The Art of Fuzzing

Introduction

- René Freingruber (r.freingruber@sec-consult.com)
  - Twitter: @ReneFreingruber
  - Security Consultant at SEC Consult
    - Reverse Engineering, Exploit development, Fuzzing
  - Trainer at SEC Consult
    - Secure Coding in C/C++, Reverse Engineering
    - Red Teaming, Windows Security
  - Speaker at conferences:
    - CanSecWest, DeepSec, 31C3, Hacktivity, BSides Vienna, Ruxcon, ToorCon, NorthSec, IT-SeCX, QuBit, DSS ITSEC, ZeroNights, Owasp Chapter, …
    - Topics: EMET, Application Whitelisting, Hacking Kerio Firewalls, Fuzzing Mimikatz, …

SEC Consult

ADVISOR FOR YOUR INFORMATION SECURITY

We are hiring!

Founded **2002**

Leading in IT-Security Services and Consulting

Strong customer base in Europe and Asia

70+ Security experts

400+ Security audits per year

Fuzzing

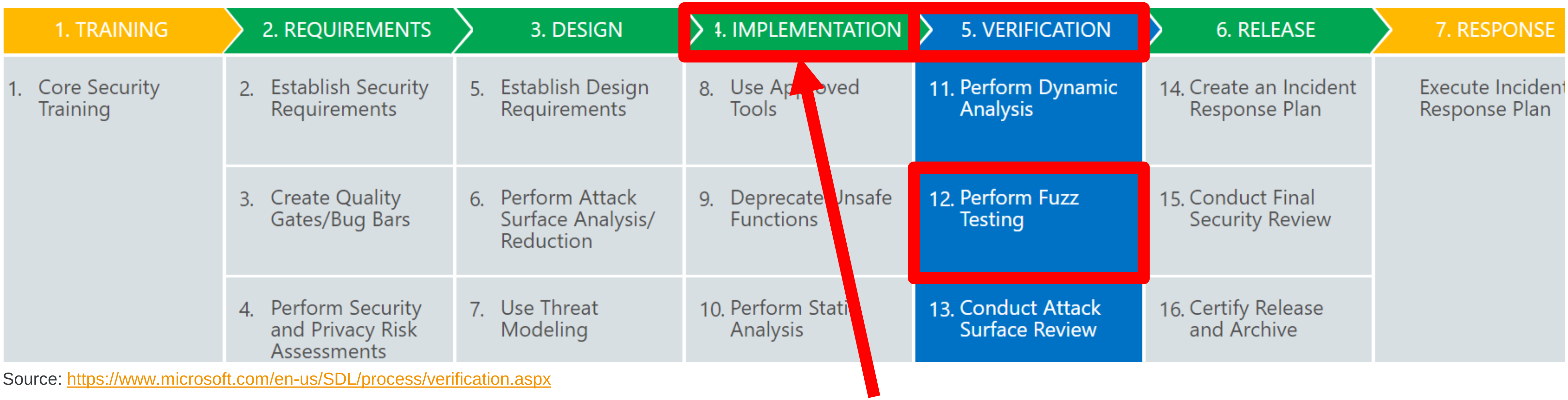
Fuzzing

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



Source: https://www.microsoft.com/en-us/SDL/process/verification.aspx

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

Why do we need Fuzzing?

SDL Phase 4 *Security Requirements*

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.**

Source: https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.aspx

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Fuzzing

- **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

Successful Fuzzing Examples

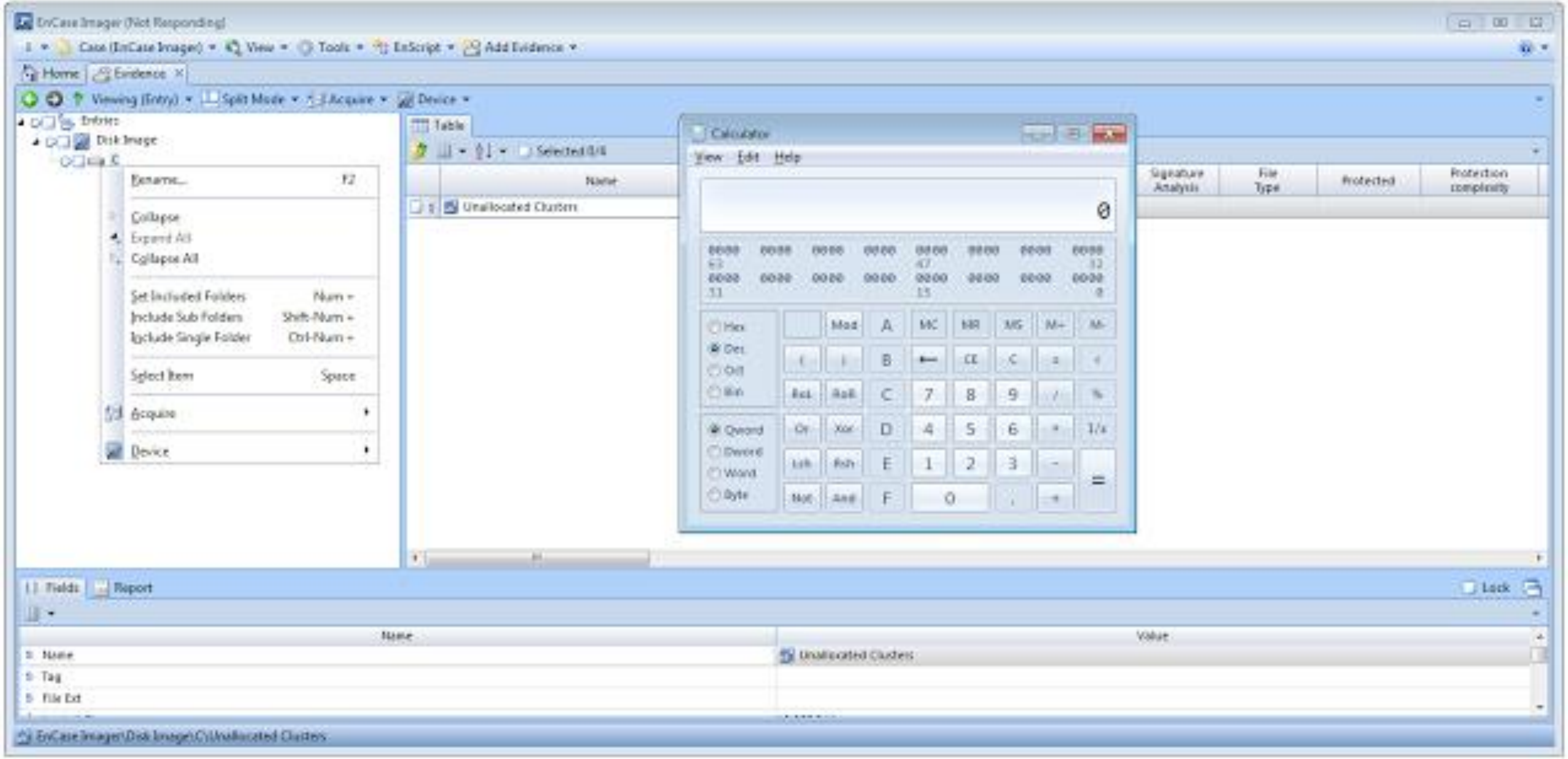
Demo Time!

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



AutoIt

- **AutoIt definition** (https://www.autoitscript.com):

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 …

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

AutoIt Demo Source Code

```
#include <AutoItConstants.au3>

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


SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

AutoIt

- **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)

```
#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
```

Demo Time!

Topic: CS GO minimize crash

Runtime: 2 min 16 sec

Description: See real-world example in action.

Recap

- **Such straight-forward fuzzing is very often very successful!**
- **Example success stories:**
    - Encase http://blog.sec-consult.com/2017/05/chainsaw-of-custody-manipulating.html
    - Counterstrike https://hernan.de/blog/2017/07/07/lock-and-load-exploiting-counter-strike-via-bsp-map-files/
    - Many others!
- But can we do better?
- What problems do you see in such fuzzing approaches?
    - GUI automation is very slow
    - Documentation and Specs must be read to write the fuzzer ➔ Time consuming task!

Feedback-based Fuzzing

Feedback based fuzzing

- **Problem:** We need to read the specification / documentation to write the fuzzer
- **Solution:** Use **feedback** from the application
- What do you think is useful feedback?

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{<empty>}

Random fuzzer action:
Queue is empty, so create a random input

Full input:
foobar

Full output:
Please enter some command

➔ New output, store the associated input in fuzzing queue (A)

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A}

Random fuzzer action:
Take A and modify it (uppercase)

Full input:
FOOBAR

Full output:
Please enter some command

➔ Output already known, so don‘t add input to Queue

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A}

Random fuzzer action:
Take A and modify it (replace it)

Full input:
command

Full output:
Please enter some command
You entered command!

➔ New output, so add input to queue (as B)

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A,B}

Random fuzzer action:
Take B and append random value

Full input:
Command
123

Full output:
Please enter some command
You entered command!

➔ Output already known, do nothing

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A,B}

Random fuzzer action:
Take A and append random value

Full input:
foobar
123

Full output:
Please enter some command

➔ Output already known, do nothing

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A,B}

Random fuzzer action:
Take B and append random value

Full input:
command
subcommand

Full output:
Please enter some command
You entered command!
You entered subcommand!

➔ New output, store input as C

Feedback based fuzzing

Consider this pseudocode:

```
printf("Please enter some command\n");
if(read_line_from_user () == "command") {
    printf("You entered command!\n");
    if(read_line_from_user() == "subcommand") {
        printf("You entered subcommand!\n");
        if(read_line_from_user() == "trigger") {
            printf("You entered trigger!\n");
            //buffer_overflow here
        }
    }
}
```

Fuzzing Queue:
{A,B,C}

Random fuzzer action:
Take C and append random value

Full input:
Command
subcommand
trigger

Full output:
Please enter some command
You entered command!
You entered subcommand!
You entered trigger!

➔ **Crash**

Feedback based fuzzing

- **I was often successful with feedback based on text-output**

- **Example:**
  - SECCON 2016 CTF – Chat binary ; nearly all CTF binaries
  - Embedded hardware admin console (text-based applications)

- **Pro:**
  - Very simple & fast to implement
  - Normal application runtime during fuzzing (no performance lose)
- **Con:**
  - Not always applicable (application does not give output messages)
  - If two different behaviors do not result in different output it‘s useless

Hints for output based fuzzing:

1. Remove default output "unknown command"
   - Prevents filling the fuzzing queue with useless commands

2. Removing user-reflected output can sometimes help
   - Example: "login MyUser1" => Output: "Hello MyUser1"

   ➔ Two different users will have "Hello MyUser1" and "Hello MyUser2" ➔ Two entries in the fuzzing queue (depends on situation if we want this or not)

   ➔ Solution: Hook fprintf (and others) to just print the format string ("Hello %s\n")

Feedback based fuzzing

Hooking fprintf:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <stdarg.h>
// gcc -shared -D_FORTIFY_SOURCE=2 -g -Wl,--no-as-needed -ldl -fPIC -Wall output.c -o output.so
static FILE *log = NULL;
int fprintf(FILE *stream, const char *format, ... ) {
  static int (*original_fprintf)(FILE *s,const char *f,...) = NULL;
  if(original_fprintf == NULL) {
    original_fprintf = (int (*)(FILE *s,const char *f,...))dlsym(RTLD_NEXT, "fprintf");
    log = fopen("/tmp/original_output.log", "w+");
  }
  // trigger original behavior (with possible flaws)
  va_list args; va_start(args, format); vfprintf(log, format, args); va_end(args);
  return original_fprintf(stream, "%s", format); // Print only format string
}
__attribute__((destructor)) void end(void) {
        if(log != NULL) { fclose(log); log = NULL; }
}
```

Feedback based fuzzing

Without

```
user@user-VirtualBox:~/test$ ./chat
Simple Chat Service

1 : Sign Up     2 : Sign In
0 : Exit
menu > 1
name > USER
Success!

1 : Sign Up     2 : Sign In
0 : Exit
menu > 2
name > USER
Hello, USER!
Success!

Service Menu
```

With

```
user@user-VirtualBox:~/test$ LD_PRELOAD=$(pwd)/output.so ./chat
Simple Chat Service

1 : Sign Up     2 : Sign In
0 : Exit
menu > 1
name > USER
Success!

1 : Sign Up     2 : Sign In
0 : Exit
menu > 2
name > USER
Hello, %s!
Success!

Service Menu
```

Feedback based fuzzing

- LD_PRELOAD and similar techniques can be used **to redirect network traffic to files for fuzzing**
  - Many fuzzers only support input via files or stdin (and not network packets)
  - Check: https://github.com/zardus/preeny
    - But it’s error prone
    - Maybe a better alternative: https://github.com/jdbirdwell/afl
- We can also **change the heap implementation** and other interesting functions…. But more to this later
- On Windows use Detours or Dynamic Instrumentation Frameworks (see later)

Demo Time!

Topic: Find the flaw(s) in SECCON CTF binary

Runtime: 1 min 16 sec

Description: Try to find the flaw(s) which are triggered during execution.

