Fuzzing closed source applications
Introduction

- René Freingruber ([r.freingruber@sec-consult.com](mailto:r.freingruber@sec-consult.com))
  - Twitter: @ReneFreingruber
  - Security Consultant at SEC Consult
    - Reverse Engineering, Exploit development, Fuzzing
  - Trainer at SEC Consult
    - Secure Coding in C/C++, Reverse Engineering
    - Red Teaming, Windows Security
  - Speaker at conferences:
    - CanSecWest, DeepSec, 31C3, Hacktivity, BSides Vienna, Ruxcon, ToorCon,
      NorthSec, IT-SeCX, QuBit, DSS ITSEC, ZeroNights, Owasp Chapter, …
    - Topics: EMET, Application Whitelisting, Hacking Kerio Firewalls, Fuzzing Mimikatz, …
Title: Fuzzing closed source applications | Responsible: R. Freingruber | Version / Date: V1.0/2017-10 | Confidentiality Class: public
© 2017 SEC Consult | All rights reserved

Founded 2002
Leading in IT-Security Services and Consulting
Strong customer base in Europe and Asia
70+ Security experts
400+ Security audits per year
Feedback-based Fuzzing
Consider this pseudocode

```c
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```
Feedback based fuzzing

Input „command\n“ results in the orange code-coverage output

```java
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```
Same for "command\nsubcommand\n"

```c
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```
Feedback based fuzzing

And so on...

```c
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```
1. Instrumentation during compilation (source code available; gcc or llvm ➔ AFL)
American Fuzzy Lop - AFL

• **One of the most famous file-format fuzzers**
  - Developed by Michal Zalewski

• Instruments application during compile time (GCC or LLVM)
  - Binary-only targets can be emulated / instrumented with qemu
  - Forks exist for PIN, DynamoRio, DynInst, syzygy, IntelPT, …
  - Simple to use!
  - Good designed! (very fast & good heuristics)

• **Strategy:**
  1. Start with a small min-set of input sample files
  2. Mutate “random” input file from queue like a dumb fuzzer
  3. If mutated file reaches new path(s), add it to queue
Feedback based fuzzing

- Just use afl-gcc instead of gcc…

```
user-VirtualBox# afl-gcc -o test2 test.c
afl-cc 2.35b by <lcamtuf@google.com>
afl-as 2.35b by <lcamtuf@google.com>
[+] Instrumented 6 locations (64-bit, non-hardened mode, ratio 100%).
user-VirtualBox# ./test2 1
Test2
Test3
user-VirtualBox# ./test2 1 2 3 4 5
Test1
Test3
```
Feedback based fuzzing

- Basic Blocks:
Feedback based fuzzing

- **Result:**
  - **Instrumentation**
  - **Store old register values**
  - **Restore old register values**
• Instrumentation tracks **edge coverage**, injected code at every basic block:

```c
cur_location = <compile_time_random_value>;
bitmap[(cur_location ^ prev_location) % BITMAP_SIZE]++;
prev_location = cur_location >> 1;
```

➔ **AFL can distinguish between**
  • A->B->C->D->E (tuples: AB, BC, CD, DE)
  • A->B->D->C->E (tuples: AB, BD, DC, CE)
American Fuzzy Lop - AFL

- Instrumentation tracks **edge coverage**, injected code at every basic block:

  cur_location = <compile_time_random_value>;
  bitmap[(cur_location ^ prev_location) % BITMAP_SIZE]++;
  prev_location = cur_location >> 1;

  ➔ AFL can distinguish between
  - A->B->C->D->E (tuples: AB, BC, CD, DE)
  - A->B->D->C->E (tuples: AB, BD, DC, CE)

  ➔ Without shifting A->B and B->A are indistinguishable
Without instrumentation just the first level will be discovered (or it would take an extremely long time).
# American Fuzzy Lop - AFL

## Process Timing

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time</td>
<td>42 days, 19 hrs, 27 min, 41 sec</td>
</tr>
<tr>
<td>Last new path</td>
<td>0 days, 1 hrs, 45 min, 10 sec</td>
</tr>
<tr>
<td>Last uniq crash</td>
<td>5 days, 19 hrs, 58 min, 31 sec</td>
</tr>
<tr>
<td>Last uniq hang</td>
<td>1 days, 16 hrs, 58 min, 37 sec</td>
</tr>
</tbody>
</table>

