# Hardened Browser

Pol Monedero

**Resum–** L'augment en l'ús de telèfons intel·ligents al llarg dels anys ha canviat el focus del desenvolupament de programari d'aplicacions natives i especifiques a una plataforma, a aplicacions en línia, i posteriorment, Aplicacions Web Progressives. Encara que aquest tipus d'aplicacions web proporcionen molts dels beneficis de les aplicacions natives a una fracció del cost, la seva seguretat ha de ser freqüentment actualitzada per prevenir atacs específics. La necessitat de ser proactiu en els aspectes de seguretat pot afluixar el ritme de desenvolupament, eliminant alguns dels avantatges de les Aplicacions Web Progressives. Aquest projecte té com a objectiu desenvolupar una aplicació nativa capaç de mostrar apps web de manera simple i segura, beneficiant tants als desenvolupadors com als usuaris. Els desenvolupadors poden relegar la seguretat de l'app al Hardened Browser, i els usuaris tindran una experiència fluida i lliure de vulnerabilitats.

**Paraules clau–** Aplicacions Web Progressives, navegador web, seguretat web, aplicacions natives, desenvolupament de programari mòbil, proves de seguretat, ciberseguretat.

**Abstract–** The increase in smartphone usage across the years has changed the focus of software development from native, platform-specific apps to online applications, and subsequently, Progressive Web Apps. Although these types of web apps provide many of the benefits of native apps at a fraction of the cost, their security must be frequently updated to prevent attacks. The need to be proactive on the security aspects of the app can slow down new features and development, which removes some of the advantages of Progressive Web Apps. The project aims to develop a native application capable of displaying these types of apps in a simple and secure form, benefiting both the developers and the users. Developers can leave the security aspects of the app to the Hardened Browser, and users will have a fluid experience free of vulnerabilities.

**Keywords–** Progressive Web Apps, web browsers, web security, native apps, mobile software development, security testing, cybersecurity.

## 1 Introduction

### 1.1 State of the art

SINCE the introduction of smartphones to a wider audience in the 2010s, web traffic from these devices began to rise from a 10% on 2012, to a 59% on 2022[1]. With such a rise in traffic, companies have changed the focus of their efforts from native, platform-specific apps to web apps. Web apps have a much wider reach, as any device with a web browser can access, and are much cheaper and faster to make, at the cost of being less capable overall. On the other hand, platform-specific applications are much more integrated into the system. They can read and write files from the local file system, access hardware via USB or Bluetooth, and interact with data stored on your device directly. Ideally, we would want the reach of web apps and the capabilities of platform-specific apps. In 2015, Alex Rusesell, a Google Chrome engineer, and Frances Berriman, a designer, coined the term **Progressive Web App (PWA)[2]**.

PWAs are web apps that have been designed to be capable, reliable and installable, so as to resemble the capabilities of platform-specific apps. Thanks to new and upcoming APIs for the web, capabilities like file system access, media controls or full clipboard support are no longer restricted to native applications. Sites like `https://whatwebcando.today/` track these advances, and at a glance, we can see just how many features are already available. Due to the advantages of PWAs, many companies have started to use this technology instead of native apps. For example, Twitter released Twitter Lite in 2017, which increased tweets sent by 75% and decreased bounce rate by 20%[3]. Hulu replaced their platform-specific desktop app

• E-mail de contacte: Pol.Monedero@autonoma.cat
• Menció realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Jordi Ventayol Marimon (DEIC)
• Curs 2022/23

in 2019 with a PWA and saw a 27% increase in return visits[4]. With an increase in the usage of PWAs, we have to take into account how secure they are. As with any web app, the security is mostly handled by the browser itself, which means a PWA is only as secure as the browser is. Due to the importance of security in today's landscape, we will be studying how safe PWAs are in relation to native apps, and if the security of these can be improved.

## 1.2 Objectives

The main goal of this project is to build a hardened mobile browser using Build38's security library, T.A.K, and any other security measures necessary for a safe browsing experience. The browser will be an android application made from the ground up, using the android web browser engine component, WebView. Before any development can be started, however, we need to have a better understanding of how PWAs, and apps in general, work.

Firstly, we will need to make a general comparison between native apps and PWAs to know more about the strengths and weaknesses of this new technology. Once we know how PWAs stack up against native apps in a broad sense, we can analyse the possible threats for each platform and devise countermeasures or mitigations. With a list of threats for PWAs and possible ways to counteract them, we will have to determine which of these can be solved with T.A.K. If a vulnerability cannot be handled by the library, we will have to study how, or if it is feasible, to implement a solution. The principal security features we think should be implemented are: to make an integrity check when the browser starts to ensure the application hasn't been tampered with locally, secure all backend channels to prevent disclosures of confidential information, securely store all confidential or sensitive data and server authentication.

