An Analysis of Network Securities in Mobile Applications

Brandon Wang

***Abstract*—Many people fall victim to online fraud and identity theft because Man in the Middle (MITM) attackers can intercept data transferred between a client and a server. The main objective of this research project is to develop a universal and robust method of analyzing the network security of mobile applications. This method, the Middle Man Defender (MMD), will defend against the MITM attackers.**

The MMD mimics a MITM attacker. It accesses the users data sent across a network. Then, the MMD will analyze the data to assess an applications security. Unlike a real malicious MITM attacker, the MMD will alert the user about the insecure application.

Multiple applications have been tested with this method. Some applications do connect to the MMD and allow itto catch the sensitive data. The most insecure applications send sensitive data in plain text without any protection. Some other applications send sensitive data in anencrypted form. Only the most secure applications request for the servers certificate and will only connect if the certificate is authentic. Therefore, if connected to the MMD, those applications will refuse to transmit any data at all.

Based on the results, most popular English apps and most banking apps do verify certificates and, hence, successfully ensure the network connection is secure. However, many other apps, including most popular Chinese shopping apps, do not ensure the connections security and expose sensitive data.

The product of this research introduces a dynamic method of detecting insecure applications.

I. Introduction

Background

Online mobile app stores often allow many developers to upload apps without thoroughly checking if the app protects its clients from network attackers. In addition, it is very convenient for phone users to download these potentially harmful and insecure apps because these mobile app stores provide a multitude of free apps. Of Google Plays 1.6 million apps, about 80% of them are free. Also, all of Baidus App Stores apps are free. As a result, hasty and inexperienced mobile users may fall victim to app developers or attackers.

Most of the insecurities that make the client vulnerable to the app developers or attackers usually have security flaws in the app source code. The app can perform the malicious actions in the background such as sending sensitive information over a network without user approval or encryption, allowing app developers to take advantage of the users. Also, the root causes of these problems can be hard to fix. Developers may have obfuscated the code, rendering the source code unreadable. They could also encrypt the entire code except for the decryption key, which only the app can use. So, even after decompiling the code, the insecurity is incurable. Some insecurities include not verifying certificates or not encrypting information over a network. Since online transactions occur after an application verifies the certificate of the server it is communicating with, if attackers perform the man in the middle attack, they can gain access to the clients personal information such as phone numbers, account name and passwords. Secure apps do verify certificates and thus they will fail the verification of the attackers certificate and will prevent any transactions.

To identify these insecurities, analyzers can mimic a malicious attacker, catch the network traffic and search for any sensitive information. And to prevent any potential attacks, clients can stop using the app or connect to a secure network.

This research is scientifically important because it designs a universally dynamic way of detecting insecure mobile applications. This dynamic method can be used to identify insecure connections on different devices such as iPhones or computers. With this new method, mobile users will not only become aware of network insecurity within apps, but also be protected from those attackers. 5 Also, during this technological era, many attackers can and have easily taken advantage of many unwary clients. This research can further extend the security measures against these attacks.

Pertinent Literature

The Public Key Infrastructure (PKI), and certificates both safeguard and authenticate data passed over a network. The PKI requires two keys - the public and private key of a website or application. The public key is distributed while the private key is held secret. The public key can only encrypt messages and the private key can only decrypt the messages that the corresponding public key encrypted. As a result, anyone can send a message to a website knowing that no one except the intended website can decrypt the message. Certificates are used to distribute the public keys and to let the client know they are dealing with the correct website. Certificates are given to websites or applications by a Certificate Authority (CA). If a CA is trusted, then a certificate distributed by the CA and the certificate owner is also trusted. Some clients check the authenticity of a certificate; if the check fails, they disconnect. Other clients, such as the applications being analyzed in this research, are not secure, do not check the authenticity of a certificate, and therefore, allowing attackers to pretend to be a targeted website. Then, the client uses the websites public key to encrypt a random symmetric key that the website will decrypt with their private key. Symmetric keys can encrypt and decrypt information. Hence, using this symmetric key, the

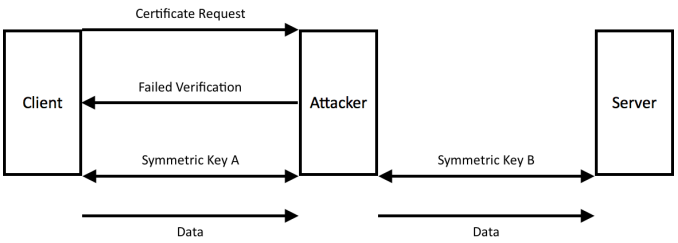

Fig. 1. Flowchart of a Man in the Middle Attack

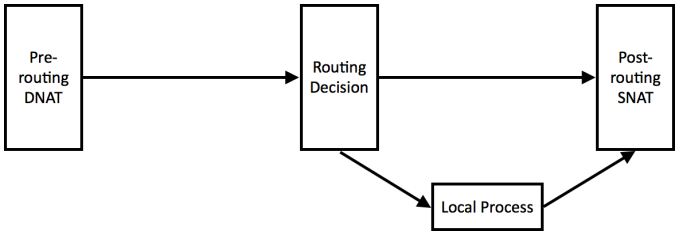

Fig. 2. Flow chart of the iptables rerouting

client and the website can continue their transaction.

In the Man in the Middle attack, the attacker gets between the client and the server of the website or application. To get between the two parties, the attacker must act as the server to the client, and as the client to the server. The attacker can then sniff out the data from the transactions between the client and server. As usual, the client will request a certificate from the web server and the attacker will emulate. However, the certificate the client receives will not be authentic. If the client is secure, it will prevent any further transactions. Otherwise, it will continue. Then the client will send an encrypted symmetric key to the attacker whom will forward it to the server. All future transactions between the client and the server will be intercepted and forwarded by the attacker. A passive attacker only eavesdrops on the information while an active attacker will alter the data maliciously.

Certificates secure the connection between the client and the server and also prevent Man in the Middle Attacks. When the client requests for a certificate, the attacker will forward that request to the server. However, when the server responds with its certificate, the attacker must replace that certificate with his own and include his own public key. He must supply the client with his public key if the attacker wishes to see the data transmitted by the client. If the attacker simply forwarded the certificate, the client will use the servers public key. As a result, the attacker cannot read the data because he does not have the servers private key. When the client receives the attackers certificate, he will fail to verify it because the certificate is not signed by a trusted CA. If the attacker does have a certificate signed by a CA, the client will see that the certificate does not correspond to the server.

Flask is a micro-framework for Python Web development and will be used to create the Man in the Middle attacker. This Flask server will act as an attacker by accepting any URL and treat it as its own. But to ensure the client does not crash, the Flask server will return the HTML data of that URL, just like the way a Man in the Middle attacker forwards information between the two parties. In the background, it must intercept and parse the HTTP requests, specifically GET and POST requests. The GET requests are stored as an Immutable Dictionary in the args attribute of the Request Object while the POST requests are stored as an Immutable Dictionary in the form attribute of the same object. Unfortunately, this server will crash if one request is invalid since it will keep trying to complete that request, holding up the future requests. So to start multiple processes of this server, Flask must be implemented with uWSGI. uWSGI is a protocol used to communicate with web servers. More importantly, it is used for load balancing and taking advantage of features HTTP cannot provide. With uWSGI, the Flask server can be run with multiple processes, and if one process is currently trying to completely an invalid request, the other process will handle future requests.

Also, the web traffic must be rerouted to this Flask server by using iptables. Within the vast majority of possible commands of the iptables, the NAT, Network Address Translation, section will be used to alter traffic flow. Performing NAT will alter either the source or destination address of a network packet and traffic. These are called SNAT and DNAT respectively. SNAT is performed during post-routing, right before the packets leave the client. DNAT is performed during pre-routing, right after the packets arrive. The objective of the attacker if the receive packets from the client and the server, so they must alter the destination address of the packets. In this research, iptables will be used to alter the destination ports of the HTTP requests by using DNAT

In addition, the Flask server must interact with the client application and do to so, both the server and the application must use XML-RPC. XML-RPC is a Remote Procedure Call method that uses XML passed through HTTP as a transport, and this enables another program or application, the XML-RPC client, to access methods from another program running a XML-RPC server. As long as the XML-RPC server is on, the XML-RPC can use the servers IP Address and port to connect. The client sends the data to the server, which will then pass that data through the procedure the client has called. This is a simple yet versatile way for the client application to communicate with the Flask server.

Purpose and Engineering Goals

The primary purpose of this project is to create a defense to protect against online attackers by detecting insecure mobile applications. The Middle Man Defender will intercept all network traffic, but will not interfere with the transactions or alter any requests. Such transactions include registration, login and during a purchase. The data caught by the MMD are recorded. In addition, if this program detects any insecurities from the request sent from a client, it will communicate with an app installed on the client device to alert the user.

Expected Outcomes

Secure applications will check if it is connecting to the

correct server by validating the servers certificate. If there is an attacker interrupting the connection, these applications will prohibit further transactions. Some applications will send sensitive data over the network but through HTTPS. To capture these requests, the program must also run on the HTTPS port. Some applications will also send sensitive data over the network through HTTP but will encrypt that data as well. The program can catch it, but the analyzer cannot read it. Many applications will send somewhat sensitive data through HTTP such as location, android model, android system and network carrier. These data are unique to the user, but will not make the user vulnerable to attackers. There are some apps that will use HTTP to send sensitive data such as username and password. These apps are the most insecure. The program should be able to catch all network traffic from any client, Android and iPhone, connected to the hotspot. It should successfully print out the GET and POST requests of any network transactions. Other transactions such as HEAD or PUT should also be able to go through the program but there will be no data from those requests to print.

Manual Identification

To identify insecure apps manually, I need to perform a Man in the Middle attack to sniff out the traffic sent by the app, then decide whether the traffic is secure. First, I installed the app onan Android device and connected that device to my computer. Then, I used Man in the Middle proxy to see the app's traffic. Finally, I parsed the traffic to see if I could extract any information.

Automative Identification

The Middle Man Defender (MMD) will be written in python. Using its subprocess module, one program, the Android emulator controller, will be able to call bash commands to control the installed Android emulator and manipulate android packages. Another program will open a Flask server on the local machine. Using this Flask server, the program can extract all the HTTP requests that passes through the local machine. This Flask server will mimic the previously used Man in the Middle proxy. The program will parse all of the URLs from the Flask server. If these URLs give unencoded information, the app will be marked as insecure.

II. Crawling

Crawling the Apps

The first step to this analysis is to download the apps from some app stores, specifically the Baidu app store and Google Play, with programs called crawlers. The Baidu app store easily allows any client to download an APK file. The Google Play store, however, uses various methods to prevent anyone from crawling. Clients must have Google accounts. These accounts cannot be automatically created as they require CAPTCHAs. Secondly, Google Play search results only list the first 500 apps. Finally, Google Play will disable an account if that account has too much download activity. To solve the first problem, I have personally created several Google Play accounts. Harvesting a few hundred is ideal, but I plan to run the crawler for some time. To avoid the second mechanism, instead of entering random search words, I start with the first few apps featured on the main page and continue adding apps by looking at their related apps. Finally, I programmed the crawlers to remain idle for five minutes after every download to avoid the third obstacle.

Architecture of the Crawler

The crawler program I wrote was written in a simple yet power language, Python. Firstly, a single program, the parser, would download the app store page and extract the URLs of the apps. Using SQL Alchemy, the program will store those URLs into a data table in a database. Secondly, a multitude of programs, the crawlers, will extract these URLs from the database. They will download that webpage and extract all the related apps on those pages and will insert these URLs into the same table. If the URL already exists in the table, these crawlers will not add it in. At the same time, the crawler will locate the meta data of the app. Thirdly, they will download the original URL it extracted. Fourthly, they will find the SHA1 value of the app and move it into the corresponding folder on a server. Finally, the crawlers will add the meta data of the app into a downloaded table in the database.

Crawling Baidu

The apps on the Baidu app store can be downloaded on any device. This makes it particularly simple for programs to crawl. This app store has a few pages of apps easily accessible from the home page. The first part of the Baidu crawler was to extract all of those pages for app URLs. There were around ten pages per category, resulting in around one hundred twenty pages of apps and two thousand app URLs.

With these two thousand app URLs, the second part of the crawler was able to easily add related apps. Since the crawler does not delete the original URL from the data table because related apps may change, the crawler adds in another value for all URLs in the table. This value is an integer value of seconds passed since January 1st, 1970. The crawler will extract the URL with the lowest time value. Then when it is done, it will put it back in with an updated time value. This ensures the crawler gets to all the apps.

Another value the crawler will add to the URLs is a priority value, either one or zero. The crawler adds a one if the URL has not been downloaded, and a zero if it has. This ensures that the crawler will extract app URLs that have not been downloaded first. After downloading the app, the crawler will change the priority value of the URL it extracted to zero.

The code that extracts data from the webpages are unique for each app store since the HTML code is unique for each app store. If this code changes, the code of the crawler would need to change. The crawler uses that code to extract the meta data of the app. These pieces of data include app name, version and total downloads.

If the app is unable to be downloaded, the crawler adds the app into an error table. This table keeps a record of all errors. Another value in this table is the amount of errors occurred for that app. If that value exceeds three, the crawler will delete

the url from the data table. The crawler will not change the priority value, but will change in time value. It will then insert the app URL back into the data table.

Once downloaded, the crawler will identify the SHA1 value of the app. This value is a forty digit hexadecimal number that is unique for all documents. The crawler uses this value to determine the name of the android package file, the corresponding folder and the primary key of the app in the table. The name of the file will be the complete forty digits and will be moved to the folder with the same first two digits. A primary key in a database makes sure no two identical items exist at the same time. The primary key of the data table is the app url, while the primary key of the downloaded table is the SHA1 value.

However, implementing only one crawler process to download all the apps on the store takes a very long time. One solution is to implement multiple crawler processes. To do so, each crawler must be able to coordinate their downloads so their actions do not disrupt the actions of another crawler process.

The first step is the initial extraction of an app URL. Since there are multiple processes, it is highly likely that two processes may extract the same URL. To avoid this, when one crawler extracts a URL, it immediately deletes that URL from the data table. This prevents any other crawler from extracting that same URL. If two crawlers extract the URL simultaneously, one will throw an error that states the URL has been deleted and the crawler will be terminated. So, there must be an exception. In this exception, the crawler will just move on to the next app.

This app URL is now removed from the database. One problem is that another crawler may add in the same app as a related app. This is a problem because now the related app has a high priority although it is being downloaded already. To avoid this, the crawler must add the app into the downloaded table after deleting it from the data table. Other crawlers will first examine both tables before determining if they should add in a related app.

The new app in the downloaded table does not have the meta data. So, in order to add the new app, the crawler will make temporary placeholders. After getting the meta data, the crawler will update the app in the download table. During the download process, if an error occurs, the crawler will add that app back into the data table.

Crawling Google Play

Crawling Google Play is more complex. Apps on this store are only downloadable on Android phones. An API is needed for the crawler to surpass this security. Also, a crawler must use a Google account to download the apps. Finally, some apps are paid and the crawler should avoid downloading those apps but also avoid encountering it twice.

First, the crawler needs to find the app URLs. The main page of the Play store does not have many more links for new collections of apps. So, the crawlers ended up with less than eighty apps to being with.

Next, the crawler needed to implement a Google Play API. This API is able to search, download and login to the play store. The crawler needs real Google accounts and Android device IDs. Also, the Google account must be registered to that Android device. Each crawler is only allowed on account since Google blocks an account from downloading too much. So after each download, the crawler will wait five minutes before extracting the next URL.

The crawler will log into an account before proceeding to the download phase. The crawler uses the log in function of the API. This function associates the Google account with the Android Device. Then, the crawler will extract a URL and find the related apps and meta data as usual. The meta data includes name, version, total downloads, size, rating, developer and more. However, there are many different versions of the same app on the Google Play store. Since there are many versions of Android, keeping older versions allow users with older android operating systems to download those apps. Since the crawlers do not download the same app URL twice, the crawlers avoid downloading the same version of the same app twice.

Then the crawler will use the download function from the API to download the app. The download function retrieves the binary download links of the Google play app, allowing the crawler to download the app. It will create a temporary Android Package file. After identifying the SHA1 value, the crawler will move the Android Package to the corresponding folder. Then it will wait five minutes before proceeding

These crawlers implement the same functions as the Baidu crawlers in order to coordinate their processes with their peers. However, the amount of Google crawler processes will be less than Baidu crawler processes. Each Google crawler needed an account and an Android device. This limits the total Google crawler processes.

Results

After a few days of downloading the apps, the Baidu crawlers began to slow down. They downloaded around ten thousand apps but the related app inflow was beginning to slow down. Soon, many of the apps in the data table had a priority of zero. These crawlers slowly inched toward fifteen thousand downloaded apps and only discovered sixteen thousand. These crawlers could implement a better system for these downloaded apps. Instead of piling all the apps in one data table, the crawlers could delete the downloaded apps from the data base. At first, this was the system the crawlers used. But then, the downloaded apps were added back to the data base to extract updated related apps. However, this problem could be solved without adding those apps back into the data table. When the data table becomes empty, the crawlers could start going through the downloaded table, using those app URLs and begin finding related apps. Combining both new and downloaded apps into one table is very complex and may lead to problems such as downloading the same apps twice. Also, by separating the new and downloaded apps, the priority value in the data table can be omitted.

Over couple months time, the Baidu crawlers amounted a total of 91,000 apps. The Google Play crawlers reached a total of 23,000 during the same time period. Every week, the server resets and all processes are stopped. Forty new processes are started manually.

III. Identification

Dynamic Identification

Dynamic analysis, also known as real-time analysis, is when the analyzer catches all the traffic an app passes through the network by preforming a man in the middle attack. This is when the attacker, the analyzer, intercepts the clients request to the server. Ordinarily, the client will send a certificate request to the server. The server will then send their certificate. The client will verify this certificate. Then it will generate a random symmetric key, encrypt this key with the servers public key found in the certificate and send it to the server. Any data sent between the client and the server will be encrypted and decrypted with this symmetric key.

In a Man in the Middle Attack, the client sends a certificate request to the server. But the attacker intercepts this certificate and sends its own certificate. The client will be unable to verify this certificate. Secure clients will refuse connection while insecure ones will ignore. The attacker will send a certificate request to the server and receive a certificate from it. The client will send data encrypted with a symmetric key the attacker and it both have. The attacker will decrypt it and then encrypt it with a symmetric key the server and the attacker both have. Now the attacker can read and modify the data the client and server are sending.

Static Identification

Static analysis is when the analyzer reads through the Java source code of the App to find any coding flaws. First, the App is decompiled with Apktool. Then, using dex2jar, the apps Dalvik byte code is converted into Java byte code. Finally, Java Decompiler will convert the Java byte code to Java source code.

This analysis has a couple of disadvantages. Firstly, App developers may have obfuscated the source code completely, rendering the Java source code unreadable. Secondly, App developers may have encrypted the entire code except for a small decryption key included. While the application and use the key to decrypt the code, analyzers are unable to read the code except for the key.

Architecture

A client will connect to a hotspot on the computer. Using iptables, all traffic headed for port 80, the HTTP requests, will be redirected to port 8080 and all traffic headed for port 443, HTTPS requests, will be redirected to port 8443.

The Middle Man Defender (MMD) will initiate a Flask server using Python. The server will be started with two sockets: one on port 8080 with an HTTP protocol and the other on port 8443 with a HTTPS protocol. The HTTPS protocol requires a certificate and public key created by OpenSSL. The certificate and public key will be sent to the client if they request a certificate. Consequently, the client application will fail to validate them. It will alert the user and refuse to connect. Applications that do not request for a certificate will allow for data transmission. The MMD will catch all the HTTP and HTTPS requests of these apps. It will parse the GET and POST requests and search for any keywords. The client app needs to go through several transactions for the MMD to catch sufficient data. Such transactions include registration, login and entering personal data. Then the MMD must respond to the client with the server data. For example, when the client goes to the login screen, the MMD must respond with the login screen data from the server. To do so, the MMD creates a URL request object with the request data from the client. It then accesses that URL request. The server will respond with a URL response object that the MMD will return to the client.

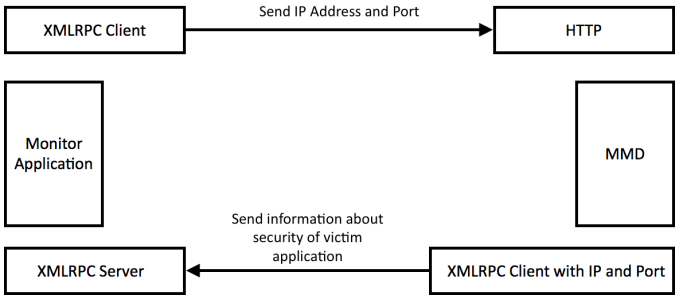

Fig. 3. Flowchart of the MMD

The communication between the monitor application and the MMD will depend on XML-RPC. The application will first become the XML-RPC client and connect to the XML-RPC server running on the MMD. The application will then send its IP address and port number to the MMD. Then, it will use that information to become the client and connect to a XML-RPC server running on the monitor application. In this architecture, the MMD can push data to the application by calling a process on the apps XML-RPC server.

IV. Results

Data Analysis Many popular shopping apps, such as JD, send sensitive client data in plain text through the insecure HTTP protocol. All of these applications are labeled insecure because they do not verify certificates and allow the client to be connected to the MMD. During registration, these apps tend to send the clients phone number and verification code in plain text. During login, these apps usually send the users username in plain text, while a few apps also send the users password in plain text. Most of the shopping apps require the user to enter in an address prior to purchasing a good. When updating the address information, the application may insecurely send the clients name, phone number and exact address.

Other applications, such as Taobao, send the client credentials in an encrypted form. While this data cannot be read by a human, malicious attackers may be able to reverse engineer the applications encryption algorithm and decrypt the data. In addition, some applications, such as cTrip, do not

leak sensitive information to the MMD. These applications may leak other non-sensitive information, such as the global coordinates of the user. However, both types of applications do not verify certificates, so they are labeled as somewhat insecure.

Finally, the most secure applications refuse to transmit data when connected to the MMD. They often throw a network error, explaining that the network is insecure or unstable. The client is forced to switch networks and these applications successfully secure client data.

Based on the results, many popular Chinese shopping applications are insecure. A majority of these shopping applications leaked client information during registration, login and the purchasing process. Some other Chinese apps, such as lifestyle, travel and social media apps, were secure, as they checked the servers certificate. In addition, most popular English applications, such as Facebook and Instagram, along with banking applications, such as ICBC and China Merchants Bank, were very secure.

V. Conclusion

This research is significant because it designs a dynamic and universal method of detecting network insecurities in mobile applications. Instead of statically analyzing an applications source code, this method analyzes the applications behavior as it runs. The Middle Man Defender (MMD) is not only more accurate, but also more efficient.

This method can be used to analyze the security of mobile applications, just it has been used in this research project. By determining if an application is secure or insecure, mobile users and determine where to use the application, on an open or private network.

In addition, this method can serve as a security measure against MiTM attacks. Although it mimics an attacker, it is a software so it will not use sensitive data it captured maliciously, unless if a user tinkers with the code.

Next Steps

There will be two major steps in the next phase of this project.

One of the next steps of this project includes securing the clients connection to the server, such that the client can safely use the MMD to browse the internet without having to switch networks. The MMD will use certain protocol and encryption to communicate with the server.

In addition, the MMD currently works on its own hotspot. In the next step, any traffic sent over any wifi network will be caught and analyzed with the MMD. To do so, this project will include additional functions similar to programs such as wireshark or tshark.