## Cycle Progress

<table>
<thead>
<tr>
<th>Stage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing</td>
<td>1550* (10.74%)</td>
</tr>
<tr>
<td>Paths timed out</td>
<td>0 (0.00%)</td>
</tr>
</tbody>
</table>

## Stage Progress

- **Now trying:** bitflip 1/1
- **Stage execs:** 880/106k (0.83%)
- **Total execs:** 4.54G
- **Exec speed:** 2338/sec

## Fuzzing Strategy Yields

- **Bit flips:** 5858/474M, 1418/474M, 557/474M
- **Byte flips:** 86/59.4M, 57/13.2M, 57/13.6M
- **Arithmetics:** 2564/725M, 79/548M, 182/375M
- **Known ints:** 162/47.6M, 359/226M, 374/425M
- **Dictionary:** 0/0, 0/0, 1061/659M
- **Havoc:** 1631/9.85M, 0/0
- **Trim:** 2.82%/4.13M, 78.13%

## Overall Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles done</td>
<td>3</td>
</tr>
<tr>
<td>Total paths</td>
<td>14.4k</td>
</tr>
<tr>
<td>Unique crashes</td>
<td>25</td>
</tr>
<tr>
<td>Unique hangs</td>
<td>161</td>
</tr>
<tr>
<td>Map coverage</td>
<td>0.39% / 18.87%</td>
</tr>
<tr>
<td>Count coverage</td>
<td>4.30 bits/tuple</td>
</tr>
<tr>
<td>Favored paths</td>
<td>2220 (15.39%)</td>
</tr>
<tr>
<td>New edges on</td>
<td>3431 (23.78%)</td>
</tr>
<tr>
<td>Total crashes</td>
<td>1286 (25 unique)</td>
</tr>
<tr>
<td>Total timeouts</td>
<td>25.5k (224 unique)</td>
</tr>
<tr>
<td>Path geometry</td>
<td></td>
</tr>
<tr>
<td>Levels</td>
<td>27</td>
</tr>
<tr>
<td>Pending</td>
<td>10.5k</td>
</tr>
<tr>
<td>Pending fav</td>
<td>1</td>
</tr>
<tr>
<td>Own finds</td>
<td>14.4k</td>
</tr>
<tr>
<td>Imported</td>
<td>n/a</td>
</tr>
<tr>
<td>Stability</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

---

*Title: Fuzzing closed source applications | Responsible: R. Freingruber | Version / Date: V1.0/2017-10 | Confidentiality Class: public*

© 2017 SEC Consult | All rights reserved
We can either start fuzzing with an empty input folder or with downloaded / generated input files.

**Empty file:**
- Let AFL identify the complete format (unknown target binaries)
- Can be very slow

**Downloaded sample files:**
- Much faster because AFL doesn’t have to find the file format structure itself
- Bing API to crawl the web (Hint: Don’t use DNS of your provider …)
- Other good sources: Unit-tests, bug report pages, …
- Problem: Many sample files execute the same code ➔ Corpus Distillation
Steps for fuzzing with AFL:

1. Remove input files with same functionality:
   Hint: Call it after tmin again (cmin is a heuristic)
   
   ```
   ./afl-cmin -i testcase_dir -o testcase_out_dir
   /path/to/tested/program [...program's cmdline...]
   ```

2. Reduce file size of input files:
   
   ```
   ./afl-tmin -i testcase_file -o testcase_out_file
   /path/to/tested/program [...program's cmdline...]
   ```

