which request and response objects to store locally for future use, what should come only from the cache and what should never be stored. This functionality is crucial for an application with offline support yet it is not available in XMLHttpRequest. Moreover, Fetch supports streaming, which is particularly useful when looking for smaller parts of larger files.

3.3 Cache API

Cache API is storage for network request and response pairs. It can store "any kind of data that can be transferred over HTTP" [14], and this data can be retrieved and used again. Unlike more traditional caching in browsers, Cache API was designed to be used by developers and software to provide offline functionality to websites and applications, in particular by Service Worker API. An example of extended control over stored files is the fact that the files stored in Cache API never expire, nor do they get updated automatically - it is up to developers to ensure that the cache stays up-to-date and is removed when it is no longer useful. Moreover, Cache API is asynchronous meaning that it can be used by Service Worker API, which is designed to be fully asynchronous.

Apart from the Service Worker, it can be accessed from both *window* object and other workers, meaning that it can be used as general storage. In the context of the service worker, the cache is often used to store the files that form the "shell" of the website or application and therefore do not change frequently - "the minimal HTML, CSS and JavaScript powering the user interface." [15]

3.4 Push API and Notification API

The core goal of Push API is to push notifications to clients. The pushed messages are then received by Service Worker API and are displayed using the Notification API even if the user's browser is closed. Push API is based on Service Worker API since the service worker provides the entry point for Push. The user subscribes to push messages via the service worker, and the service worker is then responsible for reacting to push messages appropriately, e.g., displaying a notification on the screen. The push notifications are quite interactive - apart from the usual "click" action, the messages can have multiple types of interaction, e.g., asking the user a question and giving them two answer buttons, clicking which triggers different actions. Furthermore, the push notifications can have sound and vibration which can be configured, for example, it is possible to specify the number of milliseconds a device will vibrate and the length of the pause between the vibrations.

A notable drawback to involving Service Worker API for processing the push messages is "increased resource usage, particularly of the battery" [16]. At the moment of

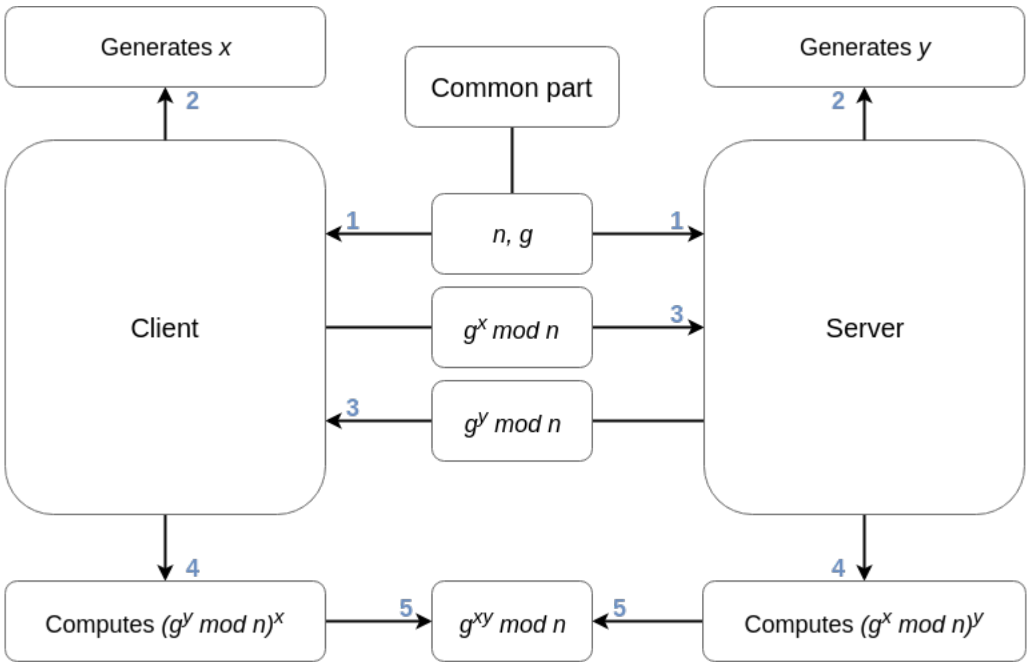


Figure 3.3: Diffie-Hellman Key Exchange

writing this paper, there is no standard for taking care of resource usage, so different browsers handle this in different ways that are discussed in later sections of this paper.

3.5 HTTPS

HTTPS is an improved version of HTTP - Hyper Text Transfer Protocol. HTTP is used for transferring data across the web, and it is used mostly for transfer of various web pages and multimedia files. While this is an incredibly useful protocol, it lacks security features and can, therefore, be exploited for malicious actions. On the higher level, HTTP can be exploited for active alteration - e.g., insertion of unauthorized advertisements into a web page [17] - and for passive observation of users. The observation is often linked with sensitive data exposure. However, it is not only the sensitive data that has to be protected - by observing browsing patterns of a user an attacker can learn their identity and use it maliciously.

HTTPS utilizes the HTTP protocol in combination with Secure Socket Layer protocol (SSL) or, with its improved version, Transport Layer Security protocol (TLS). The TLS handshake is used to establish the connection between the client and the server. First, the client communicates a series of security parameters to the server, then the server responds with its certificate and confirms whether it can use the security parameters specified by the client. This part is encrypted with asymmetric encryption, by the client using the public key of the server, which is found in the server's certificate. If the two agree on the ciphers to be used, they then proceed to negotiate the private key, via Diffie-Hellman key exchange, that will be used for encrypting the application data; the encryption, in this case, will be symmetric. Figure3.3 demonstrated the Diffie-Hellman key exchange process; note that prime numbers x and y are generated by the client and the server. Figure 3.4 demonstrates the TLS handshake.

HTTPS is required by the Service Worker and Push APIs, though they can work with HTTP in development mode, i.e., when the Service Worker and the corresponding page are accessed via the localhost address.

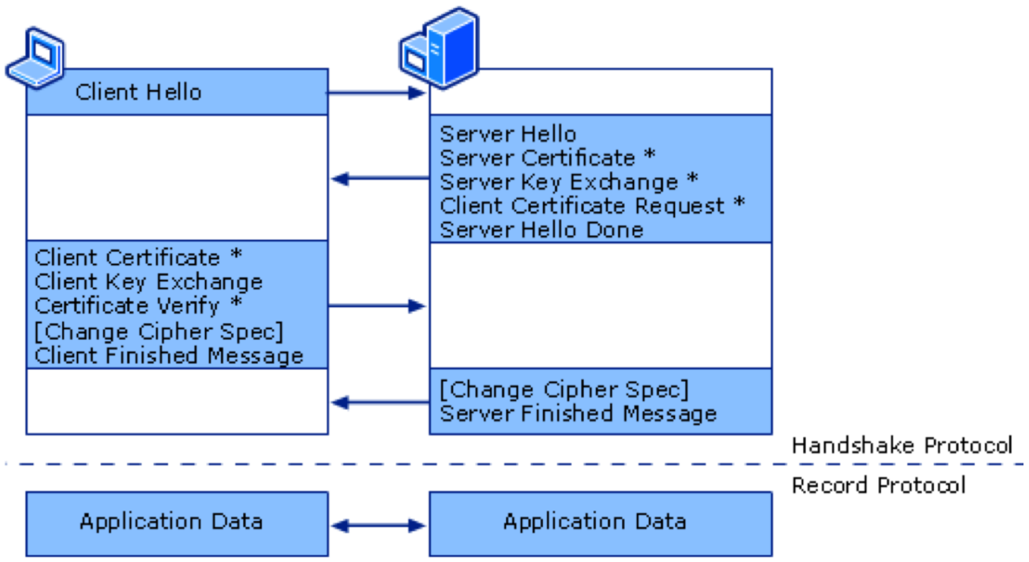


Figure 3.4: TLS Handshake [4]

3.6 Application Cache

One of the key capabilities of Service Worker is the ability to cache parts of the websites - or entire websites - and serve them to users when necessary. There have been previous attempts to design a technology to do just that; the most successful predecessor of the service worker is ApplicationCache. The websites can be divided into two categories: websites for looking up content - e.g., Wikipedia, YouTube - and for creating content, for example, Google Docs. ApplicationCache (AppCache) was designed to cache the second types of websites and allow clients to use them offline [18]. While it is possible to use AppCache for the websites type of files, many of the quirks of AppCache make it highly impractical on a bigger scale, such as Wikipedia.

