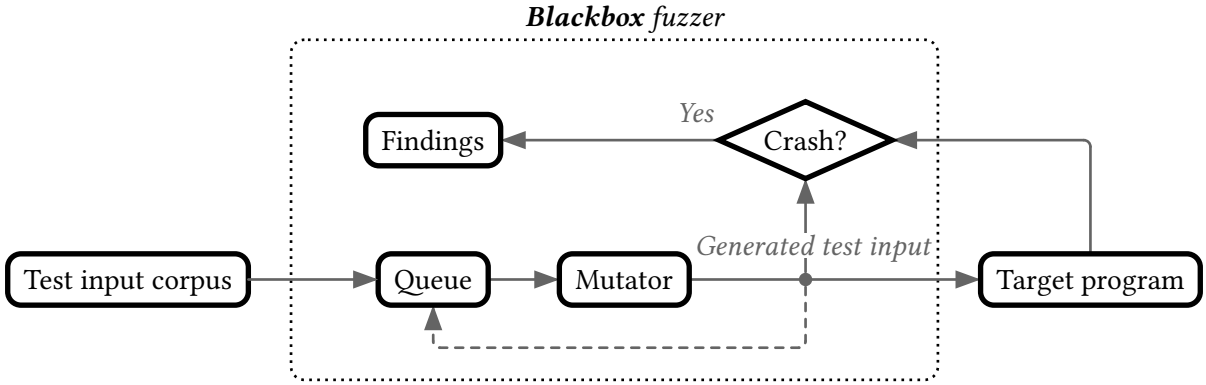
Intro to fuzzing

NET5534/NET6535

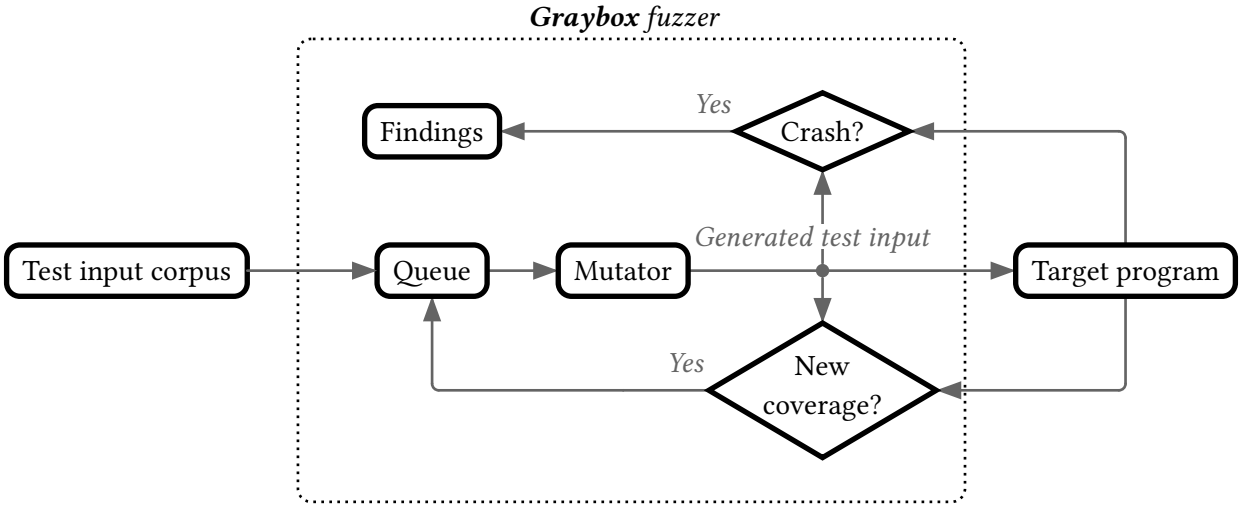
In this lab, you will learn the basics of fuzzing as a vulnerability detection approach. We will use AFL++,[1] a state-of-the-art *graybox* fuzzer.

What is fuzzing?

The term *fuzzing* (actually, *fuzz generator*) originated in the '90s[2] as an automated testing approach, during which randomly generated inputs are fed into a target program, and its behavior is monitored (e.g., for crashes). Such fuzzers were originally **blackbox**, meaning that they treated the target program as a "black box" and only looked at the *outcome* of the execution (i.e., *crash* or *no crash*). Blackbox fuzzers are **very fast**, but their efficacy is highly dependent on the inherent randomness of the process: a crash *might* or *might not* be triggered depending on the randomly generated inputs.



The next step in fuzzer evolution came in the form of **graybox** fuzzers. They address this core limitation of blackbox fuzzers by integrating a **feedback loop** which can be used to **guide** the random input generation, which we refer to as the **mutation** of the test inputs:



The trade-off here is that graybox fuzzers **optimize for an objective** (usually **code coverage**), thus guiding the mutations, in exchange for a **smaller throughput**, since now computation must be performed to evaluate the *performance* of a given test input. The motivation behind optimizing for code coverage is that, **the more code you cover, the more likely you are to uncover bugs**.

---

[1] https://github.com/AFLplusplus/AFLplusplus

[2] Miller, Barton P., Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." *Communications of the ACM* 33, no. 12 (1990): 32-44.

Modern graybox fuzzers such as AFL++ (the fuzzer you will be using today) have a notable track record in bug detection and they have been proven to be efficient **vulnerability detection tools**,[3] thus making them an effective and interesting approach in the domain of security.

Setup

You can use the dedicated Docker image containing all tools needed to complete this lab. First, you need to download the image by running the following command:

```
docker pull plumtrie/fuzzing-intro-lab:0.3.1
```

Then, you can start a Docker container using the image by running the following command:

```
docker run -it --rm plumtrie/fuzzing-intro-lab:0.3.1
```

After running this command, you should be greeted with a shell inside the container; you are now set up to complete this lab!

Problem 1

You can find the first problem under `~/problem-01` in your container. It is a fairly simple C project with a Makefile. You can build it by running `make`:

```
$ make
...
$ ls build/
vulnerable-01
```

If you try running the vulnerable program, you will see that it asks you for a password:

```
$ ./build/vulnerable-01
Password: test
Access denied >:(
```

Unfortunately, we do not know the password! We would like to look for any **potential vulnerabilities** which would allow us to **bypass the password check** and get access anyway, without a legitimate password. We can use **a fuzzer** to do just that!

Question 1.1

What do we need to fuzz the vulnerable program?

Question 1.2

How do we actually fuzz the program? How do we monitor the fuzzer?

Question 1.3 – Bonus

How can we use the crash found by the fuzzer to craft an exploit and bypass the authentication in the vulnerable program?

Problem 2

Problem 2 can be found under `~/problem-02` in your container. Just like the first problem, you can build it with `make`.

---

[3] https://aflplus.plus/#trophies

Question 2.1
Experiment with the program by running it and providing different inputs. What do you observe?

Question 2.2
How can you fuzz this program? Give a detailed step-by-step explanation.

Question 2.3
What do you observe? Does the fuzzer find any crashes? If not, why, and how can we fix it? (We know that the program contains a vulnerability.)

Question 2.4 - Bonus
How can we use the crash found by the fuzzer to craft an exploit and bypass the authentication in the vulnerable program?

Problem 3
Problem 3 can be found under `~/problem-03` in your container. Just like the first two problems, you can build it with `make`.

Question 3.1
Experiment with the program by running it and providing different inputs. What do you observe?

Question 3.2
How can you fuzz this program? Give a detailed step-by-step explanation.

Question 3.3
What do you observe? Does the fuzzer find any crashes? If not, why, and how can we fix it? (We know that the program contains a vulnerability.)

Question 3.4 - Bonus
How can we use the crash found by the fuzzer to craft an exploit and bypass the authentication in the vulnerable program?

Problem 4 – Bonus (hard)
Problem 4 can be found under `~/problem-04` in your container. Unlike the other problems, this one is **pre-built** with instrumentation; you do not have access to the source code.

The vulnerable program here is none other than the famous Unix utility Sudo.[4] We want to investigate this suspicious version of Sudo, as we suspect there might be a *backdoor* hidden within the binary. As a reminder, Sudo is most often invoked with `sudo <CMD>`, where `<CMD>` is a command to be run in the shell **as the root user**. It then attempts to authenticate the user (usually with password entry via `stdin`), and if successful, runs `<CMD>`. Since we are looking for a backdoor, the most obvious place to start from is the password checking step: could there be a *special backdoor password* that allows us to bypass authentication?

The pre-compiled Sudo binary can be found under `~/problem-04/build/bin/sudo.afl`.

Question 4.1
What is the `afl-fuzz` invocation we should use for `sudo.afl`?

Question 4.2
Suppose there *is* a special backdoor password, allowing us to bypass authentication. Would a "vanilla" graybox fuzzer—as it was presented to you at the beginning of this lab—be able to detect such a password easily? Why? Why not?

[4] https://www.sudo.ws/

Question 4.3

Recent research[5] shows that can we can sufficiently mitigate the *magic byte* problem, which is currently keeping us from fuzzing Sudo. Does AFL++ have any features that might help?

Question 4.4

Suppose we have solved all other issues and are ready to fuzz Sudo. As explained before, modern graybox fuzzers still only come with rudimentary *test oracles* out of the box. They can detect crashes or even non-crash memory errors, but they cannot detect backdoors of this form.

In order to attempt to detect this backdoor, we will leverage what we know about the internals of Sudo. We know that, at a high level, it asks for a password, checks it against some stored password, and if successful, runs the provided command. In Unix, we know that we can convert the current process into some other process via the `exec*()` API, with the most popular variant being `execve()`.[6] So, if `execve()` is called, we know that a command was launched, which means that the authentication was successful. Since we do not know the real password—and it is very unlikely that the fuzzer will randomly guess it—we must assume that there is a *backdoor* in our version of Sudo, with a hardcoded password trigger.

So, we could convert this heuristic of backdoor detection into something that AFL++ can understand, by **converting a call to `execve()` into a crash**. This can be accomplished fairly easily via another widely used technique in fuzzing: **`LD_PRELOAD` stubbing**. In short, `LD_PRELOAD` allows us to override calls to dynamic libraries (including `libc`, which contains the code for `execve()`) and replace them with *stubs*, custom implementations of library functions that we have written.

A stub is provided in your container under `~/problem-04/execve-crash.so`. If you look at its source code (`~/problem-04/execve-crash.c`), you will see that it is very simple: it just calls `abort()` whenever `execve()` is called, thus making the program which called `execve()` crash. Since crashes are discoverable by AFL++ with no extra configuration needed, this should help us convert the triggering of a backdoor into a crash.

All of that being said: how can we load such stub libraries to use with AFL++? Can we simply load them for the entire command, running `LD_PRELOAD=./execve-crash.so afl-fuzz ...`? Why? Why not?

Question 4.5

Is there actually a backdoor in this version of Sudo? If so, what is it?

Further reading (and hacking)

- **The Fuzzing Book** (https://www.fuzzingbook.org/): a comprehensive in-depth guide about learning how fuzzers work by building them.
- **The Rust Fuzz Book** (https://rust-fuzz.github.io/book/introduction.html): fuzzing can also be used to uncover implementation bugs in memory-safe languages such as Rust.
- **The LibAFL Book** (https://aflplus.plus/libafl-book/): LibAFL is a re-implementation of all the best parts of AFL++, allowing users to build custom fuzzers by slotting together building blocks and/or implementing their own.
- **The AFL++ documentation** (https://aflplus.plus/): containing lots of details about things we did not cover in this lab (e.g., parallel fuzzing, grammar fuzzing, binary-only fuzzing).
- **OSS Fuzz** (https://google.github.io/oss-fuzz/): one of the most important fuzzing initiatives for critical Open Source projects.
- **pwn.college** (https://pwn.college/): a website where you can learn about and practice on binary vulnerabilities.
- **Sudo vulnerability walkthrough** by LiveOverflow (https://www.youtube.com/playlist?list=PLhixgUqwRTjy0gMuT4C3bmjeZjuNQyqdx): a YouTube series illustrating how to find a real-life vulnerability in Sudo using fuzzing.

---

[5] Aschermann, Cornelius, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In *NDSS*, vol. 19, pp. 1-15. 2019.

[6] https://www.man7.org/linux/man-pages/man2/execve.2.html