## 1.3 Methodology

This paper consists of two main parts, comparing PWAs and native apps to each other, and developing a secure app from which to display PWAs. To make the best app possible, we needed a solid knowledge base about both native apps and PWAs. To research how PWAs came to be and how they are developing, we focused on one of the main supporters of this technology, Google. Their `web.dev` web page displays a plethora of articles and research outlining the benefits of PWAs, their inner workings, and their faults, from which we built a starting foundation of the most important facts about PWAs. We also needed to study the T.A.K library documentation, from which we had to extract the capabilities of the library and how it could help in our development. After in-depth research about native apps, PWAs and T.A.K, we started development. To build the app, we used the Android developer guides and the T.A.K documentation in order to understand and properly implement every capability necessary in an adequate timeline.

## 1.4 Planification

To be able to complete the project in the allotted time frame, we have followed the waterfall methodology. With this methodology, we can focus on each phase of the project at a time individually, and due to the fixed timeline on the project, ensure we are up to date with the planning in a simple way. To accomplish the aforementioned objectives, we will need a work schedule to match the expected hours needed for the project. Without taking into account the time required to make the paper and other documentation, we expected a total of 185h to complete the project. To better outline this planning, we have provided a Gantt diagram in appendix A.1.

# 2 Native Apps vs. PWAs

As we have discussed previously, native apps and PWAs both have different characteristics which make them a better choice depending on what we want from our application. In this section, we will note these key differences to find exactly where each one excels at.

## 2.1 General comparison

- Language
  - Native apps are developed with the languages of each platform (Java/Kotlin for Android, Objective-C/Swift for iOS). PWAs must use HTML, CSS, JavaScript and any other frameworks available for web development.
- Cost
  - For a native app, developers need to learn the language and build a version for each platform, which means more development time to first build the app and to update it afterwards. PWAs are by nature cross-platform, which means we only need a single codebase. Faster to build and update.
- Developer Convenience
  - To deliver a native app to the client, we will normally have to submit it to an app store. For each store, we will need to pass the requirements specified and pay a fee. Every time we need to make an update, we will need to do so on every platform's app. For PWAs, the user only needs a web browser and the URL pointing to the app, skipping any requirements. Developers only need to update a single project.
- User Convenience
  - Users need to commit to downloading a native app. In order to receive new updates, they might need to update it manually in the app store. PWA users can add the app to the home screen without downloading it. Updates are always seamless.

- Performance
  - Native apps have direct access to the operating system, making them efficient and fast. PWAs run from a browser, which means an extra layer of abstraction. Overall, PWAs have more latency and battery consumption.
- Capabilities
  - Native apps have full access to all the features of the device, such as geofencing (defining geographic areas and receiving notifications when the device enters or leaves it) and NFC. PWAs don't have full access to the device. Some missing capabilities are NFC, interaction with other apps, proximity sensors and ambient light detection.
- Security
  - Native apps can be secured with login functionality, multi-factor authentication, certificate pinning, mutual authentication and more. PWAs are required to run under HTTPS, which ensures that the connection between the client and the server can't be tampered with, but are much more limited if we want a hardened app, as we rely on the browser for security features.

## 2.2 Security comparison

Now that we have a better idea of what native apps and PWAs can do, we can focus on the security aspect of each one.

### 2.2.1 Native app security

Native apps are generally more secure than PWAs. This is due to having access to platform-specific security features which can't be used by PWAs. We can implement some of the best security practices by following the android documentation.[5]

- **Use TLS:** Any backend connection must be secured.
- **Certificate pinning:** Restrict which certificates are considered valid. Should only be used if we want maximum security, as changes in the server such as moving to another Certificate Authority (CA) will make the app unable to connect to the server without receiving a software update.
- **Client authentication:** Check the client certificate on the server to control access.
- **Use WebView object carefully:** WebView objects should only let users navigate to allowed sites, and JavaScript should be disabled by default.
- **Store data safely:** Store all private user data safely

### 2.2.2 PWA security

Due to PWAs being run inside a browser, most of the security in the app comes from the browser itself, which we can not modify freely. Although what we can do is limited, some of the main aspects of PWAs can be secured.

- **Use HTTPS:** Every connection must be secured.
- **Use service workers:** Service workers are scripts that allow the interception and management of network requests. They provide security by not having direct access to the DOM or cookies, and not being able to read or set prohibited headers.
- **Using the manifest file:** The manifest file provides information about the app, such as the name, description, icons etc... If a manifest file is set, it can not be modified, which will make the name, description and icon of the PWA secure.
- **Secure data:** Secure data locally to prevent Cross-site scripting (XSS)

As we have seen, native apps have more security options, with which we can customize much more how to protect our app. In PWAs, we primarily rely on the browser, which can make the app vulnerable to common web browser attacks, such as XSS or Cross-Site Request Forgery (CSRF). Nevertheless, modern web browsers use sandboxing, providing an additional layer of security for our devices.