The AppCache has many quirks which resulted in a need for a new offline technology for the web. The quirks are there in the first place because the AppCache has many implicit behaviors, which are useful when one wants to use the technology out-of-the-box, without having to go deep into the configuration. However, the problem with AppCache is that once somebody starts to adjust and configure the tool more precisely, the result is often unexpected and counter-intuitive, due to the fact that the technology "assumes" too much of its role and desired functionality. While it does not render AppCache completely useless, it means that the technology is difficult to use effectively.

Despite its flaws, AppCache provides an important feature - offline capability. It is clear that this feature is desirable, especially for mobile users, and since AppCache is challenging to use effectively, the need for a new, improved technology arises. That is where Service Worker comes into play. Continuing with the idea of offline-capability, Service Worker is even more powerful, allowing for the features described previously. However, learning from the mistakes of AppCache, Service Worker has little to no implicit behaviors. While it does mean that to utilize the technology fully one has to go through more trouble of setting it up according to their needs, it also means that the obscure quirks are no longer there - the developer can define when and how to use Service Worker with notable granularity. Thus, a service worker on the website can be as simple or as complex as the developer needs it to be.

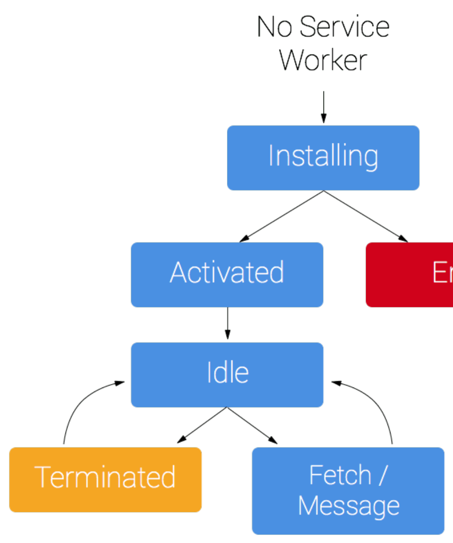


Figure 3.5: Service Worker Life Cycle [5]

```
 1  // register service worker
 2  if ('serviceWorker' in navigator) {
 3   navigator.serviceWorker.register('sw.js')
 4       .then(function (reg) {
 5           if (reg.installing) {
 6               console.log('Service worker installing')
 7               } else if (reg.waiting) {
 8                   console.log('Service worker installed')
 9               } else if (reg.active) {
10                   console.log('Service worker active')
11               }
12           swRegistration = reg
13           initializeUI()
14   }).catch(function (error) {
15     // registration failed
16     console.log('Registration failed with ' + error)
17   })
18  }
```

Listing 3: Service Worker Registration

3.7 Service Worker API

A service worker is a programmable proxy that the server installs on the client, and that is used to process all requests to the domain. It is run in a separate thread, meaning that it is not thread-blocking, and it utilizes only asynchronous technologies; the combination of the two makes it fully asynchronous. Service worker is terminated when not in use in an attempt to improve the resource usage.

Although the service worker is installed upon visiting a web page, the service worker's life cycle is not dependent upon the web page. Figure 3.5 demonstrates "overly simplified version of the service worker life cycle on its first installation." [5]

Listing 3 demonstrates registration of the service worker. The actual registration occurs in line 3, while the rest is checking the browser support for service workers, logging, and error handling. Note that the Listing 3 is a part of the main Javascript file (in this case *app.js*)

After the registration, the actual installation of the service worker takes place. Once the installation begins, the Listing 4 from *sw.js* is run, where caching begins in line 4. Most commonly, the files that are cached on installation are the files that form the shell of the app/website or the files that are often used but rarely changed, for example, as mentioned previously, menus, information pages, and CSS files.

```
 1  //service worker installation
 2  self.addEventListener('install', function (event) {
 3   event.waitUntil(
 4     caches.open('v1').then(function (cache) {
 5       return cache.addAll([
 6         //specify the files to be cached
 7       ])
 8     })
 9   )
10  })
```

Listing 4: Service Worker Installation

Now the service worker is fully installed and can be configured to one's needs. For example, the service worker can intercept *fetch* requests and act depending on the connection of the client; Listing 5 demonstrates this functionality. Note that the line 5 returns the result of *either* the cached response to the request *or* the network response to this request. This race is done to improve the user experience - it can be hard to determine whether it is faster to contact the server for a new response or obtain it from the cache, so the service worker makes both requests and displays whichever returns first.

By utilizing aforementioned technologies, Service Worker API achieves different functionality. When it intercepts requests and responses from the client and the server, it can use Cache API to store them and use in future, for example when the internet connection is slow or when there is no connection at all. Using the service worker in combination with cache allows for very granular selection of which resources should be stored, when they should be updated, and when they should be deleted. The combination of Push and Notification APIs is used for notifications from the sources that the user has subscribed to using service workers. Service Worker also supports background synchronization, though it is not yet standardized.

```
 1  //intercepting network requests
 2  self.addEventListener('fetch', function (event) {
 3   event.respondWith(
 4   caches.match(event.request).then(function (response) {
 5     return response || fetch(event.request)
 6   }).catch(function (error) {
 7     console.log(error)
 8   })
 9   )
10  })
```

Listing 5: Intercepting *fetch* Requests

4 Developing Vulnerability Matrix

The inspiration for the vulnerabilities is taken from official sources that outline the existing problems, such as sources [6, 8, 19], standards that outline what practices should be followed by the developers, such as sources [20, 21], and advanced tutorials that are published by credible sources, such as [5,7,14,16,22,23].

The threats concluded from these sources are then divided based on the part of the service worker that they target (API or technology), for example, Push API and HTTPS. Such separation allows for a more systematic and focused look into the parts of the service worker and as a result the service worker as a whole; it offers a definite conclusion regarding the security of the service worker since technology is only as safe as its weakest point.

Potential vulnerabilities found in the separate sections are then studied more closely, either by showing them through a basic demonstrator, build during the work on this project, or through studies of academic papers that point out the issues in the parts of the service worker, such as sources [9, 10, 24–31]. After the demonstration of or the review of the research done about the vulnerabilities of the technologies and APIs in question, an analysis of the vulnerabilities is produced. The vulnerabilities are measured against three criteria presented in Table 4.1, in a manner similar to that of the OWASP Top 10 [29].

Criterion	Variation	Meaning
Exploitability	Difficult	to exploit the vulnerability the attacker must be experienced, in proximity to the victim, and in a fortunate situation, e.g. a rare outdated version of the software
	Average	the vulnerability exploitation requires some level of skill and luck
	Easy	the vulnerability can be exploited by an average computer user
Severity	Minor	the impact of the attack is negligible
	Moderate	the attack has a considerable impact on user experience
	Severe	the impact of the attack is detrimental to the user's information safety
Complexity	Low	the vulnerability can be mitigated quickly and without additional monetary costs such as buying
	Moderate	the vulnerability requires some time to be mitigated
	High	the vulnerability takes a long time and additional resources to mitigate

Table 4.1: Criteria Used in the Service Worker Vulnerability Matrix

The result is the service worker vulnerability matrix that maps the technologies used by the service worker, their vulnerabilities, and the level of each category outlined above. The table 4.2 is an example of such matrix.

Once the vulnerabilities have been found and classified, various ways of mitigation are outlined, and security guidelines are presented. The remediation is based mainly on the developer guidelines written by developers from large companies - such as Mozilla and

Technology	Vulnerability	Exploitability	Severity	Complexity
API 1	V 1	difficult	minor	low
	V 2	average	moderate	moderate
API 2	V 3	easy	severe	high

Table 4.2: Example of a Service Worker Vulnerability Matrix

Google - who have been developing the service worker technology, as well as research papers used throughout this report that focus on various vulnerabilities in technologies that form the service worker. The demonstrator is used to show vulnerabilities associated with Service Worker and Push APIs.

