Hybrid Code Analysis versus State of the Art Android Backdoors

Mobile Malware is evolving… can the good guys beat the new challenges?

by Jan Miller (Joe Security LLC)

What you will learn…

How state-of-the-art Android Malware looks like
Why static analysis cannot beat modern obfuscation techniques
That dynamic and static analysis in combination is the next step in malware detection (Hybrid Code Analysis)

What you should know…

Basic knowledge of Android
Basic knowledge of Sandboxing Systems
Basic knowledge of Java

Contents

Introduction

Mainstream usage of handheld devices running the popular Android OS is the main stimulation for mobile malware evolution. The rapid growth of malware and infected Android application package (APK) files found on the many app stores is an important new challenge for mobile IT security. Sophisticated anti-reverse engineering techniques, such as encryption and heavy obfuscation, are becoming malware industry standard. In June, an unofficial, but popular app store released more than 50.000 new applications (AppBrain, 2013).

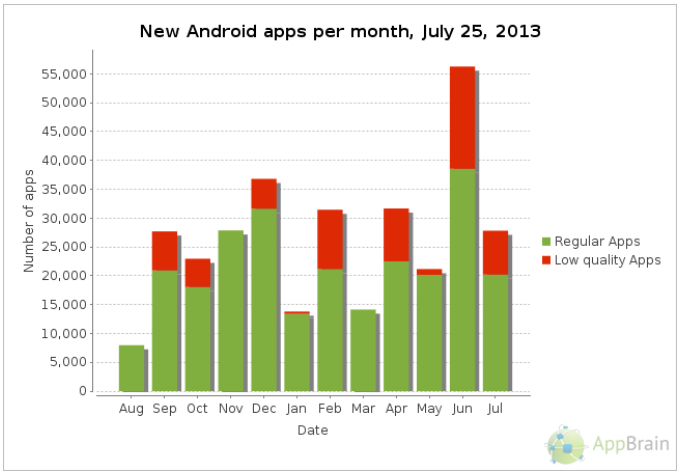


Figure 1: AppBrain New Applications Per Month Trend

The figure above outlines the rising trend of new application releases on AppBrain with a growing portion of low quality applications. About 13 billion APK file download have been registered worldwide up until today, while this is counting only the official app stores (AndroLib, 2013).

The problem we face today is that signature/pattern based detection methods that rely purely on static analysis, as implemented by most mobile anti-virus solutions, will fail in the long run, as heavy usage of java reflective invokes and encrypted data nullifies pure static analysis. Latest research is backing up this claim. Even the ten most common anti-virus applications are not resistant against simple transformation techniques, as has been shown by Rastogi et al. and their DroidChameleon framework (Rastogi, Chen, & Jiang, 2013). Of course, now one could assume that every application using heavy obfuscation is malicious, as it is obviously a clear indicator that something is trying to be hidden, but collective punishment is usually not a good idea. The reason for this being a weak criterion is the following: more and more legitimate commercial apps are implementing obfuscation techniques today to protect their intellectual property. Tools such as ProGuard obfuscate class names, method names; wrap all API calls in reflective invoke delegates to hide the real API name, et cetera. These tools are very easy to use, integrate seamlessly into the development process and popularity is growing, so it is necessary to develop stronger detection algorithms, in other words: new technology is required – and the end goal has to be malicious *behavior* detection, not pattern detection.

In this article we will first outline Android obfuscation techniques on real-world samples and outline why pure static analysis fails. Then, we will present a new technology called Hybrid Code Analysis (HCA) and

show how HCA overcomes all known obfuscation techniques and enables extraction of valuable analysis behavior data.

Terms and Definitions

In order to make the article as comprehensive as possible, the most important terms are outlined here.

Java Reflective Invokes

The Java Reflection API is originally intended to help programmers read "metadata" (like annotations or class/method names) or even change the state of objects not under direct control by setting fields or invoking even private methods. The "Uses of Reflection" is describes as the following:

"Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language." (Oracle, 2013)

First of all, as all Android Applications are based on Java code, the Java Reflection API can be used by developers in its full dimension. For malware authors and obfuscators in general, the most interesting API is the reflective invoke, because it is possible to wrap any API call in a sequence of calls from the Reflection API. First, an object of the target class is obtained using *java.lang.Class.forName(),* which in turn is used to obtain the correct method object with *java.lang.Class.getMethod()* followed by execution of the API using *java.lang.reflect.Method.invoke().* Tools that take source code as input and transform every API call into an equivalent instruction call sequence exist today. The effect is that the transformed code ends up calling only Reflection APIs and no other APIs, making static analysis difficult, as it requires analysis of the parameters and linking the method object lookup calls with the final invoke (could be spread across multiple classes). Obviously, this is not the intended use of the Reflection API.

