Safe Coding Practices Leveraging A Suite of Android Application Analysis Tools

Ryan Johnson, Angelos Stavrou
Kryptowire & George Mason University

Introduction

- Google Play (formerly Android Market) has experienced significant growth
  - Wikipedia claims Google Play currently has 800,000+ applications as of February, 2013
- Proliferation of third-party application websites
  - Can contain repackaged applications that have malicious functionality and pirated applications
  - May have little or no application vetting
- A deeper inspection of Android applications is warranted

Android Application Analysis approaches

- Static Analysis
  - Enumerate API calls
  - Permission analysis
- INTENTsify
  - Proper intent usage
  - Identify target of inter/intra-app communication
- Dynamic Analysis
  - Execute app in custom emulator using forced-path execution
  - Log parameter values to sensitive API calls
- Manual Analysis and Application instrumentation
  - Look at the code
  - Directly modify application code
  - Dump state of variables

Android Permissions

- Android applications use a permission model
  - Applications request permissions to sensitive resources that are accessed through Android API calls, Intents, and content providers which may require one or more permissions
  - Users may not be familiar with the functionality associated with each Android permission

- Application installation
  - Installation is done on an all or nothing basis in regard to an application's permissions
    - Cannot selectively deny permissions
  - Permission set stays static after installation

Android API

- Android API documentation is incomplete
  - Many API calls that require a permission do not indicate this fact on the Android Developers website

- There is a mapping from certain API calls to the Android permissions
  - Some API calls require no permission
  - Other API calls may require one or multiple permissions

- Android permissions are requested in the AndroidManifest.xml file
  - Certain permissions require a system process ID

Identifying API calls

- Enumerate all Android permissions
  - android.Manifest.permission class contains all permissions names as values of the class' constants
  - New permissions sometimes can be added with each new release of the Android OS
- Utilize established mappings from researcher
  - Berkeley researchers released a mapping
    - Also released a tool named Stowaway
  - University of Toronto researchers released an even more complete mapping from their tool named Pscout
  - Android API calls, Intents, constants, and content providers

Static Analysis Program

- Android permission mapping
  - Used Berkeley's mapping and Pscout mapping
- Process
  - Extract permissions requested from the manifest
  - Disassemble application with apktool into smali
  - Parse the smali files for permission-protected API calls and string literals
  - Record any discrepancy between requested and used permissions depending on application functionality
  - Generate output for analysis
- False positives and false negatives
  - Calls may reside in dead code
  - Calls may reside in a binary

Static Analysis Program

- Important method calls
  - Reflection
  - Commands
  - Libraries/classes loaded
  - File access
  - Media events
  - Telephony events
  - Network activity
  - Intents/broadcast receivers

Demonstration

INTENTsify Program

- Focuses on intent (mis)usage
- Static analysis with partial execution of code
- Look for possible vulnerabilities with intents
- Two main types of issues
  - Unprotected components – other applications can launch these components
  - Possible hijacking – using an implicit Intent using an action string which can cause a collision

INTENTsify Program

- Unprotected Activity/Service/Receiver
  - Components can be launched from outside the application
  - Needs to have an intent filter declared
    - Sets "exported" to true
  - Two possible fixes
    - Specifically setting "exported" to false
    - Use custom permission for component

INTENTsify Program

- Possible Broadcast/Service/Activity hijacking
  - Occurs during implicit intent calls
  - Malicious eavesdropping
  - Extract information from intent
  - Intercept intent that was sent out and send back maliciously crafted data
  - Possible fixes
    - Always use explicit calls when possible
    - Again, custom permissions

INTENTsify Program

- Internal Implicit Intent
  - Always use explicit intents for internal app calls
- System broadcast receiver without check
  - Special type of broadcast injection where the receiver is set up to only handle protected system broadcasts
  - Action string should be checked
  - Explicit call can still be made to the receiver
    - This may lead to unexpected behavior

INTENTsify Program

- Not every issue is dire in nature
- Determine if component/intent call is responsible for sensitive data/functionality
- Rule of thumb: always use explicit intents if possible

INTENTsify Program Framework

- Statically search for Intent creation
  - Once found, execute code using custom emulator until the Intent is sent or the method returns
  - If the Intent is stored in a static/instance variable, then find where the static/instance variable is referenced and continue execution from there
  - If parameter to Intent creation is a parameter to the method call which contains Intent creation, locate that method and start execution from there
    - Can backtrack methods to a user-set number of levels back
  - Examine Intent object at the time it is sent to determine if it is an explicit or implicit Intent

INTENTsify Program Framework

- Parse the AndroidManifest.xml file
  - Enumerate each application component
  - Check to see if each is has the exported tag set to false and check to see if a custom permission is required
  - The Launcher component is always exported
  - If an application component has IntentFilter(s), then the component is exported by default unless the exported tag is set to false
  - If an application component does not have any IntentFilters, then the component is not exported by default

INTENTsify Program Demo

Demonstration

Dynamic Analysis Framework

- Code analysis framework that performs forced-path execution of Android applications using commodity hardware
- Operates on an APK file and does not require source code for the application
- Uses custom emulator and does not utilize the emulator or an Android device
- Creates logs of parameter values to sensitive API calls, control flow graphs, and method call graphs