4.1 Demonstrator

As of the time of writing this thesis, the W3C specification of Service Worker implementation is a working draft [20], meaning that the implementations and guidelines on how to build service workers differ from one website to another. Currently, Service Worker is supported by Chrome, Firefox, Opera, Samsung Internet, and Safari; it is under development in Edge. With the exceptions of Safari and Edge, all companies provide some guides and examples on how to build service workers. The guides can be separated into two categories: low level (building a service worker from scratch) and high level (using external tools to build a service worker) development. Guides of both categories are designed to be entry-level, meaning that they expect the reader to be completely new to the service worker technology, though the low-level guides go into much greater depth.

Opera and Samsung have high-level guides for developing a service worker. Instead of programming a new service worker, they provide the user with the generating tools and templates - Samsung, [32]- or offer a higher level library that takes care of setting up the service worker (Opera, [33]). Because of that, it is impossible to compare the similarities and differences in their approach.

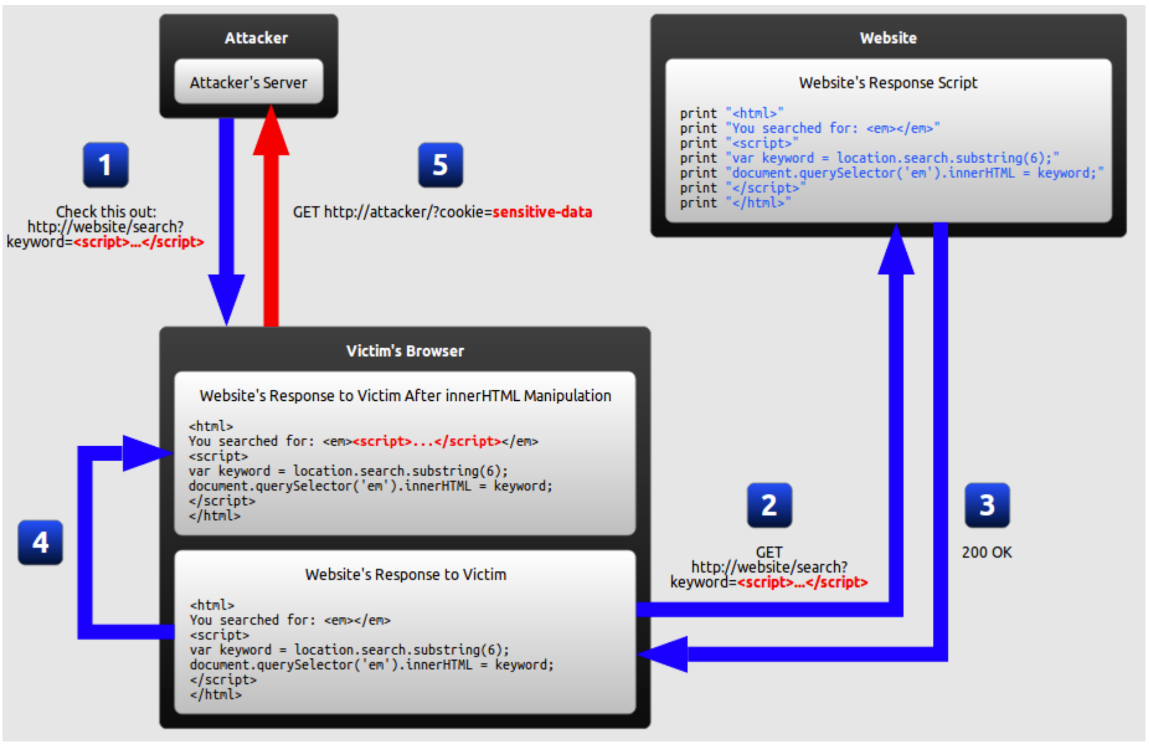
Mozilla and Google offer low-level approach - unlike the high-level guides, they set out to teach the reader to implement a service worker themselves and provide some demos to play with the written code. Low-level guides go into details of implementation and practices of all APIs and technologies related to service Worker API.

4.1.1 Client Side

The Service Worker demonstrator that was developed during this bachelor project focused mostly on the two low-level guides and their demos, as well as guidelines from W3C Working Draft for the service Worker [20]. The implementation in this project uses some code for from Google - Service Worker shell, basic offline functionality, and basic Push message notifications. While one of the initial goals of this project was to analyze the differences in implementation of service workers between the low-level guides, it turned out that they are complementary to each other - each contains information that the other is lacking, and they cross-reference each other continuously.

4.1.2 Server Side

The implementation is complementary to the research of this project and not the main focus, hence instead of creating a brand new back-end server, multiple technologies and libraries are used to form the back-end. One of this technologies is an extension for


Attacker
Attacker's Server
Website
Website's Response Script
1
Check this out:
http://website/search?
keyword=<script>...</script>
5
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim After innerHTML Manipulation
Website's Response to Victim
4
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK

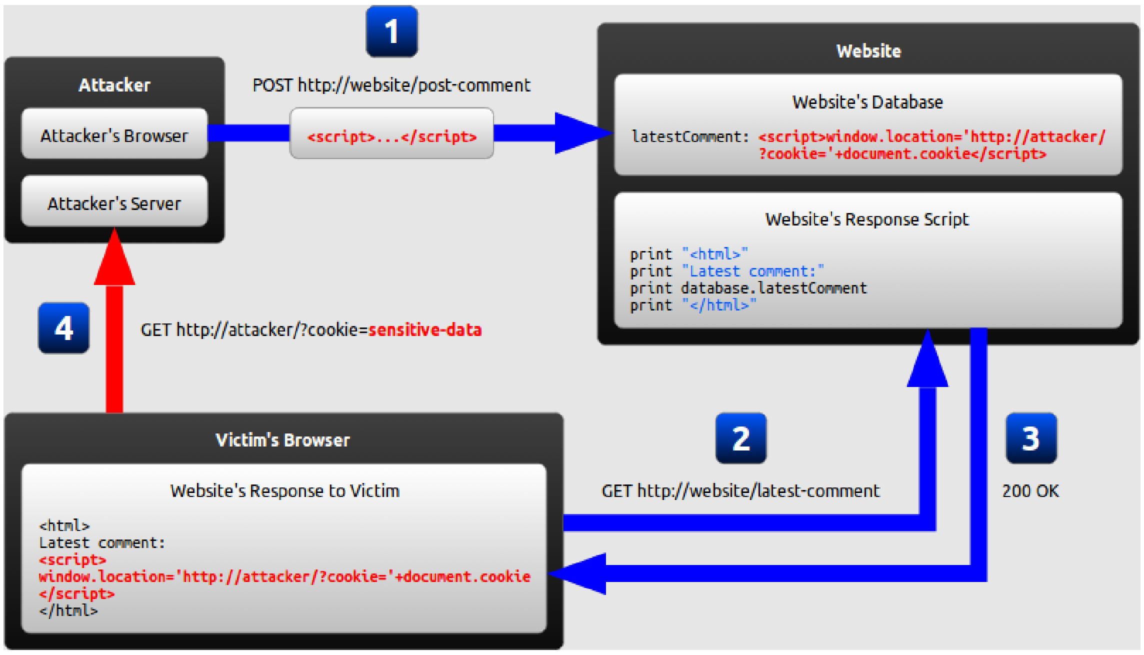
```
print "<html>"
print "You searched for: <em></em>"
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
```

```
<html>
You searched for: <em><script>...</script></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

```
<html>
You searched for: <em></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

Figure 4.6: An example of DOM-based XSS [6]

Chrome called *Web Server for Chrome* offered by *chromebeat.com*. It is an easy-to-use web server, which works over a network and can (given some additional privileges on the network) work over the internet. However, as mentioned in Section 3.5, the websites and apps that use Service Workers **must** be run over HTTPS, which is not provided by Web Server for Chrome. For that reason, *ngrok* is used - it is an application which "exposes local servers behind NATs and firewalls to the public internet over secure tunnels" [11]. While it offers many useful features, the most important features of this project are the ease of making the website available on the LAN and providing communication between the server and the client over HTTPS. *One Signal* is used for management of the clients' push notification subscriptions and the push notifications themselves. Some of the push notification testings have been done using *Google Push Companion*.


