Black-Box Fuzzing for Android Native Libraries

GIAC (GMOB) Gold Certification

Author: Nawaf Alkeraithe, alkeraithe@gmail.com
Advisor: Christopher Walker
Accepted: November 22, 2021

Abstract

Many Android application developers are adopting C\C++ native language development in their Android mobile applications to exceed Java limits for performance issues. The use of native development in Android occurs by either using a known library in the application or using an in-house developed native library. Without the source code of native libraries, this could be a blind spot for penetration testers. Demonstrating the process of finding native functions, capturing a sample input data, and writing an Android application wrapper to implement and fuzz the native functions with AFL fuzzer may prove useful for mobile penetration testers to shed light on detecting memory management issues in Android.

1. Introduction

Android offers developers the possibility of developing part of their mobile applications in C and C++ languages through Android NDK. NDK enables the developers to manage memory more effectively and perform different tasks that could consume more resources and time through Java. The use of C\C++ in Android applications could be seen in Video and Audio calling applications such as WhatsApp or even through image processing applications such as Instagram.

With the freedom and flexibility offered by C\C++, the risk of memory management vulnerabilities arises. Security researchers were able to find RCE in GIF decoding library affecting WhatsApp CVE-2019-11932 (NIST, 2020a), and another RCE affecting Instagram caused by JPEG double-free vulnerability CVE-2020-1895 (NIST, 2020b) by fuzzing open-source libraries used by those applications (Gal Elbaz. Checkpoint, 2020).

In the case of open-source libraries, mobile penetration testers could search for known vulnerabilities affecting the used library or research for 0-days in the library through static code analysis or fuzzing. Many open-source libraries offer the possibility to cross-compile the library to be built on different architectures, thus making it easier to fuzz the library in a desirable environment. However, suppose the native library is closed-source or developed in-house by the application developers. In that case, this limits the options for mobile security analysts to examine the library for security issues.

This paper will demonstrate the process that could be followed to fuzz closed-source Android native libraries. The next sections will explain how to find native libraries and functions used by the application, how to capture a sample input that will be used as a seed for afl-fuzz, and then demonstrates how to write an Android wrapper application for the native library to be in the middle between the library and AFL.

2. Android Native Development

Android NDK (Native Development Kit) offers a range of tools for the developers to develop parts of their code in native C\C++ language (Google, n.d.-c). As developers build their code, the native code is built to native shared libraries (.so), and since Android devices are shipped on different CPUs, the native code will also be compiled for each architecture (Google, n.d.-b). Android currently supports 32-bit ARM, AArch64, x86, and x86-64 through ABI (Google, n.d.-a).

For the native code to interact with Java, developers use and implement JNI (Java Native Interface). JNI introduces different sets of data types that could be mapped to Java data types, as shown in Table 1.

Table 1: Primitive Types (Oracle, n.d.-b)

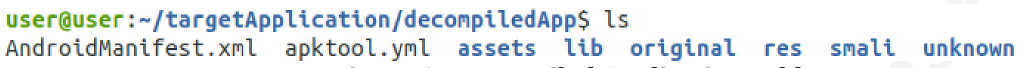
Java Type	Native Type
boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
void	void

There are two ways for the developer to declare the native methods used by the Android application through either statically following a naming convention in the native code or dynamically using the RegisterNatives method. Understanding how native functions are declared and identifying them is the first step for choosing our fuzzing targets. The following two subsections will avoid looking at native code from developers' perspective and how to develop a native code in Android applications. We will look at it from a decompiled application which is the point-of-view for most penetration testers.

2.1. Native Libraries

If there is a native code implementation, decompiling an APK file will reveal a "lib" folder where shared libraries are found. The "lib" folder may contain several sub-folders representing different compilation versions to support different CPU architecture.

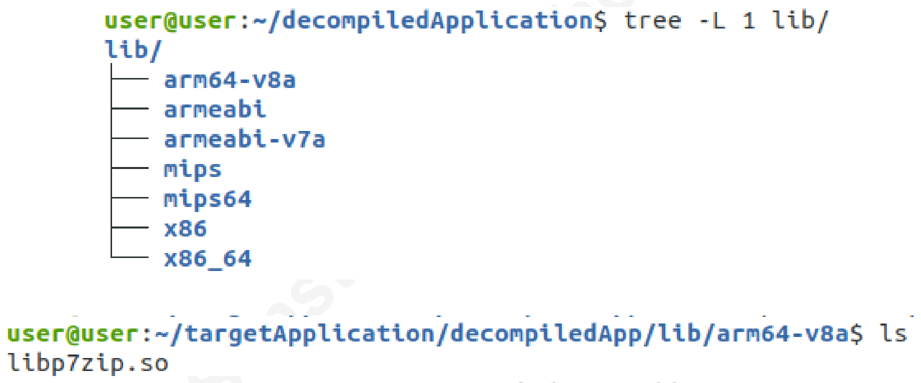
The following figure shows lib folder contents for a decompiled APK implementing native code, where the “lib” folder exists.



```
user@user:~/targetApplication/decompiledApp$ ls
AndroidManifest.xml  apktool.yml  assets  lib  original  res  smali  unknown
```

