nccgroup

DECONSTRUCTING KONY ANDROID APPLICATIONS

KONY PAST, PRESENT, AND FUTURE

Chris Weedon, June 1, 2015

Abstract

Kony Inc. implements a "write once, deploy many" IDE to simplify mobile application development and expand developer platform reach. While making development tasks easier, this makes security engineers' analysis much more difficult. Earlier revisions of the framework operated by embedding a Lua Bytecode VM within the application, while later revisions took a different approach. Regardless, the Kony Studio IDE does an effective job of obfuscating application analysis. This paper explains the different implementations used by the IDE and focuses on ways to overcome the obstacles presented.

Kony Inc., established in 2007, is one of several multi-platform mobile application development platforms that have been formed as mobile applications have become more in demand by the commercial consumer base. It is fair to say that Kony has successfully established itself in the mobile application development market and secured its place in the current multi-platform development field. In May of 2014, Kony announced it had received $50 million dollars in funding, as well as being recognized as a leader in Gartner's Magic Quadrant for Mobile Application Development Platforms[1] in 2014 and the previous year, 2013.

Multi-platform development platforms allow development teams to streamline operations and management oversight by allowing the development team to write in one programming language. However, this ability adds inherent risks. Specifically, nuances in end-user runtime environments could cause application code to be misinterpreted or executed in an insecure manner. Multi-platform development environments must tackle these issues, while still allowing the developers and end-users a seamless experience.

Digression

Kony's solution to this problem was to use the platforms' native application and execution environment as a wrapper, embed the actual application code and its runtime into this wrapper, and hook the embedded application's functions with corresponding native code translations. This type of approach is quite simple in theory as operations can be broken down into basic primitives. These primitives can be derived from basic programmatic function types. A useful mobile application function will take in one or more inputs, perform one or more operations based on these inputs, and then return a value or multiple values back to the invoking entity.

One could create a programmatic template for every unique set of inputs and corresponding outputs and pair these templates with their corresponding wrapper prototypes. Once all corresponding templates have been matched with a wrapper prototype, a map of values could be created that would generate a corresponding native mobile application wrapper hook for each embedded application function. Such exercises are out of the scope of this paper, however, a high level explanation of how multi-platform application development environments operate will be helpful in understanding the analysis process of applications developed by such an IDE.

[1] http://www.cmswire.com/cms/customer-experience/ibm-sap-adobe-lead-in-gartner-mq-for-mobile-apps-development-022081.php

Main

The Kony IDE can be separated into 3 main categories for analysis methods, at least as the author sees it. These categories have been named, rather unscientifically, Kony Past, Kony Present, and Kony Future. The content in this section will highlight the major differences between the 3 categories, outline the analysis techniques needed to properly audit applications of each category, as well as raise concerns and discuss security implications that may exist.

The first, and oldest, category to address is Kony Past. This category consists of Android applications built using versions of the Kony Studio IDE less than version 6.0 (released December 14, 2014) but greater than version 5.0. As the author could not located any Android applications that were marked with a version prior to version 5.0.

Applications in this category rely on a compiled Lua Bytecode Virtual Machine embedded into the Android APK file. This Bytecode VM, in its entirety, is the application source code. Additionally, data such as application tracking metrics, licenses, and configuration files relevant to the specific application are stored in serialized data files denoted with the ".kds" extension. These files are relatively easy to deserialize since their format is standard Java serialization version 5.0 and as such they will be discussed no further.

The Bytecode VM introduces a unique problem to the application auditor. Examining the Android application Java class files in decompiled form revealed that the Android application itself was a wrapper, and through various levels of call abstraction, all calls pointed to "main" in some way, shape, or form. The "main" class however proved to be empty. Such analysis steps were taken and are well documented in a 2012 blog post from Jason Ross of Intrepidus Group[2].

However, this blog post fell short of thoroughly analyzing the application, as it did not address the Bytecode VM. Upon investigation, failure to address the Bytecode VM was likely due to the fact that the stable version of the tool "unluac.jar" would not properly decompile the compiled Lua Bytecode, and the tool's author had not made regular updates to the project until 2014. Additionally, it is likely that the blog post author had neither the time nor resources to develop their own tool, or investigate why "unluac.jar" failed to properly decompile the Bytecode. Regardless, while novel techniques, such as memory dumping and memory image forensics to reassemble the code in memory were developed, those methods are inefficient and laborious. The current version of "unluac.jar" from the stable branch on sourceforge[3] works and can successfully decompile the compiled Bytecode VM without issue.

The steps for properly analyzing this category of Kony applications are as follows:

Extract the target Android APK from the environment and/or device.
Use apktool to decompress the Android application. Alternatively, just unzip the APK file.
Locate the file "konyappluabytecode.o.mp3" or other Lua Bytecode VM files with an alternative naming convention.
Run "java –jar unluac.jar konyappluabytecode.o.mp3 > outputfile.lua"
Proceed with static source code analysis.
Dynamic analysis can proceed as usual using ADB, supplemented with a suitable Lua debugger.

[2] https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2013/june/kony-2013-a-different-kind-of-android-reversing/
[3] http://sourceforge.net/projects/unluac/files/?source=navbar

The next category of Kony application is labeled Kony Present. This is in part due to the fact that these are the current variant of Kony applications you are likely to find in commonly used application stores, such as the Google Play Store or the Yandex Marketplace. This category consists of applications compiled by Kony Studio IDE equal to or greater than version 6.0, but less than version 6.0.3. These applications are the debut of the new application framework and the deprecation of the Lua Bytecode VM.

The new application framework consists of relying on a native shared object library file, “libkonyjsvm.so,” located in the “lib/armaebi/” directory of the application embedded within the application APK file. Using the new framework, developers can write their code in JavaScript and/or HTML5, and have it translated and executed by the shared object library, which at its core is really a V8 engine with JNI hooks for the applicable wrapper functions.

While this deprecates our previous analysis techniques, it simplifies the tools required and overall lowers the bar of entry to analyze Kony Android applications. Investigations into the application itself reveal that the new framework simply packages all the JavaScript source code into a zip file, named “startup.js” in the “assets/js/” directory within the application, which is then unpacked in memory at application runtime by the shared object library. Once the library file is loaded by the Android wrapper, it will search for the file in the application’s directory and unzip the source code into memory. Unfortunately, the source contained in the “startup.js” file is never unpacked to disk. However, it is trivial to unzip the file yourself.

The steps for properly analyzing this category of Kony applications are as follows:

Extract the target Android APK from the environment and/or device.
Use apktool to decompress the Android application. Alternatively, just unzip the APK file.
Locate the file “startup.js” and/or other files with a file header magic byte sequence of zip archive.
Extract the file archive to disk.
Proceed with static source code analysis.
Dynamic analysis can proceed as usual using ADB, supplemented with GDB for analysis of native code execution if desired.

The third and final category of Kony application is labeled Kony Future. These applications are generated by the current revision of the Kony Studio IDE (version 6.0.3) and higher. Although this version of the Kony Studio IDE was released in March 2015, a sampling of applications in the Google Play Store and Yandex Market that are known to be developed by Kony Studio IDE revealed that versions of applications built using this framework version have not yet been pushed to consumers.

While the framework maintains the same overall method of using the shared object library file to load the application code, an additional counter-measure was added. The application’s “startup.js” file is now accompanied by an additional file, “common-jslibs.kfm”, which is also in the “assets/js/” directory within the application. Additionally, both of these files have been heavily obfuscated. Analysis of these files strongly indicated that the files had been encrypted. This suspicion was later confirmed after analyzing the Kony Studio IDE application build process, and by a binary analysis of the “libkonyjsvm.so” file.

Analysis of the build process, binary analysis of relevant library files, and binary analysis of executable Kony Studio IDE code revealed that the application source code archive, “startup.js,” was being encrypted using the EVP_AES_256_CBC cipher. However, the encryption key and initialization vector (IV) are hardcoded into the Kony Studio IDE executables, as well as the “libkonyjsvm.so” shared object library files.

Further analysis of the application encryption routine revealed that although the encryption key and IV are hardcoded into the shared object library files, the key and IV by themselves cannot be used to decrypt any application source code archive. The key and IV are

supplemented by command line parameters passed into the build process by the Kony Studio IDE. These parameters help to introduce additional entropy into the encryption process and to generate an encryption key that is unique on a per-application, per-build revision basis.

However, being that AES is a symmetric encryption algorithm, these parameters need to be present during both encryption and decryption. Thus, one can conclude that the Android APK file contains the relevant parameters necessary for decryption. Close inspection of the build process was performed, and the build process script was modified to echo highly verbose debug output to the build console. This debug output correctly identified the parameters passed in as command line arguments to the encryption routine.

The parameters are the Application ID (APPID), the Package Name (packagename), and the build timestamp (timestamp). The modified debug output can be seen below:

```
C:\Users\Chris\KonySampleApps\temp\HelloWorld\build\luaandroid/dist/HelloWorld/../..
/extres/kony_loadfile.exe
C:\Users\Chris\KonySampleApps\temp\HelloWorld\build\luaandroid\dist\HelloWorld\asset
s\js\startup.js
C:\Users\Chris\KonySampleApps\temp\HelloWorld\build\luaandroid/dist/HelloWorld/asset
s/js/temp.kfm HelloWorld com.kony.HelloWorld 20150530095920
```

Given, the specific values being passed in as arguments, it was relatively easy to derive these values from the Android APK file. The APPID was simply the application name, and the packagename is self-explanatory. Both of these values can be retrieved from the “AndroidManifest.xml” file from their respective labels. The last parameter is the build timestamp, and is located in the “application.properties” file in the “assets/” directory of the APK file. Now that the encryption variables are known, it is possible to see exactly how the variables effected the resulting encryption. While the specifics of the encryption routine will not be discussed in-depth, the following high-level overview should suffice to explain the effect of these variables:

- The Timestamp and the APPID parameter are character XOR’d together.
- The result of the previous operation and the packagename parameter are character XOR’d together.
- The result of the second XOR function is passed into a SHA256 digest function along with the hardcoded encryption key.
- The result of the SHA256 digest function is truncated at 64 bytes and passed into the call to EVP_AES_256_CBC along with the hardcoded IV, the file to encrypt, the output file name, and the encryption mode switch.
- The encrypted file output is written to the output file name, and the output file is the renamed to the original input file name.

Various attempts were made to write a static decryption script, however after numerous failures, it was determined that the swiftest course of action would be to leverage the file encrypter provided by the Kony Studio IDE. The executable responsible for encrypting the source code files, “kony_loadfile.exe”, was modified. A One-Byte binary patch was applied to the call to EVP_AES_256_CBC encryption mode switch parameter. This switch parameter tells the function whether we are “1” encrypting or “0” decrypting. Patching the “0x01” value to “0x00” and then running the executable against the encrypted files successfully decrypted the files in place. The resulting output files were zip archives, as seen previously in the Kony Present category.

The steps for properly analyzing this category of Kony applications are as follows:

Extract the target Android APK from the environment and/or device.
Use apktool to decompress the Android application. Alternatively just unzip the APK file.
Locate and copy the files “startup.js” and “common-jslibs.kfm” from the “assets/js/” directory.
Download a copy of the Kony Studio IDE.
Verify the hardcoded encryption key in the “libkonyjsvm.so” file within the Android APK file and the key in the “kony_loadfile.exe” utility match.
Patch the “kony_loadfile.exe” to decrypt files.
Run the executable with the correct parameters extracted from the “AndroidMainfest.xml” file against the encrypted archive files.
Extract the decrypted file archives to disk.
Proceed with static source code analysis.
Dynamic analysis can proceed as usual using ADB, supplemented with GDB for analysis of native code execution if desired.

Conclusion

The Kony Studio IDE, in its current state provides a suitable platform for cross-platform development, as well as a formidable adversary for application auditors and analysts. While ultimately the obfuscation methods of the IDE were defeated, there hardly exists a runtime framework that would not be defeated, given an untrusted environment such as a mobile device. It is also clear that the Kony team is steadily updating its IDE processes and framework, and attention has been focused on protecting developer and company intellectual property.

One concern that should be addressed, or at the very least mentioned, is the decision to use shared object library files. While this move makes current Kony Applications considerably faster than their Lua Bytecode VM counter parts, it could also potentially introduce a serious vulnerability in the end-user mobile devices.

By relocating the bulk of the application execution to native code execution within the library through JNI calls, rather than allow the Dalvik VM to isolate the application, any vulnerability within the Kony application may effectively be able to achieve native code execution. Malicious JavaScript, delivered from an attacker to an end-user through a MitM’d network connection, would be executing within the Kony JavaScript VM, which is essentially a V8 engine. This immediately gives that attacker native code execution and the ability to leverage the JNI calls of the shared object library file to perform whatever actions are supported by the wrapper. Due to the fact that the wrapper needs to contain a prototype of all possible actions in order to be effective, the attacker would be able to execute any action in the context of the user that the library is loaded as.

Although these attack vectors were not investigated during the course of this research, it should at least be a concern given the weaknesses of current Android application permissions and sandboxing.

About the Author

Chris Weedon is a Security Engineer with NCC Group. He is primarily tasked with network and web application penetration tests. However, he enjoys exploring new technologies and devices which draws him to mobile application testing and hardware hacking. Additionally, binary

analysis, reverse engineering, and exploit development have been areas of interest and continual study for him.