# Investigating mis-implementation of SSL Libraries in Android Applications

A thesis submitted to the
Graduate School of Natural and Applied Sciences

by

Halil Avcı

in partial fulfillment for the
degree of Master of Science

in

Cybersecurity Engineering

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Cybersecurity Engineering.

**APPROVED BY:**

| | |
|---|---|
| Necati Ersen Şişeci<br>(Thesis Advisor) | ..... |
| Prof. Dr. Erkan Türe | ..... |
| Asst. Prof. Davut İncebacak | ..... |

This is to confirm that this thesis complies with all the standards set by the Graduate School of Natural and Applied Sciences of İstanbul Şehir University:

**DATE OF APPROVAL:** 10 March 2016

**SEAL/SIGNATURE:**

# Declaration of Authorship

I, Halil Avci, declare that this thesis titled, 'Investigating mis-implementation of SSL Libraries in Android Applications' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

*"Companies spend millions of dollars on firewalls, encryption and secure access devices, and it's money wasted, because none of these measures address the weakest link in the security chain."*

Kevin Mitnick

Investigating mis-implementation of SSL Libraries in Android Applications

Halil Avci

# Abstract

This thesis presents our analysis of applications that are popular at the market against SSL miss-implementation. 8.882 applications analyzed and as a result 2.354 applications have at least one miss use of SSL libraries which are Custom TrustManager, Custom HostnameVerifiers and WebViewClient libraries. After analysis phase we have created a proof of concept application as an Xposed framework plugin to identify vulnerabilities. Our conclusion is that 27 percent of applications have a vulnerability from SSL connection stand point. The main reasons for these vulnerabilities are developer errors and third party generators or libraries. Using third party libraries can cause security bugs which leads to informations leakage or exploitation.

**Keywords:** Android, SSL, Vulnerability, Custom TrustManager, Custom HostnameVerifiers, WebViewClient

# Android Uygulamalarında SSL Zafiyetlerinin Araştırılması

Halil Avcı

## Öz

Çalışmamız, Android markette yer alan popüler uygulamaların SSL zafiyetlerinin araştırılması ve bunların sunulmasını içermektedir. 8.882 uygulama analiz edilmiş ve sonuç olarak 2.354 uygulamanın en az bir SSL kütüphanesini yanlış kullandığını ortaya koyduk. Analiz aşamasını takiben uygulama üzerinde zafiyetleri ortaya çıkarmak için Xposed altyapısını kullanarak örnek bir uygulama gerçekleştirdik. SSL uygulama zafiyeti açısından incelediğimiz uygulamaların yüzde 27'sinin sorun içerdiği sonucuna vardık. Bu zafiyetlerin ana sebepleri geliştirici hataları, üçüncü parti kütüphane ve uygulamaların kullanılması olarak tespit ettik. Sorunlu üçüncü parti kütüphanelerin kullanılması mahremiyet ihlali ve zafiyetlerin istismar edilmesi gibi hatalara yol açtığını ortaya koyduk.

**Anahtar Sözcükler:** Android, SSL, Zafiyet

*I dedicate this study to my beloved family, wife Aynur and children Ayşenur and İbrahim.*

# Acknowledgments

I would like to acknowledge my deepest appreciation to my advisor Necati Ersen Şiseci for his continuous support and guidance during my study process. I would like to thank rovo89 the creator of Xposed; Anthony Desnos creator of Androguard; Chih-Wei Huang and Yi Sun creator of Android-x86 and other contributors to these projects for their great work. I like to express my sincere gratitude to my family devoted their precious time for me to accomplish the work.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **ADB** | **A**ndroid **D**ebug **B**ridge |
| **APK** | **A**ndroid **A**pplication **P**ackage |
| **CA** | **C**ertificate **A**uthority |
| **CERT** | **C**omputer **E**mergency **R**esponse **T**eams |
| **CERT-CC** | **CERT C**oordination **C**enter |
| **CMU** | **C**arnegie **M**ellon **U**niversity |
| **DEX** | **D**alvik **E**xecutable **F**ile |
| **DHCP** | **D**ynamic **H**ost **C**onfiguration **P**rotocol |
| **DNS** | **D**omain **N**ame **S**ystem |
| **EFF** | **E**lectronic **F**rontier **F**oundation |
| **HSM** | **H**ardware **S**ecurity **M**odule |
| **HTTPS** | **H**yper **T**ext **T**ransfer **P**rotocol **S**ecure |
| **JSSE** | **J**ava **S**ecure **S**ocket **E**xtension |
| **MITM** | **M**an **I**n **T**he **M**iddle |
| **NAT** | **N**etwork **A**ddress **T**ranslation |
| **SPKI** | **S**imple **P**ublic **K**ey **I**nfrastructure |
| **SSL** | **S**ecure **S**ockets **L**ayer |
| **TACK** | **T**rust **A**ssertions for **C**ertificate **K**eys |
| **TLS** | **T**ransport **L**ayer **S**ecurity |
| **VPN** | **V**irtual **P**rivate **N**etwork |

# Chapter 1

# Introduction

With the rise of smart phones people tend to use applications to ease their daily routine of their life's like reading news, gathering their emails, messaging through their social accounts, taking photos and sending them to their backup cloud, paying bills, transaction money using their bank apps and the list goes on. When we consider this small list many personal data can eavesdropped by a second person if connection is not secure, or the implementation of the secure channel is miss-implemented.

People have gain the awareness of Secure Sockets Layer (SSL) connection information when they browse the internet via browsers. Maybe they don't know the underlying technology but they aware of a green lock icon indicating that the site they are connecting is using SSL and they can "securely" connect and perform their operations. But in applications all the connection stuff is done by applications or operating system services out of sight of the user. Everything done under the hood. An average user couldn't understand if there is a connection and if there is any, couldn't understand that connection is secure or unsecure. They thrust the application provider; but sometimes application developers lack of understanding about security concepts. They generally tend to implement the business logic in an easy and time efficient manner.

Android has extension points for SSL certificate validation. This extension points can be used to harden the security of the application like certificate pinning or can be used to bypass security exceptions which is caused by forged certificates, expired certificates, the ones created by adversaries etc. Also a common issue, codes which is deployed to ease development process can be forgotten by developers at the production application. This

codes generally used to test with internal development server with bypassed certificate checks.

