Finding security vulnerabilities with modern fuzzing techniques
Introduction

- **René Freinruber** ([r.freinruber@sec-consult.com](mailto:r.freinruber@sec-consult.com))
  - Twitter: [@ReneFreinruber](https://twitter.com/ReneFreinruber)
  - BSc @ TU Vienna, Currently MSc @ Technikum Vienna
  - Senior Security Consultant at SEC Consult
    - Red Team, Reverse Engineering, Exploit development, Fuzzing
    - Trainer: Secure C/C++, Reverse Engineering and Red Teaming
  - Previous talks:
    - 2014: Bypassing EMET
      - 31C3, DeepSec, ZeroNights, RuxCon, ToorCon and NorthSec
    - 2015: Bypassing Application Whitelisting
      - CanSecWest, DeepSec, Hacktivity, NorthSec, IT-SeCX, BSides Vienna and QuBit
    - 2016: Hacking companies via firewalls
      - DeepSec, BSides Vienna, DSS ITSEC and IT-SeCX (lightning talks at recon.eu and hack.lu)
    - Since 2017 fuzzing talks
      - DefCamp, Heise devSec, IT-SeCX, BSides Vienna, RuhrSec
We are hiring!

Founded 2002 (15+ years consulting)

Strong customer base in Europe and Asia

100+ Security experts

120+ certifications

500+ Security audits per year
Some rules

• Ask anything anytime!
  • My english is not the best – please use simple words 😊

• Tell me if I’m too fast!

• Tell me if there is anything you don’t understand!

• Tell me if it’s too easy / too hard!

• Contact me:
  • E-Mail: r.freingruber@sec-consult.com
  • Twitter: @ReneFreingruber

… Tell me if you want to have a break 😊
• Some demos building on great stuff from others!
  • LibFuzzer Tutorial (From Google and Workshop from Dor1s, @Dor3s)
  • Seccon 2016 CTF chat binary
  • FuzzGoat (from fuzzstati0n)
  • Of course all the great fuzzers like AFL, LibFuzzer, WinAFL, honggfuzz, …

• Many demos just require to type in commands…
  • It's the nature of the topic (we don't want to implement everything our self)
  • I want to use the full time to learn you as much as possible (the basics!)
  • If you want “open examples” just try the learned stuff at home with some applications! (if you have questions drop me a mail or write on twitter)
Fuzzing
Definition of fuzzing (source Wikipedia):

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks.
Why do we need Fuzzing?

Microsoft Security Development Lifecycle (SDL) Process

<table>
<thead>
<tr>
<th>1. TRAINING</th>
<th>2. REQUIREMENTS</th>
<th>3. DESIGN</th>
<th>4. IMPLEMENTATION</th>
<th>5. VERIFICATION</th>
<th>6. RELEASE</th>
<th>7. RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Perform Security and Privacy Risk Assessments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


I also recommend fuzzing during implementation

Example: You finished a complex task and you are not sure if it behaves correctly and is secure

➔ Start a fuzzer over night / the weekend ➔ Check corpus
Where input to file parsing code could have crossed a trust boundary, file fuzzing must be performed on that code. [...] 

- **An Optimized set of templates must be used.** Template optimization is based on the maximum amount of code coverage of the parser with the minimum number of templates. Optimized templates have been shown to double fuzzing effectiveness in studies. **A minimum of 500,000 iterations, and have fuzzed at least 250,000 iterations since the last bug found/fixed that meets the SDL Bug Bar.**

Advantages:
- Very fast (in most cases much faster than manual source code review)
- You don’t have to pay a human, only the power consumption of a computer
- It runs 24 hours / 7 days, a human works only 8 hours / 5 days
- Scalable (want to find more bugs? ➔ Start 100 fuzzing machines instead of 1)

Disadvantages:
- Deep bugs (lots of pre-conditions) are hard to find
- Typically you can’t find business logic bugs
Types of Fuzzing

- **Grammar-based Fuzzing:**
  - Idea: Implement the file format / protocol inside your fuzzer
  - Example: “at offset 4 is an unsigned dword, at offset 10 is a checksum, at offset 14 is a null-terminated string, at offset 20 a type field, …”
  - Covers everything which you defined (but maybe forgets corner cases which you didn’t think of)
  - After the (long) initial work, the fuzzer covers lots of corner cases very fast
  - Examples: Peach, Domato, Boofuzz, Sulley, Spike, …

- **Feedback-based Fuzzing:**
  - Let the fuzzer learn the file format itself ➔ No initial work required (fast results)
  - However, learning the format can take a long time and maybe some checks could not be learned by the fuzzer
  - ➔ If we build better feedback-based fuzzers it has no (or just small) drawbacks
  - Examples: AFL, WinAFL, LibFuzzer, Honggfuzz, …
  - Side note: Use this technique for binary inputs. For “interactive” inputs (e.g.: JavaScript / DOM in browser or shell-like software which listens on a port) this technique is only partially useful.
Feedback-based Fuzzing / Coverage-guided Fuzzing
Consider this code:

```c
if (input_buffer[2] == 0x55) {
  printf("0x55 case
");
} else {
  printf("Default case
");
}
if (input_buffer[3] == 0xaa) {
  if (input_buffer[4] == 0xff) {
    if (input_buffer[5] == 0xcc) {
      printf("Vulnerability triggered!
");
    }
  }
}
```

Fuzzer queue:

Input 1: 00 00 00 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Feedback based fuzzing

Fuzz input 1: 01 00 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
**Feedback based fuzzing**

**Fuzz input 1:** 02 00 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00
**Feedback based fuzzing**

**Fuzz input 1:** FF 00 00 00 00 00

if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 01 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:

Input 1: 00 00 00 00 00 00
**Feedback based fuzzing**

**Fuzz input 1:** 00 02 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 FF 00 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Feedback based fuzzing

**Fuzz input 1:** 00 00 01 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 54 00 00 00

if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Feedback based fuzzing

**Fuzz input 1:** 00 00 55 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00

Input 2: 00 00 55 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 56 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 FF 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
**Feedback based fuzzing**

➡ **Fuzz input 1:** 00 00 00 A9 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**

Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 00 AA 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 00 AB 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:

Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 1: 00 00 00 00 00 FF

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
**Feedback based fuzzing**

**Fuzz input 2:** 01 00 55 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

**Fuzzer queue:**
- Input 1: 00 00 00 00 00 00
- Input 2: 00 00 55 00 00 00
- Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 2: 02 00 55 00 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}

if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 2: 00 00 55 AA 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
}
else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
- Input 1: 00 00 00 00 00 00
- Input 2: 00 00 55 00 00 00
- Input 3: 00 00 00 AA 00 00

<no new entry because we already saw that coverage>
Feedback based fuzzing

Fuzz input 2: 00 00 55 00 00 FF

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 3: 01 00 00 AA 00 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
- Input 1: 00 00 00 00 00 00
- Input 2: 00 00 55 00 00 00
- Input 3: 00 00 00 AA 00 00
Feedback based fuzzing

Fuzz input 3: 00 00 00 AA FF 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:
Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Input 4: 00 00 00 AA FF 00
Feedback based fuzzing

Fuzz input 3: 00 00 00 AA 00 FF

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!*\n");
        }
    }
}
```

Fuzzer queue:

- Input 1: 00 00 00 00 00 00
- Input 2: 00 00 55 00 00 00
- Input 3: 00 00 00 AA 00 00
- Input 4: 00 00 00 AA FF 00
Feedback based fuzzing

Fuzz input 4: 01 00 00 AA FF 00

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

Fuzzer queue:

Input 1: 00 00 00 00 00 00
Input 2: 00 00 55 00 00 00
Input 3: 00 00 00 AA 00 00
Input 4: 00 00 00 AA FF 00
Feedback based fuzzing

°Fuzz input 4: 00 00 00 AA FF CC

```c
if (input_buffer[2] == 0x55) {
    printf("0x55 case\n");
} else {
    printf("Default case\n");
}
if (input_buffer[3] == 0xaa) {
    if (input_buffer[4] == 0xff) {
        if (input_buffer[5] == 0xcc) {
            printf("Vulnerability triggered!\n");
        }
    }
}
```

°Fuzzer queue:

- Input 1: 00 00 00 00 00 00
- Input 2: 00 00 55 00 00 00
- Input 3: 00 00 00 AA 00 00
- Input 4: 00 00 00 AA FF 00

→ Vulnerability found!
Popular Fuzzers
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, … (more on this later!)
  • Simple to use! (start fuzzing in under 1 minute!)
  • Good designed! (very fast & good heuristics)
Feedback based fuzzing

- Consider this code (x = argc):

```c
if(x > 3) {
    puts("Test1\n");
} else {
    puts("Test2\n");
}
puts("Test3\n");
return 0;
```

```
user-VirtualBox# gcc -o test test.c
test
user-VirtualBox# ./test 1
Test2
Test3
user-VirtualBox# ./test 1 2 3 4 5 6
Test1
Test3
```
Feedback based fuzzing

- Basic Blocks:
Feedback based fuzzing

- Just use afl-gcc instead of gcc…

```bash
user-VirtualBox# afl-gcc -o test2 test.c
afl-cc 2.35b by <lcamtuf@google.com>
afl-as 2.35b by <lcamtuf@google.com>
[+1] 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

• Result:

Store old register values

Instrumentation

Restore old register values
American Fuzzy Lop - AFL

Without instrumentation just the first level will be discovered (or it would take an extremely long time)

Source:
http://lcamtuf.coredump.cx/afl_gzip.png
Practice: Lection 1

**Topic:** Lection 1 – Simple AFL fuzzing

**Duration:** 5 – 10 min

**Description:** Try AFL in action with a simple and small target.
### AFL Status Screen

#### Process Timing
- **Run Time**: 42 days, 19 hrs, 27 min, 41 sec
- **Last New Path**: 0 days, 1 hrs, 45 min, 10 sec
- **Last Unique Crash**: 5 days, 19 hrs, 58 min, 31 sec
- **Last Unique Hang**: 1 days, 16 hrs, 58 min, 37 sec

#### Cycle Progress
- **Now Processing**: 1550* (10.74%)
- **Paths Timed Out**: 0 (0.00%)

#### 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
- **Cycles Done**: 3
- **Total Paths**: 14.4k
- **Unique Crashes**: 25
- **Unique Hangs**: 161

#### Findings in Depth
- **Favored Paths**: 2220 (15.39%)
- **New Edges On**: 3431 (23.78%)
- **Total Crashes**: 1286 (25 Unique)
- **Total Tmouts**: 25.5K (224 Unique)

#### Path Geometry
- **Levels**: 27
- **Pending**: 10.5k
- **Pending Fav**: 1
- **Own Finds**: 14.4K
- **Imported**: n/a
- **Stability**: 100.00%
• 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)
  - Downside: 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
**Example:** Let’s say we want to fuzz LIEF (Library to Instrument Executable Formats from Quarkslab) with PE files

- Our real goal: Generate a good PE-corpus which we can use for fuzzing AntiVirus engines (therefore we first fuzz different open source PE libraries)
- Side note: LIEF is a very powerful PE library and my first choice in PE libraries! That’s why I have chosen it as target here!

**Step 1: Get possible input files:**

- Write a python script to grab all small .exe / .dll / .sys / … files from a workstation (execute it on Windows XP, Vista, Win7, Win8.1, Win10, …)
- Add public available corpus files: I found additional 2149 files on the internet
- Result: Many thousand files
**Step 2: Recompile application with afl-gcc**

- I modified the c++ “pe_reader” example to catch all exceptions (otherwise AFL would incorrectly identify thrown exceptions as crashes)
- Export CC & CXX and call cmake / configure

```bash
git clone https://github.com/lief-project/LIEF.git
cd LIEF
mkdir build; cd build
export CC=afl-gcc
export CXX=afl-g++
cmake -DPYTHON_VERSION=2.7 ..
make
```
Step 3: Minimize the files to a small corpus (Corpus Distillation)

- Optional: Do everything on a RAM disk (e.g.: /dev/shm):
  
  ```
  mkdir /ramdisk
  mount -t tmpfs -o size=4G tmpfs /ramdisk
  ```

- Example: The 2149 public files can be reduced to 377 files

```bash
/ramdisk# afl-cmin -i input dir/ -o input after cmin/ -- ./pe reader.cpp afl exceptions handled @@
corpus minimization tool for afl-fuzz by <lcamtuf@google.com>

[*] Testing the target binary...
[+] OK, 4393 tuples recorded.
[*] Obtaining traces for input files in 'input_dir'/...
  Processing file 2149/2149...
[*] Sorting trace sets (this may take a while)...
[+] Found 27613 unique tuples across 2149 files.
[*] Finding best candidates for each tuple...
  Processing file 2149/2149...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
  Processing tuple 27613/27613...
[+] Narrowed down to 377 files, saved in 'input after cmin/'
```
• **Step 4: Minimize file size of the files in the corpus**
  - Not very efficient in the case of PE files (byte removal / modification lead to invalid checksum ➔ different executed code ➔ AFL-tmin can’t reduce it)
  - For example: In total the filesize of all 377 files together was just reduced by 400 KB

  ```
  ./afl-tmin -i testcase_file -o testcase_out_file
  -- /path/to/tested/program [...program's cmdline...] @@
  ```

• **Step 5: Start fuzzing**

  ```
  afl-fuzz -i input_after_tmin -o output/ -M master -- ./pe_reader @@
  afl-fuzz -i input_after_tmin -o output/ -S slave1 -- ./pe_reader @@
  ```
Example

Side note:
This are in reality only 2 (not exploita.) bugs in the code.
(LIEF ships with LibFuzzer scripts!)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time</td>
<td>Cycles done: 218</td>
</tr>
<tr>
<td>Last new path</td>
<td>Total paths: 5418</td>
</tr>
<tr>
<td>Last uniq crash</td>
<td>Uniq crashes: 734</td>
</tr>
<tr>
<td>Last uniq hang</td>
<td>Uniq hangs: 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing</td>
<td>Map coverage: 6.63% / 22.81%</td>
</tr>
<tr>
<td>Paths timed out</td>
<td>Count coverage: 3.83 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>Favorited paths: 525 (9.69%)</td>
</tr>
<tr>
<td>Stage execs: 192/384 (50.00%)</td>
<td>New edges on: 943 (17.40%)</td>
</tr>
<tr>
<td>Total execs: 104M</td>
<td>Total crashes: 701k (734 unique)</td>
</tr>
<tr>
<td>Exec speed: 315.2/sec</td>
<td>Total tmoouts: 1822 (113 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flips: n/a, n/a, n/a</td>
<td>Levels: 9</td>
</tr>
<tr>
<td>Byte flips: n/a, n/a, n/a</td>
<td>Pending: 16</td>
</tr>
<tr>
<td>Arithmetics: n/a, n/a, n/a</td>
<td>Pending fav: 0</td>
</tr>
<tr>
<td>Known ints: n/a, n/a, n/a</td>
<td>Own finds: 2149</td>
</tr>
<tr>
<td>Dictionary: n/a, n/a, n/a</td>
<td>Imported: 2900</td>
</tr>
<tr>
<td>Havoc: 17/99/34.7M, 1084/63.7M</td>
<td>Stability: 100.00%</td>
</tr>
<tr>
<td>Trim: 13.12%/5.92M, n/a</td>
<td></td>
</tr>
</tbody>
</table>

Title: Finding security vulnerabilities with Fuzzing | Responsible: R. Freingruber | Version / Date: V1.0/2018-05 | Confidentiality Class: public
© 2018 SEC Consult | All rights reserved
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...]
   ```
Real World example: CVE-2009-0385

- Real world example

- CVE-2009-0385

- Vulnerability from 2009 in FFmpeg
  - Vulnerability in parsing .4xm files

- More information (on exploit development) can be found in “A bug hunter’s diary” chapter 5
Real World example: CVE-2009-0385

- Input .4xm file (with video & audio stream):
Real World example: CVE-2009-0385

- Dumb fuzzer:

```
$ python dumb_fuzz.py
Crash with flipped bytes at index 0x1ae
Crash with flipped bytes at index 0x1af
Crash with flipped bytes at index 0x1b0
Crash with flipped bytes at index 0x1b1
Crash with flipped bytes at index 0x1ce
Crash with flipped bytes at index 0x1d2
```
Real World example: CVE-2009-0385

- Original:

- Fuzzer found crash:
Real World example: CVE-2009-0385

- Verify the crash in the debugger:

```
(gdb) r /home/rfr/Desktop/C_Schulung/examples/13.Fuzzing/1.FFmpeg/input_files/test.4xm
Starting program: /home/rfr/Desktop/C_Schulung/examples/13.Fuzzing/1.FFmpeg/FFmpeg_compile
FFmpeg version UNKNOWN, Copyright (c) 2000-2009 Fabrice Bellard, et al.
configuration: --prefix=/home/rfr/Desktop/C_Schulung/examples/13.Fuzzing/1.FFmpeg/FFmpeg
libavutil 49.12.0 / 49.12.0
libavcodec 52.10.0 / 52.10.0
libavformat 52.23.1 / 52.23.1
libavdevice 52.1.0 / 52.1.0
built on Jul 19 2016 15:18:05, gcc: 4.7.2

Program received signal SIGSEGV, Segmentation fault.
0x080ab5b3 in fourxm_read_header (s=0x88d8330, ap=0xbfffff030) at libavformat/4xm.c:178
178
fourxm->tracks[current_track].adpcm = AV_RL32(&header[i + 12]);
gdb) x /i $eip
> 0x080ab5b3 <fourxm_read_header+691>: mov DWORD PTR [eax+0x10],ebp

gdb) p /x $eax
2 = 0xffffffff

gdb) p /x $ebp
3 = 0x0
```
Real World example: CVE-2009-0385

- Attacker has a write-4 vulnerability (destination and value controlled):

```c
mov DWORD PTR [eax+0x10], ebp
```

Program received signal SIGSEGV, Segmentation fault.
0x080ab5b3 in fourxm_read_header (s=0x88d8330, ap=0xbffff030) at libavformat/4xm.c:178
178

```
(gdb) p /x $eip
=> 0x80ab5b3 <fourxm_read_header+691>:
   mov DWORD PTR [eax+0x10], ebp
```

$6 = 0x055555548

$7 = 0xbbbbbbbb
Real World example: CVE-2009-0385

• Now assume we don’t have a valid 4xm file:

• Before modification:

• After modification:
Real World example: CVE-2009-0385

• The dumb fuzzer can’t find the vulnerability anymore!

Many other crashes…
But not the real vulnerability at offset 0x1ae

Reason: In error case the code dereferences the pointer to the „strk“ chunk which is in this case NULL
Practice: Lection 2

**Topic:** Lection 2 – Real World Fuzzing FFMPEG with AFL

**Duration:** 10 – 15 min

**Description:** Try to fuzz FFMPEG with AFL
### AFL with CVE-2009-0385 (FFMPEG)

**american fuzzy lop 2.19b (ffmpeg)**

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done</td>
</tr>
<tr>
<td>0 days, 18 hrs, 52 min, 48 sec</td>
<td>0</td>
</tr>
<tr>
<td>last new path</td>
<td>total paths</td>
</tr>
<tr>
<td>0 days, 0 hrs, 0 min, 25 sec</td>
<td>1179</td>
</tr>
<tr>
<td>last uniq crash</td>
<td>uniq crashes</td>
</tr>
<tr>
<td>0 days, 1 hrs, 15 min, 58 sec</td>
<td>14</td>
</tr>
<tr>
<td>last uniq hang</td>
<td>uniq hangs</td>
</tr>
<tr>
<td>0 days, 0 hrs, 12 min, 5 sec</td>
<td>73</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing</td>
<td>map density</td>
</tr>
<tr>
<td>205 (17.39%)</td>
<td>5205 (7.94%)</td>
</tr>
<tr>
<td>paths timed out</td>
<td>count coverage</td>
</tr>
<tr>
<td>14 (1.19%)</td>
<td>2.39 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</td>
<td>favored paths</td>
</tr>
<tr>
<td>havoc</td>
<td>239 (20.27%)</td>
</tr>
<tr>
<td>stage execs</td>
<td>new edges on</td>
</tr>
<tr>
<td>34.6k/160k (21.64%)</td>
<td>376 (31.89%)</td>
</tr>
<tr>
<td>total execs</td>
<td>total crashes</td>
</tr>
<tr>
<td>19.8M</td>
<td>554 (14 unique)</td>
</tr>
<tr>
<td>exec speed</td>
<td>total hangs</td>
</tr>
<tr>
<td>373.4/sec</td>
<td>19.6k (73 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips</td>
<td>levels</td>
</tr>
<tr>
<td>91/5.51M, 30/5.51M, 21/5.51M</td>
<td>4</td>
</tr>
<tr>
<td>byte flps</td>
<td>pending</td>
</tr>
<tr>
<td>1/689k, 3/7463, 7/8669</td>
<td>1143</td>
</tr>
<tr>
<td>arithmetics</td>
<td>pend fav</td>
</tr>
<tr>
<td>50/383k, 10/27.5k, 6/11.5k</td>
<td>220</td>
</tr>
<tr>
<td>known ints</td>
<td>own finds</td>
</tr>
<tr>
<td>7/35.8k, 21/203k, 34/196k</td>
<td>1178</td>
</tr>
<tr>
<td>dictionary</td>
<td>imported</td>
</tr>
<tr>
<td>0/0, 0/0, 5/48.2k</td>
<td>n/a</td>
</tr>
<tr>
<td>havoc</td>
<td>variable</td>
</tr>
<tr>
<td>893/1.55M, 0/0</td>
<td>0</td>
</tr>
<tr>
<td>trim</td>
<td></td>
</tr>
<tr>
<td>26.75%/43.3k, 98.99%</td>
<td></td>
</tr>
</tbody>
</table>

[cpu000:161%]
AFL input with invalid 4xm file (strk chunk changed to strj)

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)
• Some hints on analyzing exploitability / root cause:

• First command to execute when the application crashes is:
  • x /2i $rip
    • X… examine ➔ print something at the given location ($rip in our case)
    • $rip … instruction pointer ➔ the current instruction
    • /2i … print 2 times data interpreted as instruction

• Now we see which instruction resulted in a crash and the next step is to understand why it crashed
• **Examples:**
  
  - Mov dword ptr [rcx+0x20], eax
  - Mov rbx, qword ptr [rax]
    
    - Every time you see [ and ] it means that we read/write from RAM memory. In most cases, the address inside the brackets is therefore wrong and resulted in a crash. So we would analyze rcx and rax in the above outputs: p /x $rcx ; p /x $rax
    - In many cases, you can control rcx or rax (e.g., it contains 0x414141...) then you maybe have an arbitrary read or write. In other cases, you may have a relative read/write and in other cases it contains a fixed address which can’t be accessed (which can indicate a use-after-free bug)
    - In many other cases, rcx or rax is just zero which resulted in a null pointer exception (because our input didn’t initialized it); This is in most cases not exploitable
Examples:

• => 0x4141414141414141: Cannot access memory at address 0x4141414141414141
  • This means $rip points to 0x4141... and therefore we had most likely a stack based buffer overflow and overwrote the return address on the stack
• ret
  • Sometimes you directly crash at the “ret” instruction (which is basically a “pop rip”) if the return address is invalid. This for example is the case in ARM gdb.
• inc eax
  • No obvious reason how this instruction could crash. This often occurs if $rip points to a memory region which is not marked as executable (DEP/NX protection). Therefore “inc eax” is stored in such a region. To verify you can type “shell”, then “pidof <applicationName” and then check: cat /proc/<PID>maps if the memory range is executable or not.
• Examples:
  
  • `leave`
    • A little bit more tricky: `leave` is the same as “mov esp, ebp ; pop ebp”. The move instruction cannot crash (if it’s in an executable memory range), therefore “pop ebp” must crash. Pop ebp reads from the stack (where ESP points to)

  • `<vfprintf>` mov dword ptr [rax], r9d
    • Since the crash occurred in a standard function (vfprintf) it often helps to check the stack backtrace with “backtrace”. Check the last function call in the application ➔ Arguments to the library function are very likely incorrect.
We use **CrashWalk from Ben Nagy for Crash Triage (crash analysis)**

- [https://github.com/bnagy/crashwalk](https://github.com/bnagy/crashwalk)
- Cw triage --root afl_output –afl
- Cwdump ./crashwalk.db
- Cwfind <crash hash>

- GDB / WinDbg Plugin !exploitable

- Another great possibility on Windows is the BugId tool by SkyLined
  - [https://github.com/SkyLined/BugId](https://github.com/SkyLined/BugId)

- Symbolic execution can also help in triage
  - For example: SymGDB, Triton, PONCE, Moflow tools
Practice: Lection 3

**Topic:** Lection 3 – FuzzGoat and Crash Triage

**Duration:** 5-10 min

**Description:** Learn to perform crash triage.
Edge vs. BasicBlock Coverage

- 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)
Feedback based fuzzing

- Basic Blocks:
Edge vs. BasicBlock Coverage

• 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)

➤ **Without shifting A->B and B->A** are indistinguishable
• AFL receives after every iteration a “coverage_map”.
  • Every byte in the map represents a hitcount for an edge (or basic block)

• Hitcounts are translated to bucket indexes to mark a unique edge + hitcount combination with one bit!

// Hitcount bucket: [0] [1] [2] [3] [4-7] [8-15] [16-31] [32-127] [128+]
// Bucket Value
// One at bit offset 

• ➔ A global coverage map stores information about the already seen coverage by doing an AND after every iteration. If one iteration has at one bit a 1 where the global coverage map stores a 0 new behavior is detected ➔ Very fast check for new behavior!
Edge vs. BasicBlock Coverage

- Consider the following code; Our input file has 0xaa at offset 10, 0x00 at all other positions

```c
char *p;
// ...
if(input[10] == 0xaa) {
    p = &(input + 20);
}
// ...
if(input[4] == 0xbb) {
    printf("Input string: %s\n", p);
}
```

- BasicBlock Coverage: Vulnerability (uninitialized variable) will not be found (or very late)
- Edge Coverage: Vulnerability will be found because input will be mutated to not contain 0xaa at offset 10 (This input will be added to the queue)
Some public fuzzing numbers
Some public fuzzing numbers

- **Example:** Talk by Charlie Miller from 2010 „Babysitting an Army of Monkeys“

- Fuzzed Adobe Reader, PPT, OpenOffice, Preview

- **Strategy:** Dumb fuzzing
  - Download many input files (**PDF 80 000 files**)
  - **Minimal corpus** of input files with valgrind (**PDF 1515 files**)
  - Measure CPU to know when file parsing ended
  - Only change bytes (no adding / removing)
  - Simple fuzzer in 5 LoC
Some public fuzzing numbers

Fuzzer:

```python
numwrites = random.randint(0, int(math.ceil((float(len(buf)) / FuzzFactor)))) + 1
for j in range(numwrites):
    rbyte = random.randint(0, 255)
    rn = random.randint(0, len(buf))
    buf[rn] = '%c' % (rbyte)
```

Source: Charlie Miller „Babysitting an Army of Monkeys“
Some public fuzzing numbers

Results:

- 3 months fuzzing
- 7 Million Iterations

Crashes with unique EIP:

Source: Charlie Miller „Babysitting an Army of Monkeys“
Some public fuzzing numbers

Other numbers from Jaanus Kääp:

- Code coverage for minset calculation (no edge coverage because of speed)
- PDF ➔ initial set 400 000 files ➔ Corpus 1217 files
- DOC ➔ initial set 400 000 files ➔ Corpus 1319 files
- DOCX ➔ initial set 400 000 files ➔ Corpus 2222 files
Google fuzzed Adobe Flash in 2011:

“What does corpus distillation look like at Google scale? Turns out we have a large index of the web, so we cranked through 20 terabytes of SWF file downloads followed by 1 week of run time on 2,000 CPU cores to calculate the minimal set of about 20,000 files. Finally, those same 2,000 cores plus 3 more weeks of runtime were put to good work mutating the files in the minimal set (bitflipping, etc.) and generating crash cases. “

The initial run of the ongoing effort resulted in about 400 unique crash signatures, which were logged as 106 individual security bugs following Adobe's initial triage.

• Source: https://security.googleblog.com/2011/08/fuzzing-at-scale.html
Google fuzzed the DOM of major browsers in 2017:


We tested 5 browsers with the highest market share: Google Chrome, Mozilla Firefox, Internet Explorer, Microsoft Edge and Apple Safari. We gave each browser approximately 100,000,000 iterations with the fuzzer and recorded the crashes. (If we fuzzed some browsers for longer than 100,000,000 iterations, only the bugs found within this number of iterations were counted in the results.) Running this number of iterations would take too long on a single machine and thus requires fuzzing at scale, but it is still well within the pay range of a determined attacker. For reference, it can be done for about $1k on Google Compute Engine given the smallest possible VM size, preemptable VMs (which I think work well for fuzzing jobs as they don't need to be up all the time) and 10 seconds per run.
Google fuzzed the DOM of major browsers in 2017:

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Browser</th>
<th>Engine</th>
<th>Number of Bugs</th>
<th>Project Zero Bug IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Chrome</td>
<td>Blink</td>
<td>2</td>
<td>994, 1024</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Firefox</td>
<td>Gecko</td>
<td>4**</td>
<td>1130, 1155, 1160, 1185</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Internet Explorer</td>
<td>Trident</td>
<td>4</td>
<td>1011, 1076, 1118, 1233</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Edge</td>
<td>EdgeHtml</td>
<td>6</td>
<td>1011, 1254, 1255, 1264, 1301, 1309</td>
</tr>
<tr>
<td>Apple</td>
<td>Safari</td>
<td>WebKit</td>
<td>17</td>
<td>999, 1038, 1044, 1080, 1082, 1087, 1090, 1097, 1105, 1114, 1241, 1242, 1243, 1244, 1246, 1249, 1250</td>
</tr>
</tbody>
</table>

Some public fuzzing numbers

Google created OSS-Fuzz – Continuous Fuzzing for Open Source Software
https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html

Five months ago, we announced OSS-Fuzz, Google's effort to help make open source software more secure and stable. Since then, our robot army has been working hard at fuzzing, processing 10 trillion test inputs a day. Thanks to the efforts of the open source community who have integrated a total of 47 projects, we've found over 1,000 bugs (264 of which are potential security vulnerabilities).
Methods to extract coverage feedback
Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm ➔ AFL)
2. Emulation of binary (e.g. with qemu)
AFL qemu mode

american fuzzy lop 2.51b (chat)

- process timing
  run time: 0 days, 0 hrs, 0 min, 17 sec
  last new path: 0 days, 0 hrs, 0 min, 1 sec
  last uniq crash: none seen yet
  last uniq hang: none seen yet

- cycle progress
  now processing: 1 (5.00%)
  paths timed out: 0 (0.00%)

- stage progress
  now trying: havoc
  stage execs: 152/768 (19.79%)
  total execs: 33.3k
  exec speed: 1902/sec

- fuzzing strategy yields
  bit flips: 3/32, 1/30, 0/26
  byte flips: 0/4, 0/2, 0/0
  arithmetics: 0/0, 0/0, 0/0
  known ints: 0/22, 0/0, 0/0
  dictionary: 0/40, 0/50, 0/0
  havoc: 13/32.8k, 0/0
  trim: n/a, 0.00%

- overall results
  cycles done: 0
  total paths: 20
  uniq crashes: 0
  uniq hangs: 0

- map coverage
  map density: 0.09% / 0.30%
  count coverage: 1.27 bits/tuple

- findings in depth
  favored paths: 12 (60.00%)
  new edges on: 16 (80.00%)
  total crashes: 0 (0 unique)
  total tmouts: 0 (0 unique)

- path geometry
  levels: 2
  pending: 19
  pend fav: 12
  own finds: 19
  imported: n/a
  stability: 100.00%

[cpu:389%]
Topic: Lection 5 – AFL Qemu mode

Duration: 2-5 min

Description: Use Qemu mode to fuzz closed source binaries. Compare execution speed.
AFL Qemu mode

- Blogpost from 21-09-2018: Improving AFL’s qemu mode performance
  - From Andrea, a BSc student at University of Padova!
  - His AFL fork: https://github.com/abiondo/afl
  - Performance increase of 3x-4x times!

- Basic idea: AFL disables “block chaining” in QEMU to also trace direct jumps (with chaining it would not make the callback to log the block).

- Block chaining is important for performance, the patch from Andrea modifies the code in a way that “block chaining” can again be enabled and code gets inserted (without callbacks) ➔ Better performance

- Moreover he added code to “cache block chains” between forked childs!
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)
Disadvantage:

- It’s very slow
  - Statically setting breakpoints can speedup the process, but it’s still slow because of the debugger process switches
  - Only applicable if we remove a breakpoint after the first hit ➔ We only measure code-coverage (without a hit-count), edge-coverage not possible or extremely slow
- On-disk files are modified (statically), which can be detected with checksums (e.g. Adobe Reader .api files)
• **Advantage:**
  - Minset calculation
    - Detection if a new file has new code-coverage is very fast (native runtime) because we statically set breakpoints for unexplored code and run the application without a debugger
    - If it crashes we know it hit one of our breakpoints and therefore contains unexplored code
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, …)
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!
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 fuzzed results
Areas which influence fuzzing results
Areas which influence fuzzing results

Fuzzer speed

Fuzzer Results
Fuzzer Speed

1. Use a RAM Disk
2. Remove slow API calls
3. Fork Server (AFL’s Fork Server was designed by Jann Horn)
4. Deferred Fork Server
5. Persistent Mode (in-memory fuzzing)
6. Prevent process switches (between target application and the Fuzzer) by injecting the Fuzzer code into the target process
7. Modify the input in-memory instead of on-disk
Example to the Fork Server:

With Fork Server:

```
user@user-VirtualBox:~/tests AFL_NO_ARITH=1 AFL_PRELOAD=/home/user/test/libdislocator.so afl-fuzz -Q -x wordlist -l_input/ -o output/ -- ./chat

american fuzzy lop 2.51b (chat)

process timing
run time: 0 days, 0 hrs, 0 min, 17 sec
last new path: 0 days, 0 hrs, 0 min, 1 sec
last uniq crash: none seen yet
last uniq hang: none seen yet

overall results
cycles done: 0
total paths: 20
uniq crashes: 0
uniq hangs: 0

map coverage
map density: 0.09% / 0.30%
count coverage: 1.27 bits/tuple

findings in depth
favored paths: 12 (60.00%)
new edges on: 16 (80.00%)
total crashes: 0 (0 unique)
total tmouts: 0 (0 unique)

path geometry
levels: 2
pending: 19
pend fav: 12
own finds: 19
imported: n/a
stability: 100.00%

fuzzing strategy yields
bit flips: 3/32, 1/30, 0/26
byte flips: 0/4, 0/2, 0/0
arithmetics: 0/0, 0/0, 0/0
known ints: 0/22, 0/0, 0/0
dictionary: 0/40, 2/60, 0/0
havoc: 13/32.8k, 0/0
trim: n/a, 0.00%
```

Title: Finding security vulnerabilities with Fuzzing  |  Responsible: R. Freingruber  |  Version / Date: V1.0/2018-05  |  Confidentiality Class: public
© 2018 SEC Consult  |  All rights reserved
Example to the Fork Server:

Without Fork Server:

```
user@user-VirtualBox:/tests AFL_NO_FORKSRV=1 AFL_NO_ARITH=1 AFL_PRELOAD=/home/user/test/libdislocator.so afl-fuzz -x wordlist -Q -i input/ -o output/ ... ./chat

american fuzzy lop 2.51b (chat)

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>0 days, 0 hrs, 1 min, 0 sec</td>
<td>total paths: 12</td>
</tr>
<tr>
<td>last new path</td>
<td>uniq crashes: 0</td>
</tr>
<tr>
<td>0 days, 0 hrs, 0 min, 7 sec</td>
<td>uniq hangs: 0</td>
</tr>
<tr>
<td>last uniq crash</td>
<td></td>
</tr>
<tr>
<td>none seen yet</td>
<td></td>
</tr>
<tr>
<td>last uniq hang</td>
<td></td>
</tr>
<tr>
<td>none seen yet</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing</td>
<td>map density: 0.20% / 0.27%</td>
</tr>
<tr>
<td>0 (0.00%)</td>
<td></td>
</tr>
<tr>
<td>paths timed out</td>
<td>count coverage: 1.20 bits/tuple</td>
</tr>
<tr>
<td>0 (0.00%)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying</td>
<td>favored paths: 1 (8.33%)</td>
</tr>
<tr>
<td>havoc</td>
<td>new edges on: 10 (83.33%)</td>
</tr>
<tr>
<td></td>
<td>total crashes: 0 (0 unique)</td>
</tr>
<tr>
<td></td>
<td>total tmouts: 0 (0 unique)</td>
</tr>
<tr>
<td>stage execs: 6026/16.4k (36.78%)</td>
<td></td>
</tr>
<tr>
<td>total execs: 6244</td>
<td></td>
</tr>
<tr>
<td>exec speed: 103.4/sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips: 3/16, 1/15, 0/13</td>
<td>levels: 2</td>
</tr>
<tr>
<td>byte flips: 0/2, 0/1, 0/0</td>
<td>pending: 12</td>
</tr>
<tr>
<td>arithmetics: 0/0, 0/0, 0/0</td>
<td>pend fav: 1</td>
</tr>
<tr>
<td>known ints: 0/9, 0/0, 0/0</td>
<td>own finds: 11</td>
</tr>
<tr>
<td>dictionary: 0/20, 2/30, 0/0</td>
<td>imported: n/a</td>
</tr>
<tr>
<td>havoc: 0/0, 0/0</td>
<td>stability: 100.00%</td>
</tr>
<tr>
<td>trim: n/a, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>
```
Areas which influence fuzzing results

Fuzzer Results

Input filesize
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
Heat map of the memory dump (mimikatz access)
See below link for in-depth discussion how I fuzzed mimikatz with WinAFL:
• **Example:** Niklas B [@niklasb](https://twitter.com/niklasb) fuzzed map files in **Counter-Strike Global Offensive** and found lots of bugs/vulns with AFL Qemu mode!
  - [https://phoenhex.re/2018-08-26/csgo-fuzzing-bsp](https://phoenhex.re/2018-08-26/csgo-fuzzing-bsp)
  - You should definitely read the blog post!

• **Important decisions to mention**
  - He fuzzed the Linux binaries (with Qemu mode)
  - He fuzzed the server (command line) and not the 3D game client
  - He wrote a script to reduce input file size from 300 KB to 16 KB
  - Cite from the blog post: “Input file size matters a lot. By going down from 300KB to 16KB I gained at least a factor of 5 in performance. Probably even smaller would be even better.”
  - Initial runtime per iteration was 15+ seconds ➔ He wrote a custom wrapper which just calls the required functions ➔ ~50 exec / sec per thread
Title: The Art of Fuzzing | Responsible: R. Freingruber | Version / Date: V1.0/2017-11 | Confidentiality Class: public
© 2018 SEC Consult | All rights reserved

https://phoenhex.re/2018-08-26/csgo-fuzzing-bsp
Areas which influence fuzzing results
AFL Mutation

- **AFL performs deterministic, random, and dictionary based mutations**
  - AFL has a very good deterministic mutation algorithms

- **Deterministic mutation strategies:**
  - Bit flips
    - single, two, or four bits in a row
  - Byte flips
    - single, two, or four bytes in a row
  - Simple arithmetics
    - single, two, or four bytes
    - additions/subtractions in both endians performed
  - Known integers
    - overwrite values with interesting integers (-1, 256, 1024, etc.)
AFL Mutation

- **Random mutation strategies** performed for an input file after deterministic mutations are exhausted.

- Random mutation strategies:
  - Stacked tweaks
    - performs randomly multiple deterministic mutations
    - clone/remove part of file
  - Test case splicing
    - splices two distinct input files at random locations and joins them
• Consider Lection 6 with the following code:

```c
magicValue = *(uint64_t *)data;
if (magicValue == 0xdeadbeef13371337) {
    printf("Found magic value\n");
    *crash_ptr = 0xdeadbeef;
} else {
    printf("No magic value\n");
}
```

• Question: Can AFL identify this bug?
Practice: Lection 6

**Topic:** Lection 6 – Magic Values

**Duration:** 5-10 min

**Description:** See the impact of magic values in fuzzing.
• Circumventing Fuzzing Roadblocks with Compiler Transformation.
  • Enforce “Compiler Deoptimization” with LLVM compiler passes.
  • [URL](https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/)

```c
if (input == 0xabad1dea) {
    /* terribly buggy code */
} else {
    /* secure code */
}
```

```c
if (input >> 24 == 0xab)
    if ((input & 0xff0000) >> 16 == 0xad)
        if ((input & 0xff00) >> 8 == 0x1d)
            if ((input & 0xff) == 0xe9)
                /* terrible code */
                goto end;
}
```

/* good code */
end:
With **LD_PRELOAD** function implementations can be replaced for dynamically loaded libraries

- Just compile a library containing a function with the name of the target function which behavior you want to change
- LD_PRELOAD=/path/to/your/library.so ./target_application
- With AFL you can use AFL_PRELOAD=… afl-fuzz … -- ./target_application

Preeny contains other useful examples (especially for CTFs)

- [https://github.com/zardus/preeny](https://github.com/zardus/preeny)
- Defork: Remove fork()
- Desleep / Dealarm / Deptrace / Desrand: Often useful for CTFs
- Hint: Replace network function to read from files instead ➔ Fuzz it with AFL
• **LibTokenCap** memcmp example:

```c
#defines memcmp
int memcmp(const void* mem1, const void* mem2, size_t len) {
    if (!tokencap_is_ro(mem1)) tokencap_dump(mem1, len, 0);
    if (!tokencap_is_ro(mem2)) tokencap_dump(mem2, len, 0);

    while (len--)
        unsigned char c1 = *(const char*)mem1, c2 = *(const char*)mem2;
    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    mem1++; mem2++;

    return 0;
}
```

Log the token

Only read-only

Original memcmp implementation
**Practice: Lection 7**

**Topic:** Lection 7 – LibTokenCap

**Duration:** 5-10 min

**Description:** See LibTokenCap in action.
Task:
Go to lection 9 (skip lection 8 for the moment), copy the „chat“ binary and try to identify security vulnerabilities by playing with the binary.

Can you spot the vulnerability?
Please don’t read the solution file yet!
Did someone detect a crash in the binary?

What do you think: how many vulnerabilities are in this binary?

Other real world example: Heartbleed is a read buffer overflow and does not lead to a crash…

We (the Fuzzer) need a way to detect such flaws / vulnerabilities!
Heap Overflow Detection

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

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

Access attempt
• On Linux: **LibDislocator** (shipped with AFL)
  • Replaces the heap allocator to detect heap corruptions
  • Works also against closed source applications

• 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

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

/* We will also store buffer length and a canary below the actual buffer, so let's add 8 bytes for that. */
ret = mmap(NULL, \(1 + \text{PG\_COUNT}(\text{len} + 8)\) \* PAGE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);
if (ret == (void*)-1) {
    if (hard_fail) FATAL("mmap() failed on alloc (OOM?)");
    DEBUGF("mmap() failed on alloc (OOM?)");
    return NULL;
}
/* Set PROT\_NONE on the last page. */
if (mprotect(ret + \text{PG\_COUNT}(\text{len} + 8) \* PAGE\_SIZE, PAGE\_SIZE, PROT\_NONE)) FATAL("mprotect() failed when allocating memory");
**Topic:** Lection 8 – LibDislocator

**Duration:** 20 min

**Description:** See LibDislocator in action.
Libdislocator catches heap overflow

```bash
user@user-VirtualBox:~/test$ LD_PRELOAD=/home/user/test/libdislocator.so ./chat
Simple Chat Service
1 : Sign Up    2 : Sign In
0 : Exit
menu > 1
name > a
Success!
menu > 2
name > a
Hello, a!
Service Menu
1 : Show TimeLine    2 : Show DM    3 : Show UsersList
4 : Send PublicMessage 5 : Send DirectMessage
6 : Remove PublicMessage 7 : Change UserName
0 : Sign Out
menu >> 7
name >> abc
Speicherzugriffsfehler (Speicherabzug geschrieben)
```
• **Radamsa** is a very powerful input mutator
  • If you don’t want to write a mutator yourself, just use radamsa!
  • [https://github.com/aoh/radamsa](https://github.com/aoh/radamsa)
Radamsa

- **Problem of radamsa**: External program execution is slow (no library support)
  - Already submitted by others as issue: [https://github.com/aoh/radamsa/issues/28](https://github.com/aoh/radamsa/issues/28)

- **Example**: SECCON CTF fuzzer for the chat binary

- **Test 1**: Before every execution we mutate the input with a call to radamsa
  - **Result**: Execution speed is ~17 executions per second

- **Test 2**: Mutate input with python (no radamsa at all)
  - **Result**: Execution speed is ~740 executions per second

- ➔ Always create multiple output files (e.g.: 100 or 1000) or use IP:Port output
Radamsa

- **Testcases as input:**

  - test1.txt
    - register
    - user1
    - register
    - user2
    - login
    - user1
    - send_private_message
    - user2
    - Content of message
    - logout

  - test2.txt
    - register
    - user3
    - login
    - user3
    - delete_user

  - test3.txt
    - register
    - user4
    - login
    - user4
    - view_messages
    - logout
Radamsa

- Often seen wrong use of radamsa:

```bash
user-VirtualBox# ./radamsa test1.txt -o mutated1.txt
user-VirtualBox# ./radamsa test2.txt -o mutated2.txt
user-VirtualBox# ./radamsa test3.txt -o mutated3.txt
```

Possible output

```
register
user3
login
user3
delete user
```
Radamsa

- Correct invocation:

```
./radamsa test*.txt -n 1000 -o mutated%n.txt
```

Always generate multiple outputs (100 or 1000; 100 is recommended by radamsa)

Possible output

Combination of multiple input files!

However, merging of multiple input files is very unlikely ("send msg + delete user + view msg" will not be found within 2 hours)
• Correct selection of mutators (Example of the “chat” target):

```
user-VirtualBox% ./radamsa -l
Mutations (-m)
  ab: enhance silly issues in ASCII string data handling
  bd: drop a byte
  bf: flip one bit
  bi: insert a random byte
  br: repeat a byte
  bp: permute some bytes
  be: increment a byte by one
  bed: decrement a byte by one
  ber: swap a byte with a random one
  sr: repeat a sequence of bytes
  sd: delete a sequence of bytes
  ld: delete a line
  lds: delete many lines
  lr: duplicate a line
  li: copy a line closeby
  lr: repeat a line
```

```
ls: swap two lines
lp: swap order of lines
lis: insert a line from elsewhere
lrs: replace a line with one from elsewhere
td: delete a node
tr2: duplicate a node
ts1: swap one node with another one
ts2: swap two nodes pairwise
tr: repeat a path of the parse tree
uw: try to make a code point too wide
ui: insert funny unicode
num: try to modify a textual number
xp: try to parse XML and mutate it
ft: jump to a similar position in block
fn: likely clone data between similar positions
fo: fuse previously seen data elsewhere
nop: do nothing (debug/test)
```
Radamsa vs. Ni

- **Radamsa is written in Owl Lisp** (a functional dialect of Scheme)
  - Modifying the code is hard (at least for me because I don’t know Owl Lisp)
  - Currently no library support 😞 (➔ Slower than in-memory mutation)
  - Good mutation and gramma detection (~ 3500 lines)
  - Maintained

- **Ni is written in C**
  - Simple to modify, add to own project or compile as library (and it’s fast)
  - [https://github.com/aoh/ni](https://github.com/aoh/ni) (from the same guys)
  - Not as advanced as radamsa 😞 (~800 lines)
  - Not maintained: Last commit 2014

Ni can also merging multiple inputs

➔ Other inputs are only used during “random_block()” function
➔ Merging / Gramma detection not so advanced as with radamsa
The following table gives a speed comparison between different test setups for mutating data:

- **Numbers in the table are generated testcases per second**
- Table does not contain fuzzing or file read/write times (only generation of fuzz data)
- TC stands for number of test cases
- RD stands for RAM disk for files & programs
- Test program was a Python script
- Radamsa fast mode uses the following mutators:
  - -m bf,bd,bi,br,bp,bei,bed,ber,sr,sd
  - Taken from FAQ from [https://github.com/aoh/radamsa](https://github.com/aoh/radamsa)
## Speed comparison – input small text files

<table>
<thead>
<tr>
<th>Type of test</th>
<th>Radamsa ext.</th>
<th>Radamsa fast ext.</th>
<th>Ni ext.</th>
<th>Ni library (ctypes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input stdin (1 tc), output stdout (1 tc)</td>
<td>~265</td>
<td>~345</td>
<td>(no stdin support)</td>
<td>-</td>
</tr>
<tr>
<td>Input files (3 tc), output stdout (1 tc)</td>
<td>~255</td>
<td>~300</td>
<td>~775</td>
<td>-</td>
</tr>
<tr>
<td>Input files (3 tc), output via files (100 tc)</td>
<td>~1100</td>
<td>~1930</td>
<td>~7300</td>
<td>-</td>
</tr>
<tr>
<td>Input via files (3 tc), output via files (1000 tc)</td>
<td>~1100</td>
<td>~2150</td>
<td>~8350</td>
<td>-</td>
</tr>
<tr>
<td>Input files (3 tc), output via files (100 tc); RD</td>
<td>~1220</td>
<td>~2740</td>
<td>~7300</td>
<td>-</td>
</tr>
<tr>
<td>Input files (3 tc), output via files (1000 tc); RD</td>
<td>~1230</td>
<td>~3100</td>
<td>~8400</td>
<td>-</td>
</tr>
<tr>
<td>Input 3 samples, output one (all in-memory)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>~4000</td>
</tr>
</tbody>
</table>
The problem of the search space

The following input triggers the second Use-After-Free flaw in the chat binary:

Depth 1
- register user1
- register user2
- login user1

Depth 10
- send_private_message user2
- delete_user content
- login user2

Depth 13
- view_messages
The problem of the search space
• We need at least 7 distinct input-tokens to find the flaw (register, user1, user2, login, send_private_message, delete_user, view_message)
  • During real fuzzing we have way more inputs (all possible commands, special chars, long strings, special numbers, ….)
• After every input line we can again select one from the 7 possible input-tokens
• We have to find 13 input lines in the correct order to trigger the bug!
• For 13 input lines we have $7^{13} = 96\,889\,010\,407$ possibilities

➔ Runtime of the fuzzer to find this flaw?
➔ This is also a huge difference to file format fuzzing! File format fuzzing does not produce such huge search spaces, because “commands” can’t be sent at every node in the tree! (nodes have less children)
  ➔ AFL is not the best choice to fuzz such problems
We must reduce the search space!

- Initial Start-Sequence (Create Users) (This can be seen as our “input corpus”)
- Initial End-Sequence (Check public and private messages of all users)
- Encode the format into the fuzzer
  - Example: `send_message(username, random_string_msg)`
  - Peach Fuzzer
- But that was basically what we wanted to avoid (Fuzzer should work without modification)
- Instead of adding one command per iteration, add many commands (inputs)
  - Same when fuzzing web browsers ➔ Add thousands of html, svg, JavaScript, CSS, … lines to one test case and check for a crash
  - Important: Too many commands can create invalid inputs (e.g. invalid command ➔ Exit application)
- Additional feedback to “choose” promising entries (E.g.: prefer text output which was not seen yet, prefer fuzzer queue entries which often produce new output, …)
The problem of the search space

The following input triggers the second Use-After-Free flaw in the chat binary:

```
register user1
register user2
login user1
```

```
send_private_message user2
ccontent
delete_user
login user2
view_messages
```
**Topic:** Lection 9 – CTF Chat binary fuzzing

**Duration:** 5-10 min

**Description:** See how to fuzz a CTF binary.
Chat CTF Fuzzer

• Runtime to find the deep second UAF (Use-After-Free) vulnerability...

```
user@user-VirtualBox:~/test$ python fuzzer2.py
^Cueue: 528, runtime: 7 sec, execs: 2774, exec/sec: 357.80, crashes: 21 B0F [+], UAF1 [-], UAF2 [+]
User hit ctrl+c, stopping execution...
user@user-VirtualBox:~/test$ python fuzzer2.py
^Cueue: 8380, runtime: 141 sec, execs: 54058, exec/sec: 382.46, crashes: 255 B0F [+], UAF1 [-], UAF2 [+]
User hit ctrl+c, stopping execution...
user@user-VirtualBox:~/test$ python fuzzer2.py
^Cueue: 2732, runtime: 55 sec, execs: 18732, exec/sec: 339.05, crashes: 156 B0F [+], UAF1 [-], UAF2 [+]
User hit ctrl+c, stopping execution...
user@user-VirtualBox:~/test$ python fuzzer2.py
^Cueue: 8621, runtime: 166 sec, execs: 61845, exec/sec: 370.68, crashes: 351 B0F [+], UAF1 [-], UAF2 [+]
User hit ctrl+c, stopping execution...
```

• UAF1 was removed from patched binary because UAF1 would trigger before UAF2
• This fuzzer also works for any other CTF binary!!
Demo Time!

**Topic:** Mimikatz vs. GFlags & Application Verifier with PageHeap on Windows

**Runtime:** 3 min 15 sec

**Description:** See how to find bugs by just using the application and enabling the correct verifier settings.
• **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, …)

• If you don’t have source code: **DrMemory** (based on DynamoRio)

• **Use sanitizers during development**
  - You can also grab ASAN (address sanitizer) builds of firefox or chrome!

• I personally prefer heap libraries for fuzzing because they are faster but many people also use sanitizers for fuzzing.
**Practice: Lection 10**

**Topic:** Lection 10 – Sanitizers

**Duration:** 5-10 min

**Description:** See different sanitizers in action.
• Use ASAN / MSAN with AFL:
  • Cite from „notes_for_asan.txt“ from docs of AFL

„To compile with ASAN, set AFL_USE_ASAN=1 before calling 'make clean all'. The afl-gcc / afl-clang wrappers will pick that up and add the appropriate flags. Note that ASAN is incompatible with -static, so be mindful of that.

(You can also use AFL_USE_MSAN=1 to enable MSAN instead.)“
**Topic:** Lection 11 – DrMemory

**Duration:** 5 min

**Description:** See DrMemory in action.
Detecting not crashing vulnerabilities

- Change the heap implementation to check for dangling pointers AFTER a free() operation! (similar to MemGC)
  - Check all pointers in data section, heap and stack if they point into memory
  - Check must only be performed one time for new queue entries

Free() Detection here!

Use after free
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
...

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

Title: The Art of Fuzzing | Responsible: R. Freingruber | Version / Date: V1.0/2017-11 | Confidentiality Class: public
© 2018 SEC Consult | All rights reserved
Practice: Lection 12

**Topic:** Lection 12 – AFL-cov

**Duration:** 5 min

**Description:** See how to visualize AFL coverage.
LibFuzzer
LibFuzzer

- **LibFuzzer – Similar concept to AFL but in-memory fuzzing**
  - Requires LLVM SanitizerCoverage + writing small fuzzer-functions
  - LibFuzzer is more “a 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 also do in-memory fuzzing (persistent mode)

- **I highly recommend this tutorial:** [http://tutorial.libfuzzer.info](http://tutorial.libfuzzer.info)
- **And this workshop:** [https://github.com/Dor1s/libfuzzer-workshop](https://github.com/Dor1s/libfuzzer-workshop)
  - Our next labs are from this workshop!
Practice: Lection 13

**Topic:** Lection 13 – LibFuzzer simple example

**Duration:** 5 min

**Description:** Use LibFuzzer in an simple example.
Can you spot the vulnerability?

```c
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = s->s3->rec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload, padding = 16; /* Use minimum padding */
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
    /* snipped removed */
    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;
        /* Allocate memory for the response, size is 1 byte
        * message type, plus 2 bytes payload length, plus
        * payload, plus padding */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;
        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);
        r = dtls1_write_bytes(s, TLS1_RT_heartbeat, buffer, 1 + payload + padding);
        if (r >= 0 & & s->msg_callback)
            s->msg_callback(!, s->version, TLS1_RT_heartbeat,
                             buffer, 1 + payload + padding,
                             s, s->msg_callback arg);
```
Can you spot the vulnerability?

- This was Heartbleed from OpenSSL

Source: https://de.wikipedia.org/wiki/Heartbleed
Can you spot the vulnerability?

```c
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = t->s3->rrec.data[0], *pl;
    unsigned short hbtpe;
    unsigned int payload, padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtpe = *p++;
    n2s(p, payload);
    pl = p;
    /* snipped removed */
    if (hbtpe == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;
        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

        if (r >= 0) s->msg_callback(s->msg_callback(s, s->version, TLS1_RT_HEARTBEAT,
                                           buffer, 3 + payload + padding,
                                           s, s->msg_callback))
    }
}
```

Attacker controlled

Copies "payload" (user supplied) bytes from `pl (= p = ssl input data)` to "bp" (output buffer)

Size of "pl" is never checked!
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    static SSL_CTX *sctx = Init();
    SSL *server = SSL_new(sctx);
    BIO *sinbio = BIO_new(BIO_s_mem());
    BIO *soutbio = BIO_new(BIO_s_mem());
    SSL_set_bio(server, sinbio, soutbio);
    SSL_set_accept_state(server);
    BIO_write(sinbio, Data, Size);
    SSL_do_handshake(server);
    SSL_free(server);
    return 0;
}
**Practice: Lection 14**

**Topic:** Lection 14 – LibFuzzer Heartbleed.

**Duration:** 5 min

**Description:** Use LibFuzzer to find HeartBleed.
Practice: Lection 15

**Topic:** Lection 15 – LibFuzzer C-ares

**Duration:** 5 min

**Description:** Use LibFuzzer to find a bug in C-ares.
**Practice: Lection 16**

**Topic:** Lection 16 – LibFuzzer Woff

**Duration:** 5 min

**Description:** Use LibFuzzer to find a bug in Woff.
DynamoRio
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 trigger at specific events (new basic block / instruction which is moved into code cache, load of module, exit of process, …)
Dynamic Instrumentation Frameworks

- **For transformation time callbacks can be registered**
  - Called only once if new code gets executed
  - `drmgr_register_bb_instrumentation_event()`

- **For execution time we have two possibilities**
  - Called every time the code is executed
  - Clean calls: save full context (registers) and call a C function (slow)
  - Inject assembler instructions (fast)
    - Context not saved, tool writer must take care himself
    - Registers can be “spilled” (can be used by own instructions without losing old state)
    - DynamoRio takes care of selecting good registers & saving and restoring them

- **Nudges can be send to the process & callbacks can react on them**
  - Example: Turn logging on after the application started
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
**Topic:** Instrumentation of Adobe Reader with DynamoRio

**Runtime:** 2 min 31 sec

**Description:** Use DynamoRio to extract code-coverage of a closed-source application using only a simple command.
Topic: Determine Adobe Reader “PDF loaded” breakpoint with coverage analysis.

Runtime: 1 min 08 sec

Description: Log coverage of “PDF open” action to get a breakpoint address to detect end of PDF loading.
Practice: Lection 17

**Topic:** Lection 17 – DrCov

**Duration:** 1 min

**Description:** Use DrCov to extract coverage information.
Practice: Lection 18

**Topic:** Lection 18 – Log function calls.

**Duration:** 5 min

**Description:** Read a sample DynamoRio client code and execute it.
Practice: Lection 19

**Topic:** Lection 19 – Write a DynamoRio client to log all cmp values.

**Duration:** 15 min

**Description:** Log cmp values and use them for fuzzing.
Windows Fuzzing
WinAFL

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

- Two modes:
  - DynamoRio: Source code not required
    - Can be used to modify closed-source applications at runtime ➔ Our focus today!
  - Syzygy: Source code required

- ➔ WinAFL uses in-memory fuzzing and we therefore **must specify a target function** which should be fuzzed

```
 a8-fuzz.exe -i in -o out -D C:\work\winafl\DynamoRIO\bin64 -t 20000 --
 -coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll
 -fuzz_iterations 5000 -target_module test_gdiplus.exe
 -target_offset 0x1270 -nargs 2 -- test_gdiplus.exe @@
```
Demo Time!

**Topic:** Fuzzing mimikatz on Windows with WinAFL

**Runtime:** 10 min 39 sec

**Description:** See the WinAFL fuzzing process on Windows of binaries with source-code available in action.
Fuzzing and exploiting mimikatz

```
C:\Users\normalUser\Desktop\test_mimikatz\real_mimikatz>mimikatz.exe

mimikatz 2.1.1 (x86) built on Aug 13 2017 17:27:38

Benjamin DELPY 'gentilkiwi' ( benjamin@gentilkiwi.com )
http://blog.gentilkiwi.com/mimikatz

with 21 modules * * */

mimikatz # seckurlsa::minidump exploit.dmp
Switch to MINIDUMP : 'exploit.dmp'

mimikatz # seckurlsa::logonpasswords
Opening : 'exploit.dmp' file for minidump...
```
AutoIt v3 is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting. It uses a combination of simulated keystrokes, mouse movement and window/control manipulation in order to automate tasks ...
```autoit
#include <AutoItConstants.au3>

Run("notepad.exe")
Local $hWnd = WinWait("[CLASS:Notepad]", "", 10)
ControlSend($hWnd, "", "Edit1", "Hello World")
WinClose($hWnd)
ControlClick("[CLASS:#32770]", "", "Button3")
WinSetState("[CLASS:Notepad]", "", @SW_MAXIMIZE)
MouseMove(14,31)
MouseClick($MOUSE_CLICK_LEFT)
MouseMove(85,209)
MouseClick($MOUSE_CLICK_LEFT)
ControlClick("[CLASS:#32770]", "", "Button2")
```
Practice: Lection 20

**Topic:** Lection 20 - AutoIt

**Runtime:** 5 min

**Description:** Use AutoIt to automate some GUI.
Demo Time!

**Topic:** Real-world EnCase Imager Fuzzing (Vulnerability found by SEC Consult employee Wolfgang Ettlinger)

**Runtime:** 29 sec

**Description:** See real-world fuzzing in action.
Exploitability of the vulnerability
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
```
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?
GUI automation

- HashCalc.exe MD5 fuzzing
- GUI automation with AutoIt: \( \sim 3 \text{ exec / sec} \)
- In-Memory with debugger: \( \sim 750 \text{ exec / sec} \)
- In-Memory with DynamoRio (no instr.): \( \sim 170\,000 \text{ exec / sec} \)
Reverse Engineering Tricks for Fuzzing
How to find the target function

• Important task when fuzzing with WinAFL ➔ Find a potential target function to fuzz!
  • How can we do this (as fast as possible) if source code is not available?
  • This function must open, process and close the input file!

• Technique 1: Log CreateFile() and CloseFile()
  • Simple solution: On 32-bit we can use Process Monitor and it’s stack traces
  • API Monitor is another option
  • DynamoRio / PIN script
How to find the target function

- **Example**: We now use our PE corpus to fuzz dumpbin from Visual Studio
  - Dumpbin is internally just a wrapper to link.exe

```
C:\test_pe>link.exe /DUMP /ALL /NOLOGO calc.exe

Dump of file calc.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
  8664 machine (x64)
  6 number of sections
  82E1734B time date stamp
  0 file pointer to symbol table
  0 number of symbols
```
How to find the target function

- **Process Monitor** to find CreateFile and CloseFile
How to find the target function

- **Result:**

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Process Name</th>
<th>PID</th>
<th>Operation</th>
<th>Path</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CreateFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CloseFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CreateFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CloseFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CreateFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>18:12...</td>
<td>link.exe</td>
<td>3272</td>
<td>CloseFile</td>
<td>C:\test_pe\calc.exe</td>
<td>SUCCESS</td>
</tr>
</tbody>
</table>
```
How to find the target function

- Second `CreateFile` looks good ➔ `CreateFileW`
And the call stack of the `CloseFile()`:
How to find the target function

- **Technique 2: Memory breakpoints on input data**
  - E.g.: fuzzing network packets or from stdin
  - Set the breakpoint at `recv()` and when it triggers we are in the code which works with the data

- **Technique 3: Log every (internal) function call together with arguments**
  - After that search the log file for your input
  - Can be implemented with DynamoRio / PIN (funcap is a debugger implementation for this but it’s extremely slow)
  - Hint: Log only cross-module calls to find easy-to-target exported library functions
• **Technique 4: Measure code coverage and subtract them**
  • Execute program twice, one time trigger the target behavior, one time not
  • Extract both times code coverage (drrun -t drcov -- C:\applications.exe arg1 arg2)
  • Use IDA Pro plugin lighting house to subtract the coverage ➔ Result is code responsible for the target behavior
  • Side note: You can also extract coverage from your input corpus and use lighting house to detect code which you currently don’t reach!

• **Technique 5: Use a taint engine to follow inputs**
  • More on this later!
**Topic:** Identification of target function address of a closed-source application (HashCalc).

**Runtime:** 10 min 15 sec

**Description:** Using reverse engineering (breakpoints on function level via funcap and DynamoRio with LightHouse) to identify the target function address.
Demo Time!

**Topic:** In-memory fuzzing of HashCalc using a debugger.

**Runtime:** 4 min 21 sec

**Description:** Using the identified addresses and WinAppDbg we can write an in-memory fuzzer to increase the fuzzing speed to 750 exec / sec!
Demo Time!

**Topic:** In-memory fuzzing of HashCalc using DynamoRio.

**Runtime:** 2 min 58 sec

**Description:** Using the identified addresses and DynamoRio we can write an in-memory fuzzer to increase the fuzzing speed to 170 000 exec / sec!
Practice: Lection 21

**Topic:** Lection 21 - WinAFL GDI Fuzzing

**Runtime:** 10 min

**Description:** Using WinAFL to fuzz GDI.
GDI Fuzzing with WinAFL

• Use the recompiled GDI.exe binary (with /MT) / Install VC Redist 2010

• Debug:

```
C:\Windows\system32\cmd.exe
C:\WinAFL\bin64>C:\DynamoRIO\bin64\drrun.exe -c winafl.dll -debug -target_module GDI.exe -target_offset 0x1180 -fuzz_iterations 10 -nargs 2 -- GDI.exe input.bmp
C:\WinAFL\bin64>
```

• Fuzzing:

```
C:\WinAFL\bin64>afl-fuzz.exe -i in -o out -D C:\DynamoRIO\bin64 -t 20000 -- -coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll -fuzz_iterations 5000 -target_module GDI.exe -target_offset 0x1180 -nargs 2 -- GDI.exe @`
```
Browser fuzzing
• The discussed techniques should **NOT** be used to fuzz browsers (DOM, JS, CSS, SVG, …)

• Reason: When fuzzing a browser you want to find a combination of JS / HTML / … code which leads to a vulnerability. This is not a binary file format and therefore the discussed techniques are inefficient
  • However: If the browser parses an image, icon, audio, font, … file you can use the techniques!
  • It’s similar to the “chat” binary where AFL-style fuzzing was also inefficient
Browser Fuzzing

• For browsers you generate HTML / JS files with a grammar-based fuzzer.

• **Example:** Domato from Ivan Fratric (Google Project Zero)

```bash
workshop@workshop-VM:~/Desktop/Lections/software/domato$ python generator.py --output_dir output --no_of_files 5
Running on ClusterFuzz
Output directory: output
Number of samples: 5
Writing a sample to output/fuzz-0.html
Writing a sample to output/fuzz-1.html
Writing a sample to output/fuzz-2.html
Writing a sample to output/fuzz-3.html
Writing a sample to output/fuzz-4.html

$ ~/Desktop/firefox_asan/firefox output/fuzz-1.html
```
Practice: Lection 22

**Topic:** Lection 22 - Domato

**Runtime:** 5 min

**Description:** Use Domato to generate fuzzed HTML files.
Browser Fuzzing

- Google P0 fuzzed major browsers in 2017
  - 100 000 000 iterations per browser with 10 seconds per run
  - Chrome, Firefox and Safari with ASAN builds
  - IE and Edge with Page Heap
  - Cost: Approximately 1000 $
## Browser Fuzzing

### Results:

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Browser</th>
<th>Engine</th>
<th>Number of Bugs</th>
<th>Project Zero Bug IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Chrome</td>
<td>Blink</td>
<td>2</td>
<td>994, 1024</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Firefox</td>
<td>Gecko</td>
<td>4**</td>
<td>1130, 1155, 1160, 1185</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Internet Explorer</td>
<td>Trident</td>
<td>4</td>
<td>1011, 1076, 1118, 1233</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Edge</td>
<td>EdgeHtml</td>
<td>6</td>
<td>1011, 1254, 1255, 1264, 1301, 1309</td>
</tr>
<tr>
<td>Apple</td>
<td>Safari</td>
<td>WebKit</td>
<td>17</td>
<td>999, 1038, 1044, 1080, 1082, 1087, 1090, 1097, 1105, 1114, 1241, 1242, 1243, 1244, 1246, 1249, 1250</td>
</tr>
</tbody>
</table>
• On the Windows VM you can find the exploit for CVE-2011-2371 (Firefox Integer Overflow – Array.reduceRight)

• A very good exploit to get started with browser exploitation

• Simple compared to other exploits, works very reliable on all Windows operating systems

• Bypasses ASLR & DEP

• Does not crash the browser
My research
On the shoulders of giants

- My research builds on-top of the hard work other researchers shared with the public. Without their awesome work all of my stuff would definitely not work!

- **AFL by Michal Zalewski**
  - Simplest fuzzer to start on Linux and very efficient in finding bugs

- **LibFuzzer from LLVM**
  - Use it when you have C/C++ code and can compile with clang (faster than AFL)

- **Honggfuzz by Robert Swiecki**
  - Simpler to modify than AFL, useful when fuzzing network apps

- **WinAFL by Ivan Fratric**
  - Use it if you want to fuzz Windows software with feedback (alternative: If you don’t want to use coverage feedback you can try CERT Fuzzer, but I definitely recommend feedback-based fuzzing)
  - Based on DynamoRio by Derek Bruening
I encountered some problems when fuzzing with WinAFL:

- **Lack of a snapshot mechanism** ➔ Just jump at the end of the function back to the start without resetting the old memory state
  - Heap, Stack, global variables, TEB and PEB may change… some data (e.g.: network packets) may only be available in first iteration, access permission can change, file positions can change or be closed, locks, semaphores, critical sections, multi-threaded applications, ….
  - Lots of stuff can go wrong here
Example:

```c
target_fuzz_function(char *input, size_t len) {
    crypto_func(input, len);  // works with g_var1 & g_var2
    free(g_var1);
    g_var2->field123 = 567;
}
```

› Second iteration works with the freed variable `g_var1` (and modified `g_var2` content)

› We could tell the fuzzer to fuzz only “crypto_func”. But what if this function was inlined? What if crypto_func also works with `g_var3`? If we assume closed source applications and the target function is very big, it’s hard / time consuming to manually find these dependencies! What if the compiler changed the order?
Example: If we start to fuzz link.exe with WinAFL with the identified address we see in the log file:

```
48 Module loaded, ole32.dll
49 Module loaded, CRYPTBASE.dll
50 In OpenFileW, reading NUL
51 In pre_fuzz_handler
52 In OpenFileW, reading C:\test_pe\calc.exe
53 In post_fuzz_handler
54 In pre_fuzz_handler
55 Exception caught: c0000005
56 crashed
57 Everything appears to be running normally.
58 Coverage map follows:
59 NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL NUL
```
I encountered some problems when fuzzing with WinAFL:

• Not 100% compatible with Page Heap because of in-memory fuzzing
  • If data is not freed during the iteration, some checks are never performed!
  • If a global variable is freed during the iteration, we introduce a double-free!
  • If memory is allocated in the iteration, but not freed, we spray the heap which means we have to restart the application after some thousand iterations
• **Undetected by page heap:**

```c
Uint8_t *data = static_cast<uint8_t *>(Malloc(17));
data[-1] = 0x11; // no crash
data[17] = 0x11; // no crash
data[30] = 0x11; // no crash
// No free(data1)
```

• **Windows requires 16-byte aligned heap pointers (on x64), therefore it can only use fill patterns to detect 1 to 15 byte overflows (or negative ones); Fill patterns are just checked at free()**
  - In-memory fuzzing often doesn’t reach the free because we go to the next iteration…
  - Same applies for other heap allocation routines (new, RtlAllocateHeap, …)
• Undetected by page heap:

```c
uint8_t *data1 = static_cast<uint8_t*>(malloc(16));
printf("Data: %p\n", data1);

HANDLE hFile = CreateFileA("test.tmp", GENERIC_READ, FILE_SHARE_READ, NULL,
  if (hFile == INVALID_HANDLE_VALUE) {... }
  DWORD readBytes;
  BOOL ret = ReadFile(hFile, data1, 20, &readBytes, NULL);
  if (ret == FALSE) {
    cerr << "ERROR ReadFile" << endl;
    cerr << "CODE: " << GetLastErrorAsString() << endl;
    return -1;
  }
```
Execution with page heap enabled:

```
C:\Users\rfm\Documents\Visual Studio 2017\Projects\HeapTestings\x64\Release>HeapTestings.exe
Data: 000001ABF52E1FF0
ERROR ReadFile
CODE: Invalid access to memory location.
```

Inside WinDbg:
```
0:000> g
ModLoad: 00007ffa`5aa70000 00007ffa`5aa81000 C:\WINDOWS\System32\kernel.appcore.dll
ModLoad: 00007ffa`5c760000 00007ffa`5c7fd000 C:\WINDOWS\System32\msvcrt.dll
ModLoad: 00007ffa`5c640000 00007ffa`5c75f000 C:\WINDOWS\System32\RPCRT4.dll
ntdll!NtTerminateProcess+0x14:
00007ffa`5e7303f4 c3 ret
```

We don’t see an exception although page heap is enabled!
Our fuzzer would also miss it!
Feedback based fuzzing

- AFL injects here

- WinAFL injects here

- Where it makes sense
So I started to develop my own fuzzer to solve the problems

1. **Full logic is injected into the target application** ➔ No inter-process communication required, mutations are performed in-memory, file-reads are cached, full multi-core support

2. **Snapshot mechanism** which creates process snapshots and can quickly (!) restore the snapshot (this works with my own heap implementation which doesn’t have the problems of page heap)

3. **Taint Engine** to reduce number of bytes which must be fuzzed

4. **Playing around with new ideas from academic papers** ➔ I try to implement and test all of them
Taint Analysis

• **Taint Analysis:**
  - With a PIN / DynamoRio tool follow data-flow by tainting memory
    - Assign one bit to every byte in RAM, 0… not tainted, 1 … tainted
    - Store per tainted byte extra information (e.g. on which input bytes it depends)
    - Move taint status around with every instruction (e.g. mov rax, [memory] ➔ If [memory] is tainted rax will also be tainted; xor rax, rax ➔ Rax will be untainted) by injecting code with DynamoRio

• **Taint Analysis Tools:**
  - Libdft, Triton, bap, panda, manticore, Own DynamoRio client, …
We can write a DynamoRio/PIN tool which tracks calls and taint status.

Automatically detect target fuzz function

func1 ➔ Target Function to fuzz ➔ _start ➔ func4 ➔ func1

Open input file

func5 ➔ access ➔ func6 ➔ access ➔ func7 ➔ access ➔ func8 ➔ access

Close input file
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!

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>20 Mutations</th>
<th>Instruction X1: Read byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 2</td>
<td>50 Mutations</td>
<td>Instruction X2: Read byte 1,2,3,4</td>
</tr>
<tr>
<td>Byte 3</td>
<td>20 Mutations</td>
<td>Instruction X3: Read byte 2</td>
</tr>
<tr>
<td>Byte 4</td>
<td>10 Mutations</td>
<td>Instruction X4: Read byte 1,2</td>
</tr>
<tr>
<td>Byte 5</td>
<td>0 Mutations</td>
<td>Instruction X5: Read byte 2,3</td>
</tr>
</tbody>
</table>

- Byte 1 read by 2 instructions: 2/10 = 20%
- Byte 2 read by 5 instructions: 5/10 = 50%
- Byte 3 read by 2 instructions: 2/10 = 20%
- Byte 4 read by 1 instruction: 1/10 = 10%
- Byte 5 read by 0 instructions: 0/10 = 0%

Maybe don't fuzz this at all X2 is maybe a copy / search function
Fuzzing vs. Symbolic execution

➔ Fuzzing all bytes:

```
mov    al, [edi+0Ah]
sub    al, 30h ; '0'
xor    al, 0AAh
cmp    al, 11h
jnz    short loc_4010A1
```

➔ Input file is for example 1000 Byte (1 KB)
➔ 256 possible Byte values for 1000 Byte ➔ 256 000 potential executions (in our case ~2600)
Fuzzing vs. Symbolic execution

Symbolic execution (simplified):

Two branches, one with value $= 0x11$ and one with value $\neq 0x11$

Solution for $= 0x11$: $\text{user\_input}[0x0a] := (0x11 \oplus 0xaa) + 0x30 = 0xeb$
Fuzzing vs. Symbolic execution

➔ Taint Analysis in Fuzzing

```
mov    al, [edi+0Ah]
sub    al, 30h ; '0'
xor    al, 0AAh
cmp    al, 11h
jnz    short loc_4010A1
```

- Taint al because [edi+0x0a] depends on input[0x0a]
- This jnz depends on tainted byte input[0x0a]
- All our inputs take this path!

```
push   offset special_case ; Bug
jmp    short loc_4010A6
```
```
loc_4010A1: ; No bug
push   offset default_case
```

➔ Query for conditional jumps (dep. On our input), where all inputs take the same path

➔ Taint Engine returns input byte 0x0a ➔ Just fuzz this byte!

➔ Check the cmp operand size ➔ If it’s 1 or 2 bytes use fuzzing, if it’s 4 or 8 b. use symbolic execution
Demo Time!

- Demo: SEC Consult Fuzzer
- Fuzzer is still early alpha!
  - Release: In some months
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, feedback or grammar)
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!
Where to get more information

- Reddit:
  - [https://www.reddit.com/r/fuzzing/](https://www.reddit.com/r/fuzzing/)
  - Most fuzzing related blog posts are published here

- Rode0day:
  - [https://rode0day.mit.edu/](https://rode0day.mit.edu/)
  - A continuous bug finding competition

- DARPA Challenge set for Linux/Windows/MacOS
  - [https://github.com/trailofbits/cb-multios](https://github.com/trailofbits/cb-multios)
Thank you for your attention!

For any further questions contact me

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