

# Fuzzing for Software Security Testing

## AKOR JACOB TERUNGWA

🕒 Created	@February 21, 2025 8:22 PM
☑ Attendance Required	<input type="checkbox"/>

### 1. Install AFL:

AFL is a popular fuzzing tool that uses genetic algorithms to generate test cases. Here's how i installed AFL (American Fuzzy Lop)

**First, i updated the package list:**

- `sudo apt update`

I then **Installed AFL by using the command:**

- `sudo apt install afl`

**I also verified the installation:**

- `afl-fuzz -h`

```
jakes@jakes-virtual-machine:~$ sudo apt update
[sudo] password for jakes:
Hit:1 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:2 http://de.archive.ubuntu.com/ubuntu jammy InRelease
Hit:3 http://de.archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:4 http://de.archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
111 packages can be upgraded. Run 'apt list --upgradable' to see them.
jakes@jakes-virtual-machine:~$ sudo apt install afl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
afl is already the newest version (4.00c-1ubuntu1).
The following packages were automatically installed and are no longer required:
  libwpe-1.0-1 libwpebackend-fdo-1.0-1
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 111 not upgraded.
jakes@jakes-virtual-machine:~$ afl-fuzz -h
afl-fuzz++4.00c based on afl by Michal Zalewski and a large online community
```

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

- I created a file and saved the code in a file named `vulnerable.c`

```

GNU nano 6.2 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

For better performance and more efficient fuzzing, i considered using:

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

### Step 2: Run the Fuzzer

I created an input directory and added a seed file and ran AFL

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

```

american fuzzy lop ++4.00c {default} (./vulnerable) [fast]
┌──────────┴──────────┐
┌ process timing ───────────┐
│ run time : 0 days, 0 hrs, 4 min, 17 sec │
│ last new find : none yet (odd, check syntax!) │
│ last saved crash : 0 days, 0 hrs, 4 min, 16 sec │
│ last saved hang : none seen yet │
└──────────┴──────────┘
┌ cycle progress ───────────┐
│ now processing : 0*31 (0.0%) │
│ runs timed out : 3 (50.00%) │
└──────────┴──────────┘
┌ stage progress ───────────┐
│ now trying : havoc │
│ stage execs : 978/1024 (95.51%) │
│ total execs : 85.1k │
│ exec speed : 317.3/sec │
└──────────┴──────────┘
┌ fuzzing strategy yields ───────────┐
│ bit flips : disabled (default, enable with -D) │
│ byte flips : disabled (default, enable with -D) │
│ arithmetics : disabled (default, enable with -D) │
│ known ints : disabled (default, enable with -D) │
│ dictionary : n/a │
│ havoc/splice : 0/32.5k, 1/51.6k │
│ py/custom/rq : unused, unused, unused │
│ trim/eff : 48.39%/6, disabled │
└──────────┴──────────┘
┌──────────┴──────────┐
┌ overall results ───────────┐
│ cycles done : 20 │
│ corpus count : 6 │
│ saved crashes : 1 │
│ saved hangs : 0 │
└──────────┴──────────┘
┌ map coverage ───────────┐
│ map density : 0.00% / 0.00% │
│ count coverage : 1.00 bits/tuple │
└──────────┴──────────┘
┌ findings in depth ───────────┐
│ favored items : 1 (16.67%) │
│ new edges on : 1 (16.67%) │
│ total crashes : 17.4k (1 saved) │
│ total tmouts : 0 (0 saved) │
└──────────┴──────────┘
┌ item geometry ───────────┐
│ levels : 1 │
│ pending : 0 │
│ pend fav : 0 │
│ own finds : 0 │
│ imported : 0 │
│ stability : 100.00% │
└──────────┴──────────┘
[cpu000: 50%]

```

## Step 4: Analyze the Results

1. I checked the Output Directory:

```
jakes@jakes-virtual-machine:~$ cd output
jakes@jakes-virtual-machine:~/output$ ls
default
jakes@jakes-virtual-machine:~/output$ cd default
jakes@jakes-virtual-machine:~/output/default$ ls
cmdline  crashes  fuzz_bitmap  fuzzer_setup  fuzzer_stats  hangs  plot_data  queue
jakes@jakes-virtual-machine:~/output/default$ cd crashes
jakes@jakes-virtual-machine:~/output/default/crashes$ ls
id:000000,sig:06,src:000000+000003,time:796,execs:325,op:splice,rep:8  README.txt
jakes@jakes-virtual-machine:~/output/default/crashes$
```

## 2. I replayed the Crash using a command:

- `./vulnerable`  
`output/default/crashes/id:000000,sig:06,src:000000+000003,time:796,execs:325,op:splice,rep:8`

```
jakes@jakes-virtual-machine:~$ ./vulnerable output/default/crashes/id:000000,sig:06,src:000000+000003,time:796,execs:325,op:splice,rep:8
*** buffer overflow detected ***: terminated
Aborted
jakes@jakes-virtual-machine:~$
```

The crash occurred due to a **buffer overflow** vulnerability.

## Step 5: Fix the Vulnerability

To fix the vulnerability, i modified the code to replace `strcpy` with `strncpy` to prevent buffer overflow:

- `strncpy(buffer, input, sizeof(buffer) - 1);`
- `buffer[sizeof(buffer) - 1] = '\0';`

```
GNU nano 6.2 vulnerable.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void vulnerable_function(char *input) {
    char buffer[100];
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination
}

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;
}
```

I run the command below to removed the vulnerable files:

- `rm -rf vulnerable`

Then i recompiled and rerun the Fuzzer

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

As seen below in the output, there are no crashes which indicates that the vulnerability has been fixed

```

jakes@jakes-virtual-machine: ~
american fuzzy lop ++4.00c (default) (./vulnerable) [fast]
process timing      overall results
run time : 0 days, 0 hrs, 7 min, 29 sec  cycles done : 148
last new find : none yet (odd, check syntax!)  corpus count : 6
last saved crash : none seen yet  saved crashes : 0
last saved hang : none seen yet  saved hangs : 0
cycle progress
now processing : 0*230 (0.0%)  map coverage
runs timed out : 0 (0.00%)  map density : 0.00% / 0.00%
stage progress  count coverage : 1.00 bits/tuple
now trying : havoc  findings in depth
stage execs : 270/768 (35.16%)  favored items : 1 (16.67%)
total execs : 189k  new edges on : 1 (16.67%)
exec speed : 307.0/sec  total crashes : 0 (0 saved)
total touts : 2 (2 saved)
fuzzing strategy yields
bit flips : disabled (default, enable with -D)  lten geometry
byte flips : disabled (default, enable with -D)  levels : 1
arithmetics : disabled (default, enable with -D)  pending : 0
known ints : disabled (default, enable with -D)  pend fav : 0
dictionary : n/a  own finds : 0
havoc/splice : 0/187k, 0/1284  imported : 0
py/custom/rq : unused, unused, unused, unused  stability : 100.00%
trin/eff : 93.33%/29, disabled  [cpu000:100%]

```

## QUESTIONS:

### 1. What is the purpose of fuzzing in software security testing?

- Fuzzing is used to discover vulnerabilities, crashes, and unexpected behavior in software by feeding it random or semi-random inputs.

### 2. How does AFL generate test cases to find vulnerabilities?

- AFL uses genetic algorithms to mutate input seeds and generate new test cases. It prioritizes inputs that trigger new code paths or crashes.

### 3. What other types of vulnerabilities can fuzzing detect besides buffer overflows?

- Fuzzing can detect:
  - Memory leaks.
  - Use-after-free vulnerabilities.
  - Integer overflows.
  - Format string vulnerabilities.
  - Logic errors.

### 4. How can you improve the efficiency of a fuzzing campaign?

- Use high-quality seed inputs.
- Combine fuzzing with static analysis.
- Use parallel fuzzing.
- Implement coverage-guided fuzzing.