Fuzzing Introduction

ESIEA edition

Patrick Ventuzelo (@Pat_Ventuzelo)

- Founder & CEO of **FuzzingLabs** | Senior **Security Researcher**
  - Twitter / LinkedIn / Github / Blog
  - Fuzzing and vulnerability research
  - Development of security tools

- Training/Online courses
  - **Rust** Security Audit & Fuzzing
  - **Go** Security Audit & Fuzzing
  - **WebAssembly** Reversing & Analysis
  - Practical Web **Browser** Fuzzing

- Main focus
  - **Fuzzing**, Vulnerability research
  - Rust, Golang, **WebAssembly**, **Browsers**
  - Blockchain Security, Smart contracts

- Previous speaker at:
  - BlackHat US, OffensiveCon, REcon, RingZer0, ToorCon, hack.lu, NorthSec, etc.

What is Fuzzinglabs?

- **Training** in conferences
  - Practical Web Browser Fuzzing - link
    - (OffensiveCon, Recon, RingZer0, POC)
- **Online courses**
  - Rust Security Audit and Fuzzing - link
  - Go Security Audit and Fuzzing - link
  - C/C++ Whitebox Fuzzing - link
  - WebAssembly Reversing and Dynamic Analysis - link

- **Continuous fuzzing and audit**
  - Long-term contract (>6 months)
  - Implement fuzzing for client's codebases
  - Report bugs
- **Custom security tools on-demand**
  - Most of the time open-source tools
  - Fuzzers, static analyzers, disassembler, etc.
- **Dedicated Fuzzers**
  - Browsers
  - Telecom (5G, 4G, VoLTE, etc.)
  - Blockchains
- **Research**

We are recruiting ;)

Summary

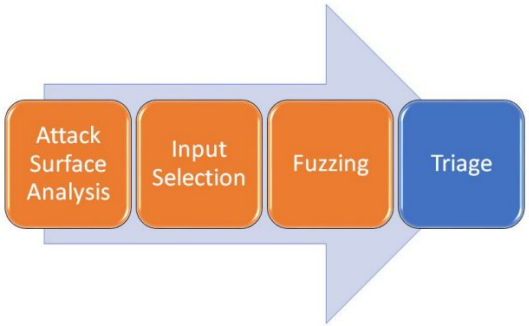
- Introduction to Fuzzing
- Blackbox Fuzzing
  - Dumb fuzzing / Smart fuzzing
  - Mutation-based fuzzing
- Coverage-guided Fuzzing
  - AFL++ / Honggfuzz
- Fuzzing Workflow
  - Corpus / Inputs selection / Code coverage / Corpus minimization
- Crashes Triaging
  - Bucketing / Crashes minimization / Debugging / Root cause analysis

Introduction to Fuzzing

What’s fuzzing?

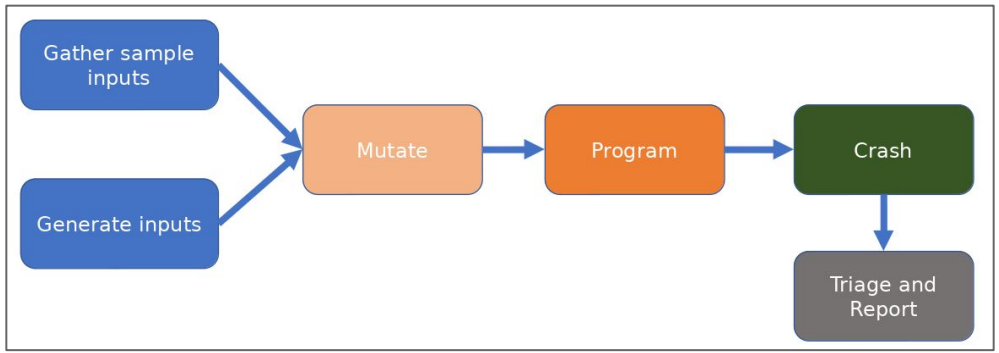
- Fuzzing or fuzz testing is an **automated software testing technique** that involves providing invalid, unexpected, or **random data as inputs** to a computer program. The program is then **monitored for exceptions** such as crashes, failing built-in code assertions, or for finding potential memory leaks and other unexpected behaviours.

