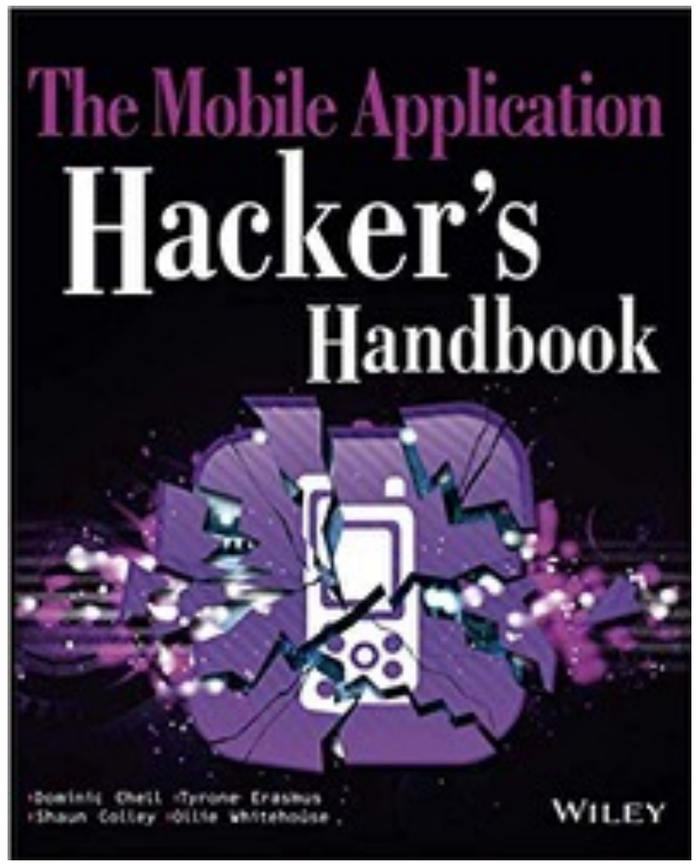
CNIT 128
Hacking Mobile Devices



9. Writing Secure Android Apps

Updated 11-14-22

Common Vulnerabilities

- Code injection
- Logic flaws
- Insecure storage
- Application configuration
- Insecure communication
- Logging

Topics

- Principle of Least Exposure
- Essential Security Mechanisms
- Advanced Security Mechanisms
- Slowing Down a Reverse Engineer

Principle of Least Exposure

Minimizing Attack Surface

- Find all entry points
  - Code exposed to inputs from outside sources
- Remove unnecessary entry points
- Perform security checks at necessary entry points

App Components

- Don't export more components than required
- The safest case is shown below
  - Most apps require some integration with other apps

```
dz> run app.package.attacksurface com.myapp.secure

Attack Surface:

  1 activities exported

  0 broadcast receivers exported

  0 content providers exported

  0 services exported
```

Data Storage

- Avoid storing unnecessary data
  - Such as passwords!
  - Private directory is protected somewhat by the sandboxing
  - SD card is less protected

Untrusted Sources

- Inputs from SD card, Internet, Wi-Fi, Bluetooth, etc.
- Verify authenticity with signature, encryption, or some other validation
- Be careful loading classes or running executables from untrusted locations
- Cryptographic protections are the best

Minimal Permissions

- Request the fewest permissions needed for your app
- This is safer, and also avoids worrying careful users
- Avoid risky permissions
  - **INSTALL_PACKAGES**
  - Using powerful shared users such as **android.uid.system**

Bundling Files in the APK

- APK can contain extra files by accident
  - May contain SSH credentials or other secrets

Essential Security Mechanisms

Review Entry Points

- Trace these functions

Table 9.1 Methods per application component that receive data from other applications

COMPONENT	METHOD
Activity	`onCreate()`
Broadcast Receiver	`onReceive()`
Content Provider	`query() insert() update() delete()openFile()`
Service	`onStartCommand()onBind()`

Permission Protection

- Exported components should be limited with permissions
  - Only available to apps with the same signature
- If you really want to offer a component for public use
  - Great care is required in the implementation

Securing Activities

Task Manager Snooping

- Remove your app from the recent app list
  - To avoid exposing private information on that image
- Put this code in **OnCreate()** to show a blank screen in the list

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_SECURE);
```

- Set this attribute in an activity to remove it entirely from the list