In this thesis, 8882 applications at the market analyzed for their SSL implementations. A risk table created according to different factors and a proof of concept application created to identify SSL miss-use at the operation system level. We are investigating secure communication. While expressing SSL we also use the word as a synonym of TLS.

There are some researches like "Why eve and mallory love android: An analysis of android ssl (in) security"[1] and "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps"[2] on the subject; we plan to focus on applications from a Turkish market user and target to extent researches one step further.

Our work is focused on SSL implementation errors which are caused by application developers. SSL protocol implementation errors like heartbleed [3] bug or vulnerabilities recently disclosed on OpenSSL [4] are different and these are out of scope of our work. Most of the vulnerable code just ignores SSL checks and returns always true. An example code from a financial application is follows:

```
class MySSLSocketFactory$1 implements javax.net.ssl.X509TrustManager {
    final synthetic com.z.zmobil.serviceclient.MySSLSocketFactory this$0;

     public void checkClientTrusted(X509Certificate[] p1, String p2)
     {
          return;
     }
     public void checkServerTrusted(X509Certificate[] p1, String p2)
     {
          return;
     }
     public X509Certificate[] getAcceptedIssuers()
     {
          return 0;
     }
}
```

# Chapter 2

# SSL & SSL Applications in Android

For encrypted communications The Secure Sockets Layer (SSL) is the de facto communication protocol between two points generally a server and a client. If the implementation is incorrect it is possible that an attacker can intercept the communication and place a man in the middle attack. SSL is based on Public-Key Infrastructure (PKI). Which means there is a public key and a private key that are paired. A server holds both public and private keys and publishes public key. Whenever a client wants to communicate through secure channel, uses this public key and initiates a handshake procedure with the server. Server proves it has the private key by signing its certificate with public-key cryptography. And a secure channel is established between two.

At this point it crucial to know that certificate is genuine and the certificate holder is the exact person that he told so. Because a valid forged certificate can be published and without validation a secure channel is established not with the server but with the attacker. To solve this issue server and clients host root certificates of Certificate Authorities (CAs) which are well known. Certificate Authorities work as a notary for certificates to approve that certificate is belong to the one that client is trying connect to or disapproves that certificate is not belong to the claimer. This is the first extension point of Android: Custom TrustManager. Custom TrustManager is the extension point where a developer can change the validation logic of certificate and CAs. Developer can introduce a new CA, can ignore well known CAs, deploys and validates self-signed certificates. This can be used for hardening like certificate pinning or disabling security like always success resulted certificate checks.

Certificate might be from a trusted source but it might still issue for someone else you are trying to connect to. So while certificate is being checked for validation, subject or subject alternative name fields compared to match the server you are trying to reach. This is the second extension point of Android: Custom HostnameVerifiers. This extension point gives flexibility to developer to check hostname with its own algorithm. With virtual hosting, when sharing a server for more than one hostname with HTTPS, this can be handy.

Third extension point is WebViewClient. Which is used to extent the secure communication that occurred in a browser component used in the application.

Those are the extension points for specific needs. Without extending secure connection in Android is as easy as:

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

# Chapter 3

# Application Testing Methodology

For our thesis we basically follow static and dynamic testing methodologies. If we break down into phases:

1. Downloading applications from App Store
2. Performing static tests for each application
3. Analyzing results
4. Performing dynamic tests for a few selected applications
5. Creating a proof of concept application to identify SSL miss-implementations in the applications.

## 3.1 Challenges

Downloading applications from App Store is the first step of testing and probably is the difficult one. After some research for gathering applications without using phone by downloading one by one we stumble upon a project at github called google-play-crawler[5] which is developed by Ali Demiröz. Google Play Crawler Java Api searches android applications on Google Play, and downloads them. To start crawling Google Play Crawler needs a list of applications or list of categories on the application market. We got the application categories from Google Play and fed Google Play Crawler with those information. When crawler started to download applications, Google Play

banned the crawler from downloading because it detected as an attack to the system for too many requests in a short period of time. Adjustments made by try and error like delaying requests and Crawler started again to download applications. As a result 8882 applications downloaded which allocates almost 16 Gigabytes of space.

Static analysis took roughly 13.500 minutes to run. The total amount of time spent to finish static analysis took more because we need to include setup time, installing applications, configuring, writing required scripts to parse output, copying files and etc.

We try to test nogotofail[6] setup from google for dynamic testing but we couldn't manage to connect OpenVPN[7] server from the device which is necessary to investigate the traffic. OpenVPN needs tun.ko kernel module which is not available on Android X86 port. We performed our dynamic analysis with the CERT Tapioca[8] virtual machine.

## 3.2 Static Analysis

After gathering applications from App Store we setup a system for Static Analysis. This setup uses Androguard and mallodroid[9].

Androguard[10] is a tool written in python to analyze, inspect and change Application packages (APK) .apk files, Android's binary xml, Android Resources and the most important one disassemble de-compilation of .dex (Dalvik virtual machine) files which is compiled code of the application.

Mallodroid is an Androguard extension, runs on top of Androguard and it finds broken SSL certificate validation in Android Apps. Mallodroid is developed by Sascha Fahl et al to perform SSL miss-implementation inspect in applications. Details of the work is explained at the Related Work at the following sections. We need to slightly change mallodroid code to handle WebViewClient extension and we deploy some exception handling. Also some performance enhancements was made.

Analyzing an application takes 2 to 5 minutes according to application size because there are many steps:

- Unpacking APK file
- Decompiling files

- Converting from DEX to java language
- Constructing function call hierarchy
- Test java code to find out if there is an extension for the methods that are related to SSL communication
- And finally test if they correctly implement SSL extension.

## 3.3 Inspection of results

Each static result of an application is written to a file. After static analysis those files collected and parsed through a program which is written in Microsoft C#. Parsed data than inserted to database to benefit from SQL capabilities to create reports of the result. Implementation types categorized, applications that uses pinned certifications identified, used Libraries classified, Custom TrustManagers identified and so on. 201 pinning implementations found out of 4590 applications that extent libraries.

