

Fuzzing into a system to reveal software defects and vulnerabilities.

Michael Caram, Jaiven Harris & Monte Scott

Lenoir-Rhyne University

CSC 434: Computer Security

Dr. Ajay Kumara

Requirements

1. Installing Kali Linux and VMware

Installing a virtual machine and an operating system on Windows or Mac before installing AFL (American Fuzzy Lop) ++

Kali Linux:

[Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution](#)

VMware for Mac:

[VMware Workstation Player | VMware](#)

Virtual Box for Windows:

[Downloads – Oracle VM VirtualBox](#)

Objectives

1. Installing AFL++ And Fuzzing a Simple C Program

Installing AFL ++ into Kali linux operating System and making a simple C program for fuzzing vulnerabilities and software defects.

YouTube video of install AFL++ and fuzzing a C Program:

[Y206 d ub H1 T7 \(youtube.com\)](#)

Website for Installing AFL++:

[GitHub - AFLplusplus/AFLplusplus: The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn_mode, and a lot more!](#)

2. How to Fuzz Arm and MIPS Binaries with Qemu and AFL++

<https://www.youtube.com/watch?v=0iyviukkANY&list=PLHGgqcJIME5mQmn9pYGCPrSVactbIwwtP&index=5>

3. How to use CmpLog feature to fuzz a binary

<https://www.youtube.com/watch?v=qZyHphVhMfQ&list=PLHGgqcJIME5mQmn9pYGCPrSVactbIwwtP&index=4>

Goal: To reveal software defects and vulnerabilities in a system by systematically testing a program with various inputs, often including invalid or unexpected data.

Purpose: To get the user to find weakness and reveal software defects in the networks, by fuzzing a C program, fuzzing network applications, and binary numbers.

Fuzzing Tools: AFL++ in Kali Linux, CMPLOG, Qemu

Introduction

What is Fuzzing?

Fuzzing, also known as fuzz testing, is a black box software testing technique and it consists of finding implementations bugs using malformed data injection in an automated fashion. The process involves providing the program with a set of inputs such as unexpected data, or random data, this will explore various execute paths within the program while uncovering bugs and security issues. There are a lot of fuzzing techniques such as random fuzzing, smart fuzzing, and hybrid fuzzing and fuzzing not only finds vulnerabilities and weaknesses, but it is also a guide to help developers in reducing potential attacks of software applications. To fuzz we would need a tool that can fuzz and there are many fuzzing tools such as ffluf, AFL (American fuzzy Lop), Atheris, and more, but the tool we are only using is AFL++.

The History of Fuzzing

Fuzzing began in developing in 1980 by a man named Professor Barton Miller with him and his team developing “bug tools” for software testing. This would make an impact of fuzzing of gaining attention with the development of peach fuzzer and it was used data models in the 1990s. Fast forwarding in the 2000s many people started to design fuzzing tools and AFL was the most popular tool being that it innovates fuzzing techniques.

What is AFL++?

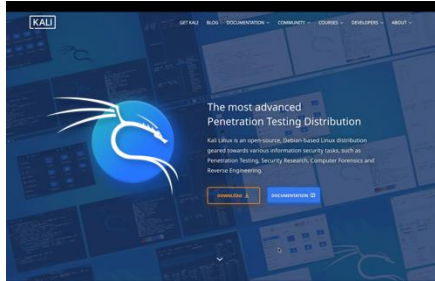
AFL++ is the daughter of the original AFL and was made in 2017 with several enhancements such as performance improvements, fuzzing strategies, stability, and reliability etc. AFL++ performs faster with more efficient fuzzing and introduces new fuzzing strategies such as AFLFast and AFLGo which these are designed to explore various aspects of target applications to improve coverages. AFL++ can be used to test smaller programs like C++ or larger pieces of software like operating systems and monitors for exceptions like crashes or compromised data.

1. Fuzzing A simple C Program

STEP 1: INSTALLING VMWARE AND KALI LINUX

Before we get started, we would need to install a virtual machine and an operating system. A virtual machine is a computer file that behaves like an actual computer and can be installed on mac or windows. The operating system that we are installing is Kali linux and it is an open-source operating system designed for digital forensics and penetration testing. Below are quick videos being how to install Kali linux and Vmware.

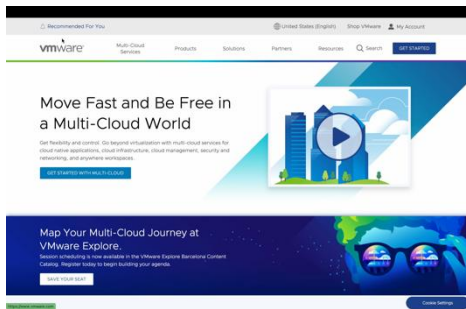
Kali Linux installation:



Website for Kali Linux:

[Kali Linux | Penetration Testing and Ethical Hacking Linux Distribution](https://www.kali.org/)

VMware installation:



Website for VMware:

[VMware Workstation Player | VMware](https://www.vmware.com/)

STEP 2: INSTALLING AFL++

Now that we have kali linux and vmware installed we can now go ahead and install AFL++ in kali Linux. We can now install AFL++ by opening a terminal and typing the commands Sudo apt install AFL++. This will ask for your password that you use to login, and it will install AFL++.

COMMAND MEANING AND STEPS:

1. **COMMAND:** Sudo apt install afl++
2. Sudo is short for 'Super User DO.' If you prefix any command with 'sudo,' it will run that command with superusers or elevated privileges.
3. Advanced package tool, or APT, is a free-software user interface that works with core libraries to handle the installation and removal of software on Debian, and Debian-based Linux distributions.
4. Install Afl++

```
(jaiven@kali)-[~/AFLplusplus]
$ sudo apt install afl++
[sudo] password for jaiven:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  afl++-doc
Suggested packages:
  gnuplot
The following NEW packages will be installed:
  afl++ afl++-doc
0 upgraded, 2 newly installed, 0 to remove and 1278 not upgraded.
Need to get 713 kB of archives.
After this operation, 3745 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://http.kali.org/kali kali-rolling/main arm64 afl++ arm64 4.08c-1 [503 kB]
Get:2 http://http.kali.org/kali kali-rolling/main arm64 afl++-doc all 4.08c-1 [210 kB]
Fetched 713 kB in 1s (918 kB/s)
Selecting previously unselected package afl++.
(Reading database ... 400682 files and directories currently installed.)
Preparing to unpack .../afl++_4.08c-1_arm64.deb ...
Unpacking afl++ (4.08c-1) ...
Selecting previously unselected package afl++-doc.
Preparing to unpack .../afl++-doc_4.08c-1_all.deb ...
Unpacking afl++-doc (4.08c-1) ...
Setting up afl++-doc (4.08c-1) ...
Setting up afl++ (4.08c-1) ...
Processing triggers for man-db (2.12.0-1) ...
Processing triggers for kali-menu (2023.4.6) ...
```

