Fuzzing: A gentle introduction.

Off-by-One Security Workshop

Whoami

- Security researcher from Belgium (insert Belgian joke here)
- I like fuzzing and building tools (to break other tools)
- My main experience is on Linux/Windows (but I am learning mobile)
- You can find some of my projects on Github https://github.com/20urc3 or articles on my website https://bushido-sec.com/

Disclaimer

There is a lot of people way more experienced and qualified than me to talk about fuzzing. Without their work, I wouldn’t be able to present (almost) any of the following tools and techniques, I’m just resting on the shoulders of giants.

Please find at the end of this presentation a “going further” section linking some of the amazing work those giants have put out there.

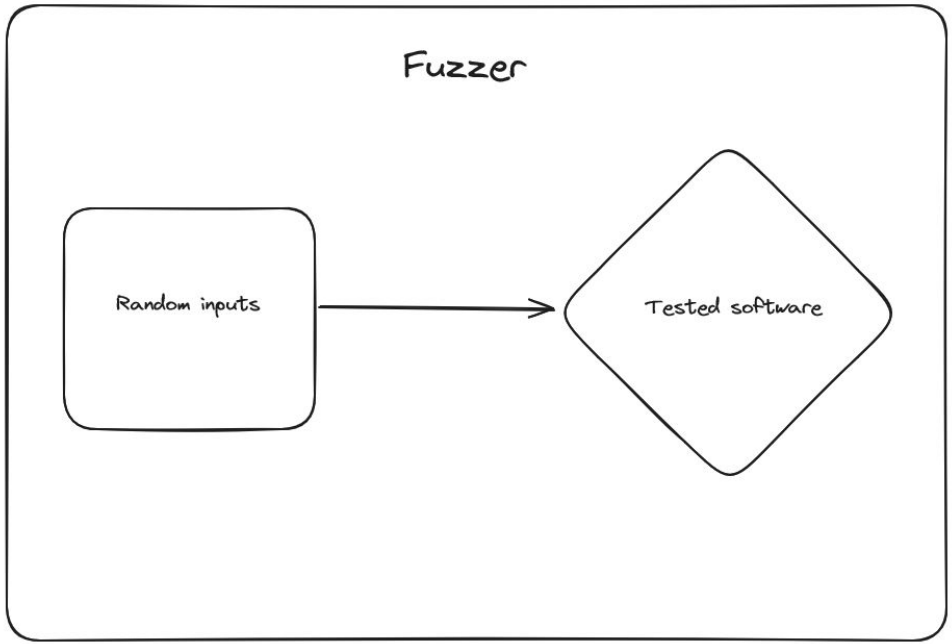
Introduction

What is fuzzing ?

In recent years, fuzzing has become a standard technique for testing various types of software. The concept of fuzzing is quite old, and numerous definitions of it exist. Instead of delving into the historical details of its creation, I will summarize it in simple terms: Throwing random inputs to a software program and observing the resulting behavior.

If you have ever played a video game for an extended period or used any software intensively, you are likely familiar with encountering unusual behaviors. These anomalies can range from the program crashing unexpectedly to finding yourself in unintended areas of the game. Additionally, you might experience issues where a properly saved document refuses to open correctly, instead displaying a series of garbled characters, thereby forcing you to redo your work from scratch.

Software can be conceptualized as a finite game, where you play by specific rules, and the game concludes once the goal is achieved. Encountering a crash or unexpected behavior can be likened to a series of actions within these rules that result in unforeseen outcomes. Fuzzing automates the recreation of such interactions, enabling developers and security researchers to identify and address new bugs within the software.