DalvikVM

Dalvik Virtual Machine (DalvikVM) is a register machine developed to execute code in a virtual environment on mobile devices. It is a core component of the Android platform. Dalvik takes Java byte code (*.class* files) as an input and transforms it to its own byte code format (*.dex* files). As Dalvik is implemented as a pure register machine[1], it uses fewer resources and has a good performance. This is an important aspect, as every APK runs in its own virtual machine.

Application Package File

Android Application Package (APK) files are actually very similar to JAR files, as it uses the same "container" concept. An APK file is a ZIP file container including a single *classes.dex* file (multiple .class files merged by the *dx* optimizer), resources and a special binary XML manifest file that defines permissions, program entry points, event handlers and other metadata.

[1] compared to a stack machine, such as the JVM, although in the JVM each operation happens at a fixed location on the stack and can be mapped to a register with JIT should java byte code be executed on register based architectures

Android Obfuscation Techniques

In this chapter we will briefly outline the most common Android Obfuscation techniques that make static analysis and reverse engineering more difficult.

Random Symbol Names

One of the most typical obfuscation techniques is obfuscation of the class names, method names, field names, member variable names, and so on. As it is very easy to extract symbol information from Java byte code, symbol names are always included and not stripped as it is possible in other languages like C. If all symbols would be stripped, things like the Java Reflection API wouldn't work. In practice that means very random package/class/method names, as can be seen in the following figure:

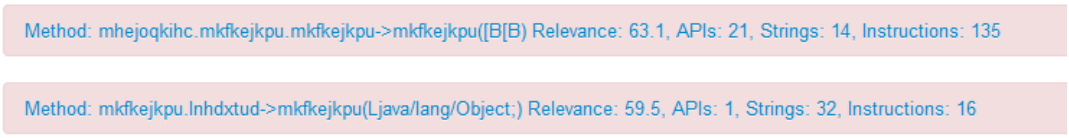


Figure 2: Random Symbol Names Distinguishable[2]

As we can see, it is quite difficult to tell the methods apart, because the same method name is being used in different classes. Looking at another sample, we can see that the method naming convention was evolved even further into enhancing obfuscation:

Figure 3: Random Symbol Names Non-Distinguishable[3]

Here, the random character set consists only of three characters "C", "I" and "O" in their different cases, the method names differ by their class name only, essentially not only making the methods non-distinguishable, but potentially misleading analysts through mix-ups. Understandably, reverse engineering the sample becomes quite difficult and one could describe this technique as "symbol stripping", as all useful descriptive symbol names are unreadable character-junk.

String Encryption

Encrypted strings make it very difficult to understand disassembly code, for example, as reflective invokes use strings as parameters in the class/method/field lookup code. Without that information it is

[2] Sample MD5 001a42a555b4bd39bf6ecd8b11441870
[3] Sample MD5 e1064bfd836e4c895b569b2de4700284

not possible to know by static analysis on what class/method a reflective invoke is operating. In other words, analysis without execution becomes extremely difficult.

17	const-string v1, "+s<e3-<:j.6.Fr3s>:^.6.Fr3s>5+s+H.e"	

19	invoke-static {v1}, Lmkfkejkpu/mkfkejkpu;->mkfkejkpu(Ljava/lang/String;)Ljava/lang/String;	• Time: 148731
		  • param0: +s<e3-<:j.6.Fr3s>:^.6.Fr3s>5+s+H.e
		  • Return:
		  • android.telephony.TelephonyManager

20	move-result-object v1	
22	invoke-static {v1}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;	
23	move-result-object v1	
25	const-string v2, "H.jA.J-].u<"	

27	invoke-static {v2}, Lmkfkejkpu/mkfkejkpu;->mkfkejkpu(Ljava/lang/String;)Ljava/lang/String;	• Time: 148743
		  • param0: H.jA.J-].u<
		  • Return:
		  • getDeviceId

