Cybersecurity Blog

Everything about threat intelligence, blue team, red team, pentesting, security audit, security review, testing and assessment.

Friday, November 18, 2016

Android Anti Java Hooking - Adding Layer to your SSL pinning and Root detection

In this article I am going to highlight the importance of why we must implement anti java hooking technique in our application. What are their advantages and disadvantages.

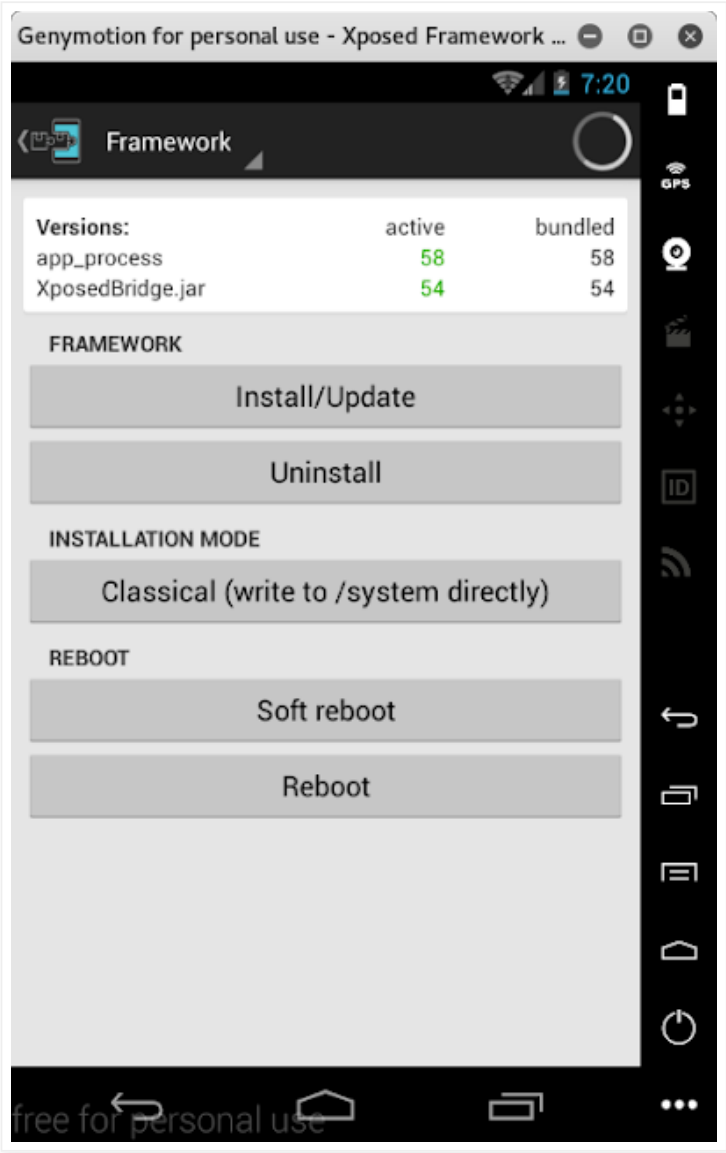
What is android hooking?

Hooking is a process of injecting malicious payload into existing running process. To illustrate that, assume we have root detection feature in our application. Using rootclock 3rd party application if root detection mechanism can be bypassed. Now mostly all these application which bypasses root detection, ssl pinning etc.. they hook into running application process. So how to be safe against these application? Here comes the android anti java hooking technique.

