Analyzing and Dissecting Android Applications for Security defects and Vulnerabilities

Rushil Shah – Security Researcher and Consultant at Blueinfy

Abstract & Problem Domain

It seems so long ago when all cell phones were capable of, was making a phone call. Smartphones have taken over the market in a frenzy. Why shouldn't they? Today email, social networking, banking – everything is possible on the go with smartphones. Smartphones come equipped with features like data, Wi-Fi, voice and GPS functions. The sudden growth in the number of applications available for smartphones does raise a certain level of concern for the user's security. A report by McAfee claims that the year 2011 will be the year when smartphones become targets for malicious software and hackers. The mobile security field is still at very nascent stage.

According to a recent market survey 50% of the devices run Google's Android operating system, 25% use Apple iOS, with the rest being shared by Blackberry, Symbian and others. In March 2011, 58 malicious applications were found in the Android Market. Before Google could remove the applications from the Android Market they were downloaded to around 260,000 devices. These applications, in the Android market, were malicious applications which contained Trojans hidden in pirated versions of legitimate applications. *DroidDream*, a malware, exploited a bug which was present in versions of Android, older than 2.2.2. Android device manufacturers and carriers work in tandem to distribute Android-based updates and had not uniformly issued patches to their customers for the DroidDream exploit, leaving users vulnerable. Google said the exploit allowed the applications to gather device specific information, as well as personal information. The exploit also allowed the applications to download additional code that could be run on the device. Many users perform credit card transactions and access social networking sites. If the applications used to perform these activities are not secure, hackers can gain access to vital information that can be misused.

Each application should be tested first and validated before it is sent to devices. More than malicious applications, it is bugs and loopholes in current applications that may be abused by hackers. Phishing attacks can be performed on many sites that are accessible with the device. Common vulnerabilities include local data storage, insecure HTTP connections, etc. A logical solution is to test the applications once they are developed. Many methods exist, through which these applications can be tested, one of which is reverse engineering. Reverse engineering is the process of discovering the working of an application through analysis of its structure, function and operation. From reverse engineering we can understand complete operations of an application. For example, it is possible to know which web services an application calls or where and how the application stores temporary user data on local storage. These are indicators of the security posture of an application. Vulnerabilities, if any exist, can be addressed.

About the Author

Rushil is working with Blueinfy in the area of Application and Mobile security consulting and research. From an academic standpoint, he has a background in computer engineering.

Android Working Model – A first cut

Before we understand Android security we must first understand how an application runs on the device. An application runs in a sandboxed architecture on the Android device. Each application has its own assigned permissions and working memory. Different applications can communicate with each other. An application can request for system resources only through intents. Each android application is complied into an Android Package (APK) file. This *apk* file is installed onto the device.

The APK file format is a variant of the JAR file format.

An APK file usually contains the following folders:

- META-INF
- res

along with a set of files:

- AndroidManifest.xml
- classes.dex
- resources.arsc

The two main files for consideration are the `Androidmanifest.xml` and `classes.dex` files.