Feedback based fuzzing

➔ **Now consider this pseudocode**

```
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```

Feedback based fuzzing

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

```
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```

Feedback based fuzzing

➔ **Same for „command\nsubcommand\n“**

```
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```

Feedback based fuzzing

➔ **And so on…**

```
if(read_line_from_user () == "command") {
    if(read_line_from_user() == "subcommand") {
        if(read_line_from_user() == "trigger") {
            //buffer_overflow here
        }
    }
}
```


SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Methods to measure code-coverage

1. Instrumentation during compilation (source code available; gcc or llvm → AFL)

American Fuzzy Lop - AFL

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

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

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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Feedback based fuzzing

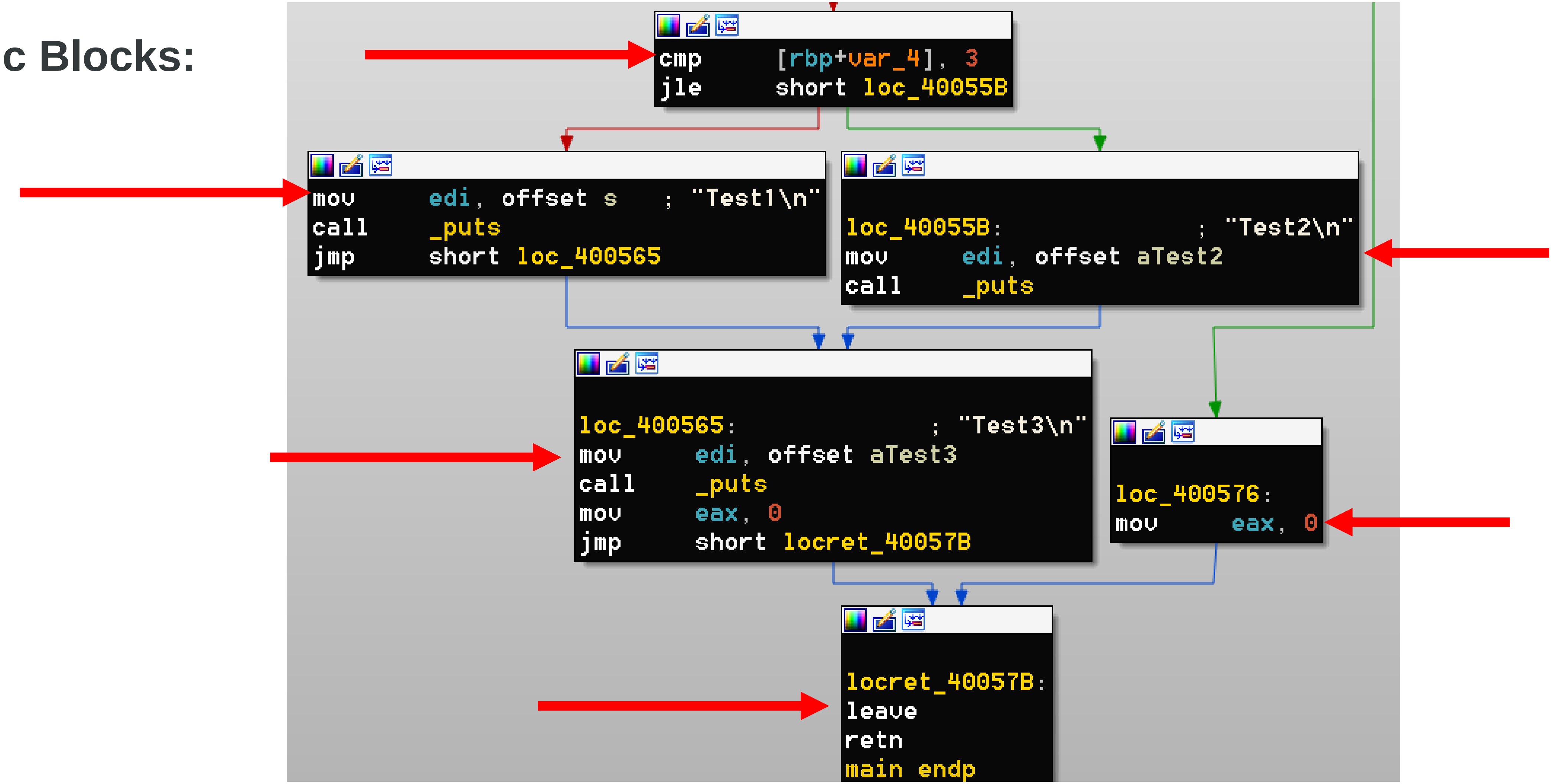
- **Consider this code (x = argc):**

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

```
user-VirtualBox# gcc -o test test.c
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…**

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

Test3

user-VirtualBox# ./test2 1 2 3 4 5
Test1

Test3
```

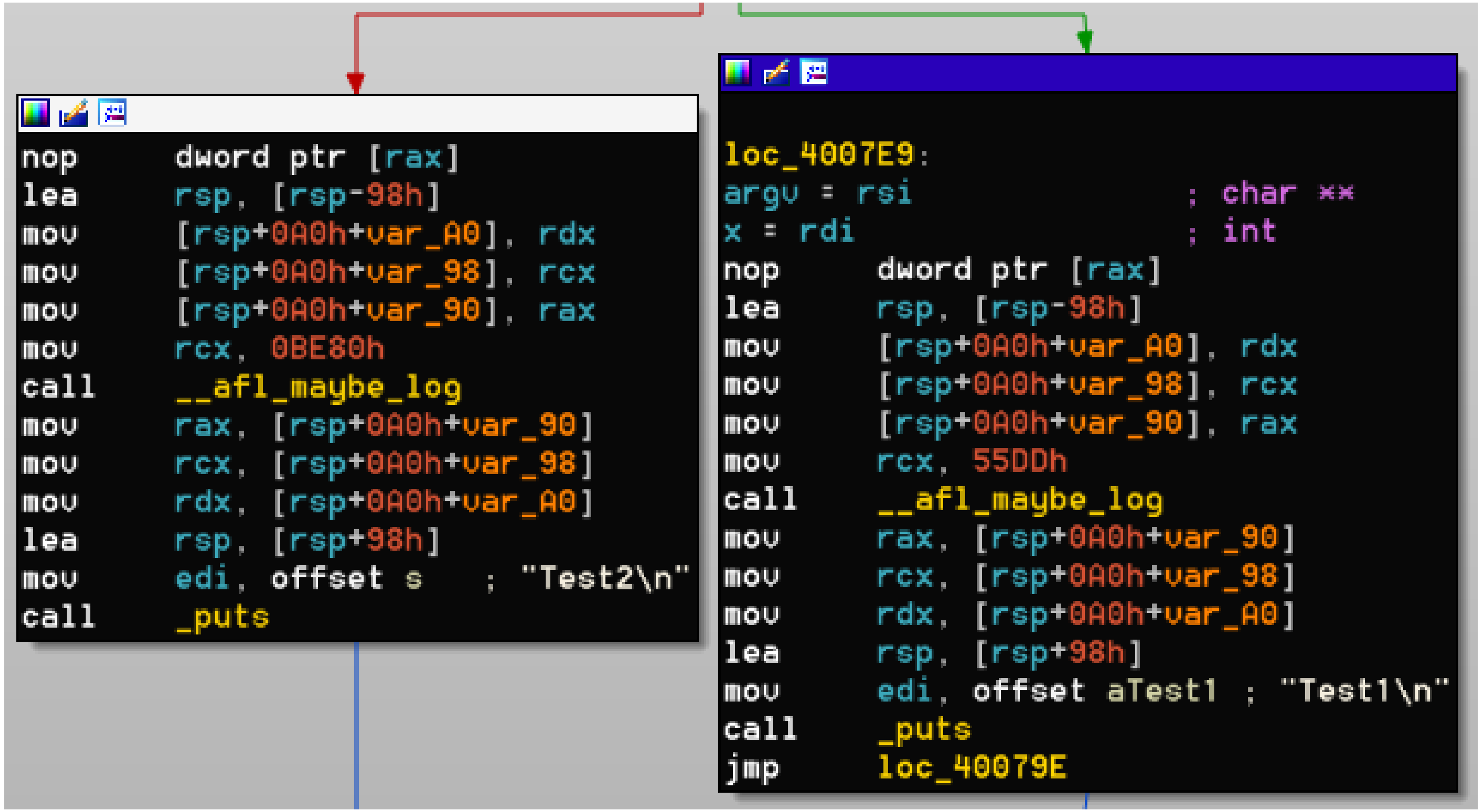
Feedback based fuzzing

- **Result:**

Store old register values

Instrumentation

Restore old register values



SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

American Fuzzy Lop - AFL

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

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

➔ AFL can distinguish between

- A->B->C->D->E (tuples: AB, BC, CD, DE)
- A->B->D->C->E (tuples: AB, BD, DC, CE)

American Fuzzy Lop - AFL

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

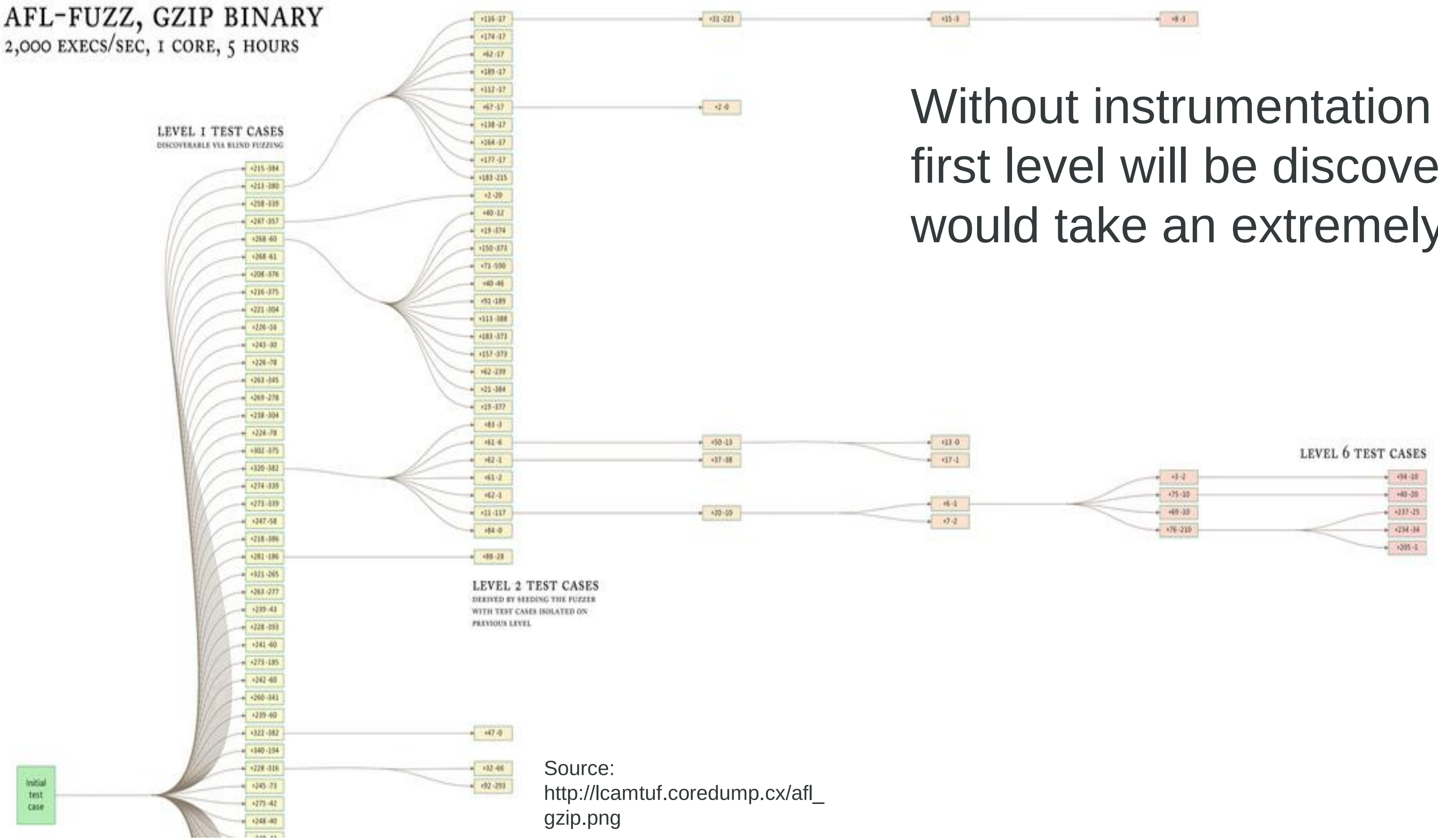
```
cur_location = <compile_time_random_value>;
bitmap[(cur_location ^ prev_location) % BITMAP_SIZE]++;
prev_location = cur_location >> 1;
```

➔ AFL can distinguish between

- A->B->C->D->E (tuples: AB, BC, CD, DE)
- A->B->D->C->E (tuples: AB, BD, DC, CE)

➔ Without shifting A->B and B->A are indistinguishable

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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Corpus Distillation

- We can either start fuzzing with an empty input folder or with downloaded / generated input files

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

- **Downloaded sample files:**
  - Much faster because AFL doesn‘t have to find the file format structure itself
  - Bing API to crawl the web (Hint: Don‘t use DNS of your provider …)
  - Other good sources: Unit-tests, bug report pages, …
  - Problem: Many sample files execute the same code ➔ **Corpus Distillation**

American Fuzzy Lop - AFL

Steps for fuzzing with AFL:

1. Remove input files with same functinality:
   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...] @@
   ```

American Fuzzy Lop - AFL

```
                 american fuzzy lop 2.49b (readelf)

- process timing ---------------------------------- overall results -
        run time : 42 days, 19 hrs, 27 min, 41 sec   cycles done : 3
   last new path : 0 days, 1 hrs, 45 min, 10 sec     total paths : 14.4k
 last uniq crash : 5 days, 19 hrs, 58 min, 31 sec   uniq crashes : 25
  last uniq hang : 1 days, 16 hrs, 58 min, 37 sec     uniq hangs : 161
- cycle progress ------------------- map coverage --------------------
  now processing : 1550* (10.74%)      map density : 0.39% / 18.87%
 paths timed out : 0 (0.00%)        count coverage : 4.30 bits/tuple
- stage progress ------------------- findings in depth ---------------
  now trying : bitflip 1/1          favored paths : 2220 (15.39%)
 stage execs : 880/106k (0.83%)      new edges on : 3431 (23.78%)
 total execs : 4.54G                total crashes : 1286 (25 unique)
  exec speed : 2338/sec              total tmouts : 25.5k (224 unique)
- fuzzing strategy yields -------------------------- path geometry ---
   bit flips : 5858/474M, 1418/474M, 557/474M          levels : 27
  byte flips : 86/59.4M, 57/13.2M, 57/13.6M           pending : 10.5k
 arithmetics : 2564/725M, 79/548M, 182/375M          pend fav : 1
  known ints : 162/47.6M, 359/226M, 374/425M        own finds : 14.4k
  dictionary : 0/0, 0/0, 1061/659M                   imported : n/a
       havoc : 1631/9.85M, 0/0                      stability : 100.00%
        trim : 2.82%/4.13M, 78.13%
                                                       [cpu003: 50%]
```

Demo Time!