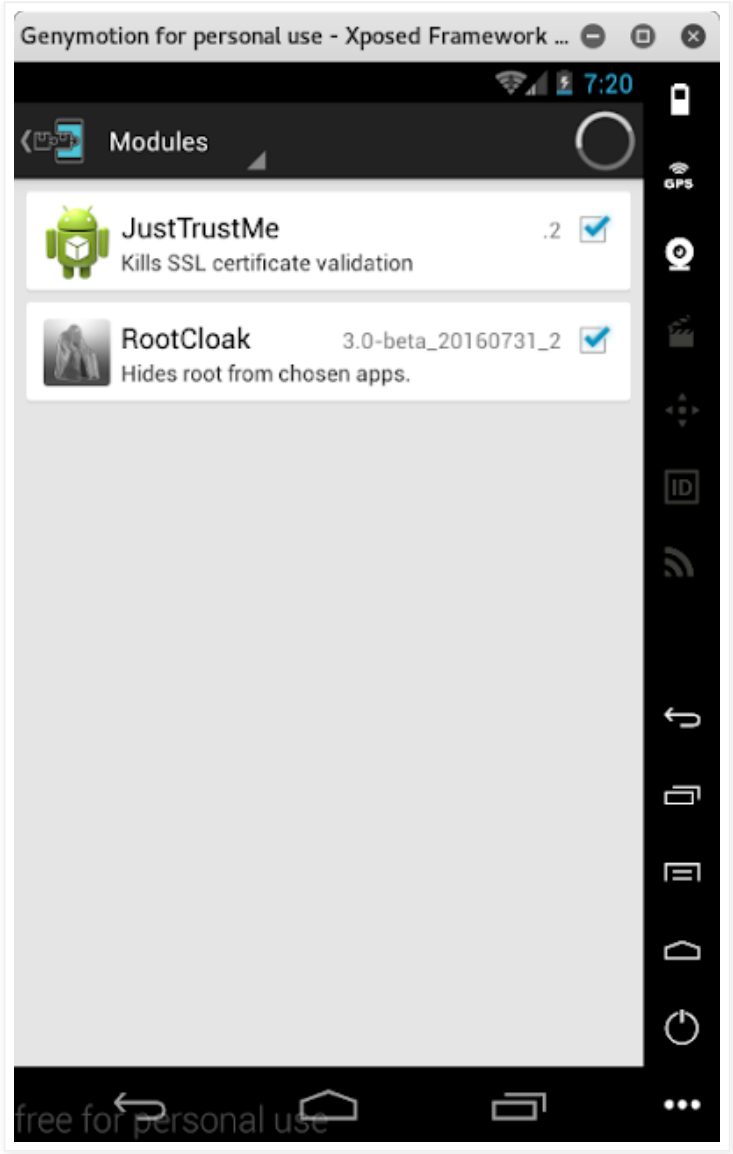
How hooking works?

Lets say DIVA application implements ssl pinning. So there is another application dubbed as "JustTrustme" which will enable attacker to perform an automated hooking into running process of DIVA application. Below screenshot is the evidence of the same.

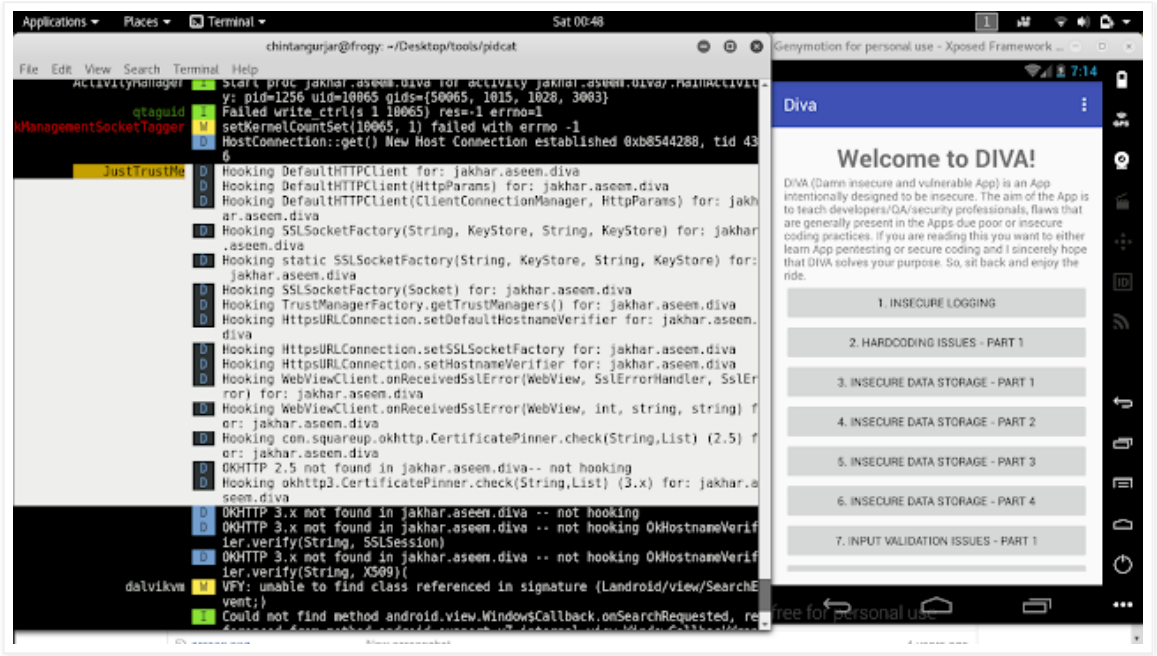
Screenshot 1: Below screenshots shows that Xposed Framework is installed and JustTrustMe module of it is active. This is basically used to bypass SSL pinning and to intercept HTTPS requests in BurpSuite interception.