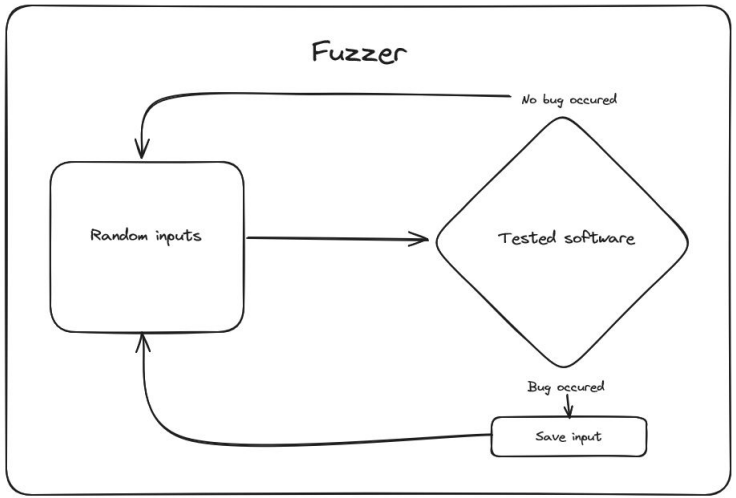


OFFBYONE

Fuzzing concepts

Observation

In our previous example, we merely replicated standard user behavior, albeit in an accelerated and automated manner. However, this approach overlooks a crucial aspect of fuzzing: the need for efficient monitoring. To effectively identify vulnerabilities, a fuzzer must be capable of tracking the program's state during testing, thereby enabling the detection and recording of crashes or unexpected behavior as well as saving the input that triggered these events.



OFFBYONE

Fuzzing concepts

Observation

Recording crashes can be a relatively straightforward process, as most operating systems provide mechanisms for monitoring processes and detecting abnormal termination signals. However, capturing unexpected behavior poses a more significant challenge. Consider, for instance, an input that triggers an out-of-bounds write bug in the tested software, but does not induce a crash since the affected memory region is not utilized or initialized by the software. The question then becomes: how can we effectively record and identify such a bug, which constitutes a valid vulnerability and a potentially exploitable primitive?

Fuzzing concepts

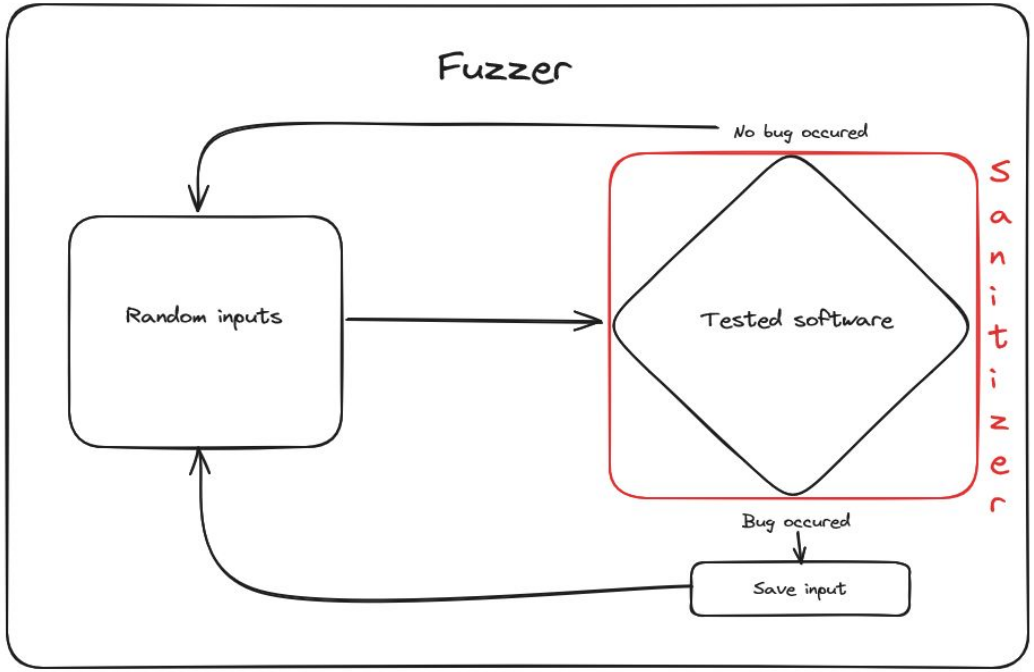
Observation

One potential solution to this challenge lies in the use of sanitizers. The LLVM project offers an extremely powerful sanitizer called AddressSanitizer (Asan), which can effectively detect memory corruption bugs. Additionally, Microsoft provides an alternative sanitizer, also called AddressSanitizer, which serves a similar purpose. For binary-only targets, tools like GFlags and page heap can be employed to detect memory-related issues.