Figure 4: Retrieving TelephonyManager and getDeviceId strings through decryption

The above figure demonstrates how important it is to have live data when understanding execution flow. Using pure static analysis, it would require reverse engineering the decryption routine, in order to obtain the decrypted payload (in this case the call to "*mkfkejkpu.mkfkejkpu->mkfkejkpu*" on line 19, Figure 4). Should the decryption routine furthermore require live data (data retrieved during execution), for example, loading a secret key stored on some web page, it becomes nearly impossible to understand execution flow with static tools alone. Crucial parts of the program behavior rely on strings, be it for reflective invokes, Web URLs or C&C server commands. This becomes extremely important, if all API calls are wrapped by reflective invokes (heavy obfuscation). That is why dynamic runtime analysis is becoming a very important tool to work against obfuscation, as string encryption is a widespread common technique today.

Wrapping API calls with reflective invokes

As mentioned already, reflective invokes allow “masquerading” the real API call when using encrypted strings in the lookup code. In the following figure we can see a very good example of how static analysis fails producing anything useful for an analyst or automatic detection algorithm:

```
15  const/16 v2, 0x10
16  const/4 v3, 0x0
18  invoke-static {v0, v2, v3}, Lcom/android/system/admin/IccIOIO;->oCIICII(III)Ljava
    /lang/String;
19  move-result-object v0
21  invoke-static {v0}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
22  move-result-object v0
23  const/16 v2, -0x13
24  const/16 v3, 0x45
25  const/4 v4, -0x4
27  invoke-static {v2, v3, v4}, Lcom/android/system/admin/IccIOIO;->oCIICII(III)Ljava
    /lang/String;
28  move-result-object v2
29  const/4 v3, 0x0
31  invoke-virtual {v0, v2, v3}, Ljava/lang/Class;->getMethod(Ljava/lang/String;[Ljava
    /lang/Class;)Ljava/lang/reflect/Method;
32  move-result-object v0
33  const/4 v2, 0x0
35  invoke-virtual {v0, v1, v2}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[Ljava
    /lang/Object;)Ljava/lang/Object;
```

Figure 5: Reflective invoke masquerades real API call

In the disassembly excerpt above, the local method invokes at line 18 and line 27 return encrypted strings that are used for the lookup calls to *java.lang.Class.forName()* and *java.lang.Class.getMethod()*. It is not deductible without execution what the actual API call at line 35 really is. Technology that combines static with dynamic analysis is needed.

Hybrid Code Analysis

Hybrid Code Analysis (HCA) is the new analysis technology that was briefly mentioned in article's introduction. In general, HCA means using static code analysis (analysis of disassembly code without execution) and dynamic code analysis (logging executed behavior through instrumentation, various implementations) in an intelligent way so that code coverage and dormant code detection is optimized. An important part is linking dynamic runtime data with the according disassembly code, thereby revealing hidden API calls in full context and all input/output data at parameter level (e.g. a decrypted string). For example, static analysis might retrieve interesting event handlers from the Manifest file prior execution, forward that information to the Sandbox and thereby help generate simulation events to maximize code coverage and trigger as much payload as possible during runtime. In other words, HCA takes the best of both worlds to improve overall malware analysis in a way superior to the techniques if they were used alone.

Using HCA to decrypt strings

Let us take a look at a good example to understand what this means: *Opfake.C*[4] is a SMS based Trojan for Android that uses String encryption heavily. Often, string decryption routines follow the same scheme and their function signature looks as following:

```
static String DecryptRoutine(String encryptedString)
```

In order to extract dynamic data from the target

This function signature translates into the following HCA directive:

```
__STATIC____ANYLOCALCLASS__;->__ANYFUNC__(Ljava/lang/String;)Ljava/lang/String;
```

Figure 6: Local DecryptString function configuration option

The above configuration option will tell HCA to log all method calls for methods that are static (see *__STATIC__* keyword), located in any class (see *__ANYLOCALCLASS__* keyword, which means any class declared in the *classes.dex* file), of any name (see *__ANYFUNC__* keyword, as the exact method name is not known ahead of time) and with the requirement of taking a *java.lang.String* object as single parameter and returning a *java.lang.String* object. This special configuration is quite specific, but flexible enough to intercept most String decryption routines without spamming the engine with too much logging data.