3. Start fuzzing:
   
   ```
   ./afl-fuzz -i testcase_dir -o findings_dir /path/to/tested/program
   [...program's cmdline...] @@
AFL with CVE-2009-0385 (FFMPEG)

- AFL input with invalid 4xm file (strk chunk changed to strj)
  
  ![Hexdump 1](image1)

- AFL still finds the vulnerability!
  - Level 1 identifies correct “strk” chunk
  - Level 2 based on level 1 output AFL finds the vulnerability (triggered by 0xffffffff)

![Hexdump 2](image2)
LibFuzzer

- LibFuzzer – Similar concept to AFL but in-memory fuzzing
  - Requires LLVM SanitizerCoverage + writing small fuzzer-functions
  - LibFuzzer is more the Fuzzer for developers
  - AFL fuzzes the execution path of a binary (no modification required)
  - LibFuzzer fuzzes the execution path of a specific function (minimal code modifications required)
    - Fuzz function1 which processes data format 1 ➔ Corpus 1
    - Fuzz function2 which processes data format 2 ➔ Corpus 2
    - AFL can be also do in-memory fuzzing (persistent mode)

- Highly recommended tutorial: [http://tutorial.libfuzzer.info](http://tutorial.libfuzzer.info)
Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm \(\rightarrow\) AFL)
2. Emulation of binary (e.g. with qemu)
Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm → AFL)
2. Emulation of binary (e.g. with qemu)
3. Writing own debugger and set breakpoints on every basicblock (slow, but useful in some situations)
Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm → AFL)
2. Emulation of binary (e.g. with qemu)
3. Writing own debugger and set breakpoints on every basicblock (slow, but useful in some situations)
4. Dynamic instrumentation of compiled application (no source code required; tools: DynamoRio, PIN, Valgrind, Frida, …)
Dynamic Instrumentation Frameworks

- **Dynamic runtime manipulation** of instructions of a running application!

- Many default tools are shipped with these frameworks
  - `drrun.exe -t drcov -- calc.exe`
  - `drrun.exe -t my_tool.dll -- calc.exe`
  - `pin -t inscount.so -- /bin/ls`

- Register callbacks, which are triggered at specific events
  - new basic block / instruction
  - load of module, exit of process, ...

- At callback (e.g. new basic block), we can further add instructions
  - Transformation time (Instrumentation Function): Analyzing a BB the first time (called once)
  - Execution time (Analysis Function): Executed always before instruction gets executed
DynamoRIO

Source: The DynamoRIO Dynamic Tool Platform, Derek Bruening, Google

Transformation time

Execution time
• **Example:** Start Adobe Reader, load PDF file, exit Adobe Reader, extract coverage data (Processing 25 PDFs with one single CPU core)

• Runtime without DynamoRio: ~30-40 seconds

• BasicBlock coverage (no hit count): 105 seconds
  • Instrumentation only during transformation into code cache (transformation time)

• BasicBlock coverage (hit count): 165 seconds
  • Instrumentation on basic block level (execution time)

• Edge coverage (hit count): 246 seconds
  • Instrumentation on basic block level (many instructions required to save and restore required registers for instrumentation code) (execution time)
DynamoRio vs PIN

- **PIN** is another dynamic instrumentation framework (older)

- Currently more people use PIN (➔ more examples are available)

- DynamoRio is noticeable faster than PIN

- But PIN is more reliable
  - DynamoRio can’t start Encase Imager, PIN can
  - DynamoRio can’t start CS GO, PIN can
  - During client writing I noticed several strange behaviors of DynamoRio
WinAFL

- **WinAFL - AFL for Windows**
  - Download: [https://github.com/ivanfratric/winafl](https://github.com/ivanfratric/winafl)
  - Developed by Ivan Fratric

- Two modes:
  - DynamoRio: Source code not required
  - Syzygy: Source code required
  - Alternative: You can easily modify WinAFL to use PIN on Windows

- Windows does not use COW (Copy-on-Write) and therefore fork-like mechanisms are not efficient on Windows!
  - On Linux AFL heavily uses a fork-server
  - On Windows WinAFL heavily uses in-memory fuzzing
How to select a target function

The target function should do these things during its lifetime:

1. **Open the input file.** This needs to happen within the target function so that you can read a new input file for each iteration as the input file is rewritten between target function runs).
2. **Parse it** (so that you can measure coverage of file parsing)
3. **Close the input file.** This is important because if the input file is not closed WinAFL won't be able to rewrite it.
4. **Return normally** (So that WinAFL can "catch" this return and redirect execution. "returning" via ExitProcess() and such won't work)

Source: https://github.com/ivanfratric/winafl FAQ
Q: Can I fuzz GUI apps with WinAFL
A: Yes, provided that
- There is a target function that behaves as explained in "How to select a target function"
- The target function is reachable without user interaction
- The target function runs and returns without user interaction
If these conditions are not satisfied, you might need to make custom changes to WinAFL and/or your target.

Source: https://github.com/ivanfratric/winafl FAQ

Autolt can easily solve this problem
DynamoRio / PIN to change instruction ptr
```
#include <AutoItConstants.au3>