Motivation

- Forced-path execution stresses application code and can reveal hidden functionality
  - Some branches require very specific conditions
  - Attempts to enter all branches if time is available
- Controls the result of the evaluation of conditional and switch statements
- Log parameters to sensitive API calls
  - Examining the parameters on a granular level will reveal intent and provide context to the call

```java
private static void executeCommand(String command) {
    try {
        Runtime.getRuntime().exec(command);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Categories of sensitive API calls

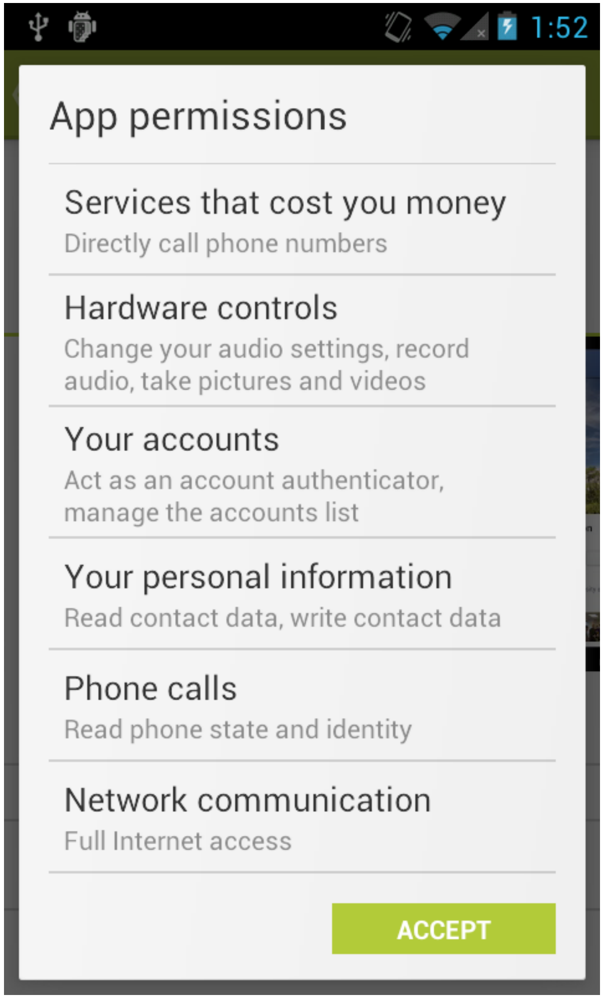
- 496 total API calls from the categories below
  - Reflection – target of reflective calls
  - Command execution – su, rm, nc
  - Network I/O – creation of sockets, data transfer
  - JNI calls – native calls
  - File I/O – file reads and writes
  - Media events – taking pictures, movies, etc.

Categories of sensitive API calls cont.

- 496 total API calls from the categories below
  - Media events – taking pictures, movies, etc.
  - Telephone events – Android ID, text messages
  - Crypto events – Key values, plain/ciphertext
  - Libraries loaded – name of library files
  - Content Providers – Databases usage, queries
  - Sending of Intents – Inter/intra-app communication
  - Location events – GPS usage
  - NFC events, Bluetooth events, etc.

Android Permissions again

- A permission may be used benignly or maliciously depending on expected behavior
  - SEND_SMS permission can be used to send SMSs to premium numbers or solely to send a thank you SMS to the user for purchasing the application
  - INTERNET permission can be used to download ads or to download an undesirable binary



Analysis Framework

- Use baksmali to obtain smali files
  - Smali is human-readable Davlik assembly
- Davlik bytecode contains 226 opcodes
- Developed a Java implementation for each Dalvik instruction
- Parse and obtain information from application's AndroidManifest.xml file
- Iterate through each application component
- Call any Java API calls and third-party libraries using reflection
  - Android API contains a subset of the Java API
- Model execution using a binary tree

Java source code and Dalvik bytecode

```
for (int i = 0; i < 3; i++) {
        if (i == 4) {
                System.out.println("Will never be reached");
        }
}
```