After comparing the characteristics of both native apps and PWAs, and enumerating their strengths and drawbacks, we can choose when to use each one. On the one hand, we should use PWAs if we want a simple app that the user does not have to download or update, and we don't want to commit to a cross-platform development. In exchange for the ease of use and development, we will sacrifice some performance and capabilities. On the other hand, we should use native apps if we need specific capabilities like geofencing or NFC, or if we want a more secure/faster application, at the expense of a longer and costlier development.

# 3 Threat Analysis

With some of the key ways of protecting native apps and PWAs, we can now focus on our specific app. It will be a hybrid Android app, using a native app shell that displays a WebView, which will connect to a PWA. As such, we will need to take into account the security of both native apps and PWAs.

To build the app in the most secure way possible, we will be following the methodology commonly used for threat analysis.

## 3.1 Assets

Firstly, we need to list which assets we have worth protecting.

- **User data**: Data should be safely stored and transmitted.
- **Software**: We want a robust app which will prevent and/or detect tampering.

- **Hardware**: The user's hardware should be sandboxed from the app.
- **Remote services**: Connecting to remote services such as the PWA should be safe.

## 3.2 Threats

With the assets to protect, we can enumerate the possible threats to our app.

- **Unauthorized access to data**
- **Insecure data flow**
- **App tampering**
- **Denial of Service**

## 3.3 Threat Matrix

With both the assets to protect and threats to be aware of, we can build a matrix to highlight which we should prioritize.

| Asset/Threat (Impact/Prob.) | User data | Software | Hardware | Remote services |
|---|---|---|---|---|
| Unauthorized access to data | B/B | C/D | D/D | B/D |
| Insecure data flow | A/A | B/B | D/D | B/C |
| App tampering | C/C | B/C | C/D | D/D |
| Denial of Service | D/D | D/D | D/D | B/B |

**Impact:** A: Critical B: High C: Moderate D: Low
**Probability:** A: Very Likely B: Likely C: Unlikely D: Very Unlikely

As we can see on the threat matrix, we should be particularly careful about data transfers, since user data can be very sensitive and is vulnerable at multiple points in its life cycle.

## 3.4 Specific Threats

Once we have highlighted the problems we should be focusing on, we can extract the specific, actual threats we need to be aware of in the app.

- Common web-based attacks such as XSS or CSFR
  - We can disable JavaScript in the WebView and reject any requests not in a whitelist of sites.
- Unencrypted traffic
  - To ensure every connection is secure, we will have to forbid any non-secure connection, such as HTTP.
- Connecting to the wrong server
  - To ensure the connection is to the correct server, we can implement certificate pinning, which will let us restrict which certificates are considered valid.
- App tampering
  - To prevent tampering, we will need to make a check to ensure the app has not been modified.
- Device tampering (Rooting, debugging, hooking)
  - To check for a tampered device, we will need to use tools which make it possible to detect it. We will need to prevent and detect the execution of custom code in our app and if the app is being debugged to prevent tampering.
- WebView
  - To ensure the WebView is as secure as possible, we will need to follow some important directives[6], such as keeping JavaScript disabled unless necessary, including JavascriptInterface (an interface which allows Javascript in a WebView to run native code in the app). Disabling Javascript will prevent many popular web-based attacks. Disabling access to resources not in the scope of our app should also be considered. To further enhance security, we should enable safe browsing[7], a google tool that analyzes and monitors the security of websites, and presents a warning if marked as insecure. Disabling metrics collection and clearing the cache and session storage periodically will improve security as well.

## 3.5 Features implementation

With a list of the main threats we need to mitigate, we can study how are we going to implement the countermeasures. As we have presented previously, we will be using the security library T.A.K by Build38 to increase the security and reliability of the app. As such, some of the threats enumerated can be mitigated by this library.

- Common web-based attacks such as XSS or CSFR.
  - We can use the WebViewClient function `shouldInterceptRequest` to intercept a request and check if it's connecting to a whitelisted URL.
- Unencrypted traffic
  - T.A.K implements the **Secure Channel** feature, which provides a way to communicate with the backend securely.
- App tampering
  - T.A.K implements integrity checks. By registering each individual instance of the app in the T.A.K Cloud when it's first installed, we can periodically use a T.A.K function to ensure the app has not been modified.

- Device tampering (Rooting, debugging, hooking)
  - T.A.K implements functions to analyze whether the device is rooted or not. Additionally, T.A.K implements tools to detect if the app has been debugged and/or hooked.
- WebView
  - Although JavaScript is disabled by default, we can also use `setJavaScriptEnabled` to choose if we want to enable it or not (although some PWAs need JavaScript in order to work). To disable access to local files, we can use `setAllowFileAccess`. On some very old android versions, access to local files from external sources is enabled, which we can disable with `setAllowFileAccessFromFileURLs` . To enable safe browsing, we have to add `EnableSafeBrowsing` on the Android Manifest. To disable metrics collection, we have to add `MetricsOptOut` to the Android Manifest. To clear the cache and session storage we can use the `WebStorage deleteAllData` function and the `CookieManager` functions `removeAllCookies` and `flush`.