Figure 1: Decompiled Application Folder

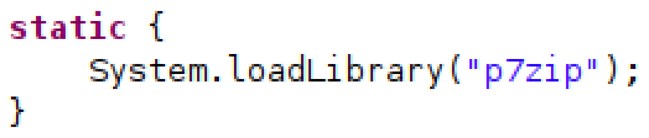
The contents of the “lib” folder show different versions of the same native code but compiled differently to support different CPU architectures.



```
user@user:~/decompiledApplication$ tree -L 1 lib/
lib/
├── arm64-v8a
├── armeabi
├── armeabi-v7a
├── mips
├── mips64
├── x86
└── x86_64

user@user:~/targetApplication/decompiledApp/lib/arm64-v8a$ ls
libp7zip.so
```

Figure 2: Contents of Library Folder

From the tool jadx-gui, the application is loading a 7z files extraction and compression library. Notice and compare the name between the following and the previous figure. The undecorated name is used as p7zip, which is mapped to libp7zip.so library file.



```
static {
    System.loadLibrary("p7zip");
}
```

Figure 3: Importing Native Library

2.2. Native Methods

After loading the native library, Android developers use the method System.loadLibrary() or ReLinker.loadLibrary() from Java/Kotlin to load their native library (Google, n.d.-d). Android developers have two ways to declare their native methods, either by declaring them in a specific naming convention or registering their methods using the RegisterNatives. To demonstrate how the naming convention works from JNI, we assume that we have the following native methods declared in a Java class:

Nawaf Alkeraithe, alkeraithe@gmail.com

```
package AppPkgName
class ApplicationClassExample {
…..
native String calculateToken(String userInput);
…
```

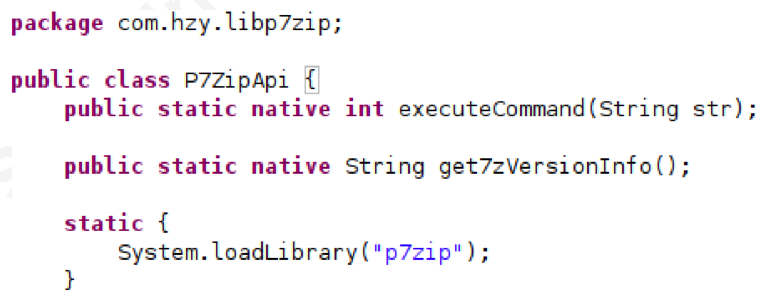
Then the native method declaration in C\C++ code would be as follow:

```
jstring Java_AppPkgName_ApplicationClassExample_calculateToken(JNIEnv *env,
jobject obj, jstring userInput)
```

The naming convention should be concatenated as below (Oracle, n.d.-a):

1- Prefix Java_
2- Package name + class name
3- Underscore "_"
4- Method name
5- If the methods are overloaded, then the method arguments should follow ("__").

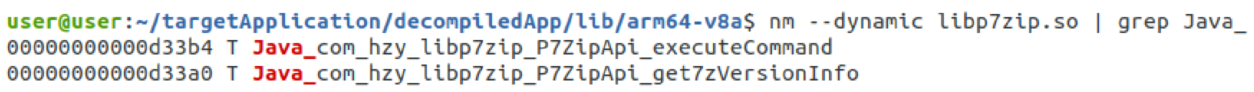
Figure 4 shows how the developer is declaring native methods used by the application.



```
package com.hzy.libp7zip;

public class P7ZipApi {
    public static native int executeCommand(String str);

    public static native String get7zVersionInfo();

    static {
        System.loadLibrary("p7zip");
    }
```

Figure 4: Native Methods Declaration

The methods declaration differs from the library side as the functions' names are mangled according to the JNI convention. Checking the native library in Figure 5 shows the method "executeCommand" with the method signature following the JNI naming convention.



```
user@user:~/targetApplication/decompiledApp/lib/arm64-v8a$ nm --dynamic libp7zip.so | grep Java_
00000000000d33b4 T Java_com_hzy_libp7zip_P7ZipApi_executeCommand
00000000000d33a0 T Java_com_hzy_libp7zip_P7ZipApi_get7zVersionInfo
```

Figure 5: Listing and Grepping Java_* Methods with "nm"

Even though this technique of declaring methods allows the developers to write less code, the exported native methods show more stack-trace error messages when an error or exception is thrown from the native code. Using the "nm" command and grepping for the “Java_” prefix from the JNI naming signature, we can find the native methods declared by JNI naming convention. From the decompiled APK, we can see what data type the native method is expecting in its arguments and what data type is returned.

