Building Advanced Coverage-guided Fuzzer for Program Binaries

NGUYEN Anh Quynh <aquynh@gmail.com>
WEI Lei

17/11/2017

Zero Nights, Moscow 2017
Self-introduction

• NGUYEN Anh Quynh, PhD <aquynh @ gmail.com>
  • Nanyang Technological University
  • Operating System, Virtual Machine, Binary Analysis, etc
  • Reverse trilogy: Capstone, Unicorn & Keystone

• WEI Lei, PhD
  • Nanyang Technological University
  • ~ 60 CVEs in Adobe, Apple, PHP etc
  • ~ 50 bug bounties from iDefense VCP, TippingPoint ZDI, and HackerOne.
Agenda

• Coverage-guided fuzzer
  • Background
  • Issues of public guided fuzzers

• Darko fuzzer
  • Features
  • Design & Implementation

• Demo & bugs found

• Conclusions
Coverage-guided Fuzzer
Fuzzer

• Automated software testing technique to find bugs
  • Feed craft input data to the program under test
  • Monitor for errors like crash/hang/memory leaking
  • Focus more on exploitable errors like memory corruption, info leaking
• Maximize code coverage to find bugs
• Blackbox fuzzing
• Whitebox fuzzing
• Graybox fuzzing
Coverage-guided fuzzer

- Instrument target binary to collect coverage info
- Mutate the input to maximize the coverage
- Repeat above steps to find bugs
  - Proved to be very effective
    - Easier to use/setup & found a lot of bugs
  - Trending in fuzzing technology
    - American Fuzzy Lop (AFL) really changed the game
Public guided fuzzers

- AFL
  - Requires source code for instrumentation build
  - Supports *nix binary via emulation mode (Qemu)
- AFL-Cygwin
  - AFL ported to Windows via Cygwin
  - Slow, buggy & development stalled
- WinAFL
  - Windows fork - needs persistent mode support
- AFL-Dyninst
  - Static-based instrumentation struggle on complicated binaries
  - No Windows support
Problems of public guided fuzzers

• Poor support for fuzzing binary
  • AFL emulation mode based on QEMU is limited
    • Only support Linux
    • Limitation of QEMU user mode emulation
  • Only WinAFL handles Windows closed source binaries
• Tricky to use
  • WinAFL persistent mode is really painful
• Suffer on performance & stability
  • DynamoRio is slow & fails to work on some large binaries
  • Needs persistent mode to perform well
DARKO Fuzzer
Darko design

- Motivation: no coverage-guided fuzzer for Windows (Dec 2015)
- Fork AFL fuzzing code & ported to Windows (Apr 2016)
  - Rewrite to work with our target instrumentation
- Support closed source binary for all platforms & architectures
  - To have a cross-platform/architecture fuzzer
  - Build our own instrumentation from scratch (Apr 2016)
    - Replaced with SKORPIO - multi-arch / platform (2017)
- Support selective binary fuzzing
- Support persistent mode
- Various other enhancements to AFL (2017)
Darko features

• Pure software-based
• Cross-platform/architecture
  • Native compiled (MSVC on Windows, GCC/Clang on *nix)
• Binary support
  • Full & selective binary fuzzing + Persistent mode
• Fast + stable
  • Stable & support all kind of binaries
  • Order of magnitude faster than DBI/Emulation approaches
Darko implementation - Overview

• AFL-compatible instrumentation
• PoC: AFL-Cygwin + PIN Probe mode (Apr 2016)
  • Applicable to user-space 32-bit Windows binaries
  • Flexible test case post-processor
  • Found bugs in Adobe Reader, Windows Journal, etc
• Static analysis + dynamic binary rewriting (SMC)
  • Speed much better than full binary DBI
• Near native execution speed, ASLR / threading compatible
• Support Windows, Linux & MacOS
• Support for non-X86 architectures underway
Challenges in static analysis

- CFG recovery: correctness v.s. completeness
  - Differentiate data (globals, vftables, jump tables) from code
  - Current effective instrumentation rate: > 60%
  - Rely on IDA Pro to handle compilers & optimizations

- Scalability
  - Tested & works well on Adobe Reader modules (< 10MB - 30MB)
  - For certain compilers, still have FP+ (e.g., mshtml.dll, ~25MB)
Instrumentation
Dynamic instrumentation

- Lesson learned from fuzzers based on DBI (Pin/DynamoRio)
  - Unstable & unreliable
  - Limitation on platforms & architectures
  - Poor performance
    - Cannot do selective instrumentation
- Hooking based mechanism
  - Lightweight & selective
  - Offline analysis on where to instrument
    - Handled with static analysis (beforehand)
Dynamic instrumentation (2)
SKORPIO instrumentation engine

- Cross-platform: Windows, MacOS, Linux, BSD, etc
- Cross architecture: X86, ARM, ARM64, Mips, PowerPC, Sparc
- Multi-level
  - Userspace & OS kernel
  - Instruction level (vs typical function-entry-only)
- Lightweight
  - Implemented in pure C, focus on low-level hooking mechanism
  - Super fast: can be 100x faster than available public hooking frameworks - thanks to many optimization
SKORPIO engine (2)

- Decode instructions at hooking place
  - Use Capstone disassembler (X86, ARM, ARM64, Mips, Sparc, PPC, ...)
- Binary rewrite on code relocation
  - Use Keystone assembler (X86, ARM, ARM64, Mips, Sparc, PPC, ...)
- Install user-provided callback at instrumentation hook
- Enable customization/optimization for all requirements
  - Hooking types (JMP or CALL, RET or naked callback)
  - Trampoline setting
  - Thread & internal memory management (OS-agnostic)
**Windows instrumentation**

- Inject instrumentation into target binary
  - Instrumentation comes in DLL form
  - `DLLMain()` runs before main program
- Considered Dynamic DLL injection, but rejected
  - Not portable
- Static inject DLL file into target binary
  - Analyze target PE file to locate Sections & Import Directory
  - Append 1 section to relocate Import Directory
  - Point Import Directory Table to the new appended section
  - Append a new entry for injected DLL
Linux & MacOS instrumentation

- LD_PRELOAD to dynamically inject instrumentation
  - Take place before main program runs
  - Linux: shared object file (.so)
  - MacOS: dynamic library (.dylib)
- Inject all instrumentation at initialisation time
  - Can be up to 100k hooks, so must do as quickly as possible
- Inject forkserver at program entry-point, so it takes over later
Detect heap memory corruption

- **Windows**
  - Enable PageHeap for fuzzing target
  - Low-level exception handling from Windows core

- **MacOS & Linux**
  - Built-in memory debugging for better control & performance
    - Overload malloc(), free() & co
    - Utilize MMU to detect overflow/underflow errors
      - Off-by-one error
    - Use-after-free error
Demo & bugs found
Some results

• PoC (Apr 2016): AFL-Cygwin + Intel PIN probe
  • Adobe Reader U3D: 2 unique bugs in 10 hours
    • CVE-2016-1116: Adobe Reader DC U3D e3_node OOB Access Vulnerability
    • Able to quickly rediscover 12+ bugs on older version:
      • CVE-2014-0523: Adobe Reader U3D Model Node Arbitrary Free Vulnerability
      • CVE-2014-0565: Adobe Reader U3D Line Set Continuation Memory Corruption
      • CVE-2014-9165: Adobe Reader U3D New Object Block Use-after-Free Vulnerability
      • CVE-2015-5586: Adobe Reader U3D Node Blocks Arbitrary Free Memory Corruption
      • CVE-2015-6683: Adobe Reader U3D Bone Weight Modifier Use-after-Free Vulnerability
      • CVE-2016-0933: Adobe Reader DC U3D Bone Weight Modifier OOB Access Vulnerability
      • CVE-2016-1037: Adobe Reader DC U3D Line Set Continuation OOB Access Vulnerability
    • More ... (will release the repros on GitHub)

• Libxml2-2.7.8.win32 - 10 unique bugs in a week

• Windows Journal - some bugs
Experiments

- Libxml2 - native, not compatible with persistent mode
- Native run with persistent mode:
  - UnRAR - persistent mode + parallel fuzzing
  - Msxml6 - persistent mode + parallel fuzzing
  - Adobe Reader - Javascript engine
  - Adobe Reader 3D
Demos - libxml2.dll
Demos - unrar.exe
Demos - afl-tmin.exe

λ afl-tmin.exe -t 1000 -i id 000002 11 -o id 000002 11.trim -- fuzz-xml.exe &>null
afl-tmin.exe 2.49s by Nguyen Ann OuyNh, 2017
Based on AFL 2.49b by lrarnuf@gmail.com

Read 68 bytes from 'id 000002 11'.
Performing dry run (timeout = 1000 ms)... Program terminates normally, minimizing in instrumented mode.
Stage #0: One-time block normalization...
Block normalization complete, 60 bytes replaced.
--- Pass #1 ---
Stage #1: Removing blocks of data...
Block length = 4, remaining size = 63
Block length = 2, remaining size = 36
Block length = 1, remaining size = 35
Block removal complete, 33 bytes deleted.
Stage #2: Minimizing symbols (8 code points)... Symbol minimization finished, 1 symbol (1 byte) replaced.
Stage #3: Character minimization...
Character minimization done, 0 bytes replaced.
--- Pass #2 ---
Stage #1: Removing blocks of data...
Block length = 2, remaining size = 35
Block length = 1, remaining size = 35
Block removal complete, 0 bytes deleted.
File size reduced by: 48.53% (to 35 bytes)
Characters simplified: 174.20%
Number of execs done: 72
Fruitless execs: path=46 crash=0 hang=0

Writing output to 'id_000002_11.trim'...
We're done here. Have a nice day!
Demos - afl-analyze.exe

λ afl-analyze.exe -q -t 1000 -i note.xml -- fuzz.xml.exe @
afl-analyze.exe 2.49b by Nguyen Anh Quynh, 2017
Based on AFL 2.49b by <lcamtuf@google.com>

[+1] Read 123 bytes from 'note.xml'.
[+1] Performing dry run (timeout = 1000 ms)... 
[+1] Analyzing input file (this may take a while)... 

01 - no-op block 
01 - superficial content 
01 - critical stream 
01 - "magic value" section

01 - suspected length field 
01 - suspected cksum or magic int 
01 - suspected checksummed block

[000000] < ? xml 1 #20 version = " 1
[000016] . 0 " > #0d #0a < note > #0d #0a #20
[000032] < to > Tove < / to > #0d #0a #20
[000048] < from > Jani < / from > #0d #0a #20
[000064] < body > Don't #20 forget #20 me #20 this #20 weekend ! < / body > #0d #0a

[+1] We're done here. Have a nice day!
### Demos - AFL -Q (Linux) vs Darko

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Run Time</th>
<th>Last New Path</th>
<th>Last Unique Crash</th>
<th>Last Unique Hang</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 days, 0 hrs, 1 min, 42 sec</td>
<td>0 days, 0 hrs, 1 min, 35 sec</td>
<td>none seen yet</td>
<td>none seen yet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Now Processing</th>
<th>Paths Timed Out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 (60.00%)</td>
<td>0 (0.00%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Favored Paths</th>
<th>New Edges on</th>
<th>Total Crashes</th>
<th>Total Mouts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 (100.00%)</td>
<td>5 (100.00%)</td>
<td>0 (0 unique)</td>
<td>0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Paths Geometry</th>
<th>Levels</th>
<th>Pending</th>
<th>Pend Fav</th>
<th>Own Finds</th>
<th>Imported</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Flips</td>
<td>0/0, 0/0, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Byte Flips</td>
<td>0/10, 0/5, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Arithmetics</td>
<td>1/509, 0/50, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Known Ints</td>
<td>0/57, 0/140, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Dictionary</td>
<td>0/0, 0/0, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Havoc</td>
<td>3/87.9, 0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
<tr>
<td>Trim</td>
<td>n/a, 8.00%</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>n/a</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU Usage</th>
<th>140%</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CPU Usage</th>
<th>137%</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Overall Results</th>
<th>Cycles Done</th>
<th>Total Paths</th>
<th>Uniq Crashes</th>
<th>Uniq Hangs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL -Q (Linux)</td>
<td>133</td>
<td>71</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Darko</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**American Fuzzy Lop 2.51b (test2)**
### American Fuzzy Lop 2.51b (Test1)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 1 min, 37 sec</td>
<td>Cycles done: 21</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 1 min, 16 sec</td>
<td>Total paths: 7</td>
</tr>
<tr>
<td>Last unique crash: 0 days, 0 hrs, 1 min, 30 sec</td>
<td>Uniq crashes: 1</td>
</tr>
<tr>
<td>Last uniq hang: none seen yet</td>
<td>Uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 6 (85.71%)</td>
<td>Map density: 0.00% / 0.01%</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.00 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: havoc</td>
<td>Favored paths: 7 (100.00%)</td>
</tr>
<tr>
<td>Stage execs: 40/512 (7.81%)</td>
<td>New edges on: 7 (100.00%)</td>
</tr>
<tr>
<td>Total execs: 102k</td>
<td>Total crashes: 42 (1 unique)</td>
</tr>
<tr>
<td>Exec speed: 1010/sec</td>
<td>Total tmouts: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flips: 0/144, 0/137, 1/123</td>
<td>Levels: 5</td>
</tr>
<tr>
<td>Byte flips: 0/18, 0/11, 0/2</td>
<td>Pending: 0</td>
</tr>
<tr>
<td>Arithmetics: 2/1008, 0/74, 0/0</td>
<td>Pending fav: 0</td>
</tr>
<tr>
<td>Known ints: 0/101, 0/303, 0/88</td>
<td>Own finds: 6</td>
</tr>
<tr>
<td>Dictionary: 0/0, 0/0, 0/0</td>
<td>Imported: n/a</td>
</tr>
<tr>
<td>Havoc: 4/90.0k, 0/10.5k</td>
<td>Stability: 300.00%</td>
</tr>
<tr>
<td>Trim: 55.56%/2, 0.00%</td>
<td>[CPU: 93%]</td>
</tr>
</tbody>
</table>

---

### American Fuzzy Lop 2.51b (Test2)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 1 min, 35 sec</td>
<td>Cycles done: 81</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 0 min, 32 sec</td>
<td>Total paths: 7</td>
</tr>
<tr>
<td>Last unique crash: 0 days, 0 hrs, 0 min, 17 sec</td>
<td>Uniq crashes: 1</td>
</tr>
<tr>
<td>Last uniq hang: none seen yet</td>
<td>Uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 6 (85.71%)</td>
<td>Map density: 0.00% / 0.01%</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.00 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: havoc</td>
<td>Favored paths: 7 (100.00%)</td>
</tr>
<tr>
<td>Stage execs: 762/7048 (35.94%)</td>
<td>New edges on: 7 (100.00%)</td>
</tr>
<tr>
<td>Total execs: 115k</td>
<td>Total crashes: 14 (1 unique)</td>
</tr>
<tr>
<td>Exec speed: 1153/sec</td>
<td>Total tmouts: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flips: 0/138, 0/123, 0/115</td>
<td>Levels: 5</td>
</tr>
<tr>
<td>Byte flips: 0/17, 0/10, 0/1</td>
<td>Pending: 0</td>
</tr>
<tr>
<td>Arithmetics: 2/952, 0/50, 0/0</td>
<td>Pending fav: 0</td>
</tr>
<tr>
<td>Known ints: 0/96, 0/280, 0/44</td>
<td>Own finds: 6</td>
</tr>
<tr>
<td>Dictionary: 0/0, 0/0, 0/0</td>
<td>Imported: n/a</td>
</tr>
<tr>
<td>Havoc: 5/112k, 0/0</td>
<td>Stability: 100.00%</td>
</tr>
<tr>
<td>Trim: 20.00%/2, 0.00%</td>
<td>[CPU: 93%]</td>
</tr>
</tbody>
</table>
Demos - MacOS
Conclusions

• **DARKO** is an advanced coverage-guided fuzzer
  • Pure software-based
  • Cross-platform/architecture
  • Binary support
    • Fuzz full binary + Persistent mode
  • Fast + stable

• **SKORPIO** engine will be released to public in near future
Questions?

Building Advanced Coverage-guided Fuzzer for Program Binaries

NGUYEN Anh Quynh <aquynh@gmail.com>

WEI Lei