- May 1 (1)
- April 10 (1)
- April 3 (2)
- 2015 (4)
- 2014 (22)
- 2013 (58)



Screenshot 2: Below screenshot shows that DIVA application has been started and system logs are stating that main activity has been launched. As soon as DIVA has started JustTrustMe application which is an Xposed Framework has started hooking into running process of this DIVA application. As a result of which, DIVA application can be bypassed against SSL pinning feature implementation.



Due to this hooking, it does the magic and disables SSL pinning for our application and post that an attacker will able to perform any changes in request and response using burpsuite.

Below I am going to highlight few anti hooking techniques. Basically our flaw will be as followed.

Disclaimer: All of these anti-hooking techniques are easy to bypass by any experienced reverse engineer. I'm just exploring how one might go about detecting that their Java application has been hooked using Substrate or the Xposed framework because at some point we will need to be able to bypass these techniques to do our jobs just like how we bypass root detection on a daily basis. The last time I looked at DexGuard and Arxan's Java protection product (GuardIT) they did not support detection of either hooking framework. I would expect similar anti-hooking techniques will be added to these Java obfuscation/protection products in the future.

Challenge: Being an application we should not ask for a system to show logs. Somehow we can achieve that however it can raise privacy concerns. For the demo purpose I have shown you hooking logs which are there on system logs. But our application is also capable of generating **STACK TRACE** error logs while 3rd party application is trying to hook. Below is the sample of stack trace error log taken from other blog.

Assuming that our application is not being hooked. So stack trace would be identical to as follows:

```
com.example.hookdetection.DoStuff->getSecret
com.example.hookdetection.MainActivity->onCreate
android.app.Activity->performCreate
android.app.Instrumentation->callActivityOnCreate
android.app.ActivityThread->performLaunchActivity
```

```
android.app.ActivityThread->handleLaunchActivity
android.app.ActivityThread->access$800
android.app.ActivityThread$H->handleMessage
android.os.Handler->dispatchMessage
android.os.Looper->loop
android.app.ActivityThread->main
java.lang.reflect.Method->invokeNative
java.lang.reflect.Method->invoke
com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run
com.android.internal.os.ZygoteInit->main
dalvik.system.NativeStart->main
```

Assuming that our application is being hooked. So stack trace would be identical to as follows:

```
com.example.hookdetection.DoStuff->getSecret
de.robv.android.xposed.XposedBridge->invokeOriginalMethodNative
de.robv.android.xposed.XposedBridge->handleHookedMethod
com.example.hookdetection.DoStuff->getSecret
com.example.hookdetection.MainActivity->onCreate
android.app.Activity->performCreate
android.app.Instrumentation->callActivityOnCreate
android.app.ActivityThread->performLaunchActivity
android.app.ActivityThread->handleLaunchActivity
android.app.ActivityThread->access$800
android.app.ActivityThread$H->handleMessage
android.os.Handler->dispatchMessage
android.os.Looper->loop
android.app.ActivityThread->main
java.lang.reflect.Method->invokeNative
java.lang.reflect.Method->invoke
com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run
com.android.internal.os.ZygoteInit->main
de.robv.android.xposed.XposedBridge->main
dalvik.system.NativeStart->main
```

Now using these our own application's stacktrace (which are not being fallen on system logs), we can decide a flow of our application.

