Small footprint inspection techniques for Android

Damien Cauquil Pierre Jaury

September 21, 2012

1 Reverse engineering and motivations

With mobile devices getting more complex everyday, users tend to store huge amounts of data and access so many services on potentially insecure networks and systems that mobile security is one of the main concerns faced by development companies and IT security experts nowadays.

Meanwhile, both for security reasons and intellectual property protection, developers are provided with a panel of optimization and obfuscation tools [2] that is getting powerful and fairly easy to include in any release process. Reverse engineering binary packages has become a full time job for security consultants, who are lacking some tools when dealing with very specific issues.

In order to completely understand motivations for small footprint inspection techniques, one first has to compare reverse engineering with physics. Reverse engineering *is* the physics of computers: experts are collecting facts and observing behaviors to establish laws and analyze system internals that could no be observed directly. Those same experts are facing numerous experimental issues, especially when studying programs specifically designed against reverse engineering techniques. One of them is very common to physics and computers: experimental and measure uncertainty.

When performing any run-time dynamic analysis, reverse engineers modify the application behavior by altering its environment: debugging meta-data, runtime breakpoints, virtual machine overlay, physical device emulator and even network traffic interception may end up in a complete different response from the target application. Studying the program necessarily involves a bias; developers and specific anti-debugging tools exploit this bias to slow down reverse engineers or lead them to wrong conclusions.

Current tools available for the Android mobile platform usually have many side effects: their footprint is so big that dynamic analysis of mobile applications is sometimes impossible. This observation motivated various research projects for dynamic analysis – mostly inspection – techniques involving a minimal footprint.

2 Small footprint inspection

2.1 Android inspection state of the art

Many tools are available for memory and execution path inspection of Android applications. The most common one is DDMS (Dalvik Debug Monitor Server), it is perfectly integrated with development environments like Eclipse and allows developers and auditors to place breakpoints, inspect both local and global variables. Yet the application has to be launched in debug mode (if not built with the debug flag).

One of the latest tools released is APKIL. It provides auditors with a complete Dalvik byte-code patching system that is able to inject monitoring instructions into application packages. Its main purpose is the inspection of Android API calls, which – as any system call – are usually perfectly relevant for analyzing internal mechanisms. It is still easily beaten by loading remote code at run-time or by spoofing usual API calls.

2.2 Service injection

The techniques we used to circumvent annoying side effects and anti-debugging protections are based on a very simple principle that malware developer already have widely explored [1]: Android applications are built upon a modular architecture, declaring possibly unrelated activities, services, etc. Thus, injecting code into an application package does not necessarily mean altering the existing Dalvik byte-code.

We tried and exploited many injection vectors, from supposed static resources to fully equipped services, and ended up dropping a service that remains completely silent until it is enabled and queried by a client application.

The injected piece of code communicates using standard service calls as a covert channel in order to grant users the ability to inspect the application memory from the inside and execute any Dalvik instruction in the same process and virtual machine as the target application. It is also able to load dynamic classes at run-time – in a very similar fashion as Meterpreter – in order to extend its functionality while keeping a minimal space footprint.

2.3 Introspection API and examples

The tool we eventually developed exposes a simple service API that may be proxified over the network and integrated in the same fashion as DDMS plugins.

It is able to perform complete activities and running services introspection, variables modification and remote method invocation as well as downloading and invoking user-defined Java/Dalvik macros at run-time.

It will be released together with example client applications. See figure 1 for the very first glimpse of an introspection client for Android.

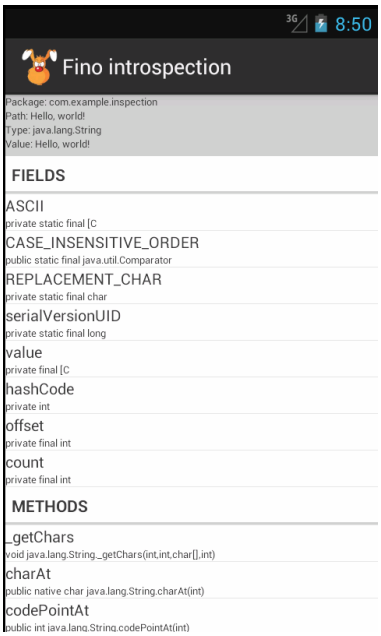

Figure 1: Fino inspection client

3 Presentation contents

The presentation would be held in English, last about 30 minutes and address the following topics:

- general considerations and real life examples about the importance of mobile security and reverse engineering on mobile platforms;
- existing inspection tools and examples of their usual side effects;
- minimal footprint inspection principles and overview of the technical implementation we came up with;
- demonstration of the released tool abilities using a couple of target applications and the Android introspection client (as well as a DDMS-like plugin if ready soon enough).

References

[1] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. 2011.

[2] Patrick Schulz. Code protection in android. 2012.