```
const/4 v0, 0x0
:goto_0
const/4 v2, 0x3
if-lt v0, v2, :cond_0
return-void
:cond_0
const/4 v2, 0x4
if-ne v0, v2, :cond_1
sget-object v2, Ljava/lang/System;->out:Ljava/io/PrintStream;
const-string v3, "Will never be reached"
invoke-virtual {v2, v3}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
:cond_1
add-int/lit8 v0, v0, 0x1
goto :goto_0
```

Moderating API calls

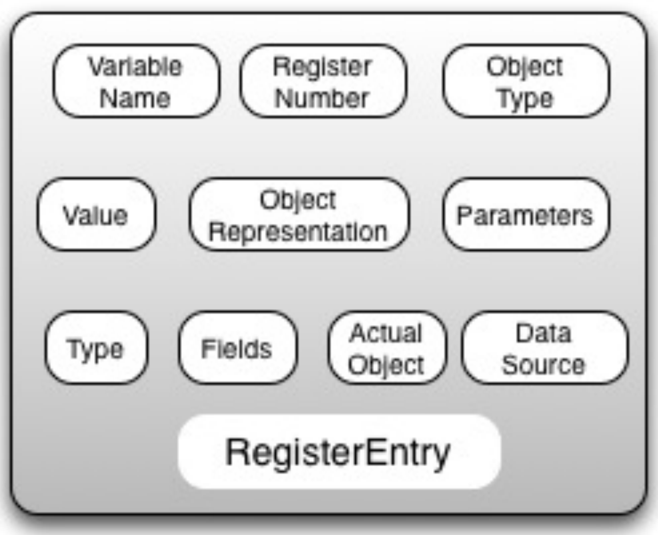
- Contains list of sensitive API calls
- During logging, the call can be blocked from executing or have its parameters changed prior to execution
- Calls that are always blocked
  - java.lang.Runtime.exec(*)
  - java.lang.System.setProperties(*)
  - java.util.concurrent.CountDownLatch.await()
  - java.lang.Runtime.exit(int) and the like
  - java.io.File.delete()
  - Calls that can block indefinitely
- Recursively traverse Method objects to reflective calls until ultimate target is found
  - Reflection may be used to call a reflective call and be embedded any number of levels deep

Bounding Execution

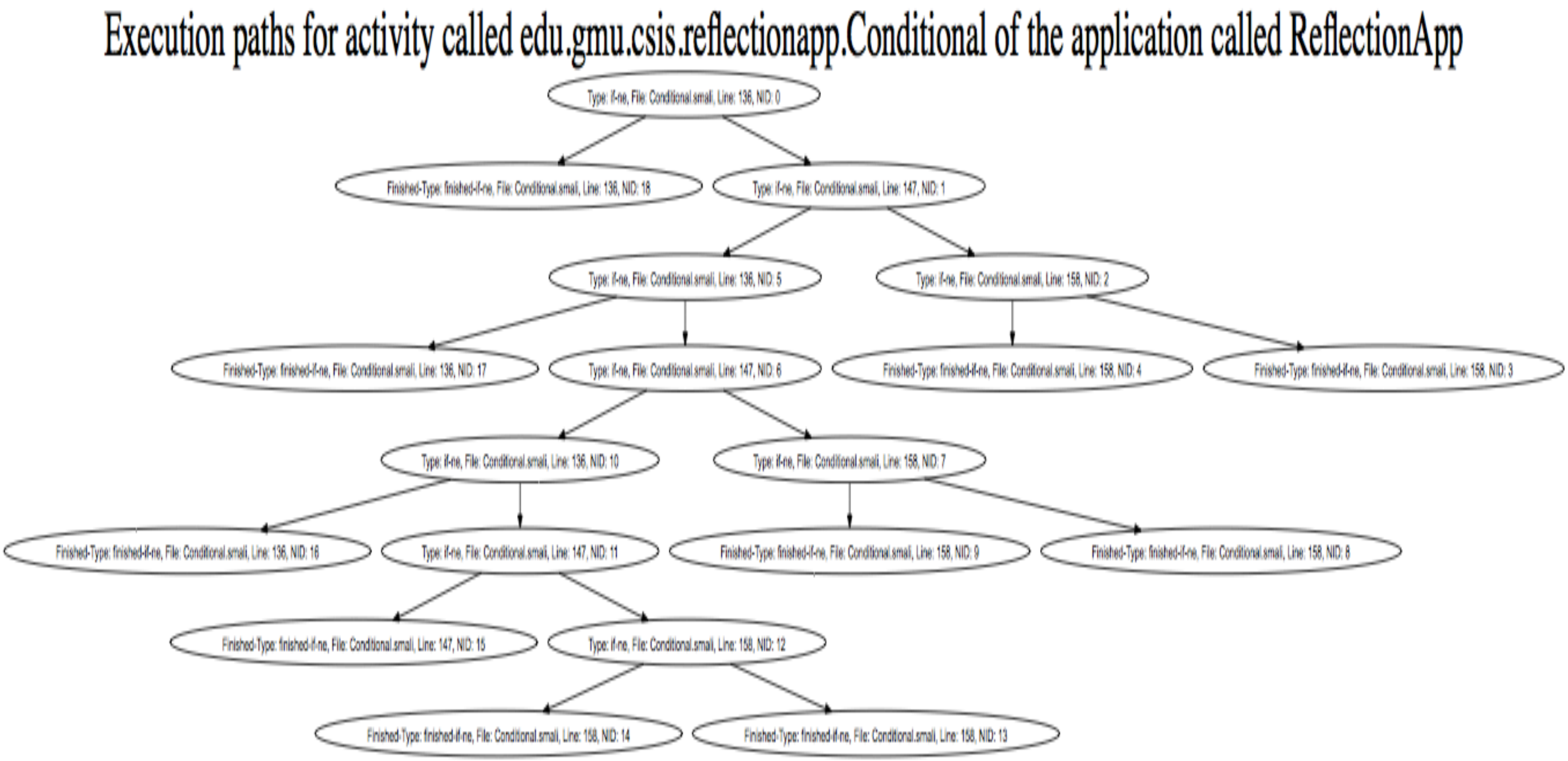
- Set upper limit to number of loop iterations
  - Or let iterate as conditions dictate
- Set upper bound to depth of recursive calls
- Detect infinite loops
  - Only iterate once and then exit
- Make attempt to detect infinite loops
- Set a time limit to strictly bound execution
- Improve performance while reducing precision

Representing Registers

- Davlik uses a register-based architecture as opposed to a stack-based architecture
- Objects and primitive data types are referenced by a register number
- Use custom data type to emulate Dalvik registers

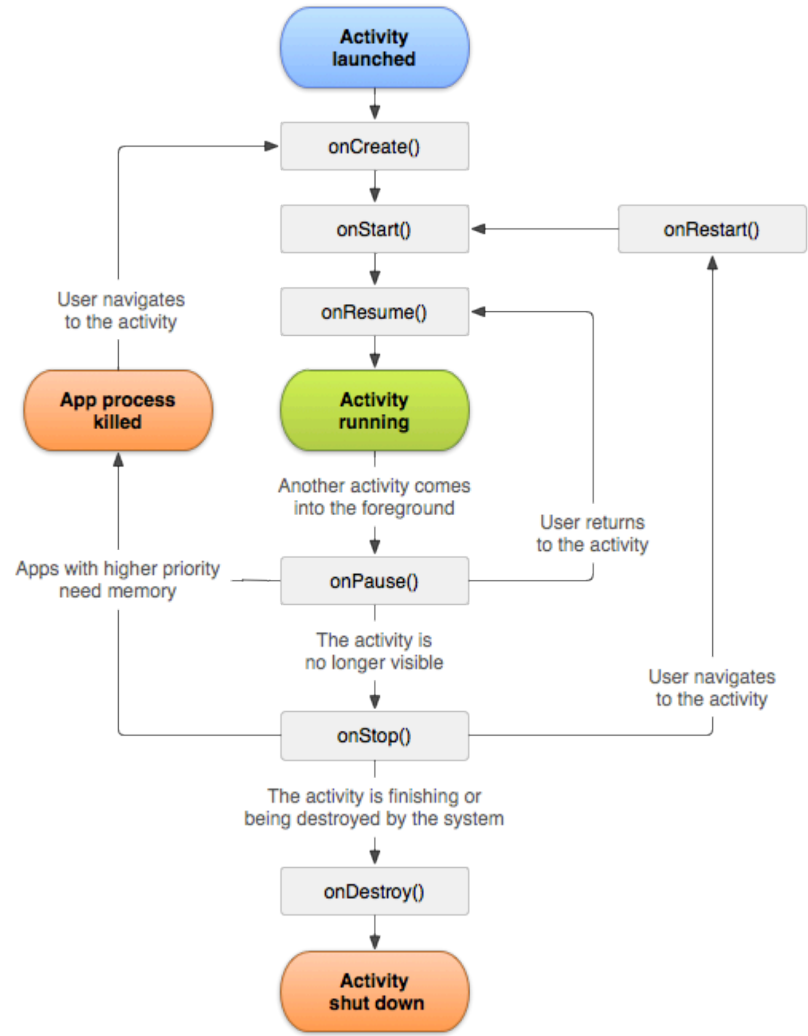


Modeling Execution



Abstracting User Input from an event-driven OS

- As application registers event listeners, immediately execute the corresponding code
- Force execution into callback methods (onResume, onLowMemory, etc.) to mimic component lifecycle
- Return static or random value for methods to obtain user-input for android.widget.TextView and its subclasses



Dynamic Analysis Program Demo

Demonstration

Case Study of Major Carrier's Mobile Application

- Output below shows fully qualified API call, parameter values, file in which API call occurs, and line number

javax.crypto.spec.SecretKeySpec.SecretKeySpec(byte[], java.lang.String)

Key (byte representation): 68 36 36 76 64 115 84 74 48 117 82 110 69 121 33 50

Key (String representation): D$$L@sTJ0uRnEy!2

Algorithm: AES

File name: redacted/smali/com/redacted/redacted/util/codec/DSSHtmlEncryption.smali

Line number: 254

javax.crypto.spec.IvParameterSpec.IvParameterSpec(byte[])

Initialization Vector (byte representation): 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48

Initialization Vector (String representation): 0000000000000000

File name: redacted/smali/com/redacted/redacted/util/codec/DSSHtmlEncryption.smali

Line number: 112

Case Study of Major Carrier's Mobile Application

- Key and IV value may be randomly generated or hard-coded

- Hard-coded keys are generally bad programming practice since they can be extracted from the application code and then used by an adversary

- Generating a random key and then using a secure key exchange protocol is the preferred way to exchange sensitive keying material

Case Study of Major Carrier's Mobile Application

- Both are implemented as final static variables in com.redacted.redacted.util.codec.DSSHtmlEncryption.java
  - Setting key value (smali format)

```
const-string v0, "D$$L@sTJ0uRnEy!2"

sput-object v0, Lcom/redacted/redacted/util/codec/DSSHtmlEncryption;->key:Ljava/lang/String;

sget-object v0, Lcom/redacted/redacted/util/codec/DSSHtmlEncryption;->key:Ljava/lang/String;

invoke-virtual {v0}, Ljava/lang/String;->getBytes()[B

move-result-object v0

sput-object v0, Lcom/redacted/redacted/util/codec/DSSHtmlEncryption;->keyValue:[B
```

- Smali is a human-readable assembly language for Dalvik

Case Study of Major Carrier's Mobile Application

- Setting of IV value shown below (smali format)

```
const/16 v0, 0x10

new-array v0, v0, [B

fill-array-data v0, :array_0

sput-object v0, Lcom/redacted/redacted/util/codec/DSSHtmlEncryption;->ivParams:[B

return-void

:array_0
.array-data 0x1
    0x30t
    0x30t
    0x30t
    0x30t
    ……….
```

Case Study of Major Carrier's Mobile Application

- Found login tokens in /data/data/redacted/shared_prefs

root@android:/data/data/redacted/shared_prefs # cat saveLoginDetails0.xml

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>

<map>

<string name="savedPassword">Skw9nzgmyaD4mPIUQguk3N+MW8vkbfd5oxTEaBwPC+k4InlIjr+HWDNaDuJGOe9W</string>

<string name="loginMode">Wireless</string>

<string name="savedWirelessNum">2028675309</string>

</map>

Case Study of Major Carrier's Mobile Application

- Tried to decrypt login token with hard-coded key and IV but it does not decrypt properly

- <string name="savedPassword">Skw9nzgmyaD4mPIUQguk3N +MW8vkbfd5oxTEaBwPC+k4InlIjr+HWDNaDuJGOe9W</string>

- Followed decryption process in DSSHtmlEncryption.java
  - Create AES key and IV value with hard-coded values
  - Decode savedPassword from Base64 String
  - Perform decryption
  - Doesn't decrypt properly (i.e., encrypted with different key)
  - Key appears to reside on Carrier's server

Case Study of Major Carrier's Mobile Application

- com.redacted.redacted.activity.login.LoginUnifiedActivity is the application component that uses the hard-coded credentials

- Occurs as a callback from startActivityForResult() API call
- Enumerate the activity application components that are called from LoginUnifiedActivity with startActivityForResult()
  - com.redacted.redacted.activity.login.UpdatePasswordActivity
  - com.redacted.redacted.dialog.DialogActivity
  - Intent action:android.intent.action.VIEW with URI of http://redacted.com/redacted
- Tracing back calls leads to the code being reachable from com.redacted.redacted.activity.login.UpdatePasswordActivity

Case Study of Major Carrier's Mobile Application

- Due to the nature of forced-path execution, we need to ensure that conditions actually exist to exercise the portion of code with the hard-coded credentials
- We utilize application repackaging to insert code to print the value of variables and to denote that portions in the code are actually reached
- Application repackaging can occur maliciously where a legitimate application is modified to infiltrate data and perform malicious activities as it masquerades as the legitimate app

Malicious / Rogue Mobile Apps - Defined

- Rogue mobile apps can be best defined as follows:
  - Created by non-authorized individuals or entities
  - Seek to confuse consumer to believe it is published from an authorized source – similar name, use of logo, or similar publisher
  - Similar to other applications but its objectives are to compromise other apps on the device

- Malware mobile apps have different objectives:
  - Similar to desktop malware or viruses – device disabling
  - Data syphon – attempt to steal device data and PII information to third parties
  - Man in the middle – serve as a proxy - behavior to end user is seamless, credentials are taken

Case Study of Major Carrier's Mobile Application

- The com.redacted.redacted.util.Logger class contains various logging methods which are called throughout the program
- Examination of these methods shows that these methods do not do perform any logging, except for when an error occurs
- These methods likely wrote to the Android OS log during development but this was removed from the production code
- We inserted code to write to the Android OS log in these methods to glean information about the application and various other locations in the code

Case Study of Major Carrier's Mobile Application

- Below is an example of adding code (in red) to the log method to make it write its parameters to the Android OS log