How it works?
Step 1: Your application starts normally.
Step 2: 3rd Party application will automatically start hooking into your application.
Step 3: Your application generates stack trace error messages inside it and decides if Xposed is found on the system or not.
Step 4: If found your application exists else runs smoothly.

How to check for specific package is installed or not?

The first thought that comes to mind is simply detecting whether or not Substrate or the Xposed framework is installed on the device. We can ask the PackageManager for a list of installed packages and flag any suspicious packages, which is a common technique used in root detection.

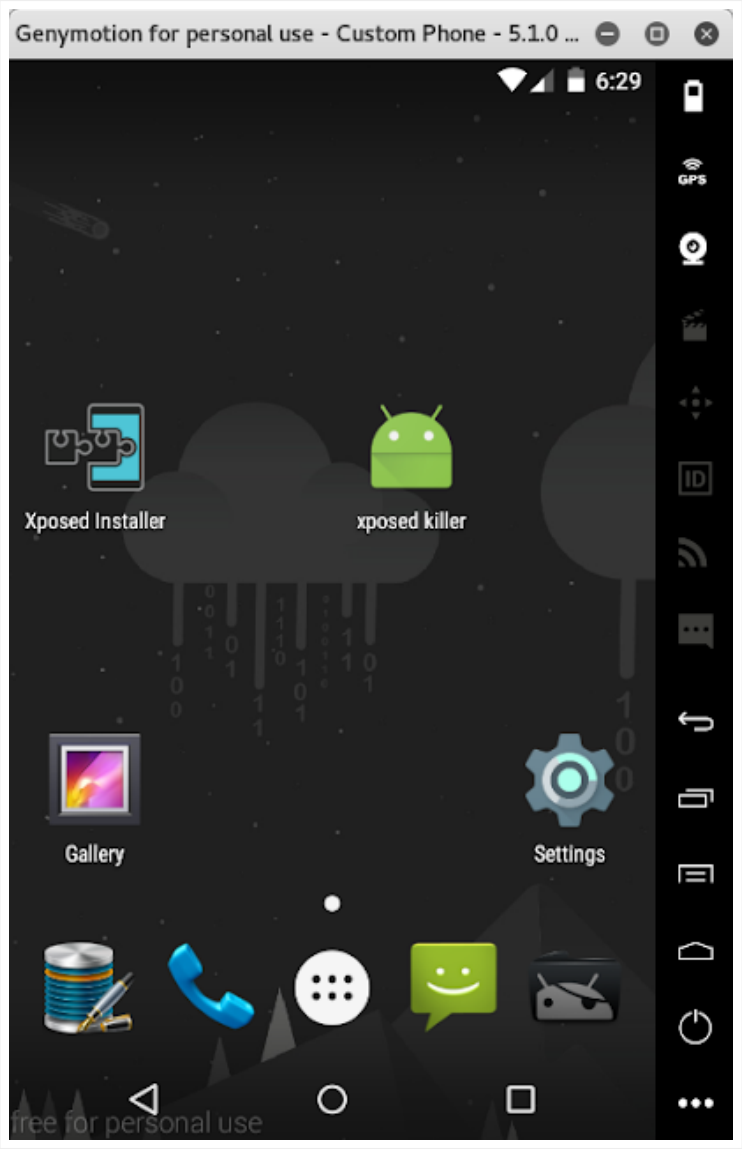
Code snippet

```
PackageManager packageManager = context.getPackageManager();
List applicationInfoList  = packageManager.getInstalledApplications(PackageManager.GET_META_DATA);

for(ApplicationInfo applicationInfo : applicationInfoList) {
 if(applicationInfo.packageName.equals("de.robv.android.xposed.installer")) {
  Log.wtf("HookDetection", "Xposed found on the system.");
 }
 if(applicationInfo.packageName.equals("com.saurik.substrate")) {
  Log.wtf("HookDetection", "Substrate found on the system.");
 }
}
```

