DROID-FF – THE ANDROID FUZZING FRAMEWORK

TWITTER : @ANTOJOSEP007

GITHUB : @ANTOJOSEPH

@WHOAMI

- security engineer @ intel
- android security enthusiast
- speaker / trainer @ Hitb Amsterdam, brucon / hackinparis / blackhat / nullcon / ground zero / c0c0n …
- when not hacking , you can see me travelling / djing / biking

DROID-FF : WHY ?

- attempts to solve fuzzing in mobile devices (* android)
- challenges in fuzzing :
  - data – generation
  - low powered devices
  - crash logging
  - crash triage
  - exploitable or not ?

DROID-FF : APPROACH

- scripts written in python
- integrates with peach / pyzuff / radamsa
- custom crash logging
- custom crash triaging
- exploitable checks via gdb plugin ☺
- Fully automated

DROID-FF : DATA GENERATION

- two approaches
  - dump fuzzing using radamsa / pyzuff
  - generation based fuzzing using peach
  - to counter checksums / magic numbers , custom fixers are added (for eg : dex repair for fixing checksums in randomly mutated dex files (credits : github.com/anestisb)
  - Grammar specified in pit files for peach

DROID-FF : FUZZING CAMPAIGN

- Runs the generated files against the target binary (for eg : /system/xbin/dexdump crash1.dex)
- Makes use of adb_android python module to push generated files to device
- Makes use of adb shell command to execute the file against the target binary
- Adds a custom log to the android logcat so that we can track any files responsible for the crash (for eg : log -p F -t CRASH_LOGGER SIGSEGV : filename.dex)

DROID-FF : BUILDING ANDROID MODULES

- Navigate to the module directory (eg : /frameworks/av/cmd/stagefright/)
- Use the make helper
  - source build/envsetup.sh
  - edit (/frameworks/av/cmd/stagefright/Android.mk) and LOCAL_MODULE =$BUILD_EXECUTABLE
  - mma (/out/target/product/generic/system/xbin/dexdump)