```
intent.addFlags(Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
```

Tapjacking

- Prevent touches from being sent through elements with this attribute:

```
android:filterTouchesWhenObscured="true"
```

- Or by using this method:

```
view.setFilterTouchesWhenObscured(true);
```

Dictionary

- Disable additions to the dictionary to keep passwords and other secrets out
- Add this attribute to an **EditText** box:

```
android:inputType="textVisiblePassword"
```

Fragment Injection

An activity can contain smaller UI elements named fragments. They can be thought of as “sub activities” that can be used to swap out sections of an activity and help facilitate alternate layouts for different screen sizes and form factors that an Android application can run on.

- https://securityintelligence.com/new-vulnerability-android-framework-fragment-injection/

Fragment Attacks

- Fragments are small UI elements that customize activities
  - But fragment injection vulnerabilities were found
- Since Android 4.4, fragments are blocked by default
- Use this code to allow a whitelist of fragments:

```
@Override
protected boolean isValidFragment(String fragmentName) {
    String[] validFragments =
                    {"com.myapp.pref.frag1",
                    "com.myapp.pref.frag2"};
    return Arrays.asList(validFragments).
contains(fragmentName);
}
```

Secure Trust Boundaries

- Make sure there's no way to open an authenticated activity from unauthenticated areas of the app
- One way: implement an app-wide authentication variable

Masking Password Displays

- Add this attribute to an **EditText** box:

```
android:inputType="textPassword"
```

Browsable Activities

- Can be used directly from a web browser
- High-value targets for attackers
- Avoid using **BROWSABLE**
- If you use it, consider all possible intents that could cause actions in your app

Securing Content Providers

Default Export Behavior

- Prior to API 17, content providers were exported by default
- To prevent this, put this code in the manifest:

```
<provider
android:name=".ContentProvider"
android:authorities="com.myapp.ContentProvider"
android:exported="false" >
</provider>
```

SQL Injection

- Use prepared statements, like this:

```
String[] userInput = new String[] {"book",
"wiley"};
Cursor c = database.rawQuery("SELECT * FROM
Products WHERE type=?
AND brand=?", userInput);
```

Directory Traversal

- The **getCanonicalPath()** method removes **..** characters and provides the absolute path to a file
- The code on the next page uses this to limit paths to the **/files/** subdirectory of the app's private data directory

```
@Override
public ParcelFileDescriptor openFile (Uri uri, String mode)
{
    try
    {
        String baseFolder = getContext().getFilesDir().getPath();
        File requestedFile = new File(uri.getPath());

        //Only allow the retrieval of files from the /files/
        //directory in the private data directory
        if (requestedFile.getCanonicalPath().startsWith(baseFolder))
            return ParcelFileDescriptor.open(requestedFile,
                    ParcelFileDescriptor.MODE_READ_ONLY);
        else

return null;
    }
    catch (FileNotFoundException e)
    {
        return null;
    }
    catch (IOException e)
    {
        return null;
    }
}
```

Pattern Matching

- Pattern-matching checks may fail for variations of the path
- Link Ch 9a

There are three different attributes that can be set to control which subset of the data the permissions apply to:

```
android:path="/subpath"
```

applies the permissions to entries within **`/subpath`**, but **not** to subdirectories of **`/subpath`**,

```
android:pathPrefix="/subpath"
```

applies the permissions to entries within subdirectories of **`/subpath`**, and