1
POST http://website/post-comment
Attacker
Attacker's Browser
<script>...</script>
Attacker's Server
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/latest-comment
3
200 OK
Victim's Browser
Website's Response to Victim

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

Figure 4.7: An example of persistent XSS [6]

4.2 Vulnerabilities

While there are a plethora of attacks related to web applications and web technologies in general, the focus of this paper is security vulnerabilities of the service worker. Therefore, the following sections contain vulnerabilities that are mostly service worker-specific. For convenience, the attacks are separated by the APIs or technologies that they target.

4.2.1 Service Worker API

Service Worker can be vulnerable to a Cross-Site Scripting (XSS) attack using JSON with Padding (JSONP). [8] An XSS attack allows the attacker to "perform cookie stealing, malware-spreading, session-hijacking, and malicious redirection." [24] There are different taxonomies of XSS - traditionally, the XSS attacks have been classified as DOM-based, Reflected, or Stored [24, 29]. However, these types are rarely well-defined, in practice, they often overlap. Therefore, a more modern taxonomy is Server XSS and Client XSS. Server XSS occurs when the attacker stores malicious script on the server and the client receives this script. Figure 4.6 is an example of DOM-based XSS, which can also be classified as a Server XSS. Client XSS means that the page that was received from the server is not malicious, but a script on the client's machine alters the page.

There are numerous ways in which an XSS attack can be executed. For example, suppose the following scenario from source [6], demonstrated in Figure 4.6:

1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request but does not include the malicious string in the response.
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.

5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.

This particular example focuses on stealing victim's cookies. Now suppose that the goal of the attacker is to install a malicious service worker on the website. Since the service worker is a programmable proxy that can span the entire domain, if the attacker can install a malicious service worker on the victim's machine, then they will execute a successful persistent man-in-the-middle (MITM) attack.

Another example of a persistent XSS attack can be seen in Figure 4.7. Consider the code snippet from the demonstrator, presented in Listing 6, which simulates the scenario presented in Figure 4.6. The users are expected to insert their name into an input field in line 1 and press the button to send it to the server. In this case, instead of sending to the server, the value is stored in *localStorage*. Line 4 represents fetching information that was stored by other users from the server (in this case it is loaded from *localStorage*). Taking into account that the input data is simply *stringified* in line 13 and *parsed* in line 16, an attacker can inject a malicious payload which will then be send to the victim once it enters the page (in this case - presses *Pull* button). An example of a malicious payload can be seen in Listing 7.

```
 1  First name: <input id="name" name="fname">
 2  <button  value="Submit" onclick="push()">Submit</button>
 3  <p id="test"></p>
 4  <button  value="previous" onclick="pull()">Pull</button>
 5  <p id="past"></p>
 6  <script>
 7    function push () {
 8    let n = document.getElementById('name')
 9    window.localStorage.setItem('name', '[]')
10    let name = JSON.parse(window.localStorage.getItem('name'))
11    name.length = 0
12    name.push({name: n.value})
13    window.localStorage.setItem('name', JSON.stringify(name))
14  }
15    function pull () {
16      let name = JSON.parse(window.localStorage.getItem('name'))
17      let p = document.getElementById('past')
18      p.innerHTML = name[0].name
19    }
20  </script>
```

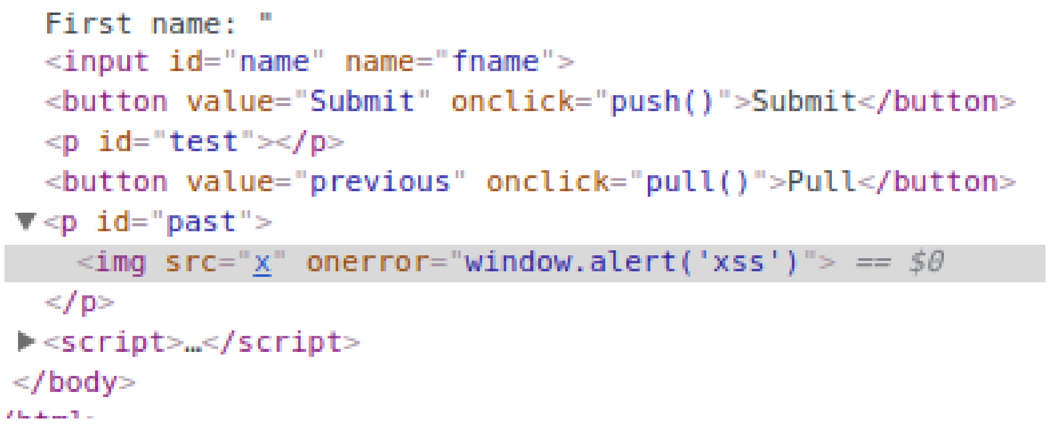
Listing 6: Unsafe Handling of User Input

In this case, once the victim enters the page, the DOM will be updated to contain the malicious script, as displayed in Figure 4.8. The victim's browser will try to fetch an image from an invalid source, which will trigger an error. The attacker specified the script that is to be executed when an error occurs, in this case, it is displaying an alert.

```
1  <image src=x onerror=window.alert('xss')>
```

Listing 7: Example XSS Payload

Referring to the examples shown in Figure 4.6 and Figure 4.7, an attacker could try to include a *fetch* in the payload to download the malicious script and then install it as shown in Listing 3, but the call would not succeed. It would fail due to the same-origin policy



```
    First name: "
    <input id="name" name="fname">
    <button value="Submit" onclick="push()">Submit</button>
    <p id="test"></p>
    <button value="previous" onclick="pull()">Pull</button>
  ▼<p id="past">
      <img src="x" onerror="window.alert('xss')"> == $0
    </p>
  ▶<script>…</script>
  </body>
```

Figure 4.8: Result of Successful XSS Attack

(SOP) that prevents downloading resources from other domains. However, the attacker can attempt to use JSONP.

Using JSON a client can send an *XMLHttpRequest* to the server to execute a function. Using JSONP a new *script* tag is appended to the HTML page and then the call is made to the function *foo* specified in the call. Therefore, *foo* must exist in the global scope during the time that the request is made and the function has to exist on the server side. Since the client does not have access to the list of the server functions, the server often declares *callback* function, which is defined by the client. Crucially, *script* tag does not comply with SOP, hence downloading something from a different origin is not an issue. Fetching resources from other domains is the primary use of JSONP in the industry. An attacker can modify the *callback* function to fetch and install a malicious service worker and take over a domain.

4.2.2 Push API

Social Engineering

"A push subscription has an associated push endpoint. It MUST be the absolute URL exposed by the push service where the application server can send push messages to. A push endpoint MUST uniquely identify the push subscription." [21] With the push endpoint the client is uniquely identified by the server, and without the endpoint, a third party cannot send any notifications to the client. However, "the application server is able to share the details necessary to use a push subscription with a third party at its own discretion." [21] This means that there is no guarantee that a push notification came to the client from the "the same origin as the web app." [21] While not a vulnerability in itself, this allows for a plethora of social engineering attacks. Suppose the following: the user entering a website X - which they deem trustworthy - allows notifications from X. Now if X provides the push endpoint of the client to another company Y - e.g., for advertising - the client will receive a notification from website/application X, which in reality was sent by Y. Y could be an obvious scam or a cover-up for a malicious website, but the client will think that the notification came from X, so they will be likely to trust it.

Push Without Notification

Another potential problem is excessive battery usage. Once the push messages arrive on the client's machine, the service worker is activated in order to deliver the push notification to the client. While this is not going to drain the battery significantly after one push message, a significant amount of them could affect the battery life. It is stated on the Mozilla Development website that "activating a service worker to deliver a push message can result in increased resource usage, particularly of the battery" [16], and that it is browser's responsibility to handle the battery usage. Chrome, for example "applies no limit, but requires that every push message causes a notification to be displayed" [16]. The need for displaying the push notifications is also stated numerous times in Chrome Push guides: "you must show a notification when you receive a push." [22] However, there is no specific information regarding what happens to the push message itself in the situation when a Push message is not displayed.

