Tom CZAYKA
tczayka@quarkslab.com

Why are Frida and QBDI a Great Blend on Android?

Pass The Salt - June 2020

quarkslab
SECURING EVERY BIT OF YOUR DATA

$ whoami

Tom CZAYKA (@bla5r)
Security engineer at Quarkslab
Mostly into reverse engineering and everything related to Android

Table of Contents

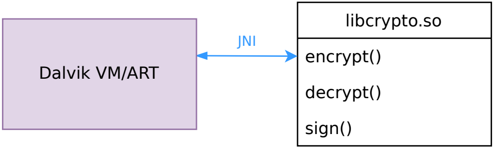
Dalvik/ART ecosystem

- When building an application, Java/Kotlin code is compiled into **Dalvik bytecode**
- Dalvik bytecode is stored in **Dalvik EXecutable** file(s), embedded in the final APK file
- **Dalvik VM** is responsible for executing Dalvik bytecode at runtime
- With **ART**, bytecode is compiled into machine code at installation (AOT) then run natively

Reverse engineering

DEX files can be easily decompiled in either Java (jadx) or smali (baksmali/apktool) representations. Doing so makes the reverse engineering process much more easier.

Java Native Interface



- Native development is still possible thanks to **Java Native Interface**
- Developers can call their own native functions from Java/Kotlin side
- JNI acts as a bridge between the Dalvik bytecode and the native code
- Code lies in shared libraries (*.so*), loaded alongside Dalvik VM/ART

Reverse engineering

Understanding a native function is more complicated since it implies reading through assembly code. Native decompilation is not as accurate as the Dalvik bytecode one.

Java code

Let's write a basic XOR function:

- Original source code

```java
public static void inPlaceXor(byte[] key, byte[] buffer) {
  for (int i = 0; i < buffer.length; i++) {
    buffer[i] = (byte)(buffer[i] ^ key[i % key.length]);
  }
}
```

- Decompiled code (jadx)

```java
public static void a(byte[] bArr, byte[] bArr2) {
  for (int i2 = 0; i2 < bArr2.length; i2++) {
    bArr2[i2] = (byte) (bArr2[i2] ^ bArr[i2 % bArr.length]);
  }
}
```

Significant differences

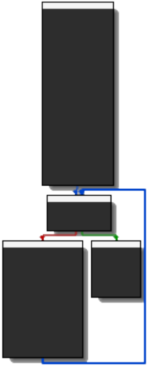
Logic remains the same, only function and variable names have been changed (Proguard).

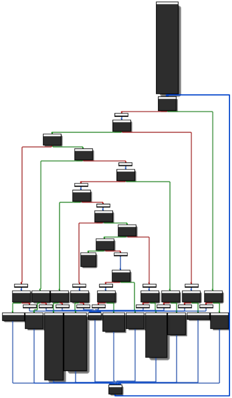
Native code

Let's now rewrite this function in C code:

```
void in_place_xor(const char *key, unsigned int key_len,
            char *output, unsigned int output_len)
{
  for (unsigned int i = 0; i < output_len; i++)
  {
    output[i] = output[i] ^ key[i % key_len];
  }
}
```

Graph view





- Without obfuscation
- With obfuscation (OLLVM)

Native debugging

GDB

LLDB

Common anti-debugging techniques

- Checking TracerPid in */proc/self/status*
- Child process attaching its parent

Developers usually take advantage of these techniques for preventing their applications from being debugged.

Table of Contents

Frida in a nutshell

FRIDA

- Created by @oleavr and @hsorbo
- https://github.com/frida/frida
- **D**ynamic **B**inary **I**nstrumentation toolkit
- Lets you inject arbitrary code into a process
- Core code written in C
- Several bindings on top (JavaScript, Python, ...)

Talking of Android

Widely used by Android reverse engineers thanks to its great integration and the convenience it brings.

In practice

- Find the address of *func_of_interest()*
- Attach the function thanks to the Interceptor module
  - Callback called **before** executing the function
  - Callback called **after** executing the function
- Print arguments and return value

```
var addr = Module.findExportByName("libjuicy.so",
           "func_of_interest");
Interceptor.attach(addr, {
  onEnter: function (args) {
    console.log("Entering func_of_interest(" +
                args[0].readCString() + ")");
  },
  onLeave: function (retval) {
    console.log("Return value: " + retval + "...");
  }
});
```

Limitations

We're here at the function level hence we can't really figure out what's going on inside.

