## Quarkslab's website

### SOCIAL

- atom feed
- twitter
- github

### CATEGORIES

- Android
- Android, ReverseEngineering
- Challenge
- Cryptography
- Development
- Exploitation
- Fuzzing
- Hardware
- Hardware, ReverseEngineering
- Kernel Debugging
- Life at Quarkslab
- Maths
- Obfuscation
- PenTest
- Program Analysis
- Programming
- ReverseEngineering
- Software
- Vulnerability

### TAGS

# When SideChannelMarvels meet LIEF

Date Thu 03 May 2018 By Romain Thomas Philippe Teuwen Category Cryptography. Tags White-box AES DFA Cryptography Android LIEF

On how we used LIEF to lift an Android x86_64 library to Linux to perform our usual white-box attacks on it.

## Introduction

For those of you following our SideChannelMarvels [1], you know that whenever we stumble on a non-commercial white-box implementation, we like to add it to the Deadpool project, a repository of various public white-box cryptographic implementations and their practical attacks.

This time, we wanted to have a look at the white-box created by Sanghwan (h2spice) Ahn and proposed during SECCON2016 CTF [2]. Apparently only PPP solved it during the competition and Sanghwan wrote himself a write-up [3].

The challenge consists in an Android APK. When you launch it, it displays a flag encrypted a random number of times (between 1 and 3601). When encrypted only once, the flag is `g1UlZafiuGdCgpTkWYjaZg3kE6qCd7kF3kV+nMKcGHc=`.

To be able to ‘plug’ the challenge into our tools, we need to get an easy access to the input and output of the AES encryption function. A quick look reveals that the actual cryptographic operations are done in a native library called `libnative-lib.so`, conveniently available for several architectures. The function `TfcqPqf1lNhu0DC2qGsAAeML0SEmOBYX4jpYUnyT8qYWIlEq(unsigned char*, unsigned char*)` is the AES encryption function we want to attack. Note that the library is obfuscated with Obfuscator-LLVM 3.6.1, as we can see from its `.comment` section.

But we're lazy, so we'd like to reuse the x86-64 version of `libnative-lib.so` under a Linux environment, where all the SideChannelMarvels toolchain is ready to crunch white-boxes. That's not that simple because, even if they look alike, dynamic libraries compiled for Android or for Linux have a number of differences and a naive attempt to load an Android dynamic library under Linux will simply fail.

Fortunately, we have a nifty tool for parsing and modifying binaries. We're talking about LIEF [4] of course!

## Converting an Android library to Linux with LIEF

The white-box is implemented in the `libnative-lib.so` which is available for ARM, AMR64, x86 and x86-64 architectures. It's a tiny library exporting one **JNI** function: `Java_kr_repo_h2spice_crypto500_MainActivity_a` and importing three functions from external libraries.

Lifting this library to Linux is possible because the three imported functions (`__cxa_finalize`, `__cxa_atexit`, `__stack_chk_fail`) are not specific to Android.