Run("notepad.exe")
Local $hWand = WinWait("[CLASS:Notepad]", "", 10)
ControlSend($hWand, "", "Edit1", "Hello World")
WinClose($hWand)
ControlClick("[CLASS:#32770]", "", "Button3")
WinSetState("[CLASS:Notepad]", "", @SW_MAXIMIZE)
MouseMove(14, 31)
MouseClick($MOUSE_CLICK_LEFT)
MouseMove(85, 209)
MouseClick($MOUSE_CLICK_LEFT)
ControlClick("[CLASS:#32770]", "", "Button2")
```
Another use case: Popup Killer

- During fuzzing applications often spawn error message; popup killer closes them
- Another implementation can be found in CERT Basic Fuzzing Framework (BFF) Windows Setup files (C++ code to monitor for message box events)

```autoIt
#include <MsgBoxConstants.au3>
While 1
    Local $aList = WinList()
    ; $aList[0][0] number elements
    ; $aList[x][0] => title ; $aList[x][1] => handle
    For $i = 1 To $aList[0][0]
        If StringCompare($aList[$i][0], "Engine Error") == 0 Then
            ControlClick($aList[$i][1], "", "Button2", "left", 2)
        EndIf
    Next
    sleep(500) ; 500 ms
WEnd
```
1. Instrumentation during compilation (source code available; gcc or llvm → AFL)
2. Emulation of binary (e.g. with qemu)
3. Writing own debugger and set breakpoints on every basicblock (slow, but useful in some situations)
4. Dynamic instrumentation of compiled application (no source code required; tools: DynamoRio, PIN, Valgrind, Frida, …)
5. Static instrumentation via static binary rewriting (Talos fork of AFL which uses DynInst framework – AFL-dyninst, should be fastest possibility if source code is not available but it’s not 100% reliable and currently Linux only); WinAFL in syzygy mode is very useful on Windows if source-code is available!
Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm → AFL)
2. Emulation of binary (e.g. with qemu)
3. Writing own debugger and set breakpoints on every basicblock (slow, but useful in some situations)
4. Dynamic instrumentation of compiled application (no source code required; tools: DynamoRio, PIN, Valgrind, Frida, …)
5. Static instrumentation via static binary rewriting (Talos fork of AFL which uses DynInst framework – AFL-dyninst, should be fastest possibility if source code is not available but it’s not 100% reliable and currently Linux only); WinAFL in syzygy mode is very useful on Windows if source-code is available!
6. Use of hardware features
   • IntelPT (Processor Tracing); available since 6th Intel-Core generation (~2015)
   • WindowsIntelPT (from Talos) or kAFL
Areas which influent fuzzzer results
Areas which influence fuzzing results
Overview: Areas which influence fuzzing results

Fork-server
Faster instrumentation code
Static vs. Dynamic Instrumentation
In-memory fuzzing
No process switches

Page heap / Heap libs
Sanitizers (ASAN, MSAN, SyzyASan, DrMemory, ..)
Dangling Pointer Check
Writeable Format Strings Check

AFL-tmin & AFL-cmin
Heat maps via Taint Analysis and Shadow Memory

Fuzzer speed
Input filesize
Detection rate
Mutators

Application aware mutators
Generated dictionaries
Append vs. Modify mode
Grammar-based mutators
Use of feedback from application

Title: Fuzzing closed source applications | Responsible: R. Freingruber | Version / Date: V1.0/2017-10 | Confidentiality Class: public
© 2017 SEC Consult | All rights reserved
Areas which influence fuzzing results

Fuzzer speed
Fuzzer Speed

1. Fork Server
2. Deferred Fork Server
3. Persistent Mode (in-memory fuzzing)
4. Prevent process switches (between target application and the Fuzzer) by injecting the Fuzzer code into the target process
5. Modify the input in-memory instead of on-disk
6. Use a RAM Disk
7. Remove slow API calls
Question 1: What is the maximum MD5 fuzzing speed with GUI automation?

Question 2: How many MD5 hashes can you calculate on a CPU per second?
• HashCalc.exe MD5 fuzzing

• GUI automation with Autolt: ~3 exec / sec

• In-Memory with debugger: ~750 exec / sec

• In-Memory with DynamoRio (no instr.): ~200 000 exec / sec
How to find the target function without source code?

