Native Protection & Mono

Note: this tutorial covers a basic understanding of the Mono/Xamarin for Android, specifically how C#/.NET is used lately to protect games and, who knows, even malwares.

Version: 0.1a

Author: Nihilus

Disclaimer

This document is not a "How to crack applications protected with native code using Mono/Xamarin", it's just a general understanding of how to detect and analyze them. The application I choose is free on the Google Play Store and the protection engine will not be cracked. Thanks to Juicy Beast Studio for the funny Knightmare Tower. I Googled a bit and found that the app is being cracked before; I want to invite everyone to buy apps or games you love because programmers need support.

Mono/Xamarin

Mono is an open source project led by Xamarin, aid to create cross-platform apps using C#/.NET. Mono can be run on many OS, included Android. Here is the official site: http://xamarin.com/android. Basically the Mono runtime contains a code execution engine that traslates ECMA C# byte codes into native code; it supports a number of processors like: ARM, MIPS, x86, x86-64 and others. The Android implementation is based on a native library called "Mono.so" that executes the DLLs coded in C#/.NET.

What is needed

- Smali/baksmali.
- A text editor such as SublimeText or Notepad++.
- Android SDK with working debugger.
- A rooted Android device.
- 7-zip or any other program that can extract *.apk files.
- A program that uses the Mono library and implements a Native Protection (I suggest Knightmare Tower).
- IDA or another *.so/*.dll analyzer (https://stackoverflow.com/questions/1563649/net-reflector-for-mono).

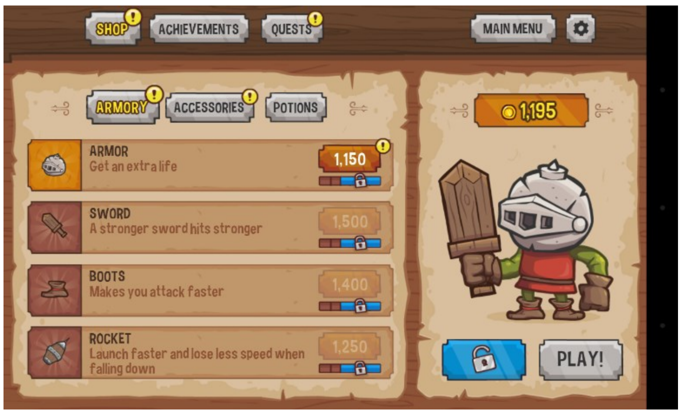
Setup

- Install an app that uses Mono (I chose Knightmare Tower) on your device.
- Go to /data/app/ on your device with a file manager or using ADB shell and copy to your computer the file com.juicybeast.knightmaretower-*.apk (You could use ADB: adb pull /data/app/ com.juicybeast.knightmaretower-*.apk, note that * stands for a number, usually "1")

- Extract classes.dex from the apk file and disassemble it with baksmali.jar (java –Xmx512M –jar baksmali.jar –o classes/)
- Now you can open all the *.smali files inside the classes folder in your favourite text editor.
- Now let's get a look to the app!

Let's start our Tower escalation

First run the game on your device and play a bit, when you die you can notice that the game has a marketplace where you can buy power-ups, accessories and potions. All these things are locked at a certain level and you have to buy the game to increase the power-ups. At this point a beast enter the Tower: the protection implemented by the programmers.

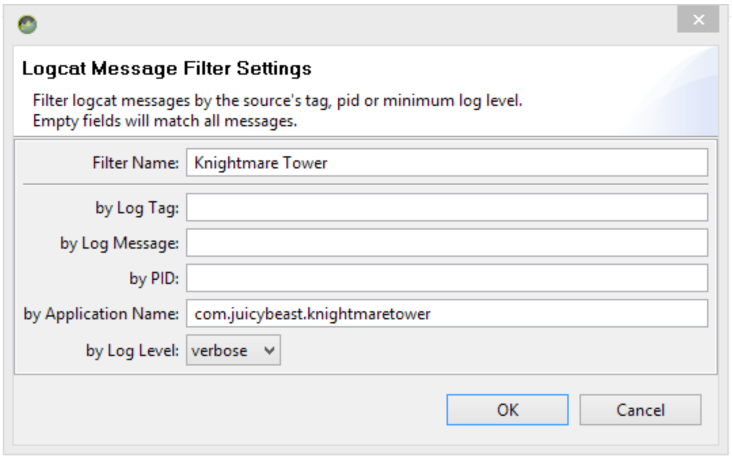


Here is the market with the locked content, and the blue Unlock/Buy button

Now we can extract the apk content on a folder and analyze it. Inside the apk we can notice a folder "lib" dedicated to Native libraries. Inside the lib folder we can find another one "armeabi-v7a" and inside it 3 "*.so" libraries. We are going to look for interesting debugging information.

The debugger treasure chest

I call it "treasure chest" because of the valuable information it gives us. Now, fire up the debugger, activate the "USB Debug" under the "Programmer settings" on your phone, connect the device to the PC and open the game. (You can find the Android Debugger inside "SDK-Folder/tools/monitor.bat"). Add a new filter to the debugger for your app; in my case here it is. I selected "verbose" log level because we don't want to miss interesting information.



Logcat Message Filter Settings

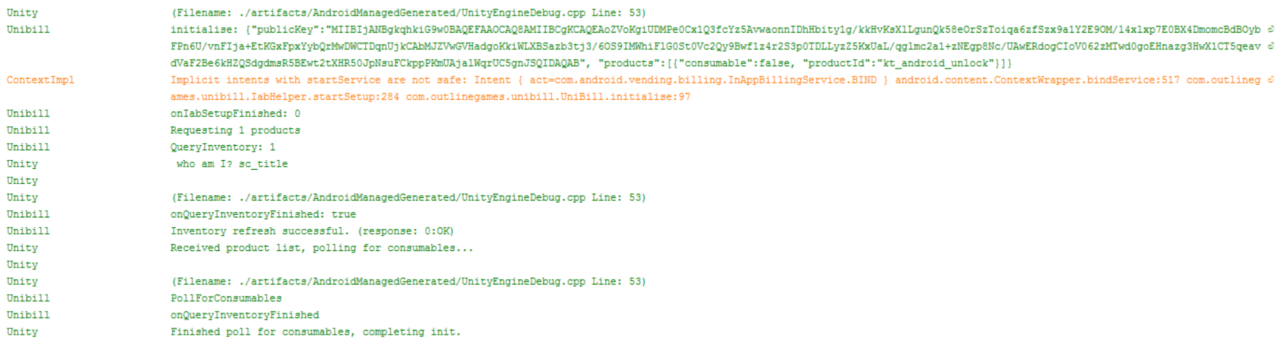
Filter logcat messages by the source's tag, pid or minimum log level.
Empty fields will match all messages.

Filter Name:	Knightmare Tower
by Log Tag:	
by Log Message:	
by PID:	
by Application Name:	com.juicybeast.knightmaretower
by Log Level:	verbose

Start the game and take a look at the bootstrap process of the app in the debugger. We can notice some interesting information regarding: refreshes of the game inventory, initialization of an in-app billing service called Unibill (here for more information: http://outlinegames.com/unibill-documentation/), initialization of the game components.

A quick look inside Unibill

Unibill is a plugin for Unity3D that works accross Apple App store, Google Play, Amazon App store, Windows Phone 8 and Windows 8.1. It wraps the raw interfaces of the mentioned stores, so Unibill is the unique interface responsible for the communication between the app and the billing system. Tower Defense uses it for the in-app purchase part (it is triggered when we push the "Unlock/Buy button").



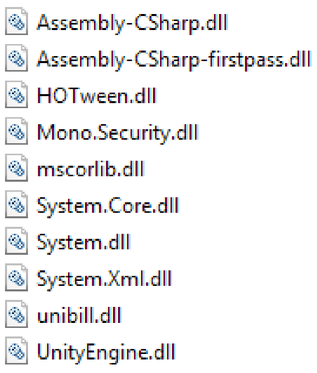
```
Unity          (Filename: ./artifacts/AndroidManagedGenerated/UnityEngineDebug.cpp Line: 53)
Unibill        initialise: {"publicKey":"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoZVoKgiUDMPe0Cx1Q3fcYz5AvwaonnIDhHbity1g/kkHvKsXlLgunQk58eOrSzToiqa6zfSzx9a1Y2E9OM/l4xlxp7E0BX4DmomcBdBOyb
               FPn6U/vnFIja+EtKGxFpxYybQrMwDWCTDqnUjkCAbMJZVwGVHadgoKkiWLXBSazb3tj3/6OS9IMWhiFlG0St0Vc2Qy9Bwf1z4r2S3p0TDLLyzZ5KxUaL/qglmc2a1+zNEgp8Nc/UAwERdogCIoV062zMTwd0goEHnazg3HwX1CT5qeav
               dVaF2Be6kHZQSdgdmsR5BEwt2tXHR50JpNsuFCkppPKmUAjalWqrUC5gnJSQIDAQAB", "products":[{"consumable":false, "productId":"kt_android_unlock"}]}
ContextImpl    Implicit intents with startService are not safe: Intent { act=com.android.vending.billing.InAppBillingService.BIND } android.content.ContextWrapper.bindService:517 com.outlineg
               ames.unibill.IabHelper.startSetup:284 com.outlinegames.unibill.UniBill.initialise:97
Unibill        onIabSetupFinished: 0
Unibill        Requesting 1 products
Unibill        QueryInventory: 1
Unity           who am I? sc_title
Unity
Unity          (Filename: ./artifacts/AndroidManagedGenerated/UnityEngineDebug.cpp Line: 53)
Unibill        onQueryInventoryFinished: true
Unibill        Inventory refresh successful. (response: 0:OK)
Unity          Received product list, polling for consumables...
Unity
Unity          (Filename: ./artifacts/AndroidManagedGenerated/UnityEngineDebug.cpp Line: 53)
Unibill        PollForConsumables
Unibill        onQueryInventoryFinished
Unity          Finished poll for consumables, completing init.
```

The initialization of the game inventory and Unibill interface

We also notice the debug strings pop out from different "processes" like "Unity" or "Unibill". Unity messages are generated by the libunity.so file, and Unibill messages are generated by the Java implementation of the app. If try to search for the debug strings in the "classes/" folder we will find only the ones generated by the Java class "UniBill.java". I gave a quick look but the game doesn't seems to use string encryption. So, where we can find the missing debug strings generated probably by the Unibill implementation?

A deeper look inside the apk resources

Now comes the strange part. Let's take a look inside the "assets" folder of our apk! Inside it we can find the videos and a folder called "bin", it stands for "binary". Go deeper and find the "Managed" folder (assets/bin/Data/Managed). The first time (I didn't know about the existence of Mono for Android) I didn't believe my eyes. Let's take a look.



Assembly-CSharp.dll
Assembly-CSharp-firstpass.dll
HOTween.dll
Mono.Security.dll
mscorlib.dll
System.Core.dll
System.dll
System.Xml.dll
unibill.dll
UnityEngine.dll

DLLs

Some DLLs (Dynamic-Link Library) in an apk package. Written in C#/.NET obviously. Inside this DLLs there is the implementation of the protection used by the game (the initial check that controls if the game is unlocked or not, and the implementation of Unibill) and here we can also find the missing debug messages. These libs are interpreted and executed by the Mono engine (libmono.so). This solution seems to guarantee higher performance than Java code only.

Conclusion

A protection coded in Java only is really simple to crack, so programmers started to distribute the protection mechanism into Native Code or using frameworks like Unibill. A casual cracker probably find more difficult to crack a protection distributed into *.so and *.DLLs.

I find that the game doesn't have a built-in integrity check to control if a DLL has been modified; the implementation of a check is really simple, so I invite all the programmers that want to use Native Protection or engines like Mono to do it.

Like I said, I'm not going to tell how to crack the app, the tutorial has only the purpose of explaining how Mono works from a really restricted point of view. So, if malware writers are going to use Mono or Native libs to accomplish evil tasks, we have to take an eye open for strange libs or assets.

Nihilus