The linked libraries of `libnative-lib.so` are *standard*: `libc`, `libstdc++` … except for `liblog`. But `libnative-lib.so` doesn't use any of `liblog` functions, as we can see in `readelf` output:

```
readelf -s -d -W ./libnative-lib.so

Dynamic section at offset 0x2ad00 contains 31 entries:
  Tag  Type    Name/Value
...
 0x01 (NEEDED) Shared library: [liblog.so]
 0x01 (NEEDED) Shared library: [libm.so]
 0x01 (NEEDED) Shared library: [libstdc++.so]
 0x01 (NEEDED) Shared library: [libdl.so]
 0x01 (NEEDED) Shared library: [libc.so]
...
Symbol table '.dynsym' contains 32 entries:
 Num: Value  Size Type    Bind   Vis      Ndx Name
   0: 00000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 00000     0 FUNC    GLOBAL DEFAULT  UND __cxa_finalize@LIBC (2)
   2: 00000     0 FUNC    GLOBAL DEFAULT  UND __cxa_atexit@LIBC (2)
   3: 04fe0   865 FUNC    GLOBAL DEFAULT   11 Java_kr_repo_h2spice_crypto500_MainActivity
_a
   4: 02070  2281 FUNC    GLOBAL DEFAULT   11 _Z48APtMDGO79Go3cbIkFca2rN0KszanZXOZ7dIPsxD
BletW5gdoPcPKci
   5: 01100  3916 FUNC    GLOBAL DEFAULT   11 _Z48DENCPKY6hzMem3SuzgIXu4u6vxbF1sajPOJ75aN
2VTdc7SCLPcPKc
   6: 00bb0  1345 FUNC    GLOBAL DEFAULT   11 _Z48KwUmSQBCaOVJKeqvABGpVnuErM7j8YCSOagNYBm
r2ah0NZBePKc
   7: 03860  6011 FUNC    GLOBAL DEFAULT   11 _Z48TfcqPqf1lNhu0DC2qGsAAeML0SEmOBYX4jpYUny
T8qYWIlEqPhS_
   8: 02050    30 FUNC    GLOBAL DEFAULT   11 _Z48h8AU0jPcyu9vXF9Kvg0bGDSl6H3TtcJIoOoU1ZO
ObCvegZ84i
   9: 00000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@LIBC (2)
  10: 02960  3836 FUNC    GLOBAL DEFAULT   11 _Z48lrsFdMdlAT0vSMVedxmqOkCBF7sCTbhCjYEp1rL
P8vatWEGDPh
  29: 2c008     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
  30: 2c008     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
  31: 2c050     0 NOTYPE  GLOBAL DEFAULT  ABS _end
```

Thus we can simply remove the `liblog` library by setting its dynamic tag to `DT_NULL`:

```python
import lief
libnative = lief.parse("libnative-lib.so")

liblog     = libnative.get_library("liblog.so")
liblog.tag = lief.ELF.DYNAMIC_TAGS.NULL
```

We also notice that the `libc` is named `libc.so` while the one on the current Linux version is named `libc.so.6`. To address this issue, one solution would be to create a symbol link of `libc.so.6` to `libc.so` and set the environment variable `LD_LIBRARY_PATH` to the directory that contains the symlink.

A more elegant solution is to rename the library with LIEF:

```python
libnative.get_library("libc.so").name = "libc.so.6"
```

Lastly, `libnative-lib.so` imports `__cxa_finalize`, `__cxa_atexit` and `__stack_chk_fail` with a specific version. The version can be seen in the imported names, next to the `@` character. For these symbols, the associated version is "LIBC" and, during the loading step, the loader will look for the `__cxa_finalize` in `libc.so.6` with **this exact version**.

But the Linux `libc.so.6` defines these symbols with a "`GLIBC_2.2.5`" version:

```
readelf -s -W /usr/lib64/libc.so.6|grep __cxa_finalize
1944: 00037cf0 535 FUNC GLOBAL DEFAULT 12 __cxa_finalize@@GLIBC_2.2.5
```

To fix the version issue, we can simply change the version to *unspecified* by setting its value to `1`:

```
for s in filter(lambda e: e.has_version, libnative.dynamic_symbols):
  if s.symbol_version.value > 1: # Library-defined version
      s.symbol_version.value = 1 # Set to unspecified
```

And then build the modified library:

```
libnative.write("libnative-fixed.so")
```

Finally, we can load and execute the lifted library with `dlopen` / `dlsym`: (error handling being stripped for readability)

```
using fnc_t = uint64_t(*)(unsigned char*, unsigned char*);

int main(void) {
  void* h = dlopen("./libnative-fixed.so", RTLD_NOW);
  void* sh = dlsym(h, "_Z48TfcqPqf1lNhu0DC2qGsAAeML0SEmOBYX4jpYUnyT8qYWIlEqPhS_");

  fnc_t AES_128_encrypt = reinterpret_cast<fnc_t>(sh);

  unsigned char plaintext[16];
  unsigned char ciphertext[16];
  fread(plaintext, 1, 16, stdin);
  AES_128_encrypt(plaintext, ciphertext);
  fwrite(ciphertext, 1, 16, stdout);
  return 0;
}
```

This native library has a special structure that enables the transformation:

1. It doesn't use functions specific to Android.
2. It doesn't use packed relocations.
3. It doesn't use exceptions.
4. It doesn't use Thread Local Storage (TLS).

