San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

2010

How Smart is your Android Smartphone?

Deepika Mulani
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation

Mulani, Deepika, "How Smart is your Android Smartphone?" (2010). *Master's Projects*. 68.
http://scholarworks.sjsu.edu/etd_projects/68

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

HOW SMART IS YOUR ANDROID SMARTPHONE?

A Project Report
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science

by
Deepika Mulani
May 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

HOW SMART IS YOUR ANDROID SMARTPHONE?

by
Deepika Mulani

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp	Department of Computer Science	Date
Dr. Jon Pearce	Department of Computer Science	Date
Dr. Chris Pollett	Department of Computer Science	Date

APPROVED FOR THE UNIVERSITY

Associate Dean	Office of Graduate Studies and Research	Date

ABSTRACT

HOW SMART IS YOUR ANDROID SMARTPHONE?

by Deepika Mulani

Smart phones are ubiquitous today. These phones generally have access to sensitive personal information and, consequently, they are a prime target for attackers. A virus or worm that spreads over the network to cell phone users could be particularly damaging.

Due to a rising demand for secure mobile phones, manufacturers have increased their emphasis on mobile security. In this project, we address some security issues relevant to the current Android smartphone framework. Specifically, we demonstrate an exploit that targets the Android telephony service. In addition, as a defense against the loss of personal information, we provide a means to encrypt data stored on the external media card. While smartphones remain vulnerable to a variety of security threats, this encryption provides an additional level of security.

ACKNOWLEDGEMENT

I would like to thank to Dr. Mark Stamp, my project advisor, for his guidance and support throughout my Masters' degree and project. I would like to thank him especially for his belief in me to work on this project. I would like to thank Dr. Chris Pollett and Dr. Jon Pearce for their guidance and suggestions while I was working on this project.

And special thanks to my dear husband Kamlesh for believing in me and being my pillar of support in all my endeavors. I thank him for being a constant source of encouragement to realize my potential and providing timely help and support in my work.

TABLE OF CONTENTS

1.0 Introduction ... 1
1.1 Objective of the Project ... 2
1.2 Order of the Project ... 2
2.0 Android Framework ... 3
2.1 Introduction ... 3
2.2 Application behavior ... 4
2.3 Application components ... 5
2.4 Application level security framework ... 6
2.5 Files and preferences ... 8
2.6 Android limitation ... 8
3.0 iPhone and Symbian Mobile Frameworks ... 9
3.1 iPhone security architecture ... 9
3.2 Symbian's security framework ... 10
4.0 Mobile Phone Risks ... 13
5.0 Best Practices ... 14
6.0 Experiment Details and Results ... 15
6.1 Exploiting telephony security ... 15
6.2 Tapping incoming call ... 17
6.3 Aborting outgoing voice call ... 21
6.4 Validation ... 21
7.0 Security Feature for Android ... 22
7.1 Encryption of files stored on SD card ... 23

7.2 Encryption limitations ... 26

7.3 Decryption algorithm ... 28

7.4 Validation ... 30

8.0 Conclusion ... 31

References ... 32

LIST OF FIGURES

Figure 1. Android architecture 4

Figure 2. Android applications component IPC 6

Figure 3. Symbian certification and signing process 12

Figure 4. XML Permissions for telephony exploitation 16

Figure 5. Install time notification of permissions 17

Figure 6. Code for receiving incoming voice call number and registering for notification 18

Figure 7. Message sent on new incoming call 19

Figure 8. Code for SMS sending 20

Figure 9. Incoming voice call notification 20

Figure 10. XML registering for outgoing call notification 21

Figure 11. Outgoing voice call aborted 22

Figure 12. Code for encryption 25

Figure 13. Encryption of file on external media 26

Figure 14. External media unavailable on encryption 27

Figure 15. Switching off USB storage 28

Figure 16. Decryption successful 29

Figure 17. Decryption failure 30

LIST OF TABLES

Table 1. Android manifest permission protection levels.. 7

1.0 Introduction

Mobile phones are no longer devices restricted to making voice calls—they can run most of the processes that one expects from a desktop computer. Mobile phones are equipped with applications such as e-mail clients, chat clients, short messaging service (SMS), and multimedia messaging service (MMS). Most smartphones are equipped with cameras so that one can have personal pictures and videos on the phone. Communication between two mobile devices is no longer limited to the services of a GSM provider. One can have two mobile phones communicate with the help of Bluetooth, external media cards, or the Internet. Thanks to the efforts of the World Wide Web Consortium (W3C) and Open Mobile Alliance (OMA), being away from one's laptop does not mean being disconnected from the rest of the Internet world.

The first mobile phone virus, Cabir, was created in 2004 and targeted for Symbian OS-based phones. This virus replicated itself on Bluetooth wireless networks [1]. Since then, there have been many similar versions of the virus and a few new ones. However, the number of mobile phone viruses is significantly fewer than computer viruses.

One major difference between PC and mobile phone viruses has been that it is more difficult for mobile virus infections to spread as fast as computer viruses can. This is due to the variety of mobile platforms; lack of documentation and lack of support tools has led to less exploitation of vulnerabilities [1].

However, the trend has been to synchronize computers with smartphones. Hence, the threat to all critical and private data has become twofold. Even worse is that smartphones come with a built-in billing system; a virus can cause immediate financial loss. Most of the threats interrupt user productivity, drain the battery, increase messaging charges, and have the potential to damage users' reputations.

With the rich variety of data centric applications available, it is important that the smartphone be made *smart* to preserve user privacy. Various platforms have evolved in the process to achieve this goal. Some of the current mobile platforms such as Symbian, Android, and so on, have taken steps to ensure that their architecture is build around security.

1.1 Objective of the Project

The objective of this project is to identify security holes and any missing security features in Android's architecture. Using this as a starting point, the goal is to develop a prototype application that serves as a justification for our findings.

1.2 Order of the Project

Section 2 begins with a discussion of the Android security architecture and some of its limitations. Section 3 briefly covers the security architecture of iPhone and Symbian, comparing them with Android's architecture. Section 4 lists some mobile phone risks, and Section 5 briefly covers some of the best practices that users of smartphones should follow for their personal security.

Section 6 discusses in detail the telephony exploitation that we have successfully deployed on Android phones. This malicious behavior tries to breach the user's privacy by retrieving all his or her contacts and sending an SMS message on each incoming voice call. It also prevents the user from dialing any numbers, hence exploiting the most basic feature of a phone.

Section 7 covers the implementation details and limitations of the security enhancement we have developed for dealing with external data. In the current Android framework, each application has private access of the data stored on the internal phone memory. However, this is

not applicable to data stored on an external media card. In our solution, we encrypt the data being written to external media and store the encryption key on the phone memory, which is private to the application that uses this solution.

Section 8 concludes our work and reiterates our goals and achievements while studying the security architecture of smartphone operating systems, and provides the scope for further study.

2.0 Android Framework

2.1 Introduction

Android is the mobile phone platform led by Google's Open Handset Allowance (OHA). Android has a unique security model in which the user is in complete control of the device. It is an open source platform based on Linux. All applications are written in Java and compiled into a custom byte-code (DEX) [6]. Each application executes in its own process with its own instance of the Dalvik virtual machine interpreter [2].

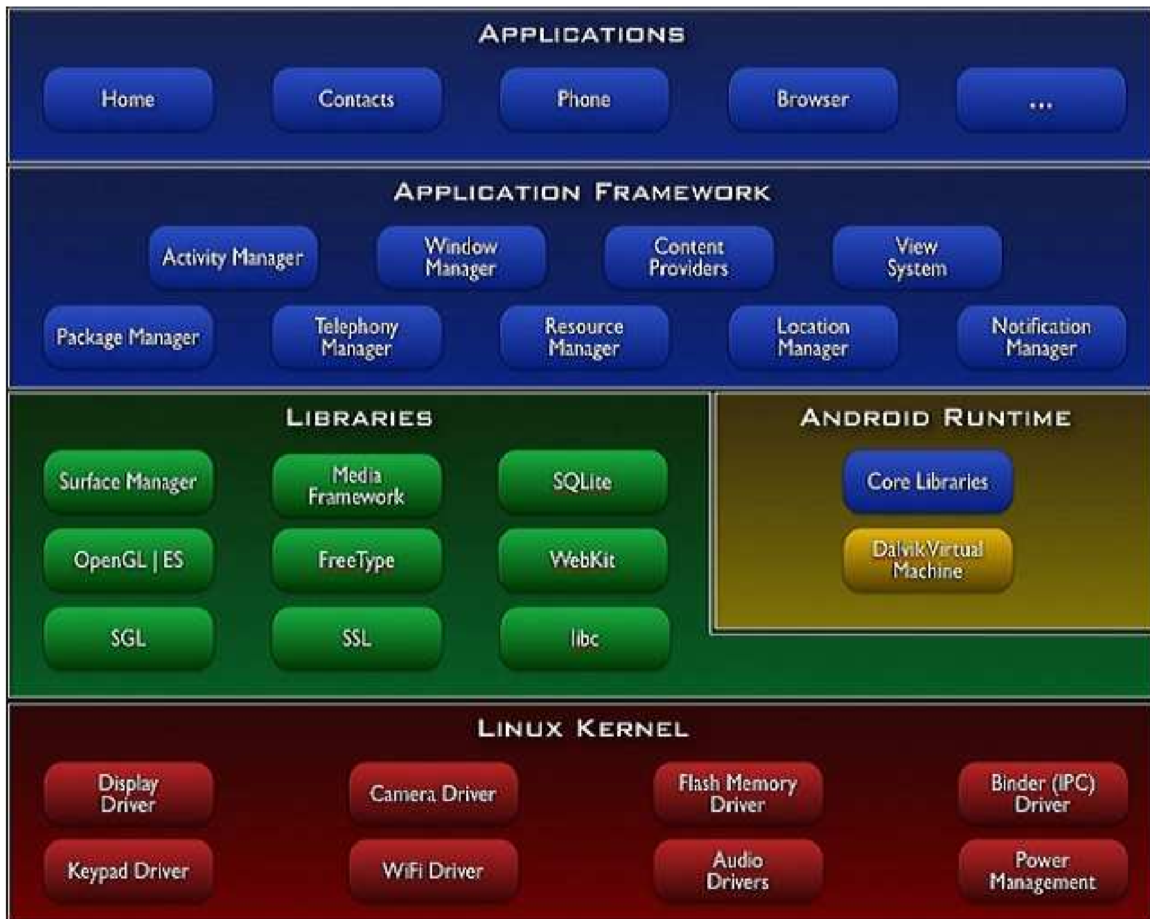

Figure 1. Android architecture [6]

2.2 Application behavior

Every application in Android runs as a separate process with a unique UID, unlike a desktop computer where all the applications run with the same UID. The UID of an application in Android protects its data. Programs cannot typically read or write each other's data, and sharing between applications must be done explicitly [3]. Due to this feature, a compromise such as a buffer overflow attack [3,17] is restricted to the application and its data. However, it is important to note that an application can launch another program that will run with the launching application's UID.

For a developer to run an application on the Android phone, his or her application needs to be signed. Developers can generate self-signed certificates and use this for code signing. Code signing is done to enable developers to update their own applications without creating complicated interfaces and permissions.

2.3 Application components

Applications are comprised of components. Components interact using *Intent* messages [6]. Recipient components assert their desire to receive Intent messages by defining *Intent filters* [6]. There are four types of components used to construct applications:

1. Activity components interact with the user via the touchscreen and keypad. Only one activity is active at a time, and processing is suspended for all other activities [5].
2. Service components provide for background processing when an application's activity leaves focus and another GUI application comes in the foreground [6].
3. Broadcast receiver components provide a general mechanism for asynchronous event notifications [6]. The receivers receive Intent messages that are implicitly addressed with action strings; for instance, dialing a number is associated with the action OUTGOING_CALL_ACTION.
4. Content provider components are the preferred method for sharing data between applications [5]. These APIs implement an SQL-like interface; however, the backend implementation is up to the application developer.

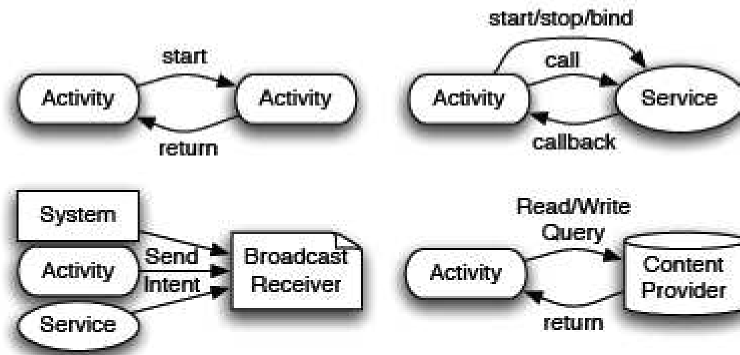

Figure 2. Android applications component IPC [3]

2.4 Application level security framework

Applications need approval to do things their user might object to, such as sending SMS messages, using the contacts database, or using the camera. To keep track of what the application is permitted to do, Android maintains manifest permissions that are enforced by the middleware reference monitor. The permission label is a unique text string that can be defined by the OS as well as by a third-party developer. These permissions indicate what resources and interfaces are available to the application at run-time. An example of a permission is READ_CONTACTS, which permits the application to read the user's address book. In addition to reading and writing data, permissions allow applications to access system services such as dialing a number without prompting the user or taking complete control of the screen and obscuring the status bar.

A developer should specify all permissions that his or her application requires in the AndroidManifest.xml file; however, it is not necessary that all permissions be granted. When the application is getting installed, the user has the choice to decide whether or not to trust the software based on the application's promised features. and the permissions required. These permissions are different from file permissions. Once an application is installed, its permissions

cannot be changed. The fewer permissions an application needs, the more comfortable the user should feel installing the application.

Permissions have a protection level. The four protection levels are outlined in Table 1.

Table 1. Android manifest permission protection levels [2]

Normal	Permissions for application features with minor consequences such as VIBRATE, which lets applications vibrate the device. Suitable for granting rights not generally of keen interest to users; users can review but may not be explicitly warned.
Dangerous	Permissions such as WRITE_SETTINGS or SEND_SMS are dangerous as they could be used to reconfigure the device or incur tolls. This level marks permissions in which the users will be interested or be potentially surprised. Android will warn users about the need for these permissions on install.
Signature	These permissions can be granted only to other applications signed with the same key as this program. This allows secure coordination without publishing a public interface.
SignatureOrSystem	Same as Signature except that programs on the system image also qualify for access. This allows programs on custom Android systems to also get the permission. This protection is to help integrate system builds and won't be typically used by developers.

The permission label policy is used to protect applications from each other and also various components within an application. In the mobile phone environment, it is difficult for the operating system to manage access control policies of hundreds of unknown applications. Therefore, Android simplifies this by having the developers define their permission labels to access their interfaces. By doing so, the developer does not need to know about existing and future applications; permission labels allow the developer to indirectly influence security decisions.

2.5 Files and preferences

Android uses UNIX-style file permissions [2]. Each application has its own area on the file system that it owns [2,16]. This is similar to programs having a home directory to go along with their User IDs. This feature is limited only to the internal phone memory and not the external memory. The standard way for applications to expose their private data to other applications is through content providers [16].

2.6 Android limitation

The current security policy of Android works on a static level only during installation to identify whether the application is permitted all the requested permissions from the user. Once the permission is granted, there is no way to govern to whom these rights are given or how they are later exercised [3]. Permissions are asserted as vague suggestions as to what kinds of protections the application desires. One must place good faith in the user and the OS to make good choices about permissions granted to the application which, in many cases, may not be the absolute best choice.

Due to the above architecture, Android system libraries have limited ability to control installed third-party applications that can be granted permissions to use their interfaces. This implies that there is no control to restrict an installed application based on its signatures. Further, it is not possible to define the desirable configurations of an installed third-party application such as the minimum version and the set of permissions it is allowed or disallowed.

This implies that Android applications built with the right set of permissions protect the system from malicious applications but provides severely limited infrastructure for applications to protect themselves.

3.0 iPhone and Symbian Mobile Frameworks

iPhone and Symbian are the two popular competitors of Android smartphones. Each of these platforms has its own security model. A comparison of these architectures is essential to understanding the current trends in mobile security.

3.1 iPhone security architecture

iPhone's security features include encryption of data in transit and authorization by strong passwords to corporate services. On iPhone 3GS, there is a new enhancement of hardware encryption of data stored on the device [12]. Users of the device can be restricted from accessing certain features by setting up device restrictions through configuration policies.

iPhone also comes with the feature of remote wipe, which is helpful in case of the device being lost or stolen. The user of iPhone can login to his or her web account and issue the remote command to securely wipe the the phone's data, making it unrecoverable. It also supports erasing of data from the device after a certain number of failed authorization attempts. To provide secure access to corporate data, iPhone also integrates with VPN technologies [12].

It is mandatory that all iPhone applications be signed. Third party applications are required to be signed by developers with an Apple-provided certificate. Runtime protection is also available, which ensures that an application has not become untrusted since the last time it was used. This is an important security feature that the Android platform lacks.

Apple's app store is a guarded community. The apps that get listed have been certified by Apple. The developers of the apps are required to be registered and pay annual subscription fees. The app and each of its versions is evaluated by the Apple team, and any app that can potentially pose a threat to personal data, contains inappropriate content, or breaks the law is rejected by Apple.

At Google's Android marketplace, the approach is fundamentally different: Any application can be uploaded to the marketplace, and Google does not evaluate these apps. What protects Android users from these apps is the concept of "capabilities" or "permissions" [8].

At installation, each app tells the Android OS what capabilities it requires. Based on the usage and claim of the applications, it is up to the user to decide if the capabilities are reasonable [8].

This system has the advantage of being enforced by a true platform. An application cannot exploit any other resources to which it is not entitled. However, the disadvantage of this trust system is that there is no way to be sure that an application will limit itself within the defined boundary once it is installed. Any application can request capabilities and appear legitimate on the surface, while in the background, it may be doing something malicious.

The bigger problem is that the marketplace relies on the user's ability to evaluate the risks of the application they want to download and run. Not all users are aware of security threats and ways that their information can be compromised.

3.2 Symbian's security framework

Symbian was one of the first smartphone operating systems, with its first phone based on Symbian v6.0, released in 2001. Nokia acquired Symbian Software Limited in 2008 and, in February 2010, Symbian source code became available as open source [14].

To free the users from the task of deciding about security of an application, Symbian OS released Symbian v9.x, which introduced the concept of platform security. This includes capabilities and Symbian signing. In early 2005, Symbian 9.1 was released and can run applications that pass the constraints set by both the Android and iPhone platforms. This means that, just as in Android, an application needs to enlist permissions or capabilities that its APIs

will use. This is exactly the UNIX-style permissions per process-based model; however, unlike Android, this application is not available for market use without the signing process.

There are two types of certificates used by the Symbian community: A developer certificate is used by the developer to sign his or her application and run on a specific phone. This developer certificate contains the requested capabilities of the application but is confined to run only on certain phones as specified by the International Mobile Equipment Identity (IMEI) which are mentioned in the certificate request process. It is not possible to request more than 20 IMEI numbers in a developer certificate. This means that a malicious application does not spread extensively as soon as it is developed.

The second certificate is the Symbian Signed Certificate, obtained through the Symbian signing process. Just like iPhone, a Symbian application needs to be signed by a certifying authority before deploying the application in the market. There is a cost associated with each signing. One needs a publisher ID from Verisign, which costs $200 per year, and the Symbian signing process has an additional cost of around $300 [9] per signature.

Once the application is ready to be deployed, it needs to be submitted to the Symbian signed site [15]. There, the application is tested against criteria specified by Symbian. After the testing is successfully completed, the application is certified and returned to the developer, who can then distribute the application. This process is shown in Figure 3.

Since signing has a cost associated with it, it is unlikely that virus writers will be submitting their apps for Symbian signing, in which case it is not possible to distribute malicious applications.

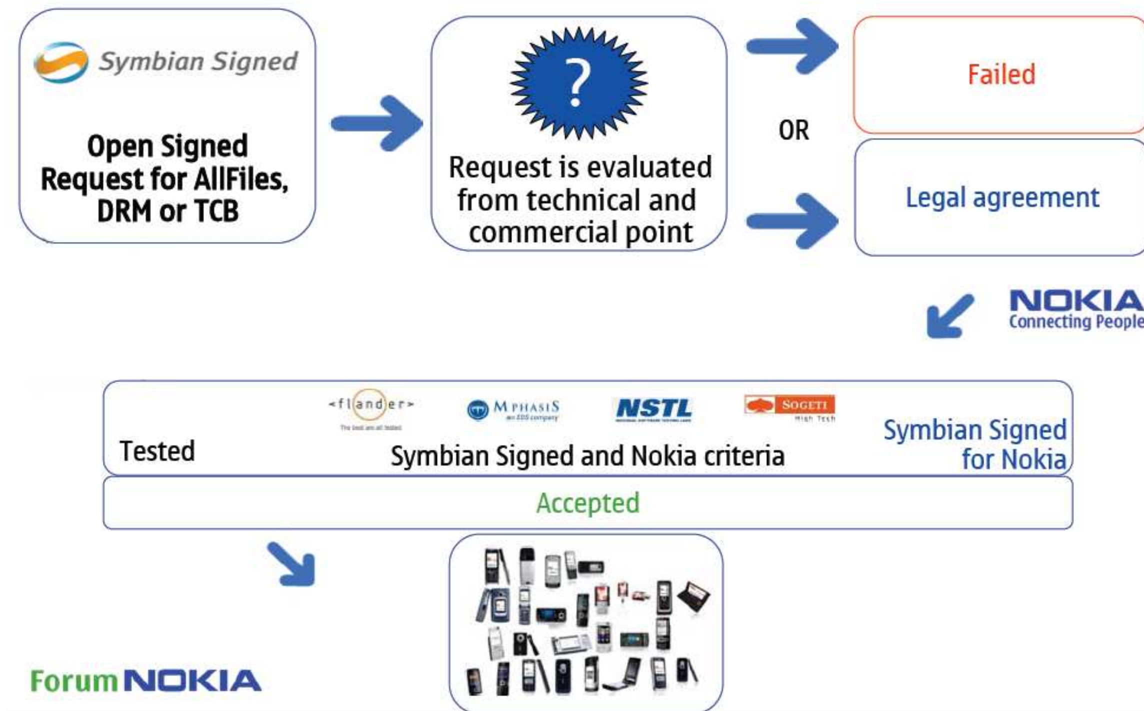

Figure 3. Symbian certification and signing process [10]

The Symbian signing process implies that the only likely way to hack a Symbian phone is by disabling its platform security feature. Once this is done, an unsigned application can be installed. Once platform security is disabled, the phone is at risk, allowing access to system files, changing how the operating system works, and access to a wide variety of viruses, malware, and so on.

Therefore, the best a user of a Symbian smartphone can do is never to disable platform security. Developers must not install applications signed with a developer certificate that can internally disable platform security. It is not possible for attackers to do so as a developer certificate can be obtained for a maximum 20 IMEI numbers and it is not possible to guess IMEI number of any phone.

Theoretically, Symbian's platform security concept is more powerful than the security model employed by current Android and iPhone smartphones because it is a fusion of the best features of these two platforms' security architecture.

With the advent of smartphones and the variety of information these can hold, it is necessary that the users of such phone become educated about the various intricacies and security risks involved in their use.

4.0 Mobile Phone Risks

The more popular an operating system, the more likely it is to be infected by a virus—just as in the PC world. Microsoft Windows is more often subject to malicious attacks than is Apple Macintosh. Similarly, in the mobile world, Android, iPhone, and Symbian-based phones are the most popular targets for attack. Several risks factors indicate that most of the mobile market is ripe for powerful new attacks. Some of these are:

1. Proof of concept viruses and variations of these have been published on the Internet (e.g., Cabir virus provided a code base for VLASCO.A and DAMPIG.A) [7].
2. Most phones do not have security software or security policies [7].
3. Smartphones are capable of high-speed data transmission [7].

Closed devices are less likely to be infected [7]. "Closed" means that the devices just make calls and have a simple address book but do not have the ability to install third-party applications.

Web browsing increases the possibility of infection. Devices that enable third-party application installation without OS restriction are at high risk for infection. The cost of a lost device is small in comparison to the loss caused by compromised sensitive data. Such loss can cause diminished customer confidence, financial loss, and brand damage.

Infection can occur or spread through any of the following:

1. Inserting infected external media card.
2. Synchronizing with a PC that installs an infected file [7].
3. Web downloads [7].
4. Bluetooth communication etc.

5.0 Best Practices

Most mobile phones operate on a corporate network but are disconnected for a long period of time and often change locations. Therefore, it is essential that corporate companies take care to prevent loss of personal sensitive information while reaping the benefits of increased productivity.

The following practices will help users take advantage of available smartphones while minimizing the security risks [7].

1. If a mobile phone is lost or stolen, the service provider or phone manufacturer should provide a device management feature to "wipe" all data.
2. Devices accessing corporate IT resources should access remote information over VPNs for secure access.
3. Only authorized applications should be provided access to the network. Authorization may be based on a user-prompted password before access begins. Authorization should not be limited to the user's acceptance at installation time.
4. Sensitive data in transit should always be encrypted. In addition, sensitive data on the phone—such as calendar entries, phonebook contacts, product prices, customer orders, and so on—should be encrypteds.

5. Smartphones should be shipped with antivirus software installed. This software should have a feature that scans for third-party downloaded apps that keep running for a long time and consistently access sensitive and private information. The virus scan software should stop such applications and inform the user of such activity, giving him or her the option of uninstalling it. Notable smartphone antivirus software includes McAfee for Microsoft Windows Mobile devices, and Symantec Mobile security for Symbian.
6. Risk assessment should be done before a business adopts the latest trend of mobility through smartphones.
7. User education and training is essential to make users aware of the risks and liabilities involved when using smartphone applications.

6.0 Experiment Details and Results

We have been successful in demonstrating Android's security limitations by exploiting Android's telephony security. In addition, as mentioned in Section 2.5, the lack of security when dealing with data stored externally has been considered, and a defense has been developed to provide privacy to applications when handling external storage media.

6.1 Exploiting telephony security

We have implemented an Android activity and a receiver application which exploits the current telephony security of the Android phone. This application intercepts any incoming call and sends SMS to all the phonebook contacts with details of the call. Further, the application receives a broadcast of an outgoing call and aborts this call.

The permissions required by our application are mentioned in the Androidmanifest.xml, as shown in Figure 4.

```
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.MODIFY_PHONE_STATE"
/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"
/>
<uses-permission
android:name="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
android:name="android.permission.READ_CONTACTS"></uses-permission>
<uses-permission android:name="android.permission.SEND_SMS"></uses-
permission>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-
permission>
```

Figure 4. XML Permissions for telephony exploitation

When the application is installed on the phone, the user is notified of these permissions, as shown in Figure 5. It is up to the user whether he or she wants to install the application or not. It is very easy to deceive a novice user by promising a false set of features. This will often cause the user accept all the enlisted permissions that the malicious application uses.

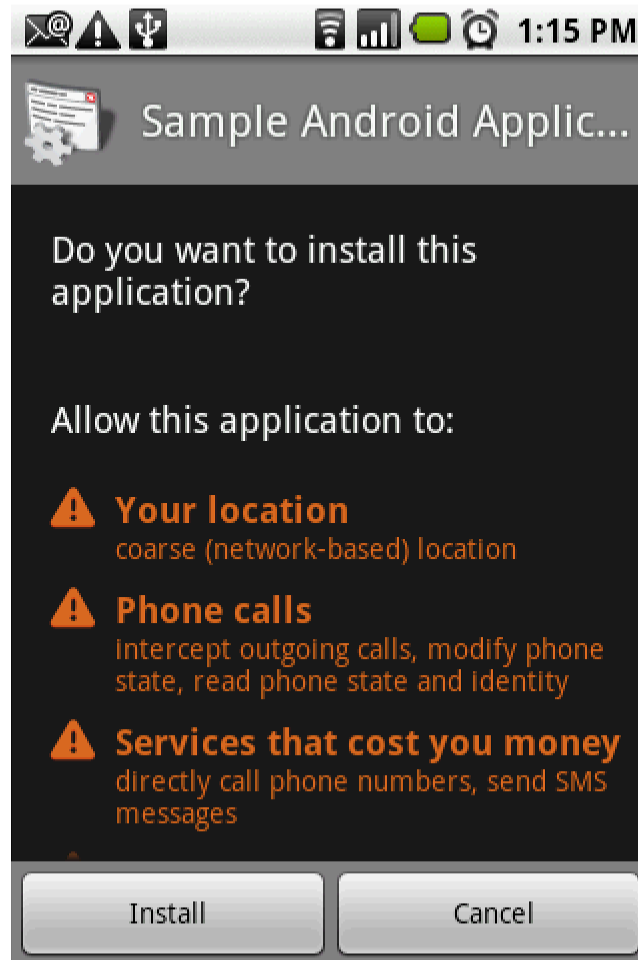

Figure 5. Install time notification of permissions

6.2 Tapping incoming call

The first malicious attack of the application is that it intercepts any incoming call (the call might be accepted or it might be a missed call). On intercepting the call, the application obtains the details of the incoming call's number. To make this a breach of privacy, upon getting this notification, the application fetches all phonebook contacts and sends SMS to all these contacts. This message is sent silently in the background without the knowledge of the mobile phone user.

To send SMS on an incoming voice call notification, we need to override the method "onCallStateChanged" of Android's "PhoneListener" class. Figure 6 depicts our implementation for this, along with the registering for notifications of change in call state.

```
// TelephonyManager
final TelephonyManager telMgr = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);

this.telMgrOutput.setText(telMgr.toString());
previousState = telMgr.getCallState();

// PhoneStateListener
PhoneStateListener phoneStateListener = new PhoneStateListener() {

@Override
public void onCallStateChanged(final int state, final String
incomingNumber)
{
            telMgrOutput.setText(getTelephonyOverview(telMgr,incoming
            Number));

            if(state == TelephonyManager.CALL_STATE_IDLE &&
      (previousState == TelephonyManager.CALL_STATE_RINGING ||
      previousState == TelephonyManager.CALL_STATE_OFFHOOK))
            {
                  sendSMS(incomingNumber);
            }
            previousState = state;
      }
};

// Registering to receive incoming call notifications
telMgr.listen(phoneStateListener,
PhoneStateListener.LISTEN_CALL_STATE);
```

Figure 6. Code for receiving incoming voice call number and registering for notification

This is followed by fetching the phonebook contacts. There is a CONTACTS content provider in the framework that contains the table People. From this table, we fetch the phone numbers of all the contacts.

Whenever we receive an incoming voice call notification, we send SMS to all the contacts. Our SMS message includes the phone number of the calling party. Figure 7 indicates

the form of the SMS message being sent by our application. To send the SMS, one needs to use the “SMSManager” class of Android. This class provides a method “sendTextMessage” in which we specify the text message and the receiver’s phone number. A code sample is shown in Figure 8. In our demo application, we have a GUI display the current call state and any incoming call number. This is shown in Figure 9.

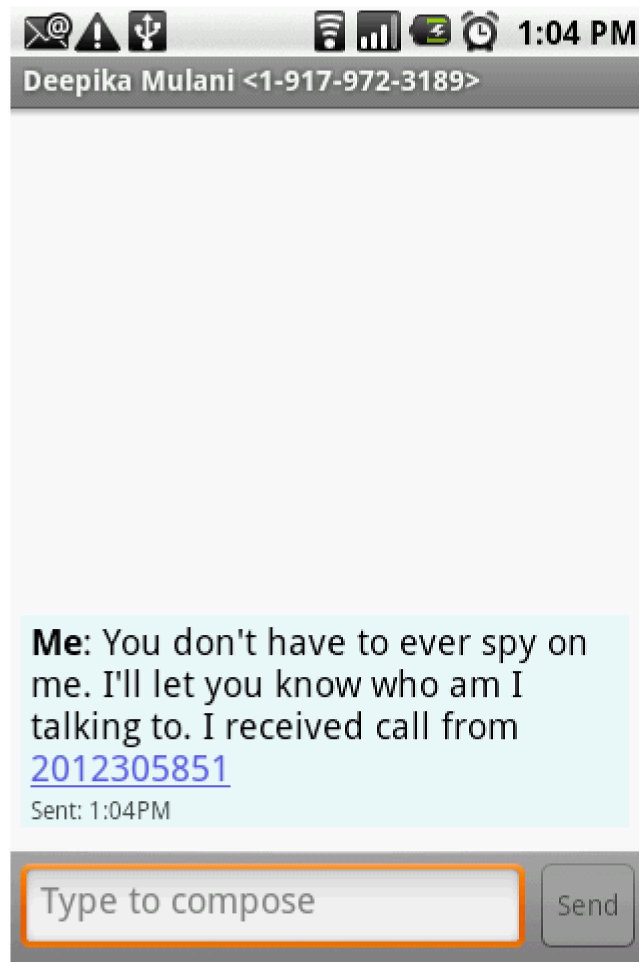

Figure 7. Message sent on new incoming call

```
PendingIntent sentPI = PendingIntent.getBroadcast(this, 0, new
Intent("SMS_SENT"), 0);
SmsManager sms = SmsManager.getDefault();
//sending message to all phone contacts
for(int i = 0; i < phoneNo.size(); i++) {
      sms.sendTextMessage(phoneNo.get(i), null, "You don't have to
ever spy on me. I'll let you know who am I talking to. I received
call from "+ number, sentPI, null);
        }
```

Figure 8. Code for SMS sending

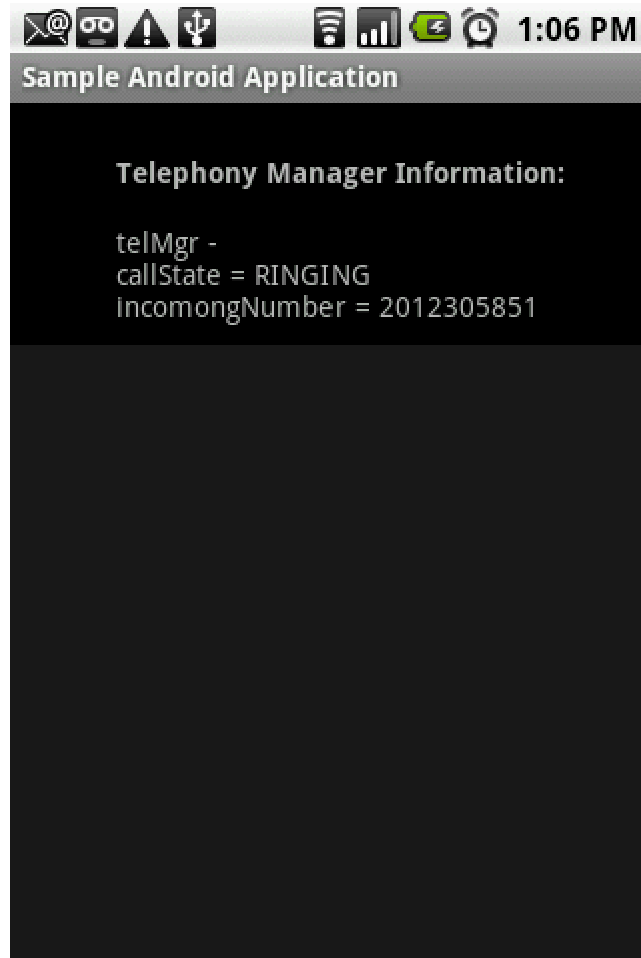

Figure 9. Incoming voice call notification

6.3 Aborting outgoing voice call

A second malicious attack involves aborting any outgoing call. To get notification of outgoing calls, we registered a class in the application as a broadcast receiver in AndroidManifest.xml. This is depicted in Figure 10. This class is derived from the class “BroadcastReceiver” and overrides the method “onReceive”.

```
<receiver android:name=".OutgoingCallReceiver">
<intent-filter>
<action android:name="android.intent.action.NEW_OUTGOING_CALL" />
</intent-filter>
</receiver>
```

Figure 10. XML registering for outgoing call notification

Now when the user tries to dial, our overridden method receives the notification, and it aborts the call. This is done by calling the method “abortBroadcast”. This method will prevent any other broadcast receivers from receiving the broadcast. As a result, the basic telephony application is not receiving the event of an outgoing call and, therefore, the user cannot dial a number. Figure 11 shows the notification that is displayed by our application when intercepting and aborting an outgoing call.

6.4 Validation

The above telephony exploitation application could have been installed under a false claim of beneficial features. However, the above two malicious attacks indicate that once an application is installed, it can have unpredictable behavior making this a Trojan horse attack. The operating system has no control over preventing such attacks.

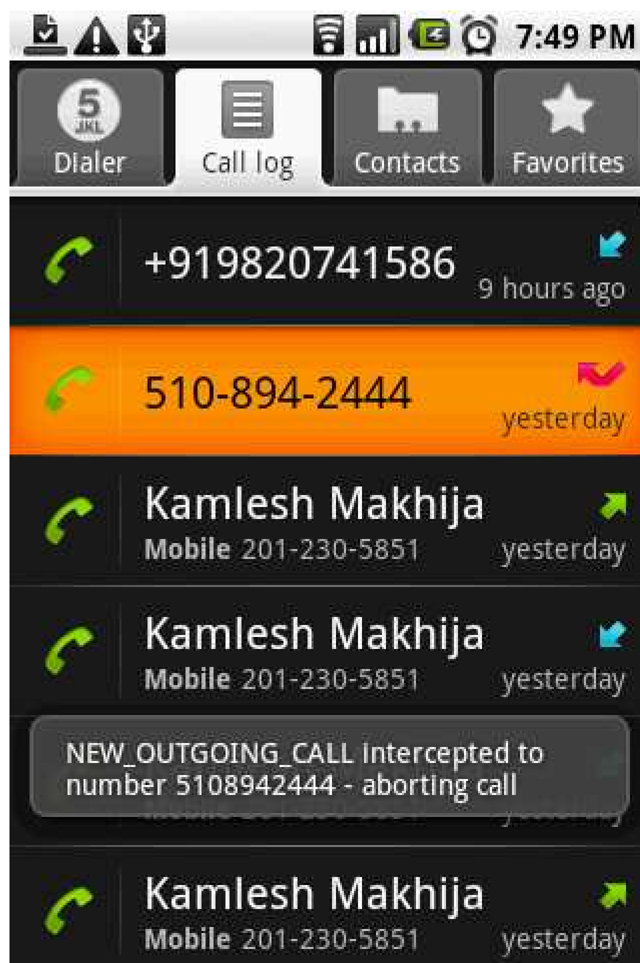

Figure 11. Outgoing voice call aborted

7.0 Security Feature for Android

As mentioned in Section 2.5, every Android application owns a private area of memory on the internal file system. As a result, every owner application has permission only to read and write files that it creates. This is because the files are created in the default private mode; that is, MODE_PRIVATE. No other application can access this file unless the owner application explicitly grants a global read-write access using the modes MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE.

However, this feature is not applicable to files stored on external media such as a Secure Digital (SD) Card. This means that if a private file needs to be stored on the SD card by an application, then such a file is accessible to any other application. It is common for applications on an Android phone to use external media for storing data since the internal phone memory is small—no more than 512MB on Nexus One [18]. Therefore, we have developed a solution to provide privacy access to files stored on the external media just as the framework provides for phone memory files.

7.1 Encryption of files stored on SD card

To overcome this limitation of storing easily unprotected files on the external memory, we have introduced an API that can be extended as a library and used by other Android applications looking for such a feature.

To use this feature, the application needs to call the library interface with the name of the file to be encrypted and a password. Every file stored on the media card is encrypted the first time by a randomly generated key. To enable decryption, the key needs to be persistent. Since we are trying to emulate the private access feature of the internal file system, we have stored the key in the application's private file space. The password provided to the interface is used to protect the encryption key.

The encryption algorithm works as follows:

1. The encryption algorithm selected is DES.
2. A symmetric key is generated for the purpose of encryption. This is done by using the KeyGenerator class of Java. We assume this provides enough randomness and generates a hard-to-guess key. Using this as the encryption key, the data to be stored on the external media card is encrypted and stored on the card.

3. From the password entered by the user, a hash is computed. The hashing algorithm used is MD5. This generates a 128-bit hash value.
4. The computed hash value serves as the key for encrypting the symmetric key. The encrypted symmetric key is stored on the internal phone memory within the application's privately accessible file space.

We define E(data,key) to mean that the algorithm encrypts "data" with the "key" and h(password) denotes a cryptographic hashing function. Then the above algorithm is as follows:

KeyGenerator("DES") → *SymmetricKey*

E(Data to store on SD Card, SymmetricKey) → *Encrypted Data on SD Card*

E(SymmetricKey, h(Password entered by the user)) → *Encrypted Key stored on phone memory in the application's private file space.*

Figure 12 is the code sample for encryption of file data and storing of the key in the private file space.

To demonstrate encryption, we have created our own user interface, as seen in Figure 13. The user needs to specify the name of the file that he or she wants to encrypt along with the password. The password is hashed and serves as a key for encrypting the symmetric key.

For the purpose of this example, since we do not have access to the private files of the phone, the files to be encrypted are on the SD Card. The file selected in Figure 13 is stored after encryption as “AalIzzWell_encrypted.mp3,” and its corresponding encrypted symmetric key is stored in the application’s private directory as “AalIzzWell_key.txt.”

```
String algorithm = "DES";
Cipher c1 = Cipher.getInstance(algorithm);
SecretKey myKey = KeyGenerator.getInstance(algorithm).generateKey();
c1.init(Cipher.ENCRYPT_MODE, myKey);
encrypted = c1.doFinal(fileText);

//encrypt the key and store that on the local storage
String name =
fileName.substring(0,fileName.lastIndexOf('.'))+"_key.txt";
FileOutputStream outStream = openFileOutput(name, MODE_PRIVATE);

byte[] raw = myKey.getEncoded();
//store the raw bytes of the encrypted key
SecretKeySpec keySpec = new
SecretKeySpec(hashPassword(passCode),0,8,algorithm);

SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance(keySpec.getAlgorithm());

Key key = keyFactory.generateSecret(keySpec);
c1.init(Cipher.ENCRYPT_MODE, key);
byte[] encryptedKey = c1.doFinal(raw);
outStream.write(encryptedKey);
outStream.close();
```

Figure 12. Code for encryption

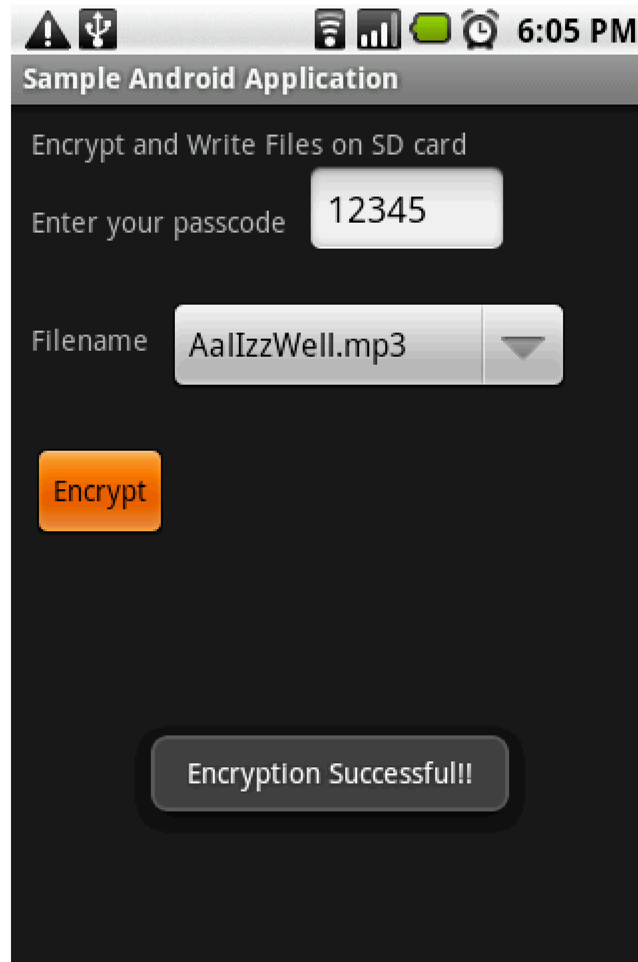

Figure 13. Encryption of file on external media

7.2 Encryption limitations

Some limitations of this encryption process are:

1. Encryption will work only if the SD Card is not being used as a USB storage medium by the computer to which the phone is connected. In this event, the user is notified, as in Figure 14, that no SD Card is available on the phone. The reason for this limitation is that the Android framework does not want to get into the situation of handling synchronization issues; to avoid this, the external card is locked for the device to which it is available. To continue, one can just scroll down the notification panel and switch off the USB storage, as seen in Figure 15.

2. For file sizes greater than 2MB, the current encryption APIs are slow and can fail. We are using the Cipher class in Java Development environment for the purpose of encryption. It appears that this is due to the limited memory available on the phone, most of which the encryption process consumes.

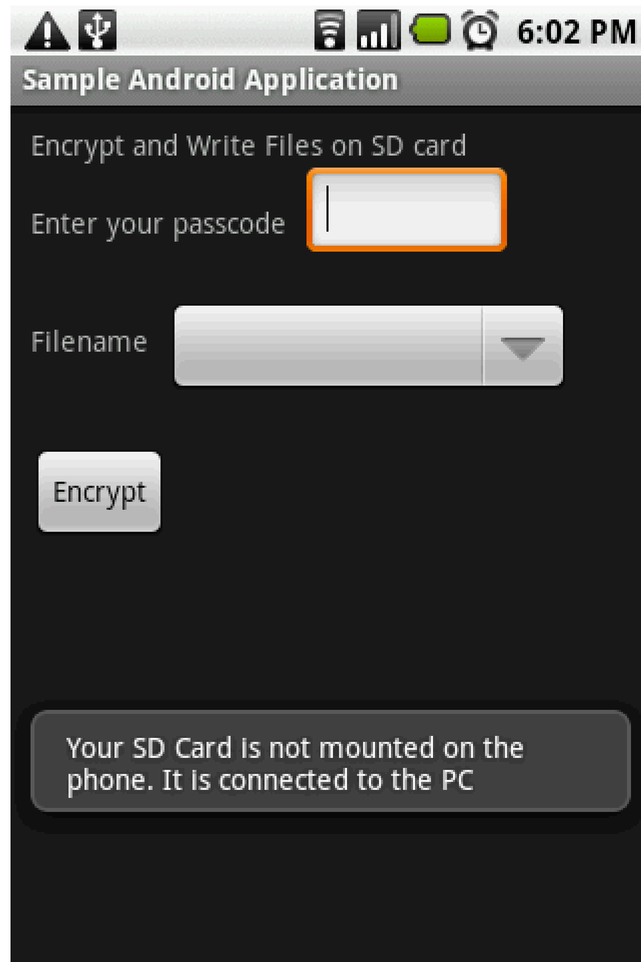

Figure 14. External media unavailable on encryption

Figure 15. Switching off USB storage

7.3 Decryption algorithm

The decryption process works in the complete reverse order of encryption. The algorithm can be broadly outlined as:

D(Encrypted symmetric key file stored on application's private file space, h(Password entered by the user)) → *SymmerticKey*

D(Encrypted file, SymmetricKey) → *Original File*

Figure 16 shows an encrypted file selected from the SD card after the user has entered the password. In case the password entered is incorrect, the user is notified that the SecretKey used

for encryption is unobtainable. This happens due to “BadPaddingException” thrown when trying to obtain the SecretKey. Figure 17 is an example of this scenario. The decrypted file of Figure 16 is Diya_decrypted.jpg.

Figure 16. Decryption successful

Figure 17. Decryption failure

7.4 Validation

To validate our encryption feature, we have checked the encrypted and the decrypted versions of various files. We have also tried the above process in different file formats such as mp3, jpg, and txt.

The encrypted versions of mp3 and jpg are not in the corresponding file format as these do not open with their respective file viewers. These files were rejected as being non-supported formats. In addition, the encrypted txt file includes non-readable characters.

On decrypting any of these encrypted versions, we found that we obtained our original data. Thus, this validates our technique of making external data securely available to an application.

8.0 Conclusion

With the current security architecture, most smartphones are vulnerable to attacks because the user of the phone is instrumental in deciding on applications to be installed on the phone. It is not easy for a user to judge applications by their description. The Android framework is one platform that expects the user to be security conscious and implicitly assumes application developers are not sinister. Because of this, a user may unknowingly install software that poses a security threat. Our telephony application provides such an example.

To free the user from making decisions as to which applications to install, the security framework could introduce the concept of a runtime check for each application. Any application not behaving in the expected manner would then raise an alarm.

We also observed that the Android security framework is confined to the storage media available on the phone. We extended this security to the external media.

We conclude that the Android security framework is susceptible to vulnerabilities and has scope for improvement. We believe that the Android security framework needs to extend its static install time “Permission”-based security model to a more dynamic runtime security provider and also incorporate security for external storage mediums.

References

[1] Hypponen M. 2006. Malware goes mobile. [Internet] Sci Am: 70−77. Available from: http://www.cs.virginia.edu/~robins/Malware_Goes_Mobile.pdf

[2] Burns J. 2009. Developing secure mobile applications for Android. Black Hat. [Internet]. Available from: https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf

[3] Ongtang M, McLaughlin S, Ench W, McDaniel P. 2009 Dec. Semantically rich application-centric security in Android. In: Proceedings of Annual Computer Security Applications Conference (ACSAC 2009) [place unknown]. [Internet]. Available from: http://www.patrickmcdaniel.org/pubs/acsac09a.pdf

[4] Hoffman D. 2007. Blackjacking security threats to BlackBerry devices, PDAs and cell phones in the enterprise. Indianapolis (IN): Wiley.

[5] Android Developers. 2007. [Internet]. Available from: http://developer.android.com/guide/topics/fundamentals.html

[6] Android Developers. 2007. [Internet]. Available from: http://developer.android.com/guide/basics/what-is-android.html, 2007

[7] Trend Micro. 2005. Security for mobile devices: Protecting and preserving productivity. [Internet]. Available from: http://us.trendmicro.com/imperia/md/content/us/pdf/products/homeandhomeoffice/mobilesecurity/wp01tmms0020060104us.pdf

[8] Garfinkel S. 2010. How Android security stacks up. [Internet]. Available from: http://www.technologyreview.com/communications/24944/page2/

[9] Burns R. Symbian [Internet]. Symbian Signed. Available from: http://developer.symbian.org/wiki/index.php/Category:Symbian_Signed

[10] Forum Nokia. [Internet]. Symbian Signed accessing manufacturer capabilities. Available from: http://www.forum.nokia.com/info/sw.nokia.com/id/043b231a-2d40-46c3-b9a8-1c0cfa46de6f/Symbian_Signed_Accessing_Manufacturer_Capabilities.html

[11] Ottaway W. QinetiQ. [Internet]. Mobile security: Cause for concern? Available from: http://apps.qinetiq.com/perspectives/pdf/EP_White_Paper4_Mobile_Sec.pdf

[12] Apple. [Internet]. iPhone in business: Security overview. Available from: http://images.apple.com/iphone/business/docs/iPhone_Security_Overview.pdf

[13] Chickowski E. 2009. 10 best practices for mobile device security. [Internet]. Available from: http://www.baselinemag.com/c/a/Mobile-and-Wireless/10-Best-Practices-for-Mobile-Device-Security/

[14] Wikipedia. [Internet]. Symbian OS. Available from: http://en.wikipedia.org/wiki/Symbian_OS

[15] Symbian Signed. [Internet]. Available from: https://www.symbiansigned.com/app/page

[16] Android Developers. [Internet]. Available from: http://developer.android.com/guide/topics/data/data-storage.html

[17] Independent Security Evaluators. [Internet]. Exploiting Android. Available from: http://securityevaluators.com/content/case-studies/android/index.jsp

[18] Wikipedia. [Internet]. Nexus One. Available from: http://en.wikipedia.org/wiki/Nexus_One