```
android:pathPattern
```

allows the use of wildcards to match the content URIs for which the permissions apply.

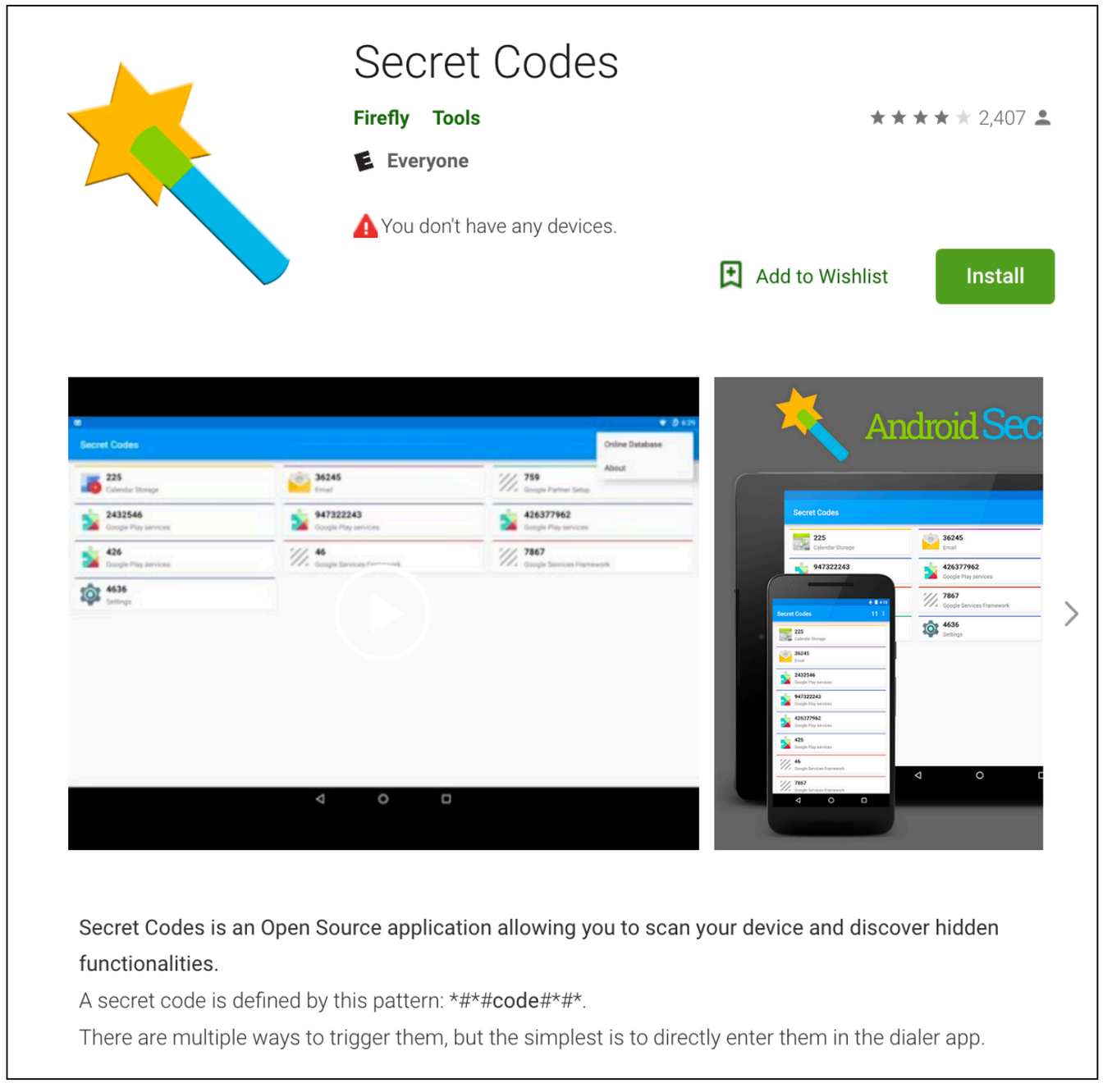
Secret Codes

```
<receiver android:name=".receiver.DiagnoserReceiver">
    <intent-filter>
        <action android:name="android.provider.Telephony.SECRET_CODE"/>
        <data android:scheme="android_secret_code" android:host="111222"/>
    </intent-filter>
</receiver>
```

- Defined in Android Manifest
- http://blog.udinic.com/2013/05/17/create-a-secret-doorway-to-your-app/

Securing Broadcast Receivers

- Secret codes are easily enumerated using apps on the Play Store
- Don't trust them



CNIT 128 Ch 9a

Storing Files Securely

Creating Files and Folders Securely

- Explicitly set permissions

```
FileOutputStream secretFile = openFileOutput("secret",
                                    Context.MODE_PRIVATE);
```

```
File newdir = getDir("newdir", Context.MODE_PRIVATE);
```

Encryption

- Use AES for symmetric encryption, avoid ECB
- Use RSA-2048 for asymmetric encryption
- Password hashing advice in textbook is wrong
  - You need salting and stretching; better to avoid doing it yourself

Random Numbers

- Random() produces the same series of numbers each time it's run from the same seed
- SecureRandom is better
- Java provides methods to seed it from a source of entropy