DROID-FF : PROCESSING THE LOGS

- Pulls the adb logcat data from the device by saving it to a file and adb pull
- Parse the log file and look for crashes ("SIGSEGV", "SIGSEGV", "SIGFPE","SIGILL")
- If a crash is found , go up the lines until you find our custom crash file name logger
- Queue the file responsible for the crash to double check

```
I/BootReceiver(  357): Copying /data/tombstones/tombstone_02 to DropBox (SYSTEM_TOMBSTONE)
F/CRASH_LOGGER( 1085): SIGSEGV : .DS_Store
F/CRASH_LOGGER( 1092): SIGSEGV : sample0-0-137960.dex
E/dalvikvm( 1094): ERROR: Bad magic number (0xe0 ad 75 28)
F/CRASH_LOGGER( 1099): SIGSEGV : sample0-137960-275921.dex
W/dalvikvm( 1101): WARNING: Odd length: expected 0, got 551843
F/libc    ( 1101): Fatal signal 11 (SIGSEGV) at 0xb6f37000 (code=2), thread 1101 (dexdump)
I/DEBUG   (   55): *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
I/DEBUG   (   55): Build fingerprint: 'Android/aosp_arm/generic:4.4.4/KTU84P/eng.anto.20160511.162155:eng/test-keys'
I/DEBUG   (   55): Revision: '0'
I/DEBUG   (   55): pid: 1101, tid: 1101, name: dexdump  >>> /system/xbin/dexdump <<<
I/DEBUG   (   55): signal 11 (SIGSEGV), code 2 (SEGV_ACCERR), fault addr b6f37000
W/NativeCrashListener(  357): Couldn't find ProcessRecord for pid 1101
I/DEBUG   (   55):     r0 00021821  r1 00000000  r2 ffec5874  r3 b6f36ffc
I/DEBUG   (   55): AM write failure (32 / Broken pipe)
I/DEBUG   (   55):     r4 00021821  r5 00021821  r6 13e7ab7e  r7 b6f3778c
I/DEBUG   (   55):     r8 80078071  r9 00000000  sl 000015af  fp 00000003
I/DEBUG   (   55):     ip 00000000  sp beaee96c  lr b6f573db  pc b6f22cbc  cpsr 80000010
I/DEBUG   (   55):     d0  05f803ce0857f87d  d1  0000000000000000
I/DEBUG   (   55):     d2  0000000000000000  d3  0000000000000000
I/DEBUG   (   55):     d4  0000000000000000  d5  41a0aff0fa000000
I/DEBUG   (   55):     d6  3f50624dd2f1a9fc  d7  4197e00f3be45a1d
I/DEBUG   (   55):     d8  0000000000000000  d9  0000000000000000
I/DEBUG   (   55):     d10 0000000000000000  d11 0000000000000000
I/DEBUG   (   55):     d12 0000000000000000  d13 0000000000000000
I/DEBUG   (   55):     d14 0000000000000000  d15 0000000000000000
I/DEBUG   (   55):     scr 00000010
I/DEBUG   (   55):
I/DEBUG   (   55): backtrace:
I/DEBUG   (   55):     #00  pc 00001cbc  /system/lib/libz.so (adler32+192)
I/DEBUG   (   55):     #01  pc 000073d7  /system/xbin/dexdump
I/DEBUG   (   55):     #02  pc 00003bf3  /system/xbin/dexdump
I/DEBUG   (   55):     #03  pc 0000367b  /system/xbin/dexdump
I/DEBUG   (   55):     #04  pc 0000393b  /system/xbin/dexdump
I/DEBUG   (   55):     #05  pc 0000e23b  /system/lib/libc.so (__libc_init+50)
I/DEBUG   (   55):     #06  pc 00001774  /system/xbin/dexdump
I/DEBUG   (   55):
I/DEBUG   (   55): stack:
I/DEBUG   (   55):         beaee92c  b6eaaee2  /system/lib/libm.so
I/DEBUG   (   55):         beaee930  b6f4e678  /system/bin/linker
```

DROID-FF : CRASH VERIFICATION

- Runs the files responsible for crash against the target binary
- In the event of a crash , android system writes tombstone files (crashdump) to the /data/tombstones directory .
- Backup the tombstone file along with the file responsible for crash
- Look for duplicate crashes by filtering the pc register value in the tombstone file and only save unique crashes

DROID-FF : PROCESS TOMBSTONES

- Unique crashes needs to be mapped to relevant source code
- Using ndk-stack and addr2line utilities (android –ndk tools) , we map the crash to a line in the android source code
- Ndk-stack :
  - /path/to/file_with_symbols
  - /path/to/tombstone_file

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'Android/aosp_arm/generic:4.4.4/KTU84P/eng.anto.20160511.162155:eng/test-keys'
Revision: '0'
pid: 1388, tid: 1388, name: stagefrightsymb  >>> ./stagefrightsymbol <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
    r0 00000000  r1 0000056c  r2 00000006  r3 00000000
    r4 00000006  r5 00000002  r6 0000056c  r7 0000010c
    r8 00000000  r9 00000001  sl ffffffff  fp 00000000
    ip b6dc3fd8  sp beeb1378  lr b6d8dead  pc b6d9ce20  cpsr 00000010
    d0  abc0a7eb00000000  d1  0000000000000000
    d2  0000000000000000  d3  0000000000000000
    d4  0000000000000000  d5  41bda19a25000000
    d6  3f50624dd2f1a9fc  d7  3eccccd3eccccccd
    d8  0000000000000000  d9  0000000000000000
    d10 0000000000000000  d11 0000000000000000
    d12 0000000000000000  d13 0000000000000000
    d14 0000000000000000  d15 0000000000000000
    scr 20000010