- https://clang.llvm.org/docs/AddressSanitizer.html
- https://learn.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170
- https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap

Fuzzing concepts

Observation



OFFBYONE

Fuzzing concepts

Mutation engine

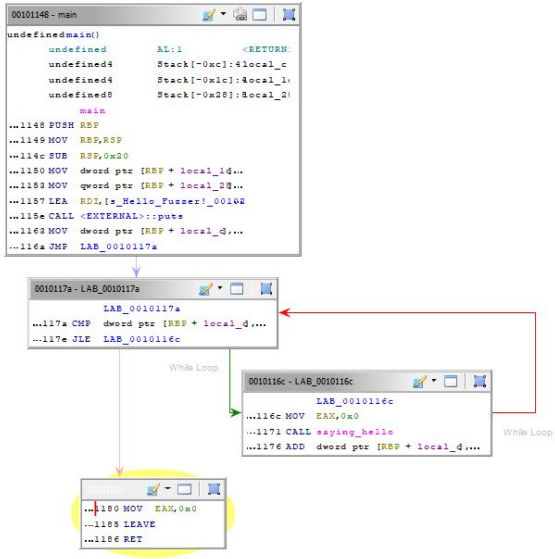
The random inputs we feed our target software don't magically appear; instead, we generate them. So far, we've adopted a purely random approach, without considering whether our inputs are actually processed by the target. While this method can be sufficient in some cases, it's far from efficient. For instance, if a file lacks the magic bytes indicating it's a .png, the target program won't even be able to parse it. This poses an intriguing challenge for any fuzzer developer. In my experience, this problem can be addressed through three primary solutions: structure-aware mutation, code-coverage mutation, and high-performance mutation.

Fuzzing concepts

Code-coverage mutation engine

When a program is compiled, it is translated into a long sequence of assembly instructions. These instructions are often grouped together to perform specific tasks and can be thought of as a so-called "block". Code-coverage-based mutation engines aim to solve the problem of generating "valid enough" files that can be processed by the program by tracking how many paths are reached when feeding the file to the target program.

Here, the disassembled code is organized into "blocks" that represent the program's execution flow. In this simple example, we can see the main block, the condition check block for the loop, the block calling the function that prints "hello", and finally, the block that exits the software. While this example is trivial, it's easy to imagine that in real-world applications, there are thousands of blocks that represent a complex flow.



OFFBYONE

Fuzzing concepts

Code-coverage mutation engine

In the example above you can see that each block are linked by a an arrow. Those are called Edges or Branches, and as you can guess, they represent all the path a program must follow in order to be executed. The fuzzer goal to know which block has been reached or not, this is made possible by using instrumentation.

Let’s take a very simple C program as example.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
prinf("Hello");
return 0;
}
```

OFFBYONE

Fuzzing concepts

Code-coverage mutation engine

You can see that some instruction are executed followed by a call to the __AFL_MAYBE_LOG function. This is one of the mechanisms that enable AFL to achieve code coverage. The assembly block is instrumented to track whether a certain point has been reached or not, and it uses this information to improve randomized inputs based on the results of the most frequently taken paths with certain types of inputs.