From the Implementation stand point 69 percent of the implemented libraries are having problem with Custom TrustManager.

TABLE 3.1: Number of Vulnerable Implementations by Implemention Type

| Implemention Type | Number of Vulnerable Implementations |
|---|---|
| WebViewClient | 9 |
| Custom TrustManager | 2.964 |
| Custom HostnameVerifiers | 1.343 |

Some of the Custom TrustManager package names are self-explanatory like TrustAllManager, TrustingX509TrustManager or SSLSocketFactoryTrustAll which are vulnerable to MITM because they are not validating SSL certificates. And PinningTrustManager is not vulnerable and its name states that it's validating certificate against a pining certificate.

Game categories have a huge Number of Vulnerable Applications compared to other Categories.

73 percent of Library Implementations are generated by third party components used by application developers. 40 percent of these implementations are ads and analytics cumulatively. This is a good point because generally these libraries doesn't collect high

TABLE 3.2: Custom TrustManager Examples

| Custom TrustManager | Is Vulnerable | Number Of Apps |
|---|---|---|
| MySSLSocketFactory | TRUE | 588 |
| DefaultTrustManager | TRUE | 247 |
| NaiveTrustManager | TRUE | 214 |
| HttpRequest | TRUE | 125 |
| PinningTrustManager | FALSE | 116 |
| SSLSocketFactoryEx | TRUE | 77 |
| TrustManagerDecorator | FALSE | 67 |
| SslUtils | TRUE | 63 |
| TrustingX509TrustManager | TRUE | 56 |
| SSLSocketFactoryTrustAll | TRUE | 54 |
| TrustManagerDelegate | FALSE | 53 |
| FileTransfer | TRUE | 68 |
| TrustEveryoneTrustManager | TRUE | 48 |
| SPSSLSocketFactory | FALSE | 42 |
| EasyX509TrustManager | FALSE | 34 |
| EasySSLSocketFactory | TRUE | 33 |
| IgnoreCertTrustManager | TRUE | 28 |
| SandboxSSLSocketFactory | TRUE | 26 |
| FTPSTrustManager | FALSE | 20 |
| HttpUtil | TRUE | 20 |
| TrivialTrustManager | TRUE | 17 |
| TrustAllManager | TRUE | 15 |
| NonValidatingTrustManager | TRUE | 14 |

sensitive data but one must aware that these libraries generally collect Meta data like person's usage habits.

A risk factor is calculated for each risky application. This risk factor is calculated with different indications like permission requests, number of downloads and category of the application. Permission requests graded for their impacts to the use like application with ACCESS_FINE_LOCATION permission top graded from the risk stand point while FLASHLIGHT permission low graded. Number of downloads is graded according to the number of user impacted from the risk like "5+" downloads impacts maximum 10 people and has low severity but "1,000,000,000+" downloads impacts maximum "2000,000,000" users and has top graded risk score. Category of the application is differentiate the factor from the risk point like the application is important if it's in the finance category but far less important if it's in one of the game categories. All points calculated and applications graded for their risk points and they are normalized with the equation 3.1 into ten risk groups. And Impact Score is calculated as in equation 3.2

$$mean = 1 + (ImpactScore - \min_{ImpactScore})) * (10 - 1)/(\max_{ImpactScore} - \min_{ImpactScore}) \quad (3.1)$$

TABLE 3.3: Number Of Vulnerable (Miss-implemented SSL) Applications by Category

| Category | Number Of Apps | Cumulative Risk | Percentage of Affected Persons |
|---|---|---|---|
| Game Total | 1.173 | 68.771 | %42,03 |
| Travel And Local | 67 | 5495 | %0,80 |
| Social | 66 | 5839 | %1,74 |
| Business | 66 | 6633 | %1,03 |
| Personalization | 64 | 5950 | %0,03 |
| Entertainment | 63 | 4135 | %6,92 |
| Communication | 61 | 8676 | %21,85 |
| Shopping | 56 | 3780 | %0,14 |
| Transportation | 56 | 4176 | %0,45 |
| Sports | 55 | 3877 | %0,38 |
| Medical | 55 | 3272 | %0,10 |
| App Wallpaper | 53 | 4382 | %12,16 |
| News And Magazines | 51 | 2888 | %1,47 |
| Finance | 51 | 4222 | %0,21 |
| Productivity | 50 | 4386 | %0,72 |
| Education | 49 | 2508 | %0,19 |
| Media And Video | 48 | 2896 | %0,60 |
| Weather | 44 | 3446 | %1,05 |
| Music And Audio | 41 | 2901 | %4,42 |
| Lifestyle | 40 | 3042 | %0,78 |
| Tools | 37 | 3398 | %1,21 |
| Health And Fitness | 35 | 2636 | %0,91 |
| Comics | 33 | 1888 | %0,07 |
| Books And Reference | 17 | 789 | %0,69 |
| Photography | 14 | 923 | %0,06 |
| Libraries And Demo | 5 | 236 | %0,00 |
| App Widgets | 4 | 368 | %0,00 |
| Total | 2.354 | 161.513 | %100,00 |

$$\begin{aligned} ImpactScore = \sum RiskAccordingtoPermissionRequests \\ + RiskAccordingtoNumberofDownloads \\ + RiskAccordingtoCategoryofTheApplication \end{aligned} \tag{3.2}$$

From the result we see that launcher applications request many important permissions and when we add up category information and download number they have the first two order within the highest ranked risk group. Then comes the messenger applications.