With a list of the capabilities of our app, and how we will be implementing them, we can start development.

# 4 Results

## 4.1 Browser Implementation

As stated previously, the objective of the project is to develop a Hardened Browser which will serve as a platform for PWAs, proving an easy way for a developer to build a PWA without compromising on the security aspects, overall giving the user a better experience.

In order to understand how the browser works internally, we have made a diagram.

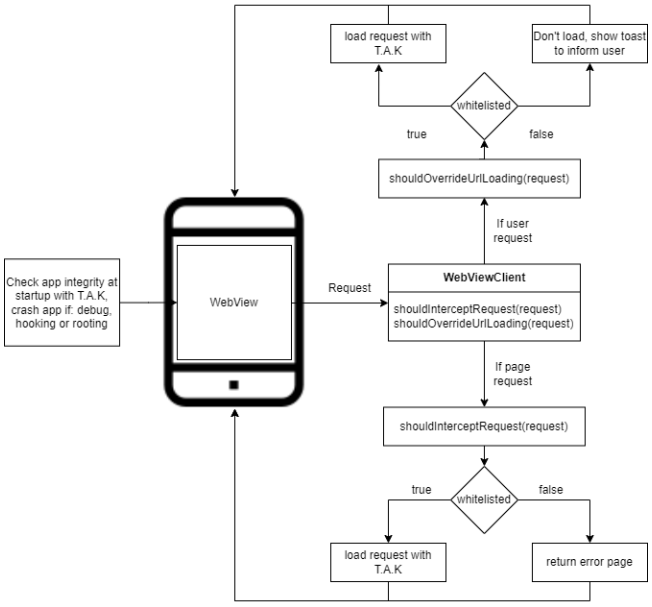



Fig. 1: Overview of the hardened browser app

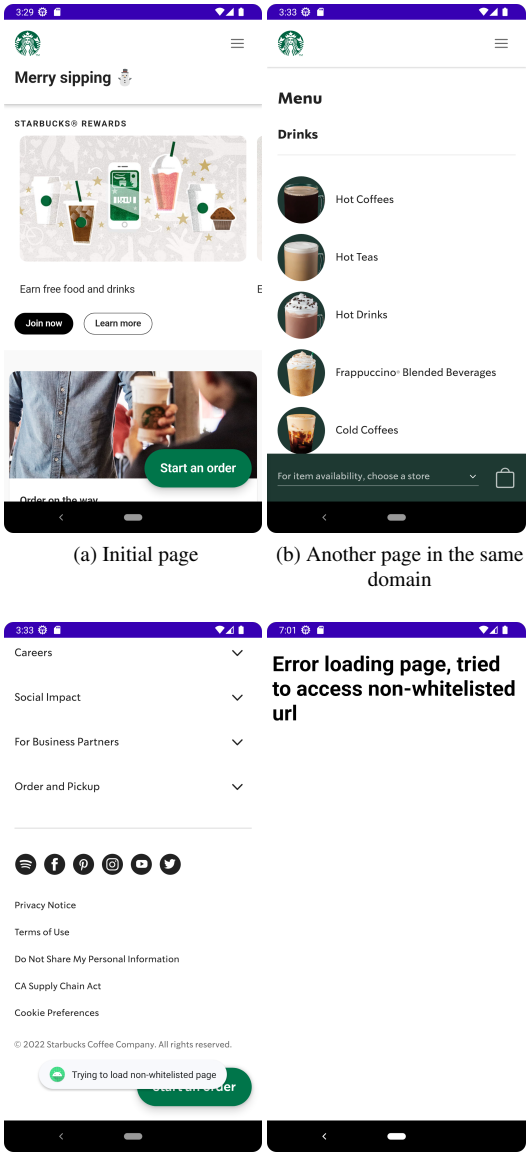
The WebView is the most important component of our app. The WebView class extends the Android View class, which allows us to display web pages in our app. It is not a fully-fledged web browser and lacks some functionality such as an address bar or navigation controls. With minimal configuration, it is very simple to connect and display a PWA in our application via a WebView. We only need to instantiate it and call the `loadUrl` method, which will load the page provided. For our purposes, however, we need to modify the WebView class in order to implement most of the features needed. We will have to modify two classes, the WebView class itself, mainly for enabling and disabling flags, and WebViewClient, which is the core of the WebView, performing most of the work, such as handling requests, key events, errors etc. Particularly, the most important function of the class for our use case is the method `shouldInterceptRequest`, which lets us intercept every request made from the WebView. This is very useful because we can restrict which requests we allow at the lowest level, and even modify how these requests are performed.