Consider the example presented in Listing 9 Note that the line 12 makes the event wait until the notification is displayed - shown to the user and either timed-out, closed, or interacted with by the user. Figure 4.9 is an example of a push message received with this code. Now consider a situation in which instead of the line 12 in the Listing 9 service worker does something without showing the notification, for example, Listing 8.

```
event.waitUntil(console.log('A Push message without a notification!'))
```

Listing 8: Printing Out a Message in Console on Receiving a Push Message

```
1  self.addEventListener('push', function (event) {
2   console.log('[Service Worker] Push Received.')
3   console.log(`[Service Worker] Push had this data:
4          "${event.data.text()}"`)
5   let text = `${event.data.text()}`
6   const title = 'Push Codelab'
7   const options = {
8     body: text,
9     icon: 'images/icon.png',
10    badge: 'images/badge.png'
11  }
12  event.waitUntil(self.registration.showNotification(title, options))})
```

Listing 9: Standard Approach to Reacting to a Push Event

According to [16], Chrome requires that the push message triggers the notification, which implies that not showing the notification should cause no reaction from Chrome. Similarly, Google states in [7] that "Chrome will only show the [default] notification [as presented in Figure 4.10] when a push message is received, and the push event in the service worker does not show a notification..." However, this does not seem to function as intended. The demonstrator service worker is configured not to display notifications on push messages and print in the console. Figure 4.11 demonstrates that the console printing works, while the default message is not being displayed to the user. The ability to execute code without notifying the user raises the issue of battery drainage.

Firefox, on the other hand, ignores the push message entirely - there is neither a printed

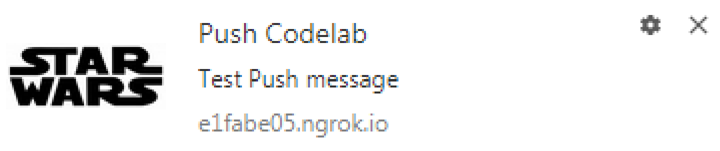
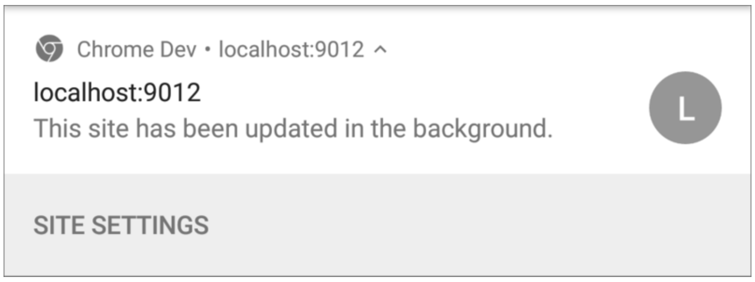

Push Codelab
Test Push message
e1fabe05.ngrok.io

Figure 4.9: Standard Push Message


Chrome Dev • localhost:9012

localhost:9012

This site has been updated in the background.

Figure 4.10: Example of a Default Message for Push Message in Chrome [7]

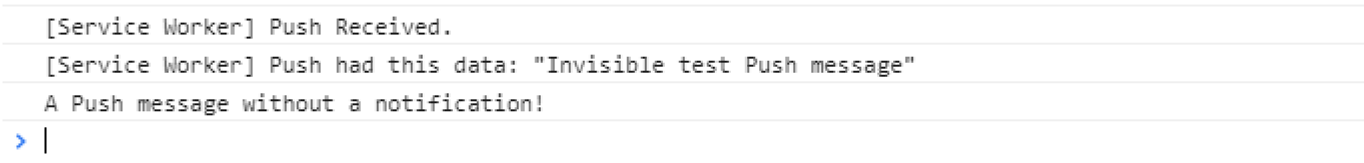

```
[Service Worker] Push Received.
[Service Worker] Push had this data: "Invisible test Push message"
A Push message without a notification!
> |
```

Figure 4.11: Console Printout of an Invisible Push Message

message in the console nor a default notification that the website has updated in the background.

4.2.3 HTTPS

The main reason as to why the service worker requires the communication to be transmitted using SSL/TLS is to protect both the client and the server from the man-in-the-middle (MITM) attacks. During the MITM attack, a third party hijacks the communication and is able to passively view the communication, or even alter the communication without the rightful parties' awareness. This attack is detrimental to the client, as their private information can be used maliciously by the attacker. For example, an attacker could get client's credit card information and steal their money. From the server's point of view, this attack is harmful because once it is revealed to the public that a company communicates with users and handles their data incautiously, it is likely to affect the company's business negatively.

Most of the attacks on HTTPS are more difficult to execute than other attacks presented in this project. This difficulty is largely due to the fact that there are two general attack scenarios - either the attacker impersonates a legitimate server and tricks the client into connecting to them, or the attacker decrypts the communication between the client and the server.

Impersonating a legitimate server is addressed by the first two HTTPS attacks, and the main difficulty with this attack vector is that the attacker needs to prevent the user from connecting to the legitimate server. The attacker is often in the proximity of the victim that is connected to a public wireless access point, e.g., in an airport or a cafe. The attacker pretends to be the access point X, then drops the victim from the legitimate access point X, for example by using Kali Linux and the *airodump* suite. The victim's machine then reconnects to the malicious access point X. Now the attacker, acting as a proxy, can redirect the victim from the legitimate domain to a fake one. If the attacker successfully impersonates the legitimate domain by using an untrustworthy certificate, or if certificate validation is not implemented correctly on the victim's machine, then the victim will not notice that they are on a fake domain.

Decrypting the communication between the client and the server can be done if the Diffie-Hellman key exchange protocol is implemented poorly on the server, e.g., the server uses small-sized keys or weak encryption ciphers. Alternatively, the attacker has a high chance of success if the server uses outdated protocol versions.

Untrustworthy Certificates
HTTPS offers protection from MITM attacks, yet it is not flawless. The client uses certificates to prove the server's identity and encrypt the communication between the client and the server using the public key stated in the certificate. However, an attacker could pretend to be a legitimate server by cryptographically impersonating the server. In theory, this should not be possible because the certificates that prove one's identity online must be issued by trustworthy certificate authorities. In practice, however, there are certificate authorities that are considered reliable, yet they "improperly issue certificates" [25]. An example of this situation is Symantec - one of the biggest certificate authorities - which has been distrusted by Google after a scandal in 2017 [25]. During the writing of this thesis project, Mozilla also made a public statement [26] showing their disdain of Symantec.

Improper Certificate Validation
Another vulnerability of HTTPS lies in the verification of certificates. A famous example of this is the Apple's "goto fail" bug, which resulted in lack of verification of certificates. This bug allowed attackers to "trick users of OS X 10.9 into accepting SSL/TLS certificates that ought to be rejected" [27], which made it much easier to impersonate a legitimate server. Similar problems have been discovered in some Microsoft packages. [34]

Poor Diffie-Hellman Implementation
Even if certificates are issued by trustworthy certificate authorities and adequately verified by the clients, there is still no guarantee that the communication is not being subject to a MITM attack. The security of HTTPS largely relies on the fact that Diffie-Hellman key exchange is secure, meaning that the communication between the client and the server is encrypted. This key exchange protocol requires the client, and the server to each calculate a large prime number and use it for computation of a shared key. However, finding a large prime number is computationally expensive, hence "the overwhelming majority [of servers] use one of a handful of primes." [9] To give the size of the "overwhelming majority," the source [9] states that "just two 512-bit primes account for 92.3% of Alexa Top 1M domains that support DHE_EXPORT, and 92.5% of all servers with browser-trusted certificates that support DHE_EXPORT." There is also evidence that in some cases - e.g., Apache servers - the primes are hard-coded into the server configuration. That means that the communication with these servers can be attacked by a MITM.

Outdated Protocol Versions
Outdated versions of TLS are riddled with vulnerabilities. Because HTTPS is a crucial technology in secure communication, "TLS and its precursor SSLv3 have been the target of a large number of cryptographic attacks in the research community, both on popular implementations and the protocol itself." [10] Apart from the aforementioned attacks, some others that have seen a considerable amount of success are DROWN, "Lucky 13, BEAST, and POODLE" [10]. Importantly, most of the successful attacks target the outdated versions of the protocol - DROWN targets SSLv2, POODLE targets SSLv3, and BEAST targets TLSv1.0.