For the second declaration method using the RegisterNatives method, developers do not need to follow the long convention name. However, they need to explicitly register each method the RegisterNatives. Now let us assume we have the following piece of code,

```
package AppPkgName
class ApplicationClassExample {
…..
native String getToken(String userInput);
```

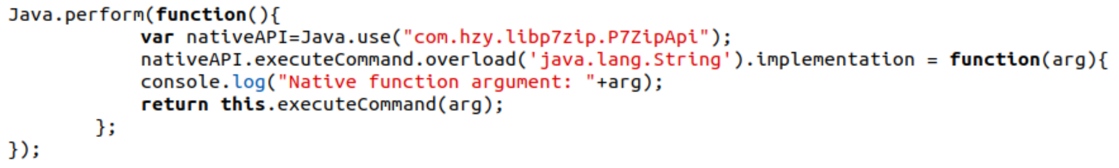
Then by using the RegisterNatives, the declaration of the native method could be as follows:

```
static const JNINativeMethod methods[] = {
            {"getToken",
"(Ljava/lang/String;)Ljava/lang/String;",(void*)getToken},
    };
    int rc = env->RegisterNatives(c, methods,
sizeof(methods)/sizeof(JNINativeMethod));
jstring getToken(JNIEnv *jenv,jobject obj,jstring userInput)
```

3. Capturing Sample Data

When trying to capture sample data of the input and output of the targeted native methods, Frida could be used to trace and monitor native methods. Android functions could be hooked with Frida to capture the input used later as a fuzzer sample.

The following Frida script hooks the native method as shown to grab a sample data that is to be provided to the native method.

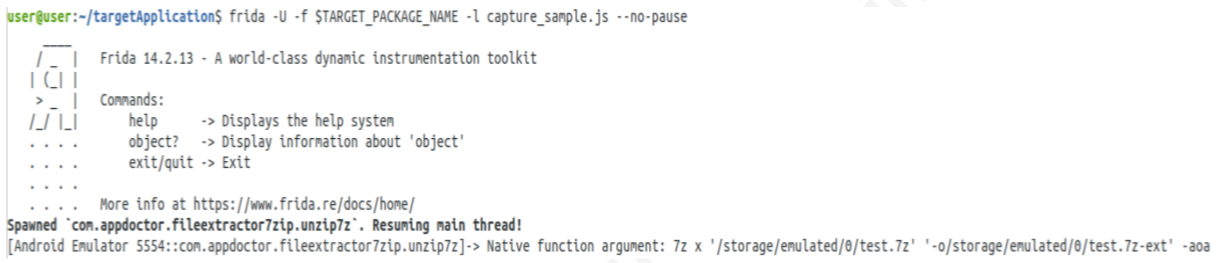


```
Java.perform(function(){
        var nativeAPI=Java.use("com.hzy.libp7zip.P7ZipApi");
        nativeAPI.executeCommand.overload('java.lang.String').implementation = function(arg){
        console.log("Native function argument: "+arg);
        return this.executeCommand(arg);
    };
});
```

Figure 6: Frida Script Hooking Native Method

Nawaf Alkeraithe, alkeraithe@gmail.com

After spawning the application with Frida, the input provided to the native function could be used as guidance to formulate an input to fuzz the targeted native method with AFL.



```
user@user:~/targetApplication$ frida -U -f $TARGET_PACKAGE_NAME -l capture_sample.js --no-pause
     ____
    / _  |   Frida 14.2.13 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/
Spawned `com.appdoctor.fileextractor7zip.unzip7z`. Resuming main thread!
[Android Emulator 5554::com.appdoctor.fileextractor7zip.unzip7z]-> Native function argument: 7z x '/storage/emulated/0/test.7z' '-o/storage/emulated/0/test.7z-ext' -aoa
```

Figure 7: Captured Input Sample from Frida

4. Wrapper Android Application

After identifying the native library and choosing its native function as a target for fuzzing, we need to prepare our fuzzer to fuzz the closed-source library. Two different main approaches could be taken; the first approach is to perform fuzzing on the target function through a Frida script that hooks the native target function. Python could be used to direct input from the terminal to an RPC exported Frida function. The second approach would be to write a harness that uses the exported target function in a standalone executable and then fuzz the harness with AFL. The figure below shows the relation of our harness between AFL and the target native method.