1. Measure code coverage (drrun –t drcov) in two program invocations, one should trigger the function, one not. Then subtract both traces (IDA Pro lighthouse)

2. Log all calls and returns together with register and stack values to a logfile. Then search for the correct input / output combination (IDA Pro funcap or a simple DynamoRio / PIN tool)

3. Place memory breakpoints on the input

4. Use a taint engine (see later)
Areas which influence fuzzing results
Input file size

- The input file size is extremely important!

- Smaller files
  - Have a higher likelihood to change the correct bit / byte during fuzzing
  - Are faster processed by deterministic fuzzing
  - Are faster loaded by the target application

- AFL ships with two utilities
  - AFL-cmin: Reduce number of files with same functionality
  - AFL-tmin: Reduce file size of an input file
    - Uses a “fuzzer” approach and heuristics
    - Runtime depends on file size
    - Problems with file offsets
Input file size

- **Example: Fuzzing mimikatz**
  - Initial memory dump: 27 004 528 Byte
  - Memory dump which I fuzzed: 2 234 Byte

→ **I’m approximately 12 000 times faster with this setup…**
  - You would need 12 000 CPU cores to get the same result in the same time as my fuzzing setup with one CPU core
  - Or with the same number of CPU cores you need 12 000 days (~33 years) to get the same result as I within one day
  - In reality it’s even worse, since you have to do everything again for every queue entry (exponential)
Heat map of the memory dump (mimikatz access)
Heat map of the memory dump (mimikatz access) - Zoomed
See below link for in-depth discussion how I fuzzed mimikatz with WinAFL:
Creation of heatmaps

- For mimikatz I used a WinAppDbg script to extract file access information
  - Very slow approach because of the Debugger
  - Can’t follow all memory copies ➔ Hitcounts are not 100% correct

- Better approach: Use dynamic instrumentation / emulation
  - libdft
  - Triton
  - Panda
  - Manticore
  - Own PIN / DynamoRio tool
We can write a DynamoRio/PIN tool which tracks calls and taint status
Automatically detect target fuzz function
Fuzzing with taint analysis

1. Typically byte-modifications are uniform distributed over the input file
2. With taint analysis we can distribute it uniform over the tainted instructions!

If this sets EFLAGS and can change a cc jump, it should give an extra boost…

Maybe don’t fuzz this at all X2 is maybe a copy / search function

<table>
<thead>
<tr>
<th>Byte</th>
<th>Mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

- Instruction X1: Read byte 2
- Instruction X2: Read byte 1,2,3,4
- Instruction X3: Read byte 2
- Instruction X4: Read byte 1,2
- Instruction X5: Read byte 2,3

<table>
<thead>
<tr>
<th>Byte</th>
<th>Read by</th>
<th>Mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>20%</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20%</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10%</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

If this sets EFLAGS and can change a cc jump, it should give an extra boost…

Maybe don’t fuzz this at all X2 is maybe a copy / search function

<table>
<thead>
<tr>
<th>Byte</th>
<th>Read by</th>
<th>Mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>20%</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20%</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10%</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
The power of dynamic instrumentation frameworks

- Automatically detect target fuzz function
- Taint engine can be used on first fuzz iteration ➔ All writes can be logged with the address to revert the memory state for new fuzz iterations
- Enable taint engine logging only for new code coverage ➔ Automatically detect which bytes make the new input unique and focus on fuzzing them!
- Call-instruction logging can be used to find interesting functions
  - Malloc / Free functions (to automatically change to own heap implementation)
    - Own heap allocator can free all chunks allocated in a fuzz iteration ➔ No mem leaks
    - Better vulnerability detection (see later slides)
