Fuzzing101

An step by step fuzzing tutorial. A GitHub Security Lab initiative https://securitylab.github.com/

Table of Contents

Exercise 1 - Xpdf

For this exercize we will fuzz **Xpdf** PDF viewer. The goal is to find a crash/PoC for **CVE-2019-13288** in XPDF 3.02.

CVE-2019-13288 is a vulnerability that may cause an infinite recursion via a crafted file.

Since each called function in a program allocates a stack frame on the stack, if a a function is recursively called so many times it can lead to stack memory exhaustion and program crash.

As a result, a remote attacker can leverage this for a DoS attack.

You can find more information about Uncontrolled Recursion vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/674.html

What you will learn

After completing this exercise, you will know the basis of fuzzing with AFL, such as:

- Compiling a target application with instrumentation
- Running a fuzzer (afl-fuzz)
- Triaging crashes with a debugger (GDB)

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😀

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir fuzzing_xpdf && cd fuzzing_xpdf/
```

To get your environment fully ready, you may need to install some additional tools (namely make and gcc)

```
sudo apt install build-essential
```

Download Xpdf 3.02:

```
wget https://dl.xpdfreader.com/old/xpdf-3.02.tar.gz
tar -xvzf xpdf-3.02.tar.gz
```

Build Xpdf:

```
cd xpdf-3.02
sudo apt update && sudo apt install -y build-essential gcc
./configure --prefix="$HOME/fuzzing_xpdf/install/"
make
make install
```

Time to test the build. First of all, You'll need to download a few PDF examples:

```
cd $HOME/fuzzing_xpdf
mkdir pdf_examples && cd pdf_examples
wget https://github.com/mozilla/pdf.js-sample-
files/raw/master/helloworld.pdf
wget http://www.africau.edu/images/default/sample.pdf
wget https://www.melbpc.org.au/wp-content/uploads/2017/10/small-example-
pdf-file.pdf
```

Now, we can test the pdfinfo binary with:

```
$HOME/fuzzing_xpdf/install/bin/pdfinfo -box -meta
$HOME/fuzzing_xpdf/pdf_examples/helloworld.pdf
```

You should see something like this:

```
antonio@ubuntu:~/fuzzing_xpdf/pdf_examples$ $HOME/fuzzing_xpdf/install/bin/pdfinfo -box
-meta $HOME/fuzzing_xpdf/pdf_examples/helloworld.pdf
Tagged:         no
Pages:          1
Encrypted:      no
Page size:      200 x 200 pts
MediaBox:           0.00     0.00   200.00   200.00
CropBox:            0.00     0.00   200.00   200.00
BleedBox:           0.00     0.00   200.00   200.00
TrimBox:            0.00     0.00   200.00   200.00
ArtBox:             0.00     0.00   200.00   200.00
File size:      678 bytes
Optimized:      no
PDF version:    1.7
```

Install AFL++

For this course, we're going to use the latest version of AFL++ fuzzer.

You can install everything in two ways:

Local installation (recommended option)

Install the dependencies

```
sudo apt-get update
sudo apt-get install -y build-essential python3-dev automake git flex
bison libglib2.0-dev libpixman-1-dev python3-setuptools
sudo apt-get install -y lld-11 llvm-11 llvm-11-dev clang-11 || sudo apt-
get install -y lld llvm llvm-dev clang
sudo apt-get install -y gcc-$(gcc --version|head -n1|sed 's/.* //'|sed
's/\..*//')-plugin-dev libstdc++-$(gcc --version|head -n1|sed 's/.*
//'|sed 's/\..*//')-dev
```

Checkout and build AFL++

```
cd $HOME
git clone https://github.com/AFLplusplus/AFLplusplus && cd AFLplusplus
export LLVM_CONFIG="llvm-config-11"
make distrib
sudo make install
```

Docker image

Install docker

```
sudo apt install docker
```

Pull the image

```
docker pull aflplusplus/aflplusplus
```

Launch the AFLPlusPlus docker container:

```
docker run -ti -v $HOME:/home aflplusplus/aflplusplus
```

and then type

```
export $HOME="/home"
```

Now if all went well, you should be able to run afl-fuzz. Just type

```
afl-fuzz
```

and you should see something like this:

```
afl-fuzz++3.12c based on afl by Michal Zalewski and a large online community

afl-fuzz [ options ] -- /path/to/fuzzed_app [ ... ]

Required parameters:
  -i dir        - input directory with test cases
  -o dir        - output directory for fuzzer findings

Execution control settings:
  -p schedule   - power schedules compute a seed's performance score:
                  fast(default), explore, exploit, seek, rare, mmopt, coe, lin
                  quad -- see docs/power_schedules.md
  -f file       - location read by the fuzzed program (default: stdin or @@)
  -t msec       - timeout for each run (auto-scaled, default 1000 ms). Add a '+'
                  to auto-calculate the timeout, the value being the maximum.
  -m megs       - memory limit for child process (0 MB, 0 = no limit [default])
  -Q            - use binary-only instrumentation (QEMU mode)
  -U            - use unicorn-based instrumentation (Unicorn mode)
  -W            - use qemu-based instrumentation with Wine (Wine mode)
```

Meet AFL++

AFL is a **coverage-guided fuzzer**, which means that it gathers coverage information for each mutated input in order to discover new execution paths and potential bugs. When source code is available AFL can use instrumentation, inserting function calls at the beginning of each basic block (functions, loops, etc.)

To enable instrumentation for our target application, we need to compile the code with AFL's compilers.

First of all, we're going to clean all previously compiled object files and executables:

```
rm -r $HOME/fuzzing_xpdf/install
cd $HOME/fuzzing_xpdf/xpdf-3.02/
make clean
```

And now we're going to build xpdf using the **afl-clang-fast** compiler:

```
export LLVM_CONFIG="llvm-config-11"
CC=$HOME/AFLplusplus/afl-clang-fast CXX=$HOME/AFLplusplus/afl-clang-fast++
./configure --prefix="$HOME/fuzzing_xpdf/install/"
make
make install
```

Now, you can run the fuzzer with the following command:

```
afl-fuzz -i $HOME/fuzzing_xpdf/pdf_examples/ -o $HOME/fuzzing_xpdf/out/ -s
123 -- $HOME/fuzzing_xpdf/install/bin/pdftotext @@
$HOME/fuzzing_xpdf/output
```

A brief explanation of each option:

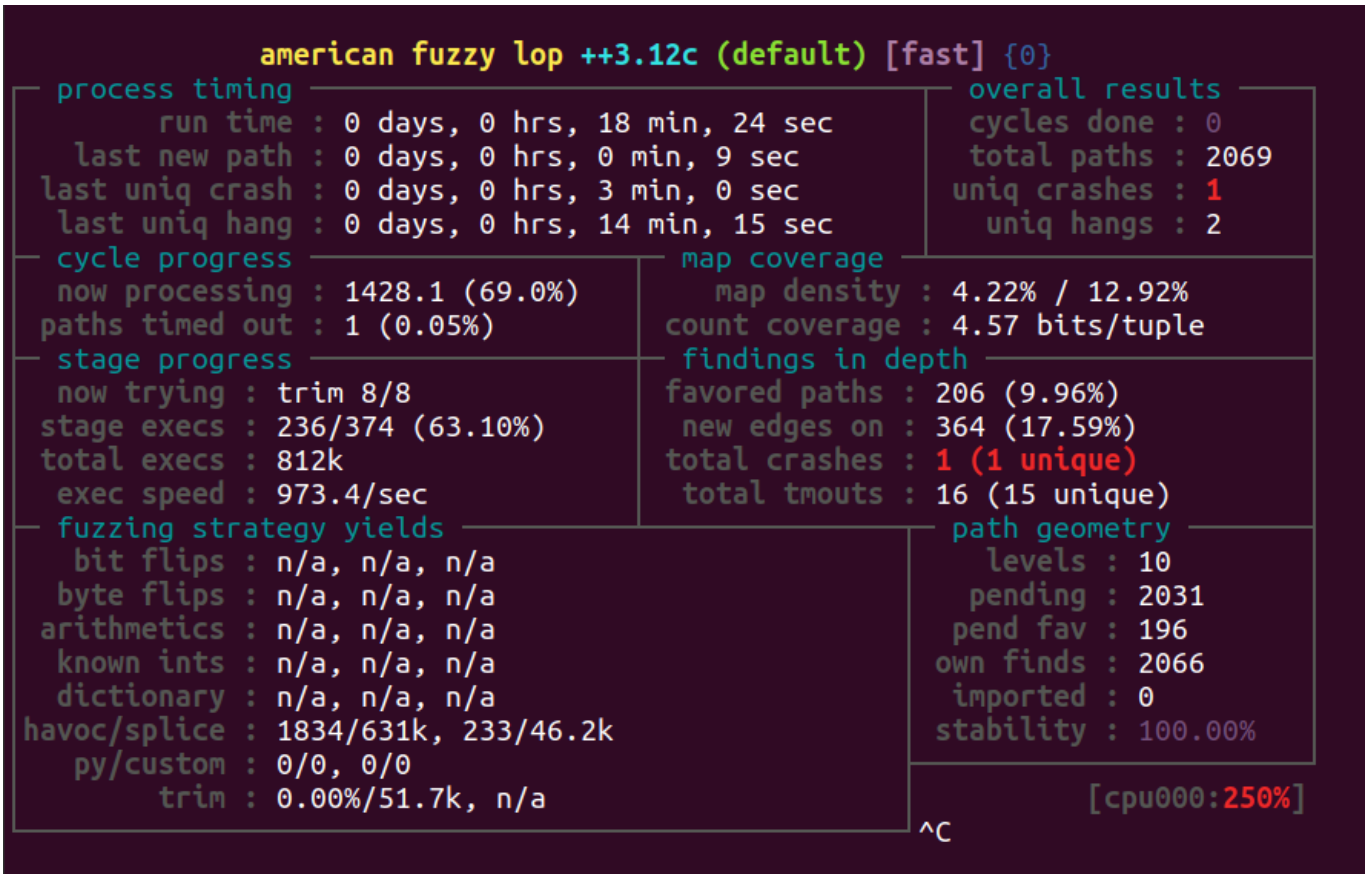
- *-i* indicates the directory where we have to put the input cases (a.k.a file examples)
- *-o* indicates the directory where AFL++ will store the mutated files
- *-s* indicates the static random seed to use
- @@ is the placeholder target's command line that AFL will substitute with each input file name

So, basically the fuzzer will run the command `$HOME/fuzzing_xpdf/install/bin/pdftotext <input-file-name> $HOME/fuzzing_xpdf/output` for each different input file.

If you receive a message like *"Hmm, your system is configured to send core dump notifications to an external utility...*", just do:

```
sudo su
echo core >/proc/sys/kernel/core_pattern
exit
```

After a few minutes you should see something like this:



You can see the **"uniq. crashes"** value in red, showing the number of unique crashes found. You can find these crashes files in the `$HOME/fuzzing_xpdf/out/` directory. You can stop the fuzzer once the first crash is found, this is the one we'll work on. It can take up to one or two hours depending on your machine performance, before you get a crash.

At this stage, you have learned:

- How to compile a target using afl compiler with instrumentation
- How to launch afl++
- How to detect unique crashes of your target

So what's next? We don't have any information on this bug, just a crash of the program... It's time for debugging and triage!

Do it yourself!

In order to complete this exercise, you need to:

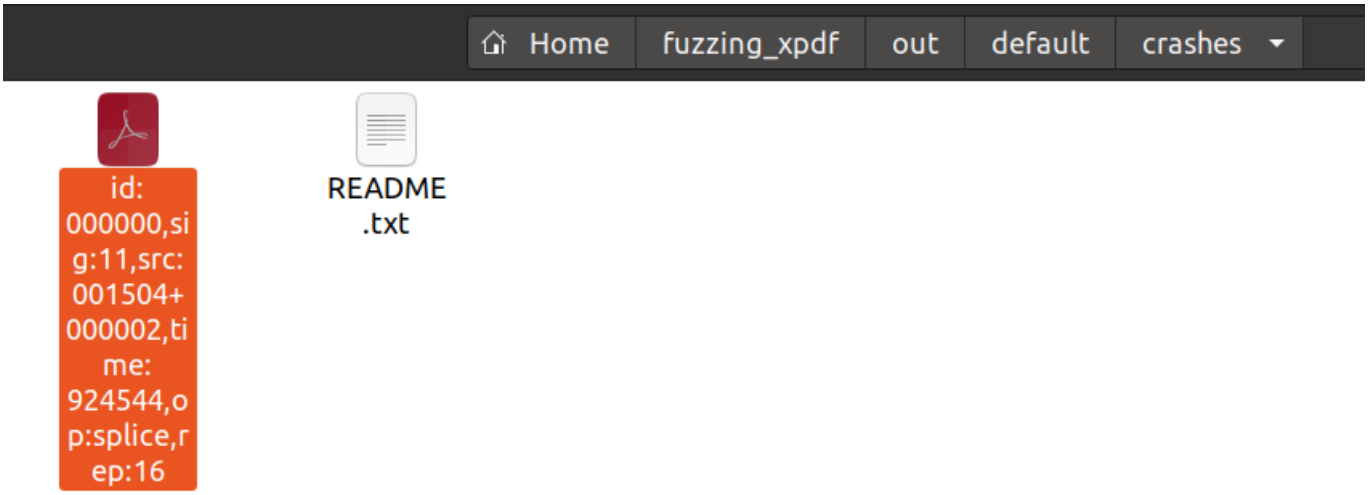
1. Reproduce the crash with the indicated file
2. Debug the crash to find the problem
3. Fix the issue

Estimated time = 120 mins

SPOILER ALERT! : Solution inside

Reproduce the crash

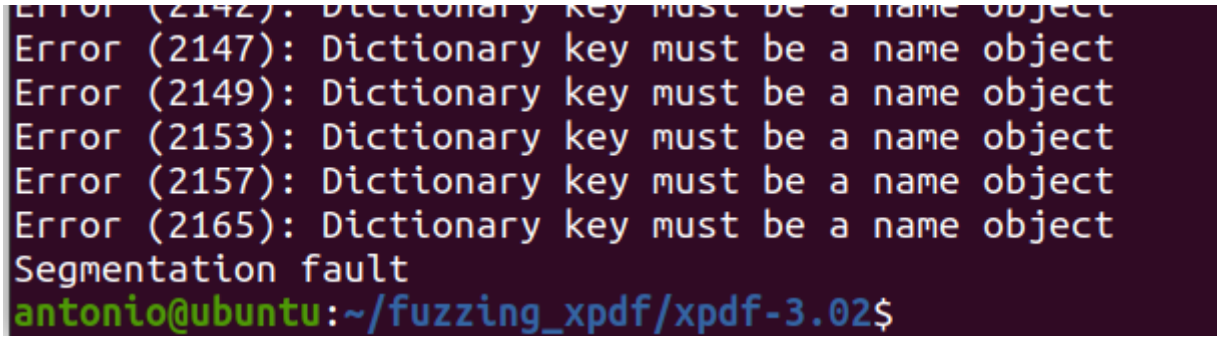
Locate the file corresponding to the crash in the `$HOME/fuzzing_xpdf/out/` directory. The filename is something like `id:000000,sig:11,src:001504+000002,time:924544,op:splice,rep:16`



Pass this file as input to pdftotext binary

```
$HOME/fuzzing_xpdf/install/bin/pdftotext
'$HOME/fuzzing_xpdf/out/default/crashes/<your_filename>'
$HOME/fuzzing_xpdf/output
```

It will cause a segmentation fault and results in a crash of the program.



Triage

Use gdb to figure out why the program crashes with this input.

- **You can take a look at http://people.cs.pitt.edu/~mosse/gdb-note.html for a good brief primer on GDB**

First of all, you need to rebuild Xpdf with debug info to get a symbolic stack trace:

```
rm -r $HOME/fuzzing_xpdf/install
cd $HOME/fuzzing_xpdf/xpdf-3.02/
make clean
CFLAGS="-g -O0" CXXFLAGS="-g -O0" ./configure --
prefix="$HOME/fuzzing_xpdf/install/"
```

```
make
make install
```

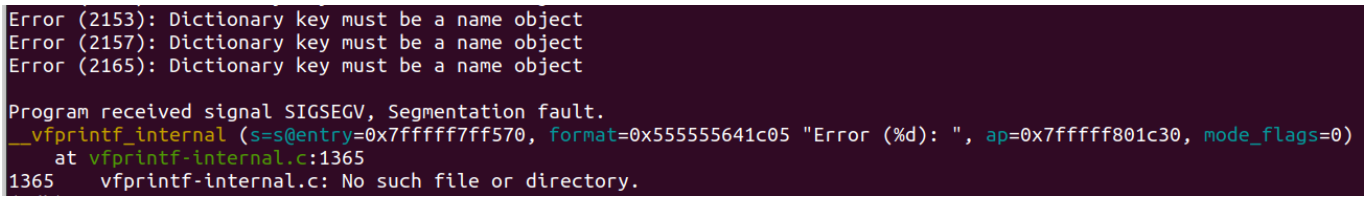
Now, you can run GDB:

```
gdb --args $HOME/fuzzing_xpdf/install/bin/pdftotext
$HOME/fuzzing_xpdf/out/default/crashes/<your_filename>
$HOME/fuzzing_xpdf/output
```

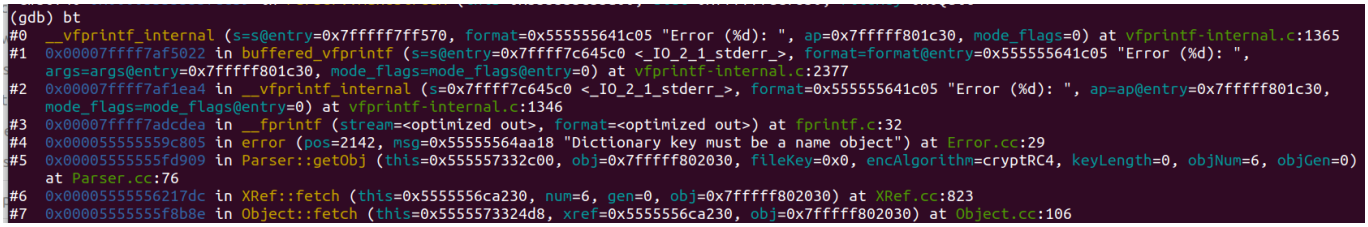
And then, type inside GDB:

```
>> run
```

If all went well, you should see the following output:



Then type `bt` to get the backtrace:



Scroll the call stack and you will see many calls of the "Parser::getObj" method that seems to indicate an infinite recursion. If you go to https://www.cvedetails.com/cve/CVE-2019-13288/ you can see that the description matches with the backtrace we got from GDB.

Fix the issue

The last step of the exercise is to fix the bug! Rebuild your target after the fix and check that your use case is not causing a segmentation fault anymore. This last part is left as exercise for the student.

Alternatively, you can download Xpdf 4.02 where the bug is already fixed, and check that the segmentation fault disappears.

Exercise 2 - libexif

This time we will fuzz **libexif** EXIF parsing library. The goal is to find a crash/PoC for **CVE-2009-3895** and another crash for **CVE-2012-2836** in libexif 0.6.14.

CVE-2009-3895 is a heap-based buffer overflow that can be triggered with an invalid EXIF image.

A heap-based buffer overflow is a type of buffer overflow that occurs in the heap data area, and it's usually related to explicit dynamic memory management (allocation/deallocation with malloc() and free() functions).

As a result, a remote attacker can exploit this issue to execute arbitrary code within the context of an application using the affected library.

You can find more information about Heap-based buffer oveflow vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/122.html

---

CVE-2012-2836 is an Out-of-bounds Read vulneratibily that can be triggered via an image with crafted EXIF tags.

An Out-of-bounds Read is a vulnerability that occurs when the program reads data past the end, or before the beginning, of the intended buffer.

As a result, it allows remote attackers to cause a denial of service or possibly obtain potentially sensitive information from process memory.

You can find more information about Out-of-bounds Read vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/125.html

What you will learn

Once you complete this exercise you will know how:

- To fuzz a library using an external application
- To use **afl-clang-lto**, a collision free instrumentation that is faster and provides better results than *afl-clang-fast*
- To use Eclipse IDE as an easy alternative to GDB console for triaging

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Do it yourself!

In order to complete this exercise, you need to:

1. Find an interface application that makes use of the libexif library
2. Create a seed corpus of exif samples
3. Compile libexif and the chosen application to be fuzzed using afl-clang-lto
4. Fuzz libexif until you have a few unique crashes
5. Triage the crashes to find a PoC for each vulnerability
6. Fix the issues

Estimated time = 6 hours

SPOILER ALERT! : Solution inside

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project we want to fuzz:

```
cd $HOME
mkdir fuzzing_libexif && cd fuzzing_libexif/
```

Download and uncompress libexif-0.6.14:

```
wget https://github.com/libexif/libexif/archive/refs/tags/libexif-0_6_14-release.tar.gz
tar -xzvf libexif-0_6_14-release.tar.gz
```

Build and install libexif:

```
cd libexif-libexif-0_6_14-release/
sudo apt-get install autopoint libtool gettext libpopt-dev
autoreconf -fvi
./configure --enable-shared=no --prefix="$HOME/fuzzing_libexif/install/"
make
make install
```

Choosing an interface application

Since libexif is a library, we'll need another application that makes use of this library and which will be fuzzed. For this task we're going to use **exif command-line**. Type the following for download and uncompressing exif command-line 0.6.15:

```
cd $HOME/fuzzing_libexif
wget https://github.com/libexif/exif/archive/refs/tags/exif-0_6_15-
release.tar.gz
tar -xzvf exif-0_6_15-release.tar.gz
```

Now, we can build and install exif command-line utility:

```
cd exif-exif-0_6_15-release/
autoreconf -fvi
./configure --enable-shared=no --prefix="$HOME/fuzzing_libexif/install/"
PKG_CONFIG_PATH=$HOME/fuzzing_libexif/install/lib/pkgconfig
make
make install
```

To test everything is working properly, just type:

```
$HOME/fuzzing_libexif/install/bin/exif
```

and you should see something like that

```
antonio@ubuntu:~/fuzzing_libexif/exif-exif-0_6_15-release$ $HOME/fuzzing_libexif/install/bin/exif
Usage: exif [OPTION...] file
  -v, --version                  Display software version
  -i, --ids                      Show IDs instead of tag names
  -t, --tag=tag                  Select tag
      --ifd=IFD                  Select IFD
  -l, --list-tags                List all EXIF tags
  -|, --show-mnote               Show contents of tag MakerNote
      --remove                   Remove tag or ifd
  -s, --show-description         Show description of tag
  -e, --extract-thumbnail        Extract thumbnail
  -r, --remove-thumbnail         Remove thumbnail
  -n, --insert-thumbnail=FILE    Insert FILE as thumbnail
  -o, --output=FILE              Write data to FILE
      --set-value=STRING         Value
  -m, --machine-readable         Output in a machine-readable (tab delimited) format
  -x, --xml-output               Output in a XML format
  -d, --debug                    Show debugging messages

Help options:
  -?, --help                     Show this help message
      --usage                    Display brief usage message
```

Seed corpus creation

Now we need to get some exif samples. We're gonna use the sample images from the following repo: https://github.com/ianare/exif-samples. You can download it with:

```
cd $HOME/fuzzing_libexif
wget https://github.com/ianare/exif-samples/archive/refs/heads/master.zip
unzip master.zip
```

As an example, we can do:

```
$HOME/fuzzing_libexif/install/bin/exif $HOME/fuzzing_libexif/exif-samples-
master/jpg/Canon_40D_photoshop_import.jpg
```

And the output should look like

```
antonio@ubuntu:~/fuzzing_libexif/exif-samples-master$ $HOME/fuzzing_libexif/install/bin/exif $HOME/fuzzing_li
bexif/exif-samples-master/jpg/Canon_40D_photoshop_import.jpg
EXIF tags in '/home/antonio/fuzzing_libexif/exif-samples-master/jpg/Canon_40D_photoshop_import.jpg' ('Motorol
a' byte order):
-------------------+-------------------------------------------------------
Tag                |Value
-------------------+-------------------------------------------------------
Orientation        |top - left
x-Resolution       |300.00
y-Resolution       |300.00
Resolution Unit    |Inch
Software           |GIMP 2.4.5
Date and Time      |2008:07:31 10:05:49
Compression        |JPEG compression
x-Resolution       |72.00
y-Resolution       |72.00
Resolution Unit    |Inch
Color Space        |sRGB
PixelXDimension    |100
PixelYDimension    |77
-------------------+-------------------------------------------------------
EXIF data contains a thumbnail (2022 bytes).
```

Afl-clang-lto instrumentation

Now we're going to build libexif using **afl-clang-lto** as the compiler.

```
rm -r $HOME/fuzzing_libexif/install
cd $HOME/fuzzing_libexif/libexif-libexif-0_6_14-release/
make clean
export LLVM_CONFIG="llvm-config-11"
CC=afl-clang-lto ./configure --enable-shared=no --
prefix="$HOME/fuzzing_libexif/install/"
make
make install
```

```
cd $HOME/fuzzing_libexif/exif-exif-0_6_15-release
make clean
export LLVM_CONFIG="llvm-config-11"
CC=afl-clang-lto ./configure --enable-shared=no --
prefix="$HOME/fuzzing_libexif/install/"
PKG_CONFIG_PATH=$HOME/fuzzing_libexif/install/lib/pkgconfig
make
make install
```

As you can see, I used **afl-clang-lto** instead of *afl-clang-fast*. In general, *afl-clang-lto* is the best option out there because it's a collision-free instrumentation and it's faster than *afl-clang-fast*.

If you are not sure about when to use *afl-clang-lto* or *afl-clang-fast* you can check the following diagram extracted from AFLplusplus : instrumenting that target

```
+--------------------------------+
| clang/clang++ 11+ is available | --> use LTO mode (afl-clang-lto/afl-
clang-lto++)
+--------------------------------+     see [instrumentation/README.lto.md]
(instrumentation/README.lto.md)
    |
    | if not, or if the target fails with LTO afl-clang-lto/++
    |
    v
+--------------------------------+
| clang/clang++ 6.0+ is available | --> use LLVM mode (afl-clang-fast/afl-
clang-fast++)
+--------------------------------+     see
[instrumentation/README.llvm.md](instrumentation/README.llvm.md)
    |
    | if not, or if the target fails with LLVM afl-clang-fast/++
    |
    v
 +--------------------------------+
 | gcc 5+ is available            | -> use GCC_PLUGIN mode (afl-gcc-
fast/afl-g++-fast)
 +--------------------------------+    see
```

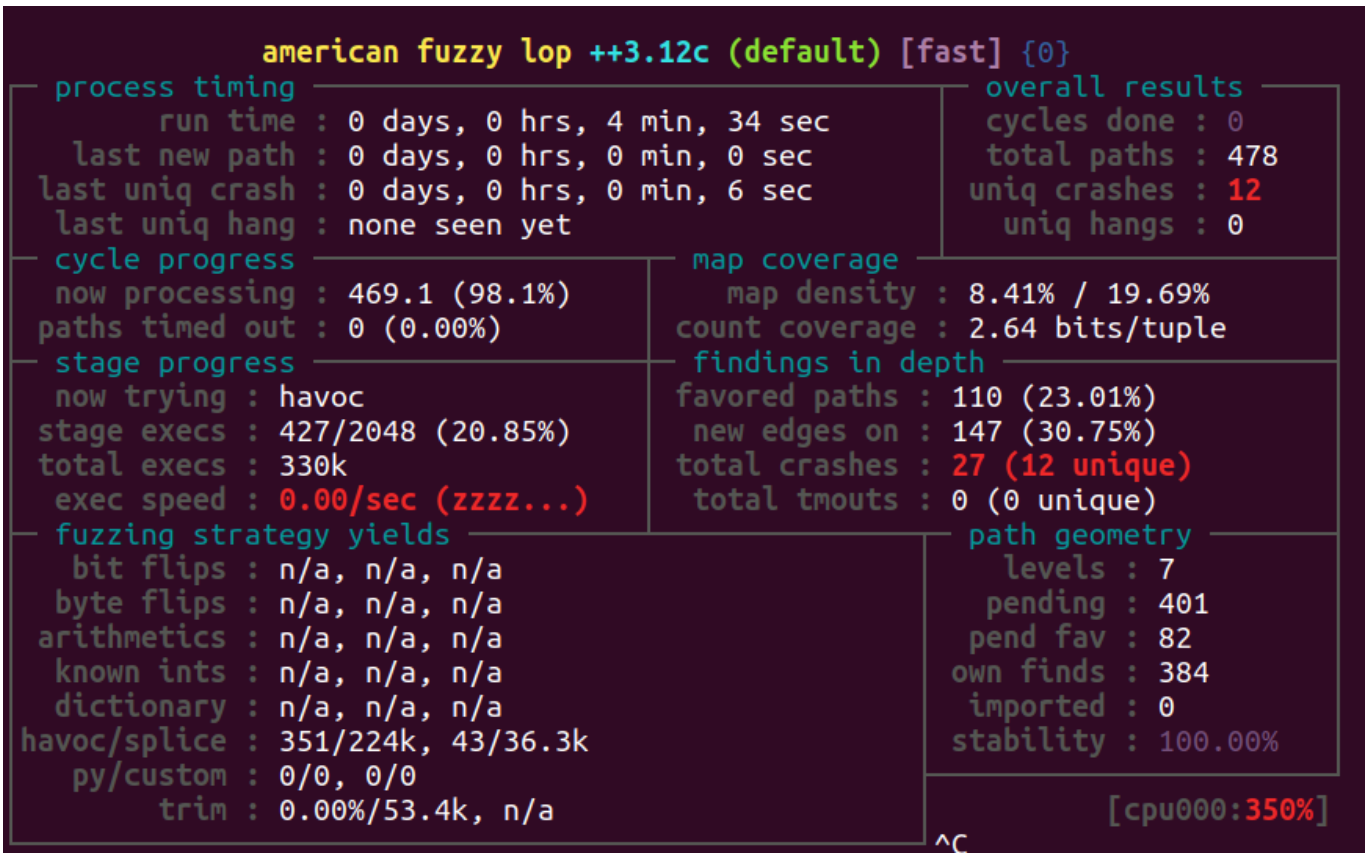
```
[instrumentation/README.gcc_plugin.md]
(instrumentation/README.gcc_plugin.md) and

[instrumentation/README.instrument_list.md]
(instrumentation/README.instrument_list.md)
    |
    | if not, or if you do not have a gcc with plugin support
    |
    v
   use GCC mode (afl-gcc/afl-g++) (or afl-clang/afl-clang++ for clang)
```

Fuzzing time

Now, you can run the fuzzer with the following command:

```
afl-fuzz -i $HOME/fuzzing_libexif/exif-samples-master/jpg/ -o
$HOME/fuzzing_libexif/out/ -s 123 --
$HOME/fuzzing_libexif/install/bin/exif @@
```

After a few minutes, you should have multiple crashes:



Triage

Eclipse setup

In the exercise 1 we learned how to use GDB console for triaging crashes. In this second exercise, we'll see how to use Eclipse-CDT for debugging purposes.

First of all, we can download it from: https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2021-03/R/eclipse-cpp-2021-03-R-linux-gtk-x86_64.tar.gz

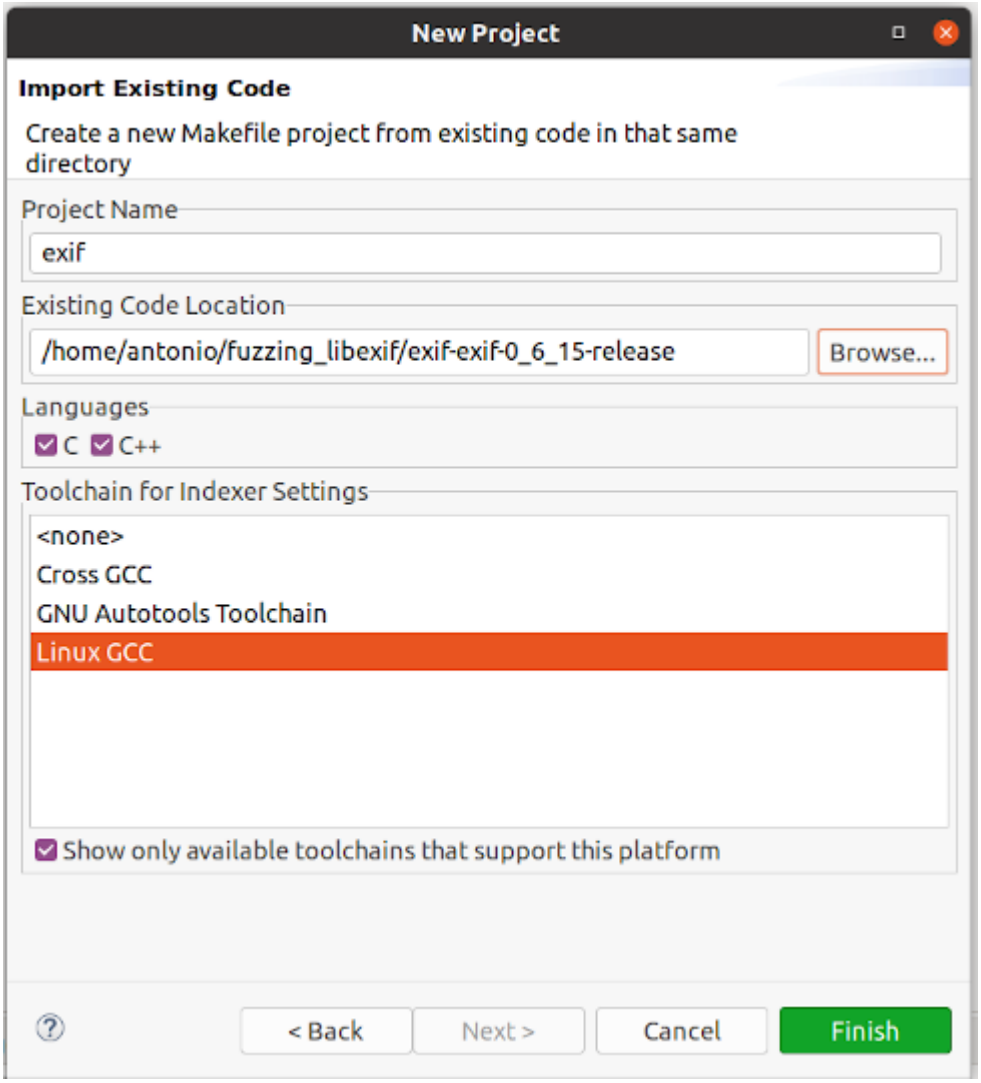
If JAVA-SDK is not installed on your system, you can install it on Ubuntu with the following command:

```
sudo apt install default-jdk
```

Then we can extract it with:

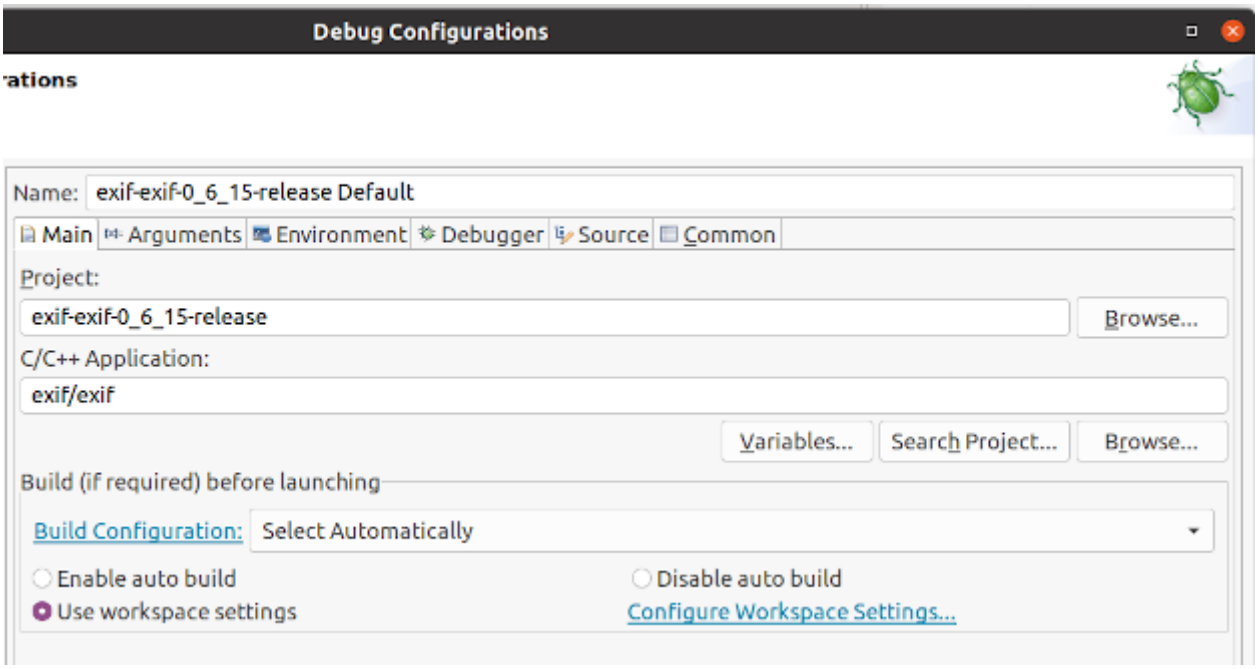
```
tar -xzvf eclipse-cpp-2021-03-R-linux-gtk-x86_64.tar.gz
```

Once we have started Eclipse-CDT, we need to import our source code into the Project Explorer. For that, we need to go to File -> Import -> and then we need to choose C/C++ -> "Existing code as makefile project". Then we need to select "Linux GCC" and browse for the Exif source code folder:

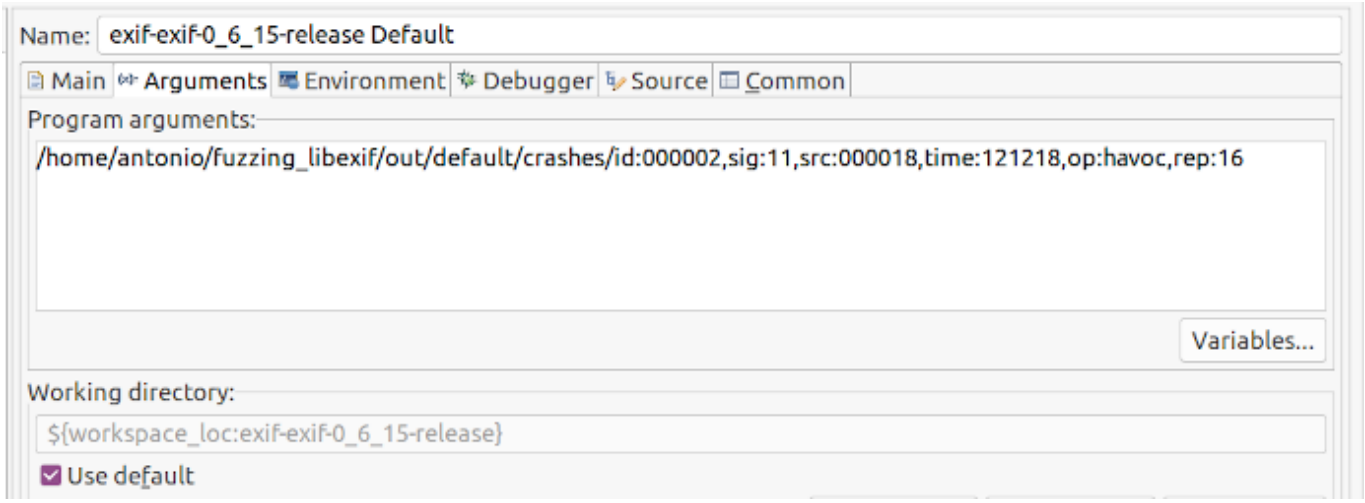


If all went well, you should be able to see the "exif" folder into the Project explorer tab.

Now we're going to configure the debug parameters. For that we need to go to `Run -> Debug Configurations`. Then we select our exif project and browse for the exif binary:

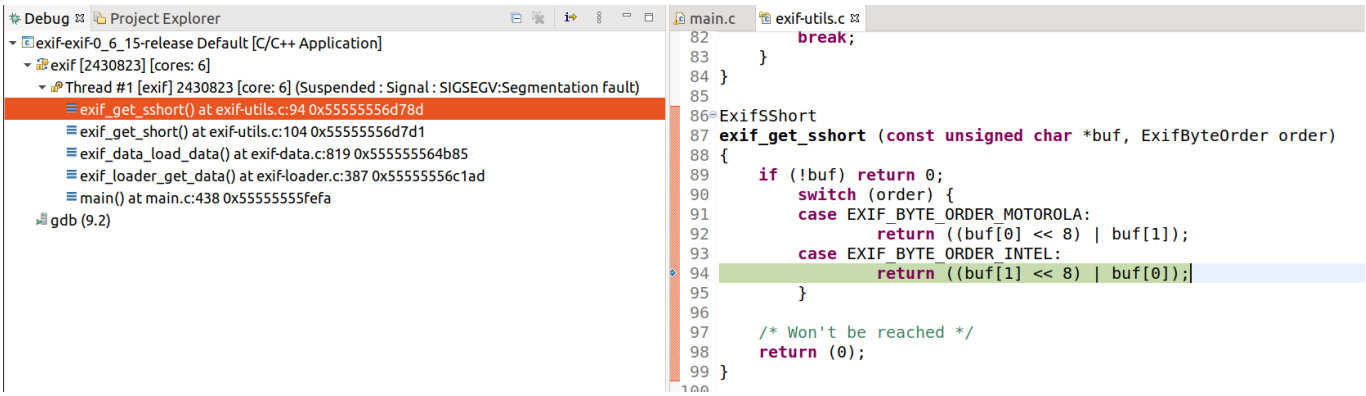


Then we need to set the input arguments. For that, go to the `"Arguments"` tab and set the path of one of the AFL crashes.



Finally, we only need to click on `"Debug"` for starting the debugging sesion and the program will stop at the beginning of the main function.

Having got to this point, we only need to click on `Run -> Resume` and the execution will stop when a segmentation fault is detected.



Fix the issues

The last step of the exercise is to fix both bugs. Rebuild your target after the fixes and check that your PoCs don't crash the program anymore. This last part is left as exercise for the student.

Solution inside

Official fixes:

- https://github.com/libexif/libexif/commit/8ce72b7f81e61ef69b7ad5bdfeff1516c90fa361
- https://github.com/libexif/libexif/commit/00986f6fa979fe810b46e376a462c581f9746e06

Alternatively, you can download a newer version of libexif, and check that both bugs have been fixed.

Exercise 3 - TCPdump

In this exercise we will fuzz **TCPdump** packet analyzer. The goal is to find a crash/PoC for **CVE-2017-13028** in TCPdump 4.9.2.

CVE-2017-13028 is an Out-of-bounds Read vulneratibily that can be triggered via a BOOTP packet (Bootstrap Protocol).

An Out-of-bounds Read is a vulnerability that occurs when the program reads data past the end, or before the beginning, of the intended buffer.

As a result, it allows remote attackers to cause a denial of service or possibly obtain potentially sensitive information from process memory.

You can find more information about Out-of-bounds Read vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/125.html

What you will learn

Once you complete this exercise you will know:

- What is **ASan (Address Sanitizer)**, a runtime memory error detection tool
- How to use ASAN to fuzz a target
- How much easy is to triage the crashes using ASan

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Do it yourself!

In order to complete this exercise, you need to:

1. Find an efficient way to fuzz TCPdump
2. Try to figure out how to enable ASan for fuzzing
3. Fuzz TCPdump until you have a few unique crashes
4. Triage the crashes to find a PoC for the vulnerability
5. Fix the issue

Estimated time = 4 hours

SPOILER ALERT! : Solution inside

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir fuzzing_tcpdump && cd fuzzing_tcpdump/
```