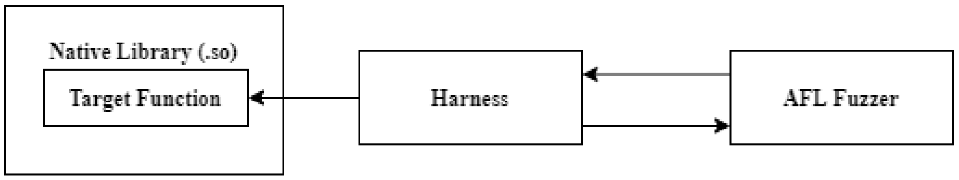
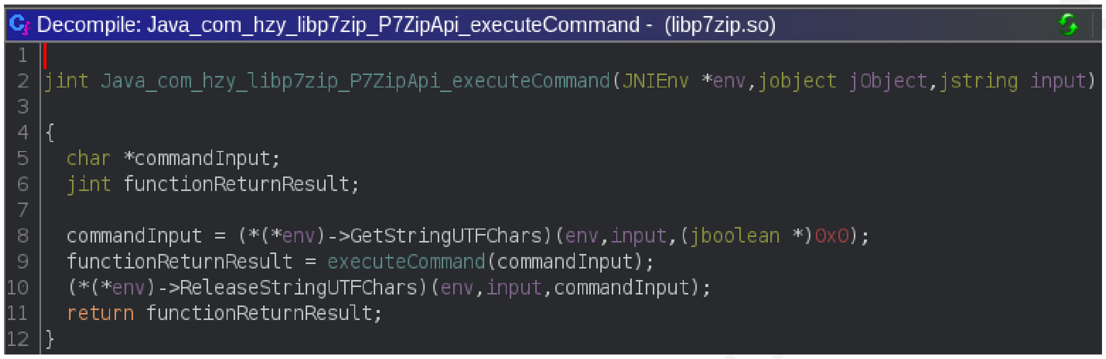


Figure 8: Relation Diagram for Harness and AFL

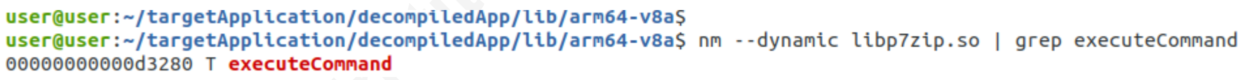
The decompiled target function from Ghidra in Figure 9 shows the last parameter in the function declaration, which is the input received from the Android Java/Kotiln side. The input is then passed to another function called “executeCommand”.


Decompile: Java_com_hzy_libp7zip_P7ZipApi_executeCommand - (libp7zip.so)

```
jint Java_com_hzy_libp7zip_P7ZipApi_executeCommand(JNIEnv *env,jobject jObject,jstring input)

{
  char *commandInput;
  jint functionReturnResult;

  commandInput = (*(*env)->GetStringUTFChars)(env,input,(jboolean *)0x0);
  functionReturnResult = executeCommand(commandInput);
  (*(*env)->ReleaseStringUTFChars)(env,input,commandInput);
  return functionReturnResult;
}
```

Figure 9: Decompiled Function from Ghidra

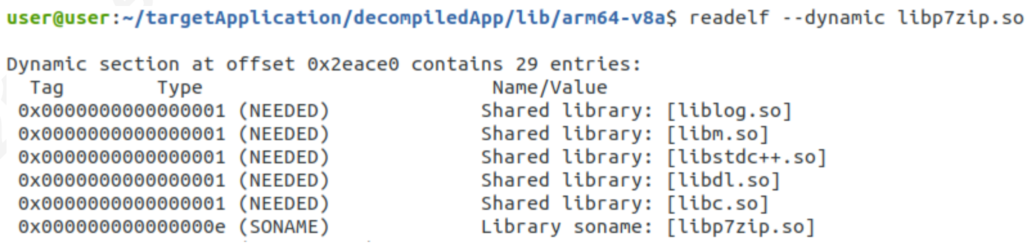
Calling the above function requires the wrapper code to provide values for the first two parameters, which are handled through Android runtime. The function "executeCommand" is an exported function that the figure below indicates using the "nm" tool.



```
user@user:~/targetApplication/decompiledApp/lib/arm64-v8a$
user@user:~/targetApplication/decompiledApp/lib/arm64-v8a$ nm --dynamic libp7zip.so | grep executeCommand
00000000000d3280 T executeCommand
```

Figure 10: Listing "executeCommand" Function from "nm"

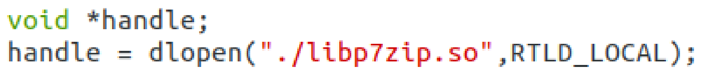
The tool "readelf" could be used to find the shared library's dependencies which need to be included when compiling the harness.



```
user@user:~/targetApplication/decompiledApp/lib/arm64-v8a$ readelf --dynamic libp7zip.so

Dynamic section at offset 0x2eace0 contains 29 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [liblog.so]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so]
 0x000000000000000e (SONAME)             Library soname: [libp7zip.so]
```

Figure 11: Listing Needed Dependencies for Target Library

In C/C++, the function dlopen() could be used to obtain a handle to a shared object library, and dlsym() is then used to obtain a function pointer to our target function "executeCommand()".



```
void *handle;
handle = dlopen("./libp7zip.so",RTLD_LOCAL);
```

Figure 12: Using dlopen()

Nawaf Alkeraithe, alkeraithe@gmail.com

From the decompiled application in Figure 9, the function takes a string argument as an input and returns an integer indicating the function execution result. The wrapper code below is an example of how to use the dlopen() function to import the Android native library and get a pointer to the target function "executeCommand" through dlsym().

```
FunctionPointerOne fPtr;
fPtr=(FunctionPointerOne)(dlsym(handle,"executeCommand"));
```

Figure 13: Using dlsym()

The type definition for the function pointer "FunctionPointerOne" in Figure 14 matches its method declaration, as shown in Figure 9.

```
typedef int (*FunctionPointerOne)(const char*);
```

Figure 14: Pointer Type Definition

Figure 15 below shows the complete wrapper code and how to pass the terminal argument to the native functions to prepare the wrapper to be in the middle between the target native function and AFL.