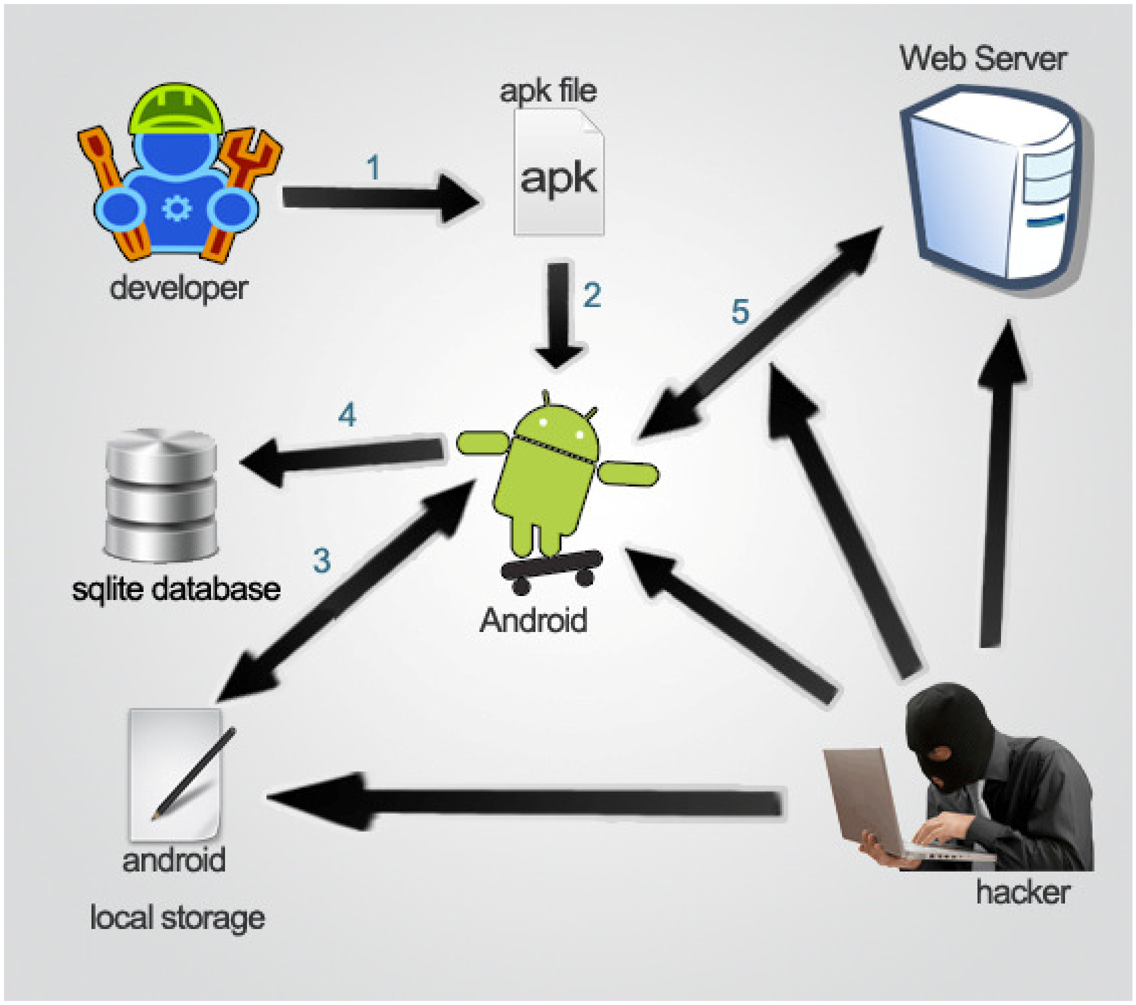
The `Androidmanifest.xml` presents essential information about the application to the Android system; information the system must have before it can run any of the application's code. Among other things, the manifest does the following:

- It contains the Java package name for the application.
- The activities, services, broadcast receivers, and content providers that the application is composed of are listed in the manifest, along with their capabilities. These declarations let the Android system know what the components are and under what conditions they can be launched.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the libraries that the application must be linked against.

The `classes.dex` file contains the *Dalvik bytecode* of all the code in the application.

Android Threat Model and OWASP Top 10

An Android application can be developed and deployed. The figure below shows how an attacker looks at the deployed application:



The following steps would be followed with their own logical outcomes.

1. The developer complies the code with an Android package file.
2. The *apk* file is deployed on the Android device.
3. Applications storing data on local storage of the device – *secure/insecure storage*?
4. Sqlite database of Android, where data and system settings are saved – *secure/insecure database calls*?
5. Device connection to the internet and to the Web server – *entry points to the application*?

As you can see, steps 3 to 5 can have security issues. These calls need to be analyzed for information leaks. An attacker can grab this information and compromise a user session. More attention needs to be paid to the OWASP top 10 list of issues or vulnerabilities in this domain.

OWASP's first draft of Top 10 mobile security risks:

Following is the list of OWASP Top 10 and vulnerabilities we need to discover during assessment and testing.

1. Insecure or unnecessary client-side data storage: Data from applications such as user credentials(username and password), credit card information may be stored on the device's memory. This data, if not properly encrypted, can be accessed by a hacker and the data stolen. Example: recent Skype vulnerability.
2. Lack of data protection in transit: if the connection between the web and the device is not secure than the transaction can be tampered with.
3. Personal data leakage: browser cache, search history records, location tracking – data, if not secured, can be accessed by the attacker.
4. Failure to protect resources with strong authentication: certain applications, like Google, have single sign-ons, which can be used by the attacker to gain access to the account.
5. Failure to implement least privilege authorization policy: Some applications may have been given more permissions than necessary. For example, a file requiring READ permission is assigned READ WRITE permission.
6. Client-side injection: Client side XSS and SQL injections can be performed on the device.
7. Client-side DOS: a particular service or application is blocked for access. For example, if the contacts list has been blocked by a DOS attack, the user will not be able to access the list to make calls.
8. Malicious third-party code: Malicious third party code installed on the device can gain access to device resources and data.
9. Client-side buffer overflow: Certain native libraries in Android are vulnerable to client side buffer overflow attacks because of improper or insufficient input/ouput validation.
10. Failure to apply server-side controls: Any attacker can pose as the client and attempt SQL Injection, XSS or other attacks.

Our objective is two-fold: to analyze an application for susceptibility to the above mentioned issues and to perform this particular task in a methodical way. For example, static analysis can be done one this `apk` file to discover vulnerabilities. Let us see how this can be done with simple ruby script(s). Let us call it *ScanDroid*.