```
SecureRandom sr = new SecureRandom();
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(256, sr);
SecretKey key = generator.generateKey();
```

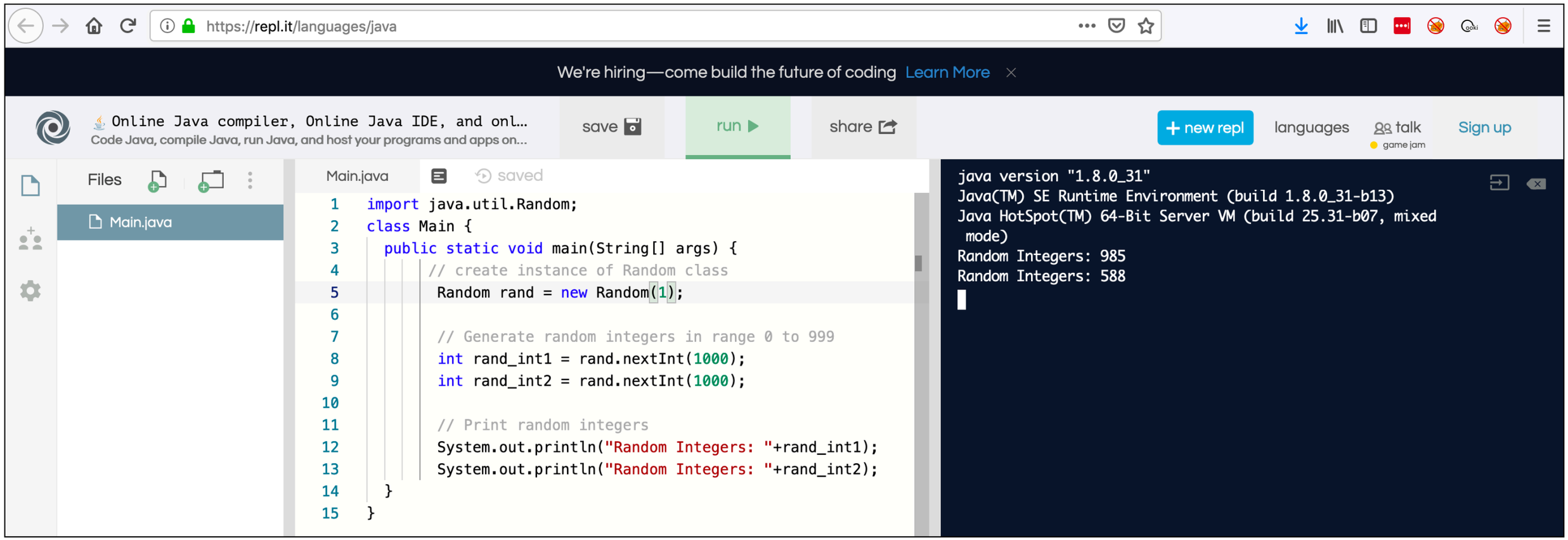
Random()

```
import java.util.Random;
class Main {
  public static void main(String[] args) {
        // create instance of Random class
         Random rand = new Random(1);

        // Generate random integers in range 0 to 999
        int rand_int1 = rand.nextInt(1000);
        int rand_int2 = rand.nextInt(1000);

        // Print random integers
        System.out.println("Random Integers: "+rand_int1);
        System.out.println("Random Integers: "+rand_int2);
  }
}
```

Online Java Tester



- replit.com
- Every run produces the same numbers

Key Generation

- PBKDF2 uses many rounds of hashing to derive a key from a password
- Key should be stored in Android Keystore

Exposing Files

- To allow specified other apps to see a file
- Those apps need **com.myapp.docs.READWITE** permission
- They can only access the **/document/** folder

```
<provider
    android:name=".DocProvider"
    android:authorities="com.myapp.docs"
    android:exported="true"
    android:permission="com.myapp.docs.READWRITE"
    android:grantUriPermissions="false">
        <grant-uri-permission android:pathPrefix="/document/" />
</provider>
```

Secure Communications

HTTPS

- HTTP is very unsafe
- HTTPS is much better, but depends on trusted Certificate Authorities (CAs)
- ***Certificate pinning*** makes HTTPS even more secure
  - Requiring a specific certificate or CA

Local Communications

- Transferring data from one app to another
- Android API is the best method
  - Activities with intent-filters
    - In more recent Android versions
      - ChooserTargets, Shortcuts, direct share targets
- Using network sockets or the clipboard is less safe

Securing WebViews

WebView

- Lets you display a Web page in an activity
- Often leads to security problems
- **Use HTTPS**
  - Contemt loaded over HTTP is subject to interception and modification
- **Disable JavaScript**
  - If you aren't using it

WebView

- **Don't Use JavaScriptInterface**
  - If you do use it, target an SDK >= 17
- **Disable Plug-ins**
  - They're deprecated for Android 4.3 and higher
- **Disable Filesystem Access**
- **Validate Web Content**

Configuring the Android Manifest

Backups and Debugging

- If **android:allowBackup** is false, an attacker can't back up files with physical access to the device
- **android:debuggable** allows debugging

API Version Targeting

- **minSdkVersion** should be as large as possible
  - Lower values remove new security fixes
  - Values below 17 export content providers by default

Android 9

- Targeting SDK 28+ gives you
  - DNS over TLS
  - Network TLS by default
    - Cleartext traffic must be explicitly set
  - Separate WebView directories for each process
    - Can't steal cookies

Enable Private DNS with 1.1.1.1 on Android 9 Pie

8/16/2018, 8:01:15 AM PDT

Stephen Pinkerton

- https://blog.cloudflare.com/enable-private-dns-with-1-1-1-1-on-android-9-pie/

Logging

- Should be disabled in release builds
- Use a centralized logging class
  - So it can be easily disabled
- ProGuard can remove logging code

Native Code

- Notoriously difficult to secure
- Limit its exposure to the outside world
- Enable exploit mitigations
  - Use latest NDK version

```
$ ./checksec.sh --file libtest.so

RELRO        STACK CANARY   NX           PIE   RPATH      RUNPATH      FILE

Full RELRO   Canary found   NX enabled DSO   No RPATH   No RUNPATH   libtest.so
```

Exploit Mitigations

- RELRO: Relocation Read-Only
  - Prevents GOT rewrites
- RPATH / RUNPATH
  - Allows attacker to load modified libraries from a user-controlled path
- Link Ch 9f

Advanced Security Mechanisms

Protection Level Downgrade

- A malicious app can define a permission first with an insecure protection level
- So your app inherits that level
- Your app can check to make sure the protection levels are intact at each entry point

Protecting Non-Exported Components

- Attacker with root permissions can interact with them
- You can add a request token to prevent that
  - Randomly generated
  - Stored in a **static** variable in memory
  - Intents must have this token to run

Slowing Down a Reverse Engineer

Obfuscation

- ProGuard -- free but very ineffective
- DexGuard -- paid version of ProGuard
- Dash-O is good but expensive ($3000)
- Arxan is another

Root Detection

- Search for **su**
- See if **default.prop** allows ADB shell to run as root
- See if **adbd** is running as root
- Look for packages with names like
  - **SuperSU** or **Superuser**

Emulator Detection

- Check for emulator build properties

```
Build.TAGS = "test-keys"

 Build.HARDWARE = "goldfish"

 Build.PRODUCT = "generic" or "sdk"

 Build.FINGERPRINT = "generic.*test-keys"

 Build.display = ".*test-keys"
```

Debugger Detection

- Attacker may have modified your app or the environment to allow debugging

You can perform a check to make sure that the application is not set as debuggable by implementing the following code:

```
boolean debuggable = (getApplicationInfo().flags &

ApplicationInfo.FLAG_DEBUGGABLE) != 0;
```

Tamper Detection

- Check signature

```
public boolean applicationTampered(Context con)
{
    PackageManager pm = con.getPackageManager();
    try
    {
        PackageInfo myPackageInfo = pm.getPackageInfo(con.getPackageName(),
                                    PackageManager.GET_SIGNATURES);

String mySig = myPackageInfo.signatures[0].toCharsString();

        //Compare against known value
        return !mySig.equals("3082...");
    }
    catch (NameNotFoundException e)
    {
        e.printStackTrace();
    }
    return false;
}
```

CNIT 128 Ch 9b