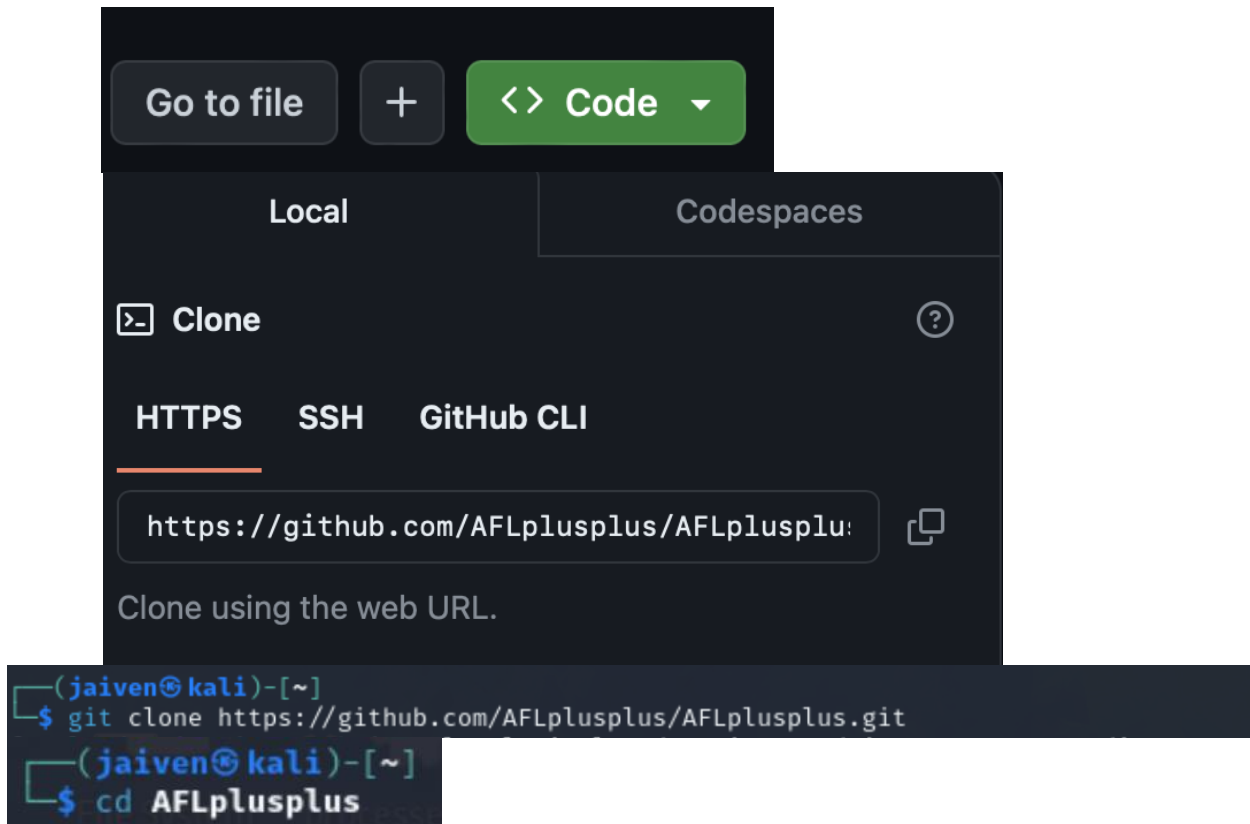
STEP 3: CLONE AFL++

After we installed AFL++ we would now need to clone it, and clone means that we are making a copy of the tool and there are a few cloning tools such as Disk Cloning, Virtual Machine Cloning and Git Repository Cloning. For the cloning tool that we are using is a git clone, and its purpose is to clone all the repositories that is hosted on a remote server that

can include projects, codes and more. We can also change the coding, create new branches etc.

STEPS:

1. Go to the website <https://github.com/AFLplusplus/AFLplusplus?tab=readme-overview> and click on the green button name code and copy the link.
2. Go to the terminal in kali linux
3. Type the command git clone and paste the link and press enter.
4. Then type "cd AFLplusplus" this would make the directory of "AFLPLUSPLUS"



STEP 4: MAKE BINARY INSTRUMENTATIONS.

After we have cloned AFL++ we now can make the binary instrumentations of it, we can install tools for AFL++ and now be able to use these fuzzing tools in the directory of the terminal. To make the binary instrumentations we would need to type the first command that

will install these fuzzing tools and the second command that will create the distribution package of AFL++ and this will take some time to create it.

```
(jaiven@kali)-[~/AFLplusplus]
$ sudo apt-get install -y build-essential python3-dev automake cmake git flex bison libglib2.0-dev libpixmap-1-dev python3-setuptools cargo libgtk-3-dev
[sudo] password for jaiven:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
build-essential is already the newest version (12.10).
python3-dev is already the newest version (3.11.4-5+b1).
automake is already the newest version (1:1.16.5-1.3).
cmake is already the newest version (3.28.3-1).
git is already the newest version (1:2.43.0-1).
flex is already the newest version (2.6.4-8.2+b2).
bison is already the newest version (2:3.8.2+dfsg-1+b1).
libglib2.0-dev is already the newest version (2.78.3-2).
libpixmap-1-dev is already the newest version (0.42.2-1).
python3-setuptools is already the newest version (68.1.2-2).
cargo is already the newest version (0.70.1-ds1-2).
libgtk-3-dev is already the newest version (3.24.41-1).
0 upgraded, 0 newly installed, 0 to remove and 1278 not upgraded.

(jaiven@kali)-[~/AFLplusplus]
$ sudo make distrib
```

STEPS:

1. Type the first command that is shown above.
2. Enter your password and let it run
3. This command will build the fuzzing tools of AFL++
4. After that's done then type the second command
5. This command will make the distribution of AFL++ and will take a while to be installed

COMMAND MEANING:

1. **sudo:** it stands for "superuser do" and is used to run commands with elevated privileges. It often requires the user to enter their password.
2. **apt-get** is a package management command-line tool for debian-based systems. It is used to handle packages, which are software applications and libraries.
3. **install:** this is the action to be performed by apt-get. it means you want to install the specified packages.
4. **-y:** this option is used to automatically answer "yes" to prompts that may come up during the installation, without requiring manual confirmation.
5. **make:** This is a build automation tool that is used to compile and build projects. It reads a file called a Makefile to determine how to build the software.
6. **distrib:** This is likely a target specified in the AFL++ Makefile that starts the distribution packaging process. It may involve compiling the code, copying necessary files, and creating a distributable package.

PACKAGES:

1. **build-essential:** A package that includes essential tools for building software on a Linux system, such as compilers and make.
2. **python3-dev:** Development files for the Python programming language. These are needed when building Python extensions or modules.

3. **automake**: A tool for automatically generating Makefile.in files from files called Makefile.am.
4. **cmake**: Cross-platform open-source make system. It is used for managing the build process of software using a compiler-independent method.
5. **git**: A distributed version control system used for tracking changes in source code during software development.
6. **flex**: A tool for generating scanners (programs that recognize lexical patterns in text).
7. **bison**: A general-purpose parser generator that converts a grammar description into a C program.
8. **libglib2.0-dev**: Development files for the GLib library, a low-level core library for the GNOME project.
9. **libpixmap-1-dev**: Development files for the Pixmap library, a low-level software library for pixel manipulation.
10. **python3-setuptools**: A package for Python that provides a higher-level interface to the low-level tools provided by distutils.]
11. **cargo**: The package manager for Rust, a programming language.
12. **libgtk-3-dev**: Development files for the GTK+ toolkit, a multi-platform toolkit for creating graphical user interfaces.

STEP 5: MAKING THE C PROGRAM.

We can now create the C program in the terminal. First to create the C program we need to make an empty file and we can do this with the touch command and name the file whatever we want with the .C in the end. We now have an empty file and now ready to create the C

to such as fuzz testing and debugging and this will also find vulnerabilities with the provided inputs in the C program. After we type the first command, we can type the second command which will fuzz the C program. After fuzzing the program, we can see many things that are happening such as the process timing, stage processing and cycle process etc. We can see the name of our file and the fuzzing tool we are using; we also see how many crashes there are and how many have been saved.

STEPS:

1. Type the first command afl-cc along with the file name that we made
2. This will compile the file and will find the vulnerabilities in the file
3. Then type the second command afl-fuzz along with the file named
4. This will fuzz the program and can show all the things that is going on with the program.
5. To exit the fuzzing tool we can simply hold control and type C.

```
(jaiven@kali)-[~/AFLplusplus]
$ afl-cc -fsanitize=address,undefined -ggdb -O0 imgRead.c -o imgRead_aflplus
afl-cc++4.10c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: LLVM-PCGUARD
SanitizerCoveragePCGUARD++4.10c
[+] Instrumented 102 locations with no collisions (non-hardened mode) of which are 6 handled and 0 unhandled selects.

(jaiven@kali)-[~/AFLplusplus]
$ afl-fuzz -i dir -o dir -m none -- ./imgRead_aflplus @@

american fuzzy lop ++4.10c {default} (./imgRead_aflplus) [explore]
┌──────────┴──────────┐
┌──────────┐          ┌──────────┐
│ process timing │      │ overall results │
├──────────┴──────────┤
│ run time : 0 days, 0 hrs, 0 min, 3 sec │ │ cycles done : 1 │
│ last new find : none yet (odd, check syntax) │ │ corpus count : 3 │
│ last saved crash : 0 days, 0 hrs, 0 min, 3 sec │ │ saved crashes : 3 │
│ last saved hang : none seen yet │ │ saved hangs : 0 │
├──────────┴──────────┤
│ cycle progress │      │ map coverage │
├──────────┴──────────┤
│ now processing : 0.5 (0.0%) │ │ map density : 5.50% / 5.50% │
│ runs timed out : 0 (0.00%) │ │ count coverage : 22.33 bits/tuple │
├──────────┴──────────┤
│ stage progress │      │ findings in depth │
├──────────┴──────────┤
│ now trying : havoc │ │ favored items : 1 (33.33%) │
│ stage execs : 42/100 (42.00%) │ │ new edges on : 1 (33.33%) │
│ total execs : 1371 │ │ total crashes : 980 (3 saved) │
│ exec speed : 390.0/sec │ │ total tmouts : 5 (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 : 1/412, 2/900 │ │ stability : 100.00% │
│ py/custom/rq : unused, unused, unused, unused │ │ │
│ trim/eff : 80.95%/5, disabled │ │ │
├──────────┴──────────┤
│ strategy: explore │ state: started :-) │
├──────────┴──────────┤
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

1st COMMAND MEANING:

1. afl-cc: This is likely a wrapper script or compiler command provided by AFL (American Fuzzy Lop), which is a fuzzer used for software testing.

2. **-fsanitize=address,undefined:** This option enables AddressSanitizer and UndefinedBehaviorSanitizer. AddressSanitizer helps find memory-related errors (e.g., out-of-bounds accesses, use-after-free), and UndefinedBehaviorSanitizer helps find undefined behavior in the C code.
3. **-ggdb:** This includes debugging information in the compiled binary, making it easier to debug using GDB (GNU Debugger).
4. **-O0:** This sets the optimization level to zero, meaning no optimization is performed. This is useful during development and debugging to make it easier to understand the generated code.
5. **imgRead.c:** This is the source code file that is being compiled. It's assumed that there is a C source code file named imgRead.c in the current directory.
6. **-o imgRead_aflplus:** This specifies the output file name for the compiled binary. In this case, the compiled program will be named imgRead_aflplus.

2ND COMMAND MEANING:

1. **afl-fuzz:** This is the AFL fuzzer command.
2. **-i dir:** Specifies the input directory (dir) from which the fuzzer will take initial test cases or seed files.
3. **-o dir:** Specifies the output directory (dir) where AFL will store its findings, including crashes and interesting test cases.
4. **-m none:** Specifies the memory limit. In this case, it's set to "none," meaning there is no specific memory limit.
5. **--:** Separates AFL-specific options from the actual command to be fuzzed.
6. **./imgRead_aflplus:** This is the path to the executable binary that you want to fuzz. In this case, it's the imgRead_aflplus program that was likely compiled with AFL-specific options.
7. **@@:** This is an AFL placeholder indicating where the fuzzer should insert the test case data. The double at symbol (@@) is often used in AFL to represent the location where the fuzzer should place its mutated or generated input s.

2. How to Fuzz Arm and MIPS Binaries with Qemu and AFL++

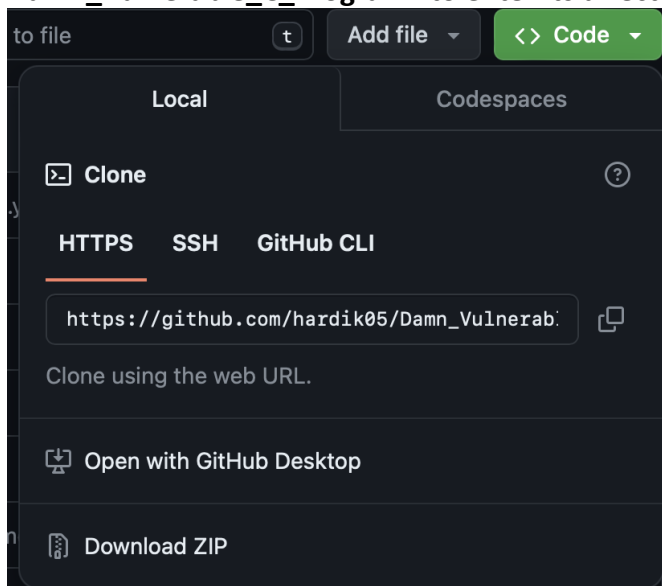
STEP 1: DOWNLOADING DAMN_VULNERABLE_C_PROGRAM

Our 2nd objective of using AFL++ is knowing how to fuzz ARM and MIPS Binaries using qemu mode in AFL++. We should already install AFL++ in kali Linux so we can continue

with the objective. But before we can do the 2nd objective, we need to clone a directory so we can be able to access the files we need to fuzz the program.

STEPS:

1. Go to the website https://github.com/hardik05/Damn_Vulnerable_C_Program and click on the green button called code.
2. Then click on the copy and paste button and go to kali Linux
3. open the terminal in kali Linux and type the command "git clone" along with the link we copy earlier.
4. After that is finished cloning we can go ahead and type the command "Damn_Vulnerable_C_Program" to enter its directory



```
(jaiven@kali)-[~]
$ git clone https://github.com/hardik05/Damn_Vulnerable_C_Program.git
Cloning into 'Damn_Vulnerable_C_Program' ...
remote: Enumerating objects: 1511, done.
remote: Counting objects: 100% (113/113), done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 1511 (delta 103), reused 73 (delta 72), pack-reused 1398
Receiving objects: 100% (1511/1511), 67.09 MiB | 24.60 MiB/s, done.
Resolving deltas: 100% (683/683), done.
Updating files: 100% (1914/1914), done.

(jaiven@kali)-[~]
$ cd Damn_Vulnerable_C_Program
```

1st COMMAND MEANING:

1.git clone: This is the Git command used to create a copy of a repository.

https://github.com/hardik05/Damn_Vulnerable_C_Program.git: This is the URL of the repository you want to clone. In this case, it's the 2."Damn_Vulnerable_C_Program" repository hosted on GitHub under the username "hardik05".

2nd COMMAND MEANING:

1. cd: This is a command used to change the current directory.
2. Damn_Vulnerable_C_Program: This is the name of the directory you want to navigate into.

STEP 2: SEEING THE C PROGRAM FOR FUZZING MIPS BINARIES

After we went ahead and enter the directory, we now have the C program that we are going to be using for the ARM and MIPS. We will then compile our C program in step 3.

STEPS:

1. Type the command “ls” in the directory
2. The C program that we are going to be using is called “imgRead.c”
3. we can see the program by typing “vi imgRead.c” and to go back to the terminal we can type “: wq”



```
(jaiven@kali)~[~/Damn_Vulnerable_C_Program]
$ ls
AFLplusplus  ReadMe.md  dir  imgRead.c  imgRead_arm  imgRead_arm64  imgRead_arm_static  imgRead_gcc_noasan  imgRead_patched.c  libAFL  linux  radamsa  windows
(jaiven@kali)~[~/Damn_Vulnerable_C_Program]
$ vi imgRead.c
```

1st COMMAND MEANING:

1. ls command is used in Unix-like operating systems (such as Linux and macOS) to list the files and directories in the current directory. It provides a simple way to view the contents of a directory.

2nd COMMAND MEANING

1. vi: This is a command-line text editor available on Unix-like operating systems, such as Linux and macOS. It is a powerful and widely used editor known for its extensive capabilities.
2. imgRead.c: This is the name of the file you want to open in the vi editor. In this case, it is assumed to be a C source code file with the extension ".c", but it could be any type of text file.

STEP 3: INSTALLING ARM AND MIPS TOOLCHAINS AND COMPILING PROGRAM

We can now compile the C program in the directory with different output names. We will also be using the packages called ARM and ARM64. These will be compiled with the C program for arm and aarch64 to check what bits they are.

STEPS:

1. Type the command “gcc” along with the C program we are going to be using and type “-o” and type “imgRead_gcc_noasan”
2. We now can file the output to see if its compile by typing “ file” then the output file
3. Type the command “arm-linux-gnueabi-gcc” with the C program and the output being “imgRead_arm”
4. We can do the same with aarch64 by typing “aarch64-linux-gnu-gcc” with the C Program and the output being “imgRead_arm64.”

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ gcc imgRead.c -o imgRead_gcc_noasan
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ file imgRead_gcc_noasan
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ arm-linux-gnueabi-gcc imgRead.c -o imgRead_arm

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ file imgRead_arm
imgRead_arm: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
ppcd
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ aarch64-linux-gnu-gcc imgRead.c -o imgRead_arm64

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ file imgRead_arm64
imgRead_arm64: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV), dynamically linked,
.0, not stripped
```

1st COMMAND MEANING:

1. gcc: This is the command for the GNU Compiler Collection, which is a compiler system for programming languages like C, C++, and others.
- imgRead.c: This is the name of the source code file you want to compile. In this case, it is assumed to be a C source code file.
- o imgRead_gcc_noasan: This option specifies the name of the output executable. In this case, it's set to "imgRead_gcc_noasan". The -o flag is followed by the desired name of the output file.

2nd COMMAND MEANING:

1. file: This is the command used to determine the file type.
2. imgRead_gcc_noasan: This is the name of the file for which you want to determine the type.

3rd COMMAND MEANING:

1.arm-linux-gnueabi-gcc: This is the cross-compiler for the ARM architecture. It allows you to compile code on a different architecture (typically x86 or x86_64) targeting ARM-based systems.

2.imgRead.c: This is the name of the source code file you want to compile. In this case, it's assumed to be a C source code file.

3.-o imgRead_arm_: This option specifies the name of the output executable. In this case, it's set to "imgRead_arm_". The -o flag is followed by the desired name of the output file.

4th COMMAND MEANING:

1.file: This is the command used to determine the file type.

2. imgRead_arm: This is the name of the file for which you want to determine the type.

5th COMMAND MEANING:

1.arm-linux-gnueabi-gcc: This is the cross-compiler used for compiling code intended to run on ARM-based systems. It generates code for the ARM architecture.

2. imgRead.c: This is the name of the source code file you want to compile. It is assumed to be a C source code file.

3. -o imgRead_arm64: This option specifies the name of the output executable. In this case, it's set to "imgRead_arm64". The -o flag is followed by the desired name of the output file.

6th COMMAND MEANING:

1.file: This is the command used to determine the file type.

2.imgRead_arm64: This is the name of the file for which you want to determine the type.

STEP 4: INSTALLING 3 DIFFERENT PACKAGES

Now that we compile our C program with different output names, we now are going to install 3 different packages called ARM, MIPS and AARCH64. Installing these three packages will take while to compile will ask if you want to compile them with a yes or no question.

STEPS:

1. In the directory we can type the commands “`sudo apt install gcc make gcc-arm-linux-gnueabi binutils-arm-linux-gnueabi`” and press enter. This will be compiling our Arm.
2. Enter your login password and let it compile, this may take a while to compile everything that needs to compile
3. Type the “`sudo apt-get install -y gcc-mips-linux-gnu`” and do the same like step 1. This will be compiling our MIPS.
4. Now we will compile the last packages by typing the command “`sudo apt install gcc make gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu`” and do the same as the first two steps. This will be compiling our AARCH64

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ sudo apt install gcc make gcc-arm-linux-gnueabi binutils-arm-linux-gnueabi
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc is already the newest version (4:13.2.0-7).
gcc set to manually installed.
make is already the newest version (4.3-4.1).
make set to manually installed.
gcc-arm-linux-gnueabi is already the newest version (4:13.2.1-6).
binutils-arm-linux-gnueabi is already the newest version (2.42-4).
binutils-arm-linux-gnueabi set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 491 not upgraded.

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ sudo apt-get install -y gcc-mips-linux-gnu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc-mips-linux-gnu is already the newest version (4:12.2.0-4).
0 upgraded, 0 newly installed, 0 to remove and 491 not upgraded.

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ sudo apt install gcc make gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc is already the newest version (4:13.2.0-7).
make is already the newest version (4.3-4.1).
gcc-aarch64-linux-gnu is already the newest version (4:13.2.0-7).
gcc-aarch64-linux-gnu set to manually installed.
binutils-aarch64-linux-gnu is already the newest version (2.42-4).
binutils-aarch64-linux-gnu set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 491 not upgraded.
```

1st COMMAND MEANING:

- 1.sudo: This command is used to execute the subsequent command with administrative privileges.
- 2.apt: This is the package management tool used in Debian-based Linux distributions, including Ubuntu.
- 3.install: This is the argument passed to apt to indicate that you want to install packages.

4.gcc: This is the GNU Compiler Collection, which is used to compile C, C++, and other programming languages.

5.make: This is a build automation tool that is used to build projects based on instructions in a Makefile.

6.gcc-arm-linux-gnu: This package provides the ARM cross-compiler, which allows you to compile code for ARM architecture on an x86 or x86_64 system.

7.binutils-arm-linux-gnu: This package provides the ARM version of the GNU Binutils, which are a collection of binary tools used for tasks such as assembling, linking, and analyzing binary files.

2nd COMMAND MEANING:

1.sudo: This command is used to execute the subsequent command with administrative privileges.

2.apt-get: This is another package management tool used in Debian-based Linux distributions, including Ubuntu. It is an alternative to apt.

3.install: This is the argument passed to apt-get to indicate that you want to install packages.

4.-y: This is an option that tells apt-get to assume "yes" as the answer to all prompts and automatically install the packages without user intervention. It's useful for unattended installations.

5.gcc-mips-linux-gnu: This package provides the GCC compiler for the MIPS architecture. It allows you to compile code for MIPS architecture on an x86 or x86_64 system.

3rd COMMAND MEANING:

1.sudo: This command is used to execute the subsequent command with administrative privileges.

2.apt: This is the package management tool used in Debian-based Linux distributions, including Ubuntu.

3.install: This is the argument passed to apt to indicate that you want to install packages.

4.gcc: This is the GNU Compiler Collection, which is used to compile C, C++, and other programming languages.

make: This is a build automation tool that is used to build projects based on instructions in a Makefile.

5.gcc-aarch64-linux-gnu: This package provides the AArch64 (ARM 64-bit) cross-compiler, which allows you to compile code for ARM 64-bit architecture on an x86 or x86_64 system.

6.binutils-aarch64-linux-gnu: This package provides the AArch64 version of the GNU Binutils, which are a collection of binary tools used for tasks such as assembling, linking, and analyzing binary files.

STEP 5: COMPILING AND INSTALLING QEMU SUPPORT FOR AFLPLUSPLUS

Now we are finally going to be using AFL++ modes called qemu mode (Quick Emulator) and it is a generic and open-source emulator that can emulate various CPUs and hardware platforms. When AFL++ operates in QEMU mode, it uses QEMU to dynamically translate

instructions from the target binary's architecture into instructions that can be executed on the host system's architecture. This allows AFL++ to fuzz binaries without requiring access to their source code or without having to recompile them for the host system's architecture.

STEPS:

1. Type the command “cd qemu mode” this will put us in the directory of AFL++ qemu mode
2. Now we will need to install the ninja Build before we can install qemu and we can do that by typing the command “sudo apt-get install ninja-build”.
3. Then we can type the command “s” and this will be set for MIPS
4. Now we can type “sudo cp ../afl-qemu-trace /usr/local/bin/afl-qemu-trace-mips” which will set a copy of the binary we have made.
5. We can repeat the same steps for Arm.

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus]
$ cd qemu_mode
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus/qemu_mode]
$ sudo apt-get install ninja-build

Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  ninja-build
0 upgraded, 1 newly installed, 0 to remove and 491 not upgraded.
Need to get 128 kB of archives.
After this operation, 446 kB of additional disk space will be used.
Get:1 http://mirrors.jevincanders.net/kali kali-rolling/main arm64 ninja-build arm64 1.11.1-2 [128 kB]
Fetched 128 kB in 1s (103 kB/s)
Selecting previously unselected package ninja-build.
(Reading database ... 412714 files and directories currently installed.)
Preparing to unpack .../ninja-build_1.11.1-2_arm64.deb ...
Unpacking ninja-build (1.11.1-2) ...
Setting up ninja-build (1.11.1-2) ...
Processing triggers for doc-base (0.11.2) ...
Processing 1 added doc-base file ...
Processing triggers for man-db (2.12.0-3) ...
Processing triggers for kali-menu (2023.4.7) ...

(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus/qemu_mode]
$ sudo CPU_TARGET=mips ./build_qemu_support.sh
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus/qemu_mode]
$ sudo cp ../afl-qemu-trace /usr/local/bin/afl-qemu-trace-mips
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus/qemu_mode]
$ sudo CPU_TARGET=arm ./build_qemu_support.sh
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program/AFLplusplus/qemu_mode]
$ sudo cp ../afl-qemu-trace /usr/local/bin/afl-qemu-trace-arm
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ file imgRead_arm
imgRead_arm: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
ipped
```

1st COMMAND MEANING:

cd: This is the command used to change directories.

qemu_mode: This is the name of the directory you want to navigate into.

2nd COMMAND MEANING:

1. **sudo:** This command is used to execute the subsequent command with administrative privileges.
2. **apt-get:** This is another package management tool used in Debian-based Linux distributions, including Ubuntu. It is an alternative to apt.
3. **install:** This is the argument passed to apt-get to indicate that you want to install packages.
4. **ninja-build:** This is the package name for the Ninja build system. Ninja is a build system similar to Make, but it is designed to be faster and more efficient.

3rd COMMAND MEANING:

1. **sudo:** This command is used to execute the subsequent command with administrative privileges.
2. **CPU_TARGET=mips:** This is an environment variable that specifies the target CPU architecture for which the QEMU support will be built. In this case, it's set to MIPS architecture.
3. **./build_qemu_support.sh:** This is the command to run the script named build_qemu_support.sh, which is presumably responsible for configuring and building QEMU support for the MIPS architecture.

4th COMMAND MEANING:

1. **sudo:** This command is used to execute the subsequent command with administrative privileges.
2. **cp:** This is the command to copy files and directories.
3. **../afl-qemu-trace:** This is the path to the afl-qemu-trace binary, which is being copied. ../ indicates that the file is located one directory level up from the current directory.

4. `/usr/local/bin/afl-qemu-trace-mips`: This is the destination path to which the `afl-qemu-trace` binary is being copied. It is being renamed to `afl-qemu-trace-mips`.

5th COMMAND MEANING:

1. `sudo`: This command is used to execute the subsequent command with administrative privileges.

2. `CPU_TARGET=arm`: This is an environment variable that specifies the target CPU architecture for which the QEMU support will be built. In this case, it's set to ARM architecture.

3. `./build_qemu_support.sh`: This is the command to run the script named `build_qemu_support.sh`, which is presumably responsible for configuring and building QEMU support for the ARM architecture.

6th COMMAND MEANING:

1. `sudo`: This command is used in Unix-like operating systems to run programs with the security privileges of another user, typically the superuser (root). It is often required for tasks that require elevated permissions.

2. `cp`: This is the command to copy files and directories.

3. `../afl-qemu-trace`: This is the path to the `afl-qemu-trace` binary, which is being copied. `../` means it's one level up from the current directory.

4. `/usr/local/bin/afl-qemu-trace-arm`: This is the destination path to which the `afl-qemu-trace` binary is being copied. It is being renamed to `afl-qemu-trace-arm`.

7th COMMAND MEANING:

`file`: This is the command used to determine the file type.

`imgRead_arm`: This is the name of the file for which you want to determine the type.

STEP 6: COMPILING IN STATIC MODE TO AVOID DEPENDENCIES ISSUE

After we compile and install `qemu` mode in `AFL++` we should be able to fuzz the programs with `qemu` mode and see how many crashes we are able to achieve. We would also need to compile the `imgRead.c` with the output being named `imgRead_arm_static` in static mode. Static mode analyzes the binary's code statically, meaning it examines the code structure, function calls, and other characteristics without running the program.

STEPS:

1. Going back to our Damn_Vulnerable_C_Program we will be compiling the imgRead.c in static mode by typing “arm-linux-gnueabi-gcc imgRead.c -o imgRead_arm_static -static” to compile
2. Now we will type the command “. /imgRead_mips” and after pressing enter we can see that there is no file or directory available.
3. We can find the file by typing the command “locate ld.so.1” which the command can find files by name, and it relies on a prebuilt index of file names to quickly locate files.
4. Now we will be able to fuzz the file by typing the command “afl-fuzz -U -i dir -o dir -- afl-qemu-trace-arm -L /usr/arm-linux-gnueabi ./imgRead_arm” and press enter.
5. We see that we are fuzzing the C Program with ARM showing how many crashes there are, how many cycles have been through and more.

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ arm-linux-gnueabi-gcc imgRead.c -o imgRead_arm_static -static

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ ./imgRead_mips
zsh: no such file or directory: ./imgRead_mips

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ locate ld.so.1
/usr/mips-linux-gnu/lib/ld.so.1
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ afl-fuzz -U -i dir -o dir -- afl-qemu-trace-arm -L /usr/arm-linux-gnueabi ./imgRead_arm

american fuzzy lop ++4.08c {default} (afl-qemu-trace-arm) [fast]
┌──────────┴──────────┐
┌──────────┐┌──────────┐┌──────────┐
│ process timing │ │ overall results │ │
│   run time   : 0 days, 0 hrs, 0 min, 4 sec │ │ cycles done  : 2 │ │
│ last new find : none yet (odd, check syntax!) │ │ corpus count : 3 │ │
│ last saved crash : none seen yet │ │ saved crashes : 0 │ │
│ last saved hang : none seen yet │ │ saved hangs  : 0 │ │
└──────────┘└──────────┘└──────────┘
┌──────────┐┌──────────┐┌──────────┐
│ cycle progress │ │ map coverage │ │
│ now processing : 0.13 (0.0%) │ │ map density  : 0.03% / 0.03% │ │
│ runs timed out : 0 (0.00%) │ │ count coverage : 1.00 bits/tuple │ │
└──────────┘└──────────┘└──────────┘
┌──────────┐┌──────────┐┌──────────┐
│ stage progress │ │ findings in depth │ │
│ now trying : havoc │ │ favored items  : 1 (33.33%) │ │
│ stage execs : 76/345 (22.03%) │ │ new edges on  : 1 (33.33%) │ │
│ total execs : 12.2k │ │ total crashes : 0 (0 saved) │ │
│ exec speed  : 3763/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 : 0/4215, 0/7920 │ │ stability   : 100.00% │ │
│ py/custom/rq : unused, unused, unused, unused │ │ │ │
│ trim/eff     : 93.22%/14, disabled │ │ │ │
└──────────┘└──────────┘└──────────┘
strategy: exploit state: started :-)
```

1st COMMAND MEANING:

- 1.arm-linux-gnueabi-gcc: This is the cross-compiler used for compiling code intended to run on ARM-based systems. It generates code for the ARM architecture.
- 2.imgRead.c: This is the name of the source code file you want to compile. It is assumed to be a C source code file.
- 3.-o imgRead_arm_static: This option specifies the name of the output executable. In this case, it's set to "imgRead_arm_static". The -o flag is followed by the desired name of the output file.
- 4.-static: This option tells the compiler to link the resulting executable statically. Statically linked executables include all necessary libraries and dependencies within the executable itself, rather than relying on shared libraries present on the system. This can result in larger executable files but ensures that the program can run on systems without the required shared libraries installed.

2nd COMMAND MEANING:

1. ./: This is a shorthand notation that refers to the current directory.
2. imgRead_mips: This is the name of the executable binary file you want to execute. It is assumed to be located in the current directory.

3rd COMMAND MEANING:

- 1.locate: This is a command-line utility used to find files by name. It searches through a pre-built database of filenames and their locations on the system, which allows it to quickly locate files without traversing the entire filesystem.
- 2.ld.so.1: This is the name of the file you want to locate. "ld.so.1" is the dynamic linker/loader file used by Unix-like operating systems during runtime to load shared libraries.

4th COMMAND MEANING:

- 1.afl-fuzz: This is the AFL command used for fuzz testing. AFL is a popular fuzzing tool for finding security vulnerabilities in software by providing mutated inputs and monitoring program behavior.
- 2.-U: This option tells AFL to run the target program under QEMU, an emulator used for system-level virtualization, to enable testing on different architectures.
- 5.-i dir: This option specifies the input directory containing initial test cases (input files).
- 6.-o dir: This option specifies the output directory where AFL will store its findings, such as crash files and other diagnostic information.
- 7.--: This is a separator to distinguish AFL's options from the command to run the target program.
- 8.afl-qemu-trace-arm: This is AFL's wrapper for QEMU when fuzzing programs for the ARM architecture.

9.-L /usr/arm-linux-gnueabi: This option specifies the directory containing the ARM cross-compiler's libraries and tools. It's used by QEMU to locate necessary libraries for running the program.

10. ./imgRead_arm: This is the target program that AFL will fuzz test. It's assumed to be compiled for the ARM architecture, and AFL will generate mutated inputs to test its behavior for potential vulnerabilities.

3. How to use CmpLog feature to fuzz a binary

STEP 1: MODIFYING DAMN VULNERABLE C PROGRAM TO VERIFY HEADER "IMG" IN THE INPUT FILES

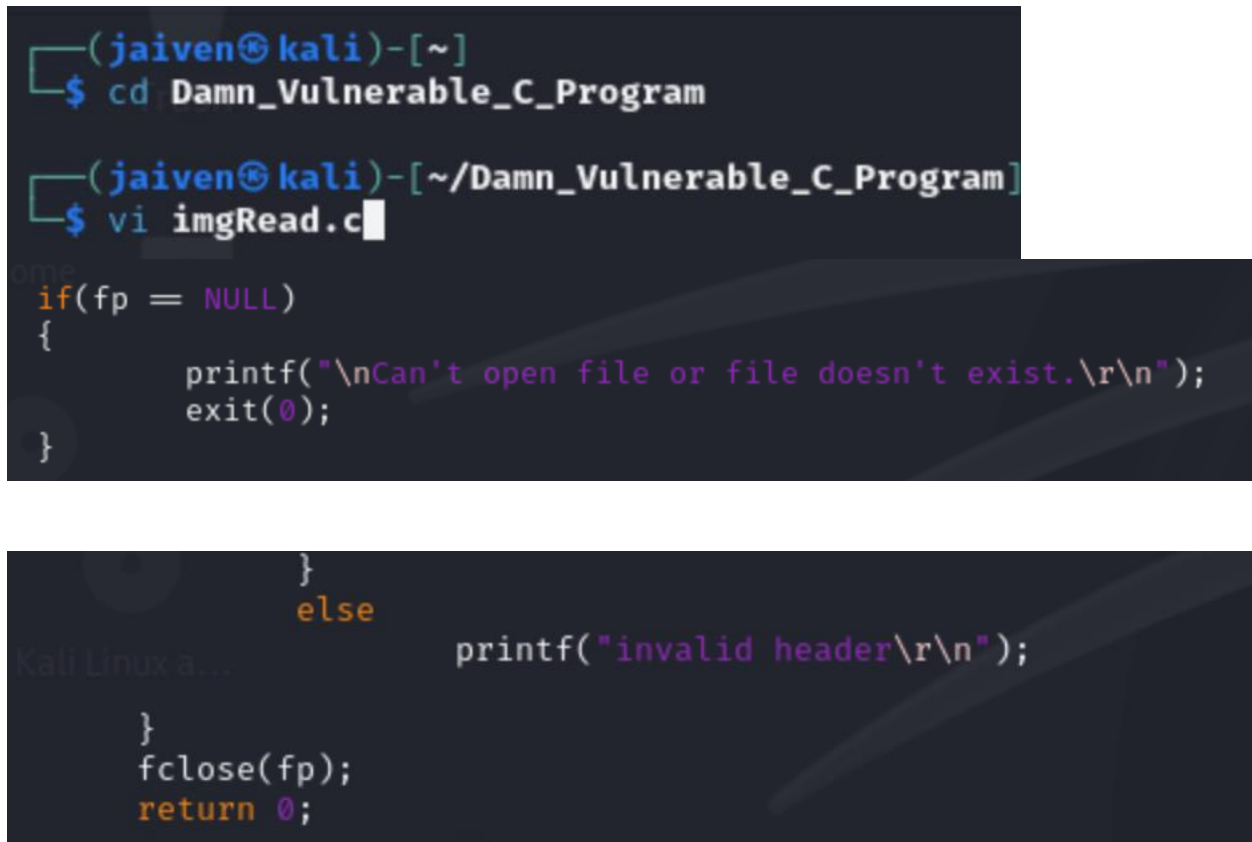
Now we are the last objective which is using Cmplog feature to fuzz a binary using AFL++.

CmpLog is a feature in AFL++ (American Fuzzy Lop Plus Plus) that enhances the fuzzing process by logging the outcomes of comparisons encountered during execution. This helps AFL++ to better understand program behavior, explore new paths more effectively, and uncover bugs or vulnerabilities that might otherwise remain hidden. But before we use Cmplog we need to modify the Damn_Vulnerable_C_Program to read the "IMG" for the Cmplog.

STEPS:

1. go to the terminal and go to the directory Damn_Vulnerable_C_Program and type the command "vi imgRead.c"

2. Delete comments of the if and the end of the bracket
3. Then scroll down a bit and delete the comments of the else and along the inside of the bracket



The first screenshot shows a terminal window with the prompt `(jaiven@kali)-[~]`. The user enters `$ cd Damn_Vulnerable_C_Program`, and the prompt changes to `(jaiven@kali)-[~/Damn_Vulnerable_C_Program]`. Then, the user enters `$ vi imgRead.c`, opening the file in the Vi editor.

The second screenshot shows the contents of `imgRead.c` in the Vi editor. The code is as follows:

```

    }
    else
        printf("invalid header\r\n");
    }
    fclose(fp);
    return 0;

```

1st COMMAND MEANING:

1. `cd`: This stands for "change directory". It's used to navigate between directories or folders within a file system.
2. `Damn_Vulnerable_C_Program`: This is the argument passed to the `cd` command. It represents the name of the directory or folder you want to change into.

2nd COMMAND MEANING:

1. `vi`: This is the command to invoke the Vi text editor. Vi is a widely used text editor on Unix-based systems, known for its powerful features and efficiency. It's a modal editor, meaning it has different modes for inserting text, editing text, and navigating within a file.
2. `imgRead.c`: This is the argument passed to the `vi` command. It represents the name of the file you want to open or create in the Vi text editor. In this case, "imgRead.c" is a C source code file, so you're instructing Vi to open this file for viewing and editing.

STEP 2: COMPILING "IMGREAD.C" PROGRAM WITH "AFL-CLANG-FAST"

After we have modified the `imgRead.c` program we need to go ahead and compile it with `afl-clang`. When we are using `afl-clang` it adds AFL-specific instrumentation to the resulting binary, which is essential for AFL's fuzzing engine to function properly. This instrumentation collects information about the program's execution and feeds it back to AFL, helping it to intelligently mutate inputs and guide the fuzzing process towards paths that haven't been explored yet.

STEPS:

1. Type the command "`afl-clang`" along with the program we are compiling and finished typing "`-fsanitize=address,undefined -o imgRead_aflplusplus`" to finish compiling the C program.

```
(jaiven@kali) - [~/Damn_Vulnerable_C_Program]
$ afl-clang imgRead.c -fsanitize=address,undefined -o imgRead_aflplusplus
afl-cc++4.08c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: LLVM-PCGUARD
SanitizerCoveragePCGUARD++4.08c
[+] Instrumented 57 locations with no collisions (non-hardened mode) of which are 2 handled and 0 unhandled selects.
```

1st COMMAND MEANING:

1. `afl-clang`: This is a wrapper script provided by AFL++ that acts as a drop-in replacement for the standard clang compiler. It is used to compile C/C++ programs with AFL instrumentation to enable fuzzing.
2. `imgRead.c`: This is the name of the C source code file you want to compile. In this case, it's "`imgRead.c`".
3. `-fsanitize=address,undefined`: These are compiler flags that enable address sanitizer (ASan) and undefined behavior sanitizer (UBSan). Address sanitizer helps detect memory errors (such as use-after-free, buffer overflow, etc.), while undefined behavior sanitizer helps detect undefined behavior in C/C++ code. These sanitizers are invaluable for finding bugs and security vulnerabilities in C/C++ programs.
4. `-o imgRead_aflplusplus`: This specifies the output file name after compilation. In this case, the compiled binary will be named "`imgRead_aflplusplus`".

STEP 3: CREATING SAMPLE INPUT FILE WITH "XXX" AS HEADER AND FUZZING THE PROGRAM.

Now that we have compile the C Program we need to go ahead and create a sample for cmplog and try to fuzz it.

STEPS:

1. Type the command "`mkdir incmplog`" this will make a new file for incmplog
2. Now type the command "`echo "xxx" > incmplog/1.img`" this will call out the "`xxx`" that we have made.
3. Next type "`hexyl incmplog/1.img`" hexyl will analyzing binary files in Kali Linux for debugging and malware analysis.

4. We can fuzz it now by typing “afl-fuzz -i incmplog -o dir -m none ./imgRead_aflplusplus @@”

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ mkdir incmplog

(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ echo "xxx" > incmplog/1.img
```

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ hexyl incmplog/1.img
```

00000000	78 78 78 0a		xxx_	
----------	-------------	--	------	--

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ afl-fuzz -i incmplog -o dir -m none ./imgRead_aflplusplus @@
```

```
american fuzzy lop ++4.08c (default) (./imgRead_aflplusplus) [fast]
process timing:
  run time : 0 days, 0 hrs, 0 min, 10 sec
  last new find : 0 days, 0 hrs, 0 min, 9 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress:
  now processing : 0.20 (0.0%)
  runs timed out : 0 (0.00%)
stage progress:
  now trying : havoc
  stage execs : 420/459 (91.50%)
  total execs : 24.5k
  exec speed : 2356/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 : 4/10.0k, 1/13.9k
  py/custom/rq : unused, unused, unused, unused
  trim/off : 8.09%/75, disabled
strategy: explore
state: started (-)
^C

overall results:
  cycles done : 3
  corpus count : 6
  saved crashes : 0
  saved hangs : 0
map coverage:
  map density : 7.58% / 10.61%
  count coverage : 70.14 bits/tuple
findings in depth:
  favored items : 2 (33.33%)
  new edges on : 2 (33.33%)
  total crashes : 0 (0 saved)
  total tmouts : 0 (0 saved)
item geometry:
  levels : 3
  pending : 2
  pend fav : 0
  own finds : 5
  imported : 0
  stability : 100.00%
[cpu001:100%]
```

1st COMMAND MEANING:

1. **mkdir:** This is the command used to create directories (folders) in a file system. It stands for "make directory".
2. **incmplog:** This is the argument passed to the mkdir command. It represents the name of the directory you want to create. In this case, the directory will be named "incmplog".

2nd COMMAND MEANING:

1. **echo "xxx":** This command simply prints the text "xxx" to the standard output.
2. **>:** This is a redirection operator in the shell. It's used to redirect the output of a command (in this case, the output of echo "xxx") to a file.
3. **incmplog/1.img:** This specifies the file path where the output of echo "xxx" will be redirected. It's creating a file named "1.img" inside the "incmplog" directory.

3rd COMMAND MEANING:

1. hexyl: This is the name of the program used to display hexadecimal representations of binary data.
2. incmplog/1.img: This is the path to the file whose hexadecimal representation you want to view. It specifies the file "1.img" located within the "incmplog" directory.

4th COMMAND MEANING:

1. afl-fuzz: This is the AFL++ command used to start the fuzzing process.
2. -i incmplog: This flag specifies the input directory containing the initial seed files for the fuzzer. AFL++ will use the files in this directory as initial inputs for the fuzzing process.
3. -o dir: This flag specifies the output directory where AFL++ will store its findings and other related data. In this case, it's named "dir".
4. -m none: This flag specifies the memory limit for the fuzzing process. Setting it to "none" means there's no specific memory limit imposed.
5. ./imgRead_aflplusplus: This is the path to the binary that AFL++ will fuzz. AFL++ will execute this binary with various mutated inputs to uncover potential bugs or vulnerabilities.
6. @@: This is a placeholder that tells AFL++ to replace it with the path to the input file being tested during each fuzzing iteration.

STEP 4: INTRODUCTION TO CMPLOG FEATURE IN AFLPLUSPLUS

Now we can use cmplog features for compiling AFL++, cmplog is CmpLog in AFL++ (American Fuzzy Lop Plus Plus) enhances the fuzzing process by logging the outcomes of comparisons encountered during the execution of the target program. When AFL++ fuzzes a program with CmpLog enabled, it instruments the binary to track the results of comparisons between variables or values.

STEPS:

1. Go to the website
<https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md> and copy the command "export AFL_LLVM_CMPLOG=1"
2. Go to the terminal and paste it, this command sets the environment variable AFL_LLVM_CMPLOG to 1, instructing AFL or AFL++

CmpLog instrumentation

The CmpLog instrumentation enables logging of comparison operands in a shared memory.

These values can be used by various mutators built on top of it. At the moment, we support the Redqueen mutator (input-2-state instructions only), for details see [the Redqueen paper](#).

Build

To use CmpLog, you have to build two versions of the instrumented target program:

- The first version is built using the regular AFL++ instrumentation.
- The second one, the CmpLog binary, is built with setting `AFL_LLVM_CMPLOG` during the compilation.

For example:

```
./configure --cc=/path/to/afl-clang-fast
make
cp ./program ./program.afl
make clean
export AFL_LLVM_CMPLOG=1
./configure --cc=/path/to/afl-clang-fast
make
cp ./program ./program.cmplog
unset AFL_LLVM_CMPLOG
```

export AFL_LLVM_CMPLOG=1

```
(jaiven@kali) - [~/Damn_Vulnerable_C_Program]
$ export AFL_LLVM_CMPLOG=1
```

1st COMMAND MEANING:

- 1. export:** This command is used to set environment variables in Unix-like operating systems. Environment variables are variables that define the environment in which programs run.
- 2. AFL_LLVM_CMPLOG:** This is the name of the environment variable being set. It's a specific variable recognized by AFL and AFL++ to enable LLVM's comparison logging feature.
- 3. 1:** This is the value assigned to the `AFL_LLVM_CMPLOG` variable. In this case, setting it to 1 indicates that the comparison logging feature should be enabled.

STEP 5: COMPILING "IMGREAD.C" WITH AFLPLUSPLUS CMPLOG FEATURE

Now we can compile the C program using afl++ with cmplogs features, this will compile the `imgRead.c` program with AFL++'s CmpLog instrumentation enabled, along with AddressSanitizer and UndefinedBehaviorSanitizer for improved error detection, and produces an output binary named `imgRead_cmplog`. This binary can then be used for fuzzing with AFL++.

STEPS:

1. Type the command `"AFL_LLVM_CMPLOG=1 afl-clang-fast imgRead.c -fsanitize=address,undefined -o imgRead_cmplog"`

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ AFL_LLVM_CMPLOG=1 afl-clang-fast imgRead.c -fsanitize=address,undefined -o imgRead_cmplog
afl-cc++4.08c by Michal Zalewski, Laszlo Szekeres, Marc Heuse - mode: LLVM-PCGUARD
Running cmplog-switches-pass by andreaforaldi@gmail.com
Hooking 1 switch instructions
Running split-switches-pass by laf.intel@gmail.com
SanitizerCoveragePCGUARD++4.08c
[+] Instrumented 82 locations with no collisions (non-hardened mode) of which are 2 handled and 0 unhandled selects.
Running cmplog-instructions-pass by andreaforaldi@gmail.com
Running cmplog-routines-pass by andreaforaldi@gmail.com
```

1st COMMAND MEANING:

1. **AFL_LLVM_CMPLOG=1:** This sets the environment variable **AFL_LLVM_CMPLOG** to 1, enabling LLVM's comparison logging feature for AFL.
2. **afl-clang-fast:** This is a version of the Clang compiler that is optimized for use with AFL. It compiles C/C++ code with AFL instrumentation, allowing AFL to perform feedback-driven fuzzing.
3. **imgRead.c:** This is the name of the C source code file you want to compile.
4. **-fsanitize=address,undefined:** These compiler flags enable address sanitizer (ASan) and undefined behavior sanitizer (UBSan), respectively. They help detect memory errors and undefined behavior in the compiled code, aiding in identifying potential bugs and vulnerabilities.
5. **-o imgRead_cmplog:** This specifies the output file name after compilation. In this case, the compiled binary will be named "imgRead_cmplog".

STEP 6: FUZZING "IMGREAD.C" PROGRAM WITH AFLPLUSPLUS IN CMPLOG MODE

Now we will fuzz the Program while using the CMPLOG mode in AFL++, and fuzzing this will show everything that the program is running. We can keep an eye on AFL++'s output directory for any crashes, hangs, or interesting paths discovered during fuzzing. You can also track the overall coverage and progress of AFL++ through its status output.

STEPS:

1. Type the command "afl-fuzz -i incmplog -o outcmplog -m none -c ./imgRead_cmplog -- ./imgRead_aflplusplus @@" and hit enter on your keyboard, this will fuzz the C Program.

```
(jaiven@kali)-[~/Damn_Vulnerable_C_Program]
$ afl-fuzz -i incmplog -o outcmplog -m none -c ./imgRead_cmplog -- ./imgRead_aflplusplus @@
```

```

american fuzzy lop ++4.08c {default} (./imgRead aflplus) [fast]
process timing
  run time      : 0 days, 0 hrs, 1 min, 26 sec
  last new find : none yet (add, check syntax!)
  last saved crash : 0 days, 0 hrs, 1 min, 18 sec
  last saved hang : none seen yet
cycle progress
  now processing : 0.160 (0.0%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 301/459 (65.58%)
  total execs : 72.9k
  exec speed : 732.2/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 : 10/72.6k, 0/0
  py/custom/rq : unused, unused, 0/1, 0/0
  trim/eff : n/a, disabled
  strategy: exploit
  state: started (-)
map coverage
  map density : 7.81% / 7.81%
  count coverage : 1.00 bits/tuple
findings in depth
  favored items : 1 (100.00%)
  new edges on : 1 (100.00%)
  total crashes : 13.7k (10 saved)
  total tmouts : 911 (0 saved)
item geometry
  levels : 1
  pending : 0
  pend fav : 0
  own finds : 0
  imported : 0
  stability : 100.00%
[cpu001:100%]
++ Testing aborted by user ++
[+] We're done here. Have a nice day!

```

1st COMMAND MEANING:

1. afl-fuzz: This is the AFL++ command for starting the fuzzing process.
2. -i incmplog: This flag specifies the input directory containing the initial seed files for the fuzzer. AFL++ will use the files in this directory as initial inputs for the fuzzing process.
3. -o outcmplog: This flag specifies the output directory where AFL++ will store its findings and other related data. In this case, it's named "outcmplog".
4. -m none: This flag specifies the memory limit for the fuzzing process. Setting it to "none" means there's no specific memory limit imposed.
5. -c ./imgRead_cmplog: This flag specifies the target binary for fuzzing. The 6. ./imgRead_cmplog is the compiled binary produced from the source code file "imgRead.c" with AFL LLVM mode enabled and comparison logging.
7. --: This double dash separates AFL++ options from the options intended for the target program.
8. ./imgRead aflplus @@: This is the command used to execute the target program under fuzzing. The @@ is a placeholder that AFL++ will replace with the path to the input file being tested during each fuzzing iteration.

STEP 7: REPLICATING CRASHES

Now that we have fuzz the C Program we can go ahead and replicate the crash that can provides important details about the crash, including its identifier, the signal that triggered it, and additional context such as the mutation strategy and repetition count. Such information is valuable for debugging and analyzing the root cause of the crash.

STEPS:

1. Type the command " cd outcmplog/default/crashes/" and the "ls" this will Navigate to the directory outcmplog/default/crashes/ and listing its contents would show the crash files generated by AFL++ during the fuzzing process
2. Now we can go back to the Damn_Vulnerable_C_Program and type "./imgRead aflplus outcmplog/default/crashes/id:000005,sig:06,src:000000,time:4079,execs:4696,op:havoc,rep:2 8" This command would execute imgRead aflplus with the input file that triggered the crash, allowing you to reproduce and debug the issue.

```

[jaiwan@kali] (~/.Damn_Vulnerable_C_Program)
$ cd outcmplog/default/crashes/
ls
README.txt
id:000000,sig:06,src:000000,time:118,execs:21,op:havoc,rep:3 id:000002,sig:06,src:000000,time:502,execs:680,op:havoc,rep:8 id:000005,sig:06,src:000000,time:4079,execs:4696,op:havoc,rep:26
id:000001,sig:06,src:000000,time:194,execs:159,op:havoc,rep:4 id:000003,sig:06,src:000000,time:608,execs:832,op:havoc,rep:7 id:000006,sig:06,src:000000,time:6039,execs:6701,op:havoc,rep:55
id:000004,sig:11,src:000000,time:1204,execs:1399,op:havoc,rep:3

```

```
[jaesend@kali:~/Dam_Vulnerable_C_Program]
$ ./ingRead_aflplus outcomp/default/crashes/id:000005,sig:06,src:000000,time:4079,execs:4696,op:havoc,rep:z8

Header width height data

+==+-----1960039422      2004318071      +==
ingRead_c7f4261 runtime error: signed integer overflow: -1960039422 * 2004318071 cannot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ingRead_c7f4261 in