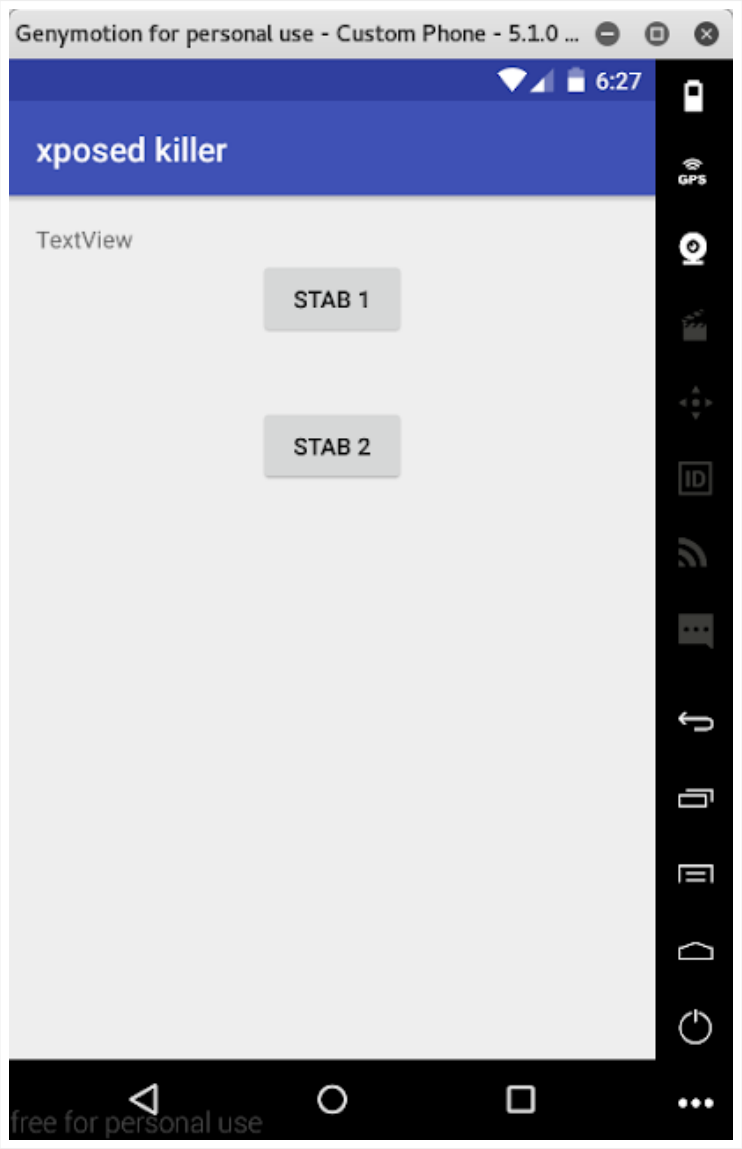
Demo: Introducing Sample Application: Xposed Killer

Here is the sample application which detects if Xposed and Cydia Substrate is installed on the rooted device or not.



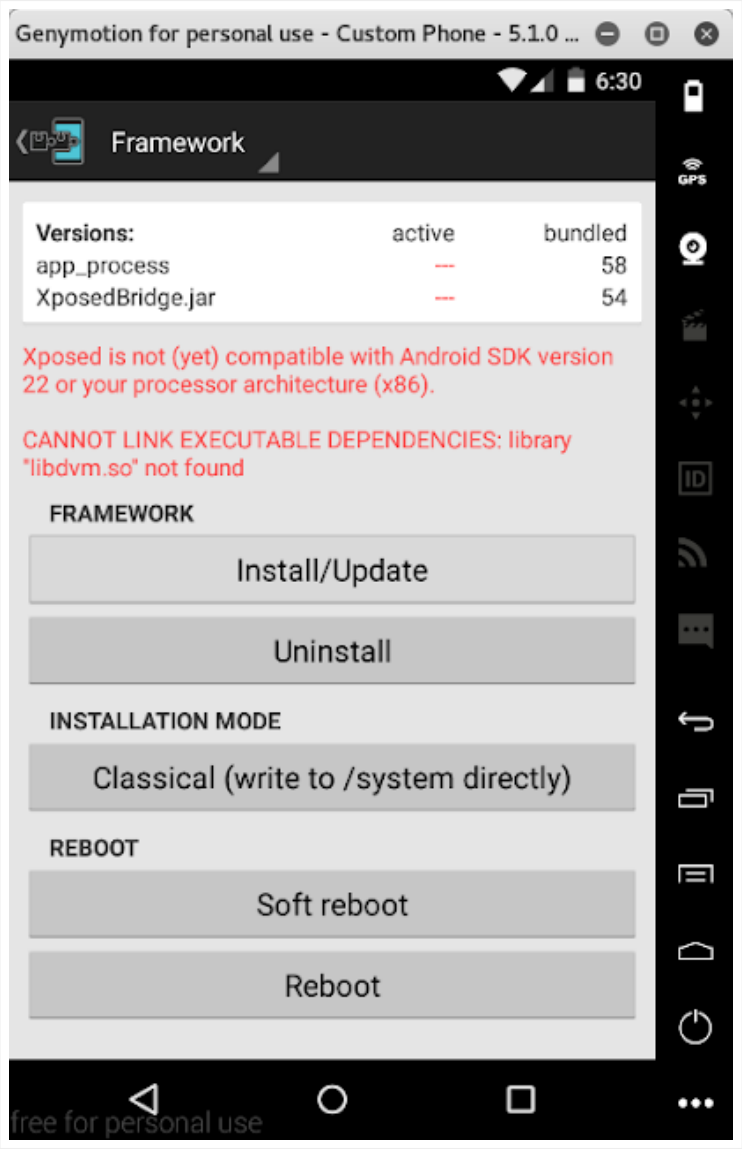
As mentioned below there are two stages below.
Stab 1: Stab 1 is used in case our application finds that Xposed Framework and Cydia Substrate is installed on the device.