```
.method public static log(Ljava/lang/String;)V
    .locals 2
    .parameter "msg"

    .prologue
    const/4 v0, 0x0

    const-string v1, "redactedrecomp-log"

    invoke-static {v1, p0}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

    invoke-static {v0, p0}, Lcom/redacted/redacted/util/Logger;->log(Ljava/lang/String;Ljava/lang/String;)V

    return-void
.end method
```

Case Study of Major Carrier's Mobile Application

- Android log showing username and password written to log in instrumented application. *The normal application does not do this*

D/redactedrecompilation-log(2934): LoginActivity.doPasswordLogin: Wireless number --->2028675309

D/redactedrecompilation-log(2934): LoginActivity.doPasswordLogin: Password --->redacted

D/redactedrecompilation-log(2934): LoginActivity.doPasswordLogin: Wireless number --->2028675309

D/redactedrecompilation-log(2934): LoginActivity.doPasswordLogin: Password --->redacted

D/redactedrecompilation-log(2934): PageCache: getPageAsStream /data/data/redacted.redacted.myWireless/files/cache/requests/getAuthentication/EN/500_0_Login_Simplified.xml

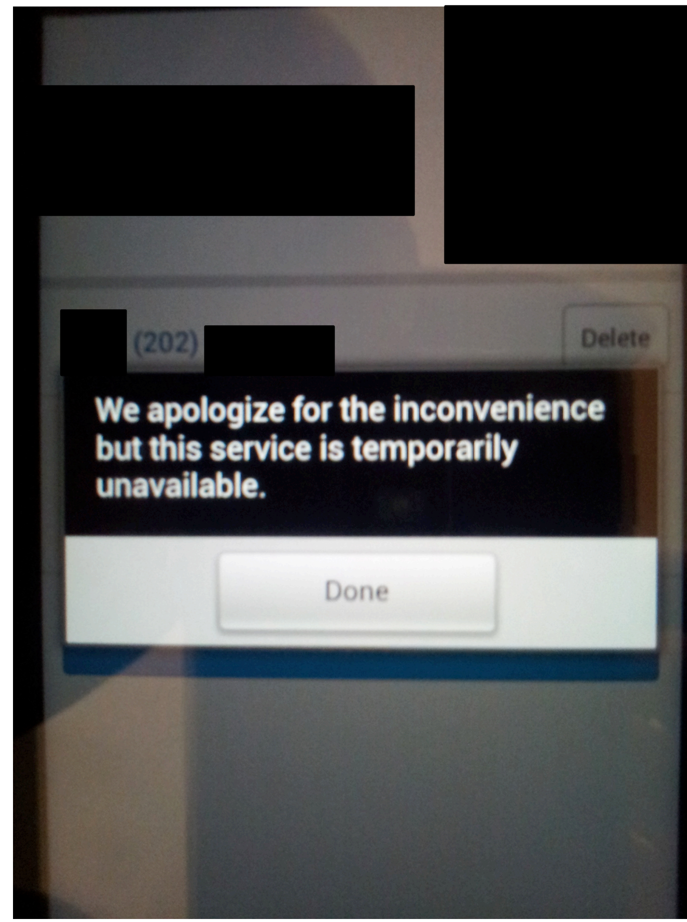
D/redactedrecompilation-log(2934): PageCache: getPageAsStream false

D/redactedrecompilation-log(2934): PageCache: opening cache/requests/getAuthentication/EN/500_0_Login_Simplified.xml

Case Study of Major Carrier's Mobile Application

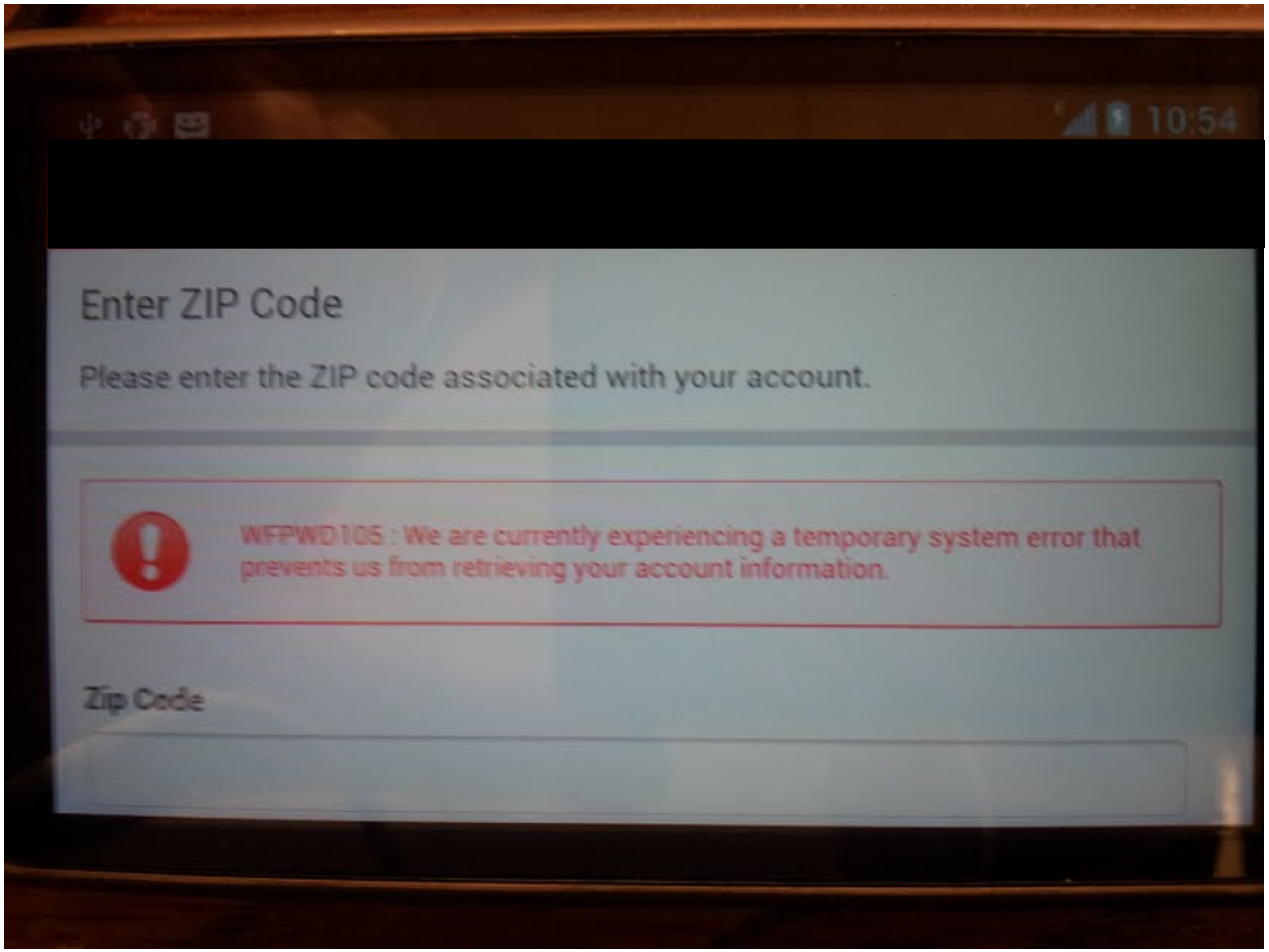
- We instrumented various portions of the code to see what is reached during manual testing of the application

- Save login credentials upon logging in
- Log out and rerun application
- Hit cancel for automatic login of saved credentials
- Navigate to Update Password
- Enter password and press update
- This service has been unavailable for 2 weeks or it will say the password does match even though this is to update the password



Case Study of Major Carrier's Mobile Application

- We also checked the forgot password option to see if it would trigger the code, but there was system error blocking the action



Case Study of Major Carrier's Mobile Application