Download and uncompress tcpdump-4.9.2.tar.gz

```
wget https://github.com/the-tcpdump-
group/tcpdump/archive/refs/tags/tcpdump-4.9.2.tar.gz
tar -xzvf tcpdump-4.9.2.tar.gz
```

We also need to download libpcap, a cross-platform library that is needed by TCPdump. Download and uncompress libpcap-1.8.0.tar.gz:

```
wget https://github.com/the-tcpdump-
group/libpcap/archive/refs/tags/libpcap-1.8.0.tar.gz
tar -xzvf libpcap-1.8.0.tar.gz
```

We need to rename `libpcap-libpcap-1.8.0` to `libpcap-1.8.0`. Otherwise, tcpdump doesn't find the `libpcap.a` local path:

```
mv libpcap-libpcap-1.8.0/ libpcap-1.8.0
```

Build and install libpcap:

```
cd $HOME/fuzzing_tcpdump/libpcap-1.8.0/
./configure --enable-shared=no
make
```

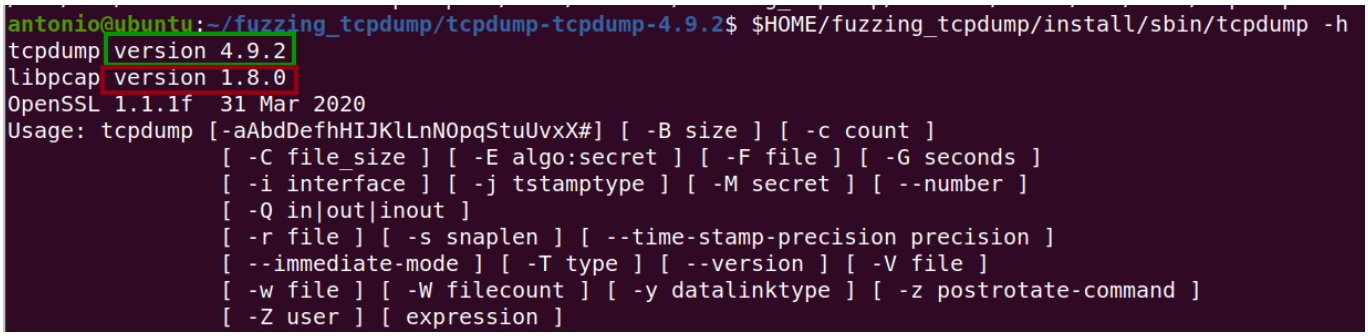
Now, we can build and install tcpdump:

```
cd $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.9.2/
./configure --prefix="$HOME/fuzzing_tcpdump/install/"
make
make install
```

To test everything is working properly, just type:

```
$HOME/fuzzing_tcpdump/install/sbin/tcpdump -h
```

and you should see something like that



Before continuing to the following step, check that your version numbers matches the numbers above

Seed corpus creation

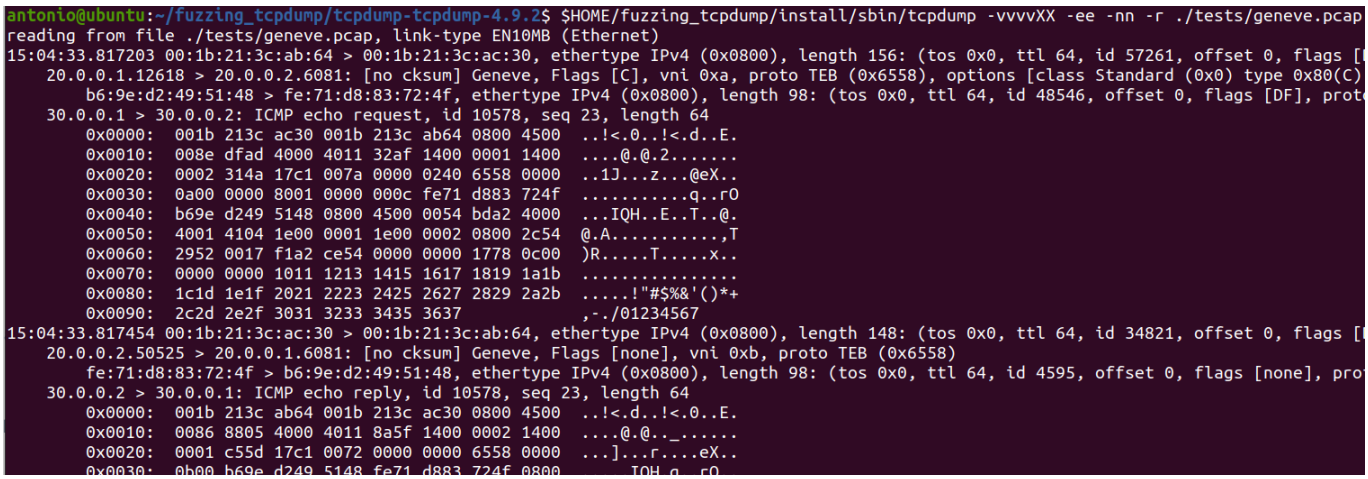
You can find a lot of .pcacp examples in the "./tests" folder. You can run these .pcap files with the following command-line:

```
$HOME/fuzzing_tcpdump/install/sbin/tcpdump -vvvvXX -ee -nn -r [.pcap file]
```

For example:

```
$HOME/fuzzing_tcpdump/install/sbin/tcpdump -vvvvXX -ee -nn -r
./tests/geneve.pcap
```

And the output should look like



AddressSanitizer

AddressSanitizer (ASan) is a fast memory error detector for C and C++. It was originally developed by Google (Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov) and first released in May 2011.

It consists of a compiler instrumentation module and a run-time library. The tool is capable of finding out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free, double-free and memory leaks bugs.

AddressSanitizer is open source and is integrated with the LLVM compiler tool chain starting from version 3.1. While it was originally developed as a project for LLVM, it has been ported to GCC and it is included into GCC versions >= 4.8

You can find more information about AddressSanitizer at the following link.

Build with ASan enabled

Now we're going to build tcpdump (and libpcap) with ASAN enabled.

First of all, we're going to clean all previously compiled object files and executables:

```
rm -r $HOME/fuzzing_tcpdump/install
cd $HOME/fuzzing_tcpdump/libpcap-1.8.0/
make clean

cd $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.9.2/
make clean
```

Now, we set `AFL_USE_ASAN=1` before calling `configure` and `make`:

```
cd $HOME/fuzzing_tcpdump/libpcap-1.8.0/
export LLVM_CONFIG="llvm-config-11"
CC=afl-clang-lto ./configure --enable-shared=no --
prefix="$HOME/fuzzing_tcpdump/install/"
```

```
AFL_USE_ASAN=1 make

cd $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.9.2/
AFL_USE_ASAN=1 CC=afl-clang-lto ./configure --
prefix="$HOME/fuzzing_tcpdump/install/"
AFL_USE_ASAN=1 make
AFL_USE_ASAN=1 make install
```

Afl-clang-lto compilation can take a few minutes to complete

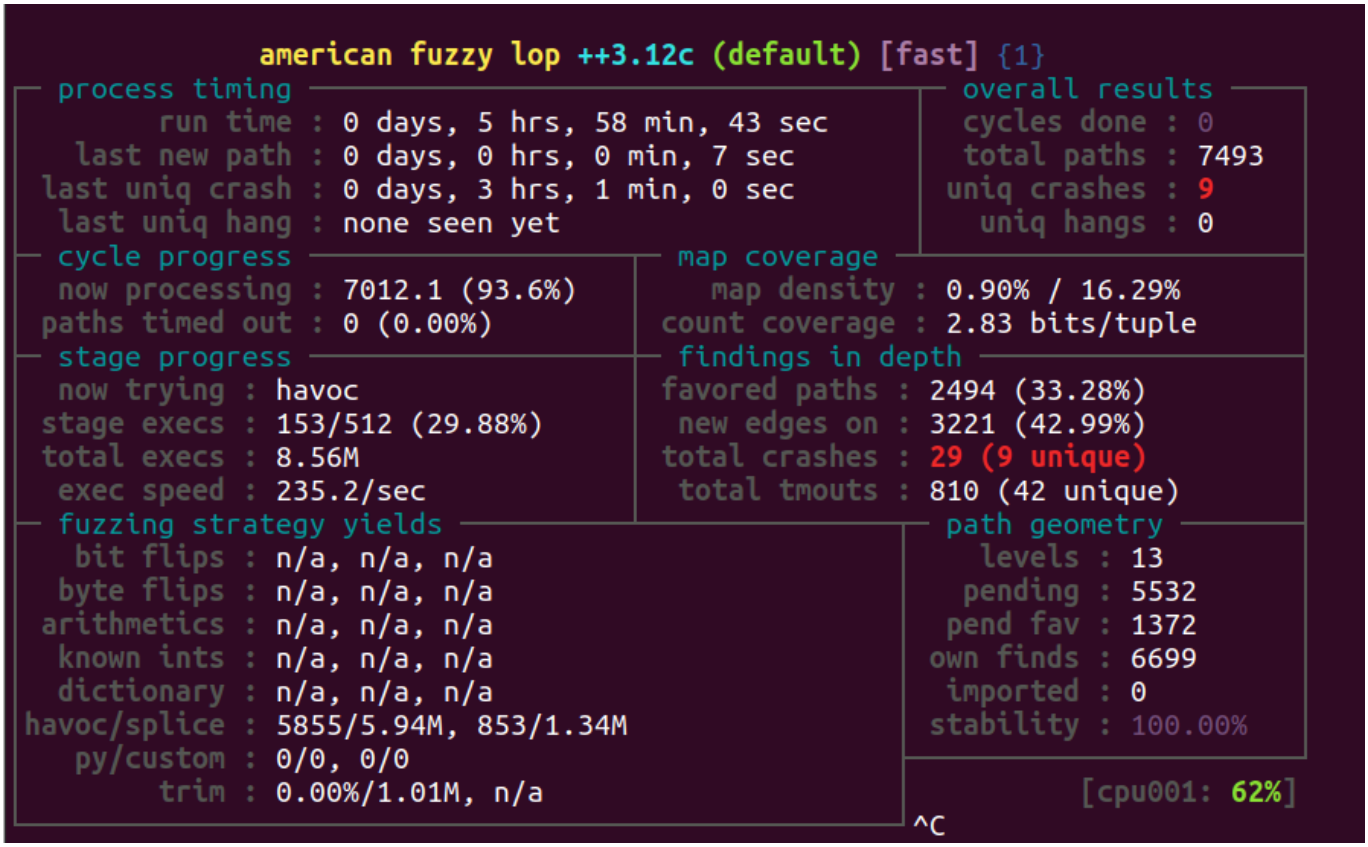
Fuzzing time

Now, you can run the fuzzer with the following command:

```
afl-fuzz -m none -i $HOME/fuzzing_tcpdump/tcpdump-tcpdump-4.9.2/tests/ -o
$HOME/fuzzing_tcpdump/out/ -s 123 --
$HOME/fuzzing_tcpdump/install/sbin/tcpdump -vvvvXX -ee -nn -r @@
```

Note: ASAN on 64-bit systems requests a lot of virtual memory. That's why I've set the flag "-m none" that disable memory limits in AFL

After a while, you should have multiple crashes:

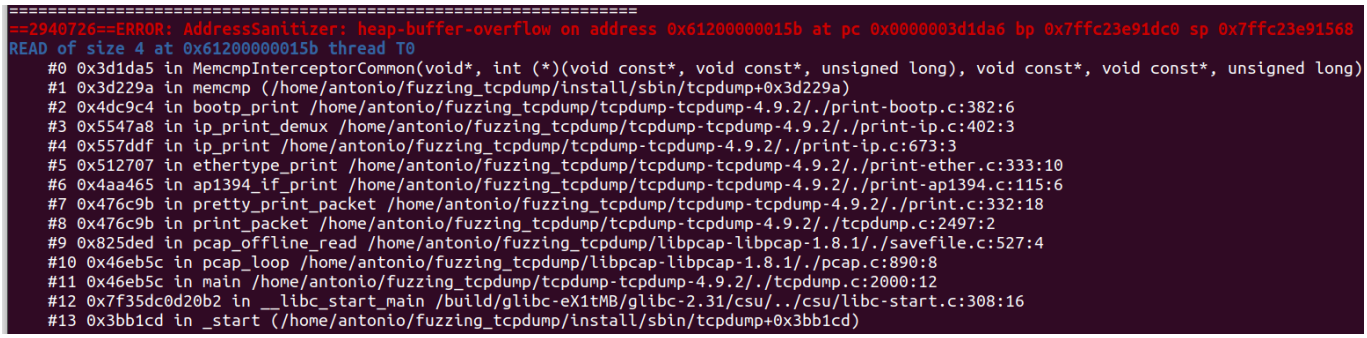


Triage

To debug a program built with ASan is so much easier than in the previous exercises. All you need to do is to feed the program with the crash file:

```
$HOME/fuzzing_tcpdump/install/sbin/tcpdump -vvvvXX -ee -nn -r
'/home/antonio/fuzzing_tcpdump/out/default/crashes/id:000000,sig:06,src:00
2318+001583,time:10357087,op:splice,rep:8'
```

and you will get a nice summary of the crash, including the execution trace:



Fix the issue

The last step of the exercise is to fix the bug! Rebuild your target after the fix and check that your PoC don't crash the program anymore. This last part is left as exercise for the student.

Solution inside

Official fix:

- https://github.com/the-tcpdump-group/tcpdump/commit/85078eeaf4bf8fcdc14a4e79b516f92b6ab520fc#diff-05f854a9033643de07f0d0059bc5b98f3b314eeb1e2499ea1057e925e6501ae8L381

Alternatively, you can download a newer version of TCPdump, and check that both bugs have been fixed.

Exercise 4 - LibTIFF

This time we will fuzz **LibTIFF** image library. The goal is to find a crash/PoC for **CVE-2016-9297** in libtiff 4.0.4 and **to measure the code coverage data** of your crash/PoC.

For more information about CVE-2016-9297 vulnerability, click me!

CVE-2016-9297 is an Out-of-bounds Read vulnerability that can be triggered via crafted TIFF_SETGET_C16ASCII or TIFF_SETGET_C32_ASCII tag values.

An Out-of-bounds Read is a vulnerability that occurs when the program reads data past the end, or before the beginning, of the intended buffer.

As a result, it allows remote attackers to cause a denial of service or possibly obtain potentially sensitive information from process memory.

You can find more information about Out-of-bounds Read vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/125.html

What you will learn

Once you complete this exercise you will know:

- How to measure code coverage using LCOV
- How to use code coverage data to improve the effectiveness of fuzzing

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Do it yourself!

In order to complete this exercise, you need to:

1. Fuzz LibTiff (with ASan enabled) until you have a few unique crashes
2. Triage the crashes to find a PoC for the vulnerability
3. Measure the code coverage of this PoC
4. Fix the issue

Estimated time = 3 hours

SPOILER ALERT! : Solution inside

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir fuzzing_tiff && cd fuzzing_tiff/
```

Download and uncompress libtiff 4.0.4:

```
wget https://download.osgeo.org/libtiff/tiff-4.0.4.tar.gz
tar -xzvf tiff-4.0.4.tar.gz
```

Now, we can build and install libtiff:

```
cd tiff-4.0.4/
./configure --prefix="$HOME/fuzzing_tiff/install/" --disable-shared
make
make install
```

As target binary we can just fuzz the `tiffinfo` binary located in the `/bin` folder. As seed input corpus, we're gonna use the sample images from the `/test/images/` folder.

To test everything is working properly, just type:

```
$HOME/fuzzing_tiff/install/bin/tiffinfo -D -j -c -r -s -w
$HOME/fuzzing_tiff/tiff-4.0.4/test/images/palette-1c-1b.tiff
```

and you should see something like that

```
antonio@ubuntu:~/fuzzing_tiff/tiff-4.0.4$ $HOME/fuzzing_tiff/install/bin/tiffinfo -D -j -c -r -s -w $HOME/fuzzing_tiff/tiff
-4.0.4/test/images/palette-1c-1b.tiff
TIFF Directory at offset 0xbd4 (3028)
  Image Width: 157 Image Length: 151
  Bits/Sample: 1
  Sample Format: unsigned integer
  Compression Scheme: None
  Photometric Interpretation: palette color (RGB from colormap)
  Samples/Pixel: 1
  Rows/Strip: 409
  Planar Configuration: single image plane
  Page Number: 0-1
  Color Map:
     0:     0     0     0
     1: 65535 65535 65535
  DocumentName: palette-1c-1b.tiff
  Software: GraphicsMagick 1.2 unreleased Q16 http://www.GraphicsMagick.org/
  1 Strips:
     0: [       8,     3020]
```

In the last command line you can see that I enabled all these flags: "-j -c -r -s -w". This is to improve the **code coverage** and increase the chances of finding the bug.

But how can we measure the code coverage of a given input case?

Code coverage

Code coverage is a software metric that show the number of times each line of code is triggered. By using code coverage we will get to know which parts of the code have been reached by the fuzzer and visualizes the fuzzing process.

First of all, we need to install lcov. We can do it with the following command:

```
sudo apt install lcov
```

Now we need to rebuild libTIFF with the `--coverage` flag (compiler and linker):

```
rm -r $HOME/fuzzing_tiff/install
cd $HOME/fuzzing_tiff/tiff-4.0.4/
make clean

CFLAGS="--coverage" LDFLAGS="--coverage" ./configure --
prefix="$HOME/fuzzing_tiff/install/" --disable-shared
make
make install
```

Then we can collect code coverage data by typing the following:

```
cd $HOME/fuzzing_tiff/tiff-4.0.4/
lcov --zerocounters --directory ./
lcov --capture --initial --directory ./ --output-file app.info
$HOME/fuzzing_tiff/install/bin/tiffinfo -D -j -c -r -s -w
```

```
$HOME/fuzzing_tiff/tiff-4.0.4/test/images/palette-1c-1b.tiff
lcov --no-checksum --directory ./ --capture --output-file app2.info
```

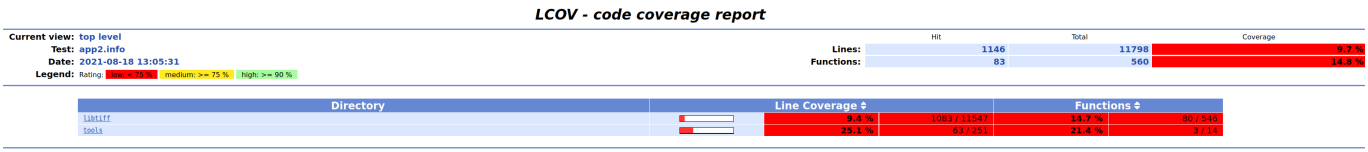
I will try to explain each of the commands:

- `lcov --zerocounters --directory ./` : Reset previous counters
- `lcov --capture --initial --directory ./ --output-file app.info` : Return the "baseline" coverage data file that contains zero coverage for every instrumented line
- `$HOME/fuzzing_tiff/install/bin/tiffinfo -D -j -c -r -s -w $HOME/fuzzing_tiff/tiff-4.0.4/test/images/palette-1c-1b.tiff` : Run the application you want to analyze . You can run it multiple times with different inputs
- `lcov --no-checksum --directory ./ --capture --output-file app2.info`: Save the current coverage state into the app2.info file

Finally, we have to generate the HTML output:

```
genhtml --highlight --legend -output-directory ./html-coverage/
./app2.info
```

If all went well, the code coverage report was created in the `html-coverage` folder. Just open the `./html-coverage/index.html` file and you should see something like this:



Now you can browse trough the different folders and files, and see the number of times each line was executed.

Fuzzing time

Now we're going to compile libtiff with ASAN enabled.

First of all, we're going to clean all previously compiled object files and executables:

```
rm -r $HOME/fuzzing_tiff/install
cd $HOME/fuzzing_tiff/tiff-4.0.4/
make clean
```

Now, we set AFL_USE_ASAN=1 before calling make:

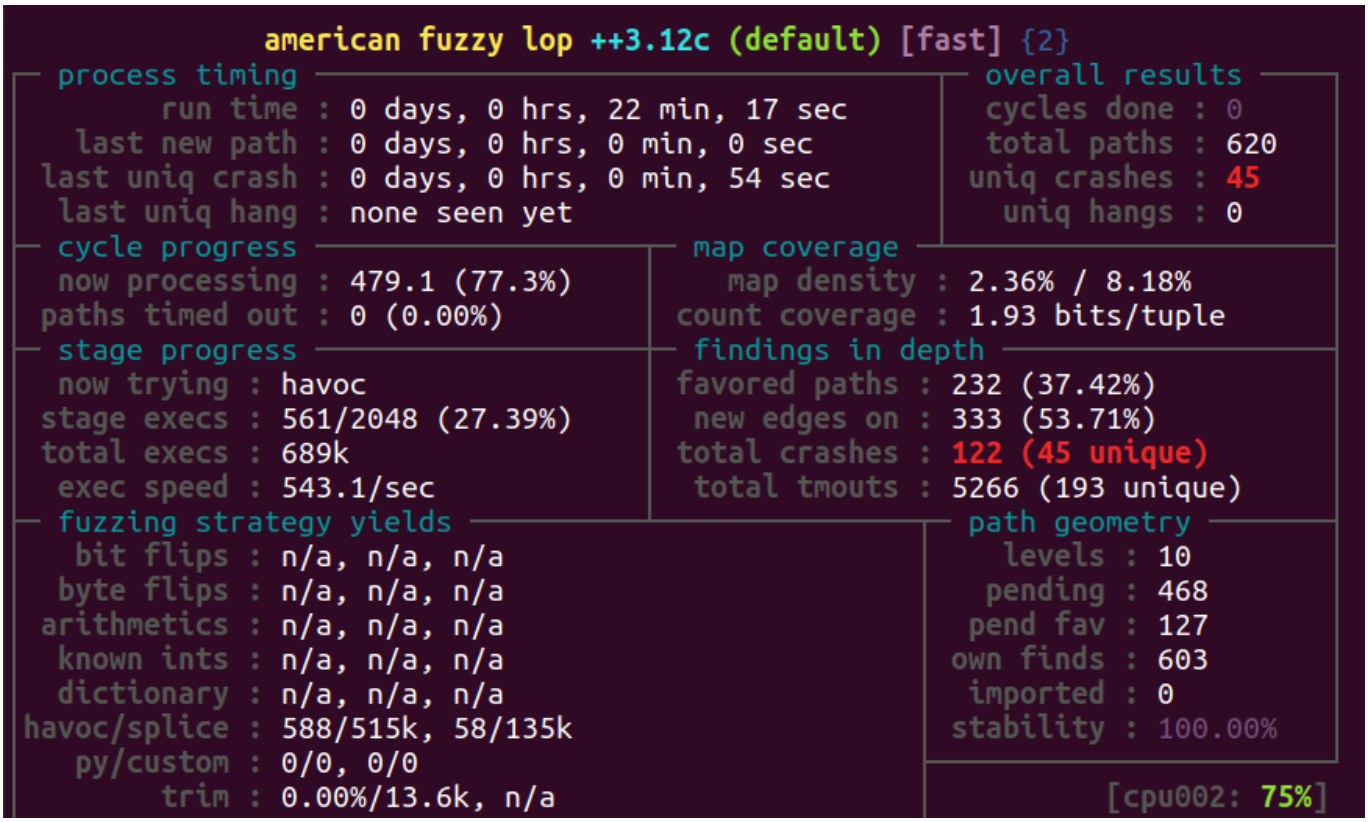
```
export LLVM_CONFIG="llvm-config-11"
CC=afl-clang-lto ./configure --prefix="$HOME/fuzzing_tiff/install/" --
disable-shared
```

```
AFL_USE_ASAN=1 make -j4
AFL_USE_ASAN=1 make install
```

Now, you can run the fuzzer with the following command:

```
afl-fuzz -m none -i $HOME/fuzzing_tiff/tiff-4.0.4/test/images/ -o
$HOME/fuzzing_tiff/out/ -s 123 -- $HOME/fuzzing_tiff/install/bin/tiffinfo
-D -j -c -r -s -w @@
```

After a few minutes you should see somethink like this:



Triage

The ASan trace may looks like:

Code coverage measure

Now, try to measure the code coverage of your PoC. In order to complete this part, **you need to obtain a coverage html report**, similar to the example above.

Fix the issue

The last step of the exercise is to fix the bug! Rebuild your target after the fix and check that your PoC don't crash the program anymore. This last part is left as exercise for the student.

Solution inside

Official fix:

- https://github.com/vadz/libtiff/commit/30c9234c7fd0dd5e8b1e83ad44370c875a0270ed

Alternatively, you can download a newer version of LibTIFF, and check that both bugs have been fixed.

Exercise 5 - LibXML2

For this exercize we will fuzz **LibXML2** XML parsing library. The goal is to find a crash/PoC for **CVE-2017-9048** in LibXML2 2.9.4.

CVE-2017-9048 is an stack buffer overflow vulnerability affecting the DTD validation functionality of LibXML2.

A stack buffer overflow is a type of buffer overflow where the buffer being overwritten is allocated on the stack.

As a result, a remote attacker can exploit this issue to execute arbitrary code within the context of an application using the affected library.

You can find more information about stack buffer oveflow vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/121.html

What you will learn

Once you complete this exercise you will know how:

- To use custom dictionaries for helping the fuzzer to find new execution paths
- To parallelize the fuzzing job accross multiple cores

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Dictionaries

When we want to fuzz complex text-based file formats (such as XML), it's useful to provide the fuzzer with a dictionary containing a list of basic syntax tokens.

In the case of AFL, such a dictionary is simply a set of words or values which is used by AFL to apply changes to the current in-memory file. Specifically, AFL performs the following changes with the values provided in the dictionary:

- Override: Replaces a specific position with n number of bytes, where n is the length of a dictionary entry.
- Insert: Inserts the dictionary entry at the current file position, forcing all characters to move n positions down and increasing file size.

You can find a good bunch of examples here

Paralellization

If you have a multi-core system is a good idea to parallelize your fuzzing job to make the most of your CPU resources.

Independent instances

This is the simplest parallelization strategy. In this mode, we run fully separate instances of afl-fuzz.

It is important to remember that AFL uses a non-deterministic testing algorithm. So, if we run multiples AFL instances, We increase our chances of success.

For this, you only need to run multiple instances of "afl-fuzz" on multiple terminal windows, setting a different "output folder" for each one of them. A simple approach is to run as many fuzzing jobs as cores are on your system.

Note: If you're using the -s flag, you need to use a different seed for each instance

Shared instances

The use of shared instances is a better approach to parallel fuzzing. In this case, each fuzzer instance gathers any test cases found by other fuzzers.

You will usually have only one master instance at a time:

```
./afl-fuzz -i afl_in -o afl_out -M Master -- ./program @@
```

and N-1 number of slaves:

```
./afl-fuzz -i afl_in -o afl_out -S Slave1 -- ./program @@
./afl-fuzz -i afl_in -o afl_out -S Slave2 -- ./program @@
...
./afl-fuzz -i afl_in -o afl_out -S SlaveN -- ./program @@
```

Do it yourself!

In order to complete this exercise, you need to:

1. Find an interface application that makes use of the LibXML2 library
2. Copy the SampleInput.xml file to your AFL input folder
3. Create a custom dictionary for fuzzing XML
4. Fuzz LibXML2 until you have a few unique crashes. I recommend you to use as many AFL instances as posible (CPU cores)
5. Triage the crashes to find a PoC for the vulnerability
6. Fix the issues

Estimated time = 3 hours

SPOILER ALERT! : Solution inside

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir Fuzzing_libxml2 && cd Fuzzing_libxml2
```

Download and uncompress libxml2-2.9.4.tar.gz

```
wget http://xmlsoft.org/download/libxml2-2.9.4.tar.gz
tar xvf libxml2-2.9.4.tar.gz && cd libxml2-2.9.4/
```

Build and install libxml2:

```
sudo apt-get install python-dev
CC=afl-clang-lto CXX=afl-clang-lto++ CFLAGS="-fsanitize=address"
CXXFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address" ./configure --
prefix="$HOME/Fuzzing_libxml2/libxml2-2.9.4/install" --disable-shared --
without-debug --without-ftp --without-http --without-legacy --without-
python LIBS='-ldl'
make -j$(nproc)
make install
```

Now, we can test that all is working OK with:

```
./xmllint --memory ./test/wml.xml
```

and you should see something like that

```
antonio@dragon:~/Fuzzing_libxml2/libxml2-2.9.4$ ./xmllint --memory './test/wml.xml'
Warning: AFL++ tools might need to set AFL_MAP_SIZE to 146056 to be able to run this instrumented program if this crashes!
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card id="card1" title="Rubriques 75008">
        <p>
                <a href="rubmenu.asp?CP=75008&amp;RB=01">Cin&#xE9;ma</a><br/>
        </p>

</card>
</wml>
```

Seed corpus creation

First of all, we need to get some XML samples. We're gonna use the **SampleInput.xml** provided in this repository:

```
mkdir afl_in && cd afl_in
wget https://raw.githubusercontent.com/antonio-
morales/Fuzzing101/main/Exercise%205/SampleInput.xml
cd ..
```

Custom dictionary

Now, you need to create an XML dictionary. Alternatively, you can use the XML dictionary provided with AFL++:

```
mkdir dictionaries && cd dictionaries
wget
https://raw.githubusercontent.com/AFLplusplus/AFLplusplus/stable/dictionar
ies/xml.dict
cd ..
```

Fuzzing time

In order to catch the bug, is mandatory to enable the `--valid` parameter. I also set the dictionary path with the **-x flag** and enabled the deterministic mutations with the **-D flag** (only for the master fuzzer):

For example, I ran the fuzzer with the following command

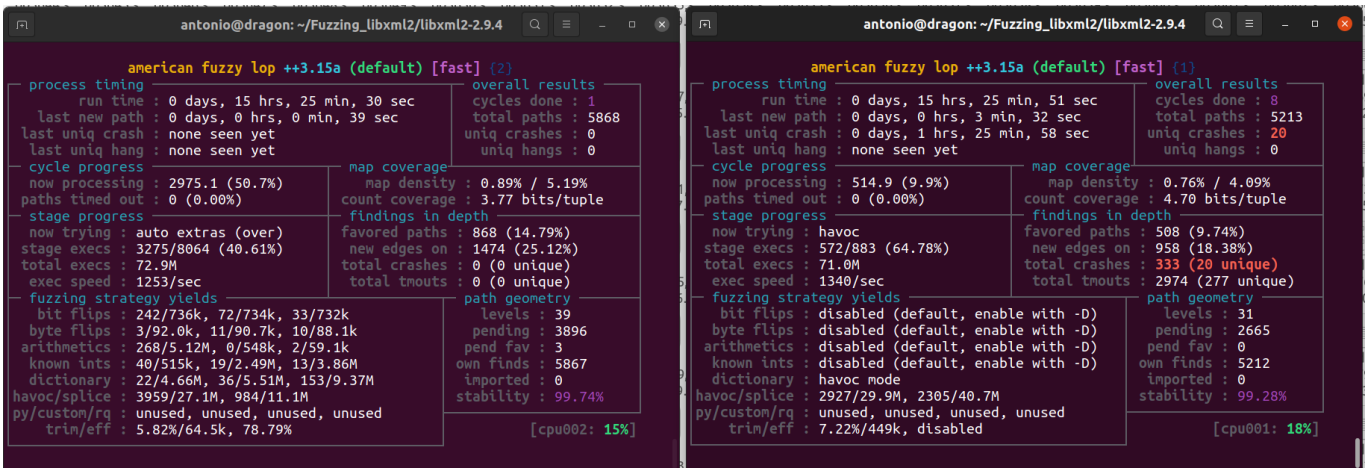
```
afl-fuzz -m none -i ./afl_in -o afl_out -s 123 -x ./dictionaries/xml.dict
-D -M master -- ./xmllint --memory --noenc --nocdata --dtdattr --loaddtd -
-valid --xinclude @@
```

You can run another slave instance with:

```
afl-fuzz -m none -i ./afl_in -o afl_out -s 234 -S slave1 -- ./xmllint --
memory --noenc --nocdata --dtdattr --loaddtd --valid --xinclude @@
```

Are you interested in fuzzing command-line arguments? Take a look to the following blog post, to the "Fuzzing command-line arguments" section.

After a while, you should have multiple crashes:



Triage

To debug a program built with ASan is so much easier than in the previous exercises. All you need to do is to feed the program with the crash file:

```
./xmllint --memory --noenc --nocdata --dtdattr --loaddtd --valid --
xinclude
'./afl_out/default/crashes/id:000000,sig:06,src:003963,time:12456489,op:ha
voc,rep:4'
```

and you will get a nice summary of the crash, including the execution trace:

```
antonio@dragon:~/Fuzzing_libxml2/libxml2-2.9.4$ ./xmllint --memory --noenc --nocdata --dtdattr --loaddtd --valid --xinclude '/ho
me/antonio/Fuzzing_libxml2/libxml2-2.9.4/afl_out_2/default/crashes/id:000000,sig:06,src:003963,time:12456489,op:havoc,rep:4'
Warning: AFL++ tools might need to set AFL_MAP_SIZE to 146056 to be able to run this instrumented program if this crashes!
================================================================
==1426439==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff995579c8 at pc 0x00000035ec79 bp 0x7fff99556570 sp 0x
7fff99555d10
WRITE of size 1100 at 0x7fff995579c8 thread T0
    #0 0x35ec78 in strcat (/home/antonio/Fuzzing_libxml2/libxml2-2.9.4/xmllint+0x35ec78)
    #1 0x5ace57 in xmlSnprintfElementContent /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/valid.c:1279:3
    #2 0x5e4b69 in xmlValidateElementContent /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/valid.c:5445:6
    #3 0x5e4b69 in xmlValidateOneElement /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/valid.c:6152:12
    #4 0xa94baf in xmlSAX2EndElementNs /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/SAX2.c:2467:24
    #5 0x4d0731 in xmlParseElement /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/parser.c:10203:3
    #6 0x4fee53 in xmlParseDocument /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/parser.c:10953:2
    #7 0x531c20 in xmlDoRead /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/parser.c:15432:5
    #8 0x3c99ba in xmlReadMemory /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/parser.c:15518:13
    #9 0x3c99ba in parseAndPrintFile /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/xmllint.c:2371:9
    #10 0x3b61b6 in main /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/xmllint.c:3767:7
    #11 0x7f49a7016564 in __libc_start_main csu/../csu/libc-start.c:332:16
    #12 0x2f755d in _start (/home/antonio/Fuzzing_libxml2/libxml2-2.9.4/xmllint+0x2f755d)

Address 0x7fff995579c8 is located in stack of thread T0 at offset 5128 in frame
    #0 0x5ddf0f in xmlValidateOneElement /home/antonio/Fuzzing_libxml2/libxml2-2.9.4/valid.c:5943
```

Fix the issue

The last step of the exercise is to fix the bug! Rebuild your target after the fix and check that your PoC don't crash the program anymore. This last part is left as exercise for the student.

Solution inside

Official fix:

- https://github.com/GNOME/libxml2/commit/932cc9896ab41475d4aa429c27d9afd175959d74

Alternatively, you can download a newer version of LibXML, and check that the bug has been fixed.

Exercise 6 - GIMP

For this exercise, we will fuzz **GIMP** image editor. The goal is to find a crash/PoC for **CVE-2016-4994** in GIMP 2.8.16.

CVE-2016-4994 is an Use-After-Free vulnerability that can be triggered via a crafted XCF file.

Use after free errors occur when a program continues to use a pointer after it has been freed.

This can have any number of adverse consequences, ranging from the corruption of valid data to the execution of arbitrary code.

You can find more information about Use-After-Free vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/416.html

What you will learn

Once you complete this exercise you will know:

- How to use persistent mode to speed up fuzzing
- How to fuzz interactive / GUI applications

Read Before Start

- I suggest you to try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Persistent mode

The AFL persistent mode is based on **in-process fuzzers**: fuzzers that make use of a single one process, injecting code into the target process and changing the input values in memory.

afl-fuzz supports a working mode that combines the benefits of in-process fuzzing with the robustness of a more traditional multi-process tool: **the Persistent Mode**.

In persistent mode, AFL++ fuzzes a target multiple times in a single forked process, instead of forking a new process for each fuzz execution. This mode can improve the fuzzing speed up to 20X times.

The basic structure of the target would be as follows:

```
    //Program initialization

    while (__AFL_LOOP(10000)) {

      /* Read input data. */
      /* Call library code to be fuzzed. */
      /* Reset state. */
    }

    //End of fuzzing
```

You can find more information about AFL++ persistent mode at:
https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md

Do it yourself!

In order to complete this exercise, you need to:

1. Find an efficient way of modifying GIMP source code to enable AFL persistent mode
2. Create a seed corpus of XCF samples
3. Optional: Create a fuzzing dictionary for the XCF file format
4. Fuzz GIMP until you have a few unique crashes. I recommend you to use as many AFL instances as possible (CPU cores)
5. Triage the crashes to find a PoC for the vulnerability
6. Fix the issues
7. Bonus: Find some of the other minor bugs

Estimated time = 7 hours

SPOILER ALERT! : Solution inside

Download and build your target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir Fuzzing_gimp && cd Fuzzing_gimp
```

Now, install the required dependencies:

```
sudo apt-get install build-essential libatk1.0-dev libfontconfig1-dev
libcairo2-dev libgudev-1.0-0 libdbus-1-dev libdbus-glib-1-dev libexif-dev
libxfixes-dev libgtk2.0-dev python2.7-dev libpango1.0-dev libglib2.0-dev
zlib1g-dev intltool libbabl-dev
```

We also need **GEGL 0.2(Generic Graphics Library)**. Unfortunately, we cannot find the gegl-0.2.0 package in our Ubuntu distribution. So, we need to download and build this library from source code. Just type:

```
wget https://download.gimp.org/pub/gegl/0.2/gegl-0.2.0.tar.bz2
tar xvf gegl-0.2.0.tar.bz2 && cd gegl-0.2.0
```

Now, we need to make two minor changes in the source code:

```
sed -i 's/CODEC_CAP_TRUNCATED/AV_CODEC_CAP_TRUNCATED/g'
./operations/external/ff-load.c
sed -i 's/CODEC_FLAG_TRUNCATED/AV_CODEC_FLAG_TRUNCATED/g'
./operations/external/ff-load.c
```

Build and install Gegl-0.2:

```
./configure --enable-debug --disable-glibtest  --without-vala --without-
cairo --without-pango --without-pangocairo --without-gdk-pixbuf --without-
lensfun --without-libjpeg --without-libpng --without-librsvg --without-
openexr --without-sdl --without-libopenraw --without-jasper --without-
graphviz --without-lua --without-libavformat --without-libv4l --without-
libspiro --without-exiv2 --without-umfpack
make -j$(nproc)
sudo make install
```

Don't worry if you see some error messages in the testing stage.

Now, we download and uncompress GIMP 2.8.16:

```
cd ..
wget https://mirror.klaus-uwe.me/gimp/pub/gimp/v2.8/gimp-2.8.16.tar.bz2
```

```
tar xvf gimp-2.8.16.tar.bz2 && cd gimp-2.8.16/
```

Time for building GIMP using **afl-clang-lto** as the compiler (it can take some time):

```
CC=afl-clang-lto CXX=afl-clang-lto++
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$HOME/Fuzzing_gimp/gegl-0.2.0/ CFLAGS="-
fsanitize=address" CXXFLAGS="-fsanitize=address" LDFLAGS="-
fsanitize=address" ./configure --disable-gtktest --disable-glibtest --
disable-alsatest --disable-nls --without-libtiff --without-libjpeg --
without-bzip2 --without-gs --without-libpng --without-libmng --without-
libexif --without-aa --without-libxpm --without-webkit --without-librsvg -
-without-print --without-poppler --without-cairo-pdf --without-gvfs --
without-libcurl --without-wmf --without-libjasper --without-alsa --
without-gudev --disable-python --enable-gimp-console --without-mac-twain -
-without-script-fu --without-gudev --without-dbus --disable-mp --without-
linux-input --without-xvfb-run --with-gif-compression=none --without-xmc -
-with-shm=none --enable-debug  --prefix="$HOME/Fuzzing_gimp/gimp-
2.8.16/install"
make -j$(nproc)
make install
```

Persistent mode

There are two very simple approaches:

- The first one is to modify the `app.c` file and include the AFL_LOOP macro into the for loop:

```
244  /*  Load the images given on the command-line.
245   */
246  if (filenames)
247    {
248      gint i;
249
250      for (i = 0; filenames[i] != NULL; i++)
251        {
252          if (run_loop){
253
254              #ifdef __AFL_COMPILER
255                while(__AFL_LOOP(1000)){
256                    file_open_from_command_line (gimp, filenames[i], as_new);
257                }
258
259                exit(0);
260
261              #else
262                file_open_from_command_line (gimp, filenames[i], as_new);
263
264              #endif
265            }
```

- The second one is to insert the AFL_LOOP macro inside the `xcf_load_invoker` function:

```
261 static GValueArray *
262 xcf_load_invoker (GimpProcedure      *procedure,
263                   Gimp               *gimp,
264                   GimpContext        *context,
265                   GimpProgress       *progress,
266                   const GValueArray  *args,
267                   GError            **error)
268 {
269   XcfInfo      info;
270   GValueArray *return_vals;
271   GimpImage   *image    = NULL;
272   const gchar *filename;
273   gboolean     success = FALSE;
274   gchar        id[14];
275
276   gimp_set_busy (gimp);
277
278   filename = g_value_get_string (&args->values[1]);
279
280 #ifdef __AFL_COMPILER
281   while(__AFL_LOOP(10000)){
282 #endif
283
284   info.fp = g_fopen (filename, "rb");
285
286   if (info.fp)
```

While the first one allows us to target different input formats, the second is faster and we will have more chances to catch the bug.

You can download the patch for the second one here

Seed corpus creation

I recommend you to create multiple GIMP projects and save them to obtain multiple .xcf samples

Alternatively, you can just copy the SampleInput.xcf file to your AFL input folder

Fuzzing time

Since the vulnerability affects GIMP core, we can save some startup time by removing plugins that we don't need:

```
rm ./install/lib/gimp/2.0/plug-ins/*
```

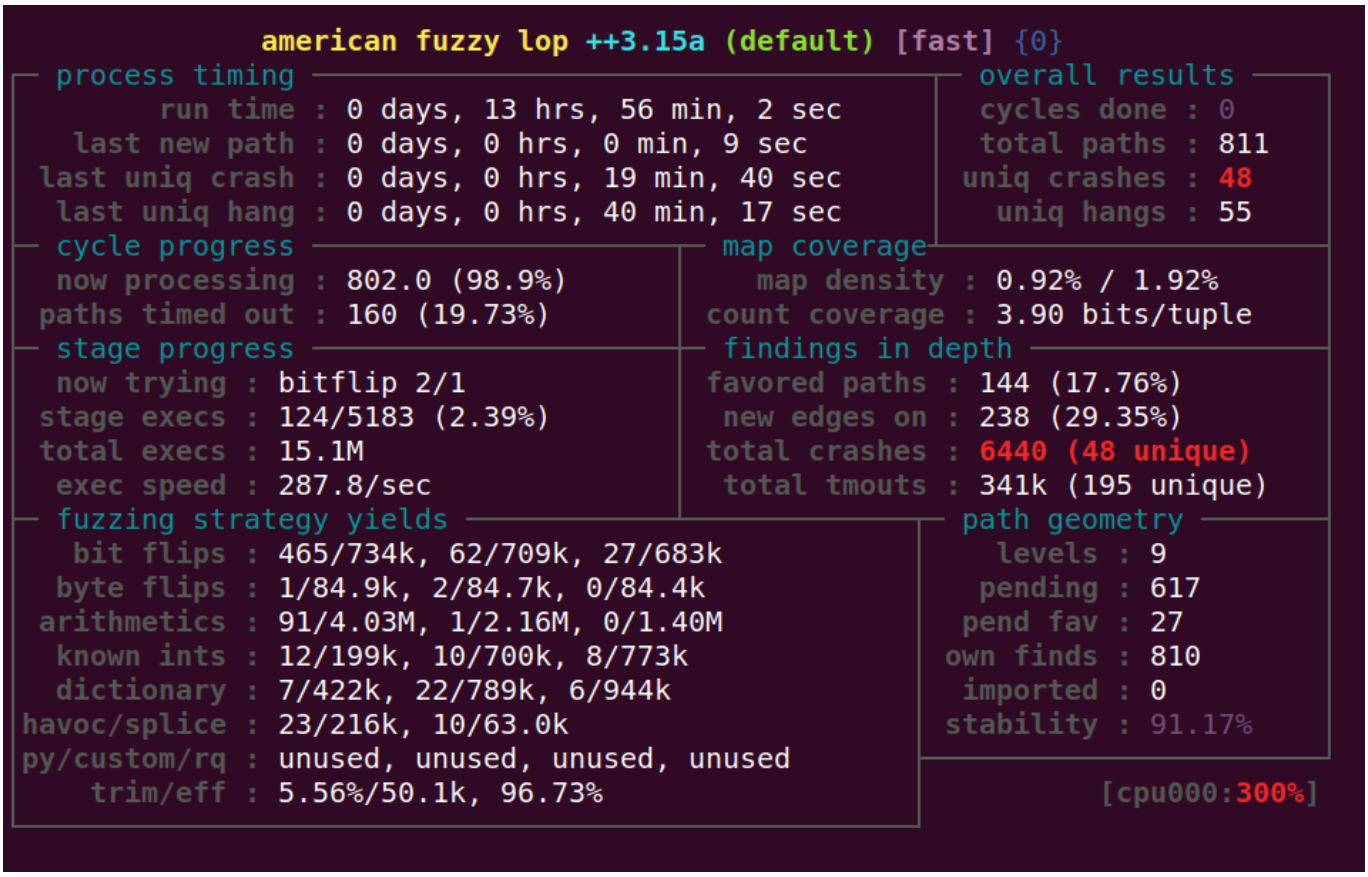
Now, you can run the fuzzer with the following command:

```
ASAN_OPTIONS=detect_leaks=0,abort_on_error=1,symbolize=0 afl-fuzz -i
'./afl_in' -o './afl_out' -D -t 100 -- ./install/bin/gimp-console-2.8 --
verbose -d -f @@
```

Some notes:

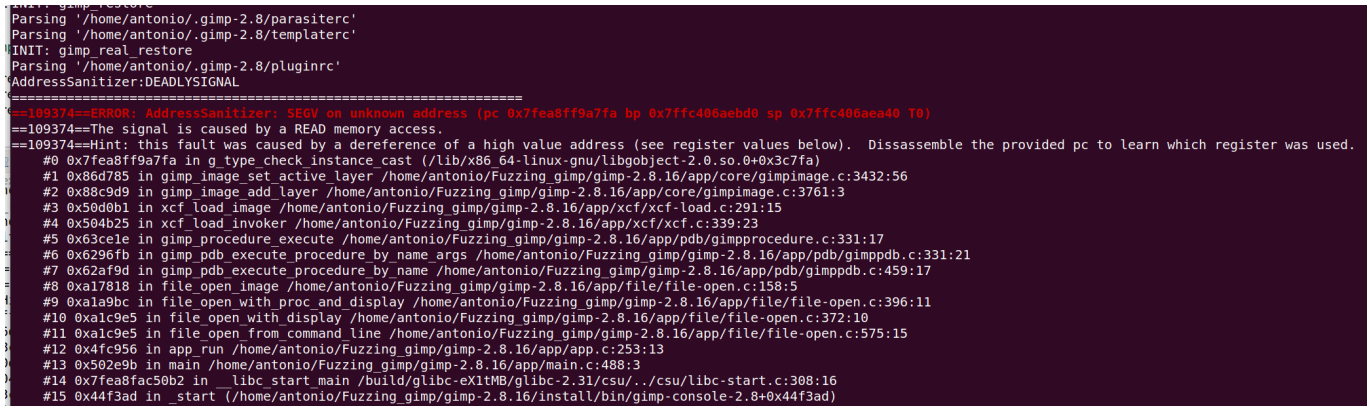
- `gimp-console-2.8` is a console-only version of GIMP
- I recommend enabling deterministic mutations (-D)
- There is also an infinite loop bug in the code, so we need to set a low timeout limit (ex. -t 100). This timeout limit depends on your machine capabilites, so you will need to adjust it for your particular case.

After a while, you should have multiple crashes:



Triage

The ASan trace may look like:



Fix the issues

The last step of the exercise is to fix both bugs. Rebuild your target after the fixes and check that your PoCs don't crash the program anymore. This last part is left as an exercise for the student.

Solution inside

Official fixes:

- https://gitlab.gnome.org/GNOME/gimp/-/commit/6d804bf9ae77bc86a0a97f9b944a129844df9395

Alternatively, you can download a newer version of GIMP, and check that both bugs have been fixed.

Bonus

There are other minor bugs in the code. I've found:

- A null dereference bug
- An infinite loop bug
- A memory exhaustion bug

I encourage you to try to find all of them!

Exercise 7 - VLC Media Player

For this exercise, we will fuzz **VLC** media player. The goal is to find a crash/PoC for **CVE-2019-14776** in VLC 3.0.7.1.

CVE-2019-14776 is an Out-of-bounds Read vulnerability that can be triggered via a crafted WMV/ASF (Windows Media Video) file.

An Out-of-bounds Read is a vulnerability that occurs when the program reads data past the end, or before the beginning, of the intended buffer.

As a result, it allows remote attackers to cause a denial of service or possibly obtain potentially sensitive information from process memory.

You can find more information about Out-of-bounds Read vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/125.html

What you will learn

Once you complete this exercise you will know:

- How to use Partial Instrumentation to only instrument the relevant parts of the program
- How to write fuzzing harnesses to test large applications more efficiently

Read Before Start

- I suggest you try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Partial Instrumentation

One of the advantages of using an evolutionary coverage-guided fuzzer is that it's capable of finding new execution paths all by itself. However, this can often also be a disadvantage. This is particularly true when we face software with a highly modular architecture (as in the case of VLC media player), where each module performs a specific task.

So, let's assume we fed the fuzzer with a valid MKV file. But after several mutations of the input file, the file "magic bytes" have changed and now the input file is viewed as an AVI file by our program. Therefore, this "mutated MKV file" is now processed by AVI Demux. After some time, the file magic bytes change once again and now the file is viewed as an MPEG file. In both cases, the potential of this newly mutated file to increase our code coverage is very poor because this new file won't have any valid syntax structure.

In short, if we don't put constraints on code coverage, the fuzzer can easily choose a wrong path which in turn makes the fuzzing process less effective.

To address this problem, AFL++ includes a Partial Instrumentation feature that allows you to specify which functions/files should be compiled with or without instrumentation. This helps the fuzzer focus on the important parts of the program, avoiding undesired noise and disturbance by exercising uninteresting code paths.

To use it, we set the environment AFL_LLVM_ALLOWLIST variable when compiling. This environment variable must point to a file containing all the functions/filenames that should be instrumented.

You can find more information about AFL++ partial instrumentation at:
https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.instrument_list.md

Do it yourself!

In order to complete this exercise, you need to:

1. Write a partial instrumentation file containing the ASF demuxing functions and filenames
2. Create a fuzzing harness for fuzzing ASF file format (persistent mode)
3. Create a seed corpus of ASF samples
4. Optional: Create a fuzzing dictionary for the ASF file format
5. Fuzz VLC until you have a few unique crashes. I recommend you to use as many AFL instances as possible (CPU cores)
6. Triage the crashes to find a PoC for the vulnerability
7. Fix the issues

Estimated time = 6 hours

SPOILER ALERT! : Solution inside

Download the target

Let's first get our fuzzing target. Create a new directory for the project you want to fuzz:

```
cd $HOME
mkdir fuzzing_vlc && cd fuzzing_vlc
```

Download and uncompress vlc-3.0.7.1.tar.xz:

```
wget https://download.videolan.org/pub/videolan/vlc/3.0.7.1/vlc-
3.0.7.1.tar.xz
tar -xvf vlc-3.0.7.1.tar.xz && cd vlc-3.0.7.1/
```

Build VLC:

```
./configure --prefix="$HOME/fuzzing_vlc/vlc-3.0.7.1/install" --disable-a52
--disable-lua --disable-qt
make -j$(nproc)
```

To test everything is working properly, just type:

```
./bin/vlc-static --help
```

and you should see something like that

```
antonio@ubuntu:~/fuzzing_vlc/vlc-3.0.7.1$ ./bin/vlc-static --help
VLC media player 3.0.7.1 Vetinari (revision 3.0.7.1-0-gf3940db4af)
Usage: vlc [options] [stream] ...
You can specify multiple streams on the commandline.
They will be enqueued in the playlist.
The first item specified will be played first.

Options-styles:
  --option  A global option that is set for the duration of the program.
   -option  A single letter version of a global --option.
   :option  An option that only applies to the stream directly before it
            and that overrides previous settings.

Stream MRL syntax:
  [[access][/demux]://]URL[#[title][:chapter][-[title][:chapter]]]
  [:option=value ...]

  Many of the global --options can also be used as MRL specific :options.
  Multiple :option=value pairs can be specified.

URL syntax:
  file:///path/file             Plain media file
  http://host[:port]/file       HTTP URL
  ftp://host[:port]/file        FTP URL
  mms://host[:port]/file        MMS URL
  screen://                     Screen capture
  dvd://[device]                DVD device
  vcd://[device]                VCD device
  cdda://[device]               Audio CD device
  udp://[[<source address>]@[<bind address>][:<bind port>]]
                                UDP stream sent by a streaming server
  vlc://pause:<seconds>         Pause the playlist for a certain time
  vlc://quit                    Special item to quit VLC
```

Seed corpus creation

You can find a lot of video samples in the ffmpeg samples repository

I advise you to pick some samples and then, use a video editor to shrink the video file to the smallest size possible.

These are some examples of open-source video editors:

- OpenShot
- Shotcut

Or more easily, just copy the short2.wmv and veryshort.wmv files to your AFL input folder.

Fuzzing harness

If you try to fuzz directly the `vlc-static` binary you'll see that AFL only gets a few executions per second. This is because VLC startup is very time-consuming. That's why I recommend you create a **custom fuzzing harness** for fuzzing VLC.

I chose to modify the `./test/vlc-demux-run.c` file to include my fuzzing harness. In this way, you can compile the harness just by doing:

```
cd test
make vlc-demux-run -j$(nproc) LDFLAGS="-fsanitize=address"
cd ..
```

Since the bug exists in the ASF demuxing, I call to the `vlc_demux_process_memory` function. This function try to demux a data buffer previously stored in the memory. You can find my code changes here

Partial instrumentation

At first, I tried just including the filenames involved in ASF demuxing. Unfortunately, this approach didn't work.

It seems that matching on filenames is not always possible, so I opted for a mixing approach including function matching and filename matching:

```
demux/asf/asf.c
demux/asf/asfpacket.c
demux/asf/libasf.c
vlc.c

#fun: Demux
fun: WaitKeyframe
fun: SeekPercent
fun: SeekIndex
fun: SeekPrepare
#fun: Control
fun: Packet_SetAR
fun: Packet_SetSendTime
fun: Packet_UpdateTime
fun: Packet_GetTrackInfo
fun: Packet_DoSkip
fun: Packet_Enqueue
fun: Block_Dequeue
fun: ASF_fillup_es_priorities_ex
fun: ASF_fillup_es_bitrate_priorities_ex
fun: DemuxInit
fun: FlushQueue
fun: FlushQueues
fun: DemuxEnd
```

You can download my partial instrumentation file here

Minor changes

To speed-up the ASF fuzzing speed, I recommend you to apply this patch to `modules/demux/libasf.c` (no more clues at the moment 😉) : speedup.patch

Fuzzing time

Time for building VLC using **afl-clang-fast** as the compiler and with ASAN enabled:

```
CC="afl-clang-fast" CXX="afl-clang-fast++" ./configure --
prefix="$HOME/fuzzing_vlc/vlc-3.0.7.1/install" --disable-a52 --disable-lua
--disable-qt --with-sanitizer=address
AFL_LLVM_ALLOWLIST=$HOME/fuzzing_vlc/vlc-3.0.7.1/Partial_instrumentation
make -j$(nproc) LDFLAGS="-fsanitize=address"
```

Build the fuzzing harness:

```
cd test
make vlc-demux-run -j$(nproc) LDFLAGS="-fsanitize=address"
cd ..
```

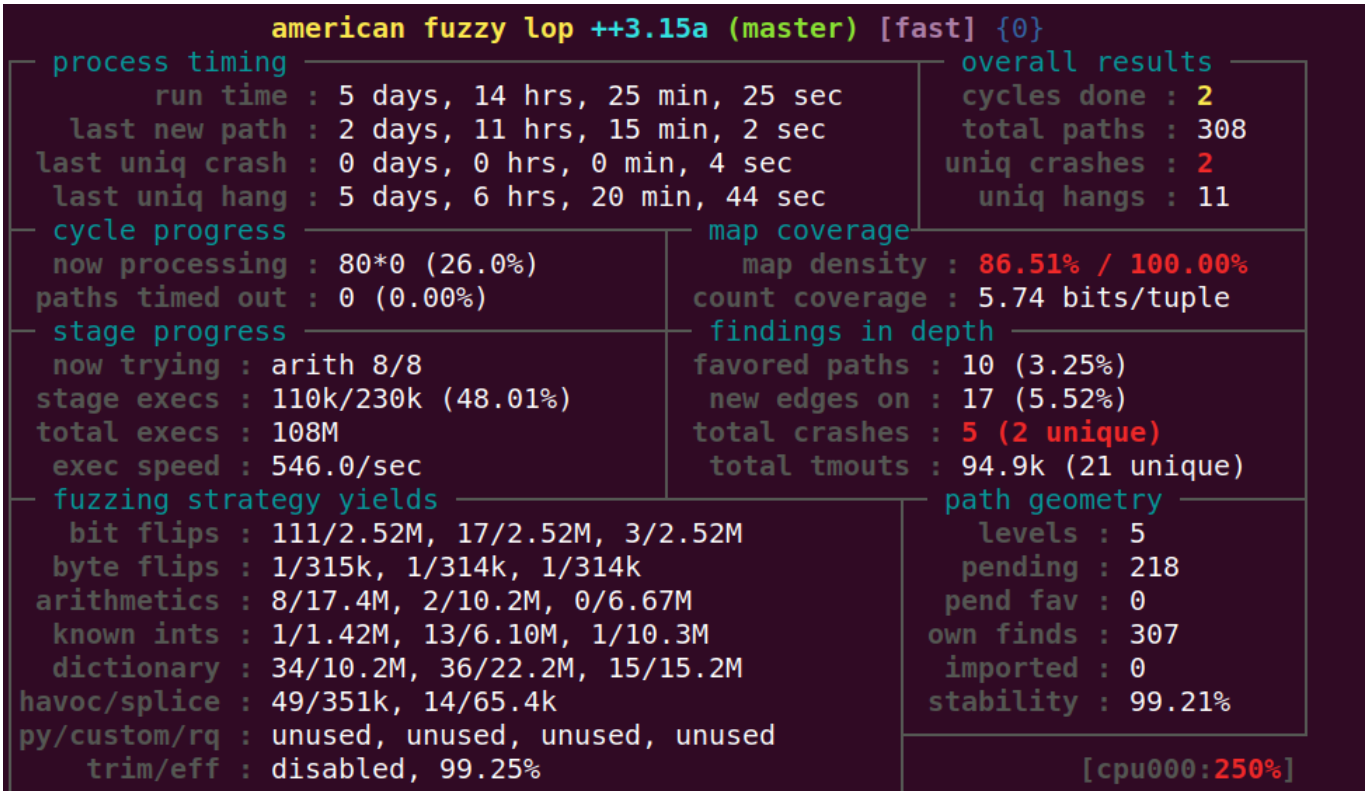
Now, you can run the fuzzer with the following command:

```
afl-fuzz -t 100 -m none -i './afl_in' -o './afl_out' -x
asf_dictionary.dict -D -M master -- ./test/vlc-demux-run @@
```

Some notes:

- The timeout parameter is heavily reliant on the computer. You will need to adjust this value.

After a while, you should have multiple crashes:



Triage

The ASan trace may look like:

```
==865747==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x611000011ba4 at pc 0x000000496967 bp 0x7ffc6ad0a130 sp 0x7ffc6ad098f8
READ of size 210 at 0x611000011ba4 thread T0
    #0 0x496966 in __asan_memcpy (/home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/vlc-demux-run+0x496966)
    #1 0x7fdca2c9b879 in memcpy /usr/include/x86_64-linux-gnu/bits/string_fortified.h:34:10
    #2 0x7fdca2c9b879 in DemuxInit /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/asf.c:1109:25
    #3 0x7fdca2c90c4c in Open /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/asf.c:169:9
    #4 0x7fdca642df7c in module_load /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/modules/modules.c:183:15
    #5 0x7fdca642df7c in vlc_module_load /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/modules/modules.c:279:23
    #6 0x7fdca6477a37 in demux_NewAdvanced /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/input/demux.c:264:29
    #7 0x7fdca6477367 in demux_New /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/input/demux.c:148:12
    #8 0x4c81b0 in demux_process_stream /home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/src/input/demux-run.c:272:22
    #9 0x4ca142 in vlc_demux_process_memory /home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/src/input/demux-run.c:357:15
    #10 0x4c7719 in main /home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/vlc-demux-run.c:106:3
    #11 0x7fdca5f8d0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16
    #12 0x41d49d in _start (/home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/vlc-demux-run+0x41d49d)

0x611000011ba4 is located 0 bytes to the right of 228-byte region [0x611000011ac0,0x611000011ba4)
allocated by thread T0 here:
    #0 0x49750d in malloc (/home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/vlc-demux-run+0x49750d)
    #1 0x7fdca2cb438d in ASF_ReadObject_stream_properties /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/libasf.c:587:13
    #2 0x7fdca2cad01f in ASF_ReadObject /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/libasf.c:1547:24
    #3 0x7fdca2cb0a3b in ASF_ReadObject_Header /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/libasf.c:213:26
    #4 0x7fdca2cad01f in ASF_ReadObject /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/libasf.c:1547:24
    #5 0x7fdca2ca7f67 in ASF_ReadObjectRoot /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/libasf.c:1716:23
    #6 0x7fdca2c90c4c in Open /home/antonio/fuzzing_vlc/vlc-3.0.7.1/modules/demux/asf/asf.c:169:9
    #7 0x7fdca642df7c in module_load /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/modules/modules.c:183:15
    #8 0x7fdca642df7c in vlc_module_load /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/modules/modules.c:279:23
    #9 0x7fdca6477a37 in demux_NewAdvanced /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/input/demux.c:264:29
    #10 0x7fdca6477367 in demux_New /home/antonio/fuzzing_vlc/vlc-3.0.7.1/src/input/demux.c:148:12
    #11 0x4ca142 in vlc_demux_process_memory /home/antonio/fuzzing_vlc/vlc-3.0.7.1/test/src/input/demux-run.c:357:15
```

Fix the issues

The last step of the exercise is to fix the bug. Rebuild your target after the fix and check that your PoC doesn't crash the program anymore. This last part is left as an exercise for the student.

Solution inside

Official fixes:

- https://github.com/videolan/vlc/commit/fdbdd677c1e6262f31771b0ba10afb24aabf108c#diff-a2217012504639027 4dd33c2cb5bb0e99d485e6708b376f40978de9534ac55a9

Alternatively, you can download a newer version of VLC, and check that both bugs have been fixed.

Exercise 8 - Adobe Reader

For this exercise, we will fuzz **Adobe Reader** application. The goal is to find an Out-of-bounds vulnerability in Adobe Reader 9.5.1.

What you will learn

Once you complete this exercise you will know:

- How to use AFL++'s QEMU mode to fuzz closed-source applications
- How to enable persistent mode in QEMU mode
- How to use QASAN, a binary-only sanitizer

Read Before Start

- I suggest you try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 20.04.2 LTS**. I highly recommend you to use **the same OS version** to avoid different fuzzing results and to run AFL++ **on bare-metal** hardware, and not virtualized machines, for best performance.

Otherwise, you can find an Ubuntu 20.04.2 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Do it yourself!

In order to complete this exercise, you need to:

1. Create a seed corpus of PDF samples
2. Enable persistent mode
3. Fuzz Adobe Reader using QEMU mode until you have some crashes

4. Triage the crashes to find a PoC for the vulnerability

Estimated time = 8 hours

SPOILER ALERT! : Solution inside

AFL++'s QEMU installation

First of all, we need afl-qemu installed in our system. You can check it with the following command:

```
afl-qemu-trace --help
```

and you should see something like that:

```
antonio@ubuntu:~/fuzzing_adobe$ afl-qemu-trace --help
usage: qemu-x86_64 [options] program [arguments...]
Linux CPU emulator (compiled for x86_64 emulation)

Options and associated environment variables:

Argument             Env-variable      Description
-h                                     print this help
-help
-g port              QEMU_GDB          wait gdb connection to 'port'
-L path              QEMU_LD_PREFIX    set the elf interpreter prefix to 'path'
-s size              QEMU_STACK_SIZE   set the stack size to 'size' bytes
-cpu model           QEMU_CPU          select CPU (-cpu help for list)
-E var=value         QEMU_SET_ENV      sets targets environment variable (see below)
-U var               QEMU_UNSET_ENV    unsets targets environment variable (see below)
-0 argv0             QEMU_ARGV0        forces target process argv[0] to be 'argv0'
-r uname             QEMU_UNAME        set qemu uname release string to 'uname'
-B address           QEMU_GUEST_BASE   set guest_base address to 'address'
-R size              QEMU_RESERVED_VA  reserve 'size' bytes for guest virtual address space
-d item[,...]        QEMU_LOG          enable logging of specified items (use '-d help' for a list of items)
-dfilter range[,...] QEMU_DFILTER      filter logging based on address range
-D logfile           QEMU_LOG_FILENAME write logs to 'logfile' (default stderr)
-p pagesize          QEMU_PAGESIZE     set the host page size to 'pagesize'
-singlestep          QEMU_SINGLESTEP   run in singlestep mode
-strace              QEMU_STRACE       log system calls
-seed                QEMU_RAND_SEED    Seed for pseudo-random number generator
-trace               QEMU_TRACE        [[enable=]<pattern>][,events=<file>][,file=<file>]
-version             QEMU_VERSION      display version information and exit

Defaults:
QEMU_LD_PREFIX  = /usr/gnemul/qemu-x86_64
QEMU_STACK_SIZE = 8388608 byte

You can use -E and -U options or the QEMU_SET_ENV and
QEMU_UNSET_ENV environment variables to set and unset
environment variables for the target process.
It is possible to provide several variables by separating them
by commas in getsubopt(3) style. Additionally it is possible to
provide the -E and -U options multiple times.
```

If it's not already installed, you can build and install it with:

```
sudo apt install ninja-build libc6-dev-i386
cd ~/Downloads/AFLplusplus/qemu_mode/
CPU_TARGET=i386 ./build_qemu_support.sh
make distrib
sudo make install
```

where `/Downloads/AFLplusplus/` is the AFL++ root folder

Adobe Reader installation

Install dependencies:

```
sudo apt-get install libxml2:i386
```

Download and uncompress `AdbeRdr9.5.1-1_i386linux_enu.deb`:

```
wget ftp://ftp.adobe.com/pub/adobe/reader/unix/9.x/9.5.1/enu/AdbeRdr9.5.1-
1_i386linux_enu.deb
```

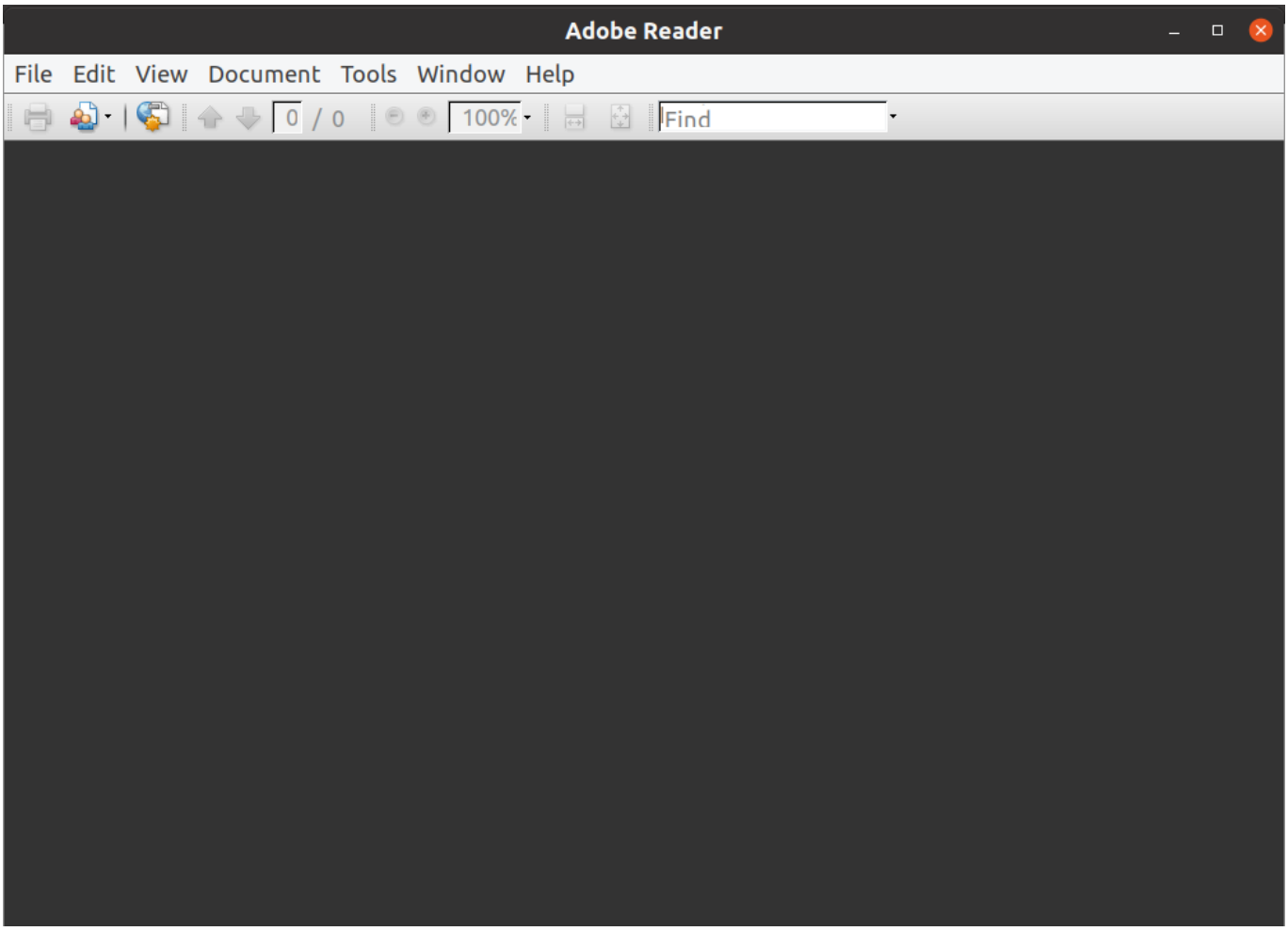
Now, we can install it with:

```
sudo dpkg -i AdbeRdr9.5.1-1_i386linux_enu.deb
```

Then just type:

```
./opt/Adobe/Reader9/bin/acroread
```

Accept the License Agreement and you should see the Adobe Reader interface:



If you type

```
/opt/Adobe/Reader9/bin/acroread -help
```

you can also see the list of available command line options

Seed corpus creation

For this exercise, I'll make use of a corpus downloaded from **SafeDocs "Issue Tracker" Corpus**. You can find more information about this PDF Corpus here.

Download and uncompress the corpus:

```
wget
https://corpora.tika.apache.org/base/packaged/pdfs/archive/pdfs_202002/lib
re_office.zip
unzip libre_office.zip -d extracted
```

Now, we will copy only files **smaller than 2 kB**, to speedup the fuzzing process:

```
mkdir -p $HOME/fuzzing_adobe/afl_in
find ./extracted -type f -size -2k \
    -exec cp {} $HOME/fuzzing_adobe/afl_in \;
```

First approach

The simplest way to fuzz closed-source applications is just running afl-fuzz with the **-Q argument**.

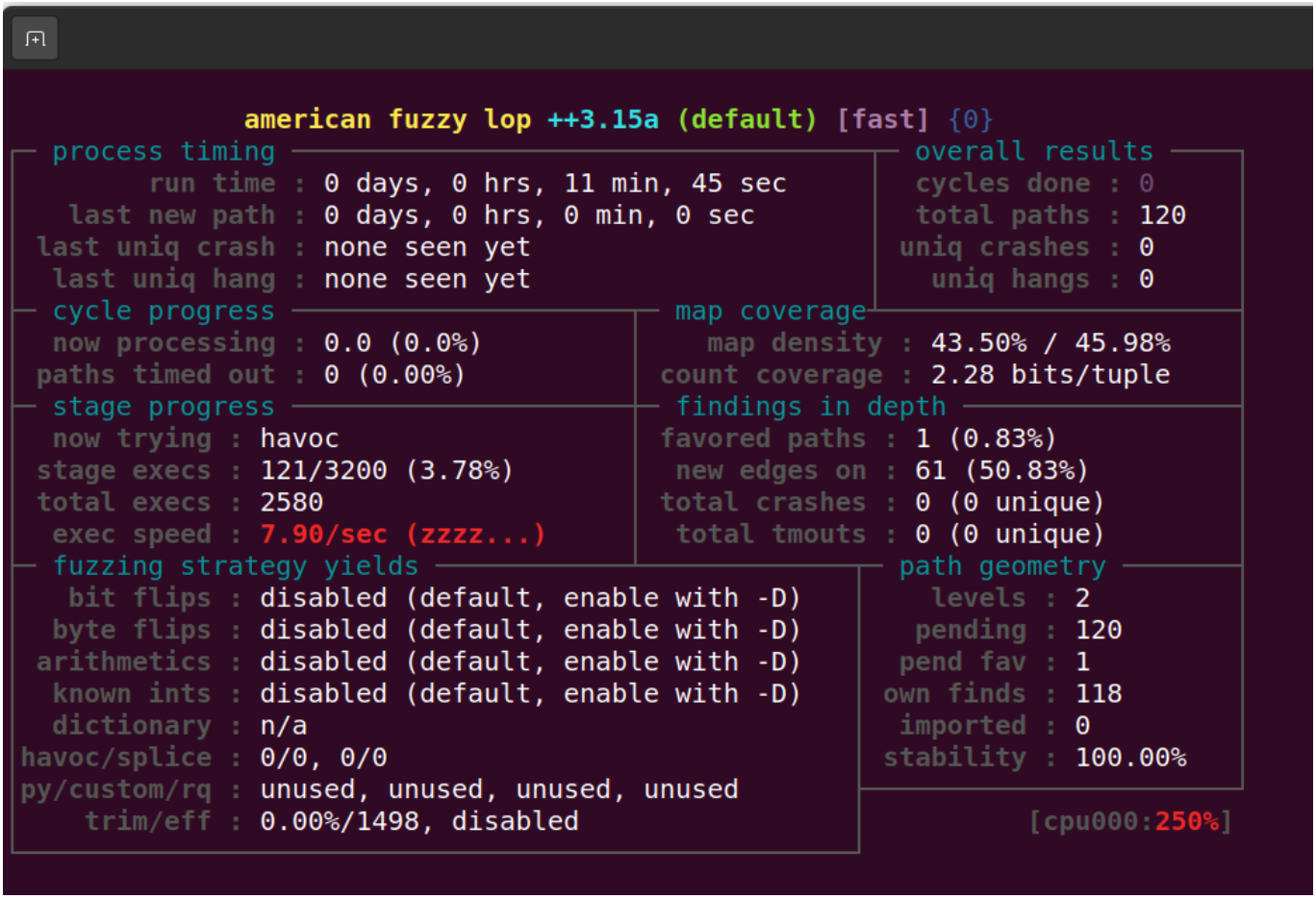
You need to take care when you run afl-fuzz, because `/opt/Adobe/Reader9/bin/acroread` is a shell-script. The real binary path is the following one `/opt/Adobe/Reader9/Reader/intellinux/bin/acroread`.

But if you try to execute it you will get an error like this: `acroread must be executed from the startup script`. That's why we need to set the `ACRO_INSTALL_DIR` and `ACRO_CONFIG` envvars. We will also set `LD_LIBRARY_PATH` to define the directory in which to search for dynamically linkable libraries.

Said that, you can run the fuzzer with the following command:

```
ACRO_INSTALL_DIR=/opt/Adobe/Reader9/Reader ACRO_CONFIG=intellinux
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:'/opt/Adobe/Reader9/Reader/intellinux/lib
' afl-fuzz -Q -i ./afl_in/ -o ./afl_out/ -t 2000 --
/opt/Adobe/Reader9/Reader/intellinux/bin/acroread -toPostScript @@
```

And you should see AFL++ running:



Now, though, the fuzzing speed is really slow: around 7 exec/s on my machine. So, how can we improve the fuzzing speed?

And the answer is... using Persistent fuzzing!

Persistent approach

As we see in exercise 6, inserting the `AFL_LOOP` is the way that we have to tell AFL++ that we want to enable the persistent mode. In this case, however, we don't have access to the source code.

We can instead make use of the AFL_QEMU_PERSISTENT_ADDR to specify the start of the persistent loop. I advise you to set this address to the beginning of a function. You can find more information about AFL_QEMU persistent options here.

For finding an appropriate offset we can make use of a disassembler like IDA or Ghidra. In my case, I chose the following offset `0x08546a00`:

```
.text:085478AC
.text:085478AC ; =============== S U B R O U T I N E =======================================
.text:085478AC
.text:085478AC ; Attributes: bp-based frame
.text:085478AC
.text:085478AC ; int __cdecl sub_85478AC(char *name, int, char)
.text:085478AC sub_85478AC proc near                   ; CODE XREF: sub_8547C66+BEB↓p
.text:085478AC                                         ; sub_8547C66+CA3↓p ...
.text:085478AC
.text:085478AC var_1435= byte ptr -1435h
.text:085478AC var_1434= dword ptr -1434h
.text:085478AC var_1430= dword ptr -1430h
.text:085478AC ptr= byte ptr -142Ch
.text:085478AC dest= byte ptr -42Ch
.text:085478AC var_2C= dword ptr -2Ch
.text:085478AC var_28= dword ptr -28h
.text:085478AC var_24= dword ptr -24h
.text:085478AC var_20= dword ptr -20h
.text:085478AC var_1C= dword ptr -1Ch
.text:085478AC var_18= dword ptr -18h
.text:085478AC var_14= dword ptr -14h
.text:085478AC var_10= dword ptr -10h
.text:085478AC name= dword ptr  8
.text:085478AC arg_4= dword ptr  0Ch
.text:085478AC arg_8= byte ptr  10h
.text:085478AC
.text:085478AC ; __unwind { // ___gxx_personality_v0
.text:085478AC push    ebp
.text:085478AD mov     ebp, esp
```

There is an easier alternative to use a disassembler: to use **callgrind**.

First of all, we will install valgrind and kcachegrind with the following command line:

```
sudo apt-get install valgrind
sudo apt-get install kcachegrind
```

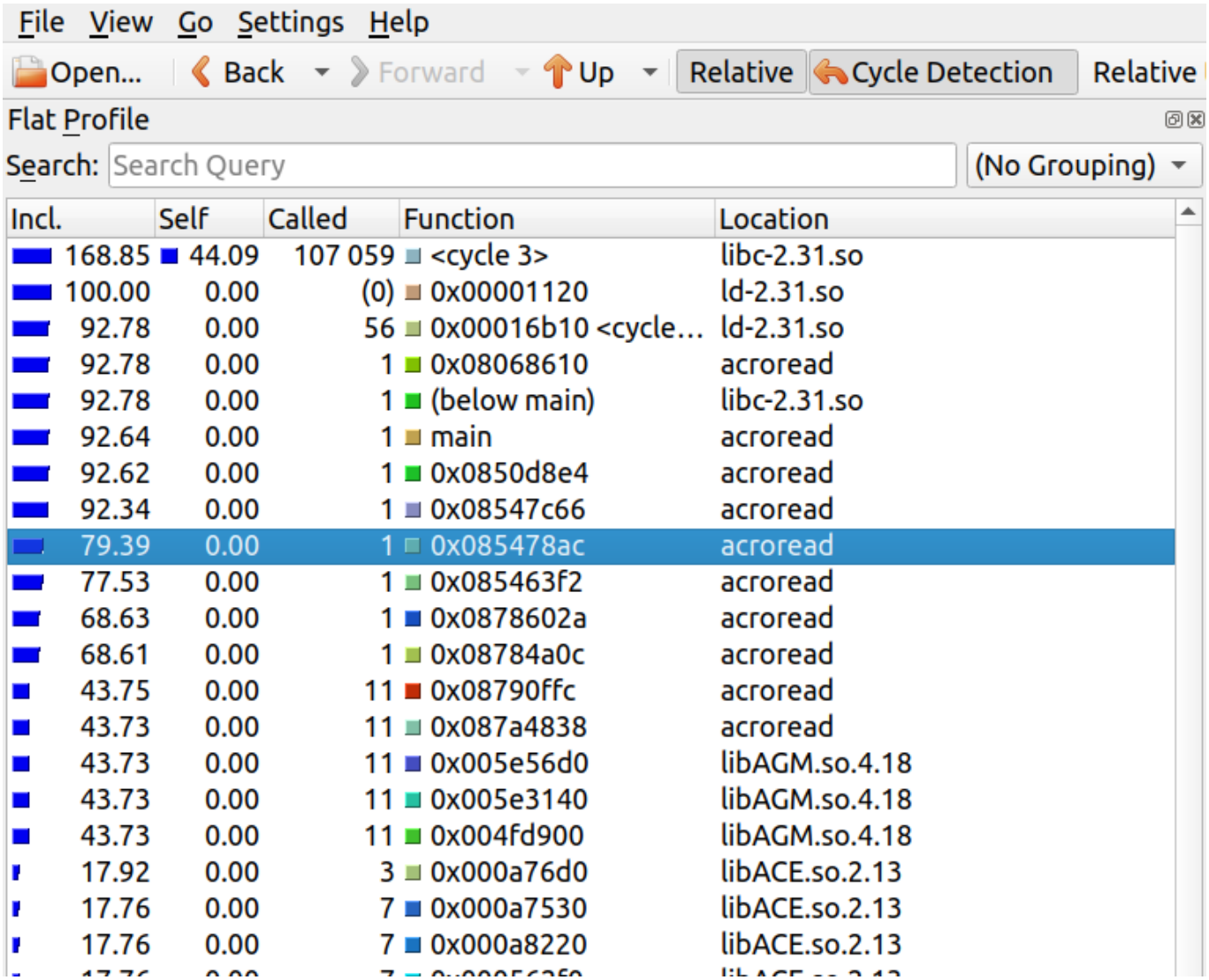
Now, we can generate a callgrind report with:

```
ACRO_INSTALL_DIR=/opt/Adobe/Reader9/Reader ACRO_CONFIG=intellinux
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:'/opt/Adobe/Reader9/Reader/intellinux/lib
' valgrind --tool=callgrind
/opt/Adobe/Reader9/Reader/intellinux/bin/acroread -toPostScript
[samplePDF]
```

where `samplePDF` is the path of a PDF sample. It will generate a `callgrind.out` output file in your current folder. Now you can visualize this report running **kachegrind**. Just type:

```
kcachegrind
```

and you should see something like this:



I recommend you to look at the `count` field in kcachegrind to identify functions that only get executed 1 time, and to try to achieve a **stability score over 90%** in afl-fuzz.

We will set also the **AFL_QEMU_PERSISTENT_GPR=1** envvar, that will save the original value of general purpose registers and restore them in each persistent cycle.

Now, we can run the fuzzer with the following command line

```
AFL_QEMU_PERSISTENT_ADDR=0x085478AC AFL_QEMU_PERSISTENT_GPR=1
ACRO_INSTALL_DIR=/opt/Adobe/Reader9/Reader ACRO_CONFIG=intellinux
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:'/opt/Adobe/Reader9/Reader/intellinux/lib
' afl-fuzz -Q -i ./afl_in/ -o ./afl_out/ -t 2000 --
/opt/Adobe/Reader9/Reader/intellinux/bin/acroread -toPostScript @@
```

As you can see, it gives a x4 improvement in time of execution. Not bad!

For small projects, this approach might be enough. In a future exercise I will explain a faster approach: **write a custom harness**.

Triage

In this last step we will try to find a PoC for our OOB read vulnerability.

Unfortunately, if we feed afl-qemu-trace with the crash file:

```
ACRO_INSTALL_DIR=/opt/Adobe/Reader9/Reader ACRO_CONFIG=intellinux
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:'/opt/Adobe/Reader9/Reader/intellinux/lib
' /usr/local/bin/afl-qemu-trace --
/opt/Adobe/Reader9/Reader/intellinux/bin/acroread -toPostScript
[crashFilePath]
```

(where `crashFilePath` is the path of the crash file), all what we can see is a message like this

```
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
```

We can get a more detailes stacktrace using **QASan**. To enable it we only need to set `AFL_USE_QASAN=1`. You can find more information about QASAN here. Now we type:

```
AFL_USE_QASAN=1 ACRO_INSTALL_DIR=/opt/Adobe/Reader9/Reader
ACRO_CONFIG=intellinux
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:'/opt/Adobe/Reader9/Reader/intellinux/lib
' /usr/local/bin/afl-qemu-trace --
/opt/Adobe/Reader9/Reader/intellinux/bin/acroread -toPostScript
[crashFilePath]
```

and we will get a nicer stacktrace:

```
=================================================================
==16290==ERROR: QEMU-AddressSanitizer: heap-buffer-overflow on address 0x09acee84 at pc 0x3ffd4102 bp 0x407fe430 sp 0x407fe3e0
READ of size 66 at 0x09acee84 thread T16290
    #0 0x3ffd4102 (/usr/lib/i386-linux-gnu/ld-2.31.so+0x1102)
    #1 0x3e0bcefb in syscall (/usr/lib/i386-linux-gnu/libc-2.31.so+0x103efb)
    #2 0x3ffc72ee in memcpy /home/antonio/Downloads/AFLplusplus/qemu_mode/libqasan/hooks.c:234
    #3 0x3ff58c1b in BIBUtilsTerminate (/opt/Adobe/Reader9/Reader/intellinux/lib/libBIBUtils.so.1.1+0x13c1b)
    #4 0x087a27dc in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x75a7dc)
    #5 0x0879ff2a in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x757f2a)

0x09aceec4 is located 0 bytes to the right of 8756-byte region [0x09accc90,0x09aceec4)
allocated by thread T16290 here:
    #0 0x3ffd4102 (/usr/lib/i386-linux-gnu/ld-2.31.so+0x1102)
    #1 0x3e0bcefb in syscall (/usr/lib/i386-linux-gnu/libc-2.31.so+0x103efb)
    #2 0x3ffc855b in __libqasan_malloc /home/antonio/Downloads/AFLplusplus/qemu_mode/libqasan/malloc.c:203
    #3 0x3ffc6fd9 in malloc /home/antonio/Downloads/AFLplusplus/qemu_mode/libqasan/hooks.c:96
    #4 0x08a3e47e in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x9f647e)
    #5 0x08aa7ab9 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0xa5fab9)
    #6 0x08a3f2e3 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x9f72e3)
    #7 0x08aa7967 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0xa5f967)
    #8 0x08aa79da in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0xa5f9da)
    #9 0x08aa7a95 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0xa5fa95)
    #10 0x08a3f2e3 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x9f72e3)
    #11 0x08a391bc in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x9f11bc)
    #12 0x088bf996 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x877996)
    #13 0x088c03f1 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x8783f1)
    #14 0x088c045f in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0x87845f)
    #15 0x08a4e248 in _localFindResource (/opt/Adobe/Reader9/Reader/intellinux/bin/acroread+0xa06248)
```

Exercise 9 - 7-Zip

For this exercise, we will fuzz **7-Zip** file archiver. The goal is to find a crash/PoC for **CVE-2016-2334** in 7-Zip 15.05.

--> Kudos to icewall for such an amazing finding.

CVE-2016-2334 is a heap-based buffer overflow that can be triggered via a crafted HFS+ image.

A heap-based buffer overflow is a type of buffer overflow that occurs in the heap data area, and it's usually related to explicit dynamic memory management (allocation/deallocation with malloc() and free() functions).

As a result, a remote attacker can exploit this issue to execute arbitrary code within the context of an application using the affected library.

You can find more information about Heap-based buffer oveflow vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/122.html

What you will learn

Once you complete this exercise you will know:

- How to use WinAFL to fuzz Windows applications

Read Before Start

- I suggest you try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- AFL uses a non-deterministic testing algorithm, so two fuzzing sessions are never the same. That's why I highly recommend **to set a fixed seed (-s 123)**. This way your fuzzing results will be similar to those shown here and that will allow you to follow the exercises more easily.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

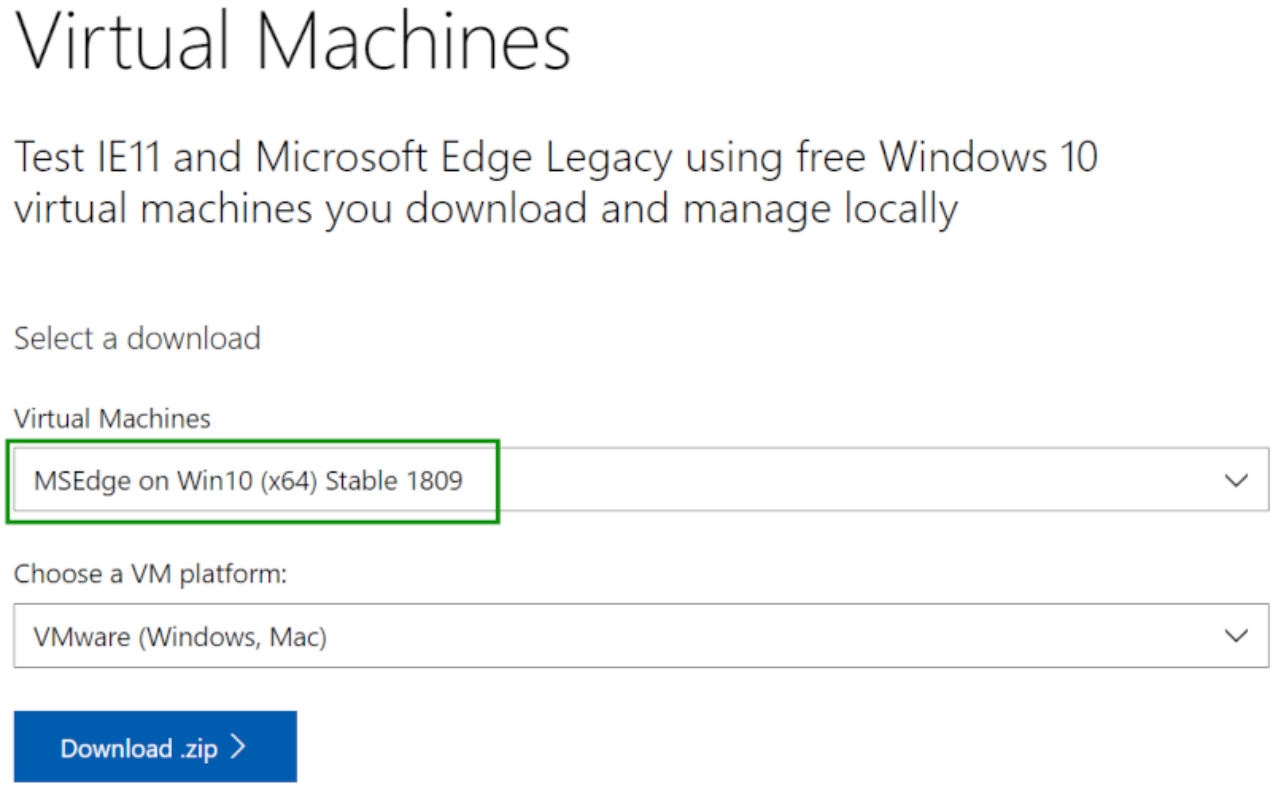
Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more fuzzing content? Do you want to share your fuzzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

This exercise has been tested on **Windows 10**. I highly recommend you to use **the same OS version** to avoid different fuzzing results

You can download a free Windows 10 VMware image at the following link. You can also use VirtualBox instead of VMware.



The password to the VM is **"Passw0rd!"**

Do it yourself!

In order to complete this exercise, you need to:

1. Download and build WinAFL (and required dependencies)
2. Create a seed corpus of HFS+ samples
3. Optional: Create a fuzzing dictionary for the HFS+ file format
4. Fuzz 7-Zip until you have a crash
5. Triage the crash to find a PoC for the vulnerability
6. Optional: Find a better *target_offset* to speed up the fuzzing process

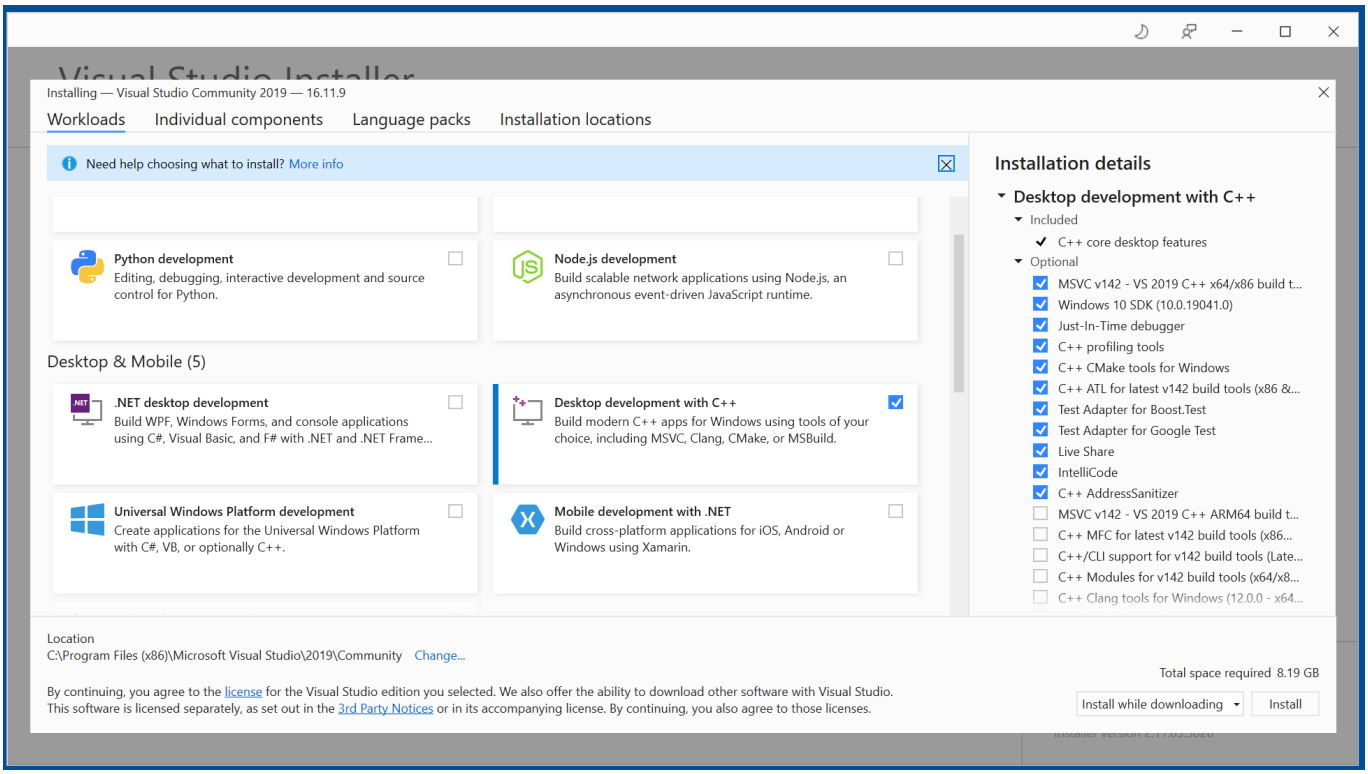
Estimated time = 8 hours

SPOILER ALERT! : Solution inside

Previous steps

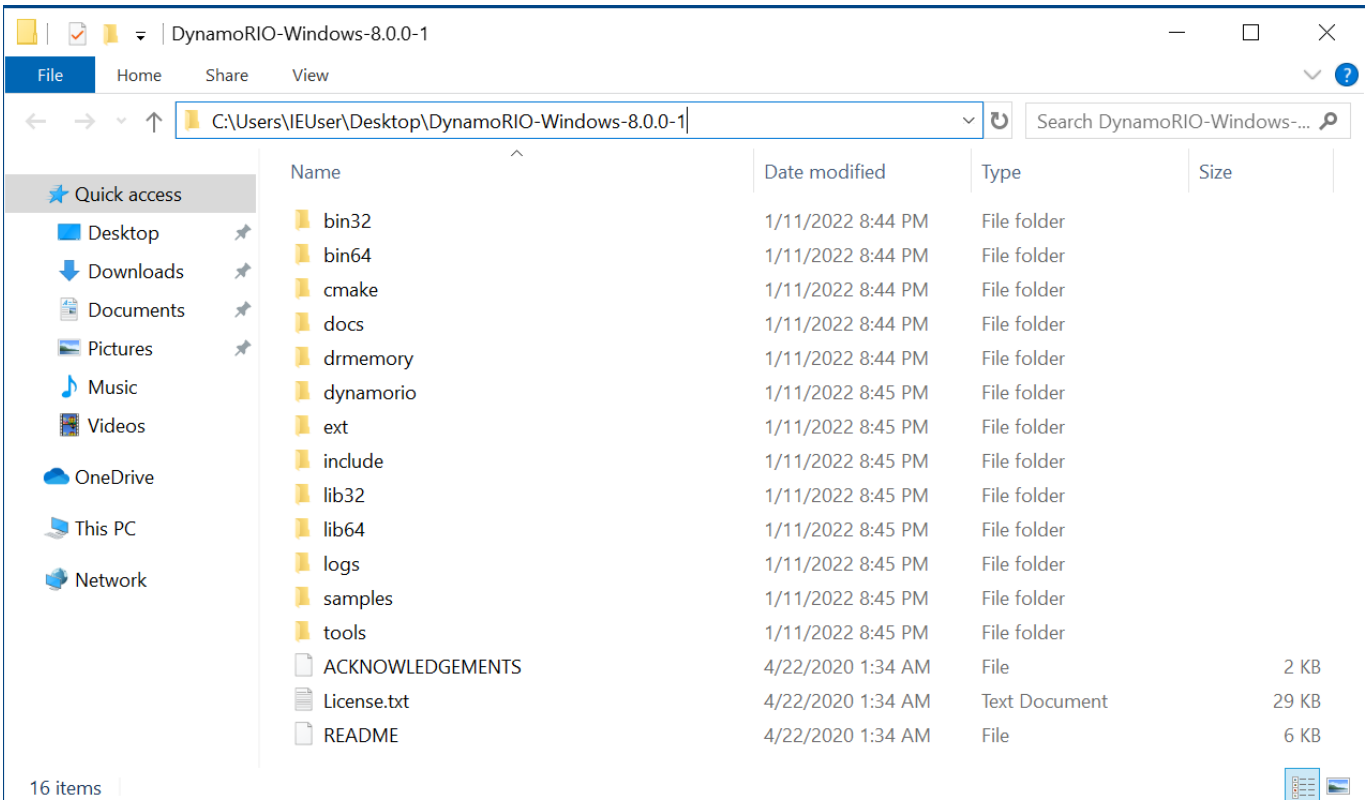
First of all, we need the **Visual Studio compiler** installed in our system. For this exercise, I recommend using Visual Studio 2019. You can find the Visual Studio 2019 Community Edition installer here.

Then we need to select and install the **"Desktop development with C++"** package:



We will also need **DynamoRIO 8.0.0**. We can get DynamoRIO Windows binary package from here.

Then, we need to extract the zip content into the Desktop, as follows:



Download and build WinAFL

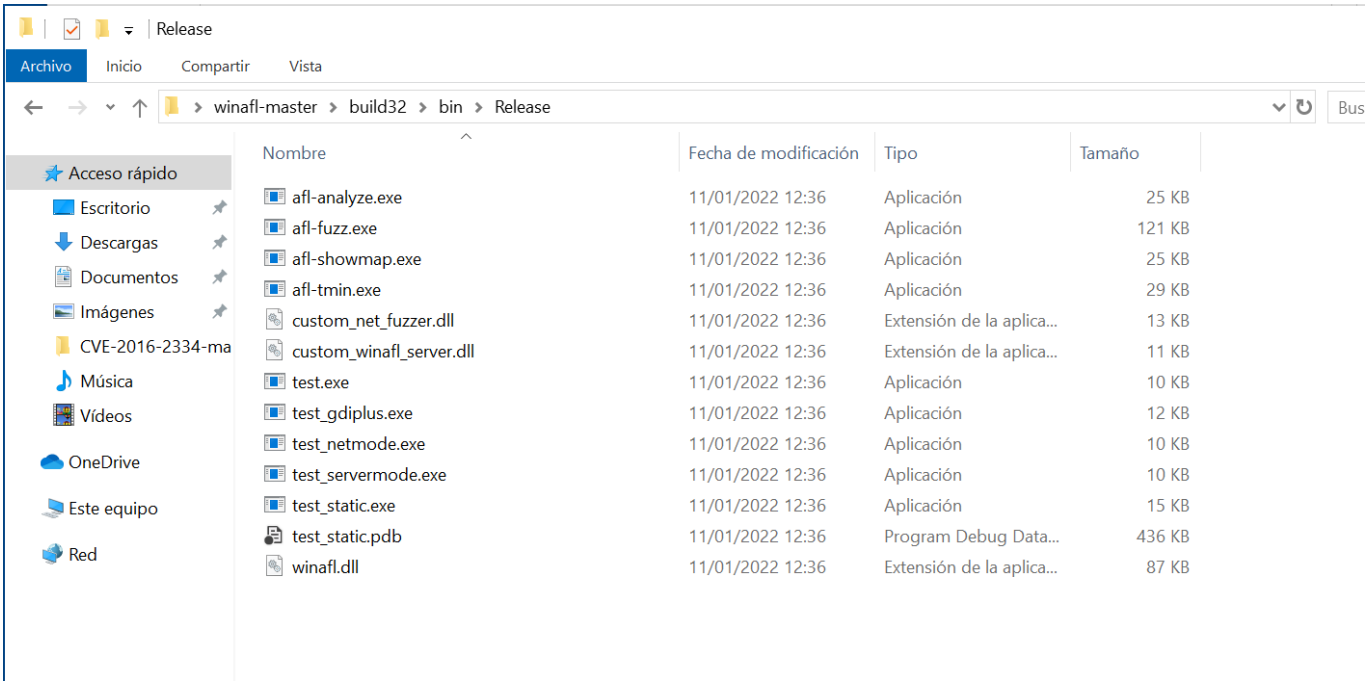
Now, we can download WinAFL from the official repository: https://github.com/googleprojectzero/winafl

After this, open **"Developer Command Prompt for VS2019"** and change the working directory to the WinAFL directory. Then type:

```
mkdir build32
cd build32
cmake -G"Visual Studio 16 2019" -A Win32 .. -
DDynamoRIO_DIR=C:\Users\IEUser\Desktop\DynamoRIO-Windows-8.0.0-1\cmake
cmake --build . --config Release
```

where `C:\Users\IEUser\Desktop\DynamoRIO-Windows-8.0.0-1\cmake` is the DynamoRIO path on your own system.

If all went well, you can now find all the WinAFL binaries in the `winafl-master\build32\bin\Release` folder:



Be careful! Don't mismatch this folder with the "bin32" folder

Download 7-Zip

Now it's time to install 7-Zip 15.05. You can find the installer here.

Seed corpus creation

I recommend you create some HFS+ images to feed your seed corpus. This is a trivial task on a Mac OS.

In Linux, you can use **hfsprogs** utility. In Windows, you can use **Paragon HFS+** (commercial software with free trial).

To make life easier, you can find an HFS example file here.

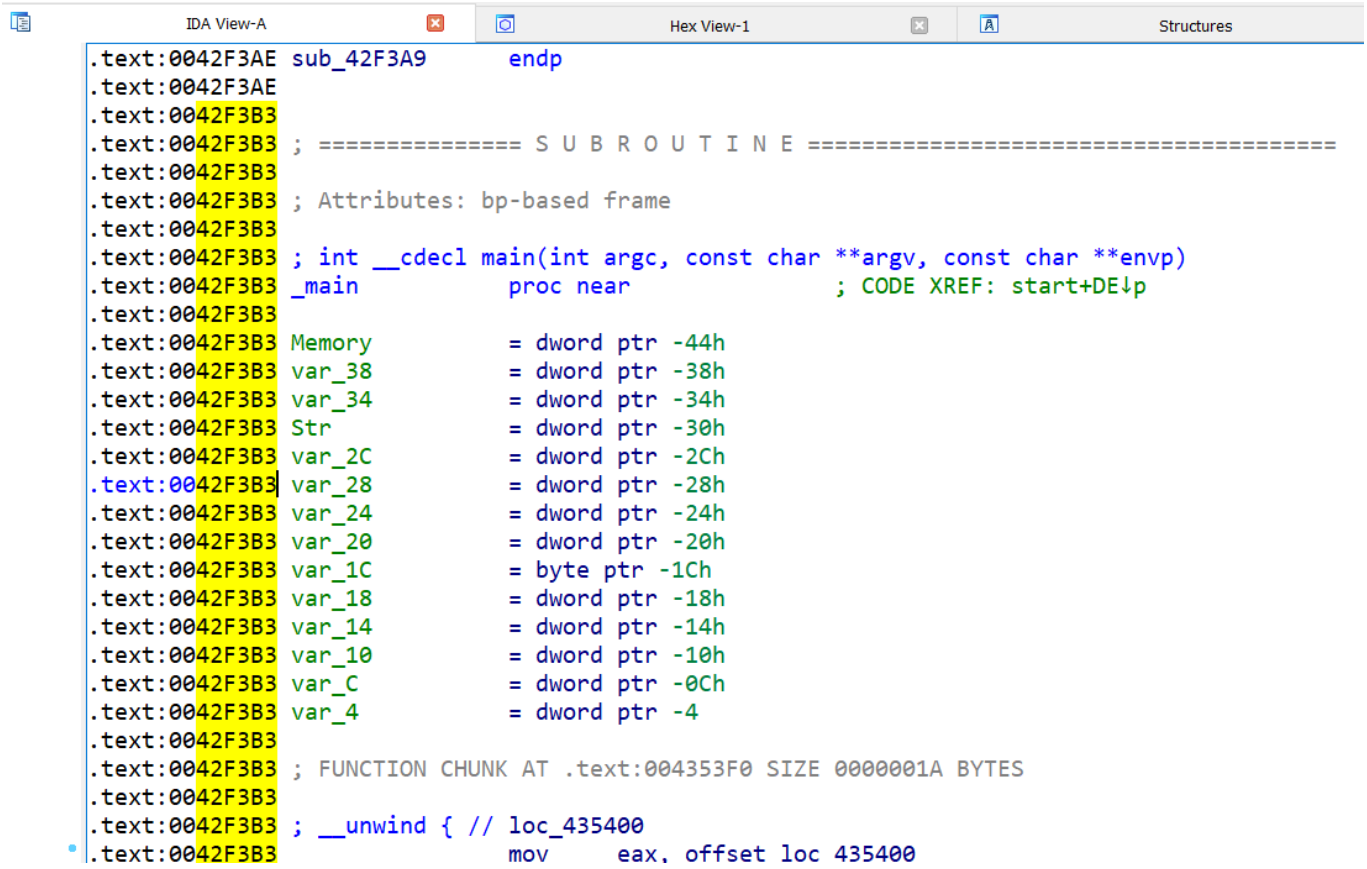
Warning! This is just a starting point, you will need to do some extra work on your own

Fuzzing time

The WinAFL command line is a little bit different than AFL++. Let's see a brief explanation of these new options:

- *-coverage_module* : module for which to record coverage. Multiple module flags are supported
- *-target_module* : module which contains the target function to be fuzzed
- *-target_offset* : offset of the method to fuzz from the start of the module

As we did in exercise 8, we need to find an appopiate function offset from where the fuzzer will loop:



We need the offset of the function from the start of the module. Since the base address is `0x400000`, we will do `0x42F3B3 - 0x400000` and will get `0x02F3B3` as a **target_offset** argument.

Now, let's check that the target is running correctly under DynamoRIO:

```
C:\Users\IEUser\Desktop\DynamoRIO-Windows-8.0.0-1\bin32\drrun.exe -c
winafl.dll -debug -target_module 7z.exe -target_offset 0x02F3B3 -
fuzz_iterations 10 -nargs 2 -- "C:\Program Files (x86)\7-Zip\7z.exe" l
C:\Users\IEUser\Desktop\input\test.img
```

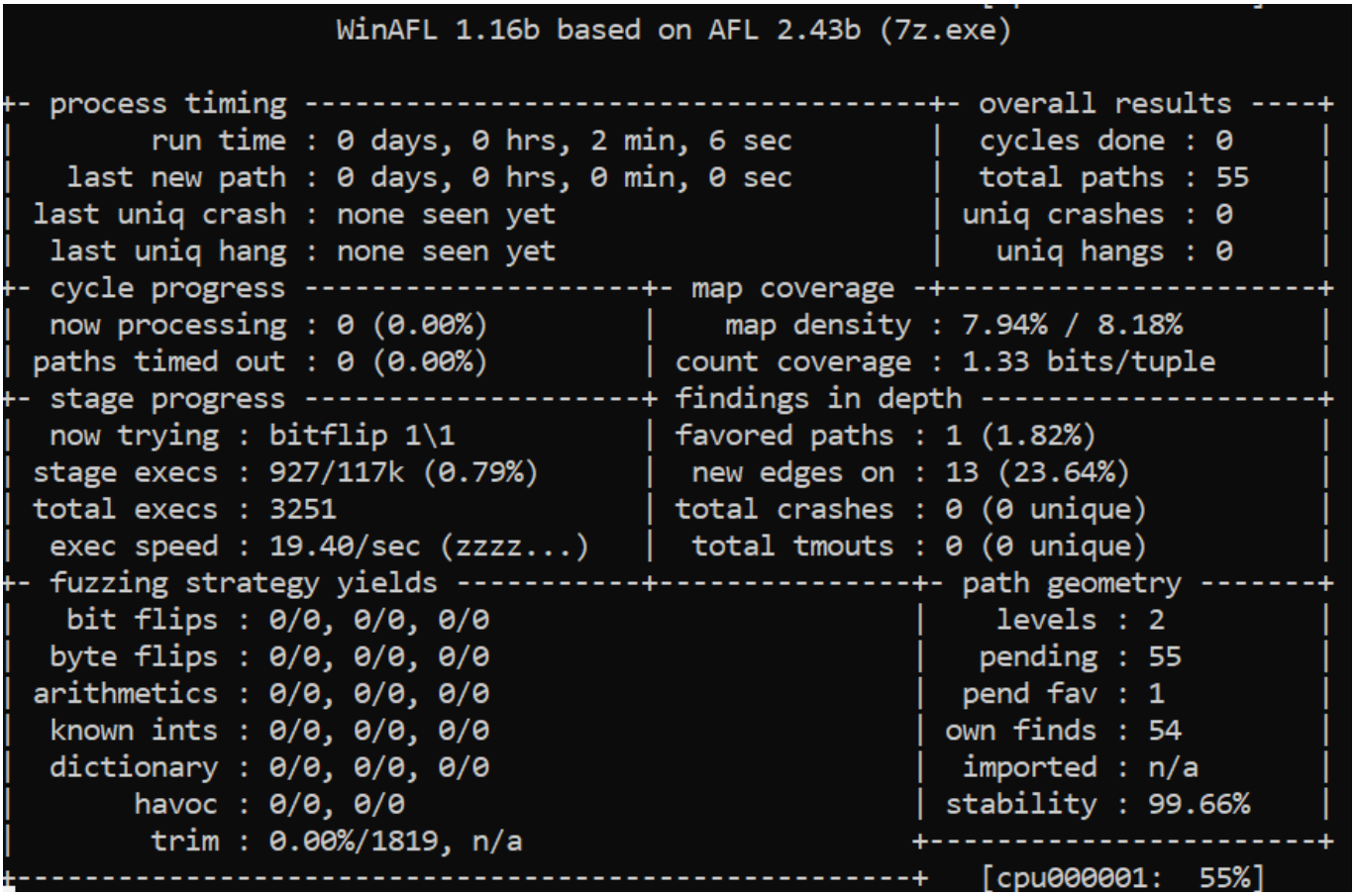
You should see the output corresponding to your target function being run 10 times after which the target executable will exit.

Finally, we can run the fuzzer with the following command:

```
afl-fuzz.exe -i C:\Users\IEUser\Desktop\afl_in -o
C:\Users\IEUser\Desktop\afl_out -t 2000 -D
```

```
C:\Users\IEUser\Desktop\DynamoRIO-Windows-8.0.0-1\bin32 -- -
coverage_module 7z.exe -coverage_module 7z.dll -target_module 7z.exe -
target_offset 0x02F3B3 -nargs 2 -- "C:\Program Files (x86)\7-Zip\7z.exe" e
-y @@
```

And you should see WinAFL running:



Find a better target_offset

The last step of the exercise is to find a better target_offset to speed up the fuzzing process. This last part is left as an exercise for the student.

Hint: 7-Zip is open-source. So you can compile it in debug mode and see function names in your debugger 😉

Exercise 10 (Final Challenge) - V8 engine

For this exercise we will fuzz **V8** Google's JavaScript and WebAssembly engine. The goal is to find a crash/PoC for **CVE-2019-5847** in v8 7.5.

CVE-2019-5847 is a vulnerability that may cause an infinite recursion via a crafted file.

A heap-based memory corruption is a type of memory corruption that occurs in the heap data area, and it's usually related to explicit dynamic memory management (allocation/deallocation with malloc() and free() functions).

As a result, a remote attacker can exploit this issue to execute arbitrary code within the context of an application using the affected library.

You can find more information about Heap-based memory corruption vulnerabilities at the following link: https://cwe.mitre.org/data/definitions/122.html

What you will learn

Once you complete this exercise you will know-how:

- How to use Fuzzilli to fuzz Javascript engines

Read Before Start

- I suggest you try to **solve the exercise by yourself** without checking the solution. Try as hard as you can, and only if you get stuck, check out the example solution below.
- If you find a new vulnerability, **please submit a security report** to the project. If you need help or have any doubt about the process, the GitHub Security Lab can help you with it 😃

Contact

Are you stuck and looking for help? Do you have suggestions for making this course better or just positive feedback so that we create more similar content? Do you want to share your fuzing experience with the community? Join the GitHub Security Lab Slack and head to the `#fuzzing` channel. Request an invite to the GitHub Security Lab Slack

Environment

All the exercises have been tested on **Ubuntu 18.04 LTS**. I highly recommend you to use **the same OS version** to be able to follow the guidelines. You can find an Ubuntu 18.04 LTS VMware image here. You can also use VirtualBox instead of VMware.

The username / password for this VM are `fuzz` / `fuzz`.

Fuzzilli

Fuzzilli is a (coverage-)guided fuzzer for dynamic language interpreters based on a custom intermediate language ("FuzzIL") which can be mutated and translated to JavaScript. It's written and maintained by Samuel Groß, saelo@google.com.

You can find more information about Fuzzilli at the official repository

Install dependencies

To get your environment fully ready, you may need to install some additional tools (ex. Swift, Git)

Let's start installing the following dependencies:

```
sudo apt --yes install clang libcurl3 libpython2.7 libpython2.7-dev
libcurl4 git
```

Now, we can download Swift:

```
cd $HOME
wget https://swift.org/builds/swift-4.2.1-release/ubuntu1804/swift-4.2.1-
RELEASE/swift-4.2.1-RELEASE-ubuntu18.04.tar.gz
tar xzvf swift-4.2.1-RELEASE-ubuntu18.04.tar.gz
sudo mv swift-4.2.1-RELEASE-ubuntu18.04 /usr/share/swift
```

We also need to configure the PATH envvar:

```
echo "export PATH=/usr/share/swift/usr/bin:$PATH" >> ~/.bashrc
source  ~/.bashrc
```

Install Fuzzilli

Time for download and build Fuzzilli:

```
cd $HOME
wget
https://github.com/googleprojectzero/fuzzilli/archive/refs/tags/v0.9.zip
unzip v0.9.zip
cd fuzzilli-0.9/
swift build -c release -Xlinker='-lrt'
```

Download and build V8 Engine

Download and setup depot_tools

V8 use a package of scripts called depot_tools to manage checkouts and code reviews. The depot_tools package includes gclient, gcl, git-cl, repo, and others. We can install it with:

```
cd $HOME
mkdir depot_tools && cd depot_tools
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
echo "export PATH=`pwd`/depot_tools:$PATH" >> ~/.bashrc
source  ~/.bashrc
```

Get V8 source code:

```
cd $HOME
mkdir Fuzzing_v8_75 && cd Fuzzing_v8_75
fetch v8
cd v8
git checkout 1ca088652d3aad04caceb648bcffef100bc4abc0
gclient sync
```

Build and test V8:

First of all we need to install build dependencies:

```
./build/install-build-deps.sh
```

Generate build files using gn:

```
gn gen out/Release "--args=is_debug=false"
```

After that, we can compile with the following command line (it may take some time):

```
ninja -C out/Release
```

Now, we can check the d8 binary with:

```
./out/Release/d8 ./test/fuzzer/parser/hello-world
```

and you should see something like this:

```
fuzz@ubuntu:~/Fuzzing_v8_75/v8$ ./out/Release/d8 ./test/fuzzer/parser/hello-world
hello world
fuzz@ubuntu:~/Fuzzing_v8_75/v8$
```

Fuzzing time

Compile v8 with coverage instrumentation:

```
cd $HOME/Fuzzing_v8_75/v8
cp ../../fuzzilli-0.9/Targets/V8/v8.patch ./
gn gen out/fuzzbuild --args='is_debug=false dcheck_always_on=true
v8_static_library=true v8_enable_slow_dchecks=true
v8_enable_v8_checks=true v8_enable_verify_heap=true
v8_enable_verify_csa=true v8_enable_verify_predictable=true
sanitizer_coverage_flags="trace-pc-guard" target_cpu="x64"'
ninja -C ./out/fuzzbuild
```

Move to fuzzilli folder:

```
cd $HOME/fuzzilli-0.9
```

First of all, we need to disable the creation of core dump files:

```
sudo sysctl -w 'kernel.core_pattern=|/bin/false'
```

Now we can run fuzzilli with:

```
swift run -Xlinker='-lrt' -c release FuzzilliCli --profile=v8
'/home/fuzz/Fuzzing_v8_75/v8/out/fuzzbuild/d8'
```

For a complete list of options, type:

```
swift run -Xlinker='-lrt' FuzzilliCli --help
```

If all went well, you should see something like:

Do it yourself!

In order to solve this challenge, you need to:

1. Download and build the vulnerable version
2. Define a custom mutator to target the vulnerable code
3. Fuzz with Fuzzilli until you have a few unique crashes
4. Triage the crashes to find a PoC for the vulnerability
5. Send your PoC (see below)

Challenge rules

The rules are as follows:

- Please, don't disclose your solution
- In order to be the winner, you must provide a valid crash/PoC file
- There will be 2 winners in total
- I'll release a new hint every week
- Send your solution by **28 February** to antoniomoralesmaldonado@gmail.com
- The winners will receive a coupon to spend in the GitHub Shop

Hints

- 1º hint:

```
git checkout 7.5.288.22
```

This challenge ended on March 1, 2022. Thank you for your submissions!! 😀