Looking back at the threats, we can mitigate four of them by making some modifications to both classes. For the main WebView, we can some change flags to improve the security, such as disabling JavaScript (already disabled by default, but necessary for many PWAs), disabling the use of the cache, disabling mixed content (safe origin loading resources from insecure origins) or enabling safe browsing, a google service that warns about possibly dangerous sites when attempting to connect. As for the WebViewClient, firstly we can override the `shouldOverrideUrlLoading`, which is called every time the user clicks on a link. As it receives the URL of the requests, we can compare it against a whitelist and permit the requests or deny them accordingly, warning the user that they tried to connect to a non-whitelisted page.

Secondly, `shouldInterceptRequest` is called every time the WebView fires any request. As such, we can similarly deny requests targeting non-whitelisted pages. Because we are intercepting the requests before they are completed, we can modify how these requests will be completed. Using the Secure Channel functionality from T.A.K, we can mitigate many threats, such as verifying the identity of the server to prevent Man-in-the-middle attacks (MITM) using certificate pinning and ensuring a secure connection to the server.

To better understand how we can filter requests, we provide below an example in pseudo-code and images detailing its usage.

In these images, we can see how the app works. When we open it, we will be directed to a webpage that has been preprogrammed on the app by the developer. We can navigate this page as we would on any other browser. If we try to click on external links which are not included in the whitelist, a message will warn us about it and the request will be rejected. If the webpage tries to redirect us to a site not whitelisted, it will show an error page, from which we will have to press the back button to go back to the site.




(a) Initial page

(b) Another page in the same domain

(c) User clicking on a page not whitelisted or without a certificate pinned

(d) Trying to load a non-whitelisted link

Fig. 2: How the app loads in different circumstances

```
def shouldInterceptRequest(request):

    #If the request is not in the whitelist, return error response
    if shouldBlockNetworkRequest(request):
        return error.html

    #Some requests, such as file://, should not be handled by T.A.K
    if takShouldHandleRequest(request):
        # Build new request based on previous request
        newRequest = RequestBuilder(request.URL)
        # Make the request and build response with T.A.K secure
        # channel. As it implements certificate pinning, only
        # will finsish  whitelisted request
        response = takSecureChannel.Request(newRequest)

    return response
```

Listing 1: Pseudocode of shouldInterceptRequest function

In this pseudo-code, we have the simplified logic of the app. If we detect that a request is not on the whitelist, we return the previously mentioned error page. Next, if T.A.K should handle the request (most likely a page or web content), we override the WebView to use the library's **SSLSocketFactory** implementation, instead of the standard Android one. In appendix A.2 we provide the full code of the most important parts.

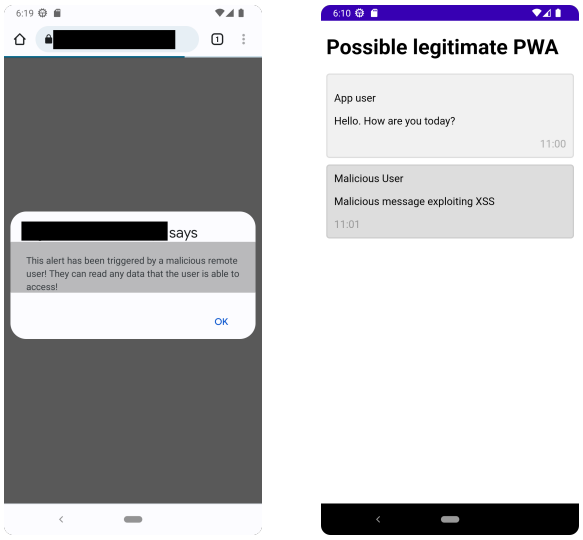
## 4.2 Security testing

In order to prove our app has properly implemented the security features proposed, we will need to test the app. We will be testing the most important capabilities implemented, and comparing how the hardened browser compares to Google Chrome, the world's most used internet browser [8].

### 4.2.1 XSS and CSFR

We will start by trying to perform a Cross Site Scripting (XSS) attack, which would allow an attacker to read any information on the site to which the user has access, carry out actions impersonating the user, or capture credentials. XSS protection relies for the most part on the website itself and how secure its user input parsing is. In our testing, we have made a simple HTML application showing two messages, one of which has an embedded script, which may get executed on the user's machine. In this case, it only shows an alert informing us about the execution of arbitrary code. Even though this example could be safely patched, as it is a simple stored XSS attack[9], there are many more ways to perform XSS. For example, the Open Web Application Security Project® (OWASP) maintains a Cheat Sheet with over a hundred different ways to implement this attack[10]. Without robust and constant enforcement of security protocols, PWAs might be vulnerable to this attack, which may put into risk sensitive or confidential user information.

As we can see in the figures, opening the website with Chrome will show an alert, which has been programmatically called by the malicious message. The Hardened Browser does not show any alert, meaning no javascript has been executed on the host machine, as it is, by default, disabled on the app. This demonstrates Chrome is vulnerable to XSS attacks by itself, meanwhile, the Hardened Browser can mitigate faults in the website security, ultimately protecting the user.