Topic: Fuzzing FFMPEG with AFL

Runtime: 7 min 33 sec

Description: See the AFL workflow (afl-cmin, afl-tmin, afl-fuzz) in action

AFL with CVE-2009-0385 (FFMPEG)

```
                 american fuzzy lop 2.19b (ffmpeg)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 18 hrs, 52 min, 48 sec     │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 25 sec       │  total paths : 1179   │
│ last uniq crash : 0 days, 1 hrs, 15 min, 58 sec      │ uniq crashes : 14     │
│  last uniq hang : 0 days, 0 hrs, 12 min, 5 sec       │   uniq hangs : 73     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 205 (17.39%)      │    map density : 5205 (7.94%)          │
│ paths timed out : 14 (1.19%)        │ count coverage : 2.39 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 239 (20.27%)           │
│ stage execs : 34.6k/160k (21.64%)   │  new edges on : 376 (31.89%)           │
│ total execs : 19.8M                 │ total crashes : 554 (14 unique)        │
│  exec speed : 373.4/sec             │   total hangs : 19.6k (73 unique)      │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 91/5.51M, 30/5.51M, 21/5.51M          │    levels : 4          │
│  byte flips : 1/689k, 3/7463, 7/8669                │   pending : 1143       │
│ arithmetics : 50/383k, 10/27.5k, 6/11.5k            │  pend fav : 220        │
│  known ints : 7/35.8k, 21/203k, 34/196k             │ own finds : 1178       │
│  dictionary : 0/0, 0/0, 5/48.2k                     │  imported : n/a        │
│       havoc : 893/1.55M, 0/0                        │  variable : 0          │
│        trim : 26.75%/43.3k, 98.99%                  ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu000:161%]
```

AFL with CVE-2009-0385 (FFMPEG)

- AFL input with invalid 4xm file (strk chunk changed to strj)

```
30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  000000000000vtrkD...000000000000000000000000
30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  000000..00..00000000000000000000000000000000
30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  00000000000000000000000000000000000000000000
30 30 73 74  72 6A 28 00  00 00 00 00  00 00 00 00  00 00 30 30  000G0000000000000000000000strj(..........00
00 00 30 00  00 00 4C 49  53 54 30 30  30 30 4D 4F  56 49 4C 49  0000000000000000000....."0..0...LIST0000MOVILI
```

- 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)

```
30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  000000..00..00000000000000000000000000000000
30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30  00000000000000000000000000000000000000000000
30 30 73 74  72 6B 28 00  00 00 FF FF  FF FF 00 00  00 00 30 30  000G0000000000000000000000strk(..........00
00 00 30 00  00 00 4C 49  53 54 30 30  30 30 4D 4F  56 49 4C 49  0000000000000000000....."0..0...LIST0000MOVILI
```

LibFuzzer

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

- Highly recommended tutorial: http://tutorial.libfuzzer.info

LibFuzzer

```
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;
}
```

Source: http://tutorial.libfuzzer.info

Demo Time!

Topic: LibFuzzer vs. OpenSSL (Heartbleed)

Runtime: 41 sec

Description: See how LibFuzzer can be used to find heartbleed in several seconds.

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

```
user@user-VirtualBox:~/test$ AFL_NO_ARITH=1 AFL_PRELOAD=/home/user/test/libdislo
cator.so afl-fuzz -Q -x wordlist -i input/ -o output/ -- ./chat

                     american fuzzy lop 2.51b (chat)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 17 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec        │  total paths : 20     │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 1 (5.00%)         │    map density : 0.09% / 0.30%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.27 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 12 (60.00%)            │
│ stage execs : 152/768 (19.79%)      │  new edges on : 16 (80.00%)            │
│ total execs : 33.3k                 │ total crashes : 0 (0 unique)           │
│  exec speed : 1902/sec              │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 3/32, 1/30, 0/26                      │    levels : 2          │
│  byte flips : 0/4, 0/2, 0/0                         │   pending : 19         │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 12         │
│  known ints : 0/22, 0/0, 0/0                        │ own finds : 19         │
│  dictionary : 0/40, 2/60, 0/0                       │  imported : n/a        │
│       havoc : 13/32.8k, 0/0                         │ stability : 100.00%    │
│        trim : n/a, 0.00%                            ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu:309%]
```

AFL qemu mode

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


                     american fuzzy lop 2.51b (chat)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 0 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 7 sec        │  total paths : 12     │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 0.20% / 0.27%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.20 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 1 (8.33%)             │
│ stage execs : 6026/16.4k (36.78%)   │  new edges on : 10 (83.33%)           │
│ total execs : 6244                  │ total crashes : 0 (0 unique)          │
│  exec speed : 103.4/sec             │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ────────────┤
│   bit flips : 3/16, 1/15, 0/13                  │    levels : 2              │
│  byte flips : 0/2, 0/1, 0/0                     │   pending : 12             │
│ arithmetics : 0/0, 0/0, 0/0                     │  pend fav : 1              │
│  known ints : 0/9, 0/0, 0/0                     │ own finds : 11             │
│  dictionary : 0/20, 2/30, 0/0                   │  imported : n/a            │
│       havoc : 0/0, 0/0                          │ stability : 100.00%        │
│        trim : n/a, 0.00%                        ├────────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu:209%]
```

AFL qemu mode

```
user@user-VirtualBox:~/test$ AFL_NO_ARITH=1 AFL_PRELOAD=/home/user/test/libdislo
cator.so afl-fuzz -Q -x wordlist -i input/ -o output/ -- ./chat

                     american fuzzy lop 2.51b (chat)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 52 min, 39 sec      │  cycles done : 2     │
│   last new path : 0 days, 0 hrs, 4 min, 45 sec       │  total paths : 323   │
│ last uniq crash : 0 days, 0 hrs, 18 min, 9 sec       │ uniq crashes : 77    │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 208 (64.40%)      │    map density : 0.36% / 0.73%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.69 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored paths : 50 (15.48%)           │
│ stage execs : 595/768 (77.47%)      │  new edges on : 92 (28.48%)           │
│ total execs : 1.94M                 │ total crashes : 10.4k (77 unique)     │
│  exec speed : 905.9/sec             │  total tmouts : 40 (14 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 44/75.1k, 25/74.9k, 4/74.6k           │    levels : 12         │
│  byte flips : 0/9387, 0/9224, 0/8903                │   pending : 160        │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 0          │
│  known ints : 0/47.0k, 0/0, 0/0                     │ own finds : 322        │
│  dictionary : 13/137k, 3/140k, 0/23.6k              │  imported : n/a        │
│       havoc : 310/1.33M, 0/0                        │ stability : 100.00%    │
│        trim : 6.86%/3482, 0.00%                     ├────────────────────────┘
^C──────────────────────────────────────────────────┘          [cpu:313%]
```

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)

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Demo Time!

Topic: Breakpoint instrumentation of Adobe Reader

Runtime: 3 min 59 sec

Description: See how to use breakpoints and a debugger to get code-coverage. See limitations of this approach.

Code-Coverage via Breakpoints

- **Disadvantage:**
  - It’s very slow
    - Statically setting breakpoints can speedup the process, but it’s still slow because of the debugger process switches
    - Only really 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)

Code-Coverage via Breakpoints

- **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
  - Often useful during reverse engineering (E.g. dump registers at every breakpoint, see later demo)

Methods to measure code-coverage

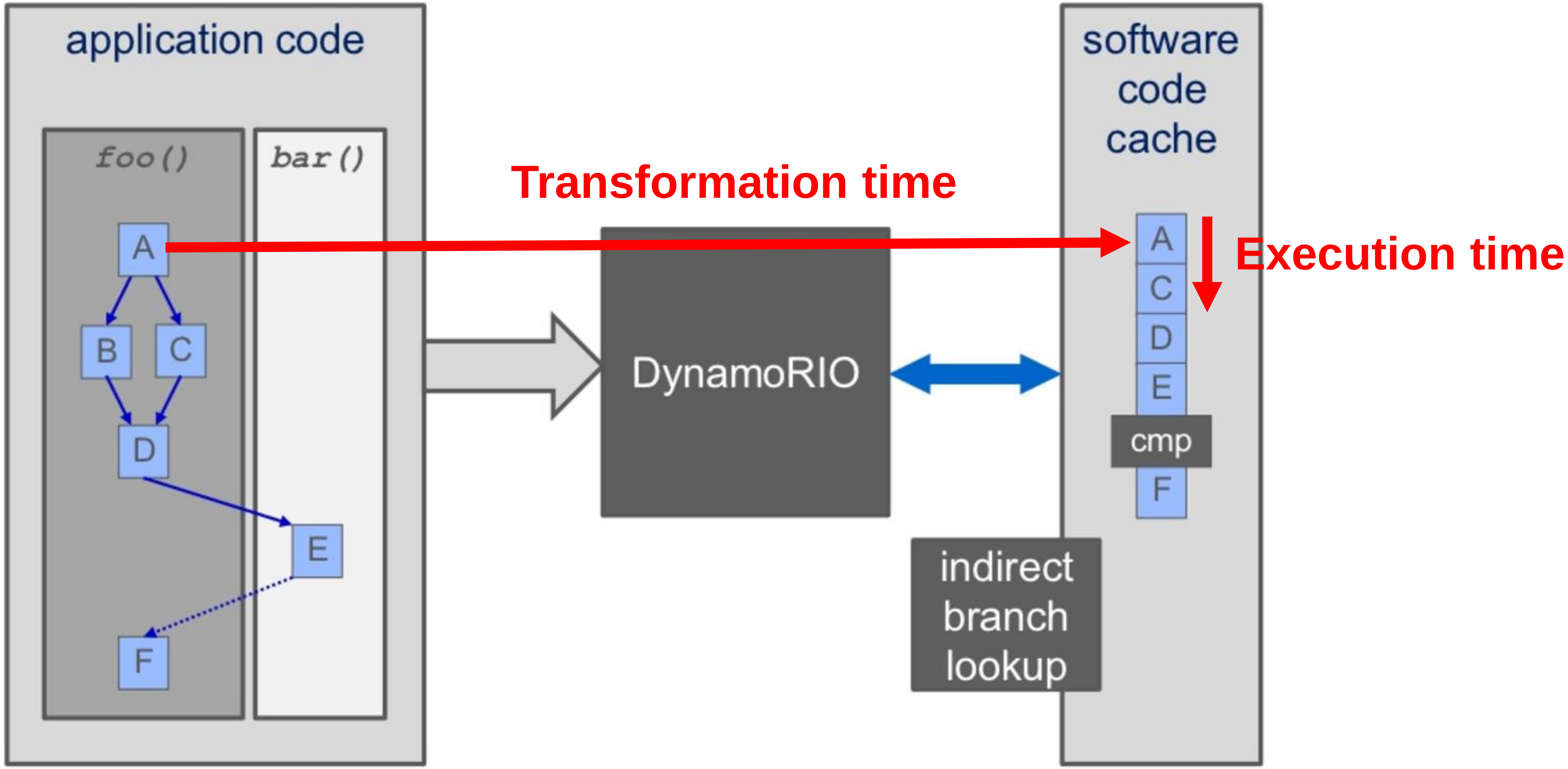
1. Instrumentation during compilation (source code available; gcc or llvm → AFL)
2. Emulation of binary (e.g. with qemu)
3. Writing own debugger and set breakpoints on every basicblock (slow, but useful in some situations)
4. Dynamic instrumentation of compiled application (no source code required; tools: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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

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 gets moved into code cache, load of module, exit of process, …)
- At callback (e.g. new basic block), we can further add instructions to the basic block which get executed every time the basic block gets executed!
  - Transformation time (Instrumentation Function): Analyzing a BB the first time (called once)
  - Execution time (Analysis Function): Executed always before instruction gets executed

DynamoRIO



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

DynamoRIO

- **For transformation time callbacks can be registered**
  - E.g.: drmgr_register_bb_instrumentation_event()
- **For execution time we have two possibilities**
  - 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

DynamoRIO

- **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

Demo Time!

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.

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Demo Time!

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.

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

WinAFL

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

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

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

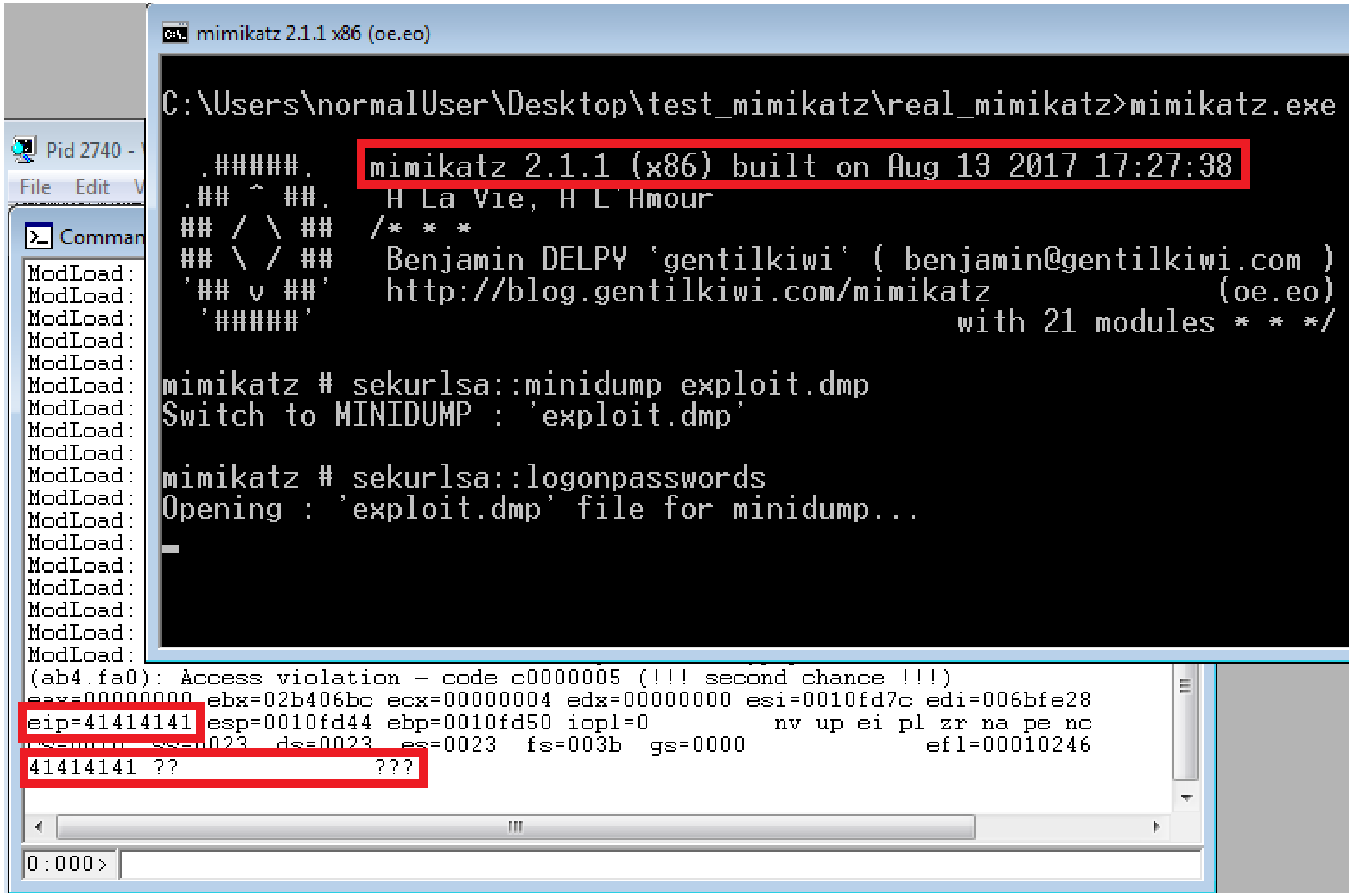
Demo Time!

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



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!**

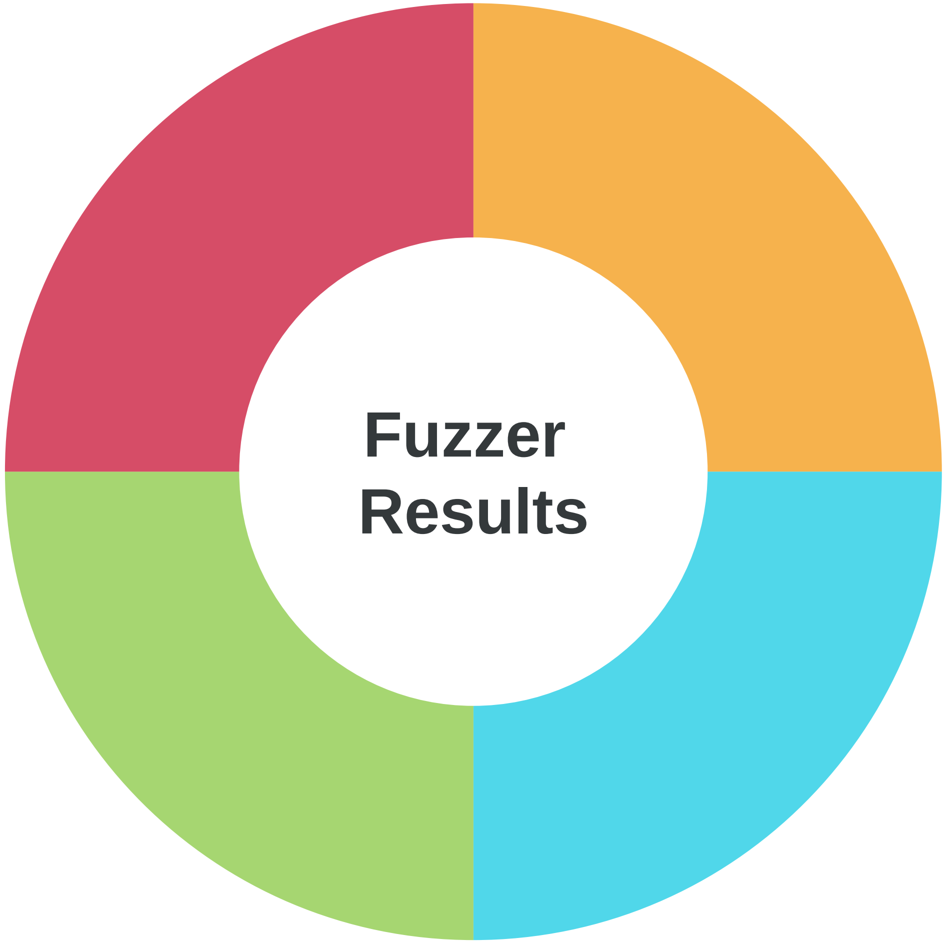
SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

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

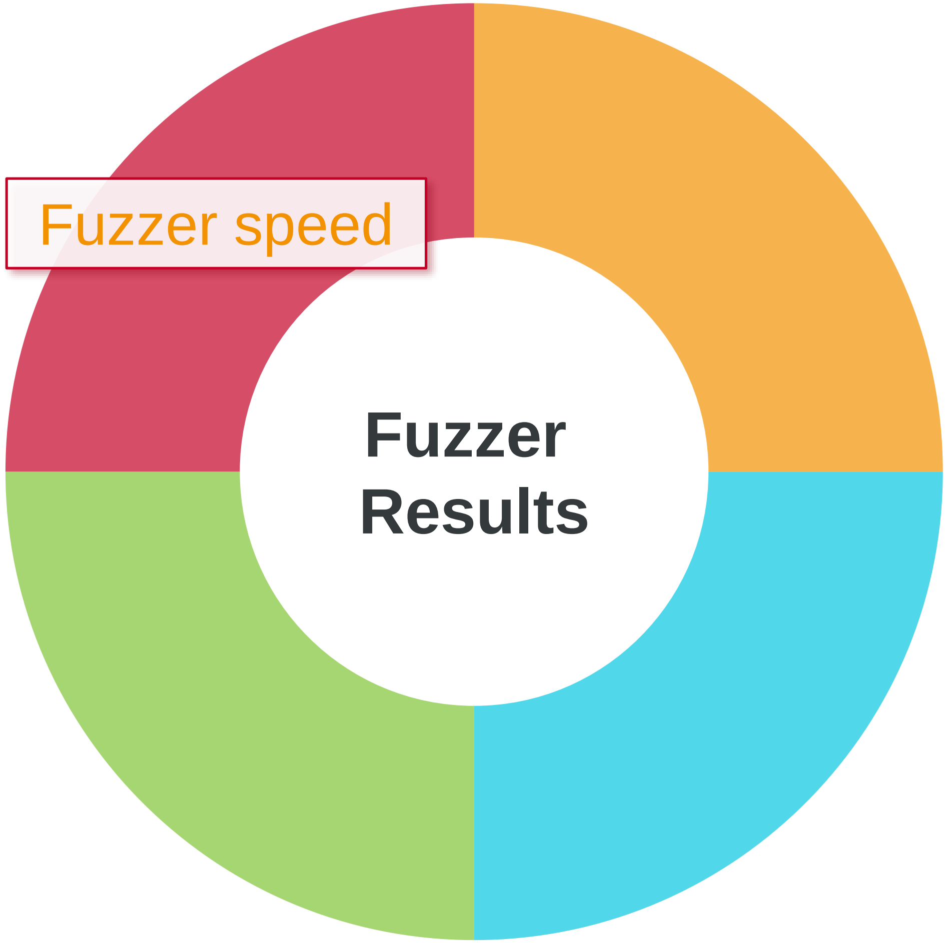
Areas which influent fuzzer results

Areas which influence fuzzing results



SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

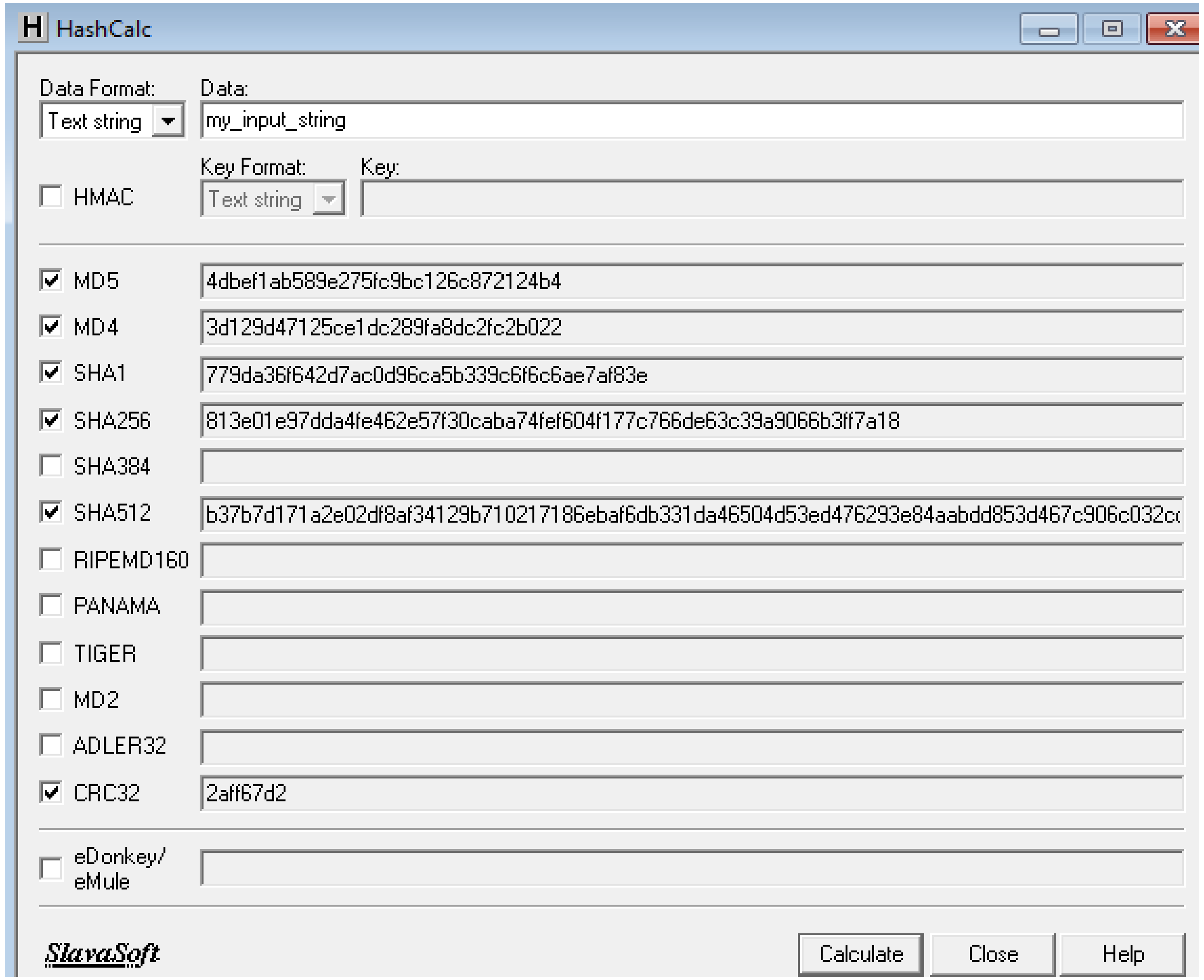
Areas which influence fuzzing results



Fuzzer Speed

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

GUI automation – Example HashCalc



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?

WinAFL

- **How to find the target function without source code?**

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

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

3. Place memory breakpoints on the input

4. Use a taint engine (see later)

Demo Time!

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

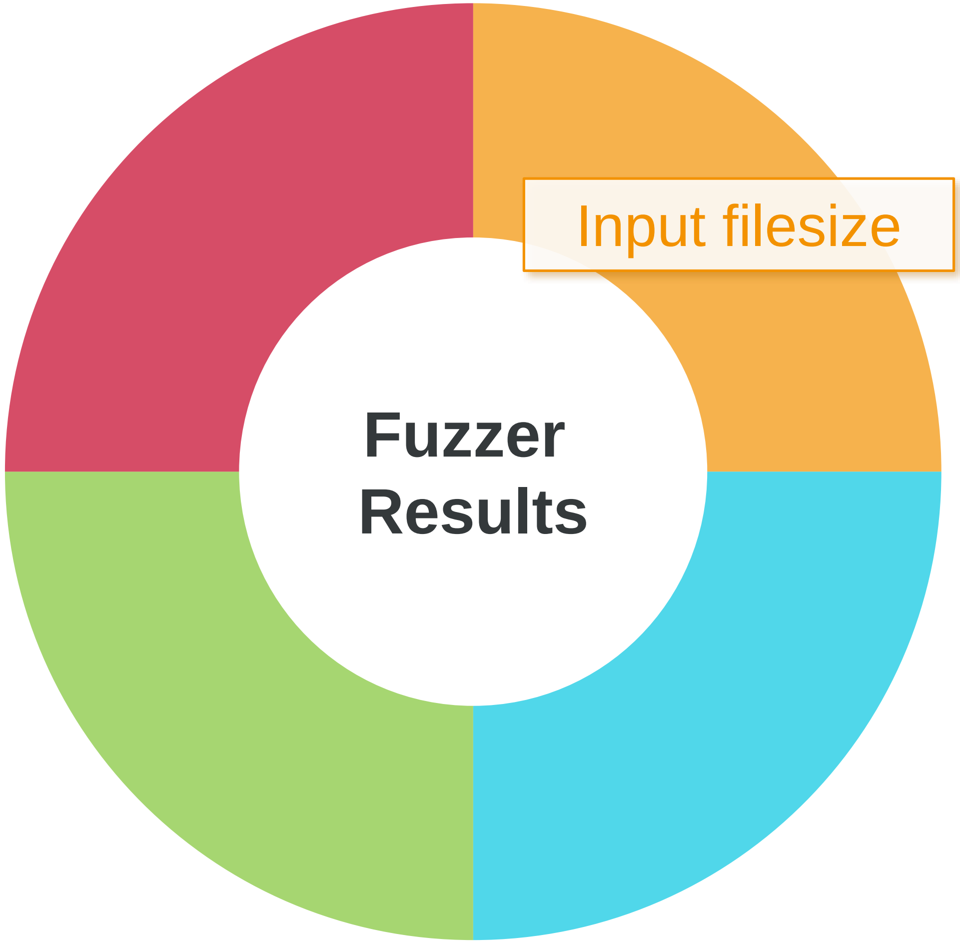
Description: Using the identified addresses and DynamoRio we can write an in-memory fuzzer to increase the fuzzing speed to

170 000 exec / sec!

GUI automation

- **HashCalc.exe MD5 fuzzing**
- GUI automation with AutoIt: **~2-3 exec / sec**
- In-Memory with debugger: **~750 exec / sec**
- In-Memory with DynamoRio (no instr.): ~**170 000 - 200 000 exec / sec**

Areas which influence fuzzing results



Input file size

- **The input file size is extremely important!**

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

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

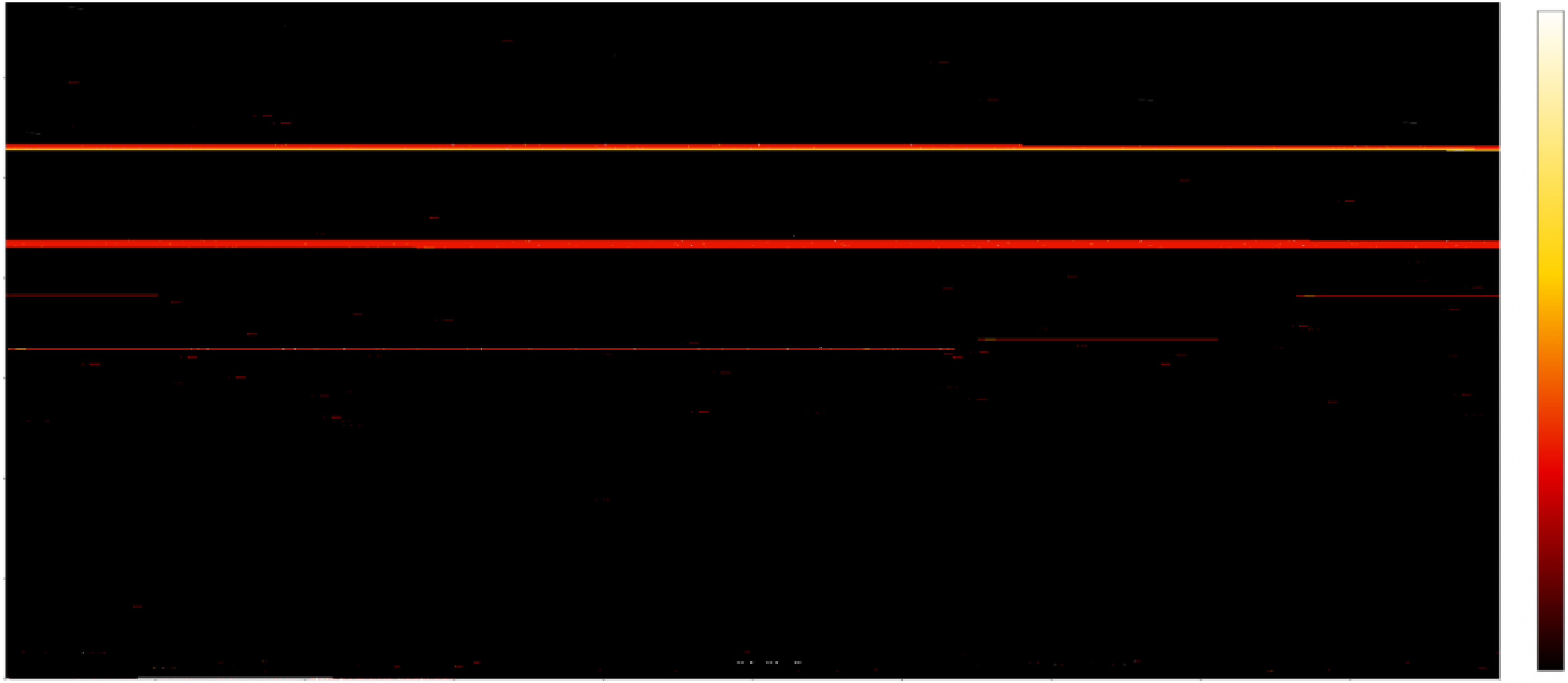
Input file size

- **Example: Fuzzing mimikatz**

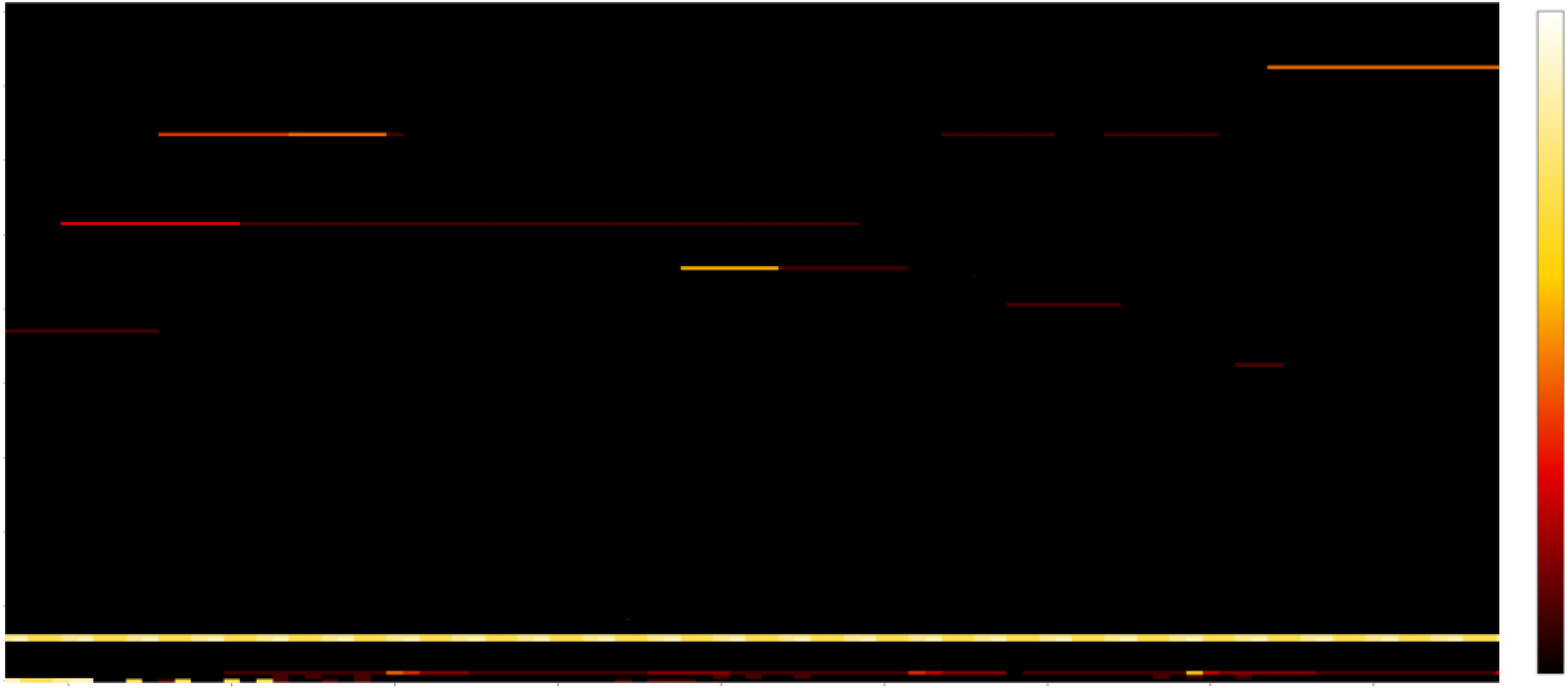
- Initial memory dump: **27 004 528 Byte**
- Memory dump which I fuzzed: **2 234 Byte**

➔ **I'm approximately 12 000 times faster with this setup…**
  - You would need 12 000 CPU cores to get the same result in the same time as my fuzzing setup with one CPU core
  - Or with the same number of CPU cores you need 12 000 days (~33 years) to get the same result as I within one day
  - In reality it's even worse, since you have to do everything again for every queue entry (exponential)

Heat map of the memory dump (mimikatz access)

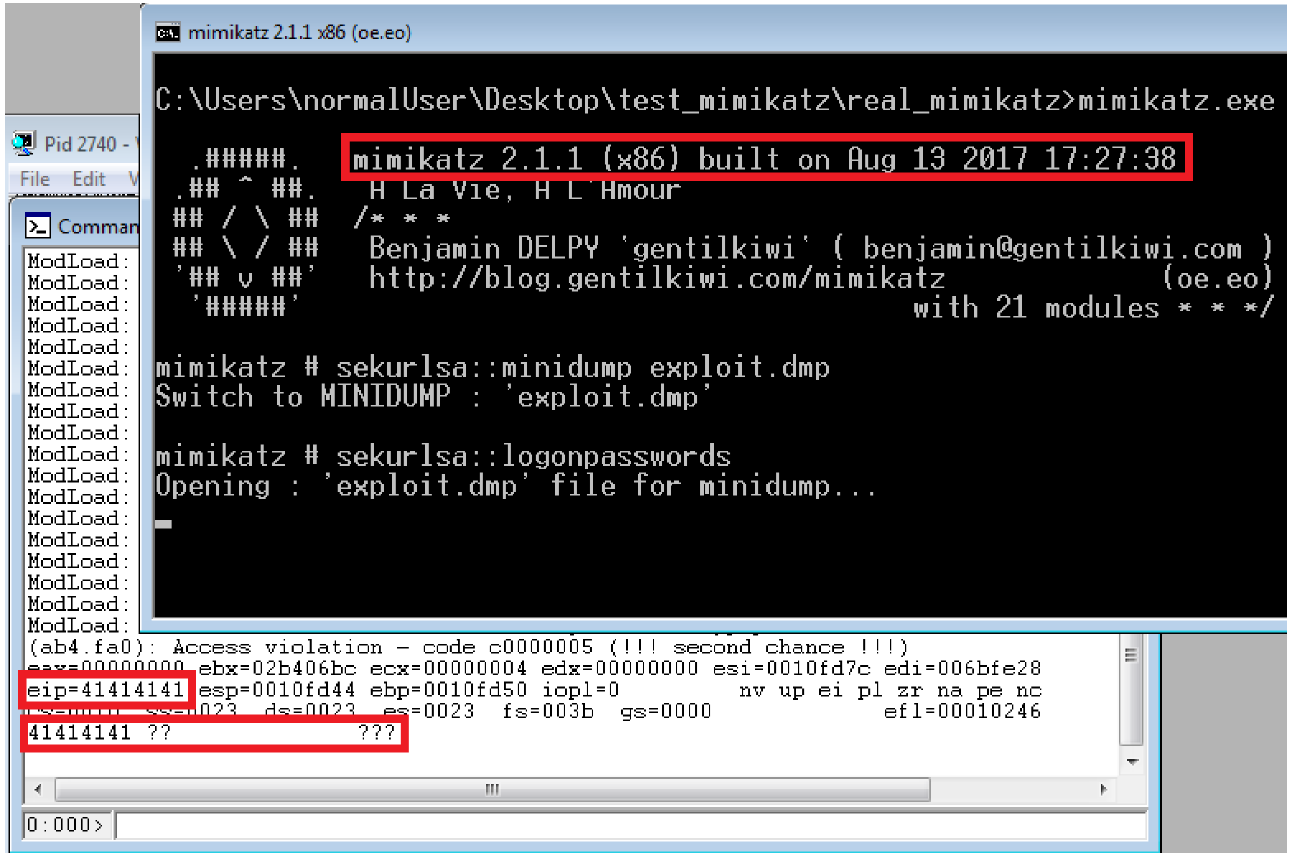


Heat map of the memory dump (mimikatz access) - Zoomed



Fuzzing and exploiting mimikatz

See below link for in-depth discussion how I fuzzed mimikatz with WinAFL:
https://www.sec-consult.com/en/blog/2017/09/hack-the-hacker-fuzzing-mimikatz-on-windows-with-winafl-heatmaps-0day/index.html



Creation of heatmaps

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

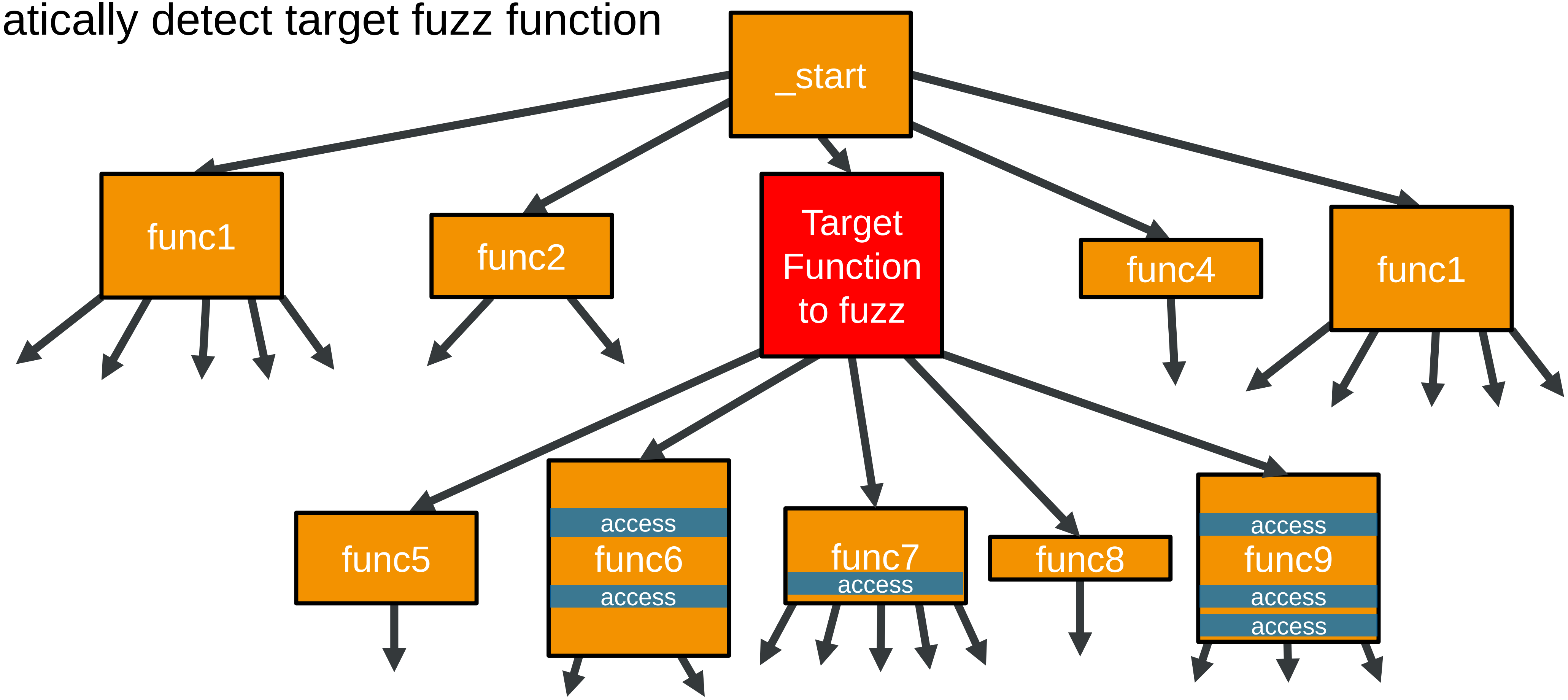
- Better approach: Use dynamic instrumentation / emulation
  - libdft
  - Triton
  - Panda
  - Manticore
  - Own PIN / DynamoRio tool

How my tool for heatmap creation work

1. Inject assembler instructions in front of all relevant application instructions (memory or register operations); Don't use clean calls because they are slow!
2. These instructions fill a buffer with "access struct" entries with the source (address or register id), the destination, the size and the semantic
   - Move semantic: `mov [0x12345678], eax`
   - Union semantic: `add EAX, [EBX]`
   - Untaint semantic: `mov EAX, 0x12345`
   - Increment hitcount semantic: `cmp, test, …`
3. After the (thread-specific) buffer is full, a clean call is made to process the access data, "timestamps" are used for multi-threaded applications
4. Shadow memory (1 byte to 1 bit) is used to indicate if a byte is tainted or not
5. AVL-like (balanced) tree is used to store a mapping from tainted memory ranges to the associated file offsets (to count hit counts for file offsets)

Combine Call-Graph with Taint-Analysis

- ➔ We can write a DynamoRio/PIN tool which tracks calls and taint status
- ➔ Automatically detect target fuzz function

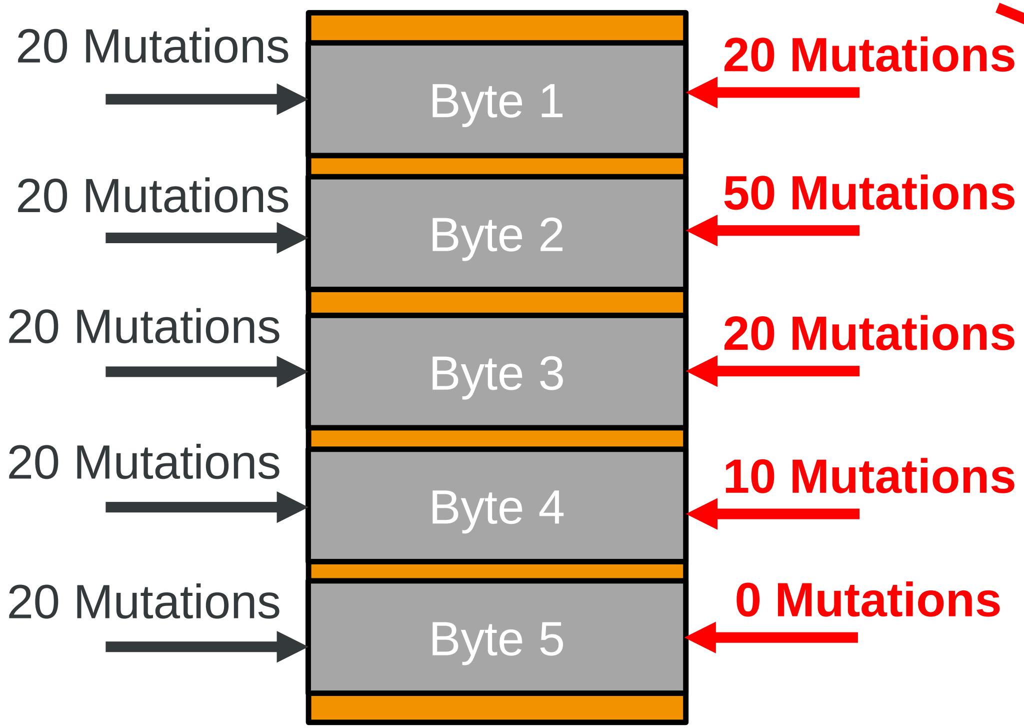


Fuzzing with taint analysis

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

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

Maybe don‘t fuzz this at all X2 is maybe a copy / search function



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

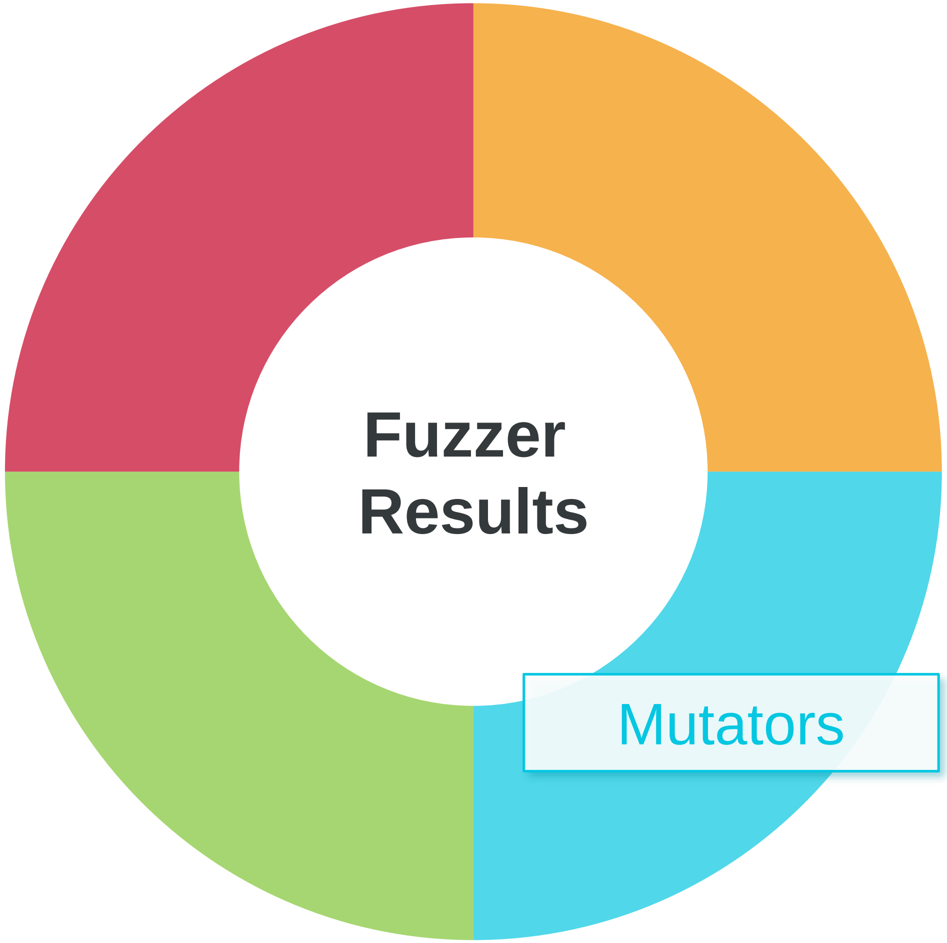
Byte 1 read by 2 instructions	2/10 = 20%
Byte 2 read by 5 instructions	5/10 = 50%
Byte 3 read by 2 instructions	2/10 = 20%
Byte 4 read by 1 instruction	1/10 = 10%
Byte 5 read by 0 instructions	0/10 = 0%

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

The power of dynamic instrumentation frameworks

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

Areas which influence fuzzing results



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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Radamsa

- **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

```
user-VirtualBox# echo "test1\n123\nbla\ntest2\nexit\n" | ./radamsa
test1
389356727742076683740647643182 8
bla
test0
exi t

user-VirtualBox# echo "test1\n123\nbla\ntest2\nexit\n" | ./radamsa
test1
123
bla
t

user-VirtualBox# echo "test1\n123\nbla\ntest2\nexit\n" | ./radamsa
test1�a���ą�l3te
�2
b
��1st2
exit
```

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

- **Example:** Our 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:**

Possible output

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

```
register
user3
login
user3
login
user3
deletete_user
```

Only variations of the current input file

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Radamsa

- **Correct invocation:**

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

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

Possible output

```
FOUND output (after 52345 executions)
register
user1
register
user2
login
user1
send_private_message
user2
Content of message
delete_user
login
user2
```

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)

Radamsa

- **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
  bei: 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
  lr2: 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 (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**

```
FOUND output (after 11450 executions)
register
user1
register
user2
login
user1
send_private_message
user3
delete user
```

Speed comparision

- 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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Speed comparision – input small text files

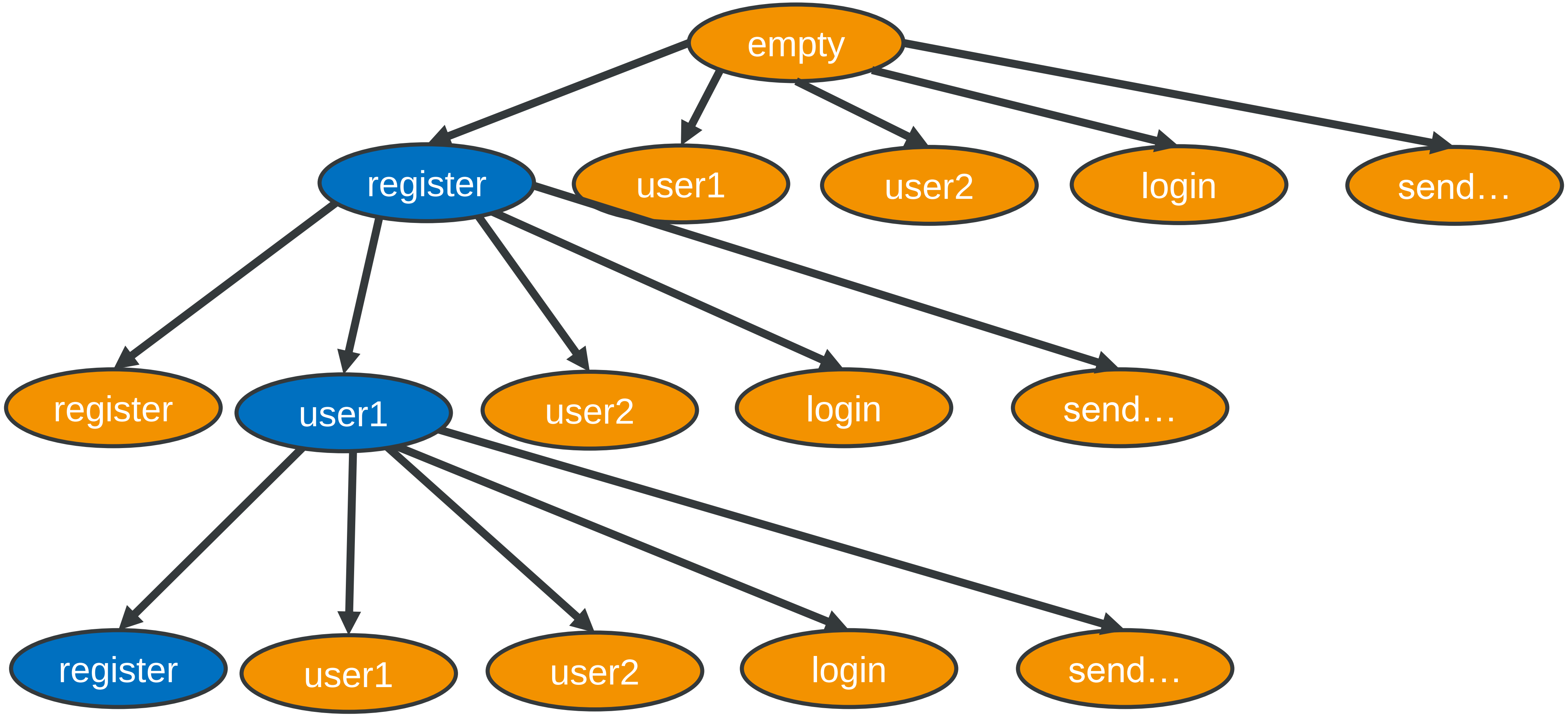
Type of test	Radamsa ext.	Radamsa fast ext.	Ni ext.	Ni library (ctypes)
Input stdin (1 tc), output stdout (1 tc)	~ 265	~ 345	(no stdin support)	-
Input files (3 tc), output stdout (1 tc)	~ 255	~300	~775	-
Input files (3 tc), output via files (100 tc)	~1100	~1930	~7300	-
Input via files (3 tc), output via files (1000 tc)	~1100	~2150	~8350	-
Input files (3 tc), output via files (100 tc); RD	~1220	~2740	~7300	-
Input files (3 tc), output via files (1000 tc); RD	**~1230**	**~3100**	**~8400**	-
Input 3 samples, output one (all in-memory)	-	-	-	~4000

The problem of the search space

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

```
register                  <- Depth 1
user1
register
user2
login
user1                     <- Depth 6
send_private_message
user2
content
delete_user               <- Depth 10
login
user2
view_messages             <- Depth 13
```

The problem of the search space



<empty>

register

register
user1

register
user1
register

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

The problem of the search space

- **We need at least 7 distinct inputs 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 inputs
- We have to find 13 inputs in the correct order to trigger the bug!
- **For 13 inputs 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

The problem of the search space

➔ **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
```

Depth 4 →

```
send_private_message
user2
content
delete_user
login
user2
view_messages
```


SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Demo Time!

