Binary Exploitation of Figlet Unix Binary

João Freitas up202100373
Rui Gonçalves up202103077

The present report, written in the context of TPAS (Teoria e Prática de Ataques de Segurança) curricular unit at FCUP (Faculdade de Ciências da Universidade do Porto), aims at describing the process of attempting to find bugs and crashes in a command-line binary for Unix-like systems. It will start by describing the goals of the project, methodologies being followed, approaches to take and technologies to look on. Then, it will be presented the binary in context of analysis, depicting initial weakness and the reasons to why it was chosen as the target binary. Following this, the working environment and tools used for exploitation will be described in detail, focusing on the quirks and constraints felt during the setup phase. The fifth chapter describes the fuzzing process and achieved results in an iterative manner. To conclude, an overview of the achieved results will be presented, as well as some remarks for future work.

1. Terminologies, Methodologies, Technologies and Goals

Binary exploitation can be defined as the process of "finding a vulnerability in the program and exploiting it to gain control of a shell or modifying the program's functions" [1]. To gain control of a shell, typically newbie attackers inject shellcode in the program through an input stream (e.g., user input(stdin)), which allows for arbitrary code execution (i.e., using a program as the interpreter of crafted binary code). Shellcode is a special type of code produced with assembly instructions, for a specific CPU architecture [2].

Despite shellcode attacks still being popular and effective for a wide range of use cases, more advanced attackers rely on *Return Oriented Programming* (ROP), a technique that exploits the program call stack by hijacking the memory and aligning the assembly instructions present on it to execute code in an arbitrary way [3]. This technique is way more effective on modern age binaries, as these are protected with specific security flags on compile time (e.g., NX, ASLR, Stack Canary) that detect if whether a program was compromised or not during runtime, via a memory corruption [3].

But how do attackers find vulnerabilities that allow arbitrary code execution?

To craft code and execute it in a target program, we first need to load it in the program memory so it can be arbitrarily triggered and executed by the program. Since modern operating systems reserve specific memory spaces for a program, no entity can write to this memory in an external manner. So, to write in the program memory, it is needed to *conduct the program to do it so*. This is where the vulnerabilities take place: there is a need to find usage of unsafe operations in a binary, so that specific input or program flow can corrupt the reserved memory. Classic vulnerabilities include:

- **Buffer Overflow**, where data bigger than the actual buffer (i.e., memory space) is written to it, which causes this data to be propagated in the memory, until a crash occurs [4];
- **Use After Free (UAF)**, which occurs when a specific chunk of memory is used after it was deallocated/freed [4, 5];
- **Format Strings**, which occurs when programs that use operations that rely on format strings (e.g., `printf` and `sprintf`) are vulnerable to format operations injection, which will force format strings functions to look for arguments to replace on the format position [4, 5].

```
char buf[8];

void vulnerable() {
    gets(buf);
}
```

Code Snippet 1 - Example of a buffer overflow. The `gets` function does not check for the buffer length, so any input with length greater than **8**, will cause a buffer overflow. Source: [5].

One very important thing to note, is that memory corruptions are far more easy to occur in "low level" programming languages such as `C` and `C++`, rather in high level ones (e.g., `Javascript`, `C#`), as the former allows for direct dynamic memory allocation and does not have memory optimizers, such as *garbage collection*. This gives an hint on the programs to choose for trying to find memory vulnerabilities.

Okay, this is great and seems easy. Can I hack the government with binary exploitation?

While theoretically you sure can exploit any binary to arbitrarily execute code, these kind of attacks have been disclosed since the late 80s, and so there is already a lot progress on detecting memory vulnerabilities in software, both during compile and runtime. So most likely you cannot exploit any binary by just slapping fingers, but that does not mean that exploiting the binary is impossible!

The state-of-the-art for analyzing and exploiting binaries divides these processes in two approaches: the **white-box** approach, where one has access to the source-code of the binary and as such can find vulnerabilities in a static analysis manner (i.e., read code and find bugs); and the **black-box** approach, which assumes that the binary is a black-box that no one knows how it is built, giving input to it and expecting to receive a specific output. One can argue that white-box analysis is the best approach to follow, as an experienced security researcher or programmer knows every weakness of a programming language. However, access to the source-code is not always available. Usually, the community relies on the black-box approach through an automated process called ***Fuzzing***.

1.1 What is Fuzzing?

OWASP defines fuzzing as a "black box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated

fashion" [6]. It is very similar to property-based testing (PBT), where by producing random input and applying mutations to customize it, the program is faced with inputs it may have not been considering, leading to unexpected or undesired behaviour.

Fuzzing is an automatic process, which makes finding bugs an exciting process! Security researchers can just leave their machines *fuzzing* the binary during the night and analyze the collected results during the day. By default, it is efficient. However, it does not grant effectiveness, as it is still the job of the researcher to prepare the fuzzing *campaign* and guide it.

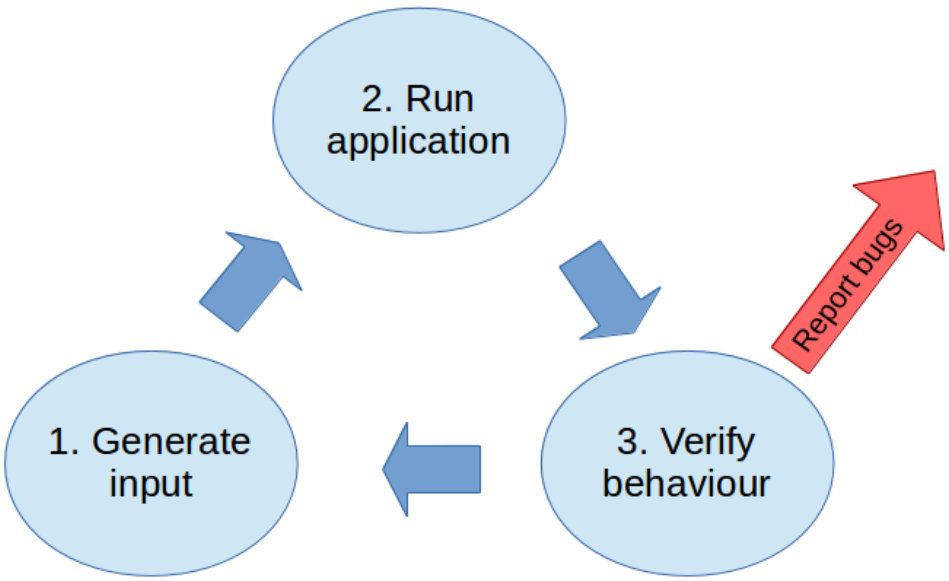

Figure 1 - Fuzzing simplified in a state diagram. You can't get a better illustrative description than this! Source: [7].

1.2 What is the trend for fuzzing technologies today?

There are different open-source tools available for fuzzing binaries, each with their own quirks and advantages. The Google fuzzing team highlights the following as the most performant ones [8], based on the benchmarks provided by the `FuzzBench` [9] tool:

- `libFuzzer`, an imperative fuzzer engine which interprets fuzzing tests written in `C` and runs them [8, 10]. libFuzzer focuses on maximizing code coverage [10], which means that it is only useful if the researcher has knowledge about the underlying source-code;
- `Honggfuzz`, a fuzzer recognized for its performance and customization options [8], serving as an alternative where defining fuzzing tests is not possible [11]. Not only it supports software, but also hardware fuzzing;
- `American Fuzzy Lop (afl)`, a guided fuzzing engine that is fast and widely recognized in the fuzzing community [8]. AFL major advantage is the fact that it is guided (i.e., fuzzer learns and reinforces the input based on crash results) [12].

The authors of the report understand that some of these fancy fuzzing terminologies are very confusing, so a glossary defined by the Google Fuzzing team [12] is provided in Appendix A.

1.3 What are the Project Goals?

The overall goal and purpose of the present project, was to get introduced in binary exploitation and try to get a crash in a target binary/program. To complement this goal, the report authors also desired to get a **CVE** (Common Vulnerability Exposure) for the found crash, meaning that the crash would require to have a pratical **PoV** (Proof of Vulnerability) to back the legitimacy and impact of the crash.

As for the report goals, the authors would like to provide a clear and descriptive framework for binary exploitation, so that newbie security researchers can understand the processes to take with easy, as well as to get their hands dirty in the shortest time.

2. Choosing the binary to exploit

Choosing a target binary to exploit, is as difficult as choosing a nickname in an online video-game: there are so many possibilities - ones more difficult than the others -. which provide a higher satisfaction of working on and getting results. It was suggested by the professor assisting the project, that the binary being picked falls under the following categories:

- Linux tools, as the majority of these are open-source and widely used;
- Video-games, because although the increase of complexity, it would be very rewarding to get a crash and at the same time be rewarded in a bug-bounty program.

From the beggining we know that we want to get a crash in order to exploit and possibly get a CVE, which is far more easy on Linux tools. But looking to video-games, getting a crash is way more rewarding than a rather simple Linux tool, altough that these involve way more complex codebases, meaning that it is far more difficult to find a crash.

Since this project is being developed in the context of the Information Security Masters @ FCUP, TPAS course, there are four more courses which the report authors are enrolled in, meaning that the available time and resources to provide on the project is scarce. Having this constraint in mind, and also taking in consideration that both authors are "working-students", we decided to stick with the easiest path, navigating towards the Linux tools path.

There are thousands of Linux tools, which shall be picked?

When searching for target Linux binaries, the authors had some extra constraints in consideration: we want to publish a write-up of our findings (the present report) publicly and tackle some software that is used everyday by Linux users. This means that the binary must be open-source (to avoid law violations) and have some reputation in the Linux or open-source community. Additionally, the binary should also be written in `C` or `C++` to ensure that memory safety can be exploited.

We have then embarked in a journey of finding a tool that best fits our requirements. Of all the tools reviewed, the following three caught our attention the most:

- `jq`, a command-line JSON processor, widely used by developers, taking as input [14];
- `sudo`, the classic tool to leverage user permissions to root, shipped in almost every Linux distribution and macOS [15];
- `figlet`, a tool that takes input from stdin and outputs a "formatted art message", according to a specific font [16].

The `jq` and `figlet` tools are very similar in terms on how these behave: they take some user input, process it, and spit some output. `sudo` is particularly interesting for the use case of gaining access to a root shell, as its goal is to run programs with root permissions.

There are many benefits in choosing each of the presented tools, however we can only choose one, so it must be a wise choice. Finding a crash in the `sudo` binary would be a pretty neat challenge, not only because it is the most widely used tool of the presented ones, but it is also a reference for fuzzing by security researchers [17]. The latest crash in `sudo` (or more precisely `sudoedit`) was found in 2021, having the **CVE-2021-3156** assigned to it [17, 18].

It is pretty safe to say that finding a crash in sudo will be hard, so let's focus on the remaining tools. The decision between whether picking `jq` or `figlet`, comes down to the hardness constraint. Upon the time of writing this report, it seems only a few people have tried and published the results for fuzzing figlet, while jq already has some publised write-ups [19, 20] (note: we used the keywords "fuzzing figlet" and "fuzzing jq" as the search query in Google, to find these results). We know that we should not judge a book by its cover (in this context, it relates to the hardness of finding a bug if someone has fuzzed or found one yet), but it seemed that figlet is the best tool to pick, as we would be sort of pioneers in its fuzzing. In addition, the figlet codebase can be summarized in one file of `~1.8k SLOC` (Single Lines of Code), meaning that it is not a complex binary, while jq codebase is far more bigger.

3. Choosing the research approach and tools

It is known that `figlet` will be the binary of study, but before getting our hands dirty, we need to establish our working model. Shall we follow the white-box or black-box approach? Figlet is open-source and so white-box is possible, but automating the crash finding seems way more fun and interesting.

If we follow the black-box approach, which tool should we adopt?

Previously in Chapter 1, we have seen the following fuzzing tools: `libFuzz`, `HongFuzz` and `AFL`. At first glance, using `libFuzzer` seems a good idea, as we have an extensive support for different kind of configurations via a test definition. However, we cannot forget that this fuzzer focuses on checking code coverage, like in unit tests, and so it might impact the research taking place. The fuzzers `AFL` and `HongFuzz` are both very performant, but the guided property of AFL is what makes it special. If we believe that the binary will behave differently on a certain input, we can sure guide AFL to fuzz it and learn from the results. It seems clear that choosing AFL will provide the best results, but that does not mean that HongFuzz is not good, as both have their quirks. AFL also

has a variant called `AFL++`, which used instead of the original AFL. We will talk more about it in Chapter 4.

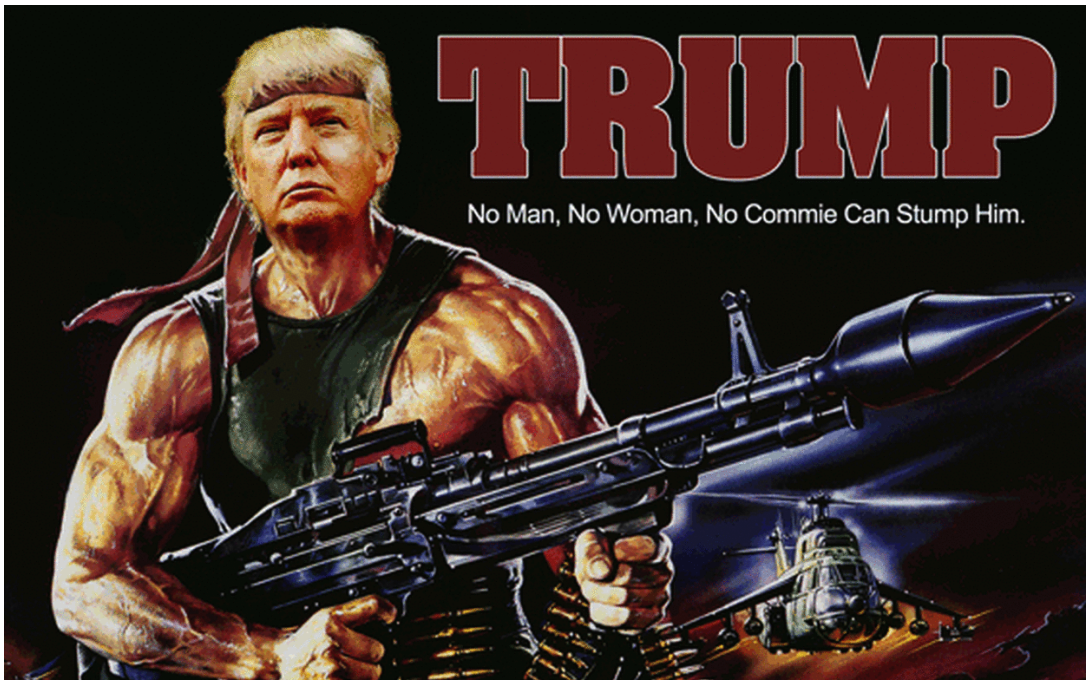

Figure 2 - AFL stands for "American Fuzzy Lop", or as we like to call it "American Fuzileiros", because you are essentially constantly firing rounds of input to the binary (This picture is obviously satire). Source: [21].

Although starting on with a black-box approach, we have later adopted an hybrid approach, by also performing static analysis of Figlet source-code. The reasons for this strategy will be explained later in Chapter 5.

4. Fuzzing setup

In this chapter we will focus on all the setup we did in order to perform our research using AFL. Let's first start by talking about the working/deployment environment, as it is a crucial part for the success of the fuzzing campaign.

Running `AFL` means executing binaries **hundreds**, **thousands** or even **million** times per second, which requires a lot of processes, CPU schedule and memory allocation - now that's a lot of work. This means that running AFL on our machines will reduce their lifespan, so we need to resort to other forms of computation. Nowadays, running or renting computing power on the cloud has never been easier, so that's what we opted to do. The `Google Cloud Platform` (GCP) is a cloud service provider by Google, which offers a **$300** credit for new users who would like to try out their services, during a period of 90 days [22]. The only thing you need to do is create a Google account (which most likely you already have for your e-mail) and provide some account info, so Google can verify that you are not a bot and bill you once the trial ends. We recognize that the billing process might be a little scary, but we have an hack for you: If you have a Revolut [23] account (which is free), you can create a temporary digital card and use it when signing up in GCP.

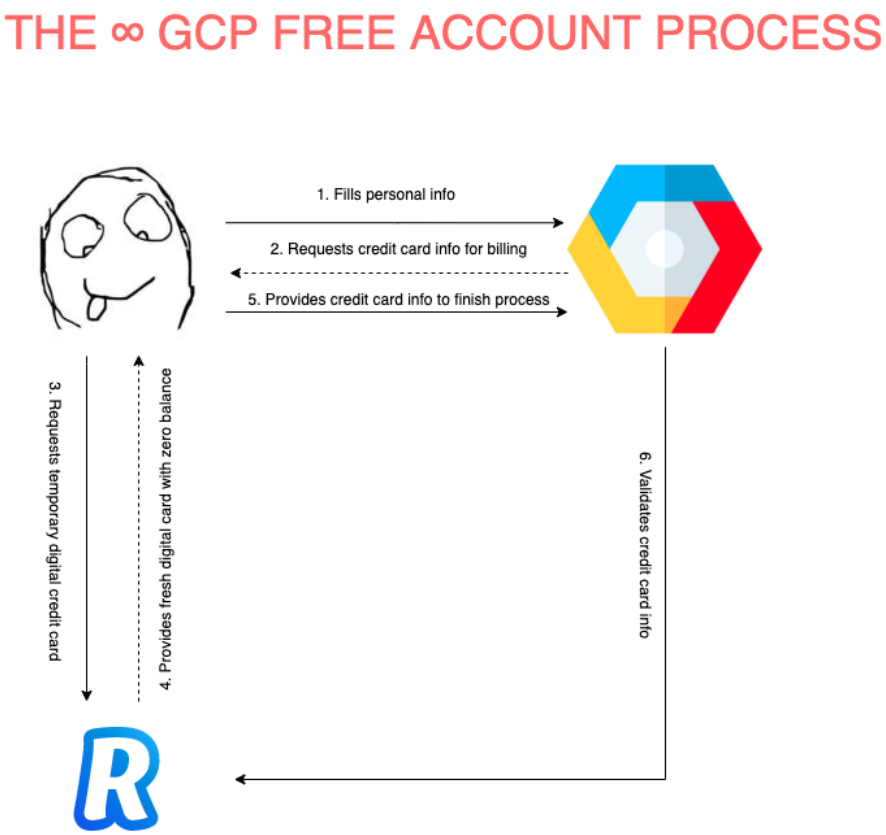

Figure 3 - The ∞ GCP free account hack. Take in consideration that creating as many accounts as you want is not ethical.

Since we are two researchers, we have created two accounts to get the most of the computing power.

GCP offers a lot of cloud services, but the one we must focus is on *Compute Engine* [24]. This service allows to create **Virtual Machine** (VM) instances on the spot, with custom CPU and Memory configurations.

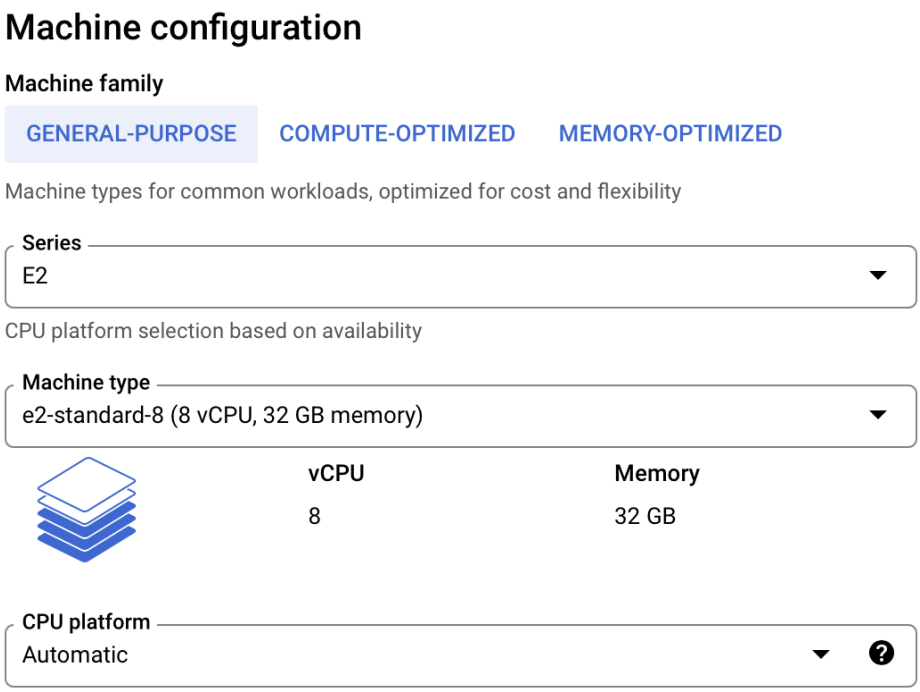

Figure 4 - You can customize your virtual machine towards a CPU/Memory optimization or pick a general purpose one. Source: [24].

For our particular scenario, we need as many computation power as possible. We also need multiple cores, as AFL is single-core execution. Knowing these requirements, we ended up choosing the `e2-highcpu-8` configuration, which provides a VM with `8 Virtual CPU` and `8GB RAM` (approx. cost **$164** per month).

As for the operating system of the VM, we chose a clean `Debian GNU/Linux 10 (buster)` image, installed on a `60 GB SSD`. Choosing Debian and not other Debian based distribution is important, as it will remove any *bloat* dependencies that may impact the overall system performance (and thus AFL performance).

Cool, we have free computing power. How do we setup AFL now?

As previously mentioned, we have majorly used `AFL++`, a variant of AFL that is more friendly to newbies and includes some improvements on the performance, instrumentation and input mutations. Setting up AFL++ can be done in two ways:

The **Docker** approach, where the main image on AFL++ needs to be pulled, installed and runned (3 commands). This is the safest approach when installing AFL++ in some Linux distributions, as Docker will guarantee that it is installed correctly and be able to execute.

```
docker pull aflplusplus/aflplusplus
docker run -ti -v /location/of/your/target:/src aflplusplus/aflplusplus
```

Code Snippet 2 - Sequence of commands required to install AFL++ using Docker. Source: [26].

The "**bare metal**" approach, where a couple of system dependencies like `llvm`, `libstd` and `python3` need to be installed. This approach is not as safe as the Docker one, as Linux is not know for its safest dependency managers, so most likely something will break.

```
sudo apt-get update
sudo apt-get install -y build-essential python3-dev automake cmake git flex bison libg
# try to install llvm 11 and install the distro default if that fails
sudo apt-get install -y lld-11 llvm-11 llvm-11-dev clang-11 || sudo apt-get install -y
sudo apt-get install -y gcc-$(gcc --version|head -n1|sed 's/.* //'|sed 's/\..*//')-plu
sudo apt-get install -y ninja-build # for QEMU mode
git clone https://github.com/AFLplusplus/AFLplusplus
cd AFLplusplus
make distrib
sudo make install
```

Code Snippet 3 - Sequence of commands required to install AFL++ via APT (bare metal approach). Source: [26].

If you follow the bare metal approach, make sure that after instalation, the `afl-fuzz` binaries are linked in the system, otherwise you will have to type the path to the respective binary each time you desire to run AFL++. If for some reason the linking of the binaries failed, you can manually do this either by a symlink or just pointing AFL directory in the `PATH` variable.

Initially the authors have setup AFL++ using Docker, as it is guaranteed that the setup will not fail. However, we have later switched to the bare metal approach, for two reasons: convenience and performance. If we use Docker, then we need to map **Docker Volumes** to a specific folder within the operating system, which requires that everything we do is under that folder. This is not very convenient, as we like to move files all around the place and be able to execute AFL in the shortest time possible. Additionally, Docker provides a couple of abstraction layers within the operating system, to make sure that the image can be "emulated", which impacts the performance of the fuzzing campaign.

If you still prefer to use AFL and not AFL++, we suggest that you use APT to install it, as it will link everything with no issues:

```
sudo apt-get update
sudo apt-get install afl
```

Code Snippet 4 - Sequence of commands required to install AFL via APT (bare metal approach).

Sweet! I've installed AFL on my machine, how do I know that everything is working?

AFL provides a couple of tools to make sure that everything is connected correctly, which we will talk in a bit. Additionally, just like in any programming language, it is recommended that you create an "Hello World" to make sure that AFL is working as expected!

To know that AFL is up and running we suggest that you run the following command: `afl-gotcpu`. The purpose of this command is to detect how many and which cores are still available for a new AFL execution. If upon running this command nothing broke, you are good to go.

Before heading towards a real world binary, it is first needed to get the hands dirty with an example binary that we know is buggy. For this, we suggest that you create a file named `main.c`, that contains the following code:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("What's your name?\n");

    char name[8];

    gets(name);
```

```
    printf("Your name is: %s\n", name);

    return 0;
}
```

Code Snippet 5 - Source-code for the vulnerable AFL Hello World program.

Then, create a file named `INSTALL` (this isn't a requirement, it is more to have a convenient way to building things) with the following content:

```
CC=afl-cc

$CC main.c -o main
```

Code Snippet 6 - Contents of `INSTALL` file. CC is the alias for the compiler that we will use to instrument our binary (in this case the default one of AFL, which is a mimic of `gcc`).

If you run the `INSTALL` file, the `main` binary compiled file will be created, already instrumented with AFL compilers.

We have previously mentioned that AFL is a guided fuzzer. This means that you will have to provide some initial input to AFL, so that he knows where to start in the bitstring generation and mutation process. Create the file `names` and place it under `input/`, with the following content:

```
Rute
João
John
Mary
Trump
Biden
```

Code Snippet 7 - Contents of `names` file.

Alright! You have everything ready for the hello-world fuzzing campaign, now you just need to start it, which can be done with the following command: `afl-fuzz -i input/ -o output/ -- ./main`

The following is taking as input the `names` file, specified under the `input/` directory and specifies the outputs folder as `output/`. The `--` flag indicates that the binary needs to read input from stdin. If the binary requires a file as input, instead of -- it would be necessary to use `@@` in the specific binary argument (e.g., ./main -f @@).

If you run the command, in a few seconds you get a couple of crashes, which means that AFL is working properly and finding crashes! If you head to the `outputs` folder, you can find all information collected by AFL. The `crashes` folder contains all the relevant crashes noted by AFL. The filename of the crash input may seem odd at first glance, but they are written in this way to

form a signature of the crash. The most imporant part of the filename is the `sig` property, which indicates the signal code received by the operating system when the program crashed. In the case of the hello-world binary, we get a `SIGSEGV` signal (signal 11), which translates to `Segmentation Violation` (aka Segmentation Fault).

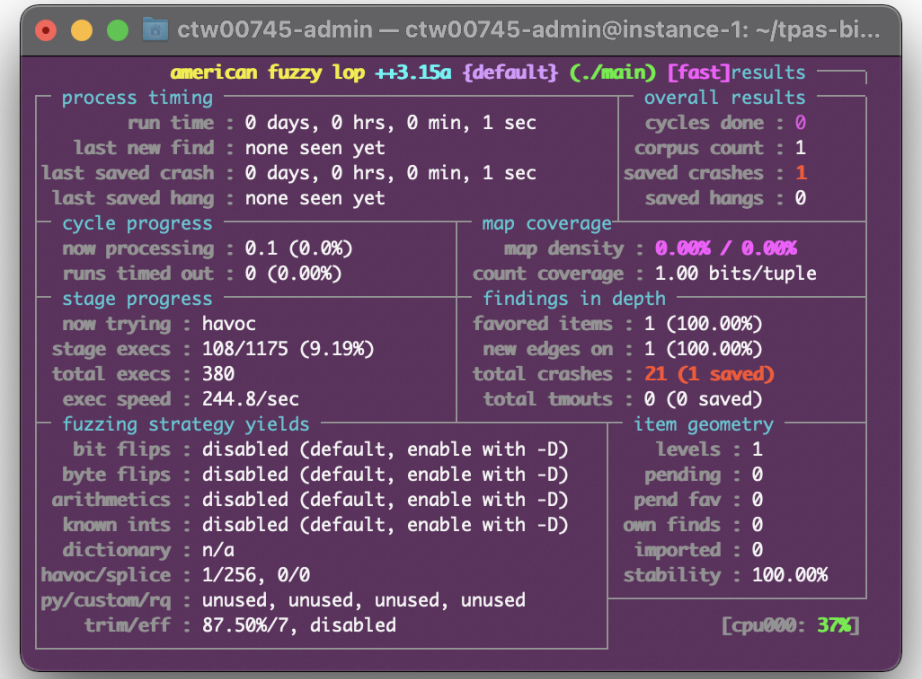

Figure 5 - AFL provides a real-time overview of the fuzzing campaign.

Wooohooo, I got my first crash! But the fuzzing seems kinda slow... Why is it only using 25% of the CPU?

AFL tries to use everything at his power to maximize the performance of the fuzzing campaign, however, just like any other program, it is restricted to the operating system. To increase AFL performance, researchers need to lower the security of their computers, so that some kernel checks do not need to be done, as well as to modify the way program processes are forked [27, 28]. There are a couple of steps that one can do in order to maximize the performance, however we would like to highlight the following, as they have increased the performance of our fuzzing in at least **12x** times:

- Run the `afl-system-config` tool after reboot. This tool reconfigures the system security and options, in order to increase execution speed [27];
- Set `AFL_TESTCACHE_SIZE` environment variable to a number between **400-500**. This variable indicates the cache size to be saved in the primary memory (RAM) for test cases, reducing the ammount of times the disk is used to persist memory;
- Use `afl-clang-fast` instrumentation mode to make use of persistent mode [28]. This mode is a game changer for AFL and AFL mantainers even refer to it as the way "professionals do

fuzzing" [28], since it increases the speed of fuzzing between 10-20x. We will not describe every detail of how persistent mode works, but it solves the bottleneck of fuzzing, the needed resources to allocate for a new program execution. In persistent mode, the binary source-code is updated so that the function being fuzzed (typically the main, program entrypoint), is called inside a loop until a crash is found. This way, the program does not need resources to be scheduled for every execution, as it is needed only one for the program that will run the loop! Figure 6 proves that with persistent mode enabled, the executions per second of the Hello World binary incrased from `~3700` to `~21000`.

- Last but not least, shutdown the VM after each 8h of execution, in order to "refresh" our CPU cores. Since we are running on the cloud, if we shutdown the VM, after some time this machine will be allocated to a different customer, meaning that we get a new VM at a different Google Datacenter.

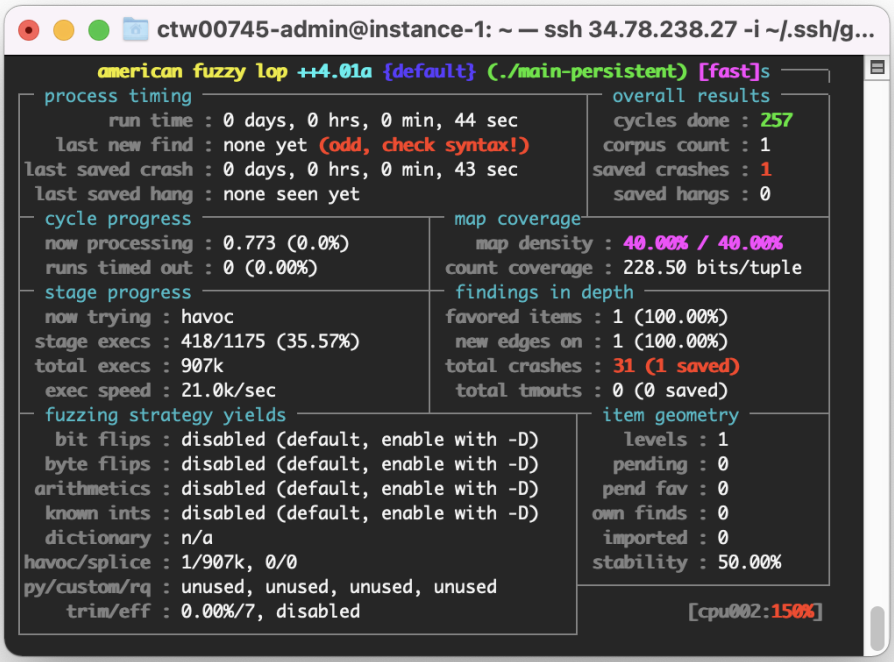

Figure 6 - Fuzzing of Hello World binary with persistent mode enabled.

Previously we have seen that the `afl-gotcpu` tool discovers which cores are still available for execution. We use this tool to spawn new AFL instances in the form of Master-Slave. This is achieved by adding the `-M <master_id>` and `-S <slave_id>` flags in the `afl-fuzz` binary.

What are AFL instrumentation modes?

For a binary to exist, one must have compiled the program source-code with a general purpose compiler like `gcc`. In AFL, programs need to be compiled with specific compilers, so that AFL can add special checks in the program and thus recognize how the program is behaving towards its input. These compilers are called instrumenters and different instrumenters provide different fuzzing results. In the Hello World example above, you have compiled the binary using the default

instrumenter: `afl-cc`, which like `gcc`, serves for general purposes. If for example you want to compile the binary using `LLVM` infrastructure, you have `afl-clang`, which differs from `afl-cc` in each owns quirks [29].

Additionally, you can also fuzz a binary that has not been compiled with a specific instrumentation mode, however AFL will not be able to understand where it is breaking and how it broke. This of course brings many disadvantages, mainly the fuzzing campaign performance decay. However, if you managed to find a crash without using a instrumentation mode, it is guaranteed that you have found a bug in the target binary, as you broke the original binary and not the instrumented one.

5. Fuzzing Figlet and Results Analysis

As previously mentioned, this chapter will describe the process of fuzzing figlet in an iterative manner, as we believe this is the best way the reader can learn from our mistakes. We would also like to note that, even though we have provided a description on how to fuzz with AFL, the first iterations did not cover some of the tips that we provide in this document. This is due to a try-error learning approach being adopted, in order to be able to learn AFL and fuzzing, as well to obtain results at the same time.

Default AFL Instrumentation Mode + Basic Input

Our first iteration was very naive and we did not expect many results from it. To test our AFL works, we used an input file with the string `abc`. For AFL to properly fuzz the binary, we had to change the way this last one is compiled. Luckily, `figlet` has a convenient way of compilation, as it provides a `Makefile` with C program build conventions. All it was needed to do was to change the variables `CC`, `CCX` and `LD` to `afl-cc`, `afl-c++` and `afl-cc`, respectively. After some minor compiling issues, we managed to setup our first fuzzing campaign ever, using the following command:

```
afl-fuzz -i figlet-in/ -o figlet-out/ -- figlet/figlet -d -
```

We were blown away by the fuzzing speed, but a bit odd regarding the feedback of AFL UI. We had no crashes and AFL was always on `havoc` stage. Since we had no experience with AFL, we thought this was something normal. With the help of the `screen` [30] command, we let AFL run for over 20h, executing more than 1B times. Despite so many executions, AFL was still stuck on havoc stage and figlet did not crash even once.

Figure 7 - In the first iteration, AFL was reporting `~3700` executions/sec, yielding > **1B** iterations in 20 hours.

We were a bit sad about the results, as we were certain that the fuzzing campaign was not properly setup. With the help of `AFL_DEBUG` environment variable and setting this one with value **1**, we were able to see what each figlet execution was outputting:

```
figlet: standard: Unable to open font file
figlet: standard: Unable to open font file
...
figlet: standard: Unable to open font file
figlet: standard: Unable to open font file
```

Code Snippet 8 - Output of figlet in the first iteration.

Figlet wasn't locating the default font files directory, neither was able to locate a file to use as font file... For some reason we set the arguments "-d -", even though that the param `-d` specifies a font directory in Figlet (facepalm).

Correcting usage of Figlet arguments

For the second iteration, we used the same input as before `abc`, but now we have corrected the way arguments are specified by AFL to Figlet. AFL provides the *macro* `AFL_INIT_ARGV`, that can be introduced in the binary main function, overriding the program parameters (in case of the `C` programming language, it overrides `argc` and `argv`). Additionally, we also changed the constant `DEFAULTFONTDIR` in Figlet source-code, so it points directly to where we had our Figlet fonts installed (in this case `/usr/share/figlet`).

```
... original header include definitions

#include "../AFLplusplus/utils/argv_fuzzing/argv-fuzz-inl.h"

... program source-code

int main(argc,argv)
int argc;
char *argv[];
{

    AFL_INIT_ARGV();

    ... remaining main function source-code
```

Code Snippet 9 - Introduction of AFL_INIT_ARGV in a `C` program

We have then compiled and rerun AFL with the following command:

```
afl-fuzz -i figlet-in/ -o figlet-out/ -- figlet/figlet
```

Notice that this time we did not specific where AFL input should be used in Figlet arguments. This is due to the macro overriding the arguments, so introducing any arguments will not have any effect. The crashes output is also different, as its scope will not be for a specific argument (e.g., message to print), but instead represents the arguments used to run the program.

With the macro introduction we started getting crashes! Unfortunately these crashes are false positives, as by running each of one with the instrumented figlet binary it crashes, but on the original binary, the program does not crash.

```
root@instance-1:/tpas-binary-exploitation# cat m1-take-2/crashes/id:000000,sig:11,src:
bash: warning: command substitution: ignored null byte in input
Segmentation fault (core dumped)

root@instance-1:/tpas-binary-exploitation# cat m1-take-2/crashes/id:000000,sig:11,src:
bash: warning: command substitution: ignored null byte in input
     _   _                 _
 ___(_) (_) __ _ ___  __| |
/ __| | | |/ _` / __|/ _` |
\__ \ |_| | (_| \__ \ (_| |
|___/\__, |\__,_|___/\__,_|
     |___/
```

Code Snippet 10 - Example of output crash when using instrumented figlet, but no crash on original figlet.

Something has got to be wrong with our instrumented binary. Maybe the instrumentation mode? Let's try compiling it with `afl-clang`. BOOM! Same crash. If it is not the instrumentation mode,

then it has to be something that we introduced in the code, but the only changes were the hardcoded font directory and macro definitions... Let's use `GDB` + `GEF` to inspect the stacktrace of the instrumented binary crash.

```
root@instance-1:/tpas-binary-exploitation# gdb figlet/figlet
> gef run < figlet-out/m1-take-2/crashes/id:000000,sig:11,src:000000,time:402,execs:57

[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────────────
$rax   : 0x00000000d5b780  →  0x0000000000000000
$rbx   : 0x00000000cf9d00  →  0x0000000000000000
...

[#0] Id 1, Name: "figlet", stopped 0x7ffff7d94fb7 in __strrchr_avx2 (), reason: SIGSEG
──────────────────────────────────────────────────────────────────────────────────────
[#0] 0x7ffff7d94fb7 → __strrchr_avx2()
[#1] 0x427b9c → getparams()
[#2] 0x42d0ac → main(argc=<optimized out>, argv=<optimized out>)
```

Code Snippet 11 - Stacktrace of instrumented figlet binary crash.

It seems the program is crashing in the `getparams` function of Figlet, which parses the program parameters, defining them as global variables. After manually tracing the `argv` array, we found out that the first position of the array was filled with a `NULL` string. It is strange for this argument to be NULL, as the first argument of a binary indicates the path of the binary in execution. If this value is set to NULL, it later creates conflict in the getparams function, specifically in the following condition:

```
void getparams()
{
    ...
    if ((myname = strrchr(Myargv[0],DIRSEP))!=NULL) {
        myname++;
    }
    else {
        myname = Myargv[0];
    }
    ...
}
```

Code Snippet 12 - figlet program name retrieval in the `getparams` function.

So indeed we got a crash in figlet... Although it is a false positive, one thing is for certain, the original figlet binary does not provide a NULL check for this argument. From this we can conclude that figlet is vulnerable on this function, but getting a crash seems highly impossible as the underlying OS always specificies the path of the executing binary as the first argument.

To provide a fix for this scenario, it is needed to include the following condition on the figlet main, which will stop the main program flow if the first argument value is found to be `NULL`:

```
int main(argc, argv)
int argc;
char *argv[];
{
    if (argv[0] == NULL)
    {
        return -1;
    }
    ...
}
```

Code Snippet 13 - Preventing figlet to crash if the first argument is set to `NULL`.

Refining Fuzzing Campaign

Iteration #3:

After the fixes had been applied, we have update the inputs so our fuzzing campaign starts getting a bit more professional. The first new input focuses on non ASCII characters and large text inputs. We the help of Python, we have generated a bitstring that contains the first 500 bytes in order to include a large set of non ASCII characters.

```
import os

with open("large-input", "w") as f:
    bytes = ""
    for c in range(0, 500):
        bytes += chr(c)
    f.write(bytes)
    f.close()
```

Code Snippet 14 - Python script to produce a bit string with the first 500 bytes (`generate_bitstring.py`).

We set AFL to run with this new input and it sure did crash, a lot. For each AFL slave, AFL saved about **30** crash results. We started manually testing random crash inputs with the instrumented and original figlet binary, and once again no crashes were detected on the latter, despite crashing in the former. At this point we thought that maybe there was some coding issue introduced by us, but we have also thought that maybe there is a crash input that crashes the original binary. However, manually testing each input will take a lot of time... This calls for an automation script to written:

```
#!/bin/bash

function test_crashes() {
    crashes_saved=$1
    figlet_path=$2

    crashes_found=0

    echo "Testing crashes with binary: $figlet_path ..."

    for crash_path in $crashes_saved
    do
        cat $crash_path | exec $figlet_path > /dev/null 2>&1
        if [[ $? -ne 0 ]]
        then
            crashes_found=$(($crashes_found+1))

            echo "Crash input crashed binary: $crash_path"
        fi
    done

    echo 'Summary:'
    echo "Crashes found: $crashes_found"
}

crash_dir=$1
original_figlet_path=$2
instrumented_figlet_path=$3

crashes_saved="$(find $crash_dir -name 'id:*,sig:*')"

test_crashes "$crashes_saved" $instrumented_figlet_path
test_crashes "$crashes_saved" $original_figlet_paths
```

Code Snippet 15 - Bash script that tests the crashed results inputs again instrumented and original figlet binaries (`test_crashes.bash`).

After running the script, we found out that the original figlet binary did not crash even once.

In the meantime we've also upgraded our AFL++ version to 4.01a, and discovered that using persistent mode with `afl-clang-fast` instrumentation, our fuzzing speed increased in 10x (we were getting constant `~300` exec/sec, improving to `~3000` exec/sec). To include the persistent mode in Figlet, we had to update its source-code with the following:

```
int main(argc, argv)
int argc;
char *argv[];
{
```

```
#ifdef __AFL_HAVE_MANUAL_CONTROL
  __AFL_INIT();
#endif

  while (__AFL_LOOP(10000))
  {
    unsigned char *buf = __AFL_FUZZ_TESTCASE_BUF;
    char *args[argc + 1];

    for (int i = 0; i < argc; i++)
    {
      args[i] = argv[i];
    }

    args[argc] = buf;

    main_original(argc + 1, args);
  }
}

int main_original(argc, argv)
int argc;
char *argv[];
{
    if (argv[0] == NULL)
    {
        return -1;
    }
    ...
}
```

Code Snippet 16 - Modification of `figlet.c` source, so persistent mode is enabled.

Such an improvement with tool and instrumentation mode update!

After thinking about new approaches for a crash, we tried to crash figlet from a side input source: the input font files. For this we had to remove the previous `AFL_INIT_ARGV` macro to resort to a manual arguments definition and use the `@@` argument for Figlet to pass the source input file, instead of its content. We copied three font files that are bundled with Figlet source, and modified two to include random characters in order to break Figlet. We then launched AFL with the following command:

```
afl-fuzz -i figlet-in/fonts/ -o figlet-out/ figlet/figlet -f @@
```

Additionally, we also tried to mess with AFL and Figlet parameters, so more randomness and different test cases were produced and tested:

For AFL:

- `-D` flag, which enables bit flipping and other sort of bitstring production operations, to increase randomness of input;
- `-g` flag, to specify the minimum length which inputs should be produced. This was extremely useful font font file fuzzing;
- `-G` flag, to specify the maximum length which input should be produced. (Same as -g but to limit the input length).

For Figlet:

- `-w`, which specifies the width of the output print (e.g., if -w = 5, then each print line shows at most 5 characters).

Unfortunately, still no crashes after a lot of fuzzing!

Blind Fuzzing the Original Figlet Binary

Our final attempt at blackbox analysis was to fuzz the original figlet binary, as getting a crash here guarantees that figlet does crash. The only downside of this is that AFL loses the guided property, and is now blind to how the binary reacts to certain inputs. Additionally, since the binary is not instrumented, it is not possible to take advantage of persistent mode, which blazingly speeds up the fuzzing process.

Using the inputs previously described, we setup 8 fuzzing instances using `AFL++` and `AFL` (4 for each). We wanted to be sure that not getting crashes was not an AFL++ issue, so dividing the analysis seemed fair. Each instance was also setup with different AFL and Figlet flags (the same mentioned above), in order to vary the achieved results. One thing to note about non-instrumented mode, is that the output folder will always be called `default`, as the Master-Slave identifiers are not available in this mode. This means that, to achieve concurrent non-instrumented runs on the same machine, different folders need to be specified, otherwise results will be overwritten.

```
afl-fuzz -n -i figlet-in/messages/ -o figlet-out/afl-ni-1 figlet
afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-2 figlet
afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-3 figlet -w 20
afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-1 figlet -w 20 -f /usr

./afl-original/afl-fuzz -n -i figlet-in/messages/ -o figlet-out/afl-ni-original-1 figl
./afl-original/afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-origina
./afl-original/afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-origina
./afl-original/afl-fuzz -n -g 2500 -i figlet-in/messages/ -o figlet-out/afl-ni-origina
```

Code Snippet 17 - Launching multiple instances of AFL++ and AFL in non-instrumented mode (`-n` is required).

Since we lost some computing power in the AFL division, we let the fuzzers running for 3 weeks (`~22 days`), daily checking for results in these. As you may expect from previous results, unfortunately both AFL versions did not catch any crash. One particular issue that we would like to

highlight on non-instrumented fuzzing, is that for some reason it is not possible to use the `afl-whatsup` tool check the campaign status. Instead we had to use `grep` to check for results:

```
grep -r 'saved_crashes' figlet-out/
```

White-box Analysis

After an intense fight on getting a crash with black-box analysis, we have decided to move forward and perform a white-box analysis of the source-code. We could either do this manually or automatically, using a tool. We decided to stick with the tool, since tools these days have a strong background knowledge of low-level programming languages like `C`, that we do not have.

We selected the `Flawfinder` tool as it was suggested by the advisor professor. This tool is specific for `C/C++` programs, and it analysis its source-code and reports possible security weaknesses for a certain risk level [33]. It can be installed using `pip`:

```
pip install flawfinder
```

To then analyse source-code, `flawfinder` binary just needs to be invoked:

```
flawfinder figlet/
```

Flawfinder comes bundled with a database of vulnerable functions and patterns, which it uses to find and classify the risk of the analysing source-code, on the scale of `0-5`. These functions can vary from:

- Input data and string operations: `strcpy`, `strcat`, `gets`, `sprintf` and `scanf`;
- String formatting and logging: `printf`, `snprintf` and `syslog`;
- I/O and Race Conditions: `access`, `chown`, `chgrp`, `chmod`, `tmpfile`, `tmpnam` and `mktemp`;
- Shell access : `exec`, `system` and `popen`;
- Random and bitstring generators: `random`.

A cool feature of Flawfinder is that it identifies the respective `CWE` (Common Weak Enumeration) of each vulnerability, next to the function definition.

We ran `flawfinder` for figlet source-code and several problems were identified, which can be found extensively in Appendix B. Several high risk vulnerabilities (3-4 level) that were identified are related to buffer overflows, which is an indicator that figlet can be exploited. However, after manually checking these, it was found that proper validations were being applied, reducing the likely of being vulnerable. Additionally, low to medium risk vulnerabilities (1-2 level) were also analyzed and in the researchers opinion, the source-code is not vulnerable.

6. Final Remarks and Future Improvements

Despite not finding a proper crash and thus try to hijack input to exploit Figlet, we enjoyed fuzzing this binary, as it enabled us to dive into the world of fuzzing and binary exploitation, as well as to

improve our critical thinking on finding new approaches to break software. We would also like to highlight that, we indeed did find a vulnerability in Figlet (`NULL` reference as the binary path), however, from our knowledge it is not feasible to exploit it.

As for future improvements, there are some remarks we also like to highlight, as they seemed interesting while learning more about AFL and fuzzing in general:

- Test different fuzzing engines: if you are a future researcher, do not stick only to a single fuzzing engine (e.g., AFL). Of course first choose the one which seems the most appropriate for the job, but go on and try others. Even better, setup an *automation* that switches between engines periodically!;
- Test different instrumenting targets: we have fuzzed using the default `gcc` instrumentation mode, `afl-cc`, as well as `LLVM` instrumentation modes `afl-clang` and `afl-clang-fast`. However there are more instrumentation modes available, that can be further explored. For example, `QEMU/FRIDA` mode can be used to fuzz non-instrumented binaries, despite not having the possibility to compile these [31];
- Use more CPU focused virtual machines on GCP: we used virtual machines from GCP that are specific for general purposes, however GCP offers CPU focused instances that might speed up the fuzzing process, as AFL needs more CPU than RAM or other resources;
- Try distributed fuzzing!: with AFL you can fuzz in parallel and in a distributed manner. The latter approach could be interesting to setup in a multiple machines environment, as the crashes and instrumentation information is synced across all machines, and so AFL can learn from the results of other machines [27];
- Host fuzzing on OSS-Fuzz: this is a Google open-source fuzzing infrastructure project, self-proclaiming as "continuous fuzzing for open source software" [32]. Researches can either setup their own fuzzing infrastructure or schedule builds on Googles infrastructure;
- Keep your AFL version updated: updating from `3.15a` to `4.01a` provided major improvements on the fuzzing speed;
- Fuzz specific functions instead of the whole program: if you have access to the binary source-code, you can setup AFL to fuzz a specific function and not the whole program! This also increases the fuzzing speed by a lot;
- Use AFL dictionaries: By default AFL uses a general-purpose dictionary, which comes with a set of words for compact data formats, like images and document languages [12]. If your binary speaks or works in a specific language, it might be useful to setup a dictionary that recognizes certain keyworks and patterns. This is extremely useful for fuzzing DSL (Domain Specific Language) parsers.

References

[1] CTF101, ‘Overview - CTF 101: Binary Exploitation’, CTF101, 2019 https://ctf101.org/binary-exploitation/overview/(accessed Feb. 8, 2022).

[2] Firewalls.com, ‘Shellcode - Firewalls.com’, Firewalls.com, 2022 https://www.firewalls.com/blog/security-terms/shellcode/#:~:text=Shellcode%20is%20a%20special%20type,control%20of%20the%20affected%20system.(accessed Feb. 8, 2022).

[3] Jonathan Salwan, ‘An introduction to the Return Oriented Programming and ROP chain generation’, 2014 http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf(accessed Feb. 8, 2022).

[4] CTF101, ‘Heap Exploitation - CTF 101’, CTF101, 2019 https://ctf101.org/binary-exploitation/heap-exploitation/(accessed Feb. 8, 2022).

[5] CS161, ‘Memory Safety Vulnerabilities | Computer Security’, textbook.cs161.org, 2022 https://textbook.cs161.org/memory-safety/vulnerabilities.html(accessed Feb. 8, 2022).

[6] OWASP, ‘Fuzzing | OWASP Foundation’, owasp.org, 2022 https://owasp.org/www-community/Fuzzing(accessed Feb. 8, 2022).

[7] Xavi, ‘Fuzzing – Finding bugs using BooFuzz (1/3) | Happy Hacking!’, xavibel.com, 2019 https://xavibel.com/2019/06/23/fuzzing-how-to-find-bugs-using-boofuzz/(accessed Feb. 8, 2022).

[8] google/fuzzing, ‘fuzzing/intro-to-fuzzing.md at master · google/fuzzing’, github.com, 2020 https://github.com/google/fuzzing/blob/master/docs/intro-to-fuzzing.md(accessed Feb. 9, 2022).

[9] FuzzBench, ‘FuzzBench’, fuzzbench.com, 2022 https://www.fuzzbench.com/(accessed Feb. 9, 2022).

[10] llvm, ‘libFuzzer – a library for coverage-guided fuzz testing. — LLVM 15.0.0git documentation’, llvm.org, 2022 https://llvm.org/docs/LibFuzzer.html(accessed Feb. 9, 2022)

[11] Honggfuzz, ‘google/honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based)’, github.com, 2022 https://github.com/google/honggfuzz(accessed Feb. 9, 2022)

[12] google/AFL, ‘google/AFL: american fuzzy lop - a security-oriented fuzzer’, github.com, 2022 https://github.com/google/AFL(accessed Feb. 9, 2022)

[13] google/fuzzing, ‘fuzzing/glossary.md at master · google/fuzzing’, github.com, 2020 https://github.com/google/fuzzing/blob/master/docs/glossary.md(accessed Feb. 9, 2022)

[14] jq, ‘jq is a lightweight and flexible command-line JSON processor.’, stedolan.github.io, 2018 https://stedolan.github.io/jq/(accessed Feb. 9, 2022)

[15] sudo, ‘sudo(8): execute command as another user - Linux man page’, linux.die.net, 2012 https://linux.die.net/man/8/sudo(accessed Feb. 9, 2022)

[16] Figlet, ‘cmatsuoka/figlet: Claudio's FIGlet tree’, github.com, 2020 https://github.com/cmatsuoka/figlet(accessed Feb. 9, 2022)

[17] LiveOverflow, 'How SUDO on Linux was HACKED! // CVE-2021-3156', youtube.com, 2021 https://www.youtube.com/watch?v=TLa2VqcGGEQ(accessed Feb. 10, 2022)

[18] CVE/MITRE, 'CVE - CVE-2021-3156', cve.mitre.org, 2021 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156(accessed Feb. 10, 2022)

[19] David MacIver, 'Use after free bug · Issue #896 · stedolan/jq', github.com, 2015 https://github.com/stedolan/jq/issues/896(accessed Feb. 10, 2022)

[20] Harrison Green, 'Learning About Structure-Aware Fuzzing and Finding JSON Bugs to Boot', forallsecure.com, 2020 https://forallsecure.com/blog/learning-about-structure-aware-fuzzing-and-finding-json-bugs-to-boot(accessed Feb. 10, 2022)

[21] Redneck Nation, 'Trump Rambo Sticker - Redneck Nation', rednecknationstrong.com, 2022 https://rednecknationstrong.com/trump-rambo-sticker/(accessed Feb. 10, 2022)

[22] Google Cloud Platform, 'Cloud Computing Services | Google Cloud', cloud.google.com, 2022 https://cloud.google.com(accessed Feb. 10, 2022)

[23] Revolut, 'Revolut – One app, all things money', revolut.com, 2022 https://www.revolut.com(accessed Feb. 10, 2022)

[24] Google Cloud Platform, 'Compute Engine: Virtual Machines (VMs) | Google Cloud', cloud.google.com, 2022 https://cloud.google.com/compute(accessed Feb. 10, 2022)

[25] google/fuzzing, 'AFL/parallel_fuzzing.txt at master · google/AFL', github.com, 2019 https://github.com/google/AFL/blob/master/docs/parallel_fuzzing.txt(accessed Feb. 10, 2022)

[26] AFLplusplus/AFLplusplus, 'AFLplusplus/INSTALL.md at stable · AFLplusplus/AFLplusplus', github.com, 2022 https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md(accessed Feb. 10, 2022)

[27] AFLplusplus/AFLplusplus, 'AFLplusplus/fuzzing_in_depth.md at stable · AFLplusplus/AFLplusplus', github.com, 2022 https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md(accessed Feb. 11, 2022)

[28] AFLplusplus/AFLplusplus, 'AFLplusplus/README.persistent_mode.md at stable · AFLplusplus/AFLplusplus', github.com, 2022 https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md(accessed Feb. 11, 2022)

[29] AFLplusplus/AFLplusplus, 'AFLplusplus/README.llvm.md at stable · AFLplusplus/AFLplusplus', github.com, 2022 https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.llvm.md(accessed Feb. 11, 2022)

[30] screen, ‘screen(1) - Linux man page’, linux.die.net, 2022 https://linux.die.net/man/1/screen(accessed Feb. 14, 2022)

[31] AFLplusplus/AFLplusplus, ‘AFLplusplus/README.md at stable · AFLplusplus/AFLplusplus’, github.com, 2021 https://github.com/AFLplusplus/AFLplusplus/blob/stable/frida_mode/README.md(accessed Feb. 15, 2022)

[32] google/oss-fuzz, ‘google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software.’, github.com, 2022 https://github.com/google/oss-fuzz(accessed Feb. 15, 2022)

[33] Flawfinder, ‘Flawfinder Home Page’, dwheeler.com, 2022 https://dwheeler.com/flawfinder/(accessed Feb. 15, 2022)

Appendixes

A – Glossary of Fuzzing Terminologies (Adapted: [13])

Glossary

Naming things is hard, so this page tries to reduce confusion around fuzzing-related terminology.

Campaign

The fuzzing journey. Represents the lifecycle of fuzzing a binary: setup, inputs, instrumentation, results, crashes, etc.

Corpus

Or **test corpus**, or **fuzzing corpus**.
A set of test inputs. In most contexts, it refers to a set of minimal test inputs that generate maximal code coverage.

Cross-pollination

The term is taken from botany, where one plant pollinates a plant of another variety. In fuzzing, cross-pollination means using a corpus for one fuzz target to expand a corpus for another fuzz target. For example, if there are two libraries that process the same common data format, it is often benefitial to cross-pollinate their respective corpora.

Dictionary

A file which specifies interesting tokens for a fuzz target. Most fuzzing engines support dictionaries, and will adjust their mutation strategies to process these tokens together.

Fuzz Target

Or **Target Function**, or **Fuzzing Target Function**, or **Fuzzing Entry Point**.
A function to which we apply fuzzing. A specific signature is required for OSS-Fuzz. Examples: openssl, re2, SQLite.

Fuzzer

The most overloaded term and used in a variety of contexts, which makes it bad. Sometimes, "Fuzzer" is referred to a fuzz target, a fuzzing engine, a mutation engine, a test generator or a

fuzzer build.

Fuzzer Build

A build that contains all the fuzz targets for a given project, which is run with a specific fuzzing engine, in a specific build mode (e.g. with enabled/disabled assertions), and optionally combined with a sanitizer. In OSS-Fuzz, it is also known as a job type.

Fuzzing Engine

A tool that tries to find interesting inputs for a fuzz target by executing it. Examples: libFuzzer, AFL, honggfuzz, etc.

See related terms Mutation Engine and Test Generator.

Mutation Engine

A tool that takes a set of testcases as input and creates their mutated versions. It is just a generator and does not feed the mutations to fuzz target. Example: radamsa (a generic test mutator).

Reproducer

Or **Test Case**.
A test input that can be used to reproduce a bug when processed by a fuzz target.

Sanitizer

A dynamic testing tool that can detect bugs during program execution. Examples: ASan, DFSan, LSan, MSan, TSan, UBSan.

Seed Corpus

A small initial corpus prepared with the intent of providing initial coverage for fuzzing. Rather than being created by the fuzzers themselves, seed corpora are often prepared from existing test inputs or may be hand-crafted to provide interesting coverage. They are often checked into source alongside fuzz targets.

Test Generator

A tool that generates testcases from scratch according to some rules or grammar. Examples: csmith (a test generator for C language), cross_fuzz (a cross-document DOM binding test generator).

Test Input

A sequence of bytes that is used as input to a fuzz target. Typically, a test input is stored in a separate file.

B – Figlet flawfinder analysis results

Here are the security scan results from Flawfinder version 2.0.19, (C) 2001-2019 David A. Wheeler.

Number of rules (primarily dangerous function names) in C/C++ ruleset: 222

> Examining figlet.c

Final Results

figlet.c:148: [4] (buffer) wcscpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using a function version that stops copying at the end of the buffer.

```
#define STRCPY(x,y) wcscpy((x),(y))
```

figlet.c:149: [4] (buffer) wcscat: Does not check for buffer overflows when concatenating to destination [MSbanned] (CWE-120).

```
#define STRCAT(x,y) wcscat((x),(y))
```

figlet.c:154: [4] (buffer) strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).

```
#define STRCPY(x,y) strcpy((x),(y))
```

figlet.c:155: [4] (buffer) strcat: Does not check for buffer overflows when concatenating to destination [MSbanned] (CWE-120). Consider using strcat_s, strncat, strlcat, or snprintf (warning: strncat is easily misused).

```
#define STRCAT(x,y) strcat((x),(y))
```

figlet.c:710: [4] (buffer) strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).

```
strcpy(fontpath,fontdirname);
```

figlet.c:713: [4] (buffer) strcat: Does not check for buffer overflows when concatenating to destination [MSbanned] (CWE-120). Consider using strcat_s, strncat, strlcat, or snprintf (warning: strncat is easily misused).

```
strcat(fontpath,name);
```

figlet.c:714: [4] (buffer) strcat: Does not check for buffer overflows when concatenating to destination [MSbanned] (CWE-120). Consider using strcat_s, strncat, strlcat, or snprintf (warning: strncat is easily misused).

```
strcat(fontpath,suffix);
```

figlet.c:718: [4] (buffer) strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).

```
strcpy(fontpath,name);
```

figlet.c:719: [4] (buffer) strcat: Does not check for buffer overflows when concatenating to destination [MSbanned] (CWE-120). Consider using strcat_s, strncat, strlcat, or snprintf (warning: strncat is easily misused).

```
strcat(fontpath,suffix);
```

figlet.c:1153: [4] (buffer) strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).

```
strcpy(outline,templine);
```

figlet.c:927: [3] (buffer) getenv: Environment variables are untrustable input if they can be set by an attacker. They can have any content and length, and the same variable can be set more than once (CWE-807, CWE-20). Check environment variables carefully before using them.

```
env = getenv("FIGLET_FONTDIR");
```

figlet.c:946: [3] (buffer) getopt: Some older implementations do not protect against internal buffer overflows (CWE-120, CWE-20). Check implementation on installation, or limit the size of all string inputs.

```
while ((c = getopt(Myargc,Myargv,"ADEXLRI:xlcrpntvm:w:d:f:C:NFskSWo"))!= -1) {
```

figlet.c:263: [2] (misc) open: Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents? (CWE-362).

```
if ((fd = open("/dev/tty",O_WRONLY))<0) return -1;
```

figlet.c:1020: [2] (integer) atoi: Unless checked, the resulting number can exceed the expected range (CWE-190). If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).

```
infoprint = atoi(optarg);
```

figlet.c:1023: [2] (integer) atoi: Unless checked, the resulting number can exceed the expected range (CWE-190). If source untrusted, check both minimum and maximum, even if the input had no

minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).

```
smushmode = atoi(optarg);
```

figlet.c:1034: [2] (integer) atoi: Unless checked, the resulting number can exceed the expected range (CWE-190). If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).

```
columns = atoi(optarg);
```

figlet.c:1134: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

```
char templine[MAXLEN+1];
```

figlet.c:1187: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

```
char fileline[MAXLEN+1],magicnum[5];
```

figlet.c:1557: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

```
char c[10];
```

figlet.c:1560: [2] (buffer) wchar_t: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

```
wchar_t wc[2];
```

figlet.c:82: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).

```
#define MYSTRLEN(x) ((int)strlen(x)) /* Eliminate ANSI problem */
```

figlet.c:147: [1] (buffer) wcslen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).

```
#define STRLEN(x) wcslen(x)
```

figlet.c:1710: [1] (buffer) getchar: Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).

```
return( getchar() ); /* no: return stdin character */
```

Analysis Summary

- Hits = 23
- Lines analyzed = 2134 in approximately 0.12 seconds (18236 lines/second)
- Physical Source Lines of Code (SLOC) = 1483
- Hits@level = [0] 39 [1] 3 [2] 8 [3] 2 [4] 10 [5] 0
- Hits@level+ = [0+] 62 [1+] 23 [2+] 20 [3+] 12 [4+] 10 [5+] 0
- Hits/KSLOC@level+ = [0+] 41.8071 [1+] 15.5091 [2+] 13.4862 [3+] 8.09171 [4+] 6.74309 [5+] 0
- Minimum risk level = 1

Not every hit is necessarily a security vulnerability. You can inhibit a report by adding a comment in this form: //flawfinder: ignore Make *sure* it's a false positive! You can use the option --neverignore to show these.

There may be other security vulnerabilities; review your code!

See 'Secure Programming HOWTO' (https://dwheeler.com/secure-programs) for more information.