Nawaf Alkeraithe, alkeraithe@gmail.com

```
#include <iostream>
#include <dlfcn.h>

using namespace std;

// int executeCommand(String argument)
typedef int (*FunctionPointerOne)(const char*);

int main(int argc,char *argv[])
{
        void *handle;
        handle = dlopen("./libp7zip.so",RTLD_LOCAL);
        char *harnessInput = argv[1];
        if(!handle)
        {
                cout << dlerror();
        }
        else
        {
                cout << "library is loaded\n";
        }

        FunctionPointerOne fPtr;
        fPtr=(FunctionPointerOne)(dlsym(handle,"executeCommand"));
        if(!fPtr)
        {
                cout << dlerror();
        }
        else
        {
                cout << "Function is found\n";
        }
        printf("Harness input is: %s",harnessInput);
        int result = fPtr(harnessInput);
        printf("returned result is %d\n",result);
        dlclose(handle);
        return 0;
}
```

Figure 15: Wrapper Code

The figure below shows the compilation and testing of the application.

```
vim3:/data/local/tmp/wrapper # g++ harness.cpp -ldl -llog -lm -lc -lstdc++ -o harness
vim3:/data/local/tmp/wrapper # ./harness "7za x"
library is loaded
Function is found
Harness input is: 7za x
7-Zip (a) [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=utf8,Utf16=on,HugeFiles=on,64 bits,8 CPUs LE)



Command Line Error:
Cannot find archive name
returned result is 7
```

Figure 16: Compiling Wrapper and Testing Target Function

Nawaf Alkeraithe, alkeraithe@gmail.com

5. Fuzzing with AFL

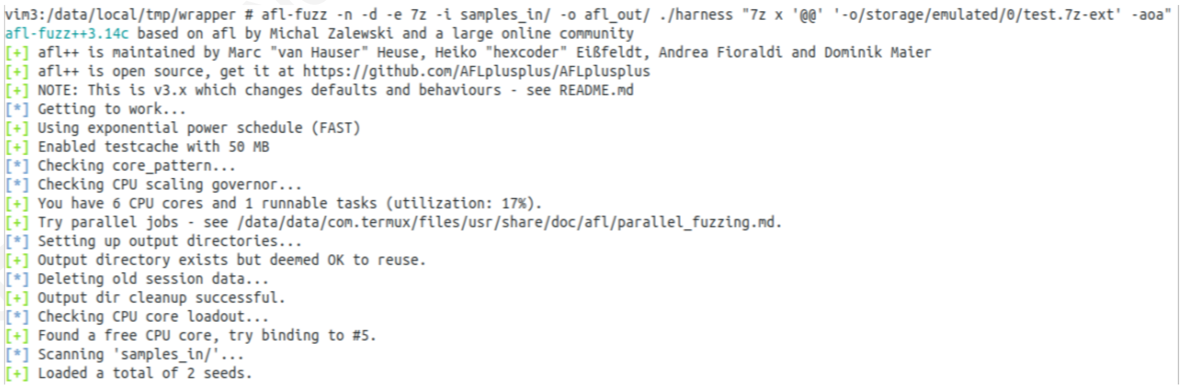
After preparing the wrapper application, then it could be fuzzed as a binary with afl-fuzz. The tool afl-fuzz requires an input directory for initial test samples and an output directory to store its findings. For this target function, it needs 7z files to extract, and therefore to prepare the environment, we will create the following folders:

- samples_in/, which contains a test.7z, and test2.7z as input samples
- afl_out/, which is needed by afl-fuzz to store findings and track testing progress

The captured input to the target function "executeCommand" is shown below.

```
"7z x '/storage/emulated/0/test.7z' '-o/storage/emulated/0/test.7z-ext' -aoa"
```

The tool afl-fuzz uses the annotation "@@" to replace its input files and test cases with the target binary. The figure below shows the command used to start afl-fuzz with the wrapper application.



```
vim3:/data/local/tmp/wrapper # afl-fuzz -n -d -e 7z -i samples_in/ -o afl_out/ ./harness "7z x '@@' '-o/storage/emulated/0/test.7z-ext' -aoa"
afl-fuzz++3.14c based on afl by Michal Zalewski and a large online community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko "hexcoder" Eißfeldt, Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours - see README.md
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
[*] Checking core_pattern...
[*] Checking CPU scaling governor...
[+] You have 6 CPU cores and 1 runnable tasks (utilization: 17%).
[+] Try parallel jobs - see /data/data/com.termux/files/usr/share/doc/afl/parallel_fuzzing.md.
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Checking CPU core loadout...
[+] Found a free CPU core, try binding to #5.
[*] Scanning 'samples_in/'...
[+] Loaded a total of 2 seeds.
```

Figure 17: Starting Wrapper with AFL

The figure below shows afl-fuzz running, and it takes a while until it finds and captures crashes to the wrapper and the target native function.