```
Dump of assembler code for function main:
   0x0000000000001180 <+0>:     lea    rsp,[rsp-0x98]
   0x0000000000001188 <+8>:     mov    QWORD PTR [rsp],rdx
   0x000000000000118c <+12>:    mov    QWORD PTR [rsp+0x8],rcx
   0x0000000000001191 <+17>:    mov    QWORD PTR [rsp+0x10],rax
   0x0000000000001196 <+22>:    mov    rcx,0x4bad
   0x000000000000119d <+29>:    call   0x12d0 <__afl_maybe_log>
   0x00000000000011a2 <+34>:    mov    rax,QWORD PTR [rsp+0x10]
   0x00000000000011a7 <+39>:    mov    rcx,QWORD PTR [rsp+0x8]
   0x00000000000011ac <+44>:    mov    rdx,QWORD PTR [rsp]
   0x00000000000011b0 <+48>:    lea    rsp,[rsp+0x98]
   0x00000000000011b8 <+56>:    endbr64
   0x00000000000011bc <+60>:    sub    rsp,0x8
   0x00000000000011c0 <+64>:    lea    rsi,[rip+0xe3d]        # 0x2004
   0x00000000000011c7 <+71>:    mov    edi,0x1
   0x00000000000011cc <+76>:    xor    eax,eax
   0x00000000000011ce <+78>:    call   0x1130 <__printf_chk@plt>
   0x00000000000011d3 <+83>:    xor    eax,eax
   0x00000000000011d5 <+85>:    add    rsp,0x8
   0x00000000000011d9 <+89>:    ret
End of assembler dump.
```

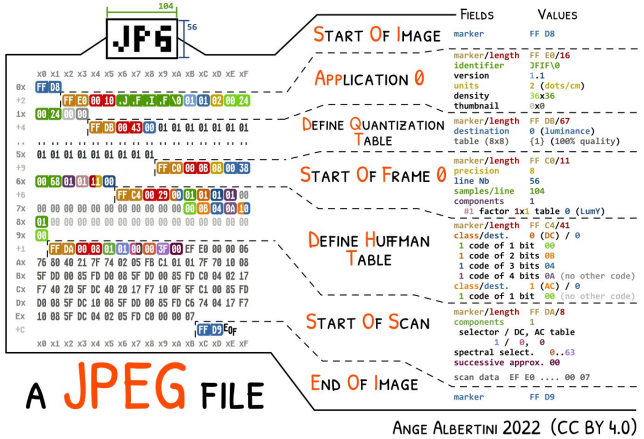
OFFBYONE

Fuzzing concepts

Structure-aware mutation engine

File formats are relatively straightforward to work with, as most of them are well-defined and have a structure that's easy to comprehend, especially when they're open-source. Take, for instance, the .jpeg file format, which has been widely used for many years and is extremely well-documented.

The amazing illustration provided by Ange Albertini, which describes the JPEG file format, is truly remarkable. For a fuzzer developer targeting this specific file format, it's relatively straightforward to create a mutation engine that preserves essential or mandatory bytes while only performing mutations on the rest of the file. This approach can be very efficient in achieving greater coverage, although it has the drawback of potentially missing bugs that a completely random file might have uncovered.



OFFBYONE

Fuzzing concepts

High performance mutation engine

Some problems can be solved with money, this is a harsh reality that every fuzzer developer must confront: if you have a large enough infrastructure to run a massive number of fuzzer instances, each performing "good enough" mutations, eventually, one of them will produce a valid enough input to be processed by the test software and uncover bugs. While this approach will waste a tremendous amount of CPU cycles, it remains a valid point. Sometimes, overcomplicating mutation engines can struggle to find what a simple, fast, and completely random file generator would have discovered.

The role of fuzzing in software security

Fuzzing is a valuable technique for both developers and security researchers. Developers can leverage fuzzing to enhance the reliability and security of their software by identifying and addressing potential issues.