Scanning APK files with ScanDroid and underlying utilities

ScanDroid reverse engineers an Android application, scans through the code and detects possible vulnerabilities for application testing. It points out some interesting calls. We can then decide whether these calls are secure or not. Reverse engineering is the process of discovering the working of an application through analysis of its structure, function and operation along with applying static code analysis on both object and source code.

This article introduces ScanDroid for Android applications, using Ruby code to show how it works and demonstrate how to implement it. This code is a prototype to highlight the capabilities of using ScanDroid. For simplicity, we will consider 3 vulnerabilities for an Android application.

1) Read/Write to local Storage
2) Access external URL
3) Make Socket Connection

This document explains the following aspects:

- ScanDroid Overview
- Using ScanDroid
- Using ScanDroid library with interactive Ruby (irb)

ScanDroid extracts these files from the APK file. It converts the Dalvik bytecode files into Java code files and then scans the code for possible vulnerabilities using the rules defined – all with just one command.

ScanDroid performs the following stages:

- **Extract**
  - Extracts all the files from the APK file
- **Xml**
  - Converts the files to readable xml formats and the classes to .smali format.
- **Dex2clas**
  - Converts the classes.dex file to separate .class files
- **Clas2jav**
  - Converts the class files to readable java code.
- **Check**
  - Checks the code for possible vulnerabilities.
- **Manifest**
  - Checks the androidmanifest.xml for additional information.

Intermediate files at each stage of the process are created for further reference. The ScanDroid library can be used to manually execute each stage of the process.

Let's walk through ScanDroid

ScanDroid is built using the Ruby scripting language. To start ScanDroid go to the Command Prompt and type in the following line:

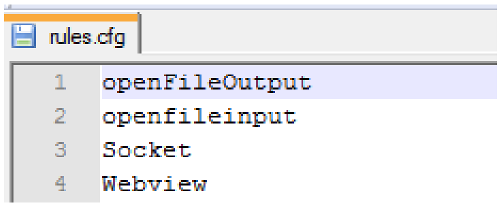
```
ScanDroid.rb –apk <application filename> -rule <rules filename>
```

For test purposes we have an Android application called `scanme`. We have the rules set up in a file called rules.cfg:

```
ScanDroid.rb –apk scanme.apk -rule c:\ScanDroid\bin\rules.cfg
```

Make sure that ScanDroid is launched from its installation folder.

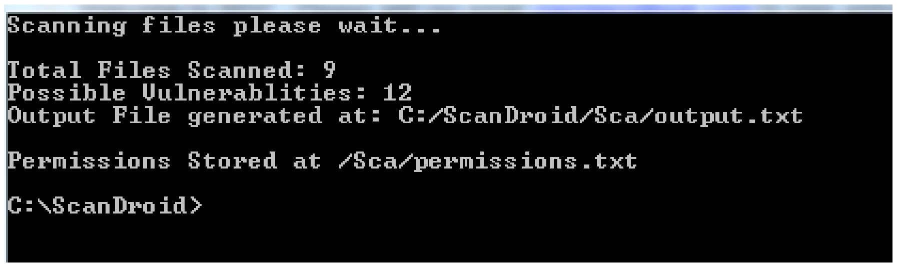
The `rules.cfg` file contains the list classes and methods used in the Java code for discovering possible vulnerabilities. The 3 possible vulnerabilities that we have considered for `ScanMe` are stored in the `rules.cfg` as shown:



```
openFileOutput
openfileinput
Socket
Webview
```

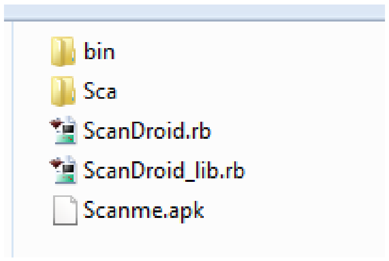
We are looking for interesting calls and trying to search through code.

The output:



```
Scanning files please wait...

Total Files Scanned: 9
Possible Vulnerablities: 12
Output File generated at: C:/ScanDroid/Sca/output.txt

Permissions Stored at /Sca/permissions.txt

C:\ScanDroid>
```

A new folder is created with the name of the first 3 characters of your Android application in the installation folder.