- Without access to the password updating service and the forgot password service, we are not able to see what the hard-coded key and IV are used to decrypt

- These may be disabled on the server end, so even if a request comes in, it can be denied and the portion of code cannot be exercised

- We did not want to use automated UI testing since we do not want to make undesired changes to the Carrier account we were testing

Case Study of Popular Finance Application

- On the application's website, they claim that your data is safe even if you lose your phone
  - This is not true assuming you can bypass the screen lock
  - Can be done with if USB debugging is enabled
  - You can delete the pattern file programmatically

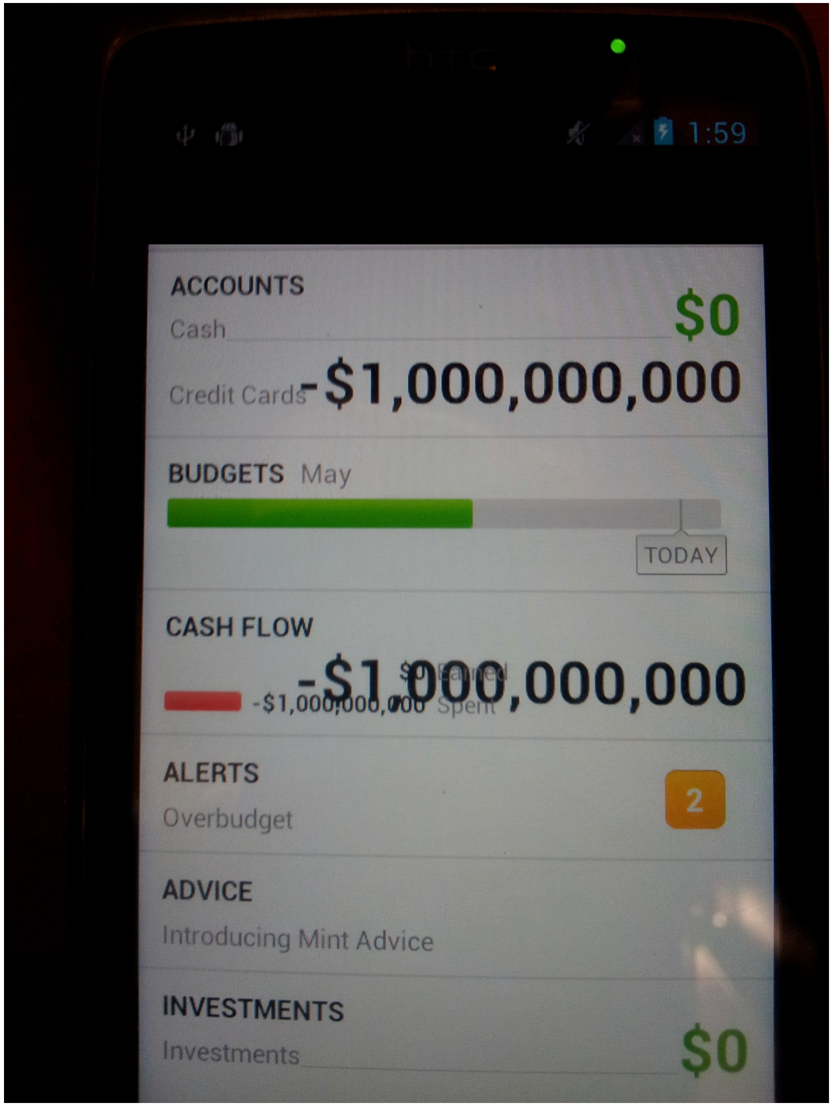
- All data stored using shared preferences
  - PIN, Login token, Redacted account number, etc.

Case Study of Popular Finance Application

- All data in the encrypted database
  - Transactions
  - Account details
  - Credit Card number (in a certain scenario)

- Created repackaged Redacted Android application that will look the same as the normal Redacted application but leak bank credentials
  - Trojan Apps are the most common vector of attack
  - Protecting the Application is the responsibility of the developer

Case Study of Popular Finance Application



- We accessed the encrypted database and made some changes using the SQLCipher API
- We modified the balance column in the account table
- We modified the expense column the in spending table

Case Study of Popular Finance Application

- Shared preferences is a mechanism to store persistent private data on a per-application basis
  - Finance app encrypts the key-value pairs upon insertion and decrypts them upon retrieval
  - Finance app also encodes the key values so they appear as a random double value (I.E., 25581291.80006)
  - Generally, keys appeared not to be stored but generated at runtime when needed

Case Study of Popular Finance Application

- Below are the contents of the shared preferences file for Finance app. Contains encrypted key-value pairs


```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<boolean name="progress" value="false" />
<string name="o4kbnBf6PHgv3jDZhdAq7w==">I+zQ6NJ2oet+2QyCU5BJ6w==</string>
<string name="UwpmMnT6MwN3YUBnAO90IA==">CRXlNEq6DmDiCSSO1uASrqQodsWG+gJIYgHSs8P
LaP0=</string>
<string name="a9yEOSD2b2gboa4Qb5lg9Q==">JFDX+I5pBWdBF91KYE+6EA==</string>
<string name="6Job4EH6sFUUrvOPHc1acQ==">u6ztZidAIg5I5EBlTwvMQw==</string>
<string name="1ebo0+6qLIYly7iQO8ISCg==">uO+sEEZqPuPuyt5llJ9yJA==</string>
<string name="e3oeiasyMIMgSaMsYLKH9w==">o+Yw21YC0BCEzGQ7KOrbAA==</string>
<string name="xrxpNHh0bsQPsNKyYQzxFw==">BIinwSZApnvPkGK1Ljomow==</string>
<string name="Bwoq4A64f0HD1NF0Z6S1rw==">R7VATmI8rR6NnTn7R4YelA==</string>
<string name="5YyNX3ZNwynfaDdLDBzdug==">k+GoFrD47p1yuYPBt6qhoA==</string>
<string name="SZSo4wZgg/EGHYim9nnD6A==">kvcBgD+tMTEheb61tDuUqw==</string>
<string name="X/yEZHfU8NJGKxqRHibkqw==">NiIuUv4X2R/qo9qmKF8iDA==</string>
<string name="TLdyJGKjVkrQc7D706I8yw==">Ew0Ey2ojH6YNmmZV2XdeLS53SWe4X63CLtlvi5G
7xmVm6vnKe7mwwAk4mBIvICgdENz3CvHoBZGIZGgV1E47eRreyIf/+ZCmIm+ZaZIVfF8=</string>
<string name="n8Ta65APIsDPi8C+MgVRkw==">MDWopa2YdM1/WsSQT/OADg==</string>
<string name="qQLLeahzLwFrhpjtrkjkVQ==">npHnmJD48TTApMRezb1pHsl/GkWcAYXnFc0SrP1
O3dOZsKj4cvNlVuv6PYHsLkQ/IZl4CbCdm35qvnz7bVWmUJLkyNz7p4NQQPDbpEyX/aovxkDAgQptiY
JTOnfW9J4d0HfZqMunYn4uoY8gMqt7jA==</string>
<string name="/h1e4aNkt4bI2sVe6Qd5tg==">2M9p/0M1pj0p9DUZaqHvEA==</string>
<null name="statusBarText" />
<string name="A8/lzuED8YNGGqV96XesXA==">tlXJV1XNT3P7Yq8a7k0e4spgEhrXe9HiTa6o6xn
1/Y0=</string>
</map>
```

Case Study of Popular Finance Application

- The key for the each device will always be the same
- The format of the key, in general, is the first 32 bytes of the String below
  - First 8 bytes of device id + ":" + android.os.Build.DEVICE + ":" next 7 bytes of device id + ":" + android.os.Build.MANUFACTURER + ":" + android.os.Build.MODEL + "@#$#@%#%#@%$@^"
- On test phone before truncation: 9a1588bf:bravo:3b1d8fbHTC:HTC Desire@#$#@%#%#@%$@^

Case Study of Popular Finance Application

- On our test phone the 32-byte key was the following
  - 9a1588bf:bravo:3b1d8fbHTC:HTC De
- Mimicking the application code, we generated a Java decryption routine to decrypt the key-value pairs in the shared preferences file

```
private static String [redacted]Decrypt(String toDecrypt) throws Exception {
    Cipher cipher = Cipher.getInstance("AES");
    byte[] keyBytes = "9a1588bf:bravo:3b1d8fbHTC:HTC De".getBytes();
    SecretKeySpec sks = new SecretKeySpec(keyBytes, "AES");
    cipher.init(Cipher.DECRYPT_MODE, sks);
    return new String(cipher.doFinal(Base64.decodeBase64(toDecrypt)));
}
```

Case Study of Popular Finance Application

- Using the decryption routine from the previous slide, we decrypted the entries in the shared preferences file and decoded the key values of the key-value pairs

- 107955989.43234 (token) - qjQsQv849IudGlIs1gD
- 67600920.12018 (passcode) – 9879
- 100066628.21724 (user id) – 65250291
- 115738811.1195 (guid) – 24B2C3F6FDAAFE9C
- 25581291.80006 (current version) – 1.5
- 8538329.56202 (last update date) – 1369626642534
- 214786401.18067 (RateMyApp Config) – 3;;1;;3;;7
- 166605410.64793 (pod cookie) – {"domain":"mobile.redacted.com","name":"redactedPN","path":"/","value":"9"}

Case Study of Popular Finance Application

- The finance app maintains an encrypted database that contains finance account information, bank account information, and account transactions

```
root@android:/data/data/com[redacted]/databases # ls -al
-rw-r--r-- app_74   app_74      68608 2013-05-26 23:50 encrypted[redacted].db
```

  - The database does not store bank credentials but has a column that could be used to do so
    - Table: fi_login – Column: blobCredentials

Case Study of Popular Finance Application

- Intermediate key has the format of the device ID + creation date + hard-coded literal String
- 9a1588bf3b1d8fb1369622564342!%$_C++J
- Device ID – android.provider.Settings.Secure.*getString(this.getContentResolver(), "android_id");* – 9a1588bf3b1d8fb
- Creation date – can be obtained from the timestamp

```
root@android:/data/data/com.[redacted]/files # cat timestamp ; echo
1369622564342
```

- Hard-coded String – in finance app's String table with a name of do_not_mess_with_me – !%$_C++J

Case Study of Popular Finance Application

- The first 32 bytes of the String are used as an AES key to encrypt the String !%$_C++J and the result will be the key that is input into the SQLCipher API

```
private static String getDBKey(String intermediate) throws Exception {
    byte[] input = null;
    if (intermediate.length() > 32) {
        byte[] intermediateBytes = intermediate.getBytes();
        input = new byte[32];
        System.arraycopy(intermediateBytes, 0, input, 0, 32);
    }
    else if (intermediate.length() < 32) {
        return "fail - needs to be at least 32 bytes in length";
    }
    else {
        input = intermediate.getBytes();
    }
    SecretKeySpec sks = new SecretKeySpec(input, "AES");
    Cipher cip = Cipher.getInstance("AES");
    cip.init(Cipher.ENCRYPT_MODE, sks);
    byte[] toEncrypt = "!%$_C++J".getBytes("UTF-8");
    byte[] ciphertext = cip.doFinal(toEncrypt);
    byte[] base64encoded = Base64.encodeBase64(ciphertext);
    return new String(base64encoded, "UTF-8");
}
```

Case Study of Popular Finance Application

- Utilize SQLCipher API as finance app does to access DB

```
private Object[] getAllTables() {
    SQLiteDatabase.loadLibs(this);
    ArrayList<String> tables = new ArrayList<String>();
    File encdatabaseFile = getDatabasePath("encrypted[redacted].db");
    SQLiteDatabase encdatabase = SQLiteDatabase.openOrCreateDatabase(encdatabaseFile, keyVal, null);
    Cursor c = encdatabase.rawQuery("select name from sqlite_master where type = 'table'", new String[0]);
    String whole = "";
    if (c.moveToFirst()) {
        do {
            whole = c.getString(0);
            tables.add(whole);
        } while (c.moveToNext());
    }
    encdatabase.close();
    Object[] star = tables.toArray();
    return star;
}
```

- Enumerate table names and query them individually
  - SELECT * FROM [TABLE_NAME];
- Table names and schemas also available in the file named db-init.sql in the finance app's assests folder

Case Study of Popular Finance Application

- The database has a few interesting tables
  - account – bank account info (balance, id, etc.)
  - transaction_bankcc – contains transaction data
  - fi_login – some login data (although placeholder appears to be present in db for bank credentials, value is always null for blobCredentials)

- CREATE TABLE `fi_login` (`id` bigint(15) NOT NULL PRIMARY KEY ON CONFLICT REPLACE,`status` int(10) NOT NULL, `lastUpdateDate` datetime, `fiName` varchar(255), `lastUpdated` varchar(255), `financeStatus` int(10), `errorMessage` varchar(255), `phone` varchar(32), `logo` varchar(128), `url` varchar(255),`csMessage` varchar(255),`csMessageLink` varchar(255), **`provideCredentials` int(1)**, **`blobCredentials` varchar**, `isManual` int(1));

Case Study of Popular Finance Application

- Another method exists to examine the contents of the database
  - Issue a few commands using ADB (Android Debugging Bridge)
  - /yourPath/android-sdk-mac_x86/platform-tools/adb shell setprop log.tag.SQLiteStatements VERBOSE
  - Does not require application instrumentation
  - Possibly requires PIN (if enabled) but does not require using SQLCipher API
  - Data from the local database can be observed when it is synced with the finance app's servers

Case Study of Popular Finance Application

- The last 4 digits of a credit card account number can be seen in the network traffic of an instrumented app
- Transactions using a credit card generally do not reveal the credit card account number
- We did notice one occurrence of the full credit card number in the encrypted database
  - Worst case scenario – A finance app user loses their cell phone and has their credit card number exposed

Case Study of Popular Finance Application

- Mobile Software Developers make mistakes…
  - Collect and/or Store PII information without notifying the End- User
  - Transmit PII information to their website or third parties
  - Enable other programs to get access to PII data
- Good Intentions but non-disclosed to the End-User
  - The application makes use of resources not disclosed to the user: Camera, GPS Location, Microphone, Read of PII (contacts, phone #s, IMEI, etc.)
  - Perform Functionality without explicit End-User permission

Conclusion

- Give your application the appropriate permission set
- Do not use hard-coded keys in an application
- Do not use keys that are predictable and easy to generate
- Obfuscate your application to make reverse engineering it more difficult (e.g., ProGuard)
- Use explicit intents and do not export application components unless necessary
- Remove sensitive logging from production app
  - Do not rely on android.util.Log.isLoggable(String, int)
- Be aware of application repackaging

Questions

Thank you!

Questions?