## 3.4 Dynamic Analysis

We used Android-x86 Project for our application test. Although testing in real devices is the desired solution but it's not practical. And creating an isolated environment is better to prevent interventions. We created a virtual machine as our Android device and

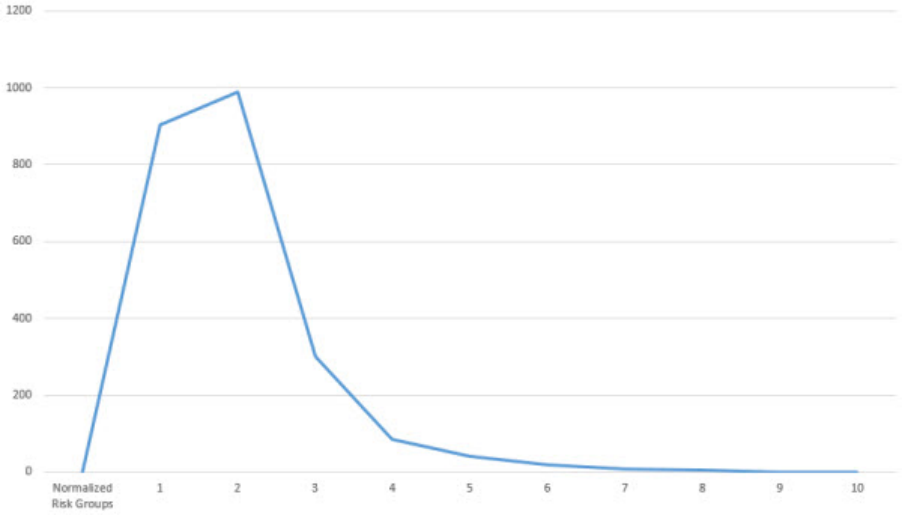


FIGURE 3.1: Number of Applications by Normalized Risk Groups

make configurations like network setup etc. Then took a snapshot for our base to revert after each test. We use adb to push our applications. And our virtual devices connection flow through our proxy environment to test for SSL vulnerabilities[11].

### 3.4.1 CERT Transparent Proxy Capture Appliance (Tapioca)

A similar test as in our scenario was held in CERT Coordination Center (CERT-CC) at Carnegie Mellon University. They evaluated some application tools used as MITM proxies but they decided to go with a transparent network layer. With a transparent network layer proxy an application is not aware that it is being proxied. So they created a preconfigured VM appliance to perform MITM analysis called CERT Tapioca[8].

The setup is simple; a VM that has two network adapters: one for the outer side and one for the local side. On the local side, it provides NAT, DHCP, and DNS capabilities. For the MITM proxy, mitmproxy software is being used. To configure things up startup scripts prepared and it's ready to use listening and manipulating traffic at port 443.

### 3.4.2 Nogotofail

Nogotofail is more complete tool to spot and fix weak TLS/SSL connections and sensitive clear text traffic which is maintained by Google. It includes testing for common SSL certificate verification issues, HTTPS and TLS/SSL library bugs, SSL and STARTTLS stripping issues, clear text issues, and more. It's a Linux based tool that depends on

Python and pyOpenSSL. It has also client tools to enrich log reports that includes extra client information application information etc. Traffic analyzer can be deployed as a router, VPN server, or proxy. There are three kind of installations first can run on an actual router, second a Linux box with two network interfaces. And the last one as a VPN server. Google also provide last one almost preconfigured as Google Compute Engine instance[6].

We tried to deploy nogotofail on a virtual machine and try to connect from real device and we couldn't succeeded. And OpenVPN client application couldn't be deployed to Android-x86 device. We decided to try Google Compute Engine instance and we succeeded. We preferred CERT Tapioca because it's simple and local.

## 3.5 Cross Reference Traversing

We can find out extension of SSL libraries by decompiling the APK files. Then we can investigate the class if it is vulnerable or not. But testing the application from the device against the vulnerability is not simple. We need the exact user interface where the classes executed. To ease the testing procedure we created a python script called XrefTree A on top of androguard to expose Cross References and traverse each path form the SSL extension class to UI interfaces. With this our test is much easier because we know how to trigger the communication.

## 3.6 Analysis and Results

We choose 41 applications for dynamic analysis from top 400 applications ordered by descending by their risk factor. The applications picked are mainly targets Turkish users and some are specifically chosen for analysis like known to be not exploitable but implemented pinning certification. Applications tested with Android X86 port on a virtual machine which's connection transparently go through from a second virtual machine that has Cert Tapioca installed. Applications tested one by one with most feasible, user like behavior. After the process we analyze traffic pcaps with wireshark providing with SSL private keys which are used for test purposes to decode SSL traffic.

The results differentiated from static analysis. Because all usage paths couldn't be covered, applications require some special info like account info, user, password entry etc. And some of them couldn't be tested that they require some components that we couldn't provide like location info or our device known to be a tablet and application doesn't provide and quits. Some of the applications has SSL connection libraries but while dynamic test they only connect through HTTP.

TABLE 3.4: Dynamic Analysis Break Down

| Analysis Result | Number of Applications |
|---|---|
| SSL Vulnerable to MITM | 4 |
| HTTP Connection Only | 6 |
| Can not Establish Connection | 27 |
| Not Applicable | 4 |

As a result four of the applications; that's a proportion of 9.76 percent in our test observed that they are vulnerable to mitm attacks. The percent is not high as static analysis. But two out of four vulnerable applications are belong to financial institutions (banks) and all of the banking transactions can be sniffed by an attacker.

# Chapter 4

# Proposed Solution

We developed a proof of concept application as an Xposed module to identify SSL miss-implementations in the applications.

## 4.1 Xposed

Xposed is an application that's bind itself to the base process of Android and provide application/method call hook points to its modules to perform various operations like UI tweaks, new features, feature enhancements etc.

Android runtime has a base process called "Zygote". Every application is started as a copy ("fork") of it. When the phone is booted /init.rc script started Zygote. The process start is done with /system/bin/app_process, which loads the needed classes and invokes the initialization methods. When a user install Xposed, an extended app_process executable is copied to /system/bin. When the system boots it's now a part of Zygote. Some initializations are done there and also the modules are loaded. Zygote gets called in the very beginning of the process.

When modifications done by decompiling, reverse engineering an APK; one has to dive into the code find the exact place to modify. After findings there are other steps patch, recompile, sign and pack for the apk. Also the application must be distributed. And when you decompile generally that means that you don't own the application and you can't sign the apk on behalf of the owner. And this patch operation is valid just for that

version, when there is an update, all the work needs to be done from scratch. Xposed has another way of doing patches: By hooking to methods of the target applications which are the smallest unit in Java. With Xposed you can inject your own code before and after methods.