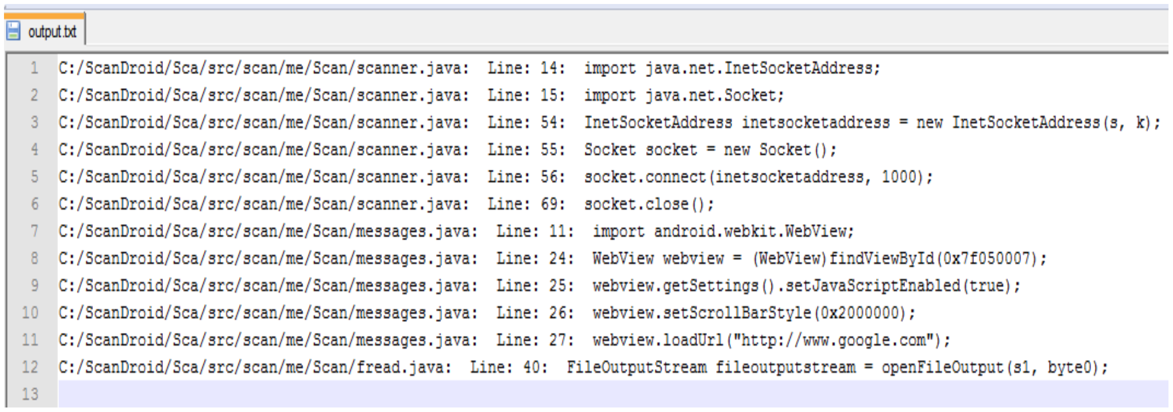


This folder has the following sub-folders:

- *Class*: contains `.class` files
- *Dex*: contains `classes.dex` file
- *Src*: contains the converted Java code

- *Xml:* contains readable xml files
- *Output.txt* : contains the lines of code where possible vulnerabilities are detected
- *Permissions.txt*: contains the list of permissions for the application

Output.txt



output.txt

```
 1  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 14:  import java.net.InetSocketAddress;
 2  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 15:  import java.net.Socket;
 3  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 54:  InetSocketAddress inetsocketaddress = new InetSocketAddress(s, k);
 4  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 55:  Socket socket = new Socket();
 5  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 56:  socket.connect(inetsocketaddress, 1000);
 6  C:/ScanDroid/Sca/src/scan/me/Scan/scanner.java:  Line: 69:  socket.close();
 7  C:/ScanDroid/Sca/src/scan/me/Scan/messages.java:  Line: 11:  import android.webkit.WebView;
 8  C:/ScanDroid/Sca/src/scan/me/Scan/messages.java:  Line: 24:  WebView webview = (WebView)findViewById(0x7f050007);
 9  C:/ScanDroid/Sca/src/scan/me/Scan/messages.java:  Line: 25:  webview.getSettings().setJavaScriptEnabled(true);
10  C:/ScanDroid/Sca/src/scan/me/Scan/messages.java:  Line: 26:  webview.setScrollBarStyle(0x2000000);
11  C:/ScanDroid/Sca/src/scan/me/Scan/messages.java:  Line: 27:  webview.loadUrl("http://www.google.com");
12  C:/ScanDroid/Sca/src/scan/me/Scan/fread.java:  Line: 40:  FileOutputStream fileoutputstream = openFileOutput(s1, byte0);
13
```

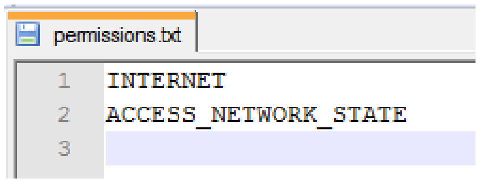
ScanDroid matches the rules in `rules.cfg` file with the Android application code. The filename, line no and the line of code are added to `output.txt` wherever a match is detected.

In our example, in line 4 of the output file a new socket is called. This code occurs in the 55th line of `scanner.java` file. A socket call is made from the application. This call can be tracked using ScanDroid.

Similarly, in the file `fread.java`, data is written to local file storage. If the data is not secured, this is a possible vulnerability. `OpenFileOutput` writes data to a file on the device.

`Webview` calls the browser of the device. `Webview.loadUrl` launches a URL in the browser. This URL can be tracked with ScanDroid to check whether the browser call is sent to only the required site or to other insecure sites too.

Permissions.txt



permissions.txt

```
INTERNET
ACCESS_NETWORK_STATE
```

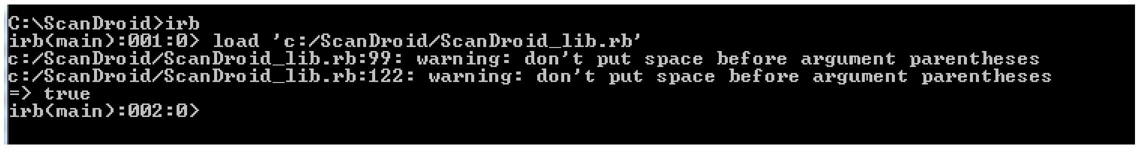
The `permission.txt` file lists the permissions of the application taken from the `Androidmanifest.xml` file.

Using ScanDroid Library with Interactive Ruby (Irb)

Each stage of ScanDroid can be executed individually using the ScanDroid Library in Irb.

Start the Irb by typing irb at the Command Prompt.