5 Results

This section covers the vulnerability matrix, which points out the vulnerabilities in the service worker and related technologies and mitigations to these vulnerabilities. Following the vulnerability matrix, the mitigations are presented. The vulnerabilities and mitigations to them are then summarized in the form of security guidelines.

5.1 Vulnerability Matrix

Technology	Vulnerability	Exploitability	Severity	Complexity
Service Worker API	JSONP XSS	easy	severe	moderate
Push API	Social Engineering	average	severe	low
	Push Without Notification	average	moderate	moderate
HTTPS	Untrustworthy Certificate	difficult	severe	high
	Improper Certificate Validation	difficult	severe	low
	Poor Diffie-Hellman Implementation	difficult	severe	high
	Outdated Protocol Versions	easy	severe	moderate

Table 5.3: The Service Worker Vulnerability Matrix

5.2 Mitigations

This section outlines the mitigations to the vulnerabilities presented in the Service Worker Vulnerability Matrix 5.3. The mitigations have been discovered from sources [5–10, 14, 16, 19–31] and experiments with the demonstrator during the work on this paper.

5.2.1 Service Worker API

XSS using JSONP can be avoided in several ways. First of all, JSONP should not be used. Instead, the developers can use Cross-Origin Resource Sharing (CORS), which is a modern substitution of JSONP. It is supported by 95.37% users globally - according to *caniuse.com* - and is a substantially safer alternative to JSONP. Additionally, it is supported in *fetch*, making it easier to use. Secondly, XSS must be prevented, for example by using the guidelines from OWASP Top 10 [29]:

1. Use frameworks that "automatically escape XSS by design" [29], such as Ruby on Rails and React JS
2. Escape "untrusted HTTP request data based on the context in the HTML output" [29]; this resolves Reflected and Stored XSS
3. Apply "context-sensitive encoding when modifying the browser document on the client side" [29]; this resolves DOM XSS

This vulnerbility is shown in Figure 4.7, and to mitigate it, the code in Listing 6 is modified as presented in Listing 10 to serialize the user input using *serialize-javascript* module. Now the HTML characters are escaped, i.e., the payload in Listing 7 is not treated as HTML code, hence the attack will not succeed.

```
function push () {
    //same code as before
    window.localStorage.setItem('name', serialize({name}))
}
function pull () {
    let name = deserialize(window.localStorage.getItem('name'))
    //same code as before
}
```

Listing 10: Safe Handling of User Input

5.2.2 Push API

Social Engineering
From the domain's point of view, in order to prevent social engineering attacks using push notifications, the domain must run extensive security checks on the party that gains access to the application endpoints. Furthermore, educating users that notifications can come from another domain through the same application is advised.

From the client's point of view, the client must consider whether it is necessary to allow notifications from particular domains and stay vigilant even after allowing push notifications. In case of the domain or its notifications starting to act suspiciously, denying push notifications is advised.

Push without notification
During the production of this paper, this vulnerability was partly addressed in a major Chrome update. [31] Figure 5.12 demonstrates that now if the service worker does not trigger the notifications for push events, Chrome notifies the user that a website has updated in the background, as described in [7]. However, this message is not displayed if the user is browsing the website when they receive a notification. Thus, although Chrome 66.0.3359.117 is more protected from this attack than Chrome 63.0.3239.132 and previous versions, this vulnerability is still not fully mitigated.

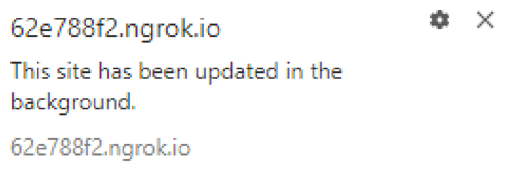
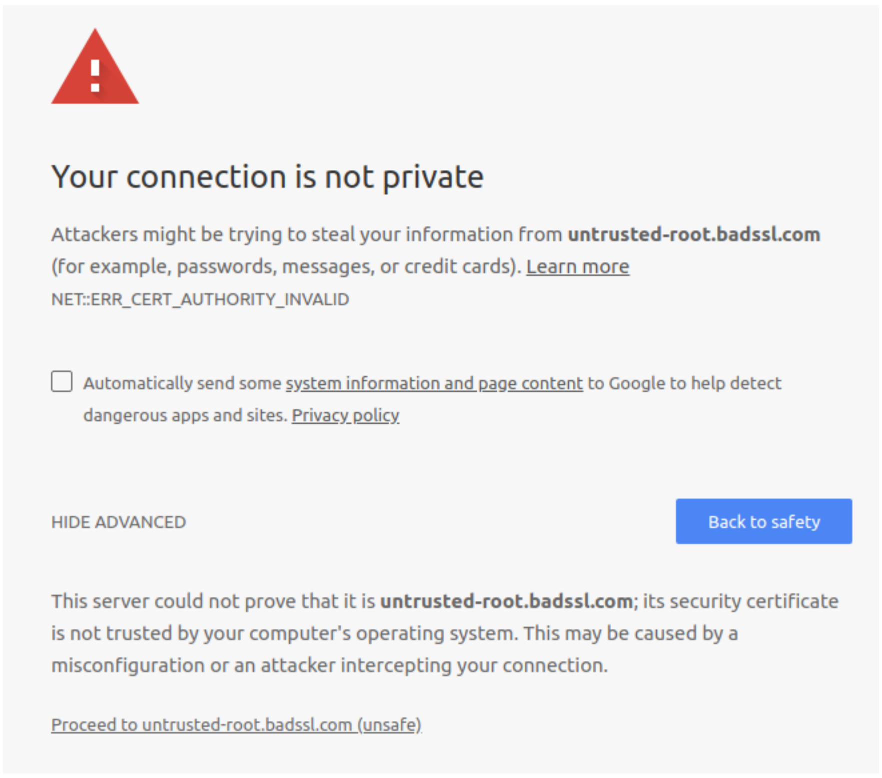

62e788f2.ngrok.io
This site has been updated in the background.
62e788f2.ngrok.io

Figure 5.12: Notification Displayed in Chrome for Misconfigured Service Worker

5.2.3 HTTPS

Untrustworthy certificate
The trustworthiness of a certificate is determined by the browser that navigates to the web page. Therefore the client does not have to do anything - most of the modern browsers block the web page when something is wrong with its certificate; Figure 5.13 demonstrates an error message displayed by Chrome when visiting a website whose certificate has been issued by an untrusted certificate authority. The job of the domain is therefore to obtain a certificate from a trustworthy certificate authority.

As a client, it is difficult to protect oneself from this type of attack. However, the clients should always keep their browsers and operating system updated, since security



Your connection is not private

Attackers might be trying to steal your information from **untrusted-root.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

☐ Automatically send some system information and page content to Google to help detect dangerous apps and sites. Privacy policy

This server could not prove that it is **untrusted-root.badssl.com**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

Proceed to untrusted-root.badssl.com (unsafe)

Figure 5.13: Error Message - Visiting a website With an Untrusted Certificate

patches tend to remove certificates from certificate authorities that have been detected to be untrustworthy. Additionally, the clients should not use sensitive web services, e.g., accessing bank or hospital accounts, when connected to public networks, e.g., in airports or cafes.

Improper Certificate Validation

Improper Certificate Validation occurs primarily due to developers disabling certificate validation during development to make the product work without requesting a valid certificate. In most cases, the issue occurs when the developers forget to re-enable certificate validation before releasing the product. To avoid that, the developers should not disable certificate validation in the first place but instead obtain a certificate specifically for development purposes.