Running *Opfake.C* with the engine configured as above, a lot of strings are suddenly decrypted. Here, the String "3F.so3ss.]j-3s" translates to "openConnection" and the DecryptString routine that is used at hundreds of code locations is the static function "mkfkejkpu" at package "mkfkejkpu", class "mkfkejkpu".[5]

[4] Sample MD5 001a42a555b4bd39bf6ecd8b11441870

[5] The referenced report is available online at www.joesecurity.org if you navigate to the sample reports.

Figure 7: Decrypted String "openConnection"

The decrypted string is information that would have been hidden, if analyzed without HCA and without such flexible configuration options, such as the template-style logging directives. Of course, should one discover an interesting function call during analysis that is not being instrumented, it is possible to update the configuration and rerun the sample for more live data extraction. Directly following the string decryption, the decrypted string is used as a parameter for *java.lang.Class.getMethod()*:

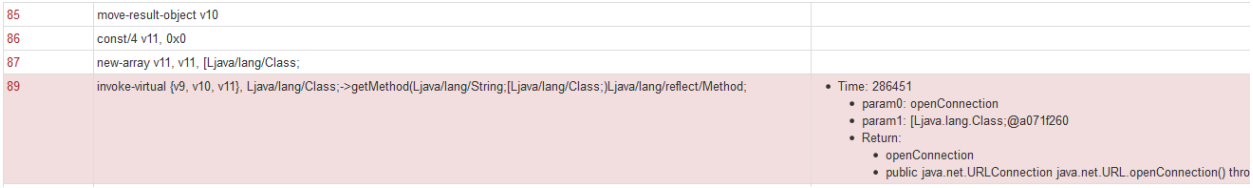


Figure 8: Decrypted String used for "getMethod" call

As the default configuration instruments all important java reflective API functions, the runtime data is available at this point and reveals the real API call. Reflective invokes are not that bad after all.

Using HCA to de-mask reflective invokes

As already mentioned, using reflection it is possible to masquerade the real API calls. As HCA remembers all java objects returned by invokes, it is easily possible to make a full association for all reflective invokes using known objects, thereby revealing the real API being called:

Figure 9: Reflective invoke resolved

As we can see in the figure above, the otherwise useless reflective invoke becomes valuable information when connecting dynamic data back to the disassembly. Suddenly it becomes a lot easier to understand the entire function (this is a good example of what Hybrid Code Analysis is all about).

Using HCA to analyze a State of the Art Android Backdoor

Let us take a look if HCA is useful on a real world, state of the art malware sample. Recently we came across a blogpost by Kaspersky (Unuchek, 2013) that introduces its readers to a new Android Backdoor Trojan as "*The most sophisticated Android Trojan*" with the name *Obad.a*, so we got curious to see whether or not HCA would be able to handle the APK[6] with the same techniques outlined in the previous chapters. Here is just a small portion of the analysis results (full details available at our company page) that shows one interesting aspect:

10	invoke-static {v1}, Lcom/android/system/admin/ocOlclCo;->ooCclcC(Ljava/lang/String;)Ljava/lang/String;	• Time: 144500 • param0: [B@a06aa5f0 • param0: su -c 'id' • param0: 7375202D632027696427 • Return: • su -c 'id' • Time: 144500 • param0: eCZyf2UidGhllw== • Return: • su -c 'id'
11	move-result-object v1	
13	invoke-virtual {v0, v1}, Ljava/lang/Runtime;->exec(Ljava/lang/String;)Ljava/lang/Process;	• Time: 144551 • param0: su -c 'id' • Return: • Process[pid=2369]

Figure 10: Superuser Shell Invoke

In the figure above we see the "DecryptString" function call (instrumented generically in the same way as outlined earlier) returning "su -c 'id'" and passing the string to *Runtime.exec()*. It is an attempt to create a superuser shell.