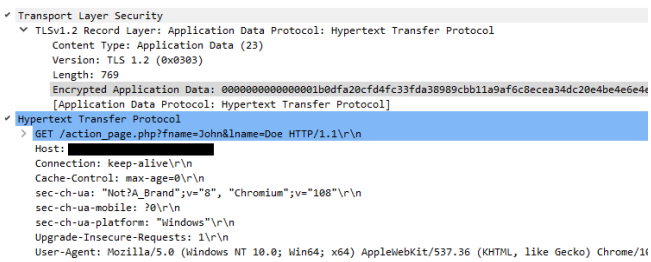

(a) Google Chrome (b) Hardened Browser

Fig. 3: Both browsers connecting to the website

### 4.2.2 Unencrypted traffic

A very important feature of any secure application is that data transmitted or received must be encrypted. On the Web, the main protocol for fetching resources is HTTP[11]. By default, HTTP does not encrypt traffic and is vulnerable to packet sniffing (a technique where an attacker can detect and observe the data flow in a network), which may put in danger sensitive user information. Although HTTP has been phasing out of the Web[12] in favour of HTTPS, which implements a TLS/SSL layer on top, many websites today may offer their contents on HTTP, in turn putting the user's data in danger. As an example, we have built a simple HTTP and HTTPS site with a form which will make a GET request, to compare the differences using Wireshark, a packet analyzer.


(a) HTTP request, which shows confidential information in plain text



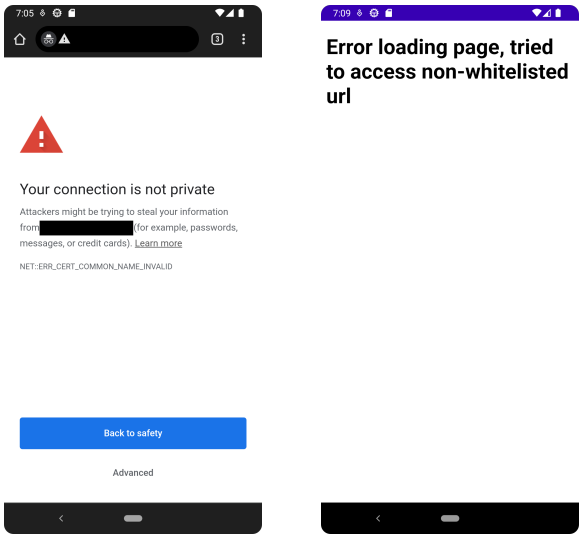

(b) HTTPS request, which shows the confidential information encrypted

Fig. 4: Both browsers connecting to the website

As we can see in the figures, the HTTP request broadcasts personal information in plain text, and any malicious actor could get access to it by sniffing the network. In contrast, the same website but serving HTTPS, sends a TLS packet, with the encrypted data in it (to confirm that both have the same data, we have decrypted it). The Hardened Browser directly disallows HTTP connections, as the WebView component doesn't allow unencrypted traffic, and the T.A.K library enforces certificate pinning, which requires HTTPS. These measures make the Hardened Browser impervious to packet sniffing, protecting the user's data and privacy.

### 4.2.3 MITM attacks

The Man-in-the-middle (MITM) attack consists in intercepting the traffic between two systems. Using this technique, an attacker may impersonate a website, altering the traffic by posing as the user in front of the server, and as the server in front of the user. This means the attacker can see all information transferred between both devices, as they are in the middle. To test the Hardened Browser against MITM attacks, we have deployed a DNS server in which we will change the route of the content we are trying to access to a possibly malicious one, which could cause the user to supply information to the malicious actor thinking it was the real website.




(a) Trying to access the site with Chrome (b) Trying to access the site with the Hardened Browser

Fig. 5: Both browsers connecting to the website via a poisoned DNS

In the figures, we can see that both browsers detect we are trying to connect to a site which doesn't have the certificate matching the domain and shows a warning. Crucially, Chrome still allows the user to bypass the warning, which is a security risk in case the user is not aware of what it entails. The Hardened Browser, due to implementing certificate pinning, refuses the connection as it can't verify the identity of the server. Although this attack normally requires access to critical components in the infrastructure (a router or access to DNS records), badly designed sites may make it easier to execute it, such as using an HTTP to HT-

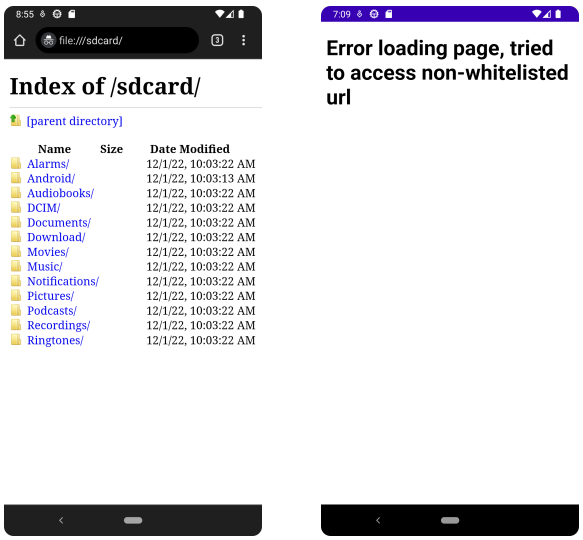
TPS (301) redirect instead of the Strict-Transport-Security response header[13] (HSTS). This type of misconfiguration would allow an attacker to perform a MITM attack, using tools like sslstrip[14]. As we can see, the Hardened Browser minimizes the possibility of a MITM attack, and defends the users against it, while Chrome only warns the user, which could be dangerous if the user ignores it.

#### 4.2.4 App tampering

The threats not covered under the WebView fall under the native app threats, such as rooting the device and debugging or hooking the app, which could undermine the mitigations performed on the WebView side. We have included a root check with a T.A.K function at startup, which can detect if a device is rooted with some accuracy, although false positives and negatives are possible. We can also configure T.A.K to crash when checking the integrity of the device at startup if tampering is detected, which would prevent any debugging and/or hooking.

#### 4.2.5 WebView Hardening

As stated when specifying the features of the browser, we can improve the WebView's security by enabling some flags in its configuration. Two of the flags we have enabled are very important for our browser's security; disabling javascript and local file access. In the XSS and CSFR section, we have seen the importance of having JavaScript disabled; it makes it much harder to exploit vulnerabilities. In the case of local file access, we don't want the browser to have access to anything that is not the web page we are trying to access, as it would open up other paths for vulnerabilities.




(a) Trying to access local files with Chrome

(b) Trying to access local files with the Hardened Browser

Fig. 6: Both browsers connecting to local files

As we can see on the screenshots, Chrome allows browsing the local file system, while the Hardened Browser disallows it, which prevents websites from accessing local files, enforcing a higher security standard.

## 5 Conclusions

The increase in smartphone usage across the years has changed the focus of development from native, platform-specific apps to web apps, and subsequently, PWAs. Although PWAs rely on a secure backbone, general-purpose browsers don't provide a level of security as high as native apps, which can lead to vulnerabilities affecting the user. To combat these threats, we have partnered with Build38, using their T.A.K security library to build a hardened native app to display PWAs. As demonstrated in the testing, the app implements a higher level of security than popular browsers like Google Chrome, preventing many forms of attacks and improving the user's experience.

Our app would be convenient for developers that need an extra degree of security but don't want to invest in a native app, as configuring it for a new project is as easy as changing the target URL and app icons. With the Hardened Browser, you can convert a PWA into a native app in a short time and distribute it in stores like the Play Store without having to make your own from scratch. For users, using the Hardened Browser would be transparent, as they would download the application as any other native app, but would enjoy a higher degree of protection against many attacks than using a normal web browser.

## Acknowledgments

Many thanks to Build38 GmbH for allowing me access to the TAK library, especially to Jordi for all the assistance across the months despite his busy schedule. I am also thankful to my family for keeping my motivation and spirits high during this time.

## References

[1] statcounter. 'Desktop vs mobile market share worldwide.' (Sep. 2022), [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#yearly-2011-2022 (visited on 22/09/2022).

[2] A. Russell. 'Progressive web apps: Escaping tabs without losing our soul.' (Jun. 2015), [Online]. Available: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (visited on 22/09/2022).

[3] Google. 'Twitter lite pwa significantly increases engagement and reduces data usage.' (May 2017), [Online]. Available: https://web.dev/twitter/ (visited on 23/09/2022).

[4] Google. 'Progressive web apps: Adoption has its benefits.' (Jan. 2022), [Online]. Available: https://web.dev/learn/pwa/progressive-web-apps/#adoption-has-its-benefits (visited on 23/09/2022).

[5] Android. 'App security best practices.' (Oct. 2022), [Online]. Available: https://developer.android.com/topic/security/best-practices (visited on 02/11/2022).

[6] Android. ‘Security tips.’ (Oct. 2022), [Online]. Available: https://developer.android.com/training/articles/security-tips#WebView (visited on 02/11/2022).

[7] Android. ‘Safetynet safe browsing api.’ (Nov. 2022), [Online]. Available: https://developer.android.com/training/safetynet/safebrowsing (visited on 02/11/2022).

[8] statcounter. ‘Browser market share worldwide.’ (Dec. 2022), [Online]. Available: https://gs.statcounter.com/browser-market-share (visited on 14/01/2023).

[9] OWASP. ‘Types of xss.’ (Feb. 2022), [Online]. Available: https://owasp.org/www-community/Types_of_Cross-Site_Scripting (visited on 14/01/2023).