Poor Diffie-Hellman Implementation

As stated in source [9], there are a few steps to be taken in order to use the Diffie-Hellman key exchange more securely:

1. Transition to elliptic curves - while there is some critique of the elliptic curves in cryptography "due to NSA influence on their design" [9], they show no weakness, have shorter keys, and have faster shared-key computations

2. Increase minimum key strengths - the minimum key strength, as suggested in source [9], should be 1024-bit long, the length of 2048-bit should be considered the secure standard

3. Avoid fixed-prime 1024-bit groups - the servers should not use the primes that are easy to compute. Subsequently, if the server has to use fixed primes instead of

computing them, the fixed primes should be updated regularly

Outdated Protocol Versions
Outdated versions of TLS are detrimental to the security of both server and the client. However, the mitigation for this vulnerability is relatively simple - upgrading to the latest version of TLS and not allowing the clients to downgrade the connection to SSL or TLSv1.0.

5.3 Security Guidelines

Table 5.4 is the security guidelines for developing and using Service Worker components safely. The "Actor" in the table is the side that ought to follow the guideline - either the Domain or the Client.

Technology	Guidelines	Actor
Service Worker API	use frameworks	Domain
	escape untrusted HTTP request data	Domain
	apply context-sensitive encoding	Domain
Push API	ensure that any third party with access to endpoints is trustworthy	Domain
	educate users about third-party notifications	Domain
	keep the browser updated	Client
HTTPS	avoid accessing sensitive web services when connected to public hot spots	Client
	use a certificate during development	Domain
	use eliptic curves	Domain
	increase minimum key strength	Domain
	avoid fixed-prime 1024-bit groups	Domain
	use TLS and keep it updated	Domain

Table 5.4: Security Guidelines

6 Discussion

"JavaScript is now [2017] the primary language of the web" [29] Therefore, JavaScript and the technologies that it utilizes are heavily targeted by attackers and are often misconfigured by less experienced developers. Furthermore, the Service Worker is a relatively new technology that is rapidly gaining popularity. The combination of these factors calls for clear guidelines on how to develop service workers in a safe manner.

6.1 Understanding-related Objectives

O1 and **O2** are aimed towards understanding why service workers are important and why they are designed in this way. This section provides the summary of answers to these objectives.

O1 - Understand why an API like Service Worker is desirable from users' and business' points of view. From users' point of view, the service worker offers convenience. The clients can use web resources when they are offline or on slow connections; if they want, they can receive notifications from the domain even when their browser is closed, and other features are currently being developed to add more functionality to service workers. With the success of Progressive Web Applications, it has been shown that offering clients convenience is very beneficial for businesses. Due to its functionality, the service worker is a key part of the progressive web applications, hence it contributes to profits of a business and the improvement of the experience for the clients.

O2 - Understand successes and failures of Service Workers' predecessors and their influence on Service Worker. The only significant predecessor of the service worker that has seen some success is Application Cache. The success of AppCache was the user convenience that it offered - the ability to use web resources offline was much needed. AppCache thrived when it was used for websites that clients used for creating content, such as Google Docs. However, Application Cache was difficult to use for developers. To deploy AppCache on a domain was challenging due to numerous implicit behaviors of the service worker, which caused unwanted behavior for the clients. Both success and failures of AppCache influenced the Service Worker tremendously because it demonstrated that the offline capability for domains was necessary and that the technology that is used to offer this functionality must be highly flexible.

6.2 Security-related Objectives

Objectives **O3 - O6** aim towards understanding security threats to the Service Worker components, classifying these threats, and mitigating them. These objectives are:

- O3 - Understand security threats related to the technologies used by Service Worker
- O4 - Make a vulnerability matrix that outlines existing Service Worker vulnerabilities
- O5 - Make a list of mitigations for discovered vulnerabilities
- O6 - Make a list of security guidelines

This section aims to discuss the artifacts presented in Section 5 that address these objectives.

6.2.1 Security Guidelines

From the Security Guidelines Table 5.4 it becomes clear that it is mostly the provider of the service worker that is responsible for running the service worker safely, the client can do very little to enhance their protection. While it is partly due to the fact that the service worker is installed without the client's awareness, the main reason is that the browser ensures that the service worker is working as intended. Thus, it is crucial that the client's browser is frequently updated.

6.2.2 Service Worker API

"XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two-thirds of all applications." [29] This makes XSS a critical attack to consider, especially taking into account that it can be exploited fairly easily, as there are tools that "can find some XSS problems automatically." [29] However, XSS prevention can be incredibly difficult - if the system has not been designed with security in mind from the beginning, and it cannot be rewritten because it is in production, patching specific XSS vulnerabilities can be challenging. For example, mitigating the vulnerability presented in Listing 6 requires separation of Javascript from the HTML, and either import of a module or additional programming, which can be cumbersome on a medium-sized domain.

During the work on this thesis, the major Chrome update [31], mentioned previously, addressed several security issues with the service worker and related APIs. Two of them are related to bypass of the Same Origin Policy of medium (CVE-2018-6093) and high (CVE-2018-6089) severity. This shows how difficult it can be to mitigate cross-site scripting vulnerabilities.

Although XSS can be challenging to patch, shift from JSONP to CORS can be made with relative ease, and that shift prevents installation of malicious service workers by the MITM. Taking into account that a MITM attack is likely to result in theft of sensitive information, such as credentials and sessions, it is one of the first vulnerabilities that should be addressed.

6.2.3 Push API

Social Engineering
The exploitability of this vulnerability varies depending on the perspective. On the one hand, it can be difficult to execute because to gain access to the endpoints the attacker needs to either gain trust from the company X that serves the application to its clients or steal the endpoints. If X thoroughly checks third-party companies before trusting them with endpoints, then it can be difficult for the attacker to maintain a trustworthy image; theft of sensitive information like endpoints is often difficult. On the other hand, if the attacker somehow obtains the endpoint, then the vulnerability becomes easy to exploit.

Similarly, the complexity of mitigation of this vulnerability varies. It might be difficult to carry out a background check on the third party companies sufficiently in-depth. Furthermore, the only way of detecting malicious notifications from a third party company which has passed the background checks successfully but has gone rogue afterward is by receiving complaints from the clients.

Push Without Notification
Push messages that do not trigger notifications are possible if a malicious service worker has been installed on the client. The results regarding the level of threat of this type of

attack are inconclusive. On the one hand, a push notification does not have a notable impact on battery [28]. Thus, one would expect a push message without notification to have even less impact on battery life of a device. On the other hand, it is important to realize that push messages could be spammed by the attacker and that the service worker can be configured to trigger more resource demanding operations on push events, such as executing code, triggering vibration and sound. The impact of spamming push messages that trigger some expensive computations remains to be seen, as it could not be tested during this project due to the usage of third-party services, such as *Google Push Companion* and *One Signal*, for managing push notifications.

6.2.4 HTTPS

Untrustworthy Certificates allow attackers to impersonate legitimate servers. Improper Certificate Validation is a similar scenario in which the attacker's certificate has an even higher chance of being trusted by the client. As described in section 4.2.3, impersonating attacks are challenging to execute, but when the attacker succeeds their impact is tremendous. The primary concern with service workers is that the attacker installs a service worker that spans a legitimate domain and then uses it to eavesdrop and steal client's information.

Since the clients do not have explicit mechanisms of defending themselves on public networks, the attack is difficult to protect against. However, taking into account the difficulty of executing these attacks properly and the inability of a domain to prevent these attacks entirely, they should not be the primary concern of the domain.

Poor Diffie-Hellman Implementation

Although "the common practice of using standardized, hard-coded, or widely shared Diffie-Hellman parameters" [9] reduces the cost of cryptanalysis of Diffie-Hellman, it is important to recognize that the cost of this cryptanalysis is still substantial. For the attack to be worth the cost, it has to be done on a large scale - e.g., National Security Agency (NSA) breaking the cipher to surveil the internet - or target the most successful businesses. This means that a small or medium-sized business is unlikely to face this attack from competitors.

Mitigating this vulnerability can also be costly. While it is possible to use a fixed prime on the server but increase its length, better mitigations require an increase in computational power (using dynamically calculated primes) or partial redesign of the system (using elliptic curves).

Outdated Protocol Versions

Starting from 31st June 2018, all versions of SSL and TLSv1.0 are prohibited by the Payment Card Industry (PCI) Security Council [19]; industries that involve managing card information are required to use TLSv1.1 or TLSv1.2. However, while SSL and TLSv1.0 are banned in the card industry, they can still be used in other areas, which calls for patching the existing vulnerabilities in TLS protocols.