Of course, in order for dynamic analysis to work, it is crucial that the target sample executes interesting payload. That is why the Sandbox is able to simulate predefined events, like incoming phone calls or an incoming SMS, in order to trigger as much payload as possible. Analyzing *Pincer.A*[7], another SMS based Trojan, showed that the malware is able to receive JSON object commands via SMS text and then executes the associated command handler accordingly. Using a custom "cookbook" (sequence of commands to execute during runtime) we were able to emulate a C&C server instructing our APK to execute a specific command handler. The full command table includes:

start_sms_forwarding	start_call_blocking	stop_sms_forwarding	stop_call_blocking
send_sms	**execute_ussd**	**ussd_query**	**simple_execute_ussd**
stop_program	**show_message**	**delay_change**	**ping**

Using the following commands

- `_JBSimulateIncomingSMS('0123456789','{""result"":""true"",""command"":""start_call_blocking"",""phone_number"":""+41987654321""}')`
- `_JBSimulateIncomingCall('+41987654321')`

we were able to trigger the phone call blocking code that in turned revealed a nice trick:

[6] Sample MD5 e1064bfd836e4c895b569b2de4700284
[7] Sample MD5 f05839eb7156b434a893bbeddb68ad85

18	invoke-virtual {v1, v2, v3}, Ljava/lang/Class;->getDeclaredMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;	• Time: 173990 • param0: getITelephony • param1: [Ljava.lang.Class;@a068c140 • Return: • getITelephony • private com.android.internal.telephony.ITelephony android.telephony.T
19	move-result-object v1	
20	const/4 v2, 0x1	
22	invoke-virtual {v1, v2}, Ljava/lang/reflect/Method;->setAccessible(Z)V **Allow access to private method**	
23	new-array v2, v4, [Ljava/lang/Object;	
25	invoke-virtual {v1, v0, v2}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;	• Time: 173991 • param0: android.telephony.TelephonyManager@a06786a8 • param1: [Ljava.lang.Object;@a068bac0 • Return: • com.android.internal.telephony.ITelephony$Stub$Proxy@a06c1990
26	move-result-object v0	
27	check-cast v0, Lcom/a/a/a/d; **Cast ITelephony interface to custom interface to "obfuscate" interface access**	
28	return-object v0	

Figure 11: Accessing the *ITelephony* private interface

In the figure above, we see how the call blocking works. The call blocking is implemented by retrieving the private *ITelephony* interface and then using a private method of the TelephonyManager *getITelephony*, which in turn allows execution of *ITelephony.endCall()* silently. If any sample is found retrieving the *ITelephony* interface in a masquerading way (using reflection), one of the configurable HCA signatures will trigger and mark the sample as malicious:

May block phone calls / Accesses private ITelephony interface	Hide sources
Source: com.security.cert.services.PhoneCallReceiver;->b:21	API Call: java.lang.Class.getDeclaredMethod("getITelephony")

Figure 12: Accessing private *ITelephony* interface Signature

The figure above shows a signature that indicates malicious behavior by the red color and conveniently references the source code location, as well. The package, class, method and line number is available and links the user directly to the disassembly code through an URI.

Using HCA to reveal emulator detection

The Reflection API can not only be used to masquerade reflective invokes, but also field accesses. In an analysis of the *Obad.a* sample mentioned previously, we found an interesting code location:

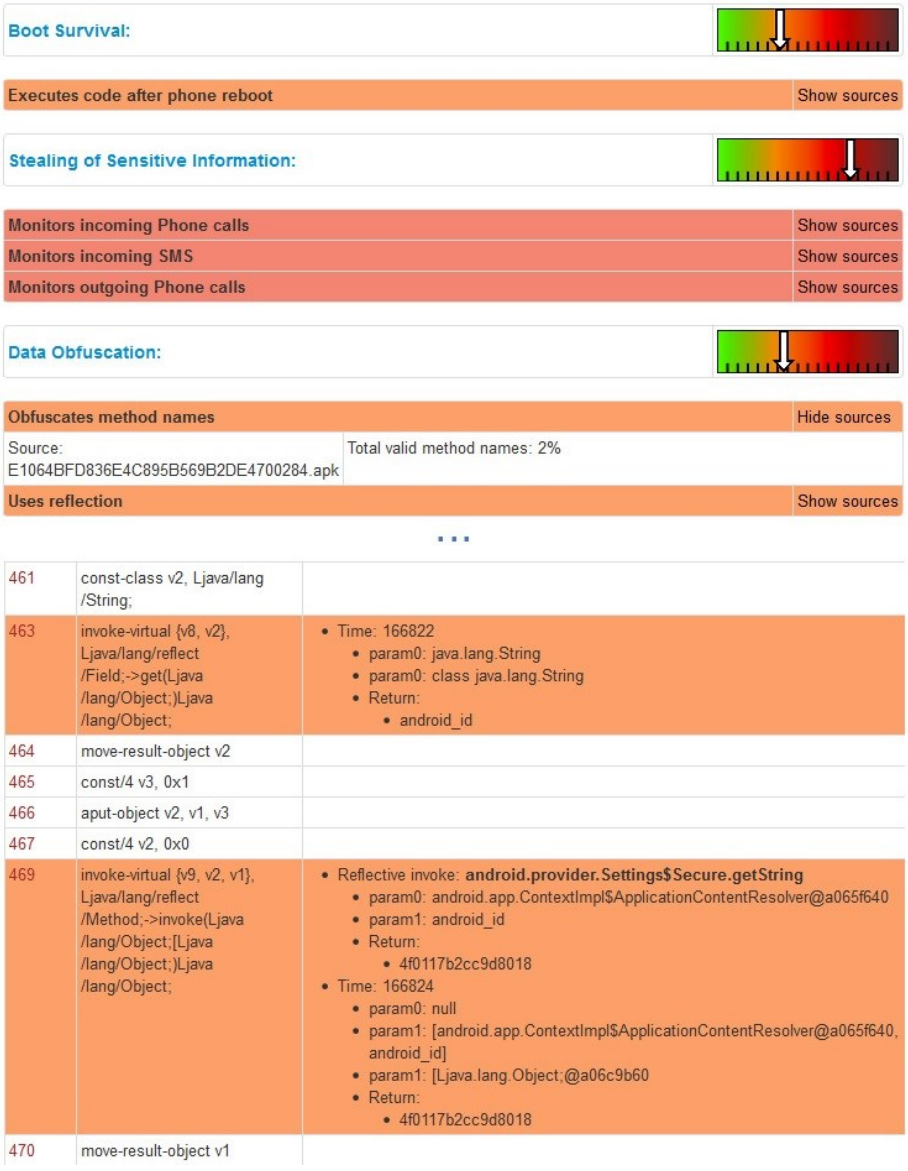
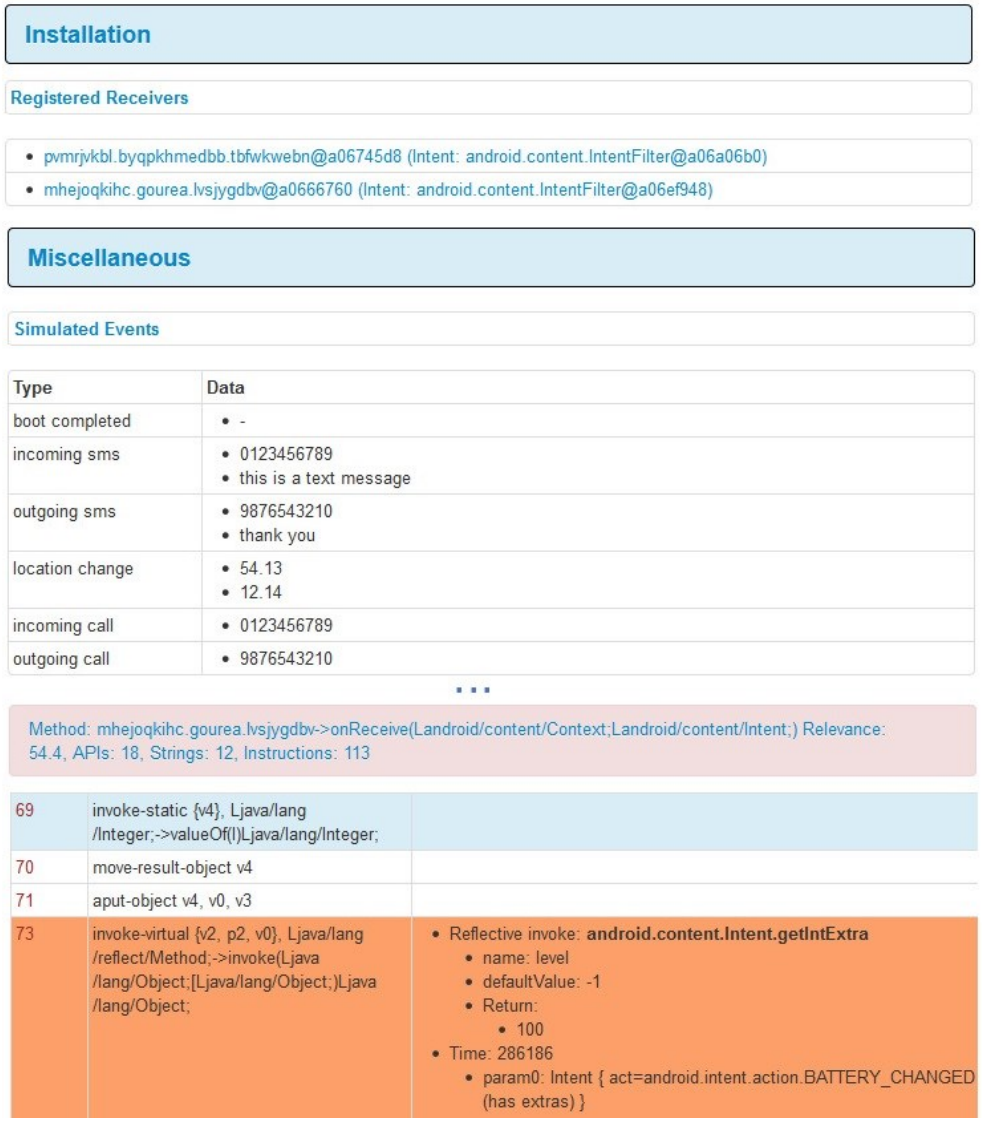