  - Compare functions ➔ Return the comparison value to the fuzzer
  - Checksum functions ➔ Automatically “remove” checksum code
  - Error-handling functions
- Focus fuzzing on promising bytes
Areas which influence fuzzing results
Areas which influence fuzzing results
Heap Overflow Detection

Page (4096 byte), read- & write-able

Content:
- Meta Data
- Heap Data 1
- Meta Data
- Heap Data 2

Heap Overflow
Heap Overflow Detection

Page (4096 byte), read- & write-able

Unused (special pattern)

Meta Data

Heap Data 1

Page (4096 byte)
NOT read- & write-able

Heap Overflow

Page (4096 byte), read- & write-able

Unused (special pattern)

Meta Data

Heap Data 2

Page (4096 byte)
NOT read- & write-able
Use-After-Free Detection

Page (4096 byte), read- & write-able

Unused (special pattern)

Meta Data

Heap Data 1

FREE

Page (4096 byte)
NOT read- & write-able
Use-After-Free Detection

Page (4096 byte) NOT read- & write-able
Meta Data
Heap Data 1

Page (4096 byte) NOT read- & write-able

Access attempt
Heap Library

• **Libdislocator** (shipped with AFL)

• **https://github.com/DhavalKapil/libdheap**

• **AFL_HARDEN=1** make (Fortify Source & Stack Cookies)

• On Windows: **Page heap with Application Verifier**

• **Own heap allocator** which checks after free() all memory locations for a dangling pointer!
  • Detect Use-After-Free at free and not at use step
  • Concept similar to MemGC protection from Edge
Detecting not crashing vulnerabilities

• **LLVM has many useful sanitizers!**
  • Address-Sanitizer (ASAN)
    • `-fsanitize=address`
    • Out-of-bounds access (Heap, stack, globals), Use-After-Free, …
  • Memory-Sanitizer (MSAN)
    • `-fsanitize=memory`
    • Uninitialized memory use
  • UndefinedBehaviorSanitizer (UBSAN)
    • `-fsanitize=undefined`
    • Catch undefined behavior (Misaligned pointer, signed integer overflow, …)

• **DrMemory (based on DynamoRio)** if source code is not available

➤ **Use sanitizers during development !!!**
Detecting not crashing vulnerabilities

- **During corpus generation don’t use sanitizers ➔ performance**
  - After we have a good corpus, start fuzzing it with sanitizers / injected libraries
  - I prefer heap libraries because they are faster and run after the first fuzzing session the corpus against binaries with sanitizers for some days
  - I don’t use heap libraries for the master fuzzer (deterministic fuzzing must be fast)

- **AFL performance example; one core; no in-memory fuzzing:**
  - x64 binary: 1400 exec / sec
  - x86 binary: 1200 exec / sec
  - x86 hardened binary: 1150 exec / sec
  - x86 hardened binary + libdislocator: 600 exec / sec
  - x86 binary with Address Sanitizer: 200 exec / sec
Rules for fuzzing
Fuzzing rules

1. Start fuzzing!
2. Start with simple fuzzing, during fuzzing add more logic to the next fuzzer version
3. Use Code/Edge Coverage Feedback
4. Create a good input corpus (via download or feedback)
5. Minimize the number of sample files and the file size
6. Use sanitizers / heap libraries during fuzzing (not for corpus generation)
7. Modify the mutation engine to fit your input data
8. Skip the “initialization code” during fuzzing (fork-server, persistent mode, …)
9. Use wordlists to get a better code coverage
10. Instrument only the code which should be tested
11. Don’t fix checksums inside your Fuzzer, remove them from the target application (faster)
12. Start fuzzing!
Fuzzing can show the presence of bugs but cannot prove the absence of bugs!
Thank you for your attention!

I wrote a vulnerability scanner that abstracts all the predicates in a binary, traverses the callgraph and generates phormulae to run then with a SMT solver. I found 1 vuln in 3 days with this tool.

He wrote a dumb ass fuzzer and found 5 vulns in 1 day.

Good thing I'm not a n00b like that guy.

Source: Twitter
For any further questions contact your SEC Consult Expert.

René Freingruber
@ReneFreingruber
r.freingruber@sec-consult.com
+43  676 840 301 749

SEC Consult Unternehmensberatung GmbH
Mooslackengasse 17
1190 Vienna, AUSTRIA

www.sec-consult.com