The attacks on the outdated SSL and TLS protocols are some of the easiest covered in this paper - some software tools can test the vulnerability of the server to certain attacks after a single click. Because of that, making sure that the protocols are being kept up-to-date is extremely important for any domain.

7 Conclusions and Future Work

The primary goal of this thesis was security examination of the service worker. Considering that the amount of information available about this topic is very limited, the service worker appeared to be incredibly unsafe to use. However, during the work on this project it became clearer that since the service worker was designed recently, it was made with many security features in mind. Furthermore, the service worker is run through a browser, and most of the modern browsers can handle the service workers safely. The weakest point of the service worker technology is cross-site scripting using JSONP, which can be difficult to remediate, but it is not impossible. The second-worst vulnerability is outdated protocol versions because they are easy to exploit and they lead to severe consequences. However, the mitigation for this vulnerability is relatively inexpensive and straightforward.

While this project took into consideration man-in-the-middle attacks and malicious domains, it did not consider situations in which the client attempts to perform some unauthorized action using the service worker. Therefore, in future, it is worthwhile investigating if the client can modify the installed service worker to gain unauthorized access to a domain's resources.

References

[1] D. Eldridge. (2017, Oct. 5) To launch a pwa, forbes had to change its culture first. NAPCO Media. Accessed: 25.03.2018. [Online]. Available: http://www.pubexec.com/article/forbes-progressive-web-app-supercharged-mobile-engagement-revenue/

[2] (2017, May 17) Twitter lite pwa significantly increases engagement and reduces data usage. Google Developers. Accessed: 13.04.2018. [Online]. Available: https://developers.google.com/web/showcase/2017/twitter

[3] B. McLain. (2017, Sep. 8) Callback hell. Accessed: 06.03.2018. [Online]. Available: http://blog.mclain.ca/assets/images/callbackhell.png

[4] B. Hoffman. (2014, Dec. 9) Ssl performance diary #4: Optimizing the tls handshake. Rigor, Inc. Accessed: 25.03.2018. [Online]. Available: https://zoompf.com/blog/2014/12/optimizing-tls-handshake/

[5] M. Gaunt. (2018, Mar. 29) Service workers: an introduction. Google Developers. Accessed: 10.04.2018. [Online]. Available: https://developers.google.com/web/fundamentals/primers/service-workers/

[6] J. Kallin and I. L. Valbuena. (2013) Excess xss. Chalmers University of Technology. Accessed: 30.04.2018. [Online]. Available: https://excess-xss.com/

[7] M. Gaunt. (2018, Jan. 3) Push events. Google Developers. Accessed: 12.04.2018. [Online]. Available: https://developers.google.com/web/fundamentals/push-notifications/handling-messages

[8] J. Archibald. (2017, May 12) Service worker security faq. The Chromium Project. Accessed: 17.04.2018. [Online]. Available: https://dev.chromium.org/Home/chromium-security/security-faq/service-worker-security-faq#TOC-Why-doesn-t-Chrome-prompt-the-user-before-registering-a-Service-Worker-

[9] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 5–17. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813707

[10] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS with SSLv2," in *25th USENIX Security Symposium*, Aug. 2016. [Online]. Available: https://drownattack.com/drown-attack-paper.pdf

[11] A. Shreve. What is ngrok? ngrok.com. Accessed: 13.04.2018. [Online]. Available: https://ngrok.com/product

[12] (2018, Mar. 31) Using promises. Mozilla Developer. Accessed: 06.03.2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

[13] (2018, Apr. 10) Using fetch. Mozilla Developer. Accessed: 23.04.2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[14] M. Scales. (2018, Jan. 3) Using the cache api. Google Developers. Accessed: 25.03.2018. [Online]. Available: https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/cache-api

[15] M. Gaunt and A. Osmani. (2015, Nov. 17) Instant loading web apps with an application shell architecture. Medium. Accessed: 15.04.2018. [Online]. Available: https://medium.com/google-developers/instant-loading-web-apps-with-an-application-shell-architecture-7c0c2f10c73

[16] (2018, Feb. 15) Push api. Accessed: 25.03.2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Push_API

[17] K. Basques. (2018, Jan. 11) Why https matters. Google Developers. Accessed: 20.03.2018. [Online]. Available: https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https

[18] J. Archibald. (2012, May 8) Application cache is a douchebag. A List Apart. Accessed: 26.03.2018. [Online]. Available: https://alistapart.com/article/application-cache-is-a-douchebag

[19] L. K. Gray. (2017, Jun 30) Are you ready for 30 june 2018? saying goodbye to ssl/early tls. The PCI Security Standards Council. Accessed: 01.05.2018. [Online]. Available: https://blog.pcisecuritystandards.org/are-you-ready-for-30-june-2018-sayin-goodbye-to-ssl-early-tls

[20] A. Russell, J. Song, J. Archibald, and M. Kruisselbrink, *Service Workers 1*, W3C Working Draft, Nov. 2 2017. [Online]. Available: https://www.w3.org/TR/service-workers-1/

[21] P. Beverloo, M. Thomson, M. van Ouwerkerk, B. Sullivan, and E. Fullea, *Push API*, W3C Working Draft, Dec. 15 2017. [Online]. Available: https://www.w3.org/TR/push-api/

[22] M. Gaunt. (2018, Jan. 3) Common notification patterns. Google Developers. Accessed: 12.04.2018. [Online]. Available: https://developers.google.com/web/fundamentals/push-notifications/common-notification-patterns

[23] S. Larsen. (2016, Jun. 17) Xss persistence using jsonp and serviceworkers. Accessed: 27.04.2018. [Online]. Available: https://c0nradsc0rner.wordpress.com/2016/06/17/xss-persistence-using-jsonp-and-serviceworkers/

[24] N. Ahuja, "Review on cross site scripting," in *International Conference on Recent innovations in Sciences, Management, Education and Technology*, ser. ICRISMET-16. Conference World, 2016, pp. 891–895. [Online]. Available: http://data.conferenceworld.in/ICRISMET/P891-895.pdf

[25] C. Doctorow. (2017, Mar. 24) Google: Chrome will no longer trust symantec certificates, 30% of the web will need to switch certificate authorities. Boing Boing. Accessed: 10.04.2018. [Online]. Available: https://boingboing.net/2017/03/24/symantec-considered-harmful.html

[26] K. Wilson. (2018, Mar. 12) Distrust of symantec tls certificates. Mozilla. Accessed: 10.04.2018. [Online]. Available: https://blog.mozilla.org/security/2018/03/12/distrust-symantec-tls-certificates/

[27] P. Ducklin. (2014, Feb. 24) Anatomy of a "goto fail" – apple's ssl bug explained, plus an unofficial patch for os x! Sophos. Accessed: 10.04.2018. [Online]. Available: https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/

[28] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 35–45.

[29] A. van der Stock, B. Glas, N. Smithline, and T. Gigler. (2018, Feb. 4) Owasp top 10 - 2017: The ten most critical web application security risks. OWASP Top 10. Accessed: 17.04.2018. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project

[30] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 66–77. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660275

[31] A. Syed. (2018, Apr. 17) Stable channel update for desktop. Google Chrome. Accessed: 08.05.2018. [Online]. Available: https://chromereleases.googleblog.com/2018/04/stable-channel-update-for-desktop.html

[32] uve. (2017, Jul. 25) A beginner's guide to service workers. Samsung Internet Developers. Accessed: 13.04.2018. [Online]. Available: https://medium.com/samsung-internet-dev/a-beginners-guide-to-service-workers-f76abf1960f6

[33] T. Ater. (2016, Jan. 19) Building offline sites with serviceworkers and upup. Opera Software ASA. Accessed: 13.04.2018. [Online]. Available: https://dev.opera.com/articles/offline-with-upup-service-workers/

[34] J. Yu and H.-K. Choi. (2017, May 12) Improper certificate validation. Snyk. Accessed: 01.05.2018. [Online]. Available: https://snyk.io/vuln/SNYK-DOTNET-MICROSOFTASPNETCOREMVCAPIEXPLORER-60110