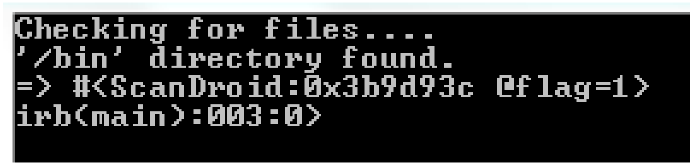
Load the ScanDroid library using the *load* command



```
C:\ScanDroid>irb
irb(main):001:0> load 'c:/ScanDroid/ScanDroid_lib.rb'
c:/ScanDroid/ScanDroid_lib.rb:99: warning: don't put space before argument parentheses
c:/ScanDroid/ScanDroid_lib.rb:122: warning: don't put space before argument parentheses
=> true
irb(main):002:0>
```

Next, create an object for the ScanDroid Class. Let this object be ***t***:

```
. t=ScanDroid.new
```



```
Checking for files....
'/bin' directory found.
=> #<ScanDroid:0x3b9d93c @flag=1>
irb(main):003:0>
```

The first step will be to extract the files from the `apk` file.

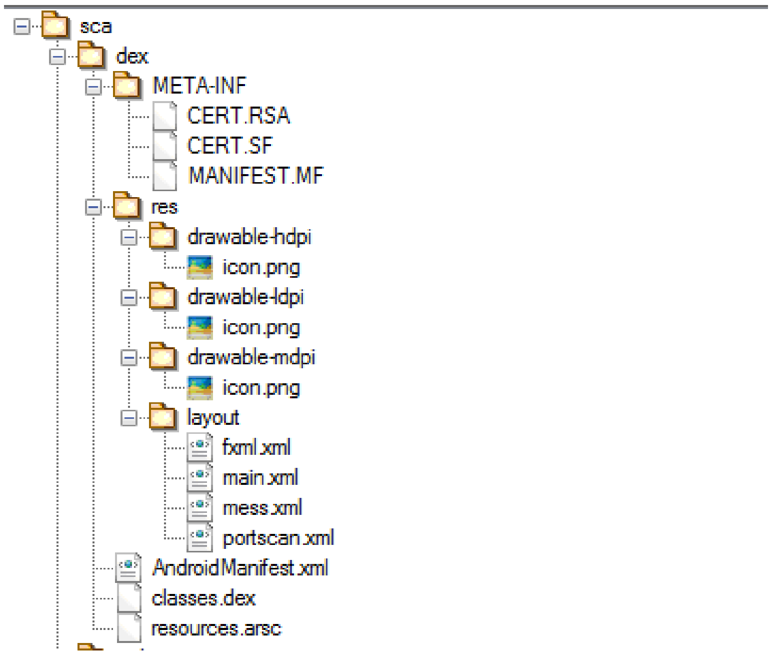
```
t.extract('filename.apk')
```

The code for the extraction:

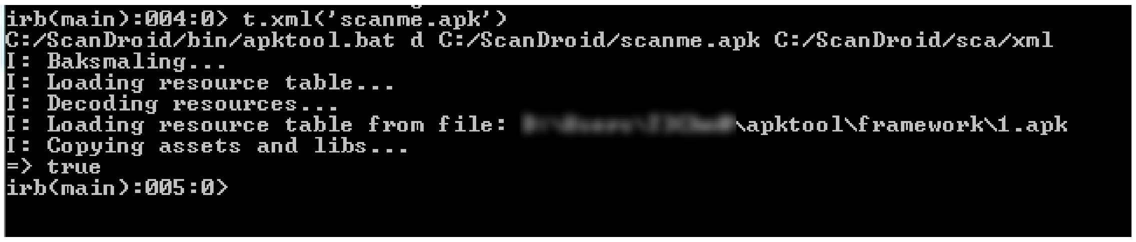
```
  Zip::ZipFile.open(filename) {|file|
     file.each do |f|
     f_path = File.join(destination, f.name)
     FileUtils.mkdir_p(File.dirname(f_path))
     file.extract(f, f_path)
     end
     }
```

This code uses a ruby gem Zip to extract all the files from the `apk` to a proper directory. Each file is checked and placed according to its zipped structure.

All the extracted files are stored inside the `dex` folder:



The next step is to convert the xml files to readable format.

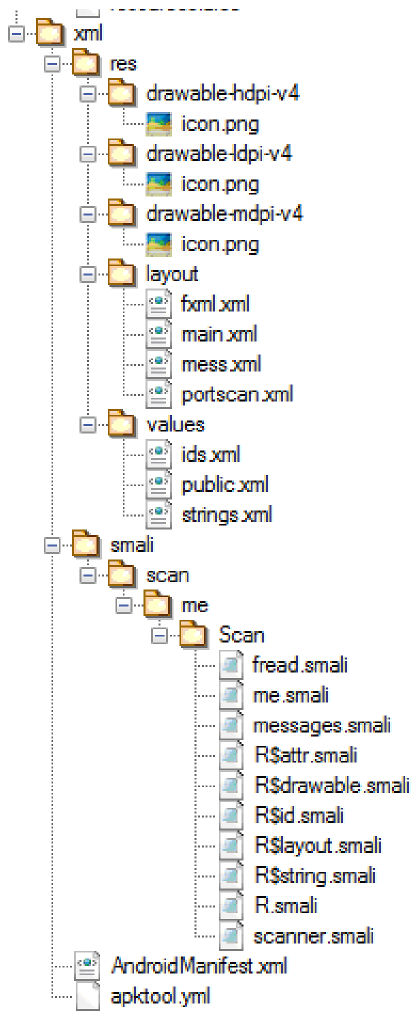


```
irb(main):004:0> t.xml('scanme.apk')
C:/ScanDroid/bin/apktool.bat d C:/ScanDroid/scanme.apk C:/ScanDroid/sca/xml
I: Baksmaling...
I: Loading resource table...
I: Decoding resources...
I: Loading resource table from file: [redacted]\apktool\framework\1.apk
I: Copying assets and libs...
=> true
irb(main):005:0>
```

