Lesson 6 – Client Code Quality Debugging and Logging

Introduction

This lesson focuses on vulnerabilities caused by bad client code quality resulting in having developer and debug options available in production releases of the application.

Android system log is a popular tool used by developers for debugging. Before Android 4.1.x (API 16) every installed application with READ_LOGS permission could have access to all logs. In API 16 it was changed, allowing application to only read its own logs. It has been implemented because of security concerns regarding reading other applications logs. Developers often log sensitive data, even passwords and whole communication with server. System log can still be accessed with Android Debug Bridge if the USB Debugging is enabled on the device. Also on rooted phones every application with root access can access whole system log.

USB Debugging is a feature of Android system which allows a connection between device and a computer with Android SDK through USB. This can allow to bypass the screen lock, install and uninstall any application, gain access to device OS shell and data. Before Android 4.2.2 (API 17) any computer connected to a device with USB Debugging enabled would automatically gain access to the debug features. Because of the risks of connecting the device to a malicious computer the Secure USB Debugging has been implemented in API 17. From this version after the first connection to the new computer, the user has to unlock the phone and explicitly accept the connected computer. There has been found a vulnerability that would allow to bypass the Secure USB Debugging feature and accept a computer on locked device[1]. This bug has been fixed in Android 4.4.3 (API 19).

Android applications have debuggable flag, which if set true allows the debugger to attach to an application, allows to use the USB Debugging to perform a heap dump, and run arbitrary code with application's permissions. It is set on by default on debug builds in Android Studio and it is disabled by default for release builds, but can be explicitly left on for release builds. In emulators and some unofficial Android ROMs (having ro.debuggable flag set to true) all applications are debuggable irrespective of build flags.

Getting started

You will need:
• a device or emulator with Android, enabled USB debugging.
• computer (Windows or Linux) with Android SDK desktop app installed.

Finding vulnerabilities

To see the logs you can use the shell or Android Device Monitor (a part of Android SDK). Connect the device to your computer. The shell the command is:

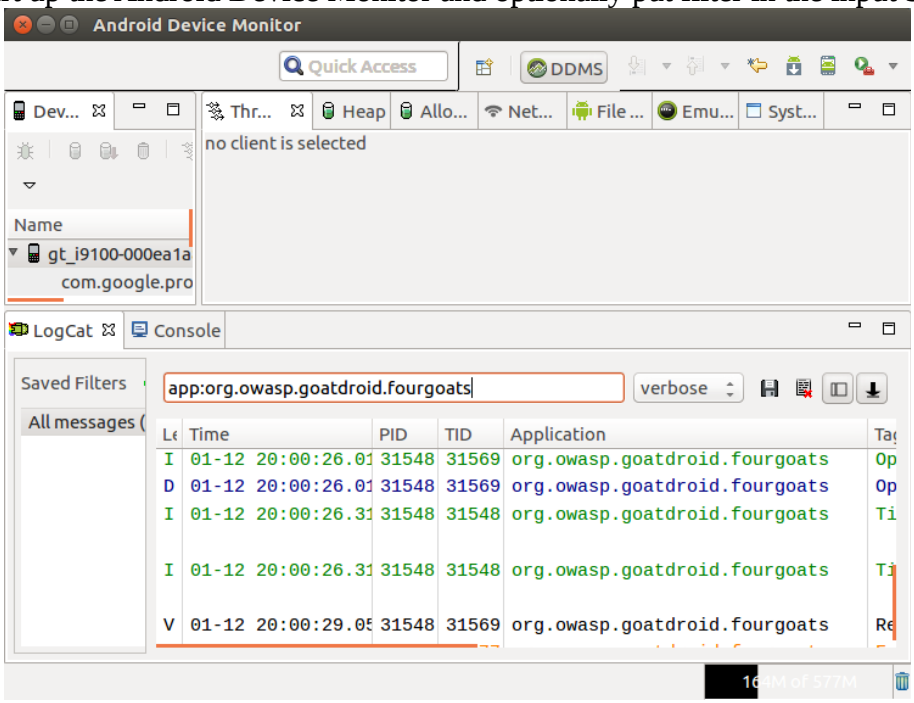
```
adb logcat
```

1 https://labs.mwrinfosecurity.com/advisories/android-4-4-2-secure-usb-debugging-bypass/

You can optionally run it with grep to limit the output only to the application you are interested with:

```
adb logcat | grep package.name
```

Or you can start up the Android Device Monitor and optionally put filter in the input box:



Start the application, carefully go through, try to crash it and observe the logs. You can find out that our application logs unsuccessful authentication attempts exposing user and password values:

```
p.goatdroid.fourgoats    Timeline        Timeline: Activity_idle id: android.os.BinderProxy@3f6aa340 t
                                         3999302
p.goatdroid.fourgoats    Failed login    Login with user11 pass1234 failed
p.goatdroid.fourgoats    RenderScript    0x4807b250 Launching thread(s), CPUs 2
```

To check if the application has debuggable flag on you can use Android Device Monitor. Debuggable applications will be displayed in the left pane of the Monitor's screen. The other way to find it out is to decompile the application and check the AndroidManifest for android:debuggable="true" flag (decompilation has been discussed in Lesson 9):

```
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:debuggable="true" android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:label="@string/app_name"
android:name="org.owasp.goatdroid.fourgoats.activities.Main">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
```

Exploiting vulnerabilities

The debuggable flag vulnerability can be exploited using the adb shell. Start the shell and run id command to check current user ID.

```
user@goatdroid:~$ adb shell
shell@GT-I9100:/ $ id
uid=2000(shell) gid=2000(shell)
groups=1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),
3002(net_bt),3003(inet),3006(net_bw_stats) context=u:r:shell:s0
```

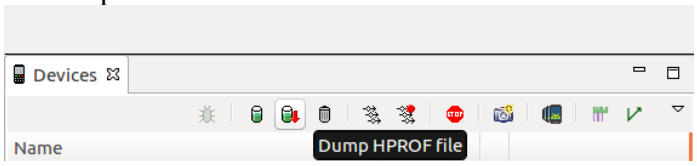
Id command returned uid of shell user – the standard user used by adb shell. Now run the run-as command and check user ID:

```
shell@GT-I9100:/ $ run-as org.owasp.goatdroid.fourgoats
shell@GT-I9100:/data/data/org.owasp.goatdroid.fourgoats $ id
uid=10090(u0_a90) gid=10090(u0_a90)
groups=1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),
3002(net_bt),3003(inet),3006(net_bw_stats) context=u:r:untrusted_app:s0
```

The command changed the user ID to u0_a90 – ID of the application's user. This allows to access database files that are not normally accessible, because of the file permissions (discussed in Lesson 2):

```
shell@GT-I9100:/data/data/org.owasp.goatdroid.fourgoats $ cd databases
shell@GT-I9100:/data/data/org.owasp.goatdroid.fourgoats/databases $ ls -la
-rw-rw---- u0_a90   u0_a90      16384 2017-01-12 19:02 userinfo.db
-rw------- u0_a90   u0_a90       8720 2017-01-12 19:02 userinfo.db-journal
shell@GT-I9100:/data/data/org.owasp.goatdroid.fourgoats/databases $ sqlite3 userinfo.db
SQLite version 3.8.6.1 2015-05-21 17:24:32
Enter ".help" for usage hints.
sqlite> .tables
android_metadata  info
sqlite>
```

Debuggable flag also allow to make the dump of heap of process memory. To do it select the application package in the left pane of Android Device Monitor and select dump hprof icon:



The hprof file will be generated shortly and the application will ask for the path to store it. The hprof file generated by Android Device Monitor is not compatible with Java Virtual Machine hprof files and cannot be used directly in memory analysis software, which accepts these files. However there is a hprof_conv tool available in Android SDK which allows to convert these files. This converted file can be opened in Eclipse Memory Analyzer Tool which is mainly used for tracking memory leaks, but can also be used for finding sensitive data in the heap dump.

But heap dump is just a binary file and can be viewed in another way. If you are looking for certain password to check its existence in the file, you can just search for it. But because the heap dump contains null bytes between every string character, you either need to use Find and Replace tool to remove all null bytes or prepare a particular search string. On Windows Notepad++ handles well such big binary files (the heap dump from Fourgoats is ~20MB), on Linux simple command line tools can be utilised. We are looking for the "pass1234" password string in the heap dump:

```
user@goatdroid:~$ cat dump.hprof | grep -Pa 'p\x00a\x00s\x00s\x001\x002\x003\x004'
#���#��0#���#��@#���##���#pass1234"#��#��_�!#���#��#��x"#��@      oЧ�# ���#’ ��# ��
#���#’oĶxoĶ####
access$900!#]@oб(##��E#]`##]`#attach!#]@oб(##C;U#T`!
```

It was found in the heap dump.

Fixing vulnerabilities

According to best practices[2] a release application should not log any data. One of the popular ways to achieve this is to make an own class inheriting from or wrapping android.util.Log, so before calling the logging method the class checks the variable to make sure if logging is enabled. Another one that is less elegant is to put every logging method call into a conditional statement:

```
if (DEBUG)
       Log.i(TAG, "message");
```

But both of these solutions till leave the String constants in the release code. These string can provide additional information helping to reverse engineer the application.
The best way to resolve this issue and completely strip the release binary from log statements is to use ProGuard (discussed in lesson 9.1). Enable the ProGuard optimization in build.gradle file and use the proper rules to strip release binary from these statements:

```
-assumenosideeffects class android.util.Log {
    public static int d(...);
    public static int v(...);
    public static int i(...);
    public static int w(...);
    public static int e(...);
}
```

Debuggable tag is disabled by default in production builds, but it could be explicitly enabled. In older versions of Android Studio it was made by putting android:debuggable=”true” in a manifest. Delete it, if exists and check build.gradle file, which is recommended for this setting now. Remove the debuggable flag from release build properties:

```
buildTypes {
       release {
       minifyEnabled false
       proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
       debuggable true
       }
}
```

If debuggable flag has been set properly the risks regarding heap dumps are really low, but in some high security applications additional measures can be taken to minimize the risks of exposure of sensitive data like passwords and private cryptographic keys. Using String objects to store them creates the problem of these data being stored in the heap until the garbage collector destroys it, even if it is no longer needed to keep them. That is why such data should be stored using primitive data types in Java, so they can be cleared immediately without waiting for the garbage collector. Sensitive strings should be store using char[] array and immediately cleared by assigning different value, when they are no longer needed.

2 https://developer.android.com/studio/publish/preparing.html