Nawaf Alkeraithe, alkeraithe@gmail.com

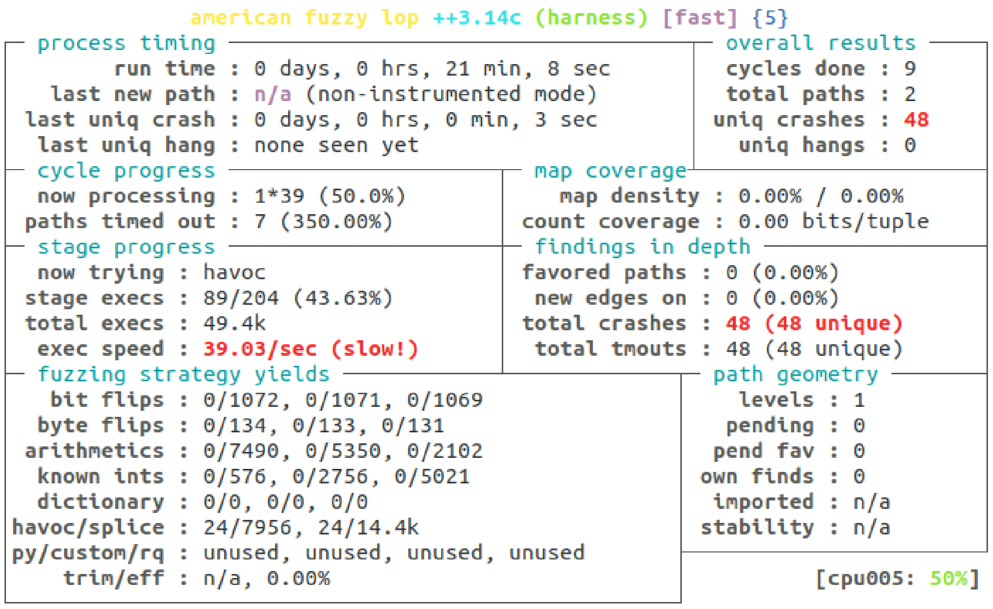

```
                 american fuzzy lop ++3.14c (harness) [fast] {5}
┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 21 min, 8 sec        │  cycles done : 9     │
│   last new path : n/a (non-instrumented mode)         │  total paths : 2     │
│ last uniq crash : 0 days, 0 hrs, 0 min, 3 sec         │ uniq crashes : 48    │
│  last uniq hang : none seen yet                       │   uniq hangs : 0     │
├─ cycle progress ──────────────────┬─ map coverage─┴──────────────────────┤
│  now processing : 1*39 (50.0%)    │    map density : 0.00% / 0.00%       │
│ paths timed out : 7 (350.00%)     │ count coverage : 0.00 bits/tuple     │
├─ stage progress ──────────────────┼─ findings in depth ──────────────────┤
│  now trying : havoc               │ favored paths : 0 (0.00%)            │
│ stage execs : 89/204 (43.63%)     │  new edges on : 0 (0.00%)            │
│ total execs : 49.4k               │ total crashes : 48 (48 unique)       │
│  exec speed : 39.03/sec (slow!)   │  total tmouts : 48 (48 unique)       │
├─ fuzzing strategy yields ─────────┴──────────┬─ path geometry ─────────┤
│   bit flips : 0/1072, 0/1071, 0/1069         │    levels : 1           │
│  byte flips : 0/134, 0/133, 0/131            │   pending : 0           │
│ arithmetics : 0/7490, 0/5350, 0/2102         │  pend fav : 0           │
│  known ints : 0/576, 0/2756, 0/5021          │ own finds : 0           │
│  dictionary : 0/0, 0/0, 0/0                  │  imported : n/a         │
│havoc/splice : 24/7956, 24/14.4k              │ stability : n/a         │
│py/custom/rq : unused, unused, unused, unused ├─────────────────────────┘
│    trim/eff : n/a, 0.00%                     │          [cpu005: 50%]
└──────────────────────────────────────────────┘
```

Figure 18: AFL Running

Once crashes are captured, afl-fuzz stores its findings in the "crashes" folder, as shown in the figure below.