Topic: Fuzzing SECCON CTF binary (text feedback)

Runtime: 3 min 21 sec

Description: See how we can enhance the Fuzzer to find the 3rd (deep) use-after-free bug!

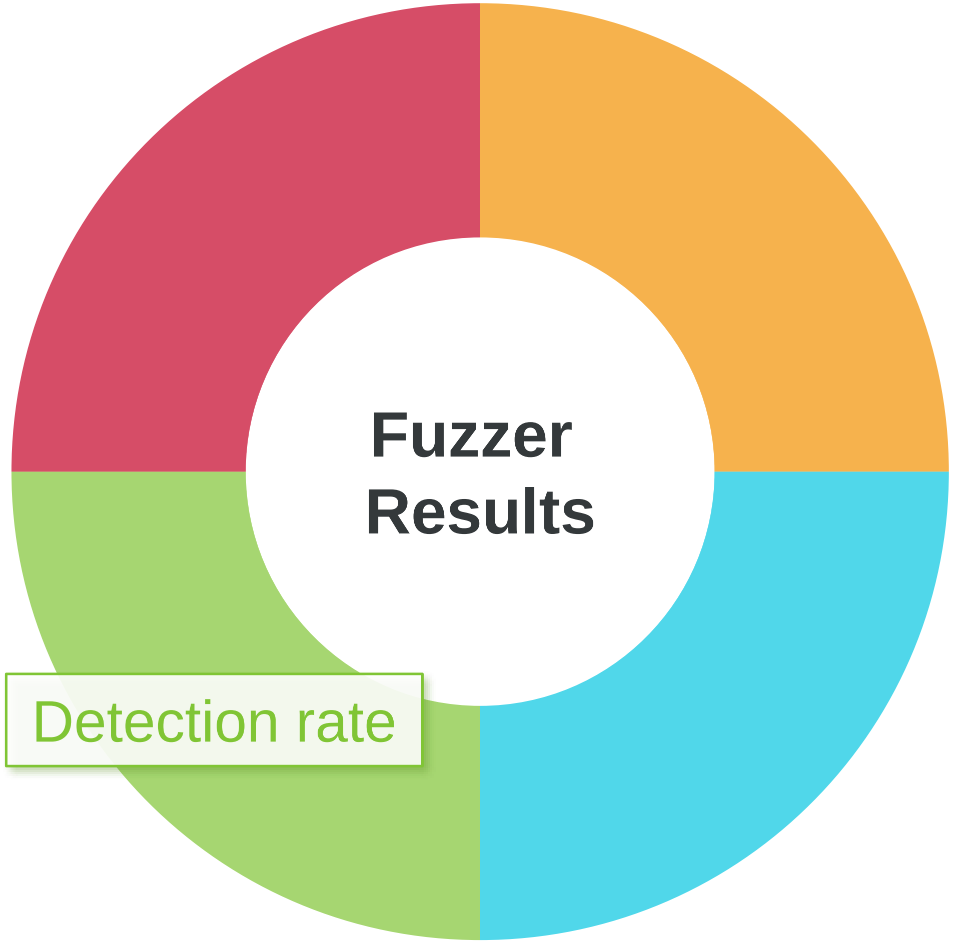
Chat CTF Fuzzer

- Runtime to find the deep second UAF (Use-After-Free) vulnerabiltiy…

```
user@user-VirtualBox:~/test$ python fuzzer2.py
^Cueue: 528, runtime: 7 sec, execs: 2774, exec/sec: 357.80, crashes: 21 BOF [+],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 BOF [+],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 BOF [+],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 BOF [+],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!!

Areas which influence fuzzing results



Detecting not crashing vulnerabilities

➔ Did you notice, that we triggered 3 (!) not crashing vulnerabilities during the „chat“ introduction demo?

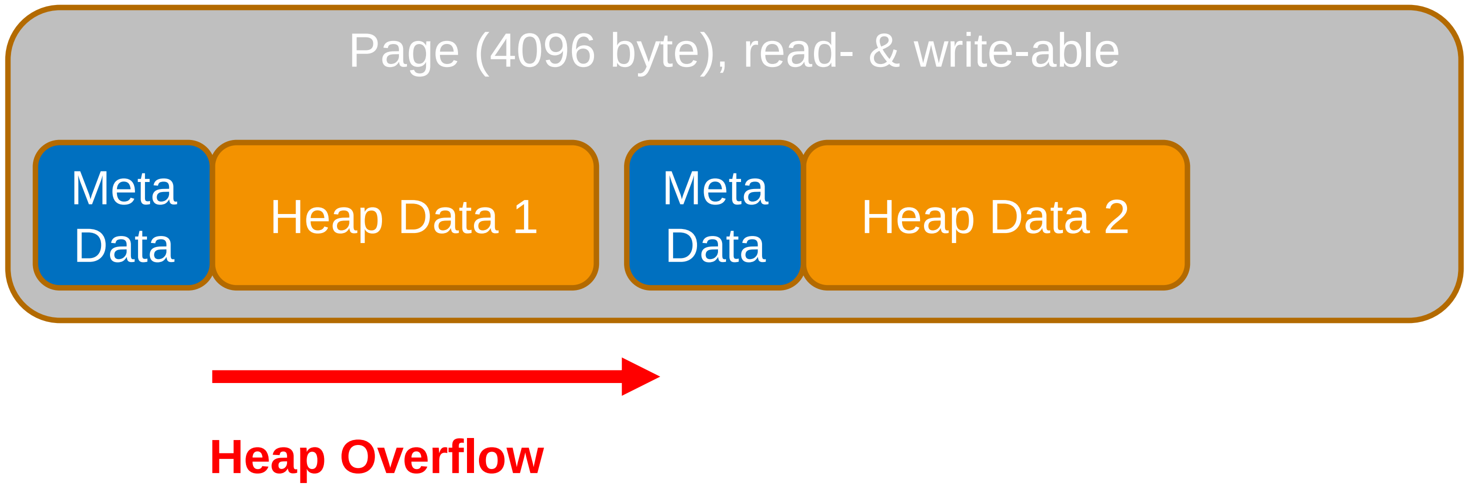
➔ And we didn‘t really see one of the bugs!

➔ Our Fuzzer would also not see the bugs…

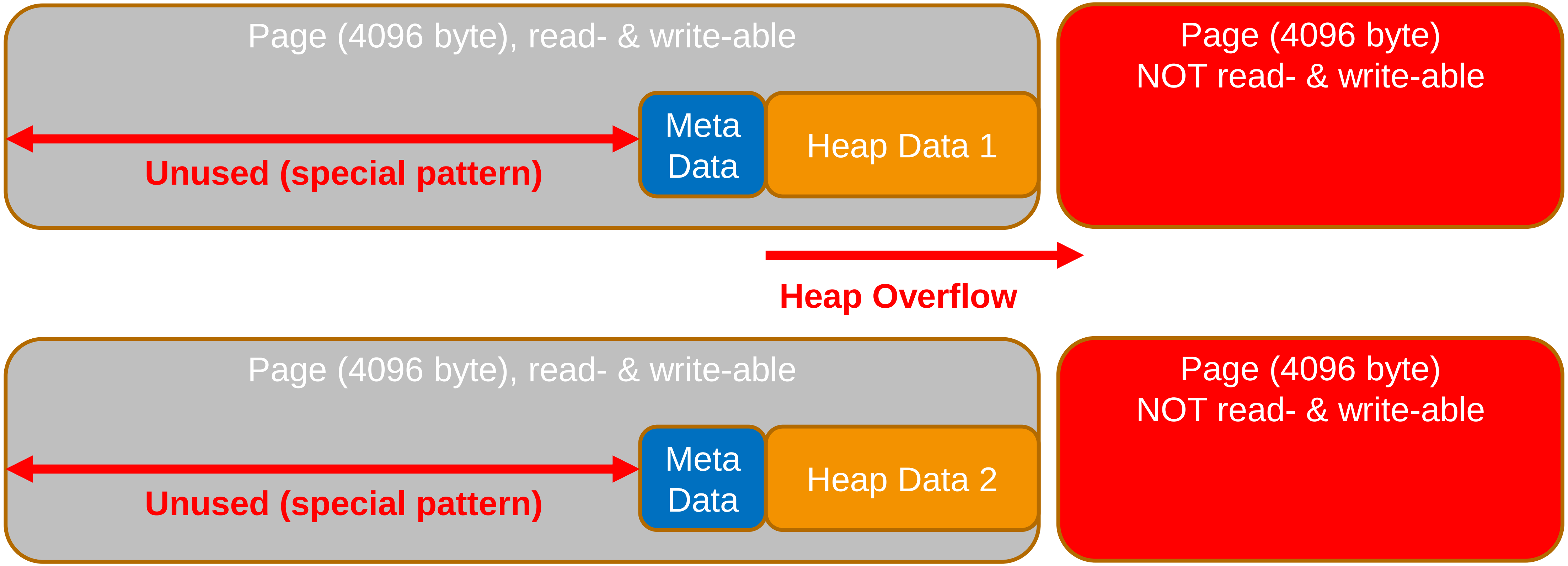
➔ Other real world example: Heartbleed is a read buffer overflow and does not lead to a crash…

➔ **We (the Fuzzer) need a way to detect such flaws / vulnerabilities!**

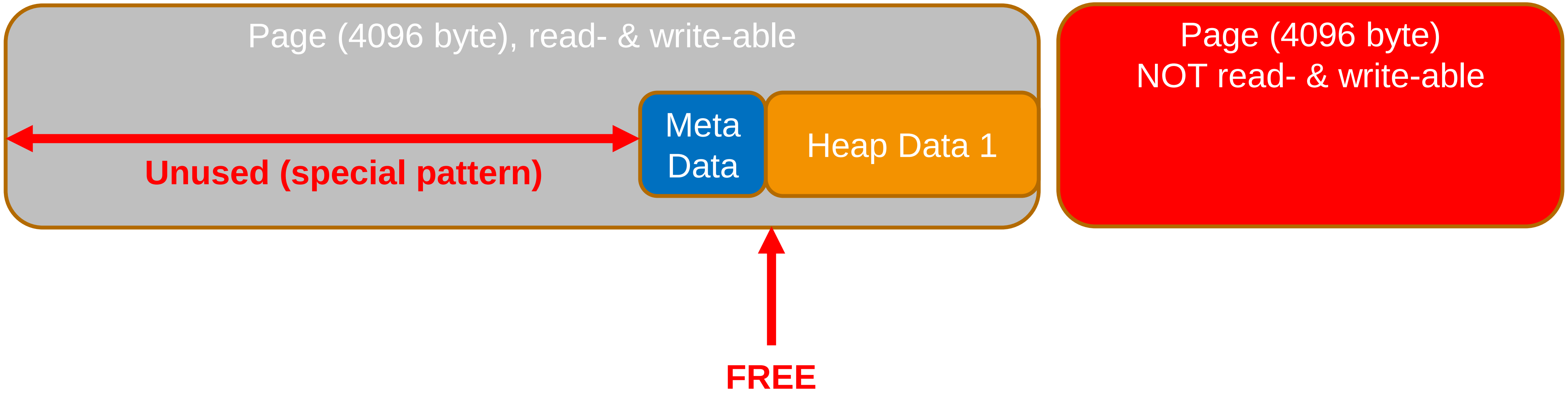
Heap Overflow Detection



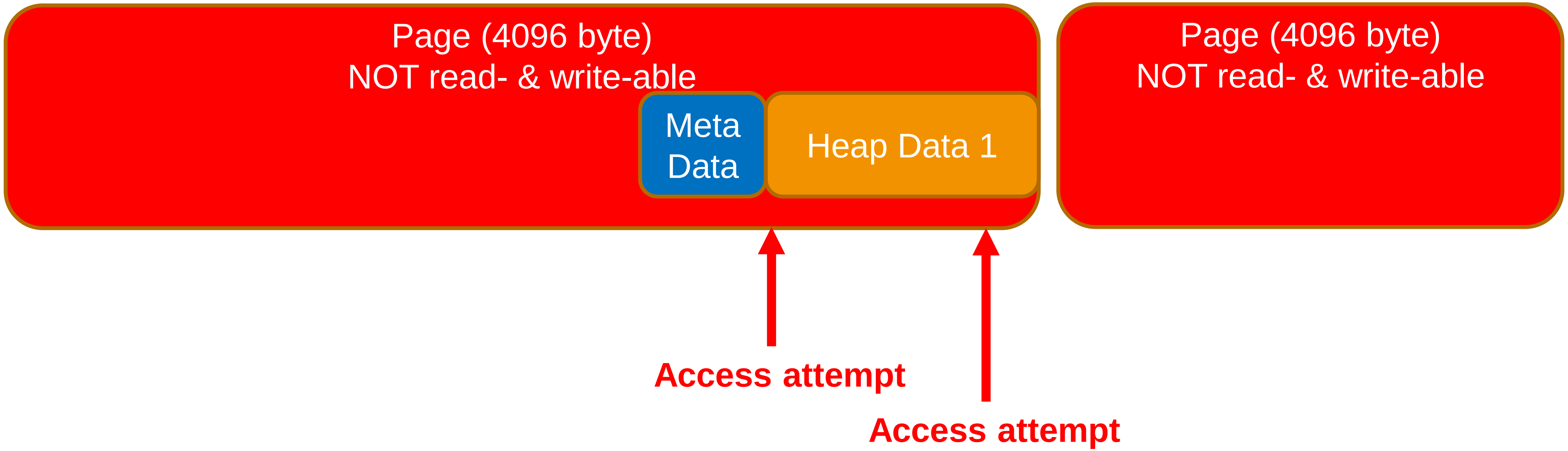
Heap Overflow Detection



Use-After-Free Detection



Use-After-Free Detection



Heap Library

- **Libdislocator** (shipped with AFL) implements exactly this

```
/* 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 + PG_COUNT(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 + PG_COUNT(len + 8) * PAGE_SIZE, PAGE_SIZE, PROT_NONE))
  FATAL("mprotect() failed when allocating memory");
```

One extra page which is not RW

- We can also set AFL_HARDEN=1 before make (Fortify Source & Stack Cookies)

Libdislocator catches heap overflow

```
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)
```


SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Libdislocator catches Use-After-Free

```
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 >>         x
Change name error...
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

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.

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Detecting not crashing vulnerabilities

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

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

➔ Use sanitizers during development !!!

Detecting not crashing vulnerabilities

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

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

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

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

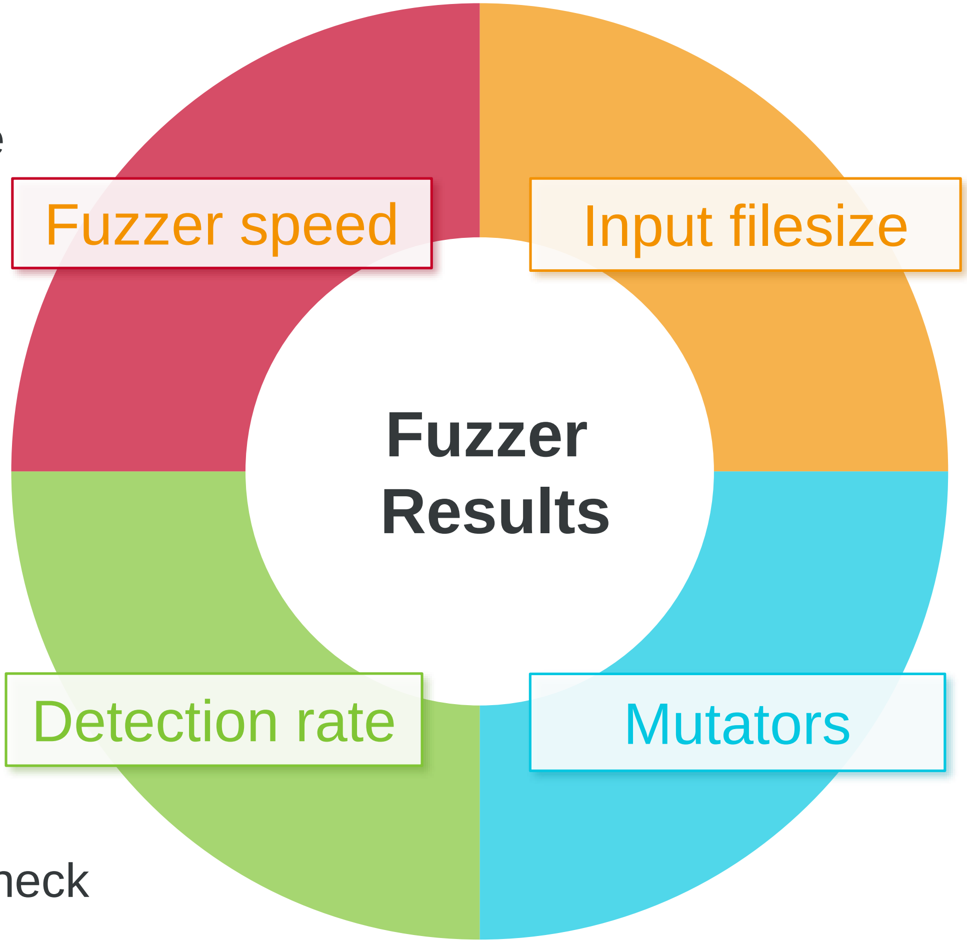
Free() Detection here! →

Use after free →

```
send_private_message
user2
content
delete_user
login
user2
view_messages
```

Overview: Areas which influence fuzzing results

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



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

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

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

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:

```
numwrites=random.randrange(math.ceil((float(len(buf)) / FuzzFactor)))+1for j in
range(numwrites):rbyte = random.randrange(256)rn =
random.randrange(len(buf))buf[rn] = "%c"%(rbyte);
numwrites=random.randrange(math.ceil((float(len(buf)) / FuzzFactor)))+1for j in
range(numwrites):rbyte = random.randrange(256)rn =
random.randrange(len(buf))buf[rn] = "%c"%(rbyte);
```

Source: Charlie Miller „Babysitting an Army of Monkeys“

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

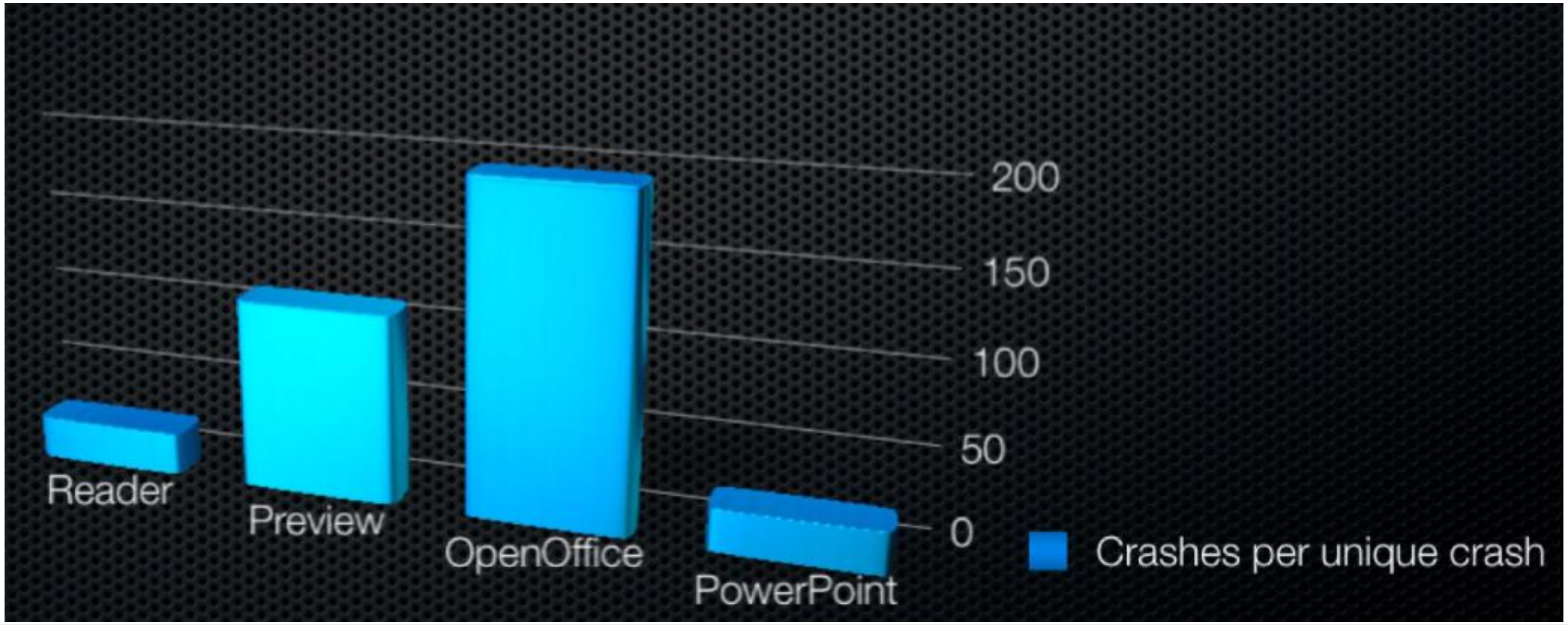
Some public fuzzing numbers

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:

- https://nordictestingdays.eu/files/files/jaanus_kaap_fuzzing.pdf
- 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

Some public fuzzing numbers

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

Some public fuzzing numbers

Google fuzzed the DOM of major browsers in 2017:

https://googleprojectzero.blogspot.co.at/2017/09/the-great-dom-fuzz-off-of-2017.html

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.

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

Rules for fuzzing

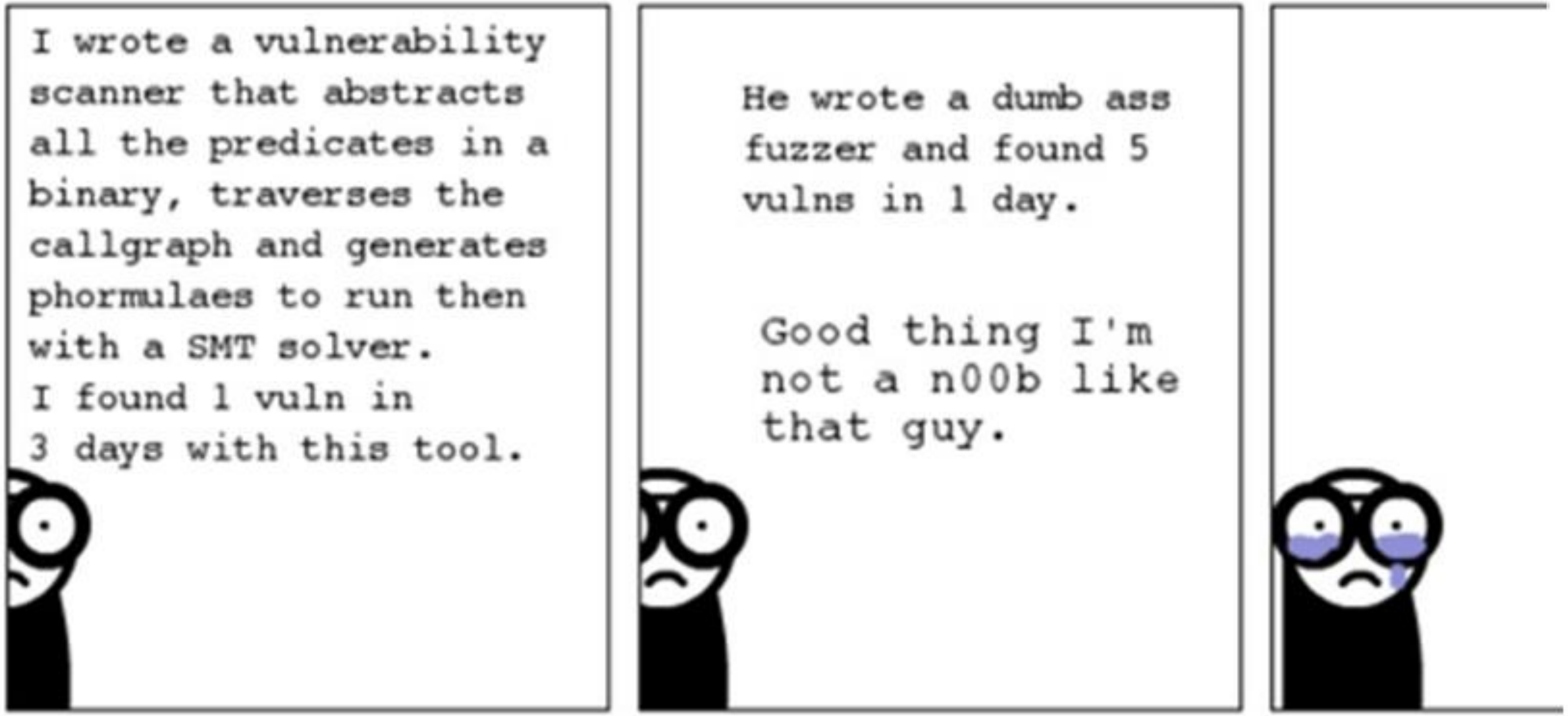
Fuzzing rules

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

A last hint…

Fuzzing can show the presence bugs but <u>can't</u> prove the absence of bugs!

Thank you for your attention!



Source: Twitter

For any further questions contact **your SEC Consult Expert.**

René Freingruber

@ReneFreingruber

r.freingruber@sec-consult.com

+43 676 840 301 749

SEC Consult Unternehmensberatung GmbH
Mooslackengasse 17
1190 Vienna, AUSTRIA

www.sec-consult.com

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY

SEC Consult in your Region.

AUSTRIA (HQ)

SEC Consult Unternehmensberatung GmbH
Mooslackengasse 17
1190 Vienna

Tel +43 1 890 30 43 0
Fax +43 1 890 30 43 15
Email office@sec-consult.com

GERMANY

SEC Consult Deutschland Unternehmensberatung GmbH
Ullsteinstraße 118, Turm B/8 Stock
12109 Berlin

Tel +49 30 30807283
Email office-berlin@sec-consult.com

SWITZERLAND

SEC Consult (Schweiz) AG
Turbinenstrasse 28
8005 Zürich

Tel +41 44 271 777 0
Fax +43 1 890 30 43 15
Email office-zurich@sec-consult.com

LITHUANIA

UAB Critical Security, a SEC Consult company
Sauletekio al. 15-311
10224 Vilnius

Tel +370 5 2195535
Email office-vilnius@sec-consult.com

SINGAPORE

SEC Consult Singapore PTE. LTD
4 Battery Road
#25-01 Bank of China Building
Singapore (049908)

Email office-singapore@sec-consult.com

CANADA

i-SEC Consult Inc.
100 René-Lévesque West, Suite 2500
Montréal (Quebec) H3B 5C9

Email office-montreal@sec-consult.com

RUSSIA

CJCS Security Monitor
5th Donskoy proyezd, 15, Bldg. 6
119334, Moscow

Tel +7 495 662 1414
Email info@securitymonitor.ru

THAILAND

SEC Consult (Thailand) Co.,Ltd.
29/1 Piyaplace Langsuan Building 16th Floor, 16B
Soi Langsuan, Ploen Chit Road
Lumpini, Patumwan | Bangkok 10330

Email office-vilnius@sec-consult.com

SEC Consult
ADVISOR FOR YOUR INFORMATION SECURITY