Figure 13: Reflective field access to lookup unique device identifier

As we can see in the figure above, a field value (in this case "android_id") is retrieved via reflection and then a reflective invoke to *android.provider.Settings.Secure.getString()* is used to get a unique device identifier that is valid for the lifetime of a device. This could be used to detect the execution environment, as the "android_id" is usually null on emulators and might cause the sample to skip executing the real payload. An otherwise common technique to detect an emulator is querying the IMEI using *TelephonyManager.getDeviceId*. Again, only technology such as HCA allows us to detect this trick and react accordingly by spoofing the "android_id" with a random value at startup, for example.

Using HCA to improve Code Coverage

Using static and dynamic analysis results, most often receivers and their intent filters defined in the *AndroidManifest.xml* file statically and registered receivers during runtime dynamically, it is possible to simulate targeted events to trigger as much as payload as possible. The more code is executed, the more dynamic data can be combined with disassembly code and the stronger HCA effects analysis results in a positive way. API call chains, parameter data, object information is combined and evaluated by behavior signatures and help analysts or machine programs obtain a deep understanding of the target sample. Let us take a look at a malware sample to demonstrate the power of HCA. Analyzing *Opfake.C* (report available on our company webpage) we can see the following data in the report (an excerpt):



Installation

Registered Receivers

- pvmrjvkbl.byqpkhmedbb.tbfwkwebn@a06745d8 (Intent: android.content.IntentFilter@a06a06b0)
- mhejoqkihc.gourea.lvsjygdbv@a0666760 (Intent: android.content.IntentFilter@a06ef948)

Miscellaneous

Simulated Events

Type	Data
boot completed	• -
incoming sms	• 0123456789 • this is a text message
outgoing sms	• 9876543210 • thank you
location change	• 54.13 • 12.14
incoming call	• 0123456789
outgoing call	• 9876543210

...

Method: mhejoqkihc.gourea.lvsjygdbv->onReceive(Landroid/content/Context;Landroid/content/Intent;) Relevance: 54.4, APIs: 18, Strings: 12, Instructions: 113

69	invoke-static {v4}, Ljava/lang /Integer;->valueOf(I)Ljava/lang/Integer;	
70	move-result-object v4	
71	aput-object v4, v0, v3	
73	invoke-virtual {v2, p2, v0}, Ljava/lang /reflect/Method;->invoke(Ljava /lang/Object;[Ljava/lang/Object;)Ljava /lang/Object;	• Reflective invoke: **android.content.Intent.getIntExtra** • name: level • defaultValue: -1 • Return: • 100 • Time: 286186 • param0: Intent { act=android.intent.action.BATTERY_CHANGED (has extras) }

Figure 14: Sample Report with Simulated Events

As we can see in the above figure, six simulated events were sent to the device ("boot completed" event, an "incoming SMS", an "outgoing SMS", et cetera) during execution. Every simulated event will

be consumed by the application if an appropriate receiver exists. In this case, a receiver was installed during runtime (the "register receiver" APIs are being hooked by the engine) and the simulated "boot completed" event caused execution of the *onReceive* method in the class *mhejoqkihc.gourea.lvsjygdbv*. The real API call is wrapped in a java reflective invoke, but the dynamic runtime data easily reveals what is happening. In this case, we see that the application is trying to read the battery changed value. This could be a sandbox system/emulator detection method, as the battery value on an emulator is usually the same on a default installation. Usually, APK emulation within a malware detection system would only execute for a short period of time, so that the battery level will always be the same initial value set by a preconfigured snapshot/default initial state. Only on a real native device would the battery value fluctuate strongly between shutdown and power up. Again, these conclusions could only be drawn using technology such as HCA.