Xposed has a plugin system that are loaded with system reboot which are special applications consist of target application hook method signatures and the code that wanted to be injected before or after the method call. When target application started, desired methods found and hooked with reflection and when the function call placed; code executed which can access all the method's parameters and it can change the values of the caller[12].

## 4.2 Trust But Verify

We developed an Xposed module named "Trust But Verify" for our proof of concept application. There are three different extension points for different parts of the SSL connection. And we hook up certain methods for each of the implementation type. After the hook we forged a certificate and test method against the certificate. If the method validates the certificate it fails. This means the application can be attacked by a man in the middle. If the method fails to validate then it is secure for improper certificates. The application is now just generating notification logs. But it can be enhanced to block for improper buggy implementations of SSL extensions.

For Custom HostnameVerifiers we hook up javax.net.ssl. HttpsURLConnection method. For WebViewClient implementation type we hook up android.webkit.WebViewClient method. And for Custom TrustManager com.android.org.conscrypt. TrustManagerImpl, checkServerTrusted and javax.net.ssl. TrustManagerFactory methods hooked up.

# Chapter 5

# Development Best Practices for Security & Privacy

## 5.1 Certificate Pinning

SSL's strength is also weakness of it: Negotiating on a key for symmetric encryption is done by asymmetric cryptography. The public key that is used for encryption is pre shared with clients via operating systems or the software programs that is used like browsers. These pre-shared pre-defined keys are called trust anchors are generated by and belong to trusted third parties called certificate authorities (CA).

As a result of EFF's SSL Observatory[13] project which aims to investigate the publicly-visible SSL certificates there were 650 Certificate Authorities on August 2010. And there are 162 root certificates CA's already installed as trusted authority on a stock rom installed Android operating system. This number can increase with carrier like AT&T, Verizon or device manufacturer like Samsung, HTC etc. customized installations[14]. Certificate authorities sign intermediate certificates that have ability to sign end user certificates (End Entity) this certificate generation and signing process is general purpose so any issuer can generate and sign certificates for any domain. When one of the certificate of CA or intermediate certificates compromised, hacker can issue a genuine certificate whichever domain he wants. And there are 162 certificate authorities for android and more than 650 certificate authorities for internet. Certificate compromisation

happens rarely but with big negative impacts on security as seen on DigiNotar, Comodo and TurkTrust cases.

DigiNotar was a Dutch certificate authority owned by VASCO Data Security International, whose security breached and incident declared on September 3, 2011. Fake DigiNotar certificates were found and hackers created fake Gmail domain certificates and used for man-in-the-middle attacks[15].

One of Comodo reseller user account compromised and created 9 certificates, across 7 different domains including www.google.com, login.yahoo.com, login.skype.com certificates were revoked after discovery[16].

On December 24, 2012 Google discovered fake certificates issued for "*.google.com" via its browser Chrome's certificate pinning for Google domains. TurkTrust certification authority (CA) has been reported that two intermediate CA certificates inadvertently issued in August 2011. The certificates were issued in error and they were for "*.ego.gov.tr" and "e-islem.kktcmerkezbankasi.org." "*.google.com" certificate issued automatically by Check Point firewall which was configured for inspection generates certificates for all SSL connections. Google Chrome and other browsers blacklisted the inadvertent intermediate CA certificates and published metadata update to block the mistaken CA certificates[17].

To mitigate the attack surface certificate pinning might be a solution for defense in depth like layering tactic. Unlike browsers or the tools that doesn't know about the connection destination of the client; application generally connects same and already known server's addresses. So some of the certificate information can be added to double check against the certificate that is going to be used while communicating. It's prevent man-in-the-middle attack with generated certificate for that domain using compromised issuer certificate except the one that's used for genuine certificate issuer. For the time being with current number of trust anchors in Android it's provide protection at a rate of %99.38

For certificate pinning hex-encoded hash of a X.509 certificate's SubjectPublicKeyInfo used. Using certificate hashes mislead wrong directions. Because there are multiple certificates with the same public key, subject name. But certificates might have different extensions, different expiry dates and there might be different certificates signed with different cryptographic hash function like one with SHA-1 and other with SHA256. Devices

might build certificates chains with an alternative version of a certificate than the one that expected. Certificate validation is started from End Entity which is leaf certificate contains a signature which must be a valid signature from its parent. So that the public key of the parent is fixed by the leaf certificate. Which's public key info hash is safe to use for pinning[18] [19]. A set of pins can be defined for certificate alternatives as well.

There's a working Android library[20] for certificate pinning by Moxie Marlinspike and example project by Ivan KuÅ¡t[21]. Required pin hash can be generated using the provided script from Marlinspike's library providing certificate file.

Pin generation:

```
$ git clone https://github.com/moxie0/AndroidPinning.git
$ cd AndroidPinning
$ python ./pin.py /path/to/cacert.pem
```

```
TrustManager[] trustManagers;
trustManagers = new TrustManager[1];
trustManagers[0] = new PinningTrustManager(
        new String[] {"f30012bbc18c231ac1a44b788e410ce754182513"});

SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, trustManagers, null);

String url = "https://encrypted.google.com/";
HttpsURLConnection urlConnection = (HttpsURLConnection)new URL(url)
        .openConnection();
urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());

InputStream in = urlConnection.getInputStream();
```

FIGURE 5.1: Using PinningTrustManager for certificate validation with pin information

There is also more rigid way for pinning: That's left behind all trust anchors, CA's left behind and use freshly forged for the application. On the server side a new strong certificate can be created to sign certificates for application. This can be done via a hardware security module (HSM) which is a physical computing device that safeguards and manages digital keys for strong authentication and provides crypto processing or with using OpenSSL. It's a best practice to keep private keys offline for security purposes. Certificates that are going to be used by application for secure communication than distributed with applications package in a keystore file which can easily be created using

keytool. And definitely application had to provide this keystore to TrustManagerFactory in order to use created certificate. This is a hardcoded way and downside of this method is it is difficult to manage certificate updates. But its provide immunity for all trust anchors compromisation risk[22].

Create a keystore using keytool:

```
$ wget http://bouncycastle.org/download/bcprov-jdk16-146.jar
$ keytool -importcert -file your_signing_certificate.pem
        -keystore yourapp.store
        -provider org.bouncycastle.jce. provider.BouncyCastleProvider
        -providerpath bcprov-jdk16-146.jar -storetype BKS
```

```
private InputStream makeRequest(Context context, URL url) throws Exception{
    AssetManager assetManager;
    assetManager = context.getAssets();
    InputStream keyStoreInputStream;
    keyStoreInputStream = assetManager.open("your.app.store");
    KeyStore trustStore;
    trustStore = KeyStore.getInstance("BKS");

    trustStore.load(keyStoreInputStream, "some password".toCharArray());

    TrustManagerFactory tmf;
    tmf = TrustManagerFactory.getInstance("X509");
    tmf.init(trustStore);

    SSLContext sslContext;
    sslContext = SSLContext.getInstance("TLS");
    sslContext.init(null, tmf.getTrustManagers(), null);

    HttpsURLConnection urlConnection;
    urlConnection = (HttpsURLConnection)url.openConnection();
    urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());

    return urlConnection.getInputStream();
}
```

FIGURE 5.2: Standard HTTPS request with pre-forged self-certificate.

You also need to be aware that it's possible to bypass SSL Pinning (pdf). However, this requires the app to be reverse engineered, re-constructed and re-run that's very unlikely to ever be possible 'on the fly' (at least on unlocked devices) as a random user gets hit by a MITM attack[23].

## 5.2 Certificate pinning in Android

With the release of Android 4.2 Jelly Bean[24] Certificate Pinning introduced as a new feature "Certificate Pinning - The libcore SSL implementation now supports certificate pinning. Pinned domains will receive a certificate validation failure if the certificate does not chain to a set of expected certificates. This protects against possible compromise of Certificate Authorities"

Pin information is stored in a file called "pins" can be located in the /data/misc/keychain directory which has a format of:

```
hostname=enforcing[true|false]|SPKI SHA512 hash, SPKI SHA512 hash,...
```

Format can be translated as there are list of SPKI hashes (SHA512) separated by commas with enforcing either true or false for a domain. There is no pre-configured built-in pins. Pin informations are valid until it is removed from the file. Pin check is integrated in libcore. If X509TrustManager implementation (TrustManagerImpl) used for validating certificate chains, pin information used for validating otherwise the standard checkServerTrusted() method doesn't consult the pin list. This is because to provide backwards compatibility from the user perspective using latest doesn't necessarily mean that your connection always validated against system-level certificate pins. Third party applications can benefit system-level certificate pin information via the new X509TrustManagerExtensions[25] SDK class which has a single method: that returns a validated chain on success or throws CertificateException if validation fails.

```
public List<X509Certificate> checkServerTrusted
        (X509Certificate[] chain, String authType, String host)
```

FIGURE 5.3: Verifies the given certificate chain.

The chain parameter holds the peer certificate chain and authType parameter is for the key exchange algorithm used. The final parameter, host, should be the hostname of the server.

**Returns**

The properly ordered chain used for verification as a list of X509Certificates.

**Throws** `CertificateException`

if the chain does not verify correctly.

Client handshake (`ClientHandshakeImpl`) and SSL socket (`OpenSSLSocketImpl`) implementations at the default SSL engine (JSSE provider) use pin information to validate for that host. If there is an entry for that host and couldn't validate certificate chain, validation fails with `CertificateException` and connection won't be established.

For the time being pins file doesn't exist in the stock roms, which means that there is no certificate pin information. Functionality implemented but it doesn't used efficient enough. The pins file is not written directly by the OS instead the pin list is updated by sending a broadcast with signed update data which is triggered by a broadcast (android.intent.action.UPDATE_PINS) that contains the new pins. As mentioned it is signed with SHA512 with RSA signature. The receiver of the broadcast (CertPinInstallReceiver) will then verify the signature and update pin information. Public key used for validation is stored as a system secure setting under the "config_update_certificate" key (usually in the secure table of the /data/data/com.android.providers.settings/databases /settings.db)[26]

## 5.3 Alternative Methods for Certificate Validation

Public Key Pinning Extension for HTTP (PKPE) by Google and Trust Assertions for Certificate Keys (TACK) by Moxie Marlinspike are two standards that have been proposed for Certificate Validation. And there is also a method called "Certificate Memorizing".

Google, proposes a new HTTP header (Public-Key-Pin, PKP) with PKPE that holds pinning information including public key hashes, pin lifetime and whether to apply pinning to subdomains of the current host. Header information is delivered to via TLS encrypted connection. And keys are validated against pre cached pins or accepted as valid for the first arrival to the expiration time. It also provides a mechanism for reporting fraudulent certificates used for MITM attacks via "report-uri" directive[27].

TACK, proposes an extension to SSL/TLS that carries pinning information signed with a dedicated 'TACK key' for hostname with an expiration time independent from its certificate. Short-lived tacks may be used to limit the effect of compromised TLS private key. It's an extension to SSL/TLS so it's backwards compatible. Tack[28] pins cached after a few visits.

```
Public-Key-Pins: max-age=31536000;
    pin-sha1="4n972HfV354KP560yw4uqe/baXc=";
    pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYpOLmCQ="

Public-Key-Pins: pin-sha1="4n972HfV354KP560yw4uqe/baXc=";
    pin-sha1="qvTGHdzF6KLavt4PO0gs2a6pQ00=";
    pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYpOLmCQ=";
    max-age=2592000

Public-Key-Pins: pin-sha1="4n972HfV354KP560yw4uqe/baXc=";
    pin-sha1="qvTGHdzF6KLavt4PO0gs2a6pQ00=";
    pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYpOLmCQ=";
    max-age=2592000; includeSubDomains
```

FIGURE 5.4: HTTP Response Header Field Examples

Certificate Memorizing method hands over the acceptance of the unknown certificate or different certificate previously seen before for a site visited to the user. There is a proof of concept Android project called MemorizingTrustManager[29] which asks the user whether to accept the certificate once, permanently or to abort the connection. The problem is that technically unsophisticated users might get confused and application usage can decrease drastically.

