#fuzzing #vulnerabilities #bufferoverflow

- **Name**: Iskander Nafikov
- **E-mail**: i.nafikov@innopolis.university
- **GitHub**: https://github.com/iskanred

Task description

Objective:

This lab aims to introduce students to the concept of fuzzing, a dynamic software testing technique used to discover vulnerabilities in software applications. By the end of this lab, students will be able to:

1. Understand the basics of fuzzing and its importance in security testing.
2. Set up a fuzzing environment.
3. Use a fuzzing tool to test a sample application.
4. Analyze the results of a fuzzing test.

Prerequisites:

- Basic understanding of programming (Python/C/C++).
- Familiarity with command-line interfaces.
- Basic knowledge of software vulnerabilities (e.g., buffer overflows, crashes).

Lab Setup

Tools Required:

1. Fuzzing Tool: AFL (American Fuzzy Lop) or libFuzzer
2. Target Application: A simple C program with potential vulnerabilities (provided below).
3. Operating System: Linux (Ubuntu recommended).

Setup Instructions:

1. Install AFLs:

```
sudo apt update
sudo apt install afl
```

2. Create a Target Application: Save the following C code as `vulnerable.c`:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void vulnerable_function(char *input) {
    char buffer[100];
    strcpy(buffer, input); // Potential buffer overflow
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        FILE *file = fopen(argv[1], "r");

        if (!file) {
            perror("Error opening file");
            return 1;
        }

        char input[1024];

        size_t bytesRead = fread(input, 1, sizeof(input) - 1, file);

        fclose(file);

        input[bytesRead] = '\0';
        vulnerable_function(input);
    } else {
        printf("Usage: %s \<filename>\n", argv[0]);
    }

    return 0;
}
```

3. Compile the Target Application with AFL:

```
afl-gcc -o vulnerable vulnerable.c
```

Lab steps

Step 1: Understand the Target Application

Task description

- Review the `vulnerable.c code` and identify the potential vulnerability (buffer overflow).

- **Buffer Overflow**
  1. The `buffer` array is declared with a fixed size of 100 characters.
  2. `strcpy` is used to copy data from `input` to `buffer` without any boundary checks.
  3. If `input` contains more than 99 characters (plus the null terminator, making it 100), the `strcpy` function will overflow the allocated memory space of `buffer`, writing beyond its bounds.

Task description

- Discuss why this vulnerability is dangerous and how it can be exploited.

Why is is dangerous?

It is dangerous because it's hard to detect for a developer and, while there are many possible attacks may be done using this vulnerability:

- **Integrity issues**
  - When a buffer overflow occurs, it can overwrite adjacent memory locations in the stack. This can corrupt local variables, control structures, and even the return address of the function.
- **RCE**
  - If an attacker can manipulate the input in such a way that it writes a specific payload into the overflow space, they may be able to control the execution flow of the program.

For instance, by overwriting the return address to point to specially createdd code that resides in the overflowed area, the attacker can execute arbitrary code.

- **DoS**
  - Buffer overflows can lead to crashes or unexpected behaviour of the application, resulting in a denial of service. If, for example, the overflows lead to memory corruption that cannot be gracefully handled, the application might terminate unexpectedly.

How it can be exploited?

Here is a general idea of how an attacker might exploit this vulnerability:

1. **Input Manipulation**:
   - The attacker crafts an input string that is longer than 100 bytes, including both payload and specific instructions meant to influence how the program executes (such as modifying the return address).
2. **Payload Insertion**:
   - The crafted input could contain:
     Here's how an attacker might exploit the buffer overflow vulnerability in the provided code, along with a breakdown of the process:

- **Example**:

```
./vulnerable
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xDE\xAD\xBE\xEF
```

  - `A`: The attacker fills the buffer with 103 `A`.
  - `\xDE\xAD\xBE\xEF`: This is an example of what might represent the address of the attacker's shellcode, which could have been inserted right before the overwrite, in a portion of the stack that the input writes to.

Step 2: Run the Fuzzer

1

Task description

Create an input directory for AFL:

```
mkdir input
echo "seed" > input/seed.txt
```

- I created a new `seed.txt` file

```
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ mkdir input
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ echo "seed" > input/seed.txt
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ls
input  vulnerable  vulnerable.c
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ cat input/seed.txt
seed
iskanred@lenovo:~/Study/iu-sd-course/lab-03$
```

2

Task description

Run AFL on the target application:

```
afl-fuzz -i input -o output ./vulnerable @@
```

- And started AFL

```
        american fuzzy lop ++4.00c {default} (./vulnerable) [fast]
┌─ process timing ────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 16 sec │  cycles done : 1      │
│   last new find : none yet (odd, check syntax!) │ corpus count : 1      │
│ last saved crash : none seen yet    │ saved crashes : 0     │
│  last saved hang : none seen yet    │  saved hangs : 0      │
├─ cycle progress ────────────────────┼─ map coverage ────────┤
│  now processing : 0.5 (0.0%)        │   map density : 0.00% / 0.00% │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple │
├─ stage progress ────────────────────┼─ findings in depth ───┤
│  now trying : havoc                 │ favored items : 1 (100.00%) │
│ stage execs : 786/1175 (66.89%)     │  new edges on : 1 (100.00%) │
│ total execs : 5752                  │ total crashes : 0 (0 saved) │
│  exec speed : 347.6/sec             │  total tmouts : 0 (0 saved) │
├─ fuzzing strategy yields ───────────┴─ item geometry ──────┤
│   bit flips : disabled (default, enable with -D) │    levels : 1         │
│  byte flips : disabled (default, enable with -D) │   pending : 0         │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 0         │
│  known ints : disabled (default, enable with -D) │ own finds : 0         │
│  dictionary : n/a                   │  imported : 0         │
│havoc/splice : 0/4956, 0/0           │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused │                       │
│    trim/eff : 20.00%/1, disabled    │                       │
└─────────────────────────────────────┘                       [cpu000: 25%]
iskanred@lenovo:/sys/devices/system/cpu$
iskanred@lenovo:/sys/devices/system/cpu$
iskanred@lenovo:/sys/devices/system/cpu$
iskanred@lenovo:/sys/devices/system/cpu$
```

- However, at this moment I realised that something goes wrong because there were no crashes or hangs at all!
- I found that the code was wrong because it must read from the file and not from the command line arguments directly, so I got the updated version of a source code and ran AFL again

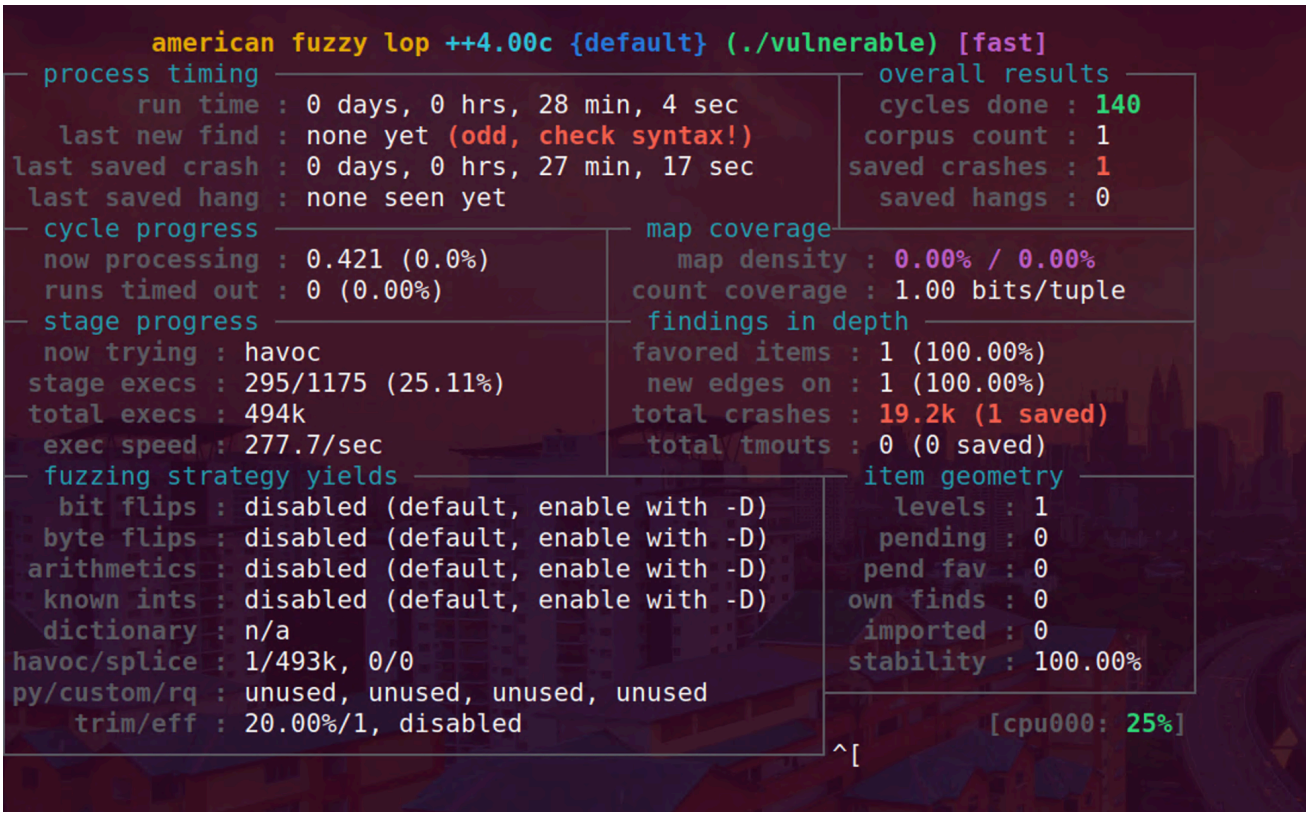
Step 3: Monitor the Fuzzing Process

Task description

Observe the AFL interface, which provides real-time statistics such as:

- **Total Executions**: Number of test cases executed.
- **Unique Crashes**: Number of unique crashes found.
- **Hangs**: Number of timeouts or hangs.
- Let the fuzzer run for 5-10 minutes.

- I made almost 30 minutes fuzzing and stopped it



- We can see that there were 19.2k crashes in total, but only 1 unique that was saved

Step 4: Analyze the Results

1

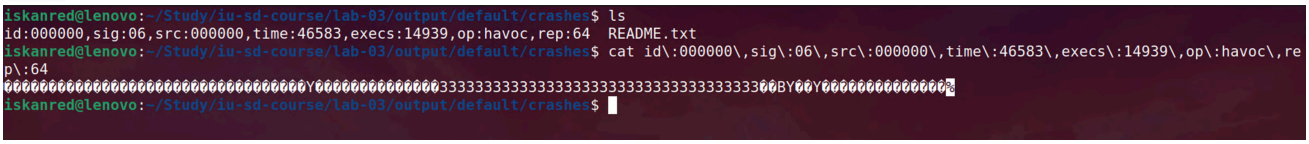
Task description

- After stopping the fuzzer, check the output directory:

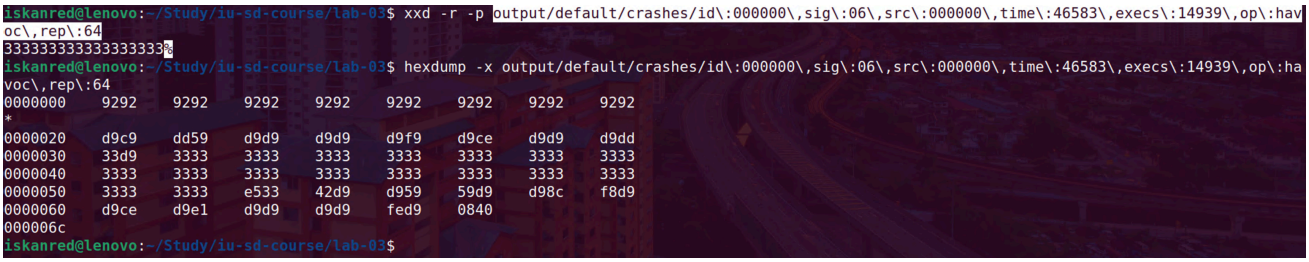
```
ls output/crashes
```

- Look for files that caused crashes (e.g., `id:000000`, `sig:11`).

- There were the file `id\:000000\,sig\:06\,src\:000000\,time\:46583\,execs\:14939\,op\:havoc\,rep\:64`



- We see that this file actually contains some bytes



2

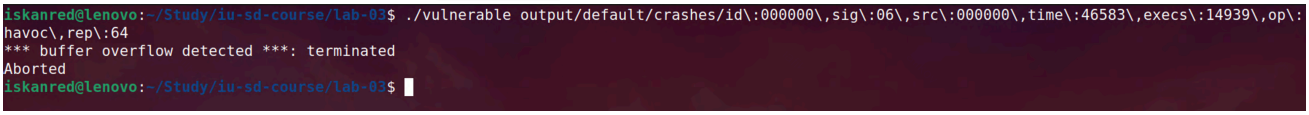
Task description

- Replay the crash:

```
./vulnerable output/crashes/id:000000,sig:11
```

- Observe the crash and discuss why it occurred.

- I replayed the crash using the `afl-gcc` compiled binary:



- And using the original `gcc` compiled binary

```
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ gcc -o vulnerable-original vulnerable.c
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./
input/              output/             vulnerable          vulnerable-original
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./vulnerable-original
Usage: ./vulnerable-original <filename>
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./vulnerable-original output/default/crashes/id\:000000\,sig\:06\,src\:000000\,time\:46583\,execs\:14
939\,op\:havoc\,rep\:64
*** stack smashing detected ***: terminated
Aborted
iskanred@lenovo:~/Study/iu-sd-course/lab-03$
```

- We can see that binary that was compiled by `afl-gcc` is more precise displaying the cause of error: buffer overflow
- At the same time a simple compiled binary displayed `stack smashing` error message.

> Stack Smashing here is actually caused due to a protection mechanism used by gcc to detect buffer overflow errors. The compiler, (in this case gcc) adds protection variables (called canaries) which have known values. An input string of size greater than 10 causes corruption of this variable resulting in `SIGABRT` to terminate the program.

- Let's try to recompile the program without this protection using `fno-stack-protector` option

```
iskanred@lenovo: ~/Study/iu-sd-course/lab-03
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ gcc -o vulnerable-original -fno-stack-protector vulnerable.c
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./vulnerable-original output/default/crashes/id\:000000\,sig\:06\,src\:000000\,time\:46583\,execs\:14
939\,op\:havoc\,rep\:64
iskanred@lenovo:~/Study/iu-sd-course/lab-03$
```

- We now see that the program was successfully executed **with an exploit** done.

Step 5: Fix the Vulnerability

Task description

- Modify the vulnerable.c code to fix the buffer overflow (e.g., use `strncpy` instead of `strcpy`).
- Recompile and rerun the fuzzer to verify the fix.

- I fixed the code changing `strcpy` to more safe `strncpy`

```
c #include <stdio.h> #include <string.h> #include <stdlib.h> void vulnerable_function(char *input) { char buffer[100]; strncpy(buffer, input, sizeof(buffer) - 1); // <-- here is the change } int main(int argc, char *argv[]) { if (argc > 1) { FILE *file = fopen(argv[1], "r"); if (!file) { perror("Error opening file"); return 1; } char input[1024]; size_t bytesRead = fread(input, 1, sizeof(input) - 1, file); fclose(file); input[bytesRead] = '\0'; vulnerable_function(input); } else { printf("Usage: %s <filename>\n", argv[0]); } return 0; }
```

- Here are the results of executing the program that was compiled using default `gcc` compiler

```
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ gcc -o vulnerable-original vulnerable.c
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./vulnerable-original output/default/crashes/id\:000000\,sig\:06\,src\:000000\,time\:46583\,execs\:14
939\,op\:havoc\,rep\:64
iskanred@lenovo:~/Study/iu-sd-course/lab-03$
```

- The results is the same for the binary compiled by `afl-gcc`

```
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ afl-gcc -o vulnerable vulnerable.c
afl-cc++4.00c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: GCC-GCC
[!] WARNING: You are using outdated instrumentation, install LLVM and/or gcc-plugin and use afl-clang-fast/afl-clang-lto/afl-gcc-fast instead!
afl-as++4.00c by Michal Zalewski
[+] Instrumented 8 locations (64-bit, non-hardened mode, ratio 100%).
iskanred@lenovo:~/Study/iu-sd-course/lab-03$ ./vulnerable output/default/crashes/id\:000000\,sig\:06\,src\:000000\,time\:46583\,execs\:14939\,op\:
havoc\,rep\:64
iskanred@lenovo:~/Study/iu-sd-course/lab-03$
```

- Finally, we can run the fuzzing and check its results after the change

```
        american fuzzy lop ++4.00c {default} (./vulnerable) [fast]
┌─ process timing ────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 12 min, 15 sec │  cycles done : 63     │
│   last new find : none yet (odd, check syntax!) │ corpus count : 1      │
│ last saved crash : none seen yet    │ saved crashes : 0     │
│  last saved hang : none seen yet    │  saved hangs : 0      │
├─ cycle progress ────────────────────┼─ map coverage ────────┤
│  now processing : 0.191 (0.0%)      │   map density : 0.00% / 0.00%  │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple │
├─ stage progress ────────────────────┼─ findings in depth ───┤
│  now trying : havoc                 │ favored items : 1 (100.00%)    │
│ stage execs : 880/1175 (74.89%)     │  new edges on : 1 (100.00%)    │
│ total execs : 224k                  │ total crashes : 0 (0 saved)    │
│  exec speed : 297.8/sec             │  total tmouts : 0 (0 saved)    │
├─ fuzzing strategy yields ───────────┴─ item geometry ──────┤
│   bit flips : disabled (default, enable with -D)  │   levels : 1          │
│  byte flips : disabled (default, enable with -D)  │  pending : 0          │
│ arithmetics : disabled (default, enable with -D)  │ pend fav : 0          │
│  known ints : disabled (default, enable with -D)  │ own finds : 0         │
│  dictionary : n/a                   │  imported : 0         │
│havoc/splice : 0/223k, 0/0           │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused      ├───────────────────────┘
│    trim/eff : 20.00%/1, disabled    │                [cpu000: 25%]
└─────────────────────────────────────┘
```

- We see that after 10 minutes there were no crashed found at all!

Questions

1

> **Task description**
>
> What is the purpose of fuzzing in software security testing?

Fuzzing is a software testing technique that involves providing a program with a large volume of randomly generated, malformed, or unexpected inputs to discover vulnerabilities and flaws in its behavior. The main purposes of fuzzing in software security testing include

- identifying security vulnerabilities such as buffer overflows and injection flaws,

- increasing software reliability by exposing hidden bugs, and stress-testing applications by simulating real-world attack scenarios.

Fuzzing can be automated, making it a cost-effective way to perform extensive testing without human intervention. It helps improve the robustness of software by allowing developers to detect issues early in the development cycle, ultimately enhancing overall security and stability.

2

> **Task description**
>
> How does AFL generate test cases to find vulnerabilities?

AFL generates test cases to find vulnerabilities using a **coverage-guided fuzzing** approach.

1. It starts with a set of valid input files and applies mutation techniques to create new test cases by altering existing inputs.
2. During execution, AFL monitors code coverage to track which parts of the program are exercised.
3. This feedback informs the fuzzer, allowing it to prioritise mutations that explore previously untested paths, thereby generating more effective test cases.
4. By intelligently focusing on areas of the code that have not been exercised, AFL increases the likelihood of discovering vulnerabilities.

3

> **Task description**
>
> What other types of vulnerabilities can fuzzing detect besides buffer overflows?

- **Use-after-free**: These occur when a program continues to use memory after it has been freed, potentially leading to arbitrary code execution.
- **Null pointer dereferences**: Attempting to access data through a pointer that has not been initialized can lead to crashes or unintended behavior.
- **Out-of-bounds accesses**: This occurs when a program reads or writes outside the allocated memory for an array or buffer, which can corrupt data or lead to crashes.
- **Integer overflows/underflows**: These happen when arithmetic operations exceed the maximum or minimum representable value, leading to unexpected behaviour.

- **Resource leaks**: Fuzzing can identify cases where resources like file handles or memory are improperly managed, leading to exhaustion and application failures.
- **Logic flaws**: By sending unexpected or malformed inputs, fuzzing may reveal logical errors in code that can be exploited, even if they don't lead to crashes.

4

Task description

How can you improve the efficiency of a fuzzing campaign?

1. **Targeted Fuzzing**: Focus on specific components or code areas that are critical or historically prone to vulnerabilities.
2. **Reduce Execution Time**: Optimise the target application for faster execution by minimising unnecessary processing or using lightweight input formats, allowing the fuzzer to test more cases in less time.
3. **Efficient Input Generation**: Employ smarter heuristics for input generation, such as minimising input sizes and focusing on valid inputs that are likely to uncover bugs rather than purely random data.
4. **Parallel Fuzzing**: Run multiple instances of the fuzzer in parallel across different machines or cores to increase the throughput of test cases executed.
5. **Automated Analysis**: Implement automated tools for crash analysis and reporting to quickly identify and triage new vulnerabilities, reducing manual overhead.