backtrace:
    #00  pc 00021e20  /system/lib/libc.so (tgkill+12)
    #01  pc 00012ea9  /system/lib/libc.so (pthread_kill+48)
    #02  pc 000130bd  /system/lib/libc.so (raise+10)
    #03  pc 00011df3  /system/lib/libc.so
    #04  pc 000216d4  /system/lib/libc.so (abort+4)
    #05  pc 00006867  /system/lib/libcutils.so (__android_log_assert+86)
    #06  pc 0000528d  /data/local/tmp/stagefrightsymbol (main+1036)
    #07  pc 0000e23b  /system/lib/libc.so (__libc_init+50)
    #08  pc 00004398  /data/local/tmp/stagefrightsymbol (_start+96)

stack:
        beeb1338  00000000
        beeb133c  00000000
        beeb1340  00000000
        beeb1344  000003ff
        beeb1348  b6ef3f20  /system/lib/libstagefright.so
        beeb134c  b6e6ad7d  /system/lib/libstagefright.so (android::MPEG4Extractor::getMetaData())
        beeb1350  b6dc3fd8  /system/lib/libc.so
        beeb1354  00000000
        beeb1358  00000001
        beeb135c  ffffffff
```

DROID-FF : PROCESS TOMBSTONE (2)

- Addr2line
  - Address of the crash obtained by running ndk-stack in the target module
  - "-C","-f", "-e", symbols_file_of_crash, address_of_the_crash
  - Output gives the function and filename responsible for the crash

DROIF-FF : EXPLOITABLE ?

- Uses a gdb plugin ”exploitable “ which supports arm
- Looks at the state of the process when in crashes / unwinds stack etc and predicts based on custom rules

- Runs using python gdb api ((gdb) source ../path/to/exploitable.py)
- Gdbserver for arm is pushed to the device

DROID-FF : EXPLOITABLE ? (2)

- root@goldfish: ./gdbserver :5039 /system/xbin/dexdump crash1.dex
- (gdb) set solib-absolute-prefixdb /path/to/symbols/
- (gdb) set solib-search-path /path/to/symbols/system/lib/
- (gdb) target remote : 5039
- (gdb) c
- Wait for process to crash or send it a kill sig (kill -9 pid)
- (gdb) exploitable
- (gdb) Stack Corruption , Exploitable : True , Description : blah blah blah

```
xbin/dexdump crash1.dex
Process /system/xbin/dexdump created; p
Listening on port 5039
Remote debugging from host 127.0.0.1
readchar: Got EOF
Remote side has terminated connection.
Listening on port 5039
Remote debugging from host 127.0.0.1
Processing 'crash1.dex'...
^C
Child terminated with signal = 0xb (SIG
GDBserver exiting
root@generic:/data/local/tmp # ./gdbser
Process dexdump created; pid = 1542
Listening on port 5039
Remote debugging from host 127.0.0.1
Processing 'crash1.dex'...
Killing all inferiors
root@generic:/data/local/tmp # ./gdbser
Process dexdump created; pid = 1549
Listening on port 5039
Remote debugging from host 127.0.0.1
Processing 'crash1.dex'...
```