Table of Contents

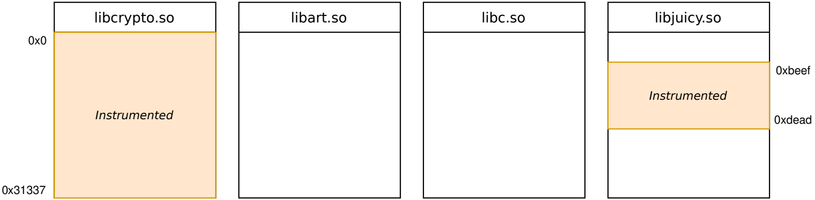
What's QBDI?

SECURING EVERY BIT OF YOUR DATA

- Initially developed by Cédric Tessier and Charles Hubain (Quarkslab)
- https://github.com/QBDI/QBDI
- LLVM-based **D**ynamic **B**inary **I**nstrumentation framework
- Designed to work on a lower layer (basic block/instruction scale)
- Provides C/C++ APIs
- **Frida integration**

Overall design

Instrumented ranges



- The QBDI engine will solely consider precise parts of the code
- Those parts users are interested in have to be defined as **intrumented ranges**
- A range can include the whole program's address space, an entire module or only a specific part of it

Overall design

Callbacks

- A callback is a user defined function that is called whenever coming across special conditions:
  - Before/after executing each instruction
  - Basic block discovery
  - Transfer execution to an uninstrumented part
- Users can register some specific **callbacks** depending on their needs

Code outside of instrumented ranges isn't considered

Callbacks won't be called if the current program counter points to an address which isn't included in a known range.

A demo is worth a thousand words

Initialisation

- Instanciate a QBDI VM
- Allocate the corresponding virtual stack

Analysis refinement

- Define instrumented ranges
- Set up callbacks

Function running

- Prepare registers and virtual stack with arguments according to the ABI
- Execute the target function through the QBDI context
- Retrieve the return value

Table of Contents

Real-world setting

Whatsapp 2.20.157
com.whatsapp

Scenario

- We have noticed an interesting library called *libwhatsapp.so*
- We would like to understand what this library is doing
- Let's dive in by looking into *JNI_OnLoad()*

Note

JNI_OnLoad() is responsible for initialisation. This function is always called right after the library loading.

Instruction tracing

Approach

Goal: recording every single executed instruction could allow us to get a thorough understanding of what this function is actually doing.
Idea: instead of letting the function run as usual, let's execute it in an instrumented context.

How to set it up?

- Replace the genuine implementation of JNI_OnLoad() thanks to Frida's Interceptor.replace()
- The brand-new implementation is responsible for
  - initialising QBDI
  - defining the whole *libwhatsapp.so*'s address space as an instrumented range
  - declaring a callback which will be called before each instruction
  - synchronising the current CPU context with the QBDI one
  - executing the real JNI_OnLoad() through QBDI
- Forward the return value to properly resume the normal execution

Instruction tracing

Outcomes

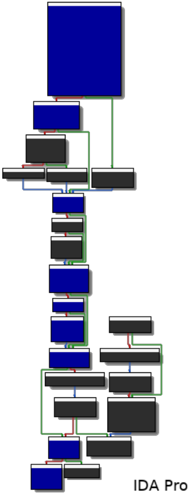
```
0x890a7edc    imul    dword ptr [esp + 4]
0x890a7ee0    mov     eax, edx
0x890a7ee2    shr     eax, 31
0x890a7ee5    sar     edx, 6
0x890a7ee8    add     edx, eax
0x890a7eea    mov     dword ptr [ecx + 4], edx
0x890a7eed    xor     eax, eax
0x890a7eef    mov     ecx, dword ptr [esi]
0x890a7ef1    cmp     ecx, dword ptr [esp + 12]
```

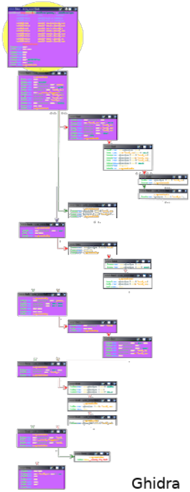
Useful but...

Knowing what instructions have been executed is valuable but not really convenient as it is.

What about integrating this information in our favourite disassembler like IDA Pro or Ghidra?

Code coverage generation





- Various plugins deal with code coverage such as Lighthouse or Dragondance
- Both require drcov files to work
- These files contain information about
  - Process' memory layout
  - Executed basic blocks
- Placing a QBDI callback which is called whenever a new basic block is discovered allows us to generate this file on our own

Stay tuned!

A follow-up blogpost coming soon on Quarkslab's blog:

https://blog.quarkslab.com

Thanks for listening!

Questions?

quarkslab

SECURING EVERY BIT OF YOUR DATA