Conclusion

We learned that heavy string obfuscation and reflective invokes are a major challenge for static analysis. In order to overcome obfuscation and the restrictions of static analysis, a Sandbox system for dynamic analysis is required. In the best case, static analysis helps dynamic analysis achieve even better results and vice versa. The requirements are:

- **Fine-Grained data logging**: A sandboxing system that gathers parameter data and return values of instrumented methods at a very low level.
- **Logging flexibility**: A powerful, generic instrumentation engine, i.e. the ability to instrument/log even user-defined methods to observe not only API calls, but get a hold of data generated by interesting local methods as well.
- **Context sensitivity**: Intelligent algorithms that link java objects and other dynamic data together to better understand the context of API calls and resolve reflective invokes.
- **Optimized code coverage**: In order to improve code coverage overall, results of static analysis prior execution should influence targeted event simulation (for example, generating events that are known to be consumed by a service).

A modern and successful Sandbox system should fulfill at least these requirements.

Summary

In this article we started out by outlining the challenges of Android Malware analysis in an environment that is evolving rapidly. We showed that heavy obfuscation is becoming a mainstream phenomenon and new technology is necessary to overcome the challenges present. String encryption and reflective invokes are very effective tools against pure static analysis and pattern detection. We introduced a new technology called Hybrid Code Analysis (HCA) that combines dynamic and static analysis in a very fine-grained, flexible and context-sensitive manner. Using HCA, all known common obfuscation techniques are overcome and using code coverage optimizing algorithms even more interesting behavior is revealed as otherwise possible. The effectiveness of HCA was demonstrated on a variety of use-cases and samples. Furthermore, HCA results are evaluated at a high level using generic behavior signatures that abstract from specific malware variants and obfuscation techniques. Thereby, malicious behavior can be detected in a very general way making reliable, long-term malicious code detection possible that is immune to obfuscation techniques. Be it in the wild or not.

About the Sandbox

The analysis system used in this article is Joe Sandbox Mobile (Joe Security LLC, 2013), which analyzes APK files in a controlled environment and monitors the runtime behavior for suspicious activities. All activities are compiled to comprehensive and detailed analysis reports. These reports contain key information about potential threats and enable cyber-security professionals to deploy, implement and develop appropriate defense and protection strategies. Hybrid Code Analysis technology and its framework is a core part of Joe Sandbox Mobile.

On the Web

Android malware analysis with Joe Sandbox Mobile is also available as a free service at www.apk-analyzer.net.

About the author

Jan Miller is a specialist for Reverse Engineering, Static Binary Analysis and Malware Signature algorithms working at Joe Security LLC, which is a globally operating, well positioned software company based in the center of Europe – Switzerland. Currently, he is researching new trends, such as dynamic and static analysis of Android based malware.

Table of Figures

Citations

AndroLib. (2013, June). *Android Market statistics from AndroLib, Androlib, Android Applications and Games*. Retrieved from http://de.androlib.com/appstats.aspx

AppBrain. (2013, June). *Number of available Android applications*. Retrieved from http://www.appbrain.com/stats/number-of-android-apps

Joe Security LLC. (2013, July). *JOE SANDBOX MOBILE - The most advanced analysis tool for Mobile Applications is now at your disposal!* Retrieved from http://www.joesecurity.org/joe-sandbox-mobile

Oracle. (2013, July). *Trail: The Reflection API*. Retrieved July 2013, from The Java Tutorials: http://docs.oracle.com/javase/tutorial/reflect/

Rastogi, V., Chen, Y., & Jiang, X. (2013). *Evaluating Android Anti-malware against.* Northwestern University, North Carolina State University.

Unuchek, R. (. (2013, June). *The most sophisticated Android Trojan*. Retrieved from http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan