Android Reconnaissance

Christopher Short
cshort@email.wm.edu

Abstract

Smartphones have allowed people from all over the world share information that otherwise would have been able to do so easily. It is estimated that there are 2.5 billion smartphones on the planet with more people obtaining them everyday. This high number of users has led to an also extraordinarily high number of applications to download with creators of these applications ranging from multinational corporations with fully staffed development teams to a high schooler who just learned how to program. Many of these applications contain known security vulnerabilities whether through ill intent or ignorant practices. In this paper, I focus on collecting data on 100 applications. These applications will be broken up into two different, even groups: an IoT application set of 50 and a random application set of 50. For sets of applications I performed the same tests and ran the same scripts, gathering as much information as a can between these two distinct groups. The focus of this paper is to discuss my findings on these particular sets of applications on whether or not they can be considered vulnerable to information or data leaks base on my defined characterization of a vulnerable application. The purpose of this paper is show the vast amount of ”vulnerable” applications that plague the Google Play Store. 68 percent of applications tested were marked as vulnerable.

1 Introduction

This problem of insecure programming is considerably larger than simply what this paper can offer. However, this does not make this research insignificant. All research performed was based around five research questions:

1. Do applications that require the user to accept all requested permissions, or otherwise terminate, have a higher chance of being classified as vulnerable?
2. Do applications that modify SSL socket factories use known broken SSL verification API calls?
3. Do applications use an already known broken modified trust manager?
4. Do applications allow all hostnames?
5. Do applications that use HTTP over than HTTPS?

The first research question addressed deals with permissions. The purpose of a permission is to protect the privacy of an Android user. Android applications must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and the internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request. In Android 6.0 and later, applications request to use permissions either at start up or when that permission is needed such as when the camera is started. In earlier versions of android permission request was dealt with at time in install time. This previous permission request model was an all-or-nothing model where if a user did not like the requested permission(s) then then the two options were to install anyway or not install at all. The new model allows the user to pick which permissions to accept and which permissions to not grant. Developers, however, have developed a method of incorporating the all-or-nothing model into the new permission request model. In this new method, if a users decides to not grant a specific permission at runtime then the application will simply terminate. Unfortunately, in the current state of this new model, there is no work around for users in the situation that they come across an application that they wish to use but is developed using this integrated all-or-nothing model.

Research questions two, three, and four deal with Secure Sockets Layer and certificate verification. Secure Sockets Layer, or SSL, is a standard security technology for establishing an encrypted link between a server and a client, typically a web server and a browser, or a mail server and a mail client. It's a security protocol. SSL uses a cryptographic system that uses two keys to encrypt data: a public key known to everyone and a private or secret key known only to the recipient of the message. This information is passed between the client and the server until the information is agreed upon to create what is known as a handshake. There are multiple entities involved with creating a handshake between a client and a server. Certificate verification is one step of this process and it too involves multiple entities. A SSL socket factory is used to dictate the exact setup of the SSL protocol to use in the handshake attempt. A trust manager is used to verify the certificate of the server. A hostname verifier is used to verify that the URL's hostname and the server's identification hostname match. If any of the verification checks fail then the handshake does not take place and the connection fails. The client can either give up or try again. On android, these verification steps can be tricky and frustrating to deal with. In this case developers can choose to use modified SSL socket factories, modified trust managers, or modify how it accepts hostnames. This act of modification alone isn't harmful and in some cases can even add security. Applications are made vulnerable, however, when developers use modified SSL socket factories, trust managers, and hostname verifiers that are very lenient or even unrestrictive. These situations can easily be taken advantage of in attacks known as man-in-the-middle attacks.

The final research question, five, dealt with the use of HTTP and HTTPS. HyperText Transfer Protocol or HTTP is the underlying protocol used by the World Wide Web and this protocol defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands. HyperText Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP) for secure communication over a computer network. In HTTPS, the communication protocol is encrypted by Secure Sockets Layer (SSL). The use of HyperText Transfer Protocol is very insecure. Without the use of a VPN, or a virtual private network, that encrypts data that is tunneled through it everything that is sent or received by either the client or the server is in plaintext and can be picked up, read and even tampered by anyone who is tapped into the ethernet wire or has a wifi card that is capable of monitor mode. This is why HyperText Transfer Protocol Secure was created, in order to help keep confidentiality between client and server, and the outside world. Even the mixed use of HTTP and HTTPS is very insure, allowing a man-in-the-middle to intercept HTTP network traffic and change all of the HTTPS link requests that come along to HTTP, and thus prevent secure, confidential communication.

The remainder of this paper proceeds as follows. Section 2 overviews our sample paper. Section 3 describes the design of our sample paper. Section 4 evaluates our solution. Section 5 discusses additional topics. Section 6 describes related work. Section 7 concludes.

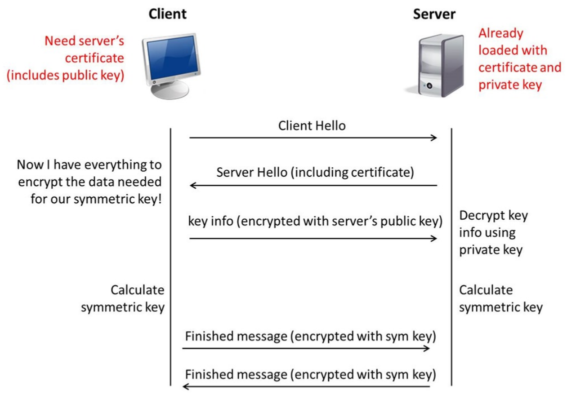
Client
Server
Need server's certificate (includes public key)
Already loaded with certificate and private key
Client Hello
Now I have everything to encrypt the data needed for our symmetric key!
Server Hello (including certificate)
key info (encrypted with server's public key)
Decrypt key info using private key
Calculate symmetric key
Calculate symmetric key
Finished message (encrypted with sym key)
Finished message (encrypted with sym key)

Figure 1: SSL handshake diagram

2 Overview

My research was split up into two different experiments. The first experiment was focused around research question one and permissions. The second experiment focused on research questions two, three, and four. Since these three research questions all involved Secure Sockets Layer and certificate verification. The third and final experiment was focused on research question five. These experiments were carried out using a combination of bash scripts and python scripts. The bash scripts were used for tree traversal, running python scripts on a current directory and finding specific lines within smali files in the files of dissembled applications. APKtool was used to dissemble the 100 applications. The scripts that were used to find, or "grep", for specific text from a file in the disassembled applications files counted the number of applications that had an occurrence of the searched text as well the frequency it occurred within each disassembled application. The python scripts were for the use of automating Androguard.

3 Design

For the first experiment, in order to find instances of developer-aided application termination, after having written a bash script to disassemble all of the android apk files using APKtool, I wrote a bash script that grepped for all instances of the finish() method. With that not working, a python script was written with androguard in order to try to again more information.

In the second experiment, in order to find occurrences of broken SSL socket factories I wrote a bash script that looked for a list of known broken SSL socket factories:

AccpetAllSSLSocketF

AllTrustingSSLSocketF

AllTrustingSSLSocketF

AllSSLSocketF

DummySSLSocketF

EasySSLSocketF

FakeSSLSocketF

InsecureSSLSocketF

NonValidatingSSLSocketF

NaiveSslSocketF

SimpleSSLSocketF

SSLSocketFUntrustedCert

SSLUntrustedSocketF

TrustAllSSLSocketF

TrustEveryoneSocketF

NaiveTrustManagerF

LazySSLSocketF

UnsecureTrustManagerF

All of these SSL socket factory classes are known to allow all certificates regardless of credentials. This list was obtained from [1], a very similar paper using androguard to look through dissembled android applications for signs of SSL misuse. While this experiment could have used androguard to look for implementations of broken SSL socket factories from this list I found that simply "grepping" for all occurrences also yielded satisfactory results. This approach also allowed me to could duplicate occurrences within a single application. For the next part of this experiment the setup was practically identical. Using the same set of disassembled applications, I wrote and ran a bash script that "grepped," or searched for occurrences broken trust managers that are known to allow verify all certificates regardless of whether or not the server's credentials matched. The following list was used:

AcceptAllTrustM

AllTrustM

DummyTrustM

EasyX509TrustM

FakeTrustM

FakeX509TrustM

FullX509TrustM

NaiveTrustM

NonValidatingTrustM

NullTrustM OpenTrustM

PermissiveX509TrustM

SimpleTrustM

SimpleX509TrustM

TrivialTrustM

TrustAllManager

TrustAllTrustM

TrustAnyCertTrustM

Unsafex509TrustM

VoidTrustM

Just as with the list of known broken SSL socket factory classes, this list of known broken trust managers was obtained from [1].

The last thing that was tested for in experiment two is the use of a hostname verifier that accepted all hostnames. This was performed through a bash script that searched, or "grepped", for all instances of one of the following known broken hostname verifiers:

AllowAllHostnames

Allow_All_Hostnames

The third experiment looked for applications that used HTTP. For the first part of the experiment, a bash script was used to search, or "grep", for applications that used HTTP. It made three different counts: how many applications used only HTTP, how many applications used a mix of HTTP and HTTPS, and how many applications only used HTTPS. Applications that use only HTTP and applications that only used HTTPS were not added selected for additional testing. Applications that used both HTTP and HTTPS were added to a text file for additional testing.

4 Evaluation

Before conducting any of the three experiments I defined what a vulnerable application is. In the context of this paper, a vulnerable application is any application that was deemed vulnerable by experiments two or three. In experiment two, applications were deemed vulnerable if they contained any use of broken SSL socket factories from the list in [1], if they contained any use of the broken trust managers from the list in [1], or if they contain any instance of a hostname verifier that allowed all hostnames. In experiment three, applications that used only HTTP were marked as vulnerable, and applications that used a mix of HTTP and HTTPS were added to a text document and subject to more testing.

Out of the set application sets of 50, 100 applications total, one of the applications in the Internet of Things application set would not disassemble properly. The tool used was APKtool. The operating system used was macOS High Sierra version 10.13.4 using bash version 3.2.57.

Answering the first research question which was tested in the first experiment proved harder than expected. Finding out if an application terminates if not all of the requested permissions were accepted isn't trivial. The only function call that terminates an activity is finish(). In just the random application set of 50 there were over 15,000 finish() function calls. So in the case that applications terminate if not all of the requested permissions are accepted is coded into a developer-made functions that are not universal. In this case there is not a trivial way to search for these functions given that there are 15,000 finish() function calls within just 50 applications. So for the scope of this paper I found that it was not feasible to completely answer this research questions and finish this experiment. This is an area for future work.

The research questions of experiment two proved to be much easier to answer and the experiment as a whole was more straight forward. 11 out of 50 applications in the random application set used broken trust managers. 6 out of 50 applications in the random application set used broken SSL socket factories. I found this to be surprising. I expected the numbers to be quiet higher. One counter to this finding is that there might exist other broken SSL socket factories and trust managers. 4 applications that were marked vulnerable were marked vulnerable by both searches. 11 out of the 50 applications used API that allowed all hostnames to be accepted.

In the Internet of Things application set, 5 out of 49 applications were marked vulnerable from using broken trust managers. 6 out of the 49 applications were marked vulnerable from using broken SSL socket factories. 3 applications that were marked vulnerable were marked vulnerable by both searches. Again, just was with the random application set, I assumed that the number of vulnerable applications in the Internet of Things application set was going to be much higher, closer to one third of the application set. 11 out of the 50 applications used API that allowed all hostnames to be accepted.

Answering the fifth research questions, using the third experiment, was also much easier than the first experiment proved to be. In the random application set, only 5 out of the 50 applications used only HTTP, these applications were vulnerable. 0 out of the 50 applications used only HTTPS. This number was much lower than I wanted it to be. I was hoping that more developers would have tried to use only HTTPS links. 45 out of the 50 applications in the random application set used a mix of HTTP and HTTPS. The use of HTTPS and HTTP does not automatically mean that the application is vulnerable.

Out of the Internet of Things application set, only 1 application out of the 49 in the application set used only HTTP. Given that this application set has only Internet of Things applications were confidential connections are far important, it was surprising, in a positive aspect, that the number of applications was so low. Following in this positivity, 15 applications out of the set of 49 applications were found to use only HTTPS. Following the previous part of the experiment on the random application set, it was exciting to see this number to be so high given that the other application set has 0 applications out of the 50 that use only HTTPS. 36 applications out of the set of 49 use a mix of HTTP and HTTPS. Just as before, using both mix of HTTP and HTTPS does not mean that the application is automatically marked as vulnerable.

5 Discussion

As stated before, many of the finds turned out to be much lower than hypothesized. Whether these hypothesis were had a cynical bias or not, were surprising. The number of applications that used broken SSL socket factories and broken trust managers was low.

Most of the interesting finds came from the androguard analysis of the applications that used a mix of HTTP and HTTPS:

the twitter API used HTTP

Google App content always used HTTPS

Google API used HTTPS

the Google Play store used HTTPS

Google docs used HTTPS

most images used HTTP

Facebook images used HTTPS

6 Related Work

This paper was largely a response to [1]. Titled Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security, this paper was combined effort between North Carolina State and Philipps University of Marburg in Germany. The authors developed their own tool, termed Mallodroid, based off of androguard to analysize android applications for SSL misuse. This paper was referenced a few within this paper, using the same list of broken trust managers and SSL socket factories.

7 Conclusion

While this paper and its work is not exhaustive by any means, it shows that on a very high level that applications tend to be developed with vulnerable tendencies. Much more in-depth research might find the number of vulnerable applications to be higher. Most of the comes down to poor development choices.

References