- Different fuzzing approaches:
  - Black box:
    - You don’t have any real knowledge of the target
    - You don’t have access to the source code
    - You are not able to recompile the target.
  - Gray box:
    - You have some knowledge of the target
    - You are not able to recompile the target.
  - White box:
    - You have access to the source code
    - You can recompile the target.



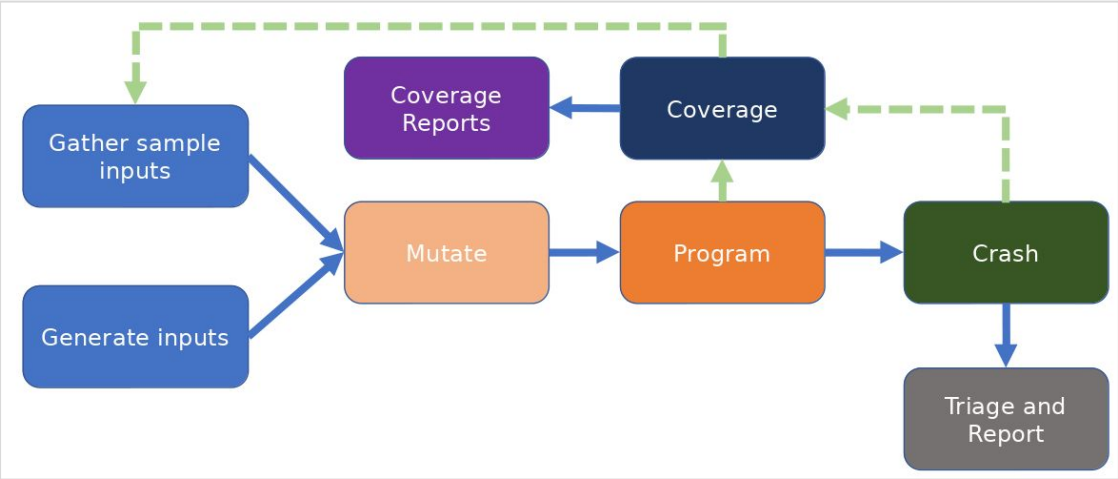
Fuzzing techniques #1 - Really basic technique

- **Dumb** fuzzing
  - Input data is corrupted randomly without awareness of expected format.
- **Smart** fuzzing
  - Input data is corrupted with awareness of the expected format, such as encodings, relations (offset, checksum, etc).
- **Mutation-based** fuzzing
  - Modification of known-valid input data is made according to certain patterns.



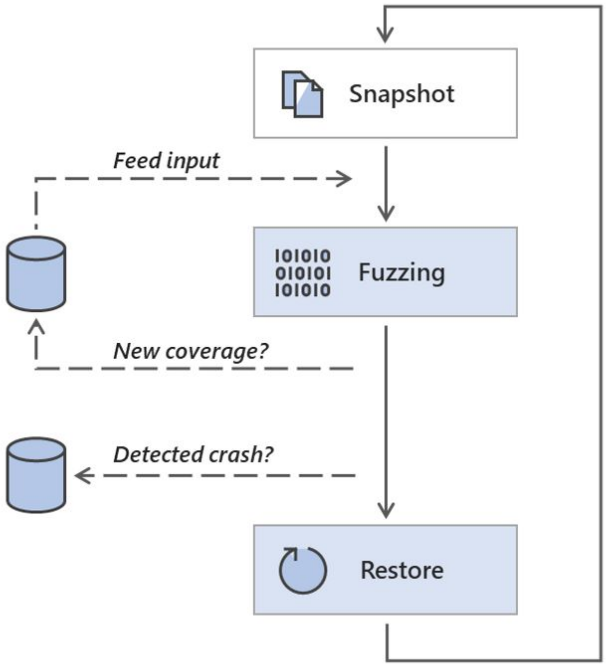
Fuzzing techniques #2 - Most common technique

- **Feedback-driven / Coverage-guided** fuzzing
  - Observe how inputs are processed to learn which mutations are interesting.
  - Save those inputs to be re-used in future iterations.



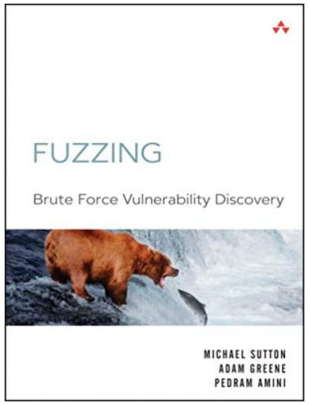
Fuzzing techniques #3 - Advanced technique

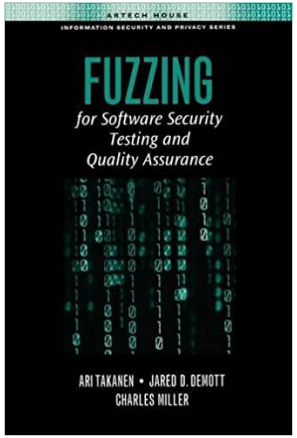
- **In-Process/In-memory/Persistent** fuzzing
  - Target and fuzz a specific function entry point of the program in only one process i.e., for every test case the process isn't restarted but the values are changed in memory.
- **Generation-based** fuzzing
  - Generate semi-well-formed inputs from scratch, based on knowledge of file format or protocol.
- **Differential** fuzzing
  - Observe if two program implementations/variants produce different outputs for the same input.
- **Snapshot** fuzzing
  - In-Process fuzzing with previous memory/register state restored for each fuzz case

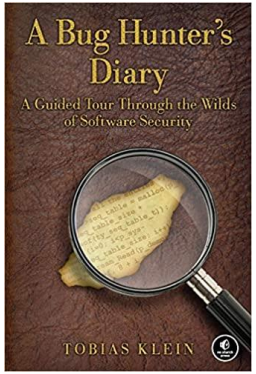


Fuzzing Books

- **Fuzzing: Brute Force Vulnerability Discovery**
  - A bit old but still a really good introduction to fuzzing
  - Prefered chapter: **7, 11, 14, 17, 19, 20 and 23**
  - paper: amazon.com
- **Fuzzing for Software Security Testing and Quality Assurance**
  - A bit old but still a good resource about advanced fuzzing concepts
  - Prefered chapter: **4, 5, 6 and 9**
  - paper: 1st edition / 2nd edition
- **A Bug Hunter's Diary**
  - **Not just about fuzzing but definitely my favorite security book**
  - Prefered chapter: ALL of them
  - paper: amazon.com
- **The Fuzzing Book:**
  - Tools and Techniques for Generating Software Tests
  - Complete interactive book to learn fuzzing concepts
  - FREE - online







Further readings

- Lightning in a Bottle - 25 Years of Fuzzing (FuzzCon 2020) - link
- Fuzzing (and bug hunting) ressources - link
- The Art of Fuzzing - slides, demos
- The Art, Science, and Engineering of Fuzzing: A Survey - paper
- Fuzzing 101 by @metzman - link
- What the Fuzz - slides, video
- Adventures in Fuzzing by @gamozolab - slides, video
- Google Fuzzing Forum (tuto, examples, etc.) - link
- MotherFuzzers/meetups: Materials from Fuzzing Bay Area meetups - link

- **Fuzzing Like A Caveman** - part #1
  - part #2 - Improving Performance
  - part #3 - Trying to Somewhat Understand The Importance Code Coverage
  - part #4 - Snapshot/Code Coverage Fuzzer!
- **Build simple fuzzer** - part#1, part#2, part#3, part#4
- **Resmack: Grammar Fuzzing Thoughts** - part #1, part#2

Blackbox Fuzzing

Build a Simple Fuzzer

Target: lci

- lci - a LOLCODE interpreter written in C - github
  - Multiple components: Parser, Interpreter
  - Selected commit
    - 6762b724361a4fb471345961b4750657783aeb3b
  - LOLCODE: An esoteric programming language
    - website, Language specification

- Look at the documentation
  - Really simple cli tool
  - Usage:
    - `./lci FILE`

```
HAI 1.2
  CAN HAS STDIO?
  VISIBLE "HAI WORLD!!!1!"
KTHXBYE
```

- Prepare the target
  - `cd intro fuzzing training`
  - `git clone https://github.com/justinmeza/lci`
  - `cd lci && git checkout` 6762b724361a4fb471345961b4750657783aeb3b

LABS: Play with lci

1. **Compile the target**
   - `cd lci`
   - `cmake . && make`

```
[ 12%] Building C object CMakeFiles/lci.dir/interpreter.c.o
[ 25%] Building C object CMakeFiles/lci.dir/lexer.c.o
[ 37%] Building C object CMakeFiles/lci.dir/main.c.o
[ 50%] Building C object CMakeFiles/lci.dir/parser.c.o
[ 62%] Building C object CMakeFiles/lci.dir/tokenizer.c.o
[ 75%] Building C object CMakeFiles/lci.dir/unicode.c.o
[ 87%] Building C object CMakeFiles/lci.dir/error.c.o
[100%] Linking C executable lci
```

2. Run the target with **valid** lolcode script
   - `./lci ./test/1.3-Tests/4-Output/5-BasicStrings/test.lol`
   - Check execution error code
     - `echo $?`
     - `0`

```
HAI 1.3
    VISIBLE "Lorem " "ipsum " "dolor " "sit"
KTHXBYE
```

3. Run the target with **invalid** lolcode
   - `./lci README.md`
   - Check execution error code
     - `echo $?`
     - `200`

```
HAI 1.2
  CAN HAS STDIO?
  VISIBLE "HAI WORLD!!!1!"
KTHXBYE
```

LABS: Create a Python script to execute lci

- Create a buffer
  - `bytearray`
- Create temporary file
  - `open / write`
- Execute lci binary from Python
  - `subprocess.Popen`
- Wait for the return code
  - `subprocess.wait()`
- Analyze and print the return code
  - `print`

- **Solution:**
  - `first_fuzzer/fuzz.py`

(SOLUTION NEXT SLIDE)

LABS: Create a Python script to execute lci

- Create a buffer
  - `bytearray`
- Create temporary file
  - `open / write`
- Execute lci binary from Python
  - `subprocess.Popen`
- Wait for the return code
  - `subprocess.wait()`
- Analyze and print the return code
  - `print`

- **Solution:**
  - `first_fuzzer/fuzz.py`

```
→  first_fuzzer python3 fuzz.py
OK - Exited with 0
```

```python
def execute(inp: bytearray):

    # Write out the input to a temporary file
    tmpfn = f"tmpinput.lol"
    with open(tmpfn, "wb") as fd:
        fd.write(inp)

    # Run lci until completion
    sp = subprocess.Popen(
        ["../lci/lci", tmpfn],
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL)
    ret = sp.wait()
    # print(ret)  # prints return code

    # Assert that the program ran successfully
    # crashes use negative numbers
    if ret < 0:
        print(f"CRASH - Exited with {ret}")
        sys.exit()
    elif ret != 0:
        print(f"ERR - Exited with {ret}")
    else:
        print(f"OK - Exited with {ret}")
```

```python
inp = b"""
HAI 1.3
VISIBLE "Lorem " "ipsum " "dolor " "sit"
KTHXBYE
"""

inp = bytearray(inp)
execute(bytearray(inp))
```

Blackbox Fuzzing

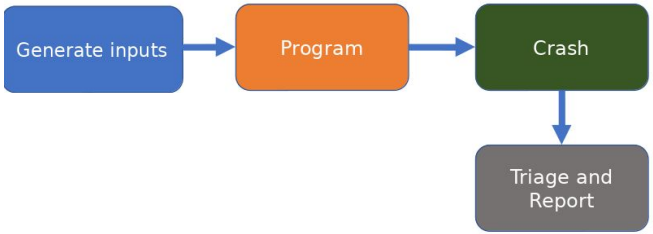
Dumb & Smart fuzzing

Dumb & Smart fuzzing

- Dumb fuzzing
  - A fuzzer that generates **completely random input** is known as a “dumb” fuzzer, as it has **no built-in intelligence** about the program it’s fuzzing. A dumb fuzzer requires the smallest amount of work to produce results. Input data is **completely random** or randomly corrupted **without awareness of expected format.**
  - Could be as simplistic as piping /dev/random into a program

- Smart fuzzing
  - Smart fuzzers are programmed with **knowledge of the input format**, i.e. a protocol definition or rules for a file format. It can then construct mostly valid input and only fuzz parts of the input within that basic format. **Input data is corrupted with awareness of the expected format.**

- Further reading:
  - How Stupid is Dumb Fuzzing? - link
  - Dumb and Smart Fuzzing - link
  - The Smart Fuzzer Revolution - link
  - Fuzzing: Breaking Things with Random Inputs - link
  - Generate Random Strings and Passwords in Python - link
  - Random string generation with upper case letters and digits - link



LABS: Fuzz lci with pure **random data**

- Generate random string length
  - `random.randint`
- Generate random string
  - `random.choices`
  - `string.ascii_letters`
- **Solution:**
  - `dumb_fuzzing/fuzz.py`

(SOLUTION NEXT SLIDE)

LABS: Fuzz lci with pure **random data**

- Generate random string length
  - `random.randint`

```python
# Generate random length
inp_len = random.randint(2, 20)
```

- Generate random string
  - `random.choices`
  - `string.ascii_letters`

```python
# Generate random string
inp = ''.join(
    random.choices(string.ascii_letters + string.digits, k=inp_len))
```

- **Solution:**
  - `dumb_fuzzing/fuzz.py`

```
ERR - Exited with 46
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 46
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 154
ERR - Exited with 46
ERR - Exited with 154
ERR - Exited with 154
```

LABS: Fuzz lci with **mutated/corrupted** lol script

- Create a buffer with valid data
  - Use existing valid tests
- Mutate buffer bytes
  - `random.int`
- **Solution:**
  - `smart_fuzzing/fuzz.py`

(SOLUTION NEXT SLIDE)

LABS: Fuzz lci with **mutated/corrupted** lol script

- Create a buffer with valid data
  - Use existing valid tests
- Mutate buffer bytes
  - `random.int`
- **Solution:**
  - `smart_fuzzing/fuzz.py`

```
ERR - Exited with 46
ERR - Exited with 202
ERR - Exited with 46
ERR - Exited with 46
ERR - Exited with 46
ERR - Exited with 202
ERR - Exited with 46
ERR - Exited with 202
```

```python
while True:

    # Reset the input each time
    inp = b"""
HAI 1.3
    VISIBLE "Lorem " "ipsum " "dolor " "sit"
KTHXBYE
"""

    inp = bytearray(inp)

    # Mutate my input
    for _ in range(random.randint(1, 10)):
        inp[random.randint(0, len(inp) - 1)] = random.randint(0, 255)

    # print(inp)

    # Execute the input to the target
    execute(inp)
```

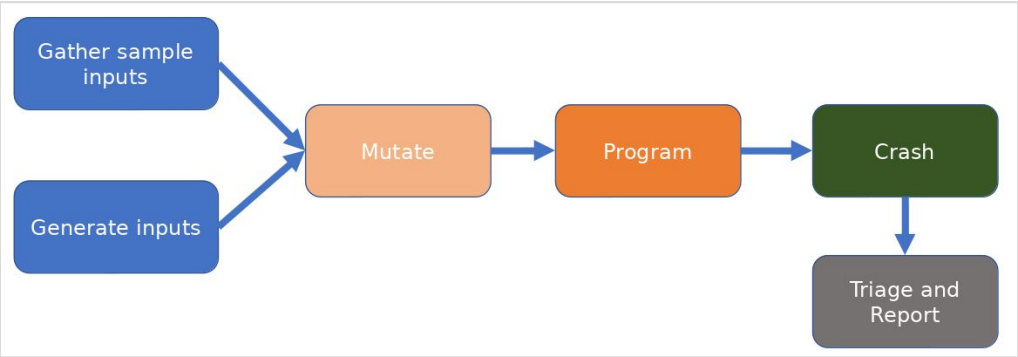
Blackbox Fuzzing

Mutation-based fuzzing

Mutation-based fuