b
UNIVERSITÄT
BERN

Automatically Retrofitting Cordova Applications for Stricter Content Security Policies

Bachelor Thesis

Basil Schöni
from
Sumiswald BE, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

06. Februar 2020

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Content Security Policy (CSP), a feature present in Android's WebView for many years, has the potential to protect against most types of code injection attacks. However, adoption rates are low and existing policies often apply weak restrictions. We investigate attack methods against WebView and how CSP can prevent them.

We found that there is a wide variety of injection vectors, ranging from external sources like NFC communications to internal ones like Android's inter-app communication. The impacts include breaches of privacy, credential stealing and further spreading of malicious code. CSP mitigates such attacks by blocking various classes of code execution, loading external data, exfiltration of data, UI manipulation and insecure connections. We propose a tool that generates such CSP definitions for pre-existing, real-world Cordova apps. To improve the strictness of these CSP definitions, our tool attempts to rewrite all Javascript APIs that are restricted by CSP. We evaluated the tool using a large data set and found that we can avoid the "script-src unsafe-inline" definition in 84.28% and the "style-src unsafe-inline" definition in 25.88% of cases. Conversely, for the "script-src unsafe-eval" definition, no application could benefit from our rewriting and for "style-src unsafe-eval", loosening strictness could only be avoided for 2.89% of applications. From this we conclude that while our approach provides significant benefits with respect to the "unsafe-inline" keywords, it is mostly ineffective in rewriting to avoid the "unsafe-eval" keywords.

We identified six patterns which limit either the strictness or the non-breaking behavior of our generated policies and two use cases which make the static generation of non-breaking policies completely impossible. We conclude that any static rewriting of Javascript APIs should apply in-depth flow analysis and be able to deal with special syntaxes introduced by the most common UI frameworks. Approaches like ours that do not apply these measures may work well for smaller applications, but will cause breaking for more complex ones.

Contents

1 Introduction

When developing a mobile application, compatibility is a concern. Even though the market for mobile operating systems is not heavily fragmented, app developers have to overcome the problem of reaching users on both the iOS and Android platforms. Since the native languages of these platforms are not compatible, for a long time, developers had a choice to make: Either limit themselves to building up a user base on one of the platforms, or develop a large portion of the same application twice. In the past few years, this problem has been tackled by a number of mobile frameworks, which allow developers to serve both of the leading operating systems with a single code base. Libraries that employ WebViews to achieve such compatibility have been around for some years and have their fixed place in the ecosystem of cross-platform mobile frameworks. The arguably most advanced mobile web framework at the time of writing is Cordova (formerly PhoneGap).

Web-based cross-platform mobile frameworks tackle the challenge of compatibility by wrapping applications inside a WebView, which is essentially a browser engine executing HTML, CSS and Javascript. Additionally, those frameworks provide interfaces to interact with the underlying operating systems, thus abstracting away their differences for the framework users. Developers then write their applications using web technologies and the interfaces provided, resulting in applications that can be executed on all major platforms. However, wrapping applications inside a WebView comes at a price. Besides reduced user experience and some performance overhead, the nature of web technologies allows attackers to perform the class of code injection attacks against such applications. Code injection attacks are possible because data and code can appear alongside each other as part of the Document Object Model (DOM). If an application accepts data from an untrusted source and adds that data to the DOM in certain unsafe ways, any code contained in that data will be executed. While such injected code is under some constraints with respect to what it can read from or write to, the mere existence of this attack vector poses a threat to the user's privacy and their device's integrity.

Code injection vulnerabilities in mobile web applications have been studied widely by researchers in the last years. While a majority of studies focused on the detection of such attack vectors, the topic of protecting against them appears to be a somewhat neglected subject. Since the year 2012, browsers have been implementing an additional layer of security called Content Security Policy (CSP), which is specifically designed against cross-site scripting attacks. CSP allows developers to apply restrictions that limit the execution of code based on the sources of that code and how it was added to the application. Because the Android WebView (which is investigated in this thesis) is based on Chrome, this security mechanism is present and ready to be deployed for mobile web applications on Android. While some of the research on mobile web applications briefly mention CSP as a protection against injection attacks and one study investigates the prevalence of its deployment amongst Cordova applications, none of the reviewed studies have actually investigated the process of adding CSP to mobile web applications in practice.

In this thesis, we investigate the different ways of attacking mobile web applications on Android and assess how Content Security Policy can protect against such attacks. Furthermore, we propose a tool that rewrites existing Cordova applications in order to avoid specific Javascript APIs that are incompatible with certain CSP definitions and therefore limit our ability to prevent certain kinds of malicious code from being executed. Our tool then analyzes the given application and statically generates a CSP definition with the goal of being as strict as possible while not breaking any functionality. We evaluate our proposed tool and run it against a large data set in order to establish empirical metrics for the feasibility and potential benefits of our approach.

Our research is guided by the following research questions:

RQ 1 What attack methods against mobile web applications exist?

RQ 2 How can CSP prevent or mitigate such attacks?

RQ 3 Can we automatically generate sensible CSP definitions for real-world Cordova apps?

- **RQ 3.1** Can we rewrite real-world Cordova apps to allow more strict CSP definitions?

RQ 3.2 What patterns (i.e. usages of specific APIs in specific ways) limit us in rewriting applications and generating CSP definitions?

RQ 4 How prevalent are the patterns we attempt to rewrite?

RQ 4.1 How prevalent are the patterns we can successfully rewrite?

RQ 4.2 How prevalent are the patterns we cannot rewrite?

The rest of this thesis is structured in the following way: Chapter 2 gives an overview of existing research on code injection vulnerabilities in mobile web applications. Chapter 3 provides an analysis on the possible attack methods and their impacts, answering our first research question. Chapter 4 describes how Content Security Policy is defined, what APIs are impacted by it and how it can prevent attacks, thus giving an answer to the second research question. In chapter 5 we present our tool and its evaluation. We describe the patterns limiting our ability to rewrite APIs and generate strict CSP definitions, which gives an answer to our third research question. The fourth research question is answered in chapter 6, which contains the results of our empirical analysis. In the analysis, we investigated the prevalence of the Javascript APIs our tool attempts to rewrite, how many apps actually benefitted from our tool and how common the usage of certain frameworks is in those apps. This gives us empirical data on the effectiveness of our tool and, in consequence, of the feasiblity of our approach. Chapter 7 is a summary of our work in which we draw our final conclusions. In the appendix to this thesis, we present a description of Android's WebView API, along with the risks introduced by it and recommendations on how to reduce them.

2
Related Work

Security in mobile apps has attracted a lot of attention in research. For instance, Ghafari et al. [10] reported 28 security vulnerabilities and their symptoms in the code that compromise security and privacy in Android apps. In this chapter, we assess the current state of research that concerns the security of mobile web applications and the application of Content Security Policy.

A. B. Bhavani [6] performed an early study on XSS attacks against WebViews. They investigate attacks performed by a malicous mobile web application that steals cookies or accesses sensitive data and then exfiltrates them using the “HttpPost” class. Cookies are stolen through overriding of the “shouldOverrideUrlLoading” method, which then uses the “getCookie” method of the “CookieManager” class to access the cookie data. Sensitive data is accessed via the permissions granted to the malicious application. Both attacks can be classified as cross-site scripting attacks that do not require usage of WebView’s “setJavascriptEnabled” method.

Jin et al. [14] carried out research on applications built with the “PhoneGap” framework. They show that unlike traditional web applications, mobile web applications are affected by many different injection vectors only present in mobile devices. Such mobile-specific channels include data read from the device’s NFC functionality, barcode reader, file system or content providers like contacts or call logs. These channels are made available by the Javascript bridges PhoneGap provides. The study proposes a tool that performs static analysis to automatically detect potential code injection vulnerabilities present in a given application. This is done by first modeling the relevant APIs of the PhoneGap framework. Then, program slices are constructed that contain code responsible for reading from an untrusted channel and passing that data to an unsafe API in order to display it to the user. Finally, taint analysis is performed on these slices to identify dangerous information flows. They investigate the applications flagged by their detection tool and propose mitigation techniques for such attacks on the framework level.

In the study by Mutchler et al. [17], a large-scale analysis of mobile web applications is carried out. The study investigates a full snapshot of all free Android applications available on the Google Play Store as of June 2014. They propose a tool that automatically detects whether applications load content insecurely (i.e. via HTTP, misconfigured HTTPS, user navigation or from insecurely stored local data), exposes stateful navigation that allows foreign apps to perform sensitive web operations, or leaks sensitive information by creating implicit intents from an overriden navigation control method (i.e. "shouldOverrideUrlLoading" or "shouldInterceptRequest"). The study also discusses several mitigations that can be applied by developers in order to avoid such vulnerabilities.

The research carried out by Chen et al. [7] extends on the work by Jin et al. [14]. They investigate an injection channel not considered by Jin et al., which leverages malicious data from HTML text input tags. A tool is implemented to detect dangerous data flows using static analysis. The sources and sinks used in this detection pipeline consist of the ones established by Jin et al., complemented by the newly found text box injection channel.

The work by Hassanshahi et al. [11] looks at what it calls "web-to-app injection" attacks. This attack type uses malicious intent links that can be clicked in a standard web browser and then trigger browsable activities of another application. Such an intent can contain payload data. A mobile web application is vulnerable to malicious intents, if it takes this payload data and treats it as a URL that is to be loaded into the WebView. Vulnerabilities like this can lead to malicious usage of the Javascript bridge or the HTML5 APIs (e.g. the geolocation API), as well as reading from the file system or redirecting the user to a phishing site. The study proposes a tool based on static analysis, symbolic execution and dynamic testing that detects vulnerabilities and confirms those vulnerabilities by generating working exploits.

Yang et al. [27] investigate vulnerabilities that can be triggered via insecure connections (HTTP or insecure HTTPS) and allow access to sensitive Javascript bridges. The study proposes a tool that performs static analysis to detect such vulnerabilities and applies dynamic analysis to confirm them.

In the study by Choi et al. [8], an environment is described that allows automatic detection of injection vulnerabilities by introducing HTML script tags into a number of different channels. A user then explores a given application's UI. If an HTML script tag is triggered, a request is sent to a server responsible for listening to such events, thus gathering data on vulnerabilities present in the app.

Yang et al. [26] implemented a tool that automatically vets the usage of Javascript bridges in hybrid applications and confirms potential vulnerabilities by generating example exploit code. They implemented a generalized model of WebView to abstract away from its different implementations, such as Android's WebView, Mozilla's rhino-based WebView or Intel's XWalkView. They present a so-called "shadowbox" data structure to properly maintain path- and value-sensitivity, in order to avoid certain types of false positives.

The usage of sensitive WebView APIs is analyzed in the study by Hidhaya et al. [12]. Based on a set of inference rules, static analysis is performed to detect attacks such as excess authorization, supplementary event-listeners, touch-jacking, UI overriding and event-simulation attacks.

Davidson et al. [9] implemented a service that provides secure WebViews to foreign apps in order to avoid code injection attacks. The service allows developers to define policies for both mobile web applications and web applications that may be displayed inside a WebView, thus offering protection against both app-to-web and web-to-app attacks. They also implemented a rewriting tool that enables compatibility with the service for third-party applications not under control of the user.

The security mechanisms of Cordova and their usage in real-world applications are investigated in the study performed by Willocx et al. [24]. They find that a majority of applications they analyzed uses outdated versions of Cordova. Furthermore, a significant portion has wildcard entries in one of the whitelisting mechanisms, thus imposing unnecessarily loose restrictions on navigation, resource access and intent creation. Finally, they perform a market analysis of cross-platform tools, Javascript frameworks and Cordova plugins. Also, guidelines for developers of applications, plugins and the Cordova framework are presented.

A somewhat different approach is taken by Rizzo et al. [21]. Instead of just detecting potential vulnerabilities, the goal is to analyze the impact of such vulnerabilities and compare this to how difficult exploitation is. As a result, a report is generated that allows classifying vulnerabilities according to these metrics. The analysis is performed statically, by replacing all instances of WebView with their specially-crafted "BabelView", which contains their attack model and provides tainted input sources. They also analyze the feasibility of such attacks, as well as correlations between different alarms reported by their tool.

A study of Javascript usage and related vulnerabilities in large mobile web applications is carried out by Song et al. [22]. They identified four types of Javascript usage, namely the local, remote, Javascript bridge and callback patterns. The first three of those can lead to one type of vulnerability each: The file-based cross-zone vulnerability, the WebView UXSS vulnerability and the Javascript-to-Java interface vulnerability. A tool for static analysis is presented that categorizes Javascript usage into the aforementioned classes and determines whether or not the specific implementations of the patterns leads to vulnerabilities in the analyzed applications.

In the work by Peguero et al. [18], the consequences of security mechanisms of the "Jade/Pug", "EJS", and "Angular" Javascript frameworks are investigated. The study differentiates between five levels of mitigation: No mitigation, custom sanitization or filtering function, external sanitization or filtering library, sanitization or filtering provided by framework plugins, and sanitization or filtering built directly into the framework. These levels correspond to how "close" to the framework code a mitigation technique is placed. They find that the closer a mitigation is to the framework, the more impact such sanitization has on the security of the application.

Bugs that are introduced by the usage of WebView are investigated by Hu et al. [13]. Six root causes of WebView-induced bugs are identified: Misalignment of WebView lifecycles, WebView evolution and varying device specifications, WebView misconfiguration, misuse of WebView APIs, heavy usage of Javascript bridges, and limitations of WebView's browser engine that were not anticipated by developers. The study also identifies the manifestations of such bugs and presents a tool that automatically detects the presence of WebView-induced bugs. While such bugs may be relevant to the security of mobile web applications, the study does not specifically analyze their potential security impacts.

Bai et al. [5] present three different solutions to analyzing the security of mobile web applications. First, they propose a method for bi-directional dynamic taint tracking across Javascript bridges. Second, a tool for detecting privacy leaks was implemented using this taint tracking method. And third, a benchmarking application was created for testing different tools that analyze the security of Javascript bridge communications. They also compare their solution to HybriDroid and Google's Play Protect and find several shortcomings in these tools that their solution is able to overcome.

In the study by Pouryousef et al. [19], the profiles of API invocations for native and hybrid applications are compared. They implement a tool that performs static analysis in order to analyze the usage of security-sensitive APIs and report on the following three metrics of a given application: Dynamicity, transferring data between Javascript and Java code, and usage of advertisement libraries. They found that hybrid applications differ significantly in their generated profiles from native apps.

An extension on the work by Jin et al. [14] is proposed by Lau [16]. In this study, an approach of identifying hand-written Javascript bridge plugins in Cordova applications is presented. The plugins are detected by scanning for previously defined “tags”, which are primarily verbs and nouns often to be found in certain plugins. For example, a plugin related to SMS functionality might often use words like “message”, “sms” or “receive”. They also implemented a slicing algorithm that is able to analyze various callback function contexts, which was a category of functions missed by previous approaches of static code injection analysis.

Weichselbaum et al. [23] investigate the adoption of Content Security Policy in a data set they consider to be representative of the whole world wide web. They concentrate on the “script-src” and “object-src” directive, since they consider those the most relevant ones to security. Several methods of circumventing CSP’s script blocking capabilities are presented, the most notable one affecting any CSP definitions that whitelist domains containing a JSONP endpoint. The authors then perform an analysis on their large-scale data set and conclude that only 0.16% of unique domains specify a CSP, and over 94% of the policies defined can be bypassed using one of the presented methods. To tackle this problem, the study proposes an alternative way of building policies for the “script-src” directive: Instead of whitelisting actual sources, any scripts should be secured with a nonce in combination with the “strict-dynamic” keyword which causes trust to be propagated to any script loaded from a trusted script. This approach can effectively prevent the JSONP circumvention method they investigated. The authors conclude their work with a case study for the “strict-dynamic” keyword and an assessment of the limitations of their proposed policy building method.

3 Security Analysis

The usage of WebView introduces certain security risks to an Android application. Most importantly, being based on web technologies, WebViews are susceptible to code injection attacks. This can lead to credential stealing, privacy leaks or other malicious behavior in benign applications. Furthermore, malicious applications that use WebViews can employ certain techniques to covertly spy on third-party content, impersonate an unsuspecting user and circumvent vetting mechanisms that newly submitted applications may have to pass to enter certain software marketplaces (such as the Google Play Store). The following chapter looks at causes and impacts of code injection attacks against benign applications, as well as possible attack and circumvention methods for malicious apps. Additionally, it provides a short overview of further security risks that do not match the aforementioned categories. In doing so, we answer our first research question: "What attack methods against mobile web applications exist?"

3.1 Code Injection

As previous research has shown, code injection is a major risk mobile web applications are affected by. A code injection attack against WebView is possible, if certain conditions are met. First, an open injection vector must be present, over which Javascript code can be introduced into the application by an attacker. We call such an injection vector a "source", since in the context of an application, this is where the data first enters the system. Second, there must be some part of code that adds data to the WebView's DOM via an insecure API. This allows any Javascript passed into the API to be executed, if

Javascript execution is not disallowed by the WebView's specific configuration. We call such an API a "sink", since this is where the data eventually flows to. Third, there must be a data flow from a source to a sink, during which no sanitization takes place to remove unwanted types of data, such as executable Javascript code. The consequences of a code injection attack are manifold, and range from stealing specific sensitive information to the remote execution of arbitrary code. In the following section, sources, sinks and impacts of code injection attacks against mobile web applications are described.

3.1.1 Injection Sources

Inspired by Jin et al. [14], we divide code injection sources into external sources and internal sources. External sources allow data to be passed from outside the device. Examples of such external sources include SMS messages, insecure connections or malicious intent links. Internal sources are those which allow a data flow within the device context. Those include content providers (like contact or calendar data), local files or inter-app communication mechanisms.

External Injection Sources

Mobile devices support a variety of channels to read external data. Some of them are specific to to mobile devices (such as SMS messages), while others can be found in most modern computers (such as Bluetooth or Wi-Fi).

Barcodes Barcodes, specifically two-dimensional ones (commonly known as "QR codes"), allow users to easily access data that is visually available through scanning them via a device's camera. The data is encoded in the barcode's pattern. If such a barcode encodes valid Javascript code, displaying the decoded data in a WebView without sanitization will result in execution of the code.

Bluetooth and Wi-Fi IDs Although Wi-Fi and Bluetooth are technologies that allow establishing a connection to another device and are therefore not intuitively considered to be able to contain any payload data that will be displayed to the user, they do provide an injection channel that can be leveraged. In order to establish a connection, a device has to scan for all available devices nearby and display their IDs to the user, who can then select the one they wish to connect to. Those IDs may appear benign on the first glance, but can contain valid code that will enter the device context when scanning for available devices. As Jin et al. [14] describe, even the limitation of ID lengths can be circumvented by splitting up code into fragments of appropriate length, multiple times setting the ID to valid code that assigns those fragments to a number of variables, and finally setting the ID to an "eval" statement that gets passed a concatenation of the variables as its argument.

Expired Domains Since companies may abandon registered domains (either because they are no longer needed or because a company goes out of business), and those domains may still be used by out-of-date applications, expired domains can act as an injection source. By taking over such an expired domain, an attacker can put malicious code on the site, which may then be loaded into a WebView, leading to the execution of that code. This has been described and confirmed by Mutchler et al. [17].

Insecure Connections A mobile web application may load content that is considered to be trusted over a connection that is not properly secured. This is the case when accessing data over plain HTTP connections or when using misconfigured HTTPS. Such a connection can be attacked via man-in-the-middle, allowing the attacker to inject Javascript code into the HTTP response. This vector, which was investigated by Mutchler et al. [17], acts as an injection source for mobile web applications.

Near-Field Communication Many modern phones and other devices contain hardware that allows data to be passed in a contactless manner. Recently, this has been widely adopted by providers of payment systems or companies that offer transportation services (for example, the Swiss Federal Railways offers the "SwissPass", which contains an RFID chip that stores a customer's transportation subscriptions and can be read out from an employee's mobile phone). Data that is read from another device or a passive RFID tag may contain Javascript code that will enter the device context over this NFC channel.

SMS Messages Via SMS, messages from third parties can be received on a user's mobile device. Such messages may contain Javascript code which will get executed if it is passed to a injection sink and no precautions are taken.

User Navigation to Malicious Sites If a WebView is configured in a way that allows users to navigate to arbitrary sites, they may be tricked into navigating to malicious content, which will then be loaded directly into the WebView. In this way, code injection can be performed. This vector was investigated by Mutchler et al. [17], who have also developed heuristics to automatically detect such injection channels.

Internal Injection Sources

Mobile devices offer a number of ways for applications to communicate with other resources or applications on the same device. This is useful for many reasons, but can be exploited by malicious or infected applications in order to inject code into another vulnerable app that may for example have less restrictive permission settings.

Content Providers A content provider is a piece of software that implements the corresponding Android interface class. It is used to provide access to data to multiple applications on a device. The most common content providers according to Jin et al. [14] are browser, calendar, call log, contacts, profile, sync adapter and user dictionary. Code can be put into such a content provider (e.g. via a contact's name or a calendar entry), which may then be displayed by a vulnerable application that receives data from this provider.

File System Applications on a device may be given access to the device's file system, in order to read and store data. Since the file system is available for any application that has the appropriate permissions, it is an easy way to target vulnerable applications. Besides the obvious possibility of putting code inside a file's content itself, Javascript may be placed in the file's names. In this way, actually accessing the manipulated file is no longer necessary. It is enough for a vulnerable application to display what is inside a given file system directory, in order for the code to be executed. To circumvent limitations in file name lengths, the same strategy as mentioned for Wi-Fi and Bluetooth ID channels can be applied. Jin et al. [14] note that this injection source can also be leveraged by an external attacker, since they may trigger a manipulated file to be downloaded onto the device's file system.

Insecurely Stored Local Content An application may load data that is locally stored on the user's device into a WebView. If this content is not properly secured, other malicious applications on the same device are able to manipulate it. This can turn such local contents into code injection sources that may be loaded into the WebView and subsequently executed. While this vector is mentioned by Mutchler et al. [17], the study does not further investigate such attacks.

Intents Intents are used in Android for inter-app communication. They describe an action that is to be performed by another application, as well as payload data that is needed in order to perform this action. Through the payload data, Javascript code may be passed, making this an injection source for a vulnerable application. If the data is displayed in an unsafe manner, the code will be executed. While such an injection source can be leveraged by malicious applications already present on a user's device, intents can also act as external injection sources. An attacker can trick a user into clicking a specially crafted, malicious intent link via the default browser. If such an intent is reacted to by a mobile web application and the intent's parameters are treated as URLs, they may be loaded into the WebView, which might lead to the execution of any Javascript code that is present. This injection channel is extensively covered by Hassanshahi et al. [11].

Local Malware If a device was previously infected by malware, this malware can act as an injection source, if it contains malicious Javascript code. This is briefly mentioned by Song et al. [22].

Metadata Media files often contain a variety of meta information, such as artist, genre, the software they were created with or the manufacturer of the device it was captured on. Document files such as .docx, .xlsx or .odt files also provide metadata fields where author, title or other information can be stored. Those metadata fields can contain executable code which may be passed to an injection sink in an unsafe way. While Jin et al. [14] classify metadata fields as external channels, we consider them to be an internal source, since the files are present on the device's storage system and therefore do not enter the device from the outside. Since the files containing such metadata reside on the file system, this channel may be leveraged by an external attacker in the same way as other file system sources (i.e. by triggering a file download from an external site).

Text Inputs As described by Chen et al. [7], HTML <input> tags with the "type" attribute set to "text" can also act as injection sources. If the value of such an element is extracted via methods like 'document.getElementById("someId").value' and then added insecurely to the DOM, any code that was present in the text input will be executed.

Third-Party Content Third-party libraries that are considered to be trusted but are not actually vetted by developers can act as injection sources for mobile web applications. Such libraries may include malicious Javascript code — possibly embedded in a local HTML file — which may be loaded into a WebView. Third-party contents are therefore possible injection sources. This kind of source is mentioned by Song et al. [22] in their analysis of Javascript usage in Android applications.

3.1.2 Injection Sinks

Code injection sinks can have their origins in various libraries used by mobile web applications. Always present in a WebView are the WebView APIs and the DOM APIs. Depending on the third-party libraries used in an application, other APIs that act as injection sinks might be present. Those include DOM-manipulating methods as present in the widely used "jQuery" library, as well as DOM-manipulating functionality used in Javascript frameworks like "Prototype" or "Angular". In the following section, only the WebView, DOM and jQuery APIs are described, since they are the most common and most important origins for injection sinks. For the security of Javascript frameworks, refer to Peguero et al. [18].

WebView API

The WebView API is used to configure a given WebView and trigger events inside that WebView. Some of those interfaces must be considered injection sinks. The unsafe functions are: “evaluateJavascript”, “loadData”, “loadDataWithBaseURL”, “loadUrl” and “postUrl”. While not all of those methods allow access to the same resources due to WebView’s same-origin policy, they all must be considered possible injection sinks, since untrusted data that flows into one of those methods can lead to the execution of Javascript code. For a detailed description of the WebView API, refer to chapter A.1.

DOM API

The DOM API provides developers with a variety of ways to manipulate the content of a document. While some of those functions and attributes allow adding content to the DOM in a safe way (i.e. they just add plain text and do not parse and execute any valid code), there are others that do act as injection sinks. In the DOM API, there are two methods and two attributes that cannot be considered safe when adding untrusted content: The “document.write” and “document.writeln” methods and the “innerHTML” and “outerHTML” attributes. Other interfaces like the “innerText” or “value” attributes can be considered safe when assigning untrusted contents.

jQuery API

Since the jQuery library provides wrappers around ‘regular’ DOM objects, there are many jQuery functions that add content directly to the DOM. While some of those functions (e.g. “text” or “val”) do not lead to execution of any code present in the arguments, the following functions should be considered unsafe: “html”, “append”, “prepend”, “before”, “after”, “replaceAll”, “replaceWith”. As is the case with unsafe DOM APIs, due care should be taken when using those functions, since they can act as injection sinks.

3.1.3 Impacts of Code Injection

Successful code injection can have a variety of consequences which threaten the privacy of the user or even the integrity of the whole device. The impacts of code injection as discussed in the investigated studies are described in the following section.

Access to Local Files

If an application invokes the “setAllowFileAccessFromFileURLs” method, any local Javascript code is permitted to access the contents of local files. Since this is the case regardless of file zones, this effectively leads to a violation of the same-origin policy. Besides the privacy threat that comes with access to file contents, the ability to check for the existence of certain files allows an attacker to build user profiles, which can be useful for recognizing different users. Davidson et al. [9] refer to this as “local storage inference”, while Song et al. [22] investigate such file accesses with respect to their violation of the same-origin policy. The latter also mentions that the “setAllowFileAccessFromFileURLs” setting was enabled by default in Android versions older than 4.2. In order to have malicious Javascript code available locally, a malicious site can trick the victim’s browser into downloading an HTML file that contains the malicious code. The site can then cause the victim’s device to load this file into a vulnerable app via an intent link. This method is discussed by Hassanshahi et al. [11].

Access to Sensitive APIs

If Javascript code is injected into a WebView, sensitive data may be accessible to the attacker. This is mainly due to some of the HTML5 APIs (like “Geolocation” or “MediaDevices”) and WebView’s Javascript bridge API. In the first case, a device’s location can be retrieved or media devices can be accessed if the appropriate permissions are granted to the application. In the latter case, any Java objects that are exposed via WebView’s Javascript bridge mechanism are available to code of any origin, even if it is embedded into a sub-frame. Since these Javascript bridges often grant access to sensitive data that may be needed for the app’s primary functionality, it is likely that Java objects are exposed which allow injected code to access private resources like the device’s camera, UUID, accelerometer or others. In Android versions prior to 4.2, it was even possible to access all public methods of an exposed Java object, which made it even more probable that sensitive data could be accessed. In newer Android versions, only methods that are specially annotated can be called over the bridge mechanism. Hidhaya et al. [12] describe this flaw under the term “excess authorization”.

In addition to these privacy-relevant APIs, an attacker may be able to steal session cookies using the “document.cookie” API. While this is not personal information in itself, it allows the attacker to pose as the user when communicating with the respective service. Because of that, private information may be stolen or actions which require authentication may be performed.

Arbitrary Code Execution

As mentioned above, in Android versions prior to 4.2, Java objects exposed over the Javascript bridge mechanism made all their public methods accessible to the Javascript code loaded inside the WebView. This allowed the execution of arbitrary Java code by leveraging Java’s reflection API in order to retrieve a runtime object which could then invoke a command line that had the same permissions as the application itself. With the introduction of the “@JavascriptInterface” annotation which has to be present on all exposed methods, this vulnerability has been closed.

Cross-site Scripting

Two possibilities of cross-site scripting arise in the context of WebViews. Obviously, both of these result in a violation of the same-origin policy. On the one hand, by invoking the “setAllowUniversalAccessFromFileURLs” method, an application can allow any local Javascript code to load any web content, regardless of http zone. As a consequence, cross-site code may be loaded into the WebView, effectively leading to cross-site scripting attacks. As with the “setAllowFileAccessFromFileURLs” method, this setting was enabled by default in Android versions older than 4.2. This is mentioned by Song et al. [22], who studied this setting with respect to its consequence of same-origin policy violation. The technique for making Javascript code available locally as mentioned above and by Hassanshahi et al. [11] applies here as well. On the other hand, if DOM-manipulating methods like “loadURL” or “loadDataWithBaseURL” are available to the injected Javascript code, remote, cross-origin Javascript code can be loaded into the WebView. This approach is investigated by Yang et al. [26].

Cross-site scripting can also be present in the form of so-called “universal cross-site scripting” (UXSS). These are vulnerabilities present in the browser engine itself which allow injected code to run in any page loaded by the a affected WebView. This can obviously have even more severe consequences than regular cross-site scripting. According to Song et al. [22], Android versions below 5.0 were affected by such UXSS vulnerabilities.

Injecting Code into Other Applications

As mentioned by Jin et al. [14], internal injection channels can be used to compromise other vulnerable applications on the same device. For example, a calendar event can be manipulated to contain Javascript code which will then get executed in other mobile web applications when they access this event.

Injecting Code into Other Devices

When an application is compromised, it may be able to leverage external injection channels in order to compromise other vulnerable devices. If valid code is for example put into the infected device’s Bluetooth ID, then a vulnerable application on another device may suffer from code injection over this external channel when scanning for Bluetooth IDs. This is also mentioned by Jin et al. [14].

Phishing

If injected code is able to manipulate the DOM, then phishing attacks are possible. This can happen by loading a malicious web page via methods like WebView's "loadUrl", which might be available over the Javascript bridge mechanism. Since the URL of the currently loaded page is not visible in a WebView, it is hard for users to detect when they are secretly directed to a phishing site.

Furthermore, an attacker may be able to add UI elements like input fields to the DOM, make them invisible via CSS stylings and place them directly over some genuine input fields already loaded in the WebView. The user will then believe they interact with the genuine form, while they actually input their sensitive data into the invisible one. With this method, credentials or other privacy-relevant information can be phished from the user.

3.2 Attack Methods for Malicious Apps

In addition to the abovementioned code injection attacks, a malicious application can use WebViews to silently spy on or manipulate third-party content that is loaded within the WebView. This can for example be relevant in the case of browser apps that are based on WebViews or other types of applications that do not own the content they display. The fact that WebViews allow the asynchronous loading of business logic code can furthermore be leveraged to bypass security mechanisms of app marketplaces.

3.2.1 Vetting Circumvention

As Yang et al. [26] describe, malicious developers can design their applications in a special way that puts the malicious behavior behind Javascript bridges and asynchronously loads any command & control logic from a remote web resource that is owned by the developers. In this way, the command & control logic is not present in the application's code at the time of the vetting process, but can easily be injected at an arbitrarily chosen point in time. This allows the circumvention of such vetting mechanisms.

3.2.2 Introspection and Manipulation of Third-Party Content

If third-party content is being loaded into a WebView, a malicious application can steal sensitive information like credentials. Davidson et al. [9] provide an example where a trusted single sign-on provider loads a login dialog into the WebView. The malicious application can then inject Javascript code into this WebView, extract the contents of the username and password input fields, and then call a Javascript bridge method via an overriden navigation callback, in order to exfiltrate the data through means like HTTP requests. This vulnerability is caused by the fact that content loaded in a WebView is assumed to be owned by that WebView. The third-party content is therefore not protected by same-origin policy. A similar example with the same causes and consequences is presented by Hidhaya et al. [12]. Here, the injected Javascript code adds event listeners to certain HTML elements that contain sensitive information. When the provided callback function is invoked, the sensitive data is stolen.

3.2.3 User Impersonation

Another method for attacking web content from a malicious app is to impersonate the user after they have authenticated themselves to a third-party web page. Since the WebView is then able to make authenticated requests to the page, the malicious app can disguise itself as the user while stealthily triggering sensitive functionalities in the third-party page. Davidson et al. [9] mention the example of a companion application for a popular website, which is a wrapper around a WebView that may be manipulated, repackaged and distributed by a malicious actor. Alternatively, malicious browser apps based on WebView could wait for specific web pages to be loaded in order to launch impersonation attacks after authentication to those pages has been completed. A special kind of impersonation attack is described by Hidhaya et al. [12]. In this attack, the malicious application simulates touch events on the loaded third-party content.

3.2.4 UI Confusion

Hidhaya et al. [12] analyze three attacks that are similar in how they obtain data which was originally intended for a trusted third-party web page. In all three attacks, the user is tricked into interacting not with the web page they intended to, but with another WebView or with the app itself. To denote this shared technique, we call this class of attacks "UI confusion attacks". In the "invisible WebView attack", an additional WebView is rendered on top of the original one. The additional WebView has its opacity modified via the "setAlpha" method. The user then interacts with this additional WebView, even though they think they are interacting with the original, underlying one. The "WebView redressing attack" is very similar to this, but instead of overlaying one WebView over the other, multiple WebViews are integrated into a single one. This as well has the consequence that the user actually interacts not with the WebView they intended to, but with another, disguised one. Finally, in the "UI-overriding attack", the fact is exploited that a WebView's HTML UI elements look very similar to Android's native UI elements. Because of this, native UI elements can be placed on top of HTML UI elements, without the user noticing this override. As a consequence, the user will interact with the actual malicious application, while they think they interact with the third-party web page that is loaded inside the WebView.

3.3 Further Security Flaws

In addition to the attack methods mentioned above, there are further security flaws that may be present in applications that use WebView.

3.3.1 Exposing Privileged Web Operations via Intents

Mutchler et al. [17] mention the problem of “exposed stateful navigation”. In this security flaw, a mobile web application listens for an intent and responds to it by calling (for example) WebView’s “postUrl” method with a specific set of parameters. If this POST request performs a privileged operation (Mutchler et al. give charging the credit card of the user as an example), then a malicious app on the same device could create such an intent, thereby invoking an operation it was not intended to perform. This problem can be mitigated by asking the user for confirmation whenever such an intent is reacted to.

3.3.2 Leaking Sensitive Data Through URL Overriding

Mutchler et al. [17] also draw attention to what they call “leaky URLs”. This flaw is present if the application overrides URL loading in such a way, that an implicit intent is created to load the URL instead of the WebView. If this URL contains sensitive information, that information may leak, because any application is allowed to handle the intent. Using custom URL schemes does not provide any protection from this leakage, since Android does not shield custom URL schemes from foreign applications. This flaw can for example be exploited in order to steal OAuth credentials.

3.3.3 Framework Vulnerabilities

In addition to those specific security flaws, the usage of hybrid application frameworks may introduce further vulnerabilities into a mobile web application, if such vulnerabilities are present in the framework code. Because framework code is seldomly checked for security problems by the users of such frameworks, choosing a strongly maintained framework library that in the best case undergoes regular security audits would be recommended. That being said, the potential risks of using a framework library should be compared to the additional security mechanisms such a framework may provide.

3.4 Case Study

To illustrate the specifics of code injection in a real-world application, we present a case study of a vulnerable insurance company app. Through the app, clients of the company can manage their insurance policies, access information regarding their insured vehicles and make claims in case of a car accident. The app has 100+ installs on the Google Play Store and requires permissions for writing to storage, accessing coarse and fine device location, accessing the internet and checking the network state. When a client wishes to make a claim following a car accident, the app allows them to describe the accident by entering information about the involved vehicles, their drivers and any witnesses.

3.4.1 Vulnerability

The application contains a vulnerability in its claim creation functionality. When a user creates a claim and then adds driver, vehicle or witness information to that claim, this information is stored in a local database. Since the information that is stored in the database is not sanitized, any data added as driver, vehicle or witness information may contain valid Javascript code.

In the view that displays a user's claim, any such information previously entered is retrieved from the database and displayed to the user. Depending on the scope we choose, the loading from the database can be interpreted as the source of our code injection. Alternatively, the HTML input tag where we enter the data to be stored in the database can be interpreted as the source. If we choose the first interpretation, the following code acts as one of the injection sources where the malicious data enters the application context. Because malicious code can be entered into either the driver, vehicle or witness information, there is more than one possible source/sink pair. Here, we present an example where the malicious data has previously been added as the driver's information.

```
function updateToLatestDrivers() {
  notesDB.transaction(function (tx) {
    tx.executeSql('SELECT * FROM otherDrivers
    where NoteID = ?',[id], updateDrivers);
  })
}
```

As can be seen from the code, a query is sent to the local database. This query retrieves the information about all the 'other' drivers involved in the accident, meaning all the drivers except for the client themselves. If some of the fields that this query returns contain valid Javascript code, this snippet acts as an injection source and the malicious code is loaded into the application.

This data is then added to the DOM in order to display it to the user. The piece of code that does this can be considered the injection sink. In case of the driver information, the sink looks as follows.

```
function updateDrivers(tx, results) {
  driverHTML += results(i).firstName;
  $("#allDrivers").prepend(driverHTML);
}
```

In this snippet, the data retrieved from the database using the abovementioned ‘updateToLatestDrivers’ function is added to the DOM. This is done using the ‘prepend’ method of the jQuery library. Since this method causes all present Javascript code to be evaluated, all malicious expressions contained in a driver’s first name will be executed.

3.4.2 Impacts

We tested this vulnerability by injecting a script that accesses the device’s geolocation and reports it back to the user. Since the application requires access to the device location by default, no further asking for permission is necessary. The script we injected looked as follows.

```
<script>
  function showPosition(position) {
    alert("Latitude: " + position.coords.latitude
      + "Longitude: " + position.coords.longitude);
  }
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(showPosition);
  }
</script>
```

This script first defines the function ‘showPosition’ which is a callback function that accepts a position object as returned by the HTML5 geolocation API and reports it back to the user via the ‘alert’ expression. After defining this callback function, the presence of the geolocation object is checked. If it is present, the current position is retrieved and outputted by calling the ‘getCurrentPosition’ function and passing to it the previously defined callback method. This results in an alert window being created and showing the device location to the user.

While the injection of this specific code remains harmless, an attacker could choose to exfiltrate the device’s location using one of the methods mentioned earlier in this chapter. Furthermore, all information displayed to the user could be extracted and sent out as well. Since this includes data on the user’s insurance claim like the involved vehicles, drivers and witnesses, sensitive information could be obtained by an attacker in this way.

While this attack cannot be performed remotely, an attacker that has onetime access to the victim's device could inject malicious code. If this malicious code exfiltrated the device's location, the attacker would receive exact coordinates whenever the respective view is navigated to by the victim, thus breaching their privacy in a recurring manner.

4
Content Security Policy

Content Security Policy is a security mechanism directed against certain classes of attacks, the most notable ones being code injection and cross-site scripting. It is implemented in all modern browsers with the first version of the standard having been published by W3C in 2012. The currently recommended version is CSP level 2, with its third iteration being a working draft. Although not yet completed, many features of CSP level 3 are already supported by major browsers like Chrome (and Android's WebView as well, since it is based on Chrome). In the following chapter, Content Security Policy is described both in terms of its mechanisms and with respect to mobile web applications and the various attacks mentioned in the previous chapter.

4.1 How CSP Works

The main purpose of Content Security Policy is to restrict the resources a given application is allowed to load from or connect to, as well as limiting the execution of certain types of code. This is achieved by sending a CSP definition along with every page that is loaded by the browser engine. Such a definition can take the form of an HTTP header (called "Content-Security-Policy") or an HTML <meta> tag which is part of the loaded document's <header> tag. A complete CSP definition consists of one or multiple policy directives, each of which restricts certain aspects of an application's behavior.

A policy directive is a string of text whose first word is the name of the directive while the subsequent words represent the (classes of) sources which are allowed for this given directive. Policy directives are whitelists, meaning that any resource not included in the directive is fully disallowed. For example, a policy directive for limiting images and icons can look like this: “img-src ‘self’ https://img-example.ch data:;”. This specific definition prevents any <img> or <link rel=“icon”> tags from loading pictures from any source that is not either local (“self”), the host “img-example.ch” reached via a HTTPS connection, or a valid “data:” URI. The semicolon marks the end of a policy directive and is necessary to seperate multiple directives. It can be omitted if no other directives follow. The specifics of how Content Security Policy works are described in the following section.

4.1.1 CSP Sources

The sources to be declared for a given directive can take one of five forms, depending on which directive they are specified for.

Default Sources

For most directives, a source can be either a host-source, a scheme-source, the string “self” or the string “none”.

A host-source is either a domain name or an IP address and can optionally include a URL scheme or a port number. Both the port number and subdomains can be specified using the wildcard “*”, which allows any value in that position. Examples for host-sources are:

- example.ch
- https://example.ch:3000
- ftp://*.example.ch
- sub.example.ch:*

A scheme-source consists of only a URL scheme name like “http:” or “ftp:”. Data schemes like “data:”, “mediastream:”, “blob:” and “filesystem:” can be used as well. The colon is mandatory and quotation marks must not be used. Such a scheme-source whitelists any source that uses the given URL scheme. Examples for scheme-sources are the aforementioned strings. Because there have been various security issues with “data:” URLs, developers are discouraged from specifying them in CSP directives, especially for the “script-src” directive.

The strings “self” and “none” are special values which must be enclosed in single quotation marks. While “self” whitelists all local resources (i.e. all resources with the same origin as the current page), “none” is the empty set which matches no resource at all. In some browsers, “blob:” and “filesystem:” resources are not included in “self”.

Special Sources for “style-src” and “script-src”

In addition to the possible sources that follow the default form, the “script-src” and “style-src” directives allow four more strings to be declared. Those are “unsafe-inline”, “unsafe-eval”, “nonce-<base64>” and “<hashing-algorithm>-<base64>”. For all of them, single quotation marks are necessary.

The “unsafe-inline” keyword allows the application of inline <style> tags and inline style attributes in case of the “style-src” directive. It can also be declared for the “script-src” directive, in which case it allows the execution of inline <script> tags and inline event handlers. The “unsafe-eval” string allows the execution of string evaluation methods for stylings, such as the “insertRule” method or assignments to the “cssText” attribute. When declared for the “script-src” directive, this keyword allows string evaluation for scripts, such as the “eval” or “window.setTimout” methods. A list of APIs that are affected by “unsafe-inline” and “unsafe-eval” can be found below in the descriptions of the “script-src” and “style-src” directives.

The nonce and hash keywords are alternatives to “unsafe-inline”, meaning that if a hash or a nonce is present alongside the “unsafe-inline” rule, the latter will be ignored. Nonces and hashes can be used to give trust to specific <style> or <script> tags. To use a nonce, a unique, base64-encoded value must be generated with every request and put into either the “style-src” or “script-src” directive. Furthermore, the “nonce” attribute must be set to that same value on every tag that shall be given trust. When using hashes, a hash value must be generated from the <style> or <script> tag’s content. That hash, prefixed by the applied hashing algorithm, must then be put into the “style-src” or “script-src” directive. Both of these methods will allow those trusted styles / scripts to be applied / executed. Starting with WebView version 59, this method can also be applied to <script> tags that load external Javascript. Examples for the nonce and hash sources are:

- style-src: “nonce-f65724a6b3”;
  to give trust to the tag <style nonce=“f65724a6b3”>
- script-src: “sha256-dd9fe856ba4d31a77ba85df53a0d77ef...”;
  to give trust to the tag <script>alert(“hash”);</script>

Further Special Sources for “script-src”

When specifying the “script-src” directive, there are three more keywords that can be declared. Those are the “unsafe-hashes” keyword, the “strict-dynamic” keyword and the "report-sample" keyword.

The "unsafe-hashes" rule along with a hash string as described above, allows the execution of event handlers that match the given hash. The advantage of this method is, that all other forms of inline scripts are still disallowed. The "strict-dynamic" keyword allows any script that is being trusted via a nonce or hash to load further scripts which will then receive the same trust themselves. This matches the expected behavior of developers when creating CSP whitelists: If a script from a given source is to be trusted, that source and any source from which further scripts are loaded must be explicitly whitelisted. With "strict-dynamic" that tediously recursive task is avoided. If the "strict-dynamic" keyword is present, all normal sources present in the "script-src" directive (i.e. any sources other than nonces, hashes or "unsafe-eval") are ignored. Finally, the "report-sample" keyword causes a sample string of any code that violates the declared policies to be included in the violation report generated by CSP.

Sources for "plugin-types"

The plugin-types directive does not use any of the sources described above. Instead, it accepts a string denoting a MIME type, such as "application/x-java-applet".

Directives Without Any Sources

There are some special directives in CSP which stand on their own and do not need declaration of further sources or keywords. Out of the directives relevant for this thesis, only the "upgrade-insecure-requests" directive falls in this category.

4.1.2 CSP Directives

There are currently 30 directives which can be set in a CSP definition. Some of those 30 directives are considered deprecated, while others are still experimental (i.e. not yet part of a finished CSP version). Since not all of those directives are relevant to this thesis, only a selection shall be described here. The directives we consider are those that are supported in WebView versions 52 and above and must be considered relevant for the security of mobile web applications. Directives like "frame-ancestors" — which may be useful, but not directly related to how secure a mobile web application can be — are omitted. The lower bound of WebView version 52 is chosen, because it is the first version which supports the "strict-dynamic" rule, which is essential for a modern definition of the "script-src" directive.

A more complete reference of CSP directives and their associated rules can be found in the MDN Web Docs [3] or the latest published version of W3C's working draft for

CSP3 [25]. However, not every API affected by a directive may be included in those references. For example, the description of the “img-src” directive in the MDN Web Docs does not include the fact that many cases of “url(...)” expressions in CSS lead to images being loaded and are therefore restricted by that directive.

Relevant Directives That Are Currently Supported

A comprehensive description of all directives we consider relevant can be found in appendix B. Only directives that are supported by Android’s WebView at the time of writing are included in this list.

Relevant Directives That Are Currently Unsupported

While the abovementioned directives are all supported by current WebView versions, there are two more directives in CSP level 3, which are relevant for WebView security, but not yet supported by the Android WebView. The first of those directives is “navigate-to”, which limits the ULRs a document is allowed to initiate navigations to. This can prevent a vulnerable WebView to be navigated to e.g. a phishing site, if an attacker was able to inject code that tries this. The other relevant directive is “trusted-types”. This directive restricts the values that known DOM XSS sinks can accept, which strongly reduces the chance of a successful injection attack against the application. The “trusted types” API is still under development. More information can be found in an “Update” post on Google Developers describing the implementation of this experimental API in Chrome [15].

4.2 How CSP Mitigates Attacks

The primary goal of Content Security Policy is to provide an additional layer of security against attacks directed at a web application. This is achieved primarily by blocking certain kinds of resources developers do not intend to use. The following section presents the most important ways in which CSP protects or mitigates attacks, thus answering our second research question: “How can CSP prevent or mitigate attacks?”

4.2.1 Preventing Code Injection

When using a strict set of rules, CSP prevents all untrusted inline scripts and inline event handlers from being executed. This means that an attacker that is able to manipulate the DOM is no longer able to add <script> tags or tags with inline event handlers added to them. Furthermore, string evaluation methods like Javascript’s “eval” are also blocked from execution by default. Since those are the primary ways of injecting Javascript code into an application, most injections can be prevented using strict CSP. However, if an attacker is able to inject data into the application via a trusted source, code injection is still possible. For mobile web applications, this can pose a risk, since local files almost always need to be able to execute code in such an app. As mentioned by Willocx et al. [24], this can lead to having to allow e.g. string evaluation for all local files, where a normal web app could specify a CSP definition that only allows string evaluation for a specific external source (where, for example, the jQuery library is hosted). As a consequence of this, an attacker could trick the browser into downloading a malicious file which could then be included in the DOM (see section 3.1.3 for details) and would be executed because of CSP allowing the source "self".

A more modern approach to defining the "script-src" directive can however avoid that problem. Since CSP level 2 allows the usage of nonces and hashes to give trust to specific scripts, developers can fully avoid the "self" keyword in their "script-src" directives. Additionally, this has become significantly easier to achieve with the "strict-dynamic" keyword. Using "strict-dynamic", developers no longer need to give trust to each script individually, but can propagate trust to scripts loaded from a trusted script, thus greatly reducing the number of nonces / hashes developers need to include in their "script-src" definitions. Starting with WebView version 59, hashes and nonces can be used with external scripts as well. For older versions, trust could theoretically be given to such scripts by defining a trusted script that loads those scripts dynamically. However, when we investigated this workaround, it did not yield reliable results. For more details on this workaround, see section 5.2.4

4.2.2 Preventing the Loading of Arbitrary Data

By restricting the sources that can be loaded from by <img> tags, "XMLHttpRequests" or other methods, CSP prevents attackers from loading arbitrary data into the application. This effectively mitigates cross-site scripting, since the inclusion of remote, attacker-controlled code is not permitted. Furthermore, the HTML tags restricted by "object-src" can be used to trick a plugin into executing Javascript code, effectively creating a loophole for code injection attacks developers might think they have closed by blocking inline scripts.

4.2.3 Preventing the Exfiltration of Data

If an attacker is able to inject code into an application, they might attempt to read and exfiltrate sensitive data such as credentials or personal information. Besides the obvious "XMLHttpRequest" and related methods, this can be achieved by creating e.g. an <img> tag which "loads" from a URL that consists of an attacker-controlled domain name and the sensitive data included into the URL's query parameters:

```
<img src=``https://evil.com?data=someSensitiveData''>
```

The attacker would then only have to look at their server logs, where an entry from the attempted connection would be present, including the sensitive data inside the query parameters. By restricting the sources for all kinds of HTML tags that create such requests, this exfiltration method can be prevented.

4.2.4 Preventing UI Manipulation

By blocking inline styles and string evaluation for styles, CSP prevents attackers from arbitrarily positioning and styling UI elements in the page. As described in section 3.1.3, this could be used to trick the user into entering credentials either into a completely new, trustworthy-looking interface, or into invisible form fields rendered on top of existing ones.

4.2.5 Preventing Insecure Connections

While HTTPS is quite common in today's web, developers might still include HTTP domains in their code by mistake. If such a request is used to load an external script or similar resource, an attacker might perform a man-in-the-middle attack. This would allow them to include malicious data like Javascript code into the response, effectively leading to code injection. By upgrading all HTTP requests to HTTPS when using the "upgrade-insecure-requests" directive, CSP can prevent this vector of injection.

5 Retrofitting Cordova Applications for CSP

As reported by Willocx et al. [24] and Weichselbaum et al. [23], the adoption rate of Content Security Policy is low for web applications in general and Cordova applications in particular. Besides increasing awareness about the benefits of CSP amongst developers, retrofitting existing applications remains a challenge to be overcome in order to protect users against code injection attacks. To tackle this issue, automatically generating CSP definitions for any given application might prove to be a valuable contribution which would help developers harden their apps with little to no effort. To assess the feasibility of this concept, we propose a tool which accepts the source code of an existing Cordova application as input, and adds a CSP definition that is as strict as possible while not breaking any functionlity of the app. Since there are certain patterns which prevent some of the most beneficial CSP rules, our solution attempts to rewrite these patterns whenever possible. In the following chapter, the approach of our tool is described, along with its limitations and an evaluation of its applicability to real-world Cordova applications.

5.1 Approach

The purpose of our proposed tool is to add a sensible CSP definition to every HTML file present in the given Cordova application. To be able to properly assess the feasibility of this, we include every CSP directive that is relevant to Cordova applications and is part of CSP level 3. For a list of those directives, refer to section 4.1.2. The tool can be divided into two preliminary steps, six main steps and three I/O steps. In the preliminary steps, all HTML files present in the application are gathered and iterated over. For every such iteration, five of the main steps are executed, while the sixth main step does not depend on specific HTML files and is therefore only executed once. In the main steps, the application's code is analyzed and rewritten, and the individual CSP rules are generated from the code. Finally, the three I/O steps are invoked whenever writing to disk is necessary, with the most important writing being performed when the combined CSP definitions are added to the respective HTML files. A schematic overview of the tool's functionality is shown in figure 5.1. Each main component is described in detail in the following section, which also answers our research question 3.1: "Can we rewrite real-world Cordova apps to allow more strict CSP definitions?"

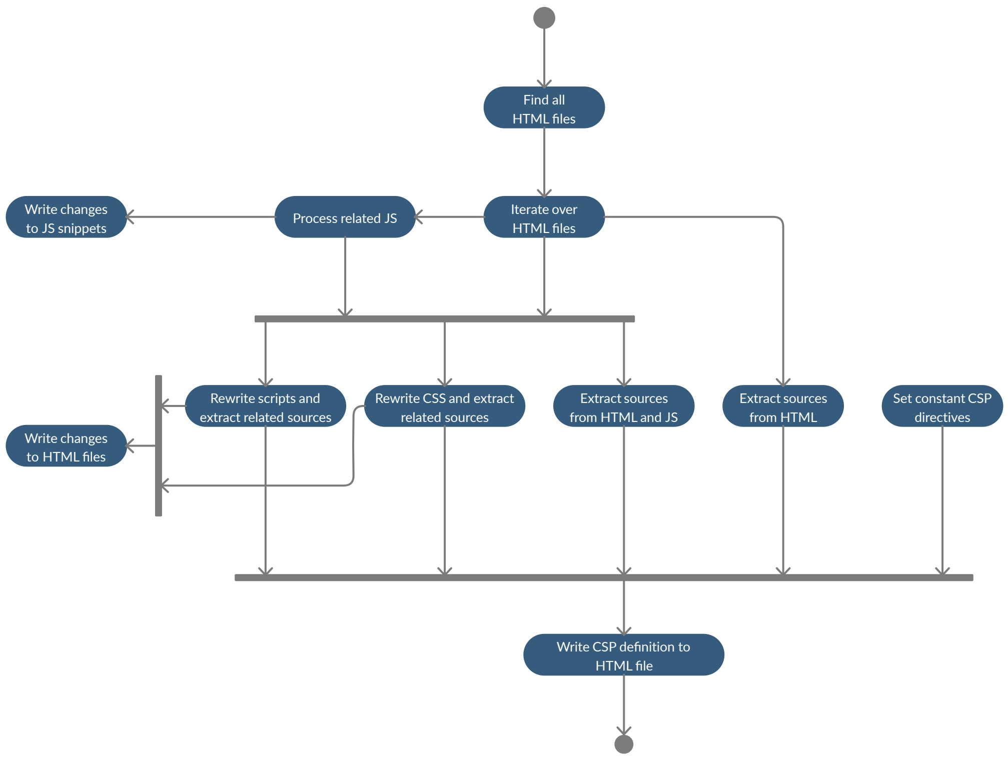
Find all HTML files
Write changes to JS snippets
Process related JS
Iterate over HTML files
Write changes to HTML files
Rewrite scripts and extract related sources
Rewrite CSS and extract related sources
Extract sources from HTML and JS
Extract sources from HTML
Set constant CSP directives
Write CSP definition to HTML file

Figure 5.1: The workflow of our implementation

5.1.1 Processing Related Javascript

As described in 4.1.2, several CSP directives are directly related to certain Javascript APIs which may or may not be blocked from execution by the set CSP definition. In order to choose what sources to set on those directives, we have to analyze all Javascript snippets which can be executed in relation to the given HTML file. Furthermore, there are some Javascript APIs that are restricted by CSP (refer to appendix B for a detailed list). When such APIs are used, we are forced to lower the strictness of our CSP definition by allowing string evaluation or inline <style> tags. To avoid this, we rewrite these APIs whenever possible and replace them with equivalent alternatives that are not affected by CSP.

The component starts by extracting all <script> tags in the current HTML file and dividing them into inline scripts (i.e. scripts whose code resides entirely inside the HTML markup) and external scripts (i.e. scripts which are loaded either from a local file or from a remote resource). For all the external scripts, we load their actual content using the reference in the tag's "src" attribute. Furthermore, we find all inline event handlers present in any HTML tag and extract their code (i.e. the content of their "onclick" / "onload" / ... attribute). For each of those snippets we have now gathered, the following steps will be executed.

Generating Hash Values

First, we generate a hash over the current JS snippet. We use this hash as the key to an associative array in which we store all the data related to this script. By doing so, we only have to process each snippet once and can easily retrieve the data related to a snippet later on in the workflow.

Extracting All URLs

After computing the hash, we find all strings in the snippet that look like valid URLs and store them in the aforementioned associative array. For this, we use a regular expression. Because in-depth flow analysis is out of scope for our tool, we whitelist all present URLs for some of the directives. Although this invariably leads to some inflation of our CSP definition, not including those URLs will often lead to broken applications.

Parsing the Code

Then, we parse the snippet using the "esprima" Javascript parser and iterate over every node of the generated abstract syntax tree. For every node, we check whether it matches one of the expressions that are interesting to us.

Extracting Variable Assignments

First, we check if the current node is an assignment expression. If it is, we store the identifier along with the value assigned to it. We will use this data later on to try to resolve any variable used in another interesting expression. Because we do not apply actual flow analysis, this naive approach will only yield results for variables that are assigned in the immediate context of their usage.

Extracting Sources for "connect-src"

After that, we check if the current node matches an expression that is relevant for the "connect-src" directive: If either a "new WebSocket", "new EventSource", "navigator.sendBeacon" or "XMLHttpRequest.open" expression is present, we extract the URL parameter and try to resolve it. If we can resolve it, we add it to the "connect-src" directive in the associative array.

Extracting Sources for "worker-src"

Then, we check for expressions related to the "worker-src" directive. If a "new Worker", "new SharedWorker" or "navigator.serviceWorker.register" expression is present, we once again try to resolve its parameter and add it to the "worker-src" directive in our associative array.

Rewriting and Hashing "style-src" APIs

If the current node matches an expression relevant to the "style-src" directive, we attempt to rewrite it. There are three types of expressions we have to rewrite.

Expressions of the forms:

```
// setAttribute
a.setAttribute(``style'', ``display:none'');

// cssText
a.style.cssText = ``background-color:none'';

// insertRule
stylesheet.insertRule(``#someId { color: white }'', 0);
```

will be changed to:

```
// setAttribute
(function(a) { a.style.display = ``none''}());
```

```
// cssText
a.style.backgroundColor = ``none'';

// insertRule
var newStyleTag = document.createElement(``style'');
newStyleTag.innerText = ``#someId { color: white }'';
document.head.appendChild(newStyleTag);
```

This has to be done because the three original expressions are restricted by CSP. If we were to keep the "setAttribute" expression, we would have to add "style-src unsafe-inline" to our CSP definition in order to not break any functionality. The same is true for the "cssText" and "insertRule" expressions, which would need the "style-src unsafe-eval" rule. Since we want to avoid both "unsafe-inline" and "unsafe-eval" to achieve a CSP definition that is as restrictive as possible, we replace the three expressions with equivalent alternatives that are not affected by CSP.

In the "insertRule" case, in addition to rewriting the expression in the Javascript snippet, we have to generate the hash value of the CSS string we add to the newly created <style> tag and add it to the "style-src" directive. Of course, all of the abovementioned rewrites are only possible if we can actually resolve the method arguments or the right hand side of the assignment expressions. If those variables are not constant or not immediately available from the context (which we have stored by extracting all assignment expressions), we have no choice but to add "unsafe-inline" (for the "setAttribute" method) or "unsafe-eval" (for the "cssText" assignment) to the "style-src" directive.

One special case occurs when the "document.createElement" method is called to create a <style> tag. If this is the case, we have to search for all expressions that add CSS code to that newly created tag, in order to generate hashes for the CSS code. This is done because we want to give trust to this specific inline style, while still disallowing all other, non-trusted inline styles. We consider two ways of adding content to such <style> tags: The "innerText" property and the "innerHTML" property:

```
var newStyleTag = document.createElement(``style'');
newStyleTag.innerText = ``#someid{background-color:blue;}'';
newStyleTag.innerHTML = ``#someid{background-color:red;}'';
```

If we can find any of those assignment expressions for a newly created <style> tag, we compute the hash of the right hand side of the expression and add that hash to the "style-src" directive. This is necessary because unlike the "script-src" directive, "style-src" does not allow the "strict-dynamic" keyword, which is why propagation of trust is not possible for <style> tags, even when created from trusted scripts. If the right hand side of such assignments is non-constant, we have to add “unsafe-inline” to our “style-src” directive, or else the application will break.

Rewriting “script-src” APIs

Finally, we check if the current node is an expression that is relevant for the “script-src” directive. This is the case for “eval”, “setTimeout”, “setInterval”, “setAttribute” and “new Function” expressions. We rewrite them as follows.

“eval” expressions like:

```
// eval with code
eval('alert(``Example'')')

// eval with JSON
eval('{``key1'': ``value1'', ``key2'': ``value2''}')
```

are rewritten to:

```
// eval with code
alert(``Example'')

// eval with JSON
JSON.parse('{``key1'': ``value1'', ``key2'': ``value2''}')
```

We differentiate between those two cases because the “eval” method is sometimes used to parse JSON-formatted strings. This rewrite is necessary, because the “eval” function is restricted by CSP and could only be executed if we specified “script-src unsafe-eval” in our definition. Because this would reduce the strictness of our CSP definition, we replace the “eval” function with equivalent expressions.

For the “setTimeout”/“setInterval” functions that have the following form:

```
setTimeout('alert(``Example'')', 1000)
setInterval('alert(``Example'')', 1000)
```

we make modifications in order to get:

```
setTimeout(function() {alert(``Example'')}, 1000)
setInterval(function() {alert(``Example'')}, 1000)
```

Of course, we only perform this rewrite if the first argument is or resolves to a string. If it is or resolves to a function, then this expression is not impaired by the “script-src” directive and can be left as it is. As with the “eval” function, this rewrite has to be performed because the “setTimeout” and “setInterval” APIs are restricted by CSP when called with string arguments. Because of that, their functionality could only be preserved if the “script-src unsafe-eval” CSP rule was added, which is something we attempt to avoid.

Finally, “new Function” expressions like:

```
new Function(``a'', ``b'', ``return a*b'')
```

are rewritten to:

```
function(a, b) {return a*b}
```

Again, this rewrite is done because CSP blocks the “Function” constructor unless “script-src unsafe-eval” is specified. Since it is our goal to be as restrictive as possible, we replace the “Function” constructor with the “function” expression, which is not restricted by CSP.

In addition to the aforementioned expressions, it is possible to add inline event handlers from Javascript in a way that does not cause trust to be propagated from scripts secured with hashes or nonces. This is the case when invoking the “setAttribute” API to add an event handler attribute. Consider the following code:

```
a.setAttribute(``onclick'', `alert(``XSS'')')
```

We will rewrite this snippet to:

```
(function(a) { a.onclick = function() {alert(``XSS'')} }())
```

In its rewritten form, this event handler will inherit the trust we later give to the script where it is created. In its original form, this trust propagation will not happen, which is why we would be forced to add “script-src unsafe-inline” to our CSP definition in order to not break any functionality.

As for the expressions relevant to “style-src”, we only perform the abovementioned modifications if the respective arguments are either a constant string or resolve to one. If this is not the case, we are forced to add “unsafe-inline” (in case of the “setAttribute” API) or “unsafe-eval” (for any other API) to the “script-src” directive in our associative array.

Writing the Changes

After checking for and potentially performing those rewrites, we have successfully processed the script. If there were actual changes to the script, we write those to their respective files and update the script’s hash value in all associated entries so we will be able to retrieve the right data later on.

When we have processed all the Javascript snippets related to the given HTML file, we can proceed to setting the actual CSP directives. Depending on which directive we have to set, there are five different approaches to take.

5.1.2 Setting Constant CSP Directives

In the most simple case, we do not have to compute anything in order to set a specific CSP directive. This is the case for the "default-src" directive, where we will always set "self" as its only source. And for the "upgrade-insecure-requests" directive, which does not need any specification other than the directive name itself. We set those two directives directly in a global associative array which is used at the very end of the loop to generate the policy for the current HTML file.

5.1.3 Extracting Sources from HTML

In the most simple of the non-constant cases, all information needed to set a CSP directive is present in the HTML file itself. This is the case for the directives "manifest-src", "base-uri", "plugin-types" and "form-action". In all of those cases, the only thing we have to do is look for the sources specified in the tag attributes relevant to them (e.g. for "base-uri", we have to look at the "href" attribute of any <base> tag in the markup). A detailed description of what tags and what attributes we have to look for can be found in 4.1.2. Again, we store the information extracted in this way in our global associative array.

5.1.4 Extracting Sources from HTML and Related Javascript

For the "connect-src" and "worker-src" directives, we not only have to look at the HTML file, but retrieve some of the data gathered in the first step of our pipeline as well. In addition to extracting the sources in the respective tag attributes, we retrieve all the sources stored while processing the Javascript snippets related to the current HTML file.

Because testing revealed many cases of dynamically added tags which referenced sources not included in the generated CSP definition, a lot of our test applications broke. Because these sources were in many cases not resolvable from the immediate context, in-depth flow analysis would be required to accurately tackle this problem. As an approximate solution, we decided to add all URLs found in the related Javascript snippets, along with "self" to all directives related to tags which are prone to dynamic insertion. Following that decision, we add those sources to the directives "connect-src", "frame-src", "img-src", "media-src", "object-src" and "worker-src". While this adds a lot of unnecessary sources to those directives and leads to very bloated CSP definitions as a whole, we deemed it necessary for our tool to not break the processed applications.

5.1.5 Rewriting CSS and Extracting Sources Related to Stylings

Some more work has to be done for the CSP directives related to the CSS code found in our HTML file. As with all non-constant directives, we first gather the sources from all relevant tag attributes present in the HTML file itself and store them for the “style-src” directive. Then, we locate any tags with inline style attributes, assign an id to them if one is not already present, create a new <style> tag which assigns to that id the CSS rules from the inline style attribute, and then remove that inline style attribute completely. This rewrite is shown in the following example.

The tag:

```
<div style=``height: 100px;''>
```

is rewritten to

```
<div id=``tmYuSGfL''>
<style>#tmYuSGfL { height: 100px; }</style>
```

This is done because the original tag contains an inline style attribute, which is blocked by CSP unless the “style-src unsafe-inline” rule is specified. Since we want to avoid this, we move the styling into its own tag.

After that, we locate all inline <style> tags and compute hashes of their contents. Those hashes are added to the “style-src” directive in our global associative array, thereby giving trust to the inline <style> tags the correspond to. We then retrieve the sources related to “style-src” we have collected from the Javascript snippets and add those as well. Finally, we iterate over all CSS rules related to the current HTML file and do three things.

First, we find any “@import” rules and add their “href” attribute to our “style-src” directive. This is necessary, because with an “@import” rule, further stylesheets from origins we might not be aware of yet can be loaded. Then, we look for “@font-face” rules and add their “src” attribute to the “font-src” directive. Lastly, we locate any CSS rules which are able to add images to the DOM and have a value which matches the regular expressions “url.$*$?”. Then we strip away the parentheses and the substring “url” and analyze the resulting URL. If it refers to a local file loaded via a remote stylesheet, we add a reference to that remote stylesheet to the “img-src” directive. Else, we add the resulting source itself to “img-src”. We have to extract those CSS rules adding images, because as mentioned in section 4.1.2, the “img-src” directive also restricts images loaded by CSS code. This is especially noteworthy, because that behavior is not mentioned in the MDN Web Docs [3].

5.1.6 Rewriting Scripts and Extracting Sources Related to Them

In the last component of our tool, we have to process all the <script> tags and inline event handlers present in the HTML file. We do this by first finding all tags with inline event handlers attached to them, creating a new <script> tag which adds the same event handler with the “addEventListener” function, inserting this <script> tag into the DOM just after the tag it refers to, and finally removing the original inline event handler. As with inline style attributes, we generate an id for the tag if one is not already present. This rewrite is illustrated in the following example.

If we have a tag with an inline event handler like:

```
<div onclick='alert(``Example'')'>
```

we rewrite this to:

```
<div id=``tmYuSGfL''>
<script>
document.getElementById(``tmYuSGfL'')
  .addEventListener(
    ``onclick'',
    function() { alert(``Example'') }
  )
</script>
```

As with inline style tags, inline event handlers as present in the original tag are blocked by CSP by default. This behaviour can be changed by adding the “script-src unsafe-inline” rule or by adding the event handler from a trusted script. Because we want to be as restrictive as possible, we use the second approach and rewrite all tags with inline event handlers accordingly.

After rewriting all inline event handlers, we collect all <script> tags which contain inline Javascript or reference external resources. We then compute their hash values and add them to the “script-src” directive in our global associative array, thus giving them explicit trust. For external scripts, we add those hashes to their “integrity” attribute as well. Then, we retrieve any sources we have set for “script-src” when processing the related Javascript snippets at the beginning of our pipeline and add them as well. Finally, we add the “strict-dynamic” keyword to enable propagation of trust to any scripts loaded by one of the scripts we have just secured with hashes.

5.1.7 Writing the Generated CSP Definition to the File

After having executed all of the abovementioned components, we now have generated a complete Content Security Policy definition which includes all directives relevant to our Cordova application. Furthermore, we have rewritten all expressions where rewriting was possible. As the final step of our program, we compose all CSP directives now generated to a complete CSP definition in the form of an HTML <meta> tag. This tag is then written into the <head> section of the current HTML file.

5.2 Evaluation

To evaluate our tool, we downloaded the latest version of every application available on the F-Droid store as of 27.11.2019. We then filtered out all the non-Cordova applications, which left us with 22 apps in total. 6 of those apps used the Ionic framework, which is built on top of Cordova. Out of those 22 applications, we successfully deployed and ran 10 (2 built with Ionic). The remaining 12 apps could not be run, mainly due to version incompatibilities we could not resolve. Together with two applications from GitHub, those formed our 12 manual test cases which we used to analyze our tool in detail. While our tool performed well for small applications that do not make heavy use of UI frameworks or libraries, apps that did use such dependencies were much more likely to break after our modifications. As a result of our analysis, we identified a number of patterns which prevent, significantly complicate or weaken the strictness of our automatically generated CSP definitions. These patterns are discussed in the following section, thereby answering research question 3.2: “What patterns limit us in rewriting applications and generating CSP definitions?”

5.2.1 Limiting Patterns Inherent to Content Security Policy

Out of all the Javascript APIs affected by Content Security Policy, there is a single API that does not have a full, CSP-compatible equivalent that we can statically replace it with. Because of that, we theoretically have no choice but to weaken our CSP in order to allow its execution. This is however a theoretical limitation we did not encounter in our test cases, which is why we sill attempt to rewrite the API with another one that is almost equivalent.

Functions Created with “new Function” Execute in Global Context

The problem with the “Function” constructor lies in the fact that it creates functions which will be executed in the global context of the application. Since “Function” relies on string evaluation and is therefore affected by the “script-src” directive, our tool attempts to rewrite any occurrences of the “Function” constructor to a regular “function” definition.

However, if a use case relies on having a specific function executed in the global context only, this rewrite would cause that constraint to be violated. While we could not find such a use case in any of the real-world apps we investigated, it remains a problem for our approach that the "function" expression is not a full equivalent to the "Function" constructor.

5.2.2 Limiting Patterns Inherent to Our Approach

There are three main limitations that come with our approach to generating CSP definitions, each of them related to some of the patterns listed below. They are caused by three design decisions we had to make for our tool. First, we do not perform any proper flow analysis on the Javascript snippets, which limits our rewriting abilities to APIs with constant parameters. Second, we do not consider any of the various methods for adding HTML elements to the DOM by adding strings which contain valid HTML markup. And third, we do not attempt to deal with any special syntax used by the various templating engines and UI frameworks possibly used amongst Cordova applications.

Adding Event Handler Attributes with Non-Constant Parameters

When event handlers are added to an HTML tag via the "setAttribute" method, CSP will block execution, even when the "strict-dynamic" keyword is specified and the originating script is secured with a hash or a nonce. As described above, we can rewrite this pattern if it is called with a constant parameter. If, however, the parameter is non-constant, we have no choice but to add "unsafe-inline" to the "script-src" directive. This is the only case we identified (apart from "implicitly" adding tags with event handlers, as described below) where we cannot avoid the "unsafe-inline" keyword for the "script-src" directive. If this pattern is present, the strictness of the generated CSP definition is greatly weakened, since the blocking of inline scripts and event handlers is considered to be the most important benefit of any Content Security Policy.

Calling Affected APIs with Non-Constant Arguments

As mentioned in the description of our approach, there is a number of Javascript APIs relying on either string evaluation or being allowed to add inline tags or attributes. This forces us to loosen the strictness of our CSP definition if those APIs are called with non-constant arguments and therefore cannot be rewritten. This group of APIs consists of the "eval", "setTimeout", "setInterval", and "Function" APIs for the "script-src" directive and the "createElement" "insertRule" and "cssText" APIs for the "style-src" directive. If any of those APIs are used with non-constant variables, we are unable to statically rewrite them witout adding some sort of metaprogramming to the Javascript code itself, which is exactly what we intend to avoid by rewriting the string evaluation APIs in the first place. Therefore, the presence of such APIs must be considered a limiting pattern for our approach.

Adding Inline Styles or Scripts Implicitly from Strings

Another limiting pattern we did not come across but could theoretically exist, is adding strings to the DOM which contain inline styles or event handlers. If for example, an element is added to the DOM by calling one of the following functions or an equivalent DOM API:

```
document.getElementById(``someid'')
  .innerHTML = ``<style>...</style>'';

document.getElementById(``someid'')
  .innerHTML = `<div style=``...''>some content</div>';

document.getElementById(``someid'')
  .innerHTML = `<div onclick=``...''>some content</div>';
```

then all of them are blocked by CSP. This can pose a problem to our approach because we do not consider such ways of "implicitly" adding HTML markup to the DOM. Any inline styles, style attributes or event handlers added like this will therefore not be caught by our tool and cannot be rewritten or secured with a hash. This pattern does not apply to <script> tags added this way, since those are not executed when added to the DOM as a string. Note that the way we added the inline event handler to the DOM in this example will not cause trust to be propagated to the event handler, even if the originating script is secured with a hash or nonce and the "strict-dynamic" keyword is present. This is contrary to the behavior of event handlers added with the following method:

```
document.getElementById(``someid'')
  .onclick = function() {...};
```

In this case, any trust given to the originating script is propagated to the event handler and it will not be blocked from execution by CSP.

Usage of HTML Templating Engines

A problem we encountered while testing our implementation was caused by HTML templating engines. Five of our test applications made use of such libraries, namely the "Onsen UI" framework, the "Ionic" framework, the "RiotJS" framework and the "AngularJS" framework. The usage of such frameworks introduces two kinds of problems for our implementation.

On the one hand, they allow developers to define templates, which are fragments of HTML markup that can be inserted into other HTML files by referencing a string defined for each template. One example of such a reference is RiotJS's special usage of <script> tags, which is explained in section 5.2.4. The problem we encounter with such templating mechanisms is that we cannot easily tell what HTML fragments are included into which "complete" HTML document. Because of that, it is made significantly harder to include all necessary sources into a CSP definition for a given HTML document. While there is always some kind of reference to the templates to be inserted, the syntax used to do this and the location of the referencing code or markup differ across frameworks. Therefore, any approach for statically generating CSP definitions would have to specially consider each framework's syntax, which greatly increases the amount of work needed to handle such applications. The rewriting of CSP-incompatible patterns like inline event handlers in template files is however not affected by this.

The second problem which comes from templating frameworks is that they often include some sort of mechanism for evaluating (mostly Javascript) expressions inside HTML files. In the RiotJS framework, expressions are evaluated when placed inside single curly braces, while AngularJS does so for code placed inside double curly braces. This impacts our pipeline because it breaks standard HTML parsers. Even though both of these patterns occured in our test applications, only the first one actually caused two of our rewritten applications to break. In both cases, we were missing sources in our CSP definition which could have been detected, had we considered the templates loaded into the respective HTML documents.

Inability to Rewrite Remote Resources

One limitation which is inherent to our rewriting approach but cannot easily be overcome is the loading of remote resources. If this happens, we are not able to rewrite any remote Javascript snippets. The only workaround to this is to download the resources and include them locally. This, however, may not be what developers want and doing this should not be considered without asking or at least notifying them.

5.2.3 Limiting Patterns Inherent to Specific Use Cases

For some applications, there are specific use cases which make it impossible to statically set a non-breaking Content Security Policy. In our analysis, we came across two such use cases.

Applications That Connect to User-Defined URL

One particularity which prevented us from statically generating a working CSP definition was present in two of our test cases. Both applications had the use case that they made connections to URLs which were dependent on the user. One of the cases is the "friimaind.piholedroid" application, which allows users to connect to their "Pi-hole", which is a service designed for the RaspberryPi that acts as an adblocker for the local network. With this app, users could monitor the behavior of their Pi-hole and retrieve statistics about blocked content. To configure the app, users have to input their Pi-hole's IP address. Because this address depends on the specific configuration of any Pi-hole setup, it is impossible to statically add it to the CSP definition.

The other case we came across was the "org.iilab.openmentoring" application. This app allows users to retrieve educational content to view inside the application. While this content is per default fetched from the URL "http://openmentoring.io", users can configure the app to connect to any valid URL. Therefore, even though we can statically include the default URL in our CSP definition, any changes the user makes to the configuration will cause the application to break.

In both of those cases, the only possible way to ensure non-breaking behavior would be that developers include appropriate logic in their application which dynamically adds the respective URLs to the app's CSP definition. Because of that, our tool is not able to properly define a non-breaking Content Security Policy for those use cases.

Applications That Connect to Dynamically Chosen API Endpoints

Another use case where our static generation approach fails, is when API endpoints are not fully known in advance. This is the case for the "io.github.whirish.tvbuses" application. This app displays a map and allows users to view bus connections for certain buses in parts of Ireland. It receives the map to display from the "mapbox.com" service. To do this, the app connects to API endpoints which are subdomains of "mapbox.com", specifically the two URLs "a.tiles.mapbox.com" and "b.tiles.mapbox.com". While "a.tiles.mapbox.com" was present in the files loaded by the application, "b.tiles.mapbox.com" was not. We assume this second URL to be chosen dynamically by the mapbox service, depending on the load each of the endpoints experience at a given time. While the documentation for the "Mapbox" service [2] advises developers to add "https://*.tiles.mapbox.com" to their "connect-src" directive, this information cannot be retrieved from the source code or any loaded resources themselves. Human intervention is therefore required to properly set a CSP definition, which is why our approach causes such applications to break.

5.2.4 Further Discoveries

In addition to the abovementioned limitations, we made two interesting observations. One concerns a method of propagating trust for external scripts when using older Chrome versions. And the other is about a contradictory error message displayed by Chrome in connection with the “unsafe-hashes” keyword.

Propagating Trust for External Scripts in Chrome Versions Below 59

With the introduction of the “strict-dynamic” keyword in Chrome version 52, specifying the “script-src” directive was made much easier. Besides not having to recursively search all loaded scripts for more loaded scripts and whitelisting all those resources explicitly, it allows developers to avoid both the “unsafe-inline” and the “self” keyword, both of which would negatively impact the benefits of adding a CSP definition to an application. However, before Chrome version 59, it was not possible to give explicit trust to external scripts via nonces and hashes, which also made it impossible to propagate such trust via “strict-dynamic”. As a workaround, an early version of our tool attempted to rewrite script tags that refer to external resources by turning them into inline script tags which could be secured with a hash and given trust to explicitly. This rewrite is illustrated by the following example.

Assume we have a number of script tags which load some Javascript code from external resources:

```
// External script
<script src=``../example.js'' defer></script>

// External RiotJS tag
<script src=``https://example.ch'' type=``riot/tag''></script>
```

We would then create a new inline script tag in which a loader function is defined and calls to that function are made for every external resource we want to load. When called, the loader function dynamically creates the same script tag previously present in the code. Since we can secure the resulting inline script tag with a hash, trust is propagated to the dynamically created external script tags.

The resulting code would then look like this:

```
<script>
// Loader function
function loadScript(url, type, async, defer, charset) {
  var s = document.createElement(``script'');
  s.src = url;
  if (type) {
      s.type = type;
  }
  if (async) {
      s.async = true;
  }
  if (defer) {
      s.defer = true;
  }
  if (charset) {
      s.charset = charset;
  }
  document.head.appendChild(s);
}

// External script
loadScript(``../example.js'', ``'', ``'', True, ``'');

// External RiotJS tag
loadScript(``https://example.ch'', ``riot/tag'', ``'', ``'', ``'');
</script>
```

We include all attributes that must be expected on script tags. This is especially important, because they often contain information vital to the proper functioning of the page as a whole. The 'type="riot/tag"' attribute present in the example illustrates this well: This type is part of the RiotJS library and is present on script tags which — contrary to what one might expect — do not load any Javascript at all, but refer to a template defined in RiotJS's own template syntax. If we were to omit this "type" attribute, the application would fail.

After implementing this compatibility workaround, we tested it with our test data set. Unfortunately, this workaround broke 5 of our 12 test cases. Most of those breaks seemed to occur because of timing problems. In one application, there were four external script tags present, along with one inline script tag. All five of those scripts depended on each other, requiring them to load in a specific order for the application to work. While that order was guaranteed with the external script tags left in place, our workaround could not hold that requirement, causing some scripts to load before their dependencies, which broke the application completely.

Although one might argue that such timing constraints should be considered bad style, we did not find this behavior to be an exception. For the sake of guaranteed functionality, we therefore abandoned this attempt at increased compatibility and returned to hashing external scripts while requiring Chrome version 59 or above for our CSP definitions to work.

Error Messages Concerning “unsafe-hashes” for Inline Style Attributes

Another discovery was made regarding the “unsafe-hashes” keyword. This keyword can be used to give trust to inline event handlers without having to specify “unsafe-inline” and thus allowing all such scripts to execute everywhere on the page. As the latest published version of the W3C Working Draft for CSP level 3 [25] states, the “unsafe-hashes” keyword should apply to inline style attributes and “javascript:” navigation targets as well. We discovered that Chrome does not conform to this description in versions 69 and 74; the “unsafe-hashes” keyword only applies to inline event handlers. While the W3C Working Draft does explicitly say that the section concerning the “unsafe-hashes” keyword is not considered normative, there is another reason for this behavior to be problematic. As we discovered while testing an application that adds new tags to the DOM which contain inline style attributes, Chrome reports having blocked such style attributes by suggesting the insertion of the appropriate sha256 hash values, which is the exact behaviour Chrome does not support. This error message therefore has the potential to cause confusion amongst developers. The solution here would obviously be to either allow securing inline style attributes with hashes when the “unsafe-hashes” keyword is present, or to remove the futile suggestion of adding such hashes to the CSP definition.

5.2.5 Preliminary Conclusion

Our tool attempts to generate a CSP definition that is as strict as possible for any given pre-existing application. To achieve that goal, it performs rewriting on the Javascript APIs which are affected by Content Security Policy. While the tool is successful in its endeavor for small applications that do not include any UI frameworks, it has its difficulties with more complex apps. The biggest shortcoming when rewriting applications is the fact that no in-depth flow analysis is performed, which would allow us to significantly increase the number of affected APIs we can modify to support a stricter CSP definition. Furthermore, because we do not perform flow analysis to gather the exact URLs passed to the Javascript APIs affected by Content Security Policy, we have to include the (assumed) superset of all URLs found. While this greatly reduces the chance that modified applications break, our resulting CSP definitions tend to be heavily bloated for applications that rely on a lot of external resources. Because we add all found URLs to up to six directives, our final CSP definition includes a lot of unnecessary sources for many directives. Also, the generated CSP strings tend to get very long for certain applications, which also reduces clarity when read by humans. The second big shortcoming of our implementation is the fact that it cannot deal with the special syntax introduced by UI frameworks. Because such frameworks are very common, this has a big negative effect on our ability to analyze and rewrite real-world applications.

That being said, the tool shows that it is possible under the vast majority of circumstances to avoid the "unsafe-inline" keyword for the "script-src" directive. This is undeniably the biggest benefit of any Content Security Policy. Furthermore, we were able to completely avoid the "self" keyword for the "script-src" directive, which greatly reduces the chance of successful CSP circumvention via local file inclusion attacks, as explained in section 4.2.1. However, there are certain use cases which introduce limitations that any static generation of CSP will fail to address. When an application connects to user-dependent resources or dynamically chooses its API endpoints, no possible extension of our approach will be able to overcome this limitation.

In summary it can be said that while our proposed tool successfully adds CSP to smaller applications, it has a non-negligible chance of breaking larger ones and will never be able to deal with certain use cases. Preliminarily answering our third research question ("Can we automatically generate sensible CSP definitions for real-world Cordova apps?"), we must conclude with a mostly negative result. Only a significantly extended approach that includes in-depth flow analysis and can deal with at least the most common UI frameworks would be able to meet the basic requirements for such a utility, when applied to non-trivial applications in the real world.

6
Empirical Analysis

To assess the prevalence of the patterns our tool attempts to rewrite, we perform an empirical analysis on a data set of 1000 Cordova applications downloaded from the AndroZoo collection [4]. We acquired the data set by downloading applications at random, checking if they are apps built with the Cordova framework, and repeating this process until we had gathered 1000 APKs in total. The downloading process was constrained by two conditions: First, we only downloaded applications originating from the Google Play Store. Second, we ignored any applications with a dexdate older than three years. We then decompiled all the APKs, and ran our tool against the resulting source codes. 175 applications could not be parsed properly, which left us with metrics for 825 apps.

Statistics were generated for all Javascript APIs that our tool attempts to rewrite. We analyze all statistics once with all files included and once with files excluded that have either “cordova”, “jquery”, “angular” or “ionic” in their names. We consider such files belonging to the respective third-party libraries and exclude them, because we cannot consider them to be under full control of the developers. Developers may not want to change such files because they are subject to change when updating an app’s dependencies, and the developers of these libraries may deal with CSP-related refactorings on their own. It can therefore make sense to exclude those files from our analysis when assessing the effectiveness of our tool. The results of the empirical analysis are presented in this chapter.

6.1 Occurrence of Impacted APIs

First we analyzed the overall occurrence of the Javascript APIs affected by Content Security Policy, thus answering our fourth research question: “How prevalent are the patterns we attempt to rewrite?”

As can be seen in table 6.1, prevalence differs greatly between the individual APIs. While either one of the “setTimeout” or “setInterval” methods was present in almost all of the analyzed applications, the “insertRule” APIs are barely being used. The high occurrence of the “eval” function is notable, since it negatively impacts both performance and security of an application, as Richards et al. [20] have discussed.

When we exclude all files belonging to one of the commonly found third-party libraries, we can see that those make a big difference for the “eval”, “setTimeout”/“setInterval”, “Function” and “cssText” APIs and a slightly smaller one for the “setAttribute” API when used to set inline style attributes. This means that a significant number of applications make no use of those APIs in their actual application code and policies are often impaired because of the libraries an application uses.

API	% Apps	% Apps (w/o library files)
setAttribute (event handler)	3.88	3.31
eval	89.45	36.89
setTimeout / setInterval	98.30	73.65
Function	61.21	29.41
setAttribute (style)	30.91	21.45
cssText	83.27	28.68
insertRule	4	4

Table 6.1: Apps containing CSP-relevant APIs

6.2 Constant vs. Non-Constant Parameters

In table 6.2, we can see how exactly the CSP-relevant APIs are used amongst the applications in our data set. It gives an answer to our research questions 4.1 (“How prevalent are the patterns we can successfully rewrite?”) and 4.2 (“How prevalent are the patterns we cannot rewrite?”). With the exception of the “cssText” API and the “setAttribute” API when used to add inline event handlers, usage with non-constant parameters (i.e. where the values of the parameters is not available from the immediate context, see section 5.1.1 for details) clearly dominates when compared to usage with non-constant ones. These relations do not change when excluding library files, as can be seen in table 6.3. We can however tell that the share of API calls coming from library files varies greatly depending on the specific API we investigate. It appears that almost a third of the non-constant “eval” invocations, almost two thirds of the “setTimeout”/“setInterval” and “cssText” invocations and more than half of the “Function” invocations come from either one of the four libraries. To clarify this point, table 6.4 shows the percentage of each API usage coming from a file belonging to one of the libraries. This means that while it is important that app developers choose their APIs wisely, they are often forced to weaken their CSP definitions by consequence of using certain third-party libraries. From the high share of constant “cssText” usage among libraries we can further conclude that library developers could easily refactor parts of their code to make it more conforming to CSP’s requirements. Of course, this must be an “all or nothing” decision; unless all (i.e. constant and non-constant) invocations of “cssText” and “insertRule” are rewritten, developers using a library cannot refrain from adding “style-src unsafe-eval” to their CSP definitions.

In addition to the abovementioned metrics, there are two numbers we have collected that are not included in the referenced tables. First, we have found exactly zero (constant) usages of the “eval” method to parse JSON strings. This is noteworthy because according to Richards et al. [20], this type of “eval” usage was quite common some years ago. Second, there are 36124 invocations of the “setTimeout”/“setInterval” APIs that do not rely on string evaluation and are therefore not relevant for Content Security Policy. They shall be mentioned here for completeness’ sake.

API	Constant parameters	Non-constant parameters
setAttribute (event handler)	32	7
eval	972	2448
setTimeout / setInterval	427	16449
Function	93	1451
setAttribute (style)	424	895
cssText	4209	2355
insertRule	0	87

Table 6.2: Number of CSP-relevant API calls

6.3 Limitations to Rewriting

As could already be suspected from the high numbers of non-constant API invocations in the section above, our approach to rewriting CSP-relevant APIs often fails to have any effect on the eventual CSP definitions for an application. Table 6.5 shows the percentage

API	Constant parameters	Non-constant parameters
setAttribute (event handler)	27	6
eval	971	1692
setTimeout / setInterval	308	6056
Function	78	696
setAttribute (style)	417	756
cssText	905	779
insertRule	0	85

Table 6.3: Number of CSP-relevant API calls, library files excluded

API	% Constant parameters	% Non-constant parameters
setAttribute (event handler)	15.63	14.23
eval	0.1	30.88
setTimeout / setInterval	27.87	63.18
Function	16.13	52.03
setAttribute (style)	1.65	15.53
cssText	78.5	66.92
insertRule	-	2.3

Table 6.4: Percentage of CSP-relevant APIs coming from library files

of applications our tool was forced to add a strictness-weakening keyword to because they included non-constant API calls. As shown, the vast majority of "unsafe-eval" keywords for both the "script-src" and "style-src" directives could not be prevented by our tool. The percentage of applications that actually benefitted from our application is shown in table 6.6. As explained in the previous chapter, only small applications that do not make much use of UI libraries could successfully be rewritten in order to prevent less strict CSP definitions. The results of our empirical analysis confirm this: While our rewriting efforts removed the need for "unsafe-inline" for the "style-src" directive in 8% of apps in our data set, the shares for the other keywords are significantly lower. Also, excluding the third-party library files has not much impact on those ratios. The empirical analysis therefore backs up our preliminary and mostly negative answer to the third research question ("Can we automatically generate sensible CSP definitions for real-world Cordova apps?").

What must be considered when reviewing those metrics is the fact that for some keywords, there are many applications that did not need any rewriting at all. For example, the "script-src: unsafe-inline" policy was necessary for 0.61% of applications, could be avoided for a further 3.27% of them, and never even had to be considered for the remaining 96.12%. When putting our statistics in relation to the number of apps that had an actual need for rewriting, we get metrics for how effective our rewriting approach is. These metrics are shown in table 6.7. There we see that our tool performed quite well for the “script-src: unsafe-inline” policy, where 84.28% of the applications that needed rewriting could benefit. With 25.88%, a much lower but still significant ratio of apps that needed rewriting could benefit with respect to the “style-src: unsafe-inline” policy. From that we can conclude that the hashing approach to the “script-src” directive is very effective and that for most applications, there is no need at all to specify “unsafe-inline” for scripts — a practice that is still very widespread amongst developers, according to Weichselbaum et al. [23] and Willocx et al. [24]. Conversely, our tool was almost completely ineffective with respect to the “unsafe-eval” keywords for both the “script-src” and “style-src” directives.

Keyword	% Apps	% Apps (w/o library files)
script-src: unsafe-inline	0.61	0.49
script-src: unsafe-eval	98.55	79.17
style-src: unsafe-inline	22.91	13.48
style-src: unsafe-eval	81.33	26.96

Table 6.5: Apps requiring extra keywords because they could not be rewritten

Keyword	% Apps	% Apps (w/o library files)
script-src: unsafe-inline	3.27	2.82
script-src: unsafe-eval	0	0.12
style-src: unsafe-inline	8	7.97
style-src: unsafe-eval	2.42	2.7

Table 6.6: Apps successfully rewritten to avoid keywords

Keyword	% Apps	% Apps (w/o library files)
script-src: unsafe-inline	84.28	85.2
script-src: unsafe-eval	0	0.15
style-src: unsafe-inline	25.88	37.16
style-src: unsafe-eval	2.89	9.1

Table 6.7: Apps successfully rewritten in relation to apps that need rewriting

6.4 Framework Usage

We also analyzed the names of all Javascript files loaded into applications in our data set. We found three third-party libraries to be common among the applications we analyzed: The jQuery library, which is the most common one with almost two thirds of apps making use of it. The Angular framework, which is present in nearly a quarter of applications. And the Ionic framework which has similar prevalence as the Angular framework. Table 6.7 shows these results in detail.

For the jQuery library, many files include a version string inside their name, which we also analyzed. We found that even though our data set only includes applications no older than three years, there is a striking tendency towards older versions of the library. These results can be seen in table 6.8. Out of the 489 jQuery files that include a version string, the large majority are files from jQuery version 1, with about half that amount from version 2 and only a twentieth using the latest version 3. Moreover, the latest minor versions used are 1.9 (highest available 1.12), 2.2 (which is the highest available) and 3.3 (highest availble 3.4), with their last updates received in February 2013, May 2016 and January 2018, respectively. This means that over 60% of the jQuery files found in the data set are more than 6 years old with almost another third being older than 3 years. This is especially relevant because jQuery uses the "eval" method internally, which might be subject to change in future versions of the library. But since adoption of new versions is obviously very low, this may not have an impact on CSP definitions for Cordova applications in the near to mid-term future.

Library	Occurrence[%]
jQuery	64.36
Angular	24.12
Ionic	23.27

Table 6.8: Percentage of apps including third-party libraries

Version	Occurrence[%]
1.x.x	64.01
2.x.x	29.86
3.x.x	5.11

Table 6.9: Percentage of files including jQuery versions

6.5 Threats to Validity

The analysis we performed does have some limitations inherent to its approach. First, the parsing of HTML files may have failed in some cases due to non-standard syntax being used. If this happens, our tool just skips the violating file and proceeds, effectively causing any Javascript snippets related to the file to be skipped as well. This can cause our metrics to be lower than they actually are.

Second, we assessed whether files belong to a third-party library like jQuery or Angular based on their names. Specifically, we looked for (case-insensitive) occurrences of the strings “cordova”, “jquery”, “angular” or “ionic”. Since we cannot rule out the fact that some files are part of a third-party library but do not include such strings in their name, there might be some false negatives present in those metrics. Furthermore, developers may have named some of their files to include such a string; e.g. because they contain a part of their codebase that interfaces with one of those libraries. In this case, there could be false positives present. Finally, there might be other large third-party libraries we have not considered in our analysis; though we have not found any indication of this fact. In this case, our conclusions about the impact of third-party libraries on CSP definitions might be incomplete.

7 Conclusion

In this thesis, we have analyzed the security of mobile web applications and how Content Security Policy can enhance it. We also proposed a tool that automatically generates CSP definitions for existing Cordova applications and attempts to rewrite any Javascript APIs that are restricted by CSP in order improve the strictness of our generated policies.

We have found that there is a variety of vectors to achieve code injection into a WebView. The sources for those vectors can be barcodes, Bluetooth or Wifi IDs, expired domains, man-in-the-middle attacks performed on insecure connections, near-field communication, SMS messages, user navigation, Android's content providers, the local file system, insecurely stored local files, Android's intent mechanism, local malware, file metadata, text inputs and third-party libraries. The sinks where data coming from such a source has to flow to in order to allow code injection can come from the WebView API, the DOM API, the jQuery API or other similar libraries.

If code injection is possible, several impacts can follow. We identified access to local files, access to sensitive APIs, arbitrary code execution, (universal) cross-site scripting, spreading of malicious code into other applications, spreading of malicious code into other devices, and phishing as possible consequences. Furthermore, malicious applications can leverage WebViews to achieve vetting circumvention, introspection or manipulation of third-party content, user impersonation and UI confusion attacks.

Content Security Policy can protect from such attacks by preventing code injection completely, preventing additional data from being loaded, preventing the exfiltration of data, preventing the manipulation of user interfaces and upgrading insecure connections. We conclude that CSP can be a reliable mitigation of attacks that may be performed on mobile web applications. However, the protecting capabilities of a CSP definition strongly depend on its strictness, which is often limited in applications that rely on certain Javascript APIs that are restricted by CSP.

The tool we proposed performs rewriting on all Javascript APIs that are restricted by one of the CSP directives. While occurrences of these APIs that include constant parameters can successfully be rewritten by our application, the ones that include dynamic arguments have to be left as they are. This leads to the fact that many applications cannot benefit from our application, since they rely on dynamic usage of said APIs. Besides this limitation of non-constant parameters which comes from the fact that we do not perform any in-depth flow analysis, we identified six more patterns that limit the generation of CSP definitions for an application.

One pattern that is inherent to the specification of CSP itself is due to the fact that there is no fully equivalent alternative to Javascript's "Function" constructor. Two patterns that are inherent to specific use cases include applications that make connections to resources chosen by the user at runtime and apps that dynamically select the API endpoints they connect to. Finally, the limitations that originate from our specific approach to this problem are the dynamic addition of HTML tags to the DOM in an "implicit" manner, the usage of HTML templating engines, the fact that we cannot rewrite remote resources, and the aforementioned API usages with non-constant parameters.

As a side discovery, we found that Chrome displays a contradictory error message in connection with CSP's "unsafe-hashes" keyword. While the W3C working draft for CSP level 3 mentions that inline style attributes can be secured with hashes when using "unsafe-hashes", Chrome does not actually support this behavior. Because this section of the working draft is not considered to be normative, this in itself is expected behavior. However, when inline style attributes are present in the DOM, Chrome outputs an error message that suggests exactly this solution of adding a hash for the CSP-violating style attributes. This contradiction between Chrome's behavior and the error message it produces could be confirmed for Chrome versions 69 and 74.

To assess the effectivity of our tool on a large scale, we performed an empirical analysis on a set of 825 Cordova applications. We found that the "eval", "setTimeout"/"setInterval", "Function" and "cssText" APIs are present in more than half of the Cordova applications, when we include library files in our analysis. However, if we exclude such library files, we find that only the "setTimeout"/"setInterval" is present in more than 50% of the apps. From this we conclude that many of the CSP-relevant

APIs are used not in the actual application code, but in the code of libraries used by these applications. Because of that, a great part of the efforts towards stricter policies would have to be made not by application developers, but by the maintainers of such libraries.

When analyzing the relation between API calls with constant parameters versus calls with non-constant parameters, we found that except for the “cssText” API, the vast majority of APIs are used with non-constant arguments. Furthermore, we found that for the four most important CSP directives (“unsafe-inline” and “unsafe-eval” for the “script-src” and “style-src” directives respectively), only a small percentage of the analyzed applications actually benefitted from our rewriting. Except for the “script-src: unsafe-inline” rule, the number of apps that could not benefit from our approach greatly outnumbers the amount of apps that could.

This confirms our conclusion on the feasibility of our approach to rewriting applications: Without applying in-depth flow analysis and the ability to deal with the special syntaxes introduced by UI frameworks, the static generation of CSP definitions can only improve the security of a small portion of applications. Presumably those that are small in size and do not make use of extensive UI frameworks. Our solution therefore yields insufficient results for most applications and any approach similar to ours should address the limiting patterns we identified in our evaluation.

A
Anleitung zu wissenschaftlichen Arbeiten

WebView

A WebView is an encapsulation mechanism which allows applying web technologies such as HTML, CSS and Javascript to build installable, possibly offline applications for mobile devices. It is based on a browser engine which renders the given code and displays it to the user. Furthermore, it provides a bridge mechanism that allows the Javascript code to interact with native code, which enables access to device resources critical for many apps, such as the camera, bluetooth functionalities or data from content providers like contacts or call logs.

There are several implementations of WebView available, including iOS's "UIWebView", Windows Phone OS's "Windows Browser" and Android's "WebView". The Android WebView, which is the only implementation in the scope of this thesis, is based on the WebKit browser engine for Android versions below 4.4. Starting from Android 4.4, Blink is used as WebView underlying browser engine.

Developers can use WebViews either to display some parts of their applications where this might be preferable (e.g. a "terms and conditions" page which needs to be updated regularly and easily). Or they can use it to build full-fledged hybrid applications where native code only acts as a wrapper and the app's business logic resides entirely inside a number of WebViews.

WebViews have a number of advantages. Most importantly, they significantly enhance portability, since web technologies are platform-agnostic. Hybrid app frameworks like Cordova streamline this by providing appropriate abstractions so developers can easily build apps that support different platforms without having to tailor their software to the specific quirks of their target operating systems. Further advantages include easier distribution of updates and more widespread familiarity with the given technologies.

As expected, there are certain downsides to using WebViews. Performance is likely to suffer due to the overhead of the browser engine, which leads to reduced user experience. Also, user interfaces in WebViews tend to violate at least some of the user expectations concerning the look and feel of the applications, since they are used to their platform's design principles which are not enforced within WebViews. Finally, using web technologies comes with a whole class of security problems that native apps would not be affected by, specifically those related to code injection vulnerabilities.

A.1 The WebView API

The WebView API consists of the main "WebView" class as well as multiple classes that are used to configure a given WebView. With respect to security, the most important classes for configuration are the "WebSettings" class and the "WebViewClient" class. In the following section, the security sensitive APIs of the those classes are discussed. All informations related to the WebView API come from the Android Developers documentation [1].

A.1.1 WebView Class

The main "WebView" class is responsible for instantiating a WebView, managing Javascript bridges and loading different types of data in different ways. It also provides a getter for its "WebSettings" object, which is described further below.

addJavascriptInterface

With the "addJavascriptInterface" method, a Java object can be injected into the given WebView. Apart from this Java object, a name is passed to the method, which can then be used inside the Javascript context of the main frame to access the injected object. For Android versions 4.2 and above, only public methods that are annotated with "@JavascriptInterface" can be accessed in this way. Prior versions allow the Javascript code to access all public methods via reflection, including inherited methods. The interactions between Javascript code and injected Java objects happen in a separate, private background thread, which is why thread safety must be maintained at all times.

removeJavascriptInterface

With the “removeJavascriptInterface” method, previously injected Java objects can be removed from the WebView. This method was introduced in Android 3.0 to allow for safer handling of Javascript bridges.

evaluateJavascript

The “evaluateJavascript” method provides another way of communicating between native Java code and the WebView’s Javascript code. It accepts a string containing Javascript code which will be executed in the context of the current page. As a second argument, it will take a callback function which is invoked upon completion of the Javascript code, if this returns a non-null value. The return value of the evaluated code will be passed to the callback function as an argument. According to Song et al. [22], it was introduced in Android 4.4 to allow Javascript to be executed in the main UI thread, which is not possible with the other APIs.

Compared to the “addJavascriptInterface” API, this method of communication between native and web code is more secure, since data is only passed to the provided, trusted Java callback method. Song et al. [22] did not find any vulnerabilities introduced by using this form of communication. It does, however, impose some limitations, since it offers a less direct way of communication compared to the Javascript bridge API. Furthermore, starting with Android 7.0, the state of an empty WebView’s Javascript code is not persisted across navigations such as the “loadUrl” method. The Android Developers documentation [1] recommends using the “addJavascriptInterface” API for use cases where persistence is needed across WebView navigations.

loadData

The “loadData” method loads a given string via a “data:” scheme URL. It accepts a string containing the data to be loaded, one denoting its MIME type and another one denoting its encoding (base64 or URL encoding). Due to the same-origin policy, any Javascript loaded with this method is not able to access content that was loaded using any URL scheme that is not a “data” scheme. This includes “http” and “https” schemes. If such data has to be accessed, the “loadDataWithBaseURL” method can be used. Content that was loaded with the “loadData” method has an origin of “null”, which should not be considered a trusted origin, since malicious code is also able to create content that has an origin of “null”.

loadDataWithBaseURL

The “loadDataWithBaseURL” method is used to load data into the WebView which is then given a specified base URL. This base URL is used for the same-origin policy and for resolving relative URLs. In addition to the parameters the “loadData” method accepts, this method also accepts the base URL and a history URL. The latter one is used as a history entry for the loaded content. If the base URL parameter denotes “http”, “https”, “ftp”, “ftps”, “about” or “javascript” as the scheme to be used, the loaded content is not able to access local files via “file” scheme URLs. If the base URL is not a valid HTTP(S) URL, then the window’s origin is set to “null”. In this case, the same caveats as for the “loadData” method apply.

loadUrl

The “loadUrl” method loads a given URL into the WebView. Optionally, the method accepts a map of HTTP headers that will be added to the request if provided. As mentioned for the “evaluateJavascript” method, calling “loadUrl” on an empty WebView destroys the state of any Javascript code previously established in that WebView.

postUrl

Using the “postUrl” method, a given URL can be loaded via the HTTP “POST” method. The data to be sent in this POST request is passed as a second parameter, alongside the URL that should be loaded.

A.1.2 WebSettings Class

The “WebSettings” class holds the state of certain settings for a given WebView. It is derived by calling the “getSettings” method of the “WebView” class.

setJavaScriptEnabled

This method allows the associated WebView to execute Javascript code. It defaults to a value of “false”.

setMixedContentMode

The “setMixedContentMode” method configures whether or not the associated WebView allows a secure origin to load contents from unsecure origins. The Android Developers documentation [1] recommends setting this to never allowing mixed content, which is the default value for Android 5.0 and above. Configuring this to always allow mixed content is “strongly discouraged”, but also used as the default value for Android 4.4 and below. There is also a third option called the “compatiblity mode”, which allows certain insecure contents to be loaded, depending on the specific release.

setAllowFileAccess

This method configures whether or not the WebView can access the file system, not including assets and resources of the containing app, which can always be accessed. It defaults to “true”.

setAllowFileAccessFromFileURLs

The “setAllowFileAccessFromFileURLs” method specifies, whether or not Javascript code that was loaded via a file scheme URL is allowed access to content that was loaded via a different file scheme URL. It defaults to “true” for Android 4.0.3 and below, while Android 4.1 and above has a default of “false”. The Android Developers documentation [1] recommends setting this to “false” in order to enable the most secure configuration. Developers are also advised to explicitly configure this to “false” for Android 4.0.3 and below, since it could lead to violation of the same-origin policy. It is noteworthy that this method does not impact resource access for non-Javascript vectors. For example, HTML elements that load resources (such as the <img> tag) are not affected by the restrictions imposed by this setting.

setAllowUniversalAccessFromFileURLs

Similarly to “setAllowFileAccessFromFileURLs”, this method specifies whether or not Javascript code that was loaded via a file scheme URL is allowed access to content that was loaded from any origin. The same defaults, caveats and recommendations as mentioned for “setAllowFileAccessFromFileURLs” apply.

A.1.3 WebViewClient Class

The “WebViewClient” class is used to configure how the associated WebView handles certain browsing events. It is created via its constructor and then passed to a WebView using the “setWebViewClient” method. This class is relevant for an app’s security mostly because it can be used to override WebView’s standard behavior when navigation events occur inside the WebView. The default behavior is to create an intent that can be handled by the default browser, which is the recommended way of handling navigation events.

shouldInterceptRequest

This is a callback method that is invoked when a resource request happens and that request is not a “javascript:” or “blob:” scheme URL and no “file:” scheme URL accessing the app's assets or resources. By default, it informs the containing application of the request and allows it to load the contents. This method can be overriden in order to configure the associated WebView's behavior when a resource request is triggered. The Android Developers documentation [1] urges developers to use care when accessing private data or the view system, since this method is not called on the UI thread. Starting with Android 4.4, WebViews are based on Blink, which introduces the additional restriction that the "shouldInterceptRequest" method is only invoked for valid URLs, but not custom URL schemes.

shouldOverrideUrlLoading

This is a callback method that is invoked when a URL is to be loaded into the associated WebView. If it returns "false", the given URL is loaded normally. If it returns "true", loading is aborted. By default, the WebView responsible for the URL load asks the Activity Manager to select a handler for the given URL. This method is called for subframes and non-HTTP(S) schemes. It is not invoked for "POST" requests. As with "shouldInterceptRequest", the "shouldOverrideUrlLoading" method is only invoked for valid URLs for Android 4.4 and above.

onReceivedSslError

This is a callback method that is invoked when an SSL error occurs. The method is expected to either cancel or proceed with the loading. The Android Developers documentation [1] recommends to always cancel the loading and only override this method for logging or displaying custom error pages. Asking the user how to handle a SSL error is discouraged, since they cannot be expected to make an informed decision on this matter. This method is only invoked on recoverable errors. By default, any loading process that triggers an SSL error is canceled.

A.2 Risks Introduced by WebView

With the current implementation of WebView in Android come some security risks that may violate user privacy and device integrity. The main risks that WebViews are affected by are explained in the following section. It provides a basic overview of the root causes that can lead to a multitude of attacks on WebViews. Such attacks are discussed in detail in chapter 3 of this thesis.

A.2.1 Code Injection

Since they are based on a browser engine, WebViews share a major security concern with traditional web apps: Mixing application data with executable code. When a WebView is rendered, the browser engine parses the whole HTML content. If it contains valid code (e.g. in a <script> tag), this code is executed. This allows the injection of unexpected Javascript code into the application, if input channels are not sanitized correctly.

In traditional web apps, the primary input channel for such attacks is the website itself. An application that allows its users to input some data and then displays it to all the users of the app without properly sanitizing it beforehand, is vulnerable to code injection. An attacker can send valid Javascript code to the application, which then stores this code on the server. If another user visits the app and gets displayed the malicious data, the victim's browser correctly identifies this as code and executes it in the context of the page.

In mobile web applications, the number of possible input channels is much greater, as Jin et al. [14] clearly demonstrate. Besides the common injection methods that also apply to traditional web apps (such as intercepting HTTP connections or leveraging existing UXSS vulnerabilities in the browser engine), mobile web applications can suffer from code injection via device specific resources like contact lists, barcode scanners, file systems and more. The attack surface with respect to code injection is therefore broader in mobile web apps as compared to traditional web apps.

Furthermore, the impact of such code injection tends to be more serious in mobile web apps than in traditional web apps. Through WebView's Javascript bridge, the otherwise sandboxed Javascript code in a web page may be given access to sensitive device resources like its location, file system, camera and others.

A.2.2 Lack of Fine-Grained SOP

One of the root causes for many attacks against WebViews is the lack of a same-origin policy that is appropriate for common applications of WebView. As Davidson et al. [9] mention, WebView's same-origin policy falls short in protecting both apps and web contents.

For apps, all granted permissions are automatically inherited by any embedded web content, including sub-frames or content loaded from untrusted origins. Furthermore, Java objects injected via the Javascript bridge mechanism are accessible by web content of any origin. Since the Javascript bridge provides often needed functionalities, developers are forced to accept reduced security to achieve these functionalities.

For web content, there is no same-origin policy being enforced for interactions from an application to its embedded web content. It is assumed that applications always own the web content that they are embedding. Therefore, malicious applications are not prevented from spying on sensitive web content by same-origin policy.

As a consequence of there not being an appropriate same-origin policy, sensitive data and functionality is often left exposed to non-trusted origins. In conjunction with the abovementioned code injection risks, this can lead to attacks that jeopardize user privacy as well as device integrity.

A.2.3 Bugs Introduced by WebViews

The additional layer of abstraction that WebView provides introduces bugs to an application. As Hu et al. [13] concluded, the causes for such bugs include misalignment of the WebView's lifecycle with that of the enclosing activity or fragment, WebView's quickly evolving nature, wildly varying device configuration, application misconfiguration, misuse of WebView APIs, the error-prone Javascript bridge mechanism, and current limitations in functionality of WebViews compared to full web browsers. While Hu et al. did not analyze the security implications of these classes of bugs, it is likely that some newly-introduced bugs can actually aid attackers in compromising a mobile web application.

A.3 Best Practices and Mitigations

To avoid the most pressing problems arising from WebView, there are some best practices and mitigation techniques that can be applied. While they cannot solve the core problems that come with using a browser engine to render application content, applying those mitigations can significantly reduce the attack surface of a given app.

A.3.1 Use Secure Channels

To prevent code injection via a man-in-the-middle attack, any contents to be displayed in the WebView should only be loaded over secure channels. This means avoiding HTTP and misconfigured HTTPS.

A.3.2 Do not Display Untrusted Contents

A WebView should not be used to display untrusted content. As the Android Developers documentation [1] mentions, WebViews are intended for displaying first-party content that is owned by the app's developers. If third-party content should be displayed, the documentation recommends creating an intent to invoke the default web browser.

A.3.3 Sanitize Untrusted Inputs

If content enters the application from an untrusted source (e.g. the barcode scanner), it should always be sanitized before being displayed inside the WebView. This is especially important for data that is transmitted via the Javascript bridge, since there is no built-in sanitization in place for such transmissions, according to Bai et al. [5].

A.3.4 Use Safe APIs to Display Untrusted Contents

If content from untrusted third-parties must be displayed within the WebView, it should only be done using safe APIs that do not execute any Javascript code that might be present inside the data. Examples of unsafe APIs include some of the DOM APIs (such as the “document.write” method or the “innerHTML” attribute), the jQuery APIs (such as the “append” method) as well as other APIs that add data to a WebView’s DOM while executing any code contained within.

A.3.5 Prevent Navigation

Since navigating inside a WebView opens up the possibility of loading content that was not intended by developers to be displayed inside the WebView, navigation should generally be disallowed. Clicking links should invoke the default web browser to view such possibly untrusted content.

A.3.6 Prevent New Windows and Popups

As mentioned in the Android Developers documentation [1], WebViews should be prevented from opening new windows and popups. This is to be done by passing “true” to the “setSupportMultipleWindows” method and not overriding the “onCreateWindow” method.

A.3.7 Handle SSL Errors Safely

The “onReceivedSslError” method of the “WebViewClient” class allows developers to change a WebView’s behavior when SSL errors are encountered. This should never be used to ignore SSL errors automatically, since this would effectively allow man-in-the-middle attacks to be performed, which can easily lead to code injection.

A.3.8 Use JS Bridges with Care

The Javascript bridge functionality provided by WebView allows injected Java objects to be accessed by Javascript code from any origin. Furthermore, Javascript bridges often expose sensitive data which should be appropriately protected. Because of those circumstances, Javascript bridges should be used with due care. They should provide only the bare necessities that cannot be avoided for the app to function properly. If possible, they should be completely avoided for Android versions 4.2 or below, since reflection can be used to perform attacks against the WebView. If Javascript bridges are truly necessary, for such versions the "removeJavascriptInterface" method should be used in order to destroy any previously injected Java objects after they were used in the intended way.

A.3.9 Use Content Security Policy

Starting with Android 4.4, WebView supports defining a Content Security Policy, a security mechanism supported by most modern browsers that mitigates code injection attacks. A closer description of Content Security Policy, its benefits and how to retrofit applications to support strict policies can be found in the main part of this thesis.

A.3.10 Update Regularly

As with all applications, it is critical to keep WebView versions up-to-date. According to Yang et al. [27], WebView was linked to the Android OS before Android 5.0, which meant that WebView could not be updated seperately. Starting with Android 5.0, WebView is its own APK and can be updated independently from the operating system, allowing for a more flexible update strategy.

Updates to WebView have in the past included changes that are relevant for application security. With Android 4.2, the @JavascriptInterface annotation was introduced, which fixed a security flaw that enabled Javascript code to access any public method of an injected Java object. With Android 4.4, additional restrictions for custom URL schemes were introduced, which disallowed invoking the "shouldInterceptRequest" and "shouldOverrideUrlLoading" methods for invalid URLs. Android 4.4 changed WebView's underlying browser engine from WebKit to Blink, which removed universal cross-site scripting (UXSS) vulnerabilities that were present in WebKit. According to Song et al. [22], Android 5.0 fixed further UXSS vulnerabilities previously present in WebView. Android 6.0 removed availability of the getClass method for Java objects injected via the Javascript bridge. Furthermore, in Android 6.0, the geolocation API was restricted to only be available on secure origins, such as HTTPS.

Changes such as those mentioned above have direct consequences for an application's security, since known weaknesses are addressed and risks can be mitigated. A sensible update strategy is therefore needed to avoid known security flaws in WebView.

A.3.11 Use Additional Security Measures

Apart from correctly using the WebView APIs and regularly updating both the operating system and the WebView package, developers can rely on additional security measures to improve the security of their applications.

When a hybrid app framework is used, supplementary security mechanisms may be available that can mitigage some of WebView's shortcomings. For example, the Cordova framework allows developers to use several whitelisting mechanisms in order to limit network requests, navigation and intent creation. Willocx et al. [24] have carried out in-depth research on Cordova's security mechanisms and their usages.

Futhermore, external tools can be used to either detect present vulnerabilities or provide additional security at runtime. Several previous studies have proposed tools based on static or dynamic analysis, that automatically detect open injection vectors or other vulnerabilities caused by WebView. Refer to chapter 2 for studies proposing such tools. Other researchers like Davidson et al. [9] have implemented services that run alongside any mobile web application and provide secure WebViews that allow extending the access policies provided by WebView. Developers can define so-called "dynamic access policies" that enable fine-grained restrictions on resource usage for both native applications and web content, thus protecting benign apps from malicious web content and benign web content from malicious apps. They also implemented a rewriting tool which can make any mobile web application compatible with their service.

B
CSP Directives

B.1 connect-src

The “connect-src” directive limits the resources that can be connected to using certain APIs. Obviously, any API that abstracts away from these interfaces is also restricted by “connect-src”. An example for this would be the “jQuery.ajax” method, which internally uses the “XMLHttpRequest” object. The following APIs are restricted by the “connect-src” directive:

- HTML’s <a> tags using the “ping” attribute:
  <a ping=“https://example.ch”>
- Web API’s “WindowOrWorkerGlobalScope.fetch” method:
  fetch(new Request(“https://example.ch”))
- Web API’s “XMLHttpRequest” object:
  new XMLHttpRequest().open(“GET”, “https://example.ch”)
- Web API’s “WebSocket” object:
  new WebSocket(“https://example.ch”)
- Web API’s “EventSource” object:
  new EventSource(“https://example.ch”)
- Web API’s “navigator.sendBeacon” method:
  navigator.sendBeacon(“https://example.ch”, <data>)

B.2 default-src

The “default-src” directive is a fallback for all so-called “fetch directives”. If any of those directives is not defined, the resources specified in “default-src” will be used to restrict that directive instead. If a directive is present in the CSP definition, “default-src” is completely overriden and will not be taken into consideration for this directive. This means that for the CSP definition “default-src ‘self’ https://example.ch; img-src ‘self’;”, images are only allowed to load from local resources, even though “default-src” specifies “example.ch” as a valid resource as well. The directives for which “default-src” will serve as fallback are:

- child-src
- connect-src
- font-src
- frame-src
- img-src
- manifest-src
- media-src
- object-src
- prefetch-src
- script-src
- script-src-elem
- script-src-attr
- style-src
- style-src-elem
- style-src-attr
- worker-src

B.3 font-src

The “font-src” directive limits the resources which can be loaded via CSS’s “@font-face” rule:

- @font-face src: url(“https://example.ch”);

B.4 frame-src

The “frame-src” directive limits the resources which can be loaded in sub-frames:

- HTML’s <frame> tag (deprecated):
  <frame src=“https://example.ch”>
- HTML’s <iframe> tag:
  <iframe src=“https://example.ch”>

B.5 img-src

The “img-src” directive limits the loading of images and icons. The following APIs are restricted:

- HTML’s <img> tag:
  <img src=“https://example.ch”>
- HTML’s <link> tag with the “rel” attribute set to “icon”:
  <link rel=“icon” href=“https://example.ch”>
- CSS rules when they load an image and refer to an external resource:
  background-image: url(“https://example.ch”)

B.6 manifest-src

The “manifest-src” directive limits the resources which can referenced in HTML’s <link> tag with the “rel” attribute set to “manifest”:

- <link rel=“manifest” href=“https://example.ch”>

B.7 media-src

The “media-src” directive limits the resources from which HTML’s media elements can load data:

- HTML’s <audio> tag:
  <audio src=“https://example.ch”>
- HTML’s <video> tag:
  <video src=“https://example.ch”>
- HTML’s <track> tag:
  <track src=“https://example.ch”>

B.8 object-src

The “object-src” directive limits the resources from which certain HTML tags associated with browser plugins are allowed to load. The same HTML tags are governed additionally by the “plugin-types” directive (see below).

- HTML’s <object> tag:
  <object data=“https://example.ch”>
- HTML’s <embed> tag:
  <embed src=“https://example.ch”>
- HTML’s <applet> tag:
  <applet archive=“https://example.ch”>

B.9 script-src

The “script-src” directive is arguably the most important CSP directive. It applies the following restrictions:

- It restricts the allowed sources for external scripts:
  <script src=“https://example.ch”>
- It restricts the execution of the following inline script variants
  - Inline script tags:
    <script>alert(“inline script”)</script>
  - Inline event handlers:
    <div onclick=“alert(‘inline event handler’)”>
- It restricts the execution of the following string evaluation variants
  - Javascript’s “eval” method:
    eval(‘alert(“string evaluation”)’)
  - Javascript’s “Function” object:
    new Function(“factor1”, “factor2”, “return factor1*factor2”)
  - The “WindowOrWorkerGlobalScope.setTimeout” method if called with a string argument:
    window.setTimeout(‘alert(“setTimeout”)’, 1000)
  - The “WindowOrWorkerGlobalScope.setInterval” method if called with a string argument:
    window.setInterval(‘alert(“setInterval”)’, 1000)

  - Some other, non-standard methods of “WindowOrWorkerGlobalScope”:
    WindowOrWorkerGlobalScope.setImmediate()
    WindowOrWorkerGlobalScope.execScript()

B.10 style-src

Just as the “script-src” directive does for Javascript, the “style-src” directive enforces several restrictions for CSS rules:

- It restricts the allowed sources for external stylesheets:
  <link rel=“stylesheet” href=“https://example.ch”>
- It restricts the allowed sources for CSS’s “@import” rule:
  @import url(“https://example.ch”)
- It restricts the application of the following inline style variants:
  - The application of inline styles:
    <style> #someid { height: 100px; } </style>
  - The application of inline style attributes:
    <div style“height: 100px”>
  - The application of dynamically set inline style attributes:
    someElement.setAttribute(“style”, “height: 100px;”)
- It restricts the execution of the following string evaluation variants:
  - The execution of the “CSSStyleSheet.insertRule” method:
    someStylesheet.insertRule(“#someid { height: 100px; }”)
  - The execution of the “CSSGroupingRule.insertRule” method:
    someGroupingRule.insertRule(“#someid { height: 100px; }”)
  - The execution of assignments to the “CSSStyleDeclaration.cssText” property:
    someElement.style.cssText = “height: 100px;”
- It restricts CSS rules loaded via the “Link” HTTP header (not relevant for mobile web applications)

B.11 worker-src

The “worker-src” directive limits the resources workers can be loaded from. The following Javascript APIs are restricted:

- The Web API's "Worker" object:
  new Worker("https://example.ch")
- The Web API's "SharedWorker" object:
  new SharedWorker("https://example.ch")
- The Web API's "navigator.serviceWorker.register" method:
  navigator.serviceWorker.register("https://example.ch")

B.12 base-uri

The "base-uri" directive limits the resources which can be set as the base href for the given page:

- <base href="https://example.ch">

B.13 plugin-types

The "plugin-types" directive defines the plugins that are allowed to handle data loaded with the HTML tags mentioned under "object-src". The following conditions must be met for the given data to be loaded:

- The HTML tag must declare the MIME type of the data:
  <object type="application/x-shockwave-flash" data="https://example.ch">
- The data that is loaded by the tag has to actually match the declared MIME type
- The declared MIME type must be whitelisted in the "plugin-types" directive

B.14 form-action

The "form-action" directive limits the resources an HTML <form> tag can be submitted to, which is specified in the "action" attribute:

- <form action="https://example.ch">

B.15 upgrade-insecure-requests

One of the more unusual CSP directives is the "upgrade-insecure-requests" directive. Instead of limiting valid resources to connect to, it causes any HTTP request made from the application to be changed to an HTTPS request. For example, the tag <img src=“http://example.ch”> is automatically upgraded by the browser to <img src=“https://example.ch”>. This behavior does not apply to <a> tags, which will navigate to their specified URL, even with this directive set.

Bibliography

[1] Android Developers: API Reference. `https://developer.android.com/reference`. Accessed: 2019-12-13.

[2] Mapbox GL JS: CSP Directives. `https://docs.mapbox.com/mapbox-gl-js/overview/#csp-directives`. Accessed: 2019-12-13.

[3] MDN Web Docs. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy`. Accessed: 2019-12-13.

[4] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[5] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Transactions on Information Forensics and Security*, 14(3):677–692, March 2019.

[6] A. B. Bhavani. Cross-site Scripting Attacks on Android WebView. *International Journal of Computer Science and Network*, 2(2), 2013.

[7] Y. Chen, A. B. Jeng, H. Lee, and T. Wei. DroidCIA: A Novel Detection Method of Code Injection Attacks on HTML5-Based Mobile Apps. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1014–1021, Aug 2015.

[8] S. Y. Choi and H. Y. Lee. Toward Automated Scanning for Code Injection Vulnerabilities in HTML5-Based Mobile Apps. In *2016 International Conference on Software Security and Assurance (ICSSA)*, pages 24–24, Aug 2016.

[9] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha. Secure Integration of Web Content and Applications on Commodity Mobile Operating Systems. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications*

Security, ASIA CCS '17, page 652–665, New York, NY, USA, 2017. Association for Computing Machinery.

[10] M. Ghafari, P. Gadient, and O. Nierstrasz. Security Smells in Android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130, Sep. 2017.

[11] B. Hassanshahi, Y. Jia, R. H.C. Yap, P. Saxena, and Z. Liang. Web-to-Application Injection Attacks on Android: Characterization and Detection. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 577–598, Cham, 2015. Springer International Publishing.

[12] S. F. Hidhaya and A. Geetha. Detection of Vulnerabilities Caused by WebView Exploitation in Smartphone. In *2017 Ninth International Conference on Advanced Computing (ICoAC)*, pages 357–364, Dec 2017.

[13] J. Hu, L. Wei, Y. Liu, S. Cheung, and H. Huang. A Tale of Two Cities: How WebView Induces Bugs to Android Applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 702–713, New York, NY, USA, 2018. Association for Computing Machinery.

[14] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code Injection Attacks on HTML5-Based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 66–77, New York, NY, USA, 2014. Association for Computing Machinery.

[15] K. Kotowicz. Trusted Types help prevent Cross-Site Scripting. `https://developers.google.com/web/updates/2019/02/trusted-types`. Accessed: 2019-12-13.

[16] P. T. Lau. Scan Code Injection Flaws in HTML5-Based Mobile Applications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 81–88, April 2018.

[17] P. Mutchler, A Doupé, J. Mitchell, C. Kruegel, and G. Vigna. A Large-Scale Study of Mobile Web App Security. In *MoST 2015*, 2015.

[18] K. Peguero, N. Zhang, and X. Cheng. An Empirical Study of the Framework Impact on the Security of JavaScript Web Applications. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 753–758, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[19] S. Pouryousef, M. Rezaiee, and A. Chizari. Let me Join Two Worlds! Analyzing the Integration of Web and Native Technologies in Hybrid Mobile Apps. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1814–1819, Aug 2018.

[20] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] C. Rizzo, L. Cavallaro, and J. Kinder. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 25–46, Cham, 2018. Springer International Publishing.

[22] W. Song, Q. Huang, and J. Huang. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[23] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1376–1387, New York, NY, USA, 2016. Association for Computing Machinery.

[24] M. Willocx, J. Vossaert, and V. Naessens. Security Analysis of Cordova Applications in Google Play. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] World Wide Web Consortium. Content Security Policy Level 3 (Working Draft). `https://www.w3.org/TR/2018/WD-CSP3-20181015/`. Accessed: 2019-12-13.

[26] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and Scalably Vetting JavaScript Bridge in Android Hybrid Apps. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 143–166, Cham, 2017. Springer International Publishing.

[27] L. Yang, X. Cui, C. Wang, S. Guo, and X. Xu. Risk Analysis of Exposed Methods to JavaScript in Hybrid Apps. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 458–464, Aug 2016.