=>DIB30E(BNDR) AddressSanitizer: heap-buffer-overflow on address 0xfffff78007b3 at pc 0xaaaaadbae008 bp 0xffffd2eb3a58 ff 0xffffd2eb3a58
WRITE of size 50 at 0xfffff78007b3 thread t=0
#0 0xaaaaadbae01c in __asan_memcpy (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_aflplus+0xa867c) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)
#1 0xaaaaadbae08c in ProcessImage (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_c7f4261:0x874) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)
#2 0xaaaaadbae0d8 in main /home/jaiven/Dam_Vulnerable_C_Program/ingRead_c7f4261:1312:
#3 0xffffda26dc6c (/lib/xarch64-linux-gnu/libc.so.6+0x26dc6c) (BuildId: 4d8fd32dca3057802b8662a79f3531cc4ee67254)
#4 0xffffdd26e994 in _libc_start_main (/lib/xarch64-linux-gnu/libc.so.6+0x26e994) (BuildId: 4d8fd32dca3057802b8662a79f3531cc4ee67254)
#5 0xaaaaad37bec in start (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_aflplus+0x27bec) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)

0xfffff78007b3 is located 0 bytes to the right of 1-byte region [0xfffff78007b0,0xfffff78007b1)
allocated by:
#0 0xaaaaadba12bc in __interceptor_malloc (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_aflplus+0xa12bc) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)
#1 0xaaaaadbae054 in ProcessImage (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_c7f4261:0x874) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)
#2 0xaaaaadbae0d8 in main /home/jaiven/Dam_Vulnerable_C_Program/ingRead_c7f4261:1312:
#3 0xffffda26dc6c (/lib/xarch64-linux-gnu/libc.so.6+0x26dc6c) (BuildId: 4d8fd32dca3057802b8662a79f3531cc4ee67254)
#4 0xffffdd26e994 in _libc_start_main (/lib/xarch64-linux-gnu/libc.so.6+0x26e994) (BuildId: 4d8fd32dca3057802b8662a79f3531cc4ee67254)
#5 0xaaaaad37bec in start (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_aflplus+0x27bec) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/jaiven/Dam_Vulnerable_C_Program/ingRead_aflplus+0xa867c) (BuildId: 42e9b520151cd2eb3aa5eff3be835d829892) in __asan_memcpy
Shadow bytes around the buggy address:
 0x200ff00ba0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00bb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00bc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00bd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00be0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
->0x200ff00bf0: fa fa fa fa fa fa [0] fa fa fa fa fa fa fa fa fa
 0x200ff00c00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00d00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00e00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff00f00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff01000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff01100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff01200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff01300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x200ff01400: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: f0
Fixed heap redzone: fd
Stack left redzone: fe
Stack mid redzone: ff
Stack right redzone: fb
Stack after return: f5
Stack use after scope: f6
Global redzone: gg
Global init order: ff
Poisoned by user: ff
Container overflow: fc
Array cookie: cc
```

1st COMMAND MEANING:

1. **cd:** This stands for "change directory", and it's used to navigate between directories or folders within a file system.
2. **outcmplog/default/crashes/:** This is the path to the directory you want to change into. It specifies the "crashes" directory within the "default" directory, which is inside the "outcmplog" directory.

2nd COMMAND MEANING:

1. `./imgRead aflplus`: The name of the binary that crashed.
`outcmplog/default/crashes/id:000005,sig:06,src:000000,time:4079,execs:4696,op:havoc,rep:2`
2. 8: This is the path and filename of the specific crash file generated by AFL++. It contains several pieces of information:

- 3. **id:000005**: The unique identifier for this crash.
- 4. **sig:06**: The signal number that caused the crash (in this case, signal 6).
- 5. **src:000000**: The AFL++ internal source location identifier (not directly related to the source code).
- 6. **time:4079**: The time (in milliseconds) since AFL++ started fuzzing when the crash occurred.
- 7. **execs:4696**: The number of executions (test cases) that were performed before the crash was discovered.
- 8. **op:havoc**: The AFL++ mutation strategy that caused the crash (in this case, "havoc" indicates a basic mutation).
- .rep:28**: The AFL++ repetition count, indicating how many times this crash was encountered during fuzzing.

CODE

```

/*
Author: Hardik Shah
Email: hardik05@gmail.com
Web: http://hardik05.wordpress.com
*/

//a vulnerable c program to explain common vulnerability types
//fuzz with AFL

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

```



```

struct Image
{
    char header[4];
    int width;
    int height;
    char data[10];
};

void stack_operation(){
    char buff[0x1000];
    while(1){
        stack_operation();
    }
}

int ProcessImage(char* filename){
    FILE *fp;
    struct Image img;

    fp = fopen(filename,"r");    //Statement 1

    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.\r\n");
        exit(0);
    }

    while(fread(&img,sizeof(img),1,fp)>0)
    {
        //if(strcmp(img.header,"IMG")==0)
        //{
            printf("\n\tHeader\twidth\theight\tdata\t\r\n");

        printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);

        //integer overflow 0x7FFFFFFF+1=0
        //0x7FFFFFFF+2 = 1
        //will cause very large/small memory allocation.
        int size1 = img.width + img.height;
        char* buff1=(char*)malloc(size1);

```

```

//heap buffer overflow
memcpy(buff1,img.data,sizeof(img.data));
free(buff1);
//double free
if (size1/2==0){
    free(buff1);
}
else{
    //use after free
    if(size1/3 == 0){
        buff1[0]='a';
    }
}

//integer underflow 0-1=-1
//negative so will cause very large memory allocation
int size2 = img.width - img.height+100;
//printf("Size1:%d",size1);
char* buff2=(char*)malloc(size2);

//heap buffer overflow
memcpy(buff2,img.data,sizeof(img.data));

//divide by zero
int size3= img.width/img.height;
//printf("Size2:%d",size3);

char buff3[10];
char* buff4 =(char*)malloc(size3);
memcpy(buff4,img.data,sizeof(img.data));

//OOBR read bytes past stack/heap buffer
char OOBRead = buff3[size3];
char OOBRead_heap = buff4[size3];

//OOBW write bytes past stack/heap buffer
buff3[size3]='c';
buff4[size3]='c';

if(size3>10){
    //memory leak here
    buff4=0;
}

```

```

    }
    else{
        free(buff4);
    }
    int size4 = img.width * img.height;
    if(size4/2==0){
        //stack exhaustion here
        stack_operation();
    }
    else{
        //heap exhaustion here
        char *buff5;
        do{
            buff5 = (char*)malloc(size4);
        }while(buff5);
    }
    free(buff2);
//}
//else
//    printf("invalid header\r\n");

}
fclose(fp);
return 0;
}

int main(int argc,char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "no input file\n");
        exit(-1);
    }
    ProcessImage(argv[1]);
    return 0;
}

```

References

1. AFLplusplus. (n.d.). *AFLplusplus/aflplusplus: The fuzzer AFL++ is AFL with community patches, QEMU 5.1 upgrade, collision-free coverage, enhanced LAF-Intel & redqueen, aflfast++ power schedules, MOPT mutators, unicorn_mode, and a lot more!*. GitHub.
<https://github.com/AFLplusplus/AFLplusplus>
2. Shah, H. (n.d.). *Fuzzing_in - Hardik Shah*. YouTube.
<https://www.youtube.com/@MrHardik05/playlists>
3. *Hardik05 - overview*. GitHub. (n.d.). <https://github.com/hardik05/>

4. *How to install AFL++ on Kali Linux*. Installati.one. (2023, June 1).

[https://installati.one/install-afl++-](https://installati.one/install-afl++-kalilinux/#:~:text=afl%2B%2B%20is%3A%20American%20fuzzy%20lop%20is%20a%20fuzzer,improves%20the%20functional%20coverage%20for%20the%20fuzzed%20code.)

[kalilinux/#:~:text=afl%2B%2B%20is%3A%20American%20fuzzy%20lop%20is%20a%20fuzzer,improves%20the%20functional%20coverage%20for%20the%20fuzzed%20code.](https://installati.one/install-afl++-kalilinux/#:~:text=afl%2B%2B%20is%3A%20American%20fuzzy%20lop%20is%20a%20fuzzer,improves%20the%20functional%20coverage%20for%20the%20fuzzed%20code.)

5. Nesbo, E. (2022, June 26). *What is fuzzing in cybersecurity?*. MUO.

<https://www.makeuseof.com/what-is-fuzzing/>

6. *Penetration testing and ethical hacking linux distribution*. Kali Linux. (2024, March 1).

<https://www.kali.org/>

7. *VMware workstation player | vmware*. VMware. (n.d.).

<https://www.vmware.com/products/workstation-player.html>

8. Shah, H. (2022, March 11). *[fuzzing with aflplusplus] how to fuzz arm and MIPS binaries with QEMU and AFL++*. YouTube.

<https://www.youtube.com/watch?v=0iyviukkANY>

9. AFLplusplus. (n.d.). *AFLplusplus/instrumentation/readme.cmplog.md at stable aflplusplus/aflplusplus*. GitHub.

<https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md>

10. Shah, H. (2022, January 3). *[fuzzing with aflplusplus] how to use CmpLog feature to fuzz a binary*. YouTube.

<https://www.youtube.com/watch?v=qZyHphVhMfQ&list=PLHGgqcJIME5mQmn9pYGCPrSVactbIwwtP&index=4>