Conversely, security researchers can utilize fuzzing to uncover vulnerabilities and potentially exploit them. As a security researcher, fuzzing is a particularly useful technique for automating the testing pipeline of various targets.

For instance, one could compile and fuzz every new version of a specific program known to have potential bugs or critical components. It is essential to note that the role of fuzzing is not to replace manual source code auditing or static analysis but rather to provide a practical and easy-to-use tool that complements these methods.

Existing fuzzing types

Open source

- Easier to work with.
- More tested, lower chance to find something.
- Easier to develop exploit for.

Closed source

- More difficult to work with.
- Include Reverse Engineering and patching phase.
- Harder to exploit.
- Less tested, higher chance to find something.

Existing fuzzing types

Network protocol fuzzing
Parser fuzzing
Grammar-based fuzzing
Snapshot fuzzing
(Linux) Kernel fuzzing
Smart contract fuzzing

Pre-requisites

I tried my best to provide a “just run me” script that would install everything for this workshop on your machine, however you might still encounter some issues. On the next slide you will find the complete list of tools we are going to use, you can find the installation process for each on their respective readme.

The installation script is:

- https://github.com/20urc3/Talks/blob/main/Off-By-One/install_linux.sh

Tools for the workshop:

- AFL++: https://aflplus.plus/
- Fuzzilli: https://github.com/googleprojectzero/fuzzilli
- Jackalope: https://github.com/googleprojectzero/Jackalope
- Syzkaller: https://github.com/google/syzkaller
- Python3: https://www.python.org/downloads/

Workshop

Writing a fuzzer

For those who are genuinely committed to software testing and fuzzing, creating a custom tool is an inevitable necessity. The notion that fuzzers should be regularly discarded and rebuilt for each new target is a sentiment shared by many. While I believe that certain components of the code can be reused or features developed to endure across multiple generations of fuzzers, I firmly concur with the idea of crafting a new fuzzer tailored to each specific target encountered. This approach not only enhances coding proficiency but also refines one's thought process, compelling the developer to thoroughly understand the target and devise an optimal strategy for exhaustive testing of the target software.

Writing a fuzzer

Sending random inputs to a target program
Creating a mutation engine
Monitor target state
Record crash and save interesting input cases

Example: https://github.com/20urc3/Aplos

Code-coverage fuzzing with AFL++

We will follow this tutorial: https://bushido-sec.com/index.php/2023/06/19/the-art-of-fuzzing/

```
# You can obtain Vim for the first time with:
git clone https://github.com/vim/vim.git

# CD into the Vim directory
cd vim

# Rolling back to a vulnerable version
git checkout v8.1.2122
```

Code-coverage fuzzing with AFL++

We will follow this tutorial: https://bushido-sec.com/index.php/2023/06/19/the-art-of-fuzzing/

```
# Configure VIM to be compiled with AFL options
CC=afl-clang-fast CXX=afl-clang-fast++ ./configure --with-features=huge --enable-gui=none

# Compiling
make -j4
```

```
# cd into the src folder
cd src/

# Creating corpus folder and 2 basic corpus file
mkdir corpus output
echo "a*b\+\|[0-9]\|\d{1,9}" > corpus/1 ; echo "^\d{1,10}$" > corpus/2

# Adding regex dictionnary
wget https://raw.githubusercontent.com/vanhauser-thc/AFLplusplus/master/dictionaries/regexp.dict
```

Code-coverage fuzzing with AFL++

We will follow this tutorial: https://bushido-sec.com/index.php/2023/06/19/the-art-of-fuzzing/

```
# Running the fuzzing campaign
afl-fuzz -m none -i corpus -o output ./vim -u NONE -X -Z -e -s -S @@ -c ':qa!'
```

Windows binaries fuzzing with Jackalope

Jackalope is another great tool originally created by Ivan Fratric, it's in a way the successor of WinAFL. It allows you to run fuzzing campaign on MacOS, Windows, Linux.

We will follow this tutorial: https://bushido-sec.com/index.php/2023/06/25/the-art-of-fuzzing-windows-binaries/

Linux kernel fuzzing with Syzkaller

All the instructions are detailed here:
https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md

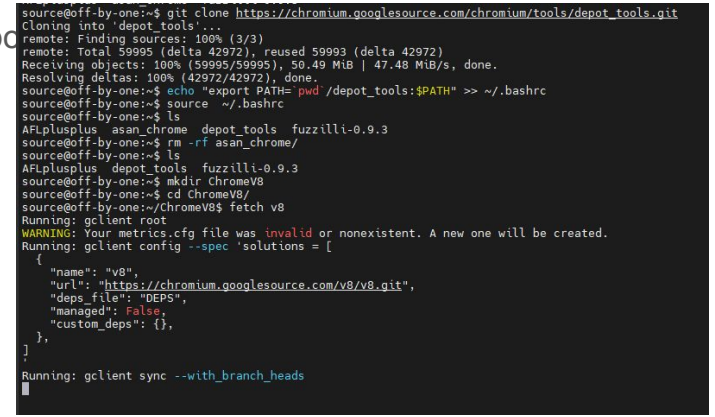
Bonus: Fuzzing with Fuzzilli

Fuzzing chrome V8:

A lot of us have seen on socials post talking about vulnerabilities in the javascript engine v8 used by many browser. Here we are going to fuzz directly this engine using a Javascript engine fuzzer developed by Google Project Zero team called Fuzzilli. A researcher named Antonio Morales made a very amazing series of tutorial about fuzzing, one of them treated the topic of fuzzing v8 engine, https://github.com/antonio-morales/Fuzzing101/tree/main/Exercise%2010 the following demonstration is directly inspired from his work and not mine.

Download and compile v8:

- git clone https://chromium.googlesource.com/chromium/tools/depo
- echo "export PATH=`pwd`/depot_tools:$PATH" >> ~/.bashrc
- source ~/.bashrc
- mkdir chrome v8
- cd chrome v8
- fetch v8
- gclient sync



OFF BY ONE

Bonus: Fuzzing with Fuzzilli

Fuzzing chrome V8:

Download and compile v8:

- git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
- echo "export PATH=`pwd`/depot_tools:$PATH" >> ~/.bashrc
- source ~/.bashrc
- mkdir chrome v8
- cd chrome v8
- fetch v8
- gclient sync

```
source@off-by-one:~$ git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
Cloning into 'depot_tools'...
remote: Finding sources: 100% (3/3)
remote: Total 59995 (delta 42972), reused 59993 (delta 42972)
Receiving objects: 100% (59995/59995), 50.49 MiB | 47.48 MiB/s, done.
Resolving deltas: 100% (42972/42972), done.
source@off-by-one:~$ echo "export PATH=`pwd`/depot_tools:$PATH" >> ~/.bashrc
source@off-by-one:~$ source  ~/.bashrc
source@off-by-one:~$ ls
AFLplusplus  asan_chrome  depot_tools  fuzzilli-0.9.3
source@off-by-one:~$ rm -rf asan_chrome/
source@off-by-one:~$ ls
AFLplusplus  depot_tools  fuzzilli-0.9.3
source@off-by-one:~$ mkdir ChromeV8
source@off-by-one:~$ cd ChromeV8/
source@off-by-one:~/ChromeV8$ fetch v8
Running: gclient root
WARNING: Your metrics.cfg file was invalid or nonexistent. A new one will be created.
Running: gclient config --spec 'solutions = [
  {
    "name": "v8",
    "url": "https://chromium.googlesource.com/v8/v8.git",
    "deps_file": "DEPS",
    "managed": False,
    "custom_deps": {},
  },
]
'
Running: gclient sync --with_branch_heads
```

OFF BY ONE

Bonus: Fuzzing with Fuzzilli

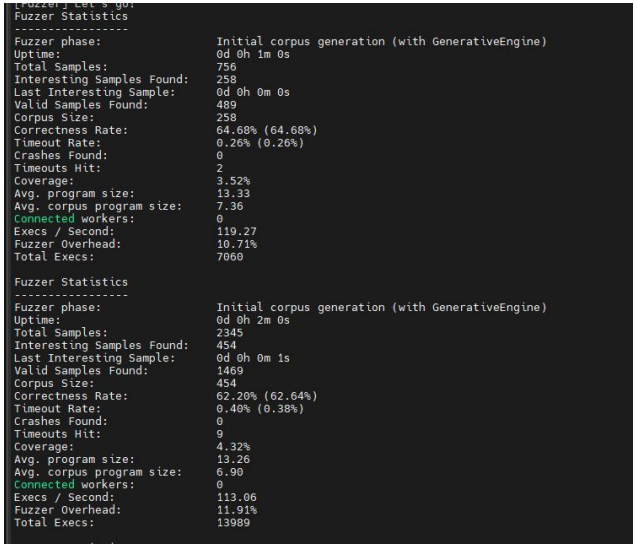
Fuzzing chrome V8:

- gn gen out/fuzzbuild --args='is_debug=false dcheck_always_on=true v8_static_library=true v8_enable_verify_heap=true v8_fuzzilli=true sanitizer_coverage_flags="trace-pc-guard" target_cpu="x64"'
- ninja -C ./out/fuzzbuild d8

Bonus: Fuzzing with Fuzzilli

Fuzzing chrome V8:

- swift run FuzzilliCli --profile=v8 --storagePath=./out/Storage /home/source/ChromeV8/v8/out/fuzzbuild/d8



OFFBYONE

Bonus: Fuzzing with Fuzzilli

Fuzzing chrome V8:

Starting a tool without knowledge about it or the target is a waste of time and energy. You should take some time reading the documentation of fuzzilli to scale it properly. You could use multiple machine for example. Also, compiling the target (v8 or other) yourself allows you to:

- Remove part you have no interest in (better speed exec)
- Focus on certain type of bug (removing sanitizer you have no interest in)
- etc.

In conclusion: Don’t be lazy. Browser and V8 fuzzing are definitely a thing, you could find interesting (and valuable) bugs there, however if you run the same tool as everyone on a target that is heavily tested, chance are that you find nothing.

Bug analysis and triage

Real-world workflow for bug hunting

Usually, I try to find a target that already possesses a bug bounty program or has been covered by some vendors like ZDI. Once I identify a software I have a minimum interest in, I start by writing or using the most simplest fuzzer possible. The goal isn't necessarily to achieve results but simply gain a basic understanding of how to fuzz this target. This achieved, I start reading documentation; people skip this way too often and dive into the code. One advice: stick to the documentation the longest you can, because once you can't read documentation but need to read the code to understand a feature, that's when the hard work starts.

At this stage, I might tweak the fuzzer a bit, change the corpus, add some extensions, write a simple harness to focus on a specific part of the target. Then it's time to read the codebase and actually gain knowledge about the target software, how it works, what its dependencies are, what the threat model is, etc. Finally, you are ready to write an efficient fuzzer; you know the target, its internals, you might even have already smelled something about this or that part of the code. You can write or use the final fuzzer.

After that, it's a matter of optimization. How to achieve better persistence, is snapshot fuzzing possible, can you improve the corpus, can you get rid of certain parts of the target that you have only small interest in, etc. Then, it's time to wait and hope. Usually, after a few dozen billion executions, I know I won't find anything there, at least with the method I use, and decide to move on.

Conclusions

Pros:

- Fuzzing is fun.
- It's automated.
- It can uncover very weird cases.
- It's easy to reproduce bug.

Cons:

- It's limited.
- It's not that automated.
- It can miss big part of problems.