The first point is very uncommon for JNI libraries and this transformation won't be possible for usual libraries.

# Eventually breaking the white-box

Now that we got a Linux binary of the AES white-box with standardized input/output, we're back into usual white-box attacks business. The Differential Fault Analysis attack on white-box using our tools is largely explained in a previous blogpost. In short, we inject statically some faults in the white-box tables (here, we'll shoot on the entire `.rodata` section of the dynamic library), execute the AES on a constant input, and observe the output for faults.

These steps are automated in the `deadpool_dfa.Acquisition` function, part of our SideChannelMarvels/Deadpool repository. Once we collected enough faulty outputs, we can apply a well-known DFA attack to recover the AES key, which is implemented in the `phoenixAES.crack` function from the SideChannelMarvels/JeanGrey repository.

```
#!/usr/bin/env python3

import deadpool_dfa
import phoenixAES

def processinput(iblock, blocksize):
    return (bytes.fromhex('%0*x' % (2*blocksize, iblock)), None)

def processoutput(output, blocksize):
    return int.from_bytes(output, byteorder='big', signed=False)

engine = deadpool_dfa.Acquisition(
    # main white-box executable
    targetbin='./main64',
    # file where to inject faults, and a reference copy
    targetdata='./libnative-fixed.so', goldendata='./libnative-fixed.so.gold',
    # hook to the DFA library, to validate faulty outputs
    dfa=phoenixAES,
    # hooks to process I/O as expected by the white-box executable
    processinput=processinput, processoutput=processoutput,
    # some tuning, telling we want to try up to single byte faults
    verbose=2, minleaf=1, minleafnail=1,
    # the libnative-fixed.so .rodata section address range
    addresses=[0x6350,0x2b490]
)
outputs = engine.run()[0][0]
phoenixAES.crack(outputs)
```

Execution:

```
...
Lvl 016 [0x000226DF-0x000226E0[ xor 0x86 -> B25BE351AD6986FF15D1E152E7802EC7 GoodEncFault
Column:1 Logged
Lvl 016 [0x000226DF-0x000226E0[ xor 0x69 -> B235E351806986FF15D1E1A4E780A6C7 GoodEncFault
Column:1 Logged
Saving 17 traces in dfa_enc_20180427_112029-112038_17.txt
Last round key #N found:
040D08DA68001026F3DC0D68897148B4
```

The DFA recovers the last (tenth) round key but the AES key schedule is invertible so we can go back to the original AES key:

```
$ aes_keyschedule 040D08DA68001026F3DC0D68897148B4 10
K00: 6C2893F21B6185E8567238CB78184945
```

The key falls in 10.2s and 3300 executions. This is indeed the correct AES key:

```
echo g1UlZafiuGdCgpTkWYjaZg3kE6qCd7kF3kV+nMKcGHc=|base64 -d|\
    openssl enc -d -aes-128-ecb -nopad -K 6C2893F21B6185E8567238CB78184945
```

```
SECCON{owSkwPeH1CHQdPV9KWrSmz9n}
```

## Final words

We hope this little exercise will make you feel like using our tools!

The whitebox and all the scripts to convert the library and apply the attack are available online [5] and LIEF has its own website [4].

Thanks to all Quarkslab colleagues who proofread this article and provided valuable feedback.

[1] Side-Channel Marvels repository, on GitHub.
[2] SECCON2016 Online CTF-Binary / Crypto500 Obfuscated AES, archived here.
[3] Sanghwan's Obfuscated AES Write-Up, in english, korean and japanese
[4] *(1, 2)* Library to Instrument Executable Formats
[5] SECCON 2016 Obfuscated AES artifacts in Deadpool

## Comments



**h2spice** • 3 years ago

Awesome! quarkslab

• Reply • Share ›