According to an academic paper[30] published by Adam Bates et al. from University of Florida SSL certificate verification can be done via dynamic linking for C/C++ SSL implementations. When an SSL library entry function is called their implementation took over SSL verification to enforce the correct SSL certification verification procedure. Without needing to change application code it can sit on top of implementations. Dynamic linking can be implemented as byte code instrumentation for JVM based applications. For Android applications it can be implemented as a core module or as byte code instrumentation between system and user modules.

Perspectives Project[31] at Carnegie Mellon University and Convergence project by Moxie Marlinspike, aim to eliminate pre authenticated Certificate Authorities (CA). Instead they propose agile, secure, and distributed notary based certificate validation. By eliminating CAs users are immune to CA compromises. A user can choose to trust several notaries; no single point of failure. And several notaries can vouch for a single site. Convergence implemented as a Firefox browser add-on and a server-side notary daemon.

# Chapter 6

# Related Work

On October 18, 2012, Sascha Fahl et al. from Leibniz University of Hannover and Philipps University of Marburg published a paper titled as "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security"[1]. They downloaded 13.500 popular free apps for the research from Google's Play Market and studied their properties with respect to the usage of SSL. The results showed that 1.074 apps potentially vulnerable to SSL MITM Vulnerability and they picked randomly 100 potentially vulnerable apps for manual dynamic analysis yielding 41 vulnerable apps to MITM attack. For static code analysis they have built MalloDroid, a tool to detect apps that potentially use SSL/TLS inadequately or implement incorrectly and thus are potentially vulnerable to MITM attacks. They also presented an online survey to explore whether or not the user can assess the security of a connection in the Android browser. Our work is an update for their reports with current applications for a Turkish market user and also completes their work with an application to identify flaws at run time.

Sounthiraraj, Sahs, Greenwood, Lin and Khan from The University of Texas at Dallas published a paper[2] about an application which combines static and dynamic analysis techniques to perform automated, large-scale SSL MITM Vulnerabilities detection for Android applications called SMV-HUNTER. Their application step forward from others with the ability of identifying UI targets to trigger vulnerabilities found from static analysis phase. Output of the static analysis also prepares some relevant input variables for performing automatic UI exploration while attempting MITM attacks. SMV-HUNTER

application itself didn't published to public. Our work has reports with statuses of current applications for a Turkish market user and also we published an application which identifies flaws at run time. Our work can be enhanced with automated tests as in this work.

Will Dormann, who is a member of CERT(computer emergency response team) at Carnegie Mellon University (CMU) published a blog post titled as "Finding Android SSL Vulnerabilities with CERT Tapioca"[32] about automated discovery of SSL vulnerabilities in Android applications. At CERT they've created a Linux distribution for MITM analysis called CERT Tapioca[33]. With the use of CERT Tapioca and some scripts they managed to automate tests and they notified application developers with vulnerability details. Some of the applications and application generation frameworks like AppsGeyser got fixed with in the following timeframe. They also published and maintained a list of vulnerable applications as an Android application SSL spreadsheet[34] at Google docs. Dormann also inserted vulnerable applications to CVE (Common Vulnerabilities and Exposures) database. CVE is the de facto standard for tracking vulnerabilities in applications. Up to this date 1.000.462 applications tested and 23.667 of them have failed dynamic testing. This work also presented at RSA Conference 2015[35] in San Francisco by Dormann and Montelibano; presentation subject was "How We Discovered Thousands of Vulnerable Android Apps in One Day". Our work is an update for their reports with current applications for a Turkish market user and also completes their work with an application to identify flaws at run time.

Fireeye researchers Adrian Mettler, Yulong Zhang, Vishwanath Raman published an article on 20th August 2014 titled as "SSL Vulnerabilities: Who listens when Android applications talk?"[36] about SSL Vulnerabilities in Android applications. Researchers reviewed the 1,000 most-downloaded free applications in the Google Play store as of July 17, 2014 with their commercial product "Mobile Threat Prevention" platform. Article contains summary of their findings that 614 applications that use SSL/TLS to communicate with a remote server, 448 ( $\sim$73%) of them do not check certificates. Article also includes some references for best practices. Our work is an update for their reports with current applications for a Turkish market user and also completes their work with an application to identify flaws at run time.

# Chapter 7

# Conclusions

In this thesis, we plan to analyze applications that are popular at the market against SSL miss-implementations. This miss-implementations can cause leakage of private information like financial data, health records or passwords of the users of that application.

Our study began with working on theoretical information like Android operation system internals, SSL implementations while application development and etc. After theoretical information it's time to analyses the applications. Second phase began with gathering applications from application market which is Google Play Store. 8882 applications gathered from 45 different categories. Those applications go through a static analysis and identified as safe, potentially risky application. Risky applications analyzed for a second time for their permission requests. Then a small number of selected application go through a dynamic analysis.

A risk factor is calculated for each risky application. This risk factor is calculated with different indications like permission requests, number of downloads and category of the application. All factors calculated and applications graded for their risk points and they are normalized in to ten risk groups.

At the end of our analysis we investigate our findings. A risk factor is calculated from that findings for each risky application. 2354 application has at least one miss use of SSL libraries out of 8882.

Our findings from dynamic analysis was: Applications are less vulnerable compared to static analysis results. The vulnerable libraries might not be used or they are used by

a specific path application. But results confirmed that the impact of vulnerabilities can be enormous.

As a conclusion, we observed that almost 27 percent has a vulnerability from SSL connection stand point. The main reasons for these vulnerabilities are developer errors and third party generators or libraries. Using third party libraries can cause security bugs which leads to informations leakage or exploitation. A developer must have deeply understanding of what he is developing especially about security concept. Users need to be cautious and take necessary security precautions while using applications. Always they need to consider about security. Android application market can enforce some implementation procedures for SSL extension libraries.

# Appendix A

# Cross Reference Traversing

```
import sys,string
from androguard.core.bytecodes.dvm import DalvikVMFormat
from androguard.core.bytecodes.apk import APK
from androguard.core.analysis.analysis import uVMAnalysis
from androguard.core.analysis.ganalysis import GVMAnalysis

def XrefTraverse(methods, class_name, method_name, depth):
    depth += 1
    for m in methods:
        if m.class_name == class_name and m.name == method_name:
            if depth == 0:
                print (m.class_name + " -> " + m.name)
            for item in m.XREFfrom.items:
                if item[0].class_name != class_name \\
                        or item[0].name != method_name:
                    for x in range(1, depth):
                        sys.stdout.write('--')
                    sys.stdout.write ('>' + item[0].class_name + "->" \\
                                      + item[0].name + "\n")
                    XrefTraverse(methods, \\
                                 item[0].class_name, item[0].name, depth)
```

```
if len(sys.argv) > 2:
    filename = sys.argv[1]
    class_name = sys.argv[2]
    class_name = 'L' + class_name.replace(".", "/") + ";"
    #print class_name
    method_name = '<init>'
    d = DalvikVMFormat(APK(filename, False).get_dex())
    d.create_python_export()
    dx = uVMAnalysis(d)
    gx = GVMAnalysis(dx, None)
    d.set_vmanalysis(dx)
    d.set_gvmanalysis(gx)
    d.create_xref()

    XrefTraverse(d.get_methods(), class_name, method_name, 0)
else:
    print "usage: XrefTree.py [filename] [class_name]"
    print "usage: XrefTree.py filename.apk com.xyz.abc"
```

# Bibliography


[1] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012. URL `http://android-ssl.org/files/p50-fahl.pdf`.

[2] D. Greenwood, J. Sounthiraraj, G. Sahs, Z. Khan, and L. Lin. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. 2014. URL `https://www.utdallas.edu/~zxl111930/file/NDSS14b.pdf`.

[3] The heartbleed bug, April 2014. URL `http://heartbleed.com/`.

[4] Openssl vulnerabilities. URL `https://www.openssl.org/news/vulnerabilities.html`.

[5] A. Demiröz. Google play crawler java api, February 2014. URL `https://github.com/Akdeniz/google-play-crawler`.

[6] Google. nogotofail: An on-path blackbox network traffic security testing tool, November 2014. URL `https://github.com/google/nogotofail`.

[7] J. Yonan. Openvpn, May 2001. URL `https://openvpn.net/`.

[8] W. Dormann. Announcing cert tapioca for mitm analysis, August 2014. URL `https://insights.sei.cmu.edu/cert/2014/08/-announcing-cert-tapioca-for-mitm-analysis.html`.

[9] S. Fahl. Mallodroid, August 2013. URL `https://github.com/sfahl/mallodroid`.

[10] A. Desnos. Androguard, June 2012. URL `https://github.com/androguard`.

[11] C. Huang. Android-x86 - porting android to x86. URL `http://www.android-x86.org/`.

[12] rovo89. Xposed bridge, July 2013. URL https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[13] The Electronic Frontier Foundation. The eff ssl observatory, July 2010. URL https://www.eff.org/observatory.

[14] Android Developer Team. Ca certs, . URL https://android.googlesource.com/platform/libcore/+/master/luni/src/main/files/cacerts/.

[15] Wikipedia. Diginotar, September 2011. URL http://en.wikipedia.org/wiki/DigiNotar.

[16] Comodo. Comodo fraud incident, March 2011. URL https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html.

[17] A. Langley. Enhancing digital certificate security, January 2013. URL http://googleonlinesecurity.blogspot.com.tr/2013/01/enhancing-digital-certificate-security.html.

[18] A. Langley. Public key pinning, May 2011. URL https://www.imperialviolet.org/2011/05/04/pinning.html.

[19] Open Web Application Security Project. Certificate and public key pinning. URL https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning.

[20] M. Marlinspike. A standalone library project for certificate pinning on android.

[21] K. Ivan. Example of certificate pinning on android, May 2014. URL https://github.com/ikust/hello-pinnedcerts.

[22] M. Marlinspike. Your app shouldn't suffer ssl's problems, December 2011. URL http://thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha.

[23] D. Andzakovic. Bypassing ssl pinning on android via reverse engineering, May 2014. URL https://dl.packetstormsecurity.net/papers/general/android-sslpinning.pdf.

[24] Android Developer Team. Jelly bean, . URL http://developer.android.com/about/versions/jelly-bean.html.

[25] Android Developer Team. X509trustmanagerextensions, . URL http://developer.android.com/reference/android/net/http/X509TrustManagerExtensions.html.

[26] N. Elenkov. Certificate pinning in android 4.2, December 2012. URL http://nelenkov.blogspot.com.tr/2012/12/certificate-pinning-in-android-42.html.

[27] R. Sleevi, C. Evans, and C. Palmer. Public key pinning extension for http. 2015.

[28] M. Marlinspike. Trust assertions for certificate keys. 2013.

[29] G. Lukas. Memorizingtrustmanager - the android trustmanager, March 2014. URL https://github.com/ge0rg/MemorizingTrustManager/wiki.

[30] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. Butler, and A. Alkhelaifi. Securing ssl certificate verification through dynamic linking. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 394–405. ACM, 2014.

[31] Perspectives project. URL http://perspectives-project.org/.

[32] W. Dormann. Finding android ssl vulnerabilities with cert tapioca, September 2014. URL https://insights.sei.cmu.edu/cert/2014/09/-finding-android-ssl-vulnerabilities-with-cert-tapioca.html.

[33] W. Dormann. Announcing cert tapioca for mitm analysis, August 2014. URL https://insights.sei.cmu.edu/cert/2014/08/-announcing-cert-tapioca-for-mitm-analysis.html.

[34] W. Dormann. Android apps that fail to validate ssl. URL https://docs.google.com/spreadsheets/d/1t5GXwjw82SyunALVJb2w0zi3FoLRIkfGPc7AMjRF0r4/edit#gid=123856677.

[35] J. Montelibano and W. Dormann. How we discovered thousands of vulnerable android apps in one day, April 2015. URL https://www.rsaconference.com/events/us15/agenda/sessions/1638/how-we-discovered-thousands-of-vulnerable-android.

[36] A. Mettler, Y. Zhang, and V. Raman. Ssl vulnerabilities: Who listens when android applications talk?, August 2014. URL `https://www.fireeye.com/blog/threat-research/2014/08/ssl-vulnerabilities-who-listens-when-android-applications-talk.html`.