- Sometimes reproducing is related to environmental factor and make the fuzzer life tricky

Like everything in life for each good side, there is a downside. Fuzzing isn't perfect, and it's not meant to be. It allow the researcher to automate a big part of bug hunting, but it won't spare you to do your homework and actually study your target in depth.

Conclusions

My 2 cents: Don't EVER fall in love with the technique. It's the goal that matters, the technique, as elegant as it can be, is only the tool that help you reach your goal. Finding bugs isn't about writing the most sophisticated and elegant tool, but simply finding meaningful bugs.

Thanks for watching :-)

The giants

We forgot too often to mention people that inspired us, here is **a non-exhaustive** list of interesting people you can follow: @corelanc0d3r @ifratic @richinseattle @chompie1337 @gamozolabs @Nosoynadiemas @taviso @Void_Sec @ReneFreingrube @hackerschoice @0vercl0k @alisaesage

Going further

Trainings:

- Fuzzing.io: http://fuzzing.io/

- Corelan: https://www.corelan-training.com/

Going further

Books:

- Fuzzing: Brute-force vulnerability discovery, by Michael Sutton, Adam Greene, Pedram Amini

Fuzzing Against the Machine: Automate vulnerability research with emulated IoT devices on QEMU

by Antonio Nappa, Eduardo Blázquez

- The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities (Volume 1 of 2) 1st Edition

by Mark Dowd, John McDonald, Justin Schuh

Going further

Articles:

- h0mbre - Fuzzer development series - https://h0mbre.github.io/New_Fuzzer_Project/

- Finding 0days with Jackalope - https://www.trellix.com/blogs/research/finding-0-days-with-jackalope/

- Modern harnessing - https://blog.haboob.sa/blog/modern-harnessing-meets-in-memory-fuzzing

- Fuzzing closed source applications - https://def.camp/wp-content/uploads/dc2017/Day%201_Rene_Fuzzing_closed_source_applications_DefCamp.pdf

- Un-bee-lievable Performance: Fast Coverage-guided Fuzzing with Honeybee and Intel Processor Trace - https://blog.trailofbits.com/2021/03/19/un-bee-lievable-performance-fast-coverage-guided-fuzzing-with-honeybee-and-intel-processor-trace/

- Jaanus Kääp - Building Windows Kernel fuzzer - https://www.youtube.com/watch?v=mpXQvto4Vy4

Going further

Articles:

- Ke Liu - Dig Into the Attack Surface of PDF and Gain 100+ CVEs in 1 Year - https://www.blackhat.com/docs/asia-17/materials/asia-17-Liu-Dig-Into-The-Attack-Surface-Of-PDF-And-Gain-100-CVEs-In-1-Year-wp.pdf

- Antonio Morales Fuzzing 101 - https://github.com/antonio-morales/Fuzzing101

- Mateusz Jurczyk - Effective file format fuzzing - https://www.blackhat.com/docs/eu-16/materials/eu-16-Jurczyk-Effective-File-Format-Fuzzing-Thoughts-Techniques-And-Results.pdf

- Bruno Oliveira - Coverage Guided Fuzzing – Extending Instrumentation to Hunt Down Bugs Faster! - https://blog.includesecurity.com/2024/04/coverage-guided-fuzzing-extending-instrumentation/

- A new Solid attack surface against Acrobat and Foxit Editor https://blog.haboob.sa/blog/a-new-solid-attack-surface-against-acrobat-and-foxit-editor

Going further

Tools:

- AFL: https://afl-1.readthedocs.io/en/latest/
- AFL++: https://aflplus.plus/
- LibFuzzer: https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md
- Honggfuzz: https://github.com/google/honggfuzz
- Fuzzilli: https://github.com/googleprojectzero/fuzzilli
- Dharma: https://blog.mozilla.org/security/2015/06/29/dharma/
- Domato: https://github.com/googleprojectzero/domato
- WinAFL: https://github.com/googleprojectzero/winafl
- Jackalope: https://github.com/googleprojectzero/Jackalope
- WTF: https://github.com/0vercl0k/wtf
- Syzkaller: https://github.com/google/syzkaller
- Echidna: https://github.com/crytic/echidna

OFF BY ONE

Going further

Tools:

- Fuzzing corpus - https://www.powerofcommunity.net/poc2018/jaanus.pdf
- Windows fuzzing harness template - https://github.com/australeo/windows_fuzzing_harness/blob/main/harness.cpp
- Fuzzowski - https://github.com/nccgroup/fuzzowski
- AFLnet - https://github.com/aflnet/aflnet
- Rewind - https://github.com/quarkslab/rewind
- BooFuzz - https://github.com/jtpereyda/boofuzz
- BugID - https://github.com/SkyLined/BugId