```
(gdb) source ../../../exploitable/exploitable
exploitable/          exploitable.egg-info/
(gdb) source ../../../exploitable/exploitable
exploitable/          exploitable.egg-info/
(gdb) source ../../../exploitable/exploitable/
__init__.py     exploitable.py  lib/            test
(gdb) source ../../../exploitable/exploitable/exploi
(gdb) target remote :5039
Remote debugging using :5039
0xb6f6da40 in ?? ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xb6f51cbc in ?? ()
(gdb) exploitable
Description: Possible stack corruption
Short description: PossibleStackCorruption (7/22)
Hash: 985e1c0964c97d64b0cff1ef2db5160f.985e1c0964c97
Exploitability Classification: EXPLOITABLE
Explanation: GDB generated an error while unwinding
at were not mapped in the inferior's process address
n outside the default stack region. These conditions
nsidered exploitable.
Other tags: SourceAv (19/22), AccessViolation (21/22
(gdb)
```

DROID-FF : ACHIEVEMENTS

- A lot of crashes , A LOOOOOOOTTTT !
- Fuzzing made easier and available for the masses
- Mostly automated
- Easily customizable
- Python ☺
- Source : github.com/antojoseph/droid-ff

DROID-FF : FUTURE IMPROVEMENTS

- Integration with ASAN
- Add support for automated gdb exploitability test and reporting
- Instrumented fuzzing ?
- Automate posting of exploitable crashes to android security group ? ☺

AFL FOR ANDROID

- Intel open sourced their implementation of afl on android
- Responsible for a lot of stagefright crashes
- Instrumented fuzzing helps in better coverage of all code paths

HONGFUZZ

- Runs within android
- Ported to android by (github.com/anestisb)
- Easy to get up and running with and very useful for quickly fuzzing binaries
- Built-in native crash logging mechanism (over-rides android debuggered)

THANKS

- @Ananth Srivastava – for all the packaging and suggestions
- @Sumanth Naropanth – for being a cool manager
- @jduck – for inspiration to write fuzzers and all the help from droidsec irc (those series of stagefrights ☺)
- @anestisb – for those tools and articles on android fuzzing
- @Alexandru Blanda – for his work in MFF and being a good friend
- @Stephen Kyle - for his articles on fuzzin in ARM
- @flanker_hqd– BH Presentation on Fuzzing Parcels

HOW TO : DROID-FF

HOW TO : DROID-FF : STEP 1

- Python droid-ff.py
  - Select the data –generator to use
  - Options (bitflipper / radamsa / peach)

- On Error :
  - (make sure you have the python requirements installed)
  - pyZUFF
  - adb_android

HOW TO : DROID-FF : STEP 2

- To Run the fuzzing campaign :
  - Have android emulator up and running
  - Android avd
  - Start the emulator
  - Test by checking “adb devices ” in the console
  - Once running , run droif-ff.py and select option 2

HOW TO : DROID-FF : STEP 3

- Find crashes in your fuzzing campaign
  - Make sure your emulator is still running and you can connect via adb
  - Select step 3 in droid-ff.py
  - This will pull the adb logs and search for any crashes and identify the files responsible
  - On Error :
    - Make sure you have all the required directories in the fuzzer folder , else moving files will fail
    - Manually verify the logcat output and check if the script missed any crashes (very unlikely)

HOW TO : DROID-FF : STEP4

- To avoid false positives :
  - All files which are identified to case a crash in the previous step is run again
  - If crash happens , a resulting tombstone file is created
  - We pull the tombstone file and identify the pc address of the crash
  - We keep a dict of key value pairs of {pc , filename } and unique crashes are identified and moved to a separate folder

HOW TO DROID-FF : STEP 5

- The crashes are resolved to a filename and method :
  - Using ndk-stack tool , binary with symbols and the tombstone file , we can print the stack frame
  - Using addr2line , address from ndk-stack and binary with symbols , we resolve the crash to a line and method in the sourcode

HOW TO : DROID-FF : STEP 6

- Check exploitability :
  - Uses a gbd plugin which supports linux arm
  - Loaded via .gdbinit
  - Set symbol search path in gdb
  - adb forward tcp:5039 tcp:5039
  - gdbserver runs on the android device and listens on tcp port 5039
  - gdb connects to gdb server and continues the execution of the binary until fault
  - On fault signal , run exploitable and prints the result

WHAT NEXT ?

- Do a second round of manual analysis to make sure the bug is exploitable
- Reproduce the bug in different devices / architectures
- Report and exhaustive security bug report to the android security team
- If you are lucky , get your android – security bounty $$$

THANKS ☺

- Questions please …