Stab 2: Stab 2 is used in case our application finds that Xposed Framework is active on the device.

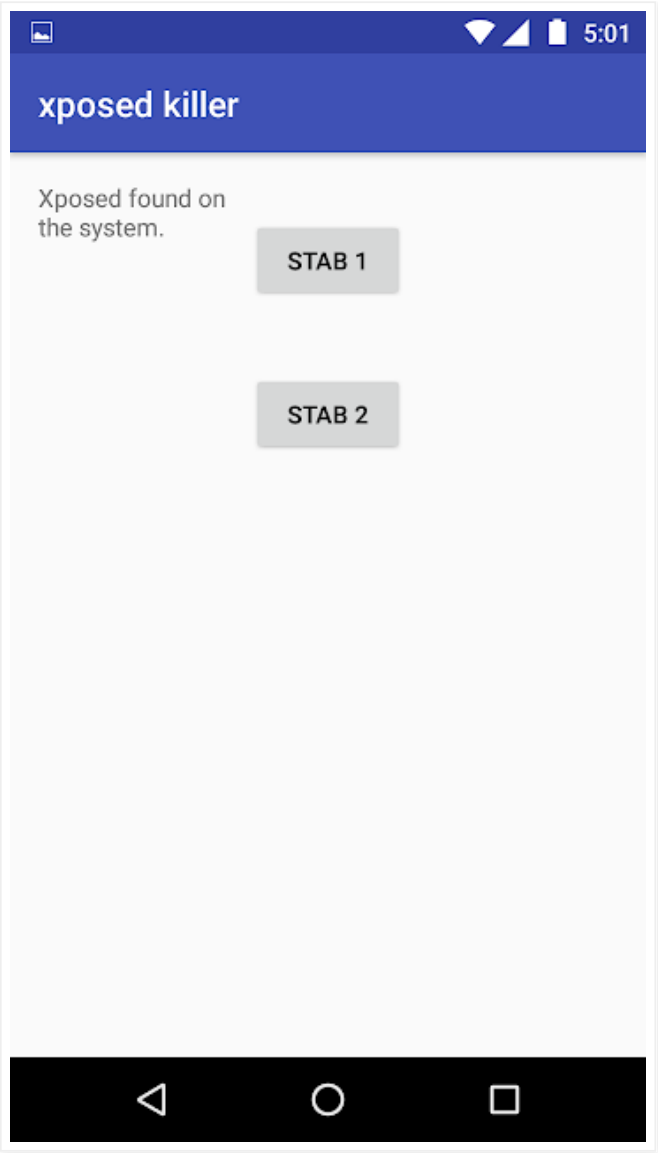


Implementation Scenario:

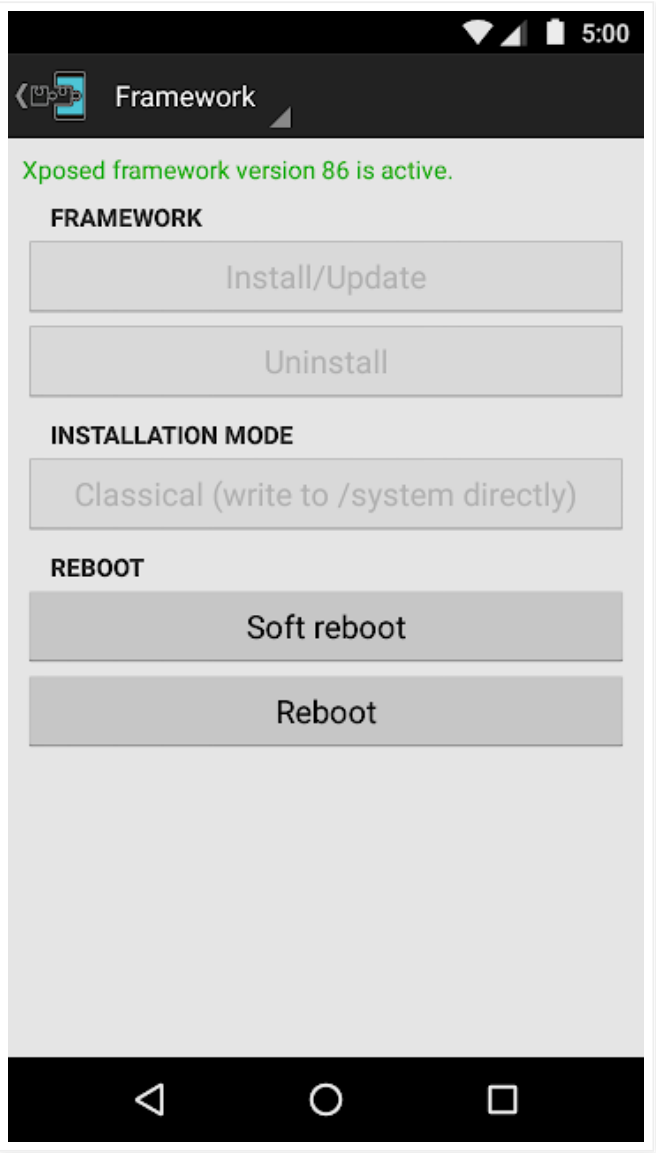
Case 1: Xposed installer(apk) is installed on the device but not active. In this case observe how xposed killer will react.

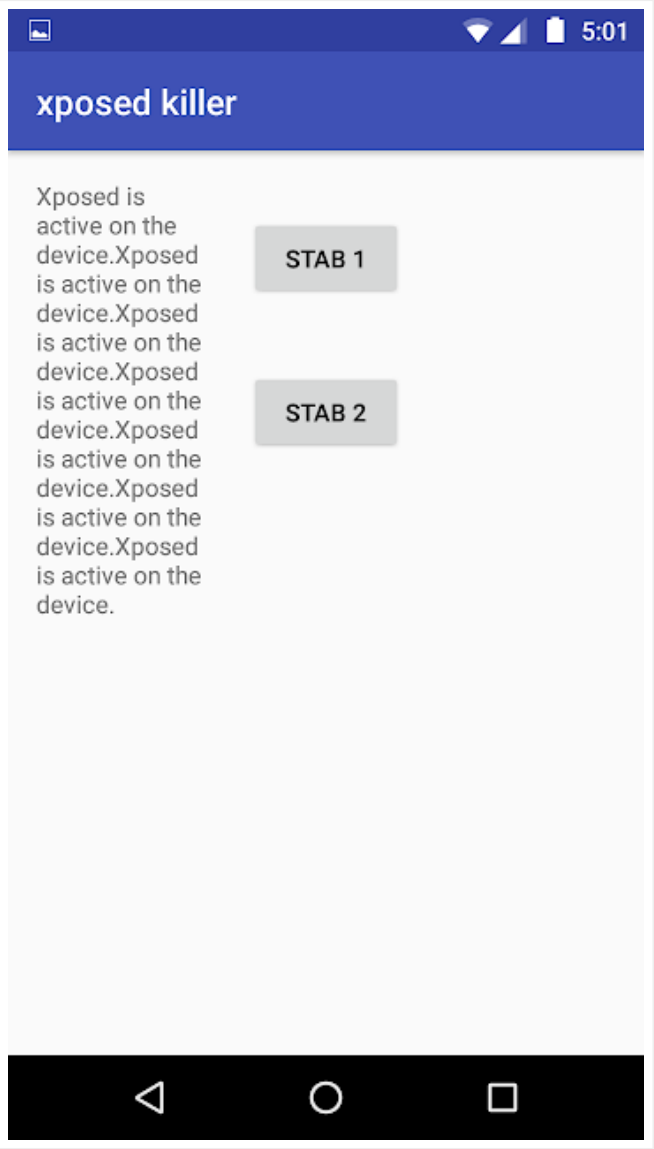


Now let us run our xposed killer app. By clicking on stab 1 it shows that Xposed is found on the device by clicking on stab 2. It will show xposed is not active hence it will not give any result.



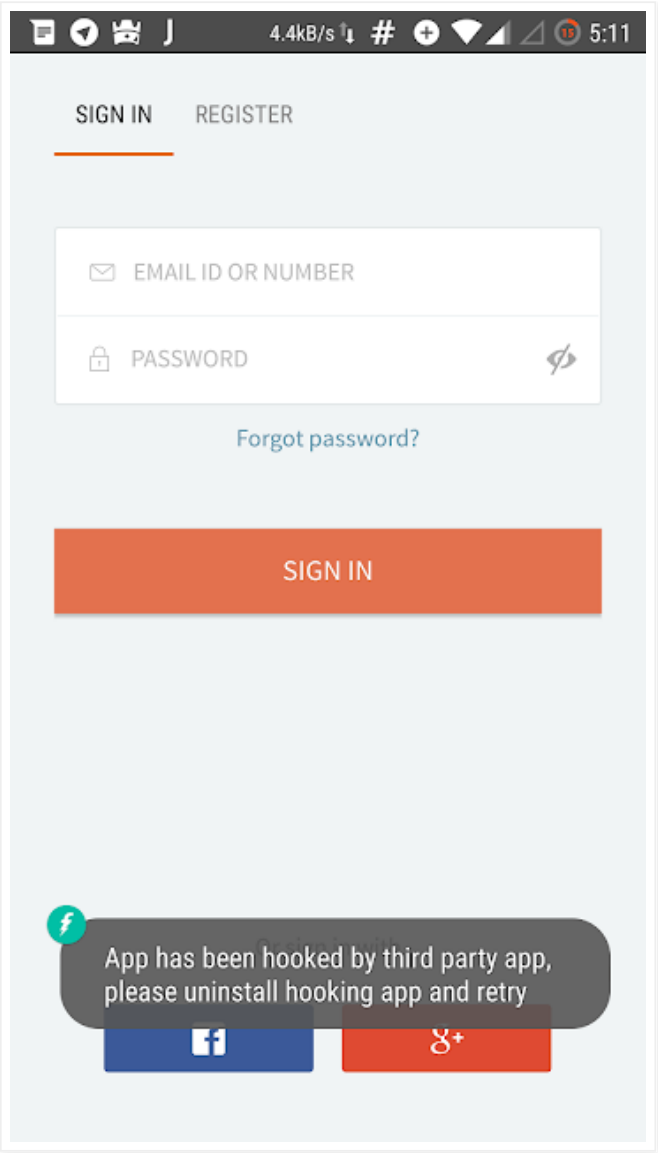
Case 2: Xposed installer(apk) is installed on the device and it is active.


5:00
Framework
Xposed framework version 86 is active.
FRAMEWORK
Install/Update
Uninstall
INSTALLATION MODE
Classical (write to /system directly)
REBOOT
Soft reboot
Reboot



Now what?
Now what? you have the statistics. If you find that Xposed is active and installed then kill your application by showing custom error message.

Any real life example?
Yes. Freecharge.



Reference:

1. http://d3adend.org/blog/?p=589

Posted by Frogy at 11/18/2016

Labels: android application, android security, anti java hooking, how to defeat xposed framework, root detection, ssl pinning

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.