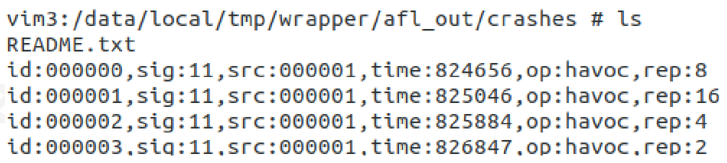

```
vim3:/data/local/tmp/wrapper/afl_out/crashes # ls
README.txt
id:000000,sig:11,src:000001,time:824656,op:havoc,rep:8
id:000001,sig:11,src:000001,time:825046,op:havoc,rep:16
id:000002,sig:11,src:000001,time:825884,op:havoc,rep:4
id:000003,sig:11,src:000001,time:826847,op:havoc,rep:2
```

Figure 19: Crashes Folder from AFL

We will rename each test case as "crash#.7z" and test each crash on the Android application to verify captured crashes.

Nawaf Alkeraithe, alkeraithe@gmail.com

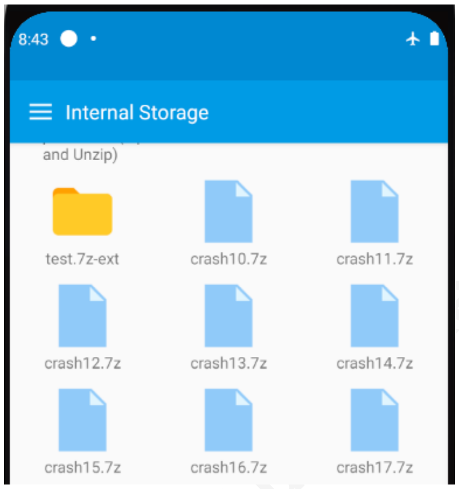
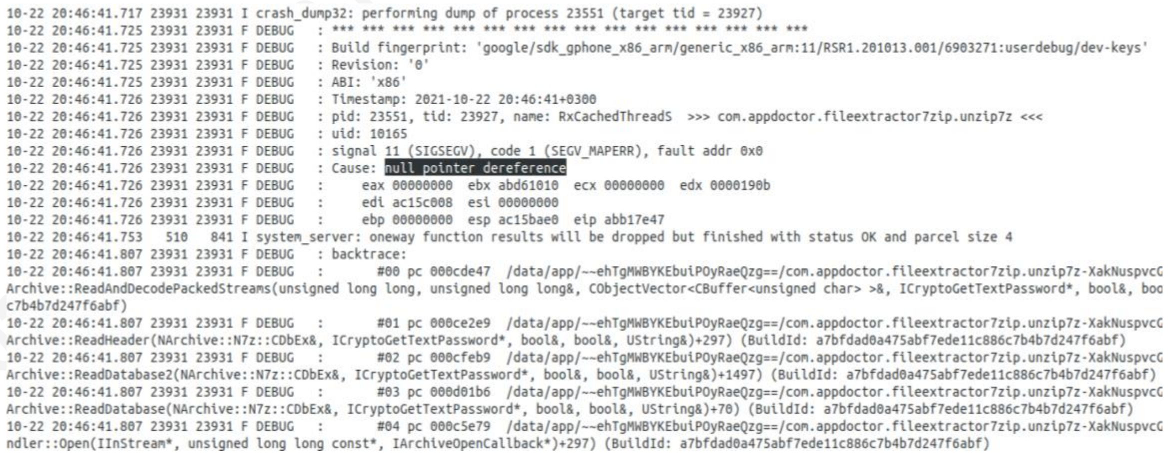


Figure 20: Files Listing from Target Application

After selecting any crash files, the application will start the extraction process through the native method "executeCommand", which is noticed through "adb logcat", as shown in the figure below.



```
10-22 20:46:41.717 23931 23931 I crash_dump32: performing dump of process 23551 (target tid = 23927)
10-22 20:46:41.725 23931 23931 F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
10-22 20:46:41.725 23931 23931 F DEBUG   : Build fingerprint: 'google/sdk_gphone_x86_arm/generic_x86_arm:11/RSR1.201013.001/6903271:userdebug/dev-keys'
10-22 20:46:41.725 23931 23931 F DEBUG   : Revision: '0'
10-22 20:46:41.725 23931 23931 F DEBUG   : ABI: 'x86'
10-22 20:46:41.726 23931 23931 F DEBUG   : Timestamp: 2021-10-22 20:46:41+0300
10-22 20:46:41.726 23931 23931 F DEBUG   : pid: 23551, tid: 23927, name: RxCachedThreadS  >>> com.appdoctor.fileextractor7zip.unzip7z <<<
10-22 20:46:41.726 23931 23931 F DEBUG   : uid: 10165
10-22 20:46:41.726 23931 23931 F DEBUG   : signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
10-22 20:46:41.726 23931 23931 F DEBUG   : Cause: null pointer dereference
10-22 20:46:41.726 23931 23931 F DEBUG   :     eax 00000000  ebx abd61010  ecx 00000000  edx 0000190b
10-22 20:46:41.726 23931 23931 F DEBUG   :     edi ac15c008  esi 00000000
10-22 20:46:41.726 23931 23931 F DEBUG   :     ebp 00000000  esp ac15bae0  eip abb17e47
10-22 20:46:41.753   510   841 I system_server: oneway function results will be dropped but finished with status OK and parcel size 4
10-22 20:46:41.807 23931 23931 F DEBUG   : backtrace:
10-22 20:46:41.807 23931 23931 F DEBUG   :       #00 pc 000cde47  /data/app/~~ehTgMWBYKEbuiPOyRaeQzg==/com.appdoctor.fileextractor7zip.unzip7z-XakNuspvcG
Archive::ReadAndDecodePackedStreams(unsigned long long, unsigned long long&, CObjectVector<CBuffer<unsigned char> >&, ICryptoGetTextPassword*, bool&, boo
c7b4b7d247f6abf)
10-22 20:46:41.807 23931 23931 F DEBUG   :       #01 pc 000ce2e9  /data/app/~~ehTgMWBYKEbuiPOyRaeQzg==/com.appdoctor.fileextractor7zip.unzip7z-XakNuspvcG
Archive::ReadHeader(NArchive::N7z::CDbEx&, ICryptoGetTextPassword*, bool&, bool&, UString&)+297) (BuildId: a7bfdad0a475abf7ede11c886c7b4b7d247f6abf)
10-22 20:46:41.807 23931 23931 F DEBUG   :       #02 pc 000cfeb9  /data/app/~~ehTgMWBYKEbuiPOyRaeQzg==/com.appdoctor.fileextractor7zip.unzip7z-XakNuspvcG
Archive::ReadDatabase2(NArchive::N7z::CDbEx&, ICryptoGetTextPassword*, bool&, bool&, UString&)+1497) (BuildId: a7bfdad0a475abf7ede11c886c7b4b7d247f6abf)
10-22 20:46:41.807 23931 23931 F DEBUG   :       #03 pc 000d01b6  /data/app/~~ehTgMWBYKEbuiPOyRaeQzg==/com.appdoctor.fileextractor7zip.unzip7z-XakNuspvcG
Archive::ReadDatabase(NArchive::N7z::CDbEx&, ICryptoGetTextPassword*, bool&, bool&, UString&)+70) (BuildId: a7bfdad0a475abf7ede11c886c7b4b7d247f6abf)
10-22 20:46:41.807 23931 23931 F DEBUG   :       #04 pc 000c5e79  /data/app/~~ehTgMWBYKEbuiPOyRaeQzg==/com.appdoctor.fileextractor7zip.unzip7z-XakNuspvcG
ndler::Open(IInStream*, unsigned long long const*, IArchiveOpenCallback*)+297) (BuildId: a7bfdad0a475abf7ede11c886c7b4b7d247f6abf)
```

Figure 21: Crash Logs from ADB Logcat

6. Conclusion

The C\C++ native code developed in Android applications might be necessary for some applications to perform data processing such as image processing and video calls more efficiently. Developers can use different open-source native libraries in their applications. However, some choose to write closed-source native code. As demonstrated in this paper, security researchers can write wrapper applications to use the target function and fuzz it with AFL to ensure thorough coverage in their Android applications security assessments. Memory management issues might lead to critical security issues compromising the security of applications users' and bringing back vulnerabilities that do not exist in Java or Kotlin applications.

References

Gal Elbaz. Checkpoint. (2020, September 24). *#Instagram_RCE: Code execution vulnerability in Instagram app for Android and iOS*. Retrieved from https://research.checkpoint.com/2020/instagram_rce-code-execution-vulnerability-in-instagram-app-for-android-and-ios/

Google. (n.d.-a). Android ABIs. Retrieved October 11, 2021, from https://developer.android.com/ndk/guides/abis

Google. (n.d.-b). Concepts. Retrieved October 11, 2021, from https://developer.android.com/ndk/guides/concepts?hl=en

Google. (n.d.-c). Get started with the NDK. Retrieved October 12, 2021, from https://developer.android.com/ndk/guides

Google. (n.d.-d). JNI tips. Retrieved from https://developer.android.com/training/articles/perf-jni#native-libraries

NIST. (2020a, October 27). *Cve-2019-11932*. Retrieved from NVD - National Vulnerability Database website: https://nvd.nist.gov/vuln/detail/CVE-2019-11932

NIST. (2020b, April 10). *Cve-2020-1895*. Retrieved from NVD - National Vulnerability Database website: https://nvd.nist.gov/vuln/detail/CVE-2020-1895

Oracle. (n.d.-a). Design overview. Retrieved October 11, 2021, from https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html

Oracle. (n.d.-b). JNI types and data structures. Retrieved October 24, 2021, from https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html

Nawaf Alkeraithe, alkeraithe@gmail.com