For xml, we use an external tool called `apktool`

```
cmd=Dir.getwd+"/bin/apktool.bat d "+cur+"/"+filename+" "+cur+"/#{filename[0,3]}/xml"
puts cmd
system(cmd)
```

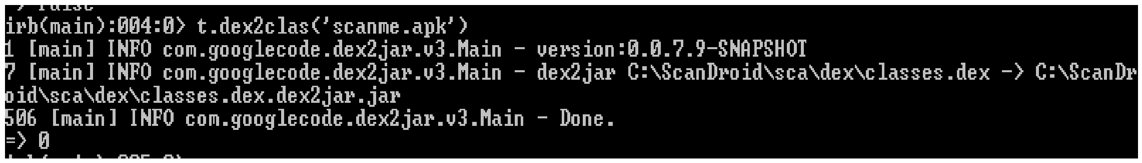
This executes the apktool. The first parameter is the location of the `apk` file and the second is the destination folder.



```
xml
    res
        drawable-hdpi-v4
            icon.png
        drawable-ldpi-v4
            icon.png
        drawable-mdpi-v4
            icon.png
        layout
            fxml.xml
            main.xml
            mess.xml
            portscan.xml
        values
            ids.xml
            public.xml
            strings.xml
    smali
        scan
            me
                Scan
                    fread.smali
                    me.smali
                    messages.smali
                    R$attr.smali
                    R$drawable.smali
                    R$id.smali
                    R$layout.smali
                    R$string.smali
                    R.smali
                    scanner.smali
    AndroidManifest.xml
    apktool.yml
```

Now, we need to convert the `dex` files to `class` files.

We shall use the `dex2jar` tools to convert the *dalvik* code to Java .jar files.



```
irb(main):004:0> t.dex2clas('scanme.apk')
1 [main] INFO com.googlecode.dex2jar.v3.Main - version:0.0.7.9-SNAPSHOT
7 [main] INFO com.googlecode.dex2jar.v3.Main - dex2jar C:\ScanDroid\sca\dex\classes.dex -> C:\ScanDr
oid\sca\dex\classes.dex.dex2jar.jar
506 [main] INFO com.googlecode.dex2jar.v3.Main - Done.
=> 0
```

```
cmd=Dir.getwd+"/bin/dex2jar/dex2jar.bat"+cur+"/#{filename[0,3]}/dex"+"/clas
ses.dex"
    system(cmd)
    dest=Dir.getwd+"/#{filename[0,3]}/class"
    jar2zip=Dir.getwd+"/#{filename[0,3]}/dex/classes.dex.dex2jar.jar"
    Zip::ZipFile.open(jar2zip) {|file|
      file.each do |f|
      f_path = File.join(dest, f.name)
      FileUtils.mkdir_p(File.dirname(f_path))
      file.extract(f, f_path)
      end
```

The *jar* file generated is then extracted using the Zip gem of Ruby to get the class files which are stored in the class directory.

Once the Class files are generated we need to convert these to Java code.

We move to the `clas2jav` stage

To call the method we execute:

```
t.clas2jav ('applicationname.apk')
```

Again, here we use a Java decomplier JAD. The method executes the command:

```
jad –o –r –sjava –d /src  /class /**/*.class
```

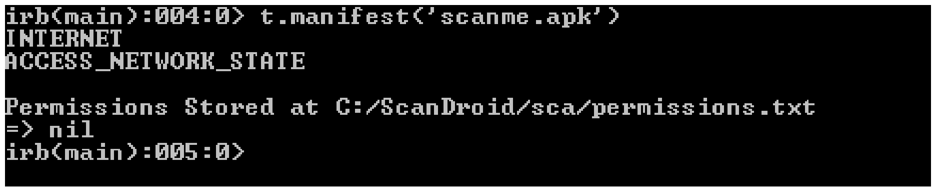
`/src` is the destination folder where all the *java* files will be saved.

`/class` is where all our *class* files are saved.

`/**/*.class` tells JAD to convert all the `.class` files, including those within sub folders.

Now we have the complete code of the application. ScanDroid can now scan the files and check for vulnerabilities.

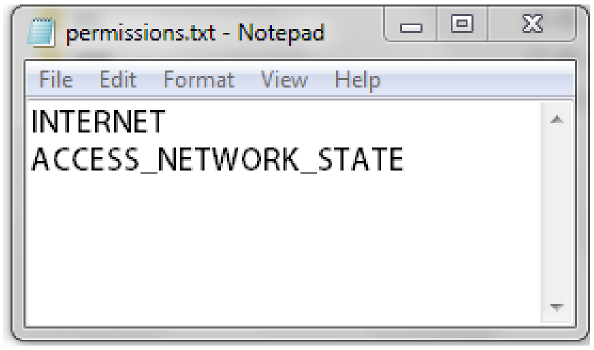
First we scan the `AndroidManifest.xml` for permissions of the application:



```
irb(main):004:0> t.manifest('scanme.apk')
INTERNET
ACCESS_NETWORK_STATE

Permissions Stored at C:/ScanDroid/sca/permissions.txt
=> nil
irb(main):005:0>
```

```
File.open("#{cur}/#{filename[0,3]}/xml/AndroidManifest.xml","r") do
|infile|

        while(line=infile.gets)

          if line[/android.permission./]
```

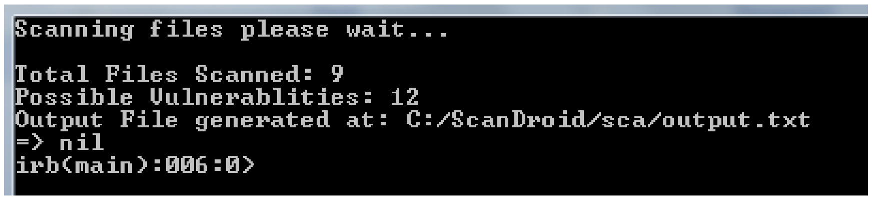
Each of the permissions is defined with the preceding string 'android.permission'. ScanDroid scans the code for this string and, if a match if found, saves the lines to a file to be printed later.



```
INTERNET
ACCESS_NETWORK_STATE
```

We need to scan the code that we decompiled for vulnerabilities:

```
t.check('scanme.apk','c:/ScanDroid/bin/rules.cfg')
```



```
Scanning files please wait...

Total Files Scanned: 9
Possible Vulnerablities: 12
Output File generated at: C:/ScanDroid/sca/output.txt
=> nil
irb(main):006:0>
```

```
if  File.file?(entry) and entry[/.+\.java$/]
          File.open(entry,"r") do |infile|
            while(line = infile.gets)
              c=c+1
              File.open(rulefile,"r") do |rule|
                while(r=rule.gets)
                  exp= r.chomp
                  if line[/#{exp}/i]
                      Store the line in the file.
```

The code scans each file in the `src` directory and matches each line with the rules defined in the file `rules.cfg`. A confirmed match saves the line to the `output.txt` file.

Conclusion:

The mobile security field is still, at its infant stage. With the recent increase in the number of vulnerabilities found on smartphones, a lot of attention is needed to make smartphones more secure. Android is now the fastest spreading mobile operating system and therefore requires more focus on security. Tools like ScanDroid help in analyzing applications and thus help in making more secure applications.

Appendix A – ScanDroid Code

ScanDroid.rb

```
require 'ScanDroid_lib.rb'
def run
  t=ScanDroid.new
  case ARGV[0]
    when "-apk"
      @filename=ARGV[1]
    else
      system('cls')
      puts "Error in correct syntax!\nTry: ScanDroid.rb -apk [Filename.apk] -rule [rule_file]"
      Process.exit
  end
  case ARGV[2]
    when "-rule"
      @rule=ARGV[3]
    else
      system('cls')
      puts "Error in correct syntax!\nTry: ScanDroid.rb -apk [Filename.apk] -rule [rule_file]"
      Process.exit
  end
  if !@filename
    puts "Enter Filename(including extension): "
    @filename=gets.chomp
  end
  if !@rule
    puts "Enter Rules File(including extension): "
    @rule=gets.chomp
  end
  if !File.exists?(@filename)
    puts "\nCannot find #{@filename}! Please make sure the path of the file is correct."
    Process.exit
  end
  if !File.exists?(@rule)
    puts "\nCannot find #{@rule}! Please make sure the path of the file is correct."
    Process.exit
  end
  puts "Working on "+@filename+" ... "
  t.extract(@filename)
  system('cls')
  t.xml(@filename)
  system('cls')
  t.dex2clas(@filename)
  system('cls')
  t.clas2jav(@filename)
  system('cls')
  t.manifest(@filename)
  system('cls')
  t.check(@filename,@rule)
  puts ("\nPermissions Stored at /#{@filename[0,3]}/permissions.txt\n")
end
system('@echo off')
system('prompt.')
run()
system('prompt')
```