[10] OWASP. ‘Xss filter evasion cheat sheet¶.’ (Jan. 2023), [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html (visited on 14/01/2023).

[11] Mozilla. ‘An overview of http.’ (Jan. 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (visited on 12/01/2023).

[12] Google. ‘Https encryption on the web.’ (Jan. 2019), [Online]. Available: https://transparencyreport.google.com (visited on 12/01/2023).

[13] Mozilla. ‘Strict-transport-security.’ (Jan. 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security (visited on 12/01/2023).

[14] L1ghtn1ng. ‘Sslstrip.’ (Jun. 2022), [Online]. Available: https://github.com/L1ghtn1ng/sslstrip (visited on 13/01/2023).

# APPENDIX

## A.1 Planification Gantt diagram

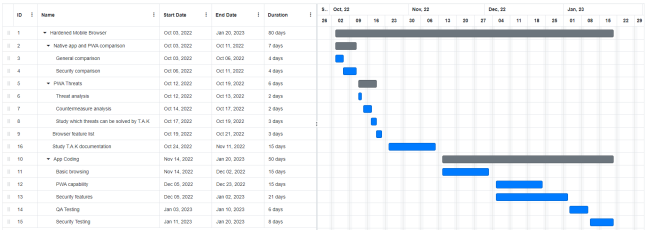



Fig. 7: Planning Gantt diagram

## A.2 Hardened Browser Code

### A.2.1 WebView Client

```
private fun takShouldHandleRequest(request: WebResourceRequest?): Boolean {
        return request?.url?.scheme?.startsWith("file://") != true

    }

private fun shouldBlockNetworkRequest(request: WebResourceRequest?): Boolean {
    val parsedUrl = request?.url?.host
    if (parsedUrl != null) {
        if (parsedUrl.contains(Constants.TARGET_PWA_URL) || parsedUrl in Constants.ALLOWED_URLS) {
            return false
        }
    }
    Log.i("TAK", "Blocked request "+ request?.url)
    return true
}

//Called when the user interactively requests a resource (click a link)
override fun shouldOverrideUrlLoading(
    view: WebView?,
    request: WebResourceRequest?
): Boolean {

    if (shouldBlockNetworkRequest(request)) {
        Toast.makeText(context, "Trying to load non-whitelisted page", Toast.LENGTH_SHORT).show()
        return true
    }
    return false
}

//Called when any resource is requested, if whitelisted continue, else, don't load
// (might break some functionality)
override fun shouldInterceptRequest(
    view: WebView?,
    request: WebResourceRequest?
): WebResourceResponse? {

    var tempReturn: WebResourceResponse? = null

    if (shouldBlockNetworkRequest(request)) {
        return WebResourceResponse("", "", context.assets.open("custom_pages/certificateError.html"))
    }

    if (takShouldHandleRequest(request)){
            try {
                var mimeType = "text/html"


                val newRequest = Request.Builder()
                    .url(request?.url.toString())
                    .build()

                val response = makeTAKrequest(newRequest)

                if (request?.url?.toString()?.contains("css") == true)
                    mimeType = "text/css"
                if (request?.url?.toString()?.contains("js") == true)
                    mimeType = "text/javascript"

                Log.i("TAK", "Loading  " + request?.url.toString())

                tempReturn = WebResourceResponse(
                    mimeType,
                    "utf-8",
                    response.body?.byteStream()
                )

            } catch (exception: Exception) {
                Log.i("TAK", "Exception getting url with TAK " + request?.url + exception)
                return if (Constants.SHOULD_LOAD_PAGE_ON_TAK_EXCEPTION) {
                    null
                } else {
                    WebResourceResponse("text/html",
                        "UTF-8",
                        context.assets.open("custom_pages/certificateError.html"))
                }
            }
    }
    return tempReturn
}
```

### A.2.2 WebView

```
init {

        // Some PWA just don't work without Javascript
        this.settings.javaScriptEnabled = Constants.SHOULD_ENABLE_JAVASCRIPT
        // Set custom webViewClient for our webview
        this.webViewClient = CustomWebViewClient(context)
        // Maximum cache size is 20mb as seen on line 98 on the source code,
        // which is not enough for most PWAs, we might as well disable it.
        // https://android.googlesource.com/platform/external/webkit/+/refs/heads/master/Source/WebKit/android/WebCoreSupport/WebCache.cpp
        this.settings.cacheMode = WebSettings.LOAD_NO_CACHE
        // Never allow secure origin to load resources from insecure origins
        this.settings.mixedContentMode = WebSettings.MIXED_CONTENT_NEVER_ALLOW
        //Whether the app can load local files
        this.settings.allowFileAccess = false
        //Enable safe browsing (Also in manifest)
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            this.settings.safeBrowsingEnabled = true
        }

    }
```