ScanDroid_lib.rb

```
require 'zip/zip'
require 'find'
class ScanDroid
  def initialize
    system('cls')
    @flag=1
    puts "Checking for files.... "
    if File.directory?("bin")
      puts "'/bin' directory found."
    else
      puts "'/bin' directory could not be found. \nPlease run the program from the correct
folder."
      Process.exit
    end
  end
  def extract(filename)
      destination=Dir.getwd+"/#{filename[0,3]}/dex"
      begin
      Zip::ZipFile.open(filename) {|file|
      file.each do |f|
      f_path = File.join(destination, f.name)
      FileUtils.mkdir_p(File.dirname(f_path))
      file.extract(f, f_path)
      end
      }
      rescue
        puts "\nError Unpacking the File! Delete all Previously generated files of this apk."
        Process.exit
      end
  end
  def xml(filename)
    cur=Dir.getwd
    if File.exists?("#{cur}/bin/apktool.bat")
    cmd=Dir.getwd+"/bin/apktool.bat d "+cur+"/"+filename+" "+cur+"/#{filename[0,3]}/xml"
    puts cmd
    system(cmd)
    else
      puts "Cannot Find apktool.bat!"
      Process.exit
    end
  end
  def dex2clas(filename)

    cur=Dir.getwd
    begin
    if File.exists?("#{cur}/bin/dex2jar/dex2jar.bat")
    cmd=Dir.getwd+"/bin/dex2jar/dex2jar.bat "+cur+"/#{filename[0,3]}/dex"+"/classes.dex"
    system(cmd)
    dest=Dir.getwd+"/#{filename[0,3]}/class"
    jar2zip=Dir.getwd+"/#{filename[0,3]}/dex/classes.dex.dex2jar.jar"
    Zip::ZipFile.open(jar2zip) {|file|
      file.each do |f|
      f_path = File.join(dest, f.name)
      FileUtils.mkdir_p(File.dirname(f_path))
      file.extract(f, f_path)
      end
      }
      else
        puts "\nCannot find dex2jar.bat in the bin/dex2jar/dex2jar.bat directory!"
        Process.exit
      end
      rescue
        puts "\nError Converting Dex to Class!"
        Process.exit
      end
      Dir.chdir(cur)
  end
  def clas2jav(filename)
      cur=Dir.getwd
      cmd=cur+"/bin"
      system(cmd)
      Dir.chdir(Dir.getwd+"/bin")
      cmd="jad -o -r -sjava -d"+cur+"/#{filename[0,3]}/src
"+cur+"/#{filename[0,3]}/class"+"/**/*.class"
      if File.exists?("jad.exe")
      system(cmd)
      else
        puts "\nJava Decompiler file jad.exe could not be found!"
        Process.exit
      end
      Dir.chdir(cur)
  end
  def check(filename,rulefile)
      cur=Dir.getwd
      @flag=1
      system('cls')
      puts ("Scanning files please wait...")
      fc=0
      vc=0
      Find.find(cur+"/#{filename[0,3]}/src") do |entry|
        fc=fc+1
        if  File.file?(entry) and entry[/.+\.java$/]
```

```
          c=0
          File.open(entry,"r") do |infile|
            while(line = infile.gets)
              c=c+1
              File.open(rulefile,"r") do |rule|
                while(r=rule.gets)
                  exp= r.chomp
                  if line[/#{exp}/i]
                      open(cur+"/#{filename[0,3]}/output.txt","a") do |f| f.puts
(File.expand_path(entry)+":  Line: #{c}:  "+line.strip+"\n") end
                      vc=vc+1
                  end
                end
              end
            end
          end
        end
      end
      puts ("\nTotal Files Scanned: #{fc}\nPossible Vulnerablities: #{vc}\nOutput File
generated at: #{cur}/#{filename[0,3]}/output.txt\n")
    end
    def manifest(filename)
      cur=Dir.getwd
      File.open("#{cur}/#{filename[0,3]}/xml/AndroidManifest.xml","r") do |infile|
        while(line=infile.gets)
          if line[/android.permission./]
            line=line.strip
            line=line[50,150]
            line=line.strip
            line=line.squeeze(" ")
            line=line.chop.chop.strip
            line=line.chop
            puts line
            open(cur+"/#{filename[0,3]}/permissions.txt","a") do |f| f.puts (line+"\n") end
          end
        end
      end
      puts ("\nPermissions Stored at #{cur}/#{filename[0,3]}/permissions.txt\n")
    end
end
```

Appendix A – Tools

Dex2jar can be found at: http://code.google.com/p/dex2jar/

Apktool can be found at: http://code.google.com/p/android-apktool/

Java Decompiler can be found at: http://code.google.com/p/android-apktool/

References:

Mcafee 2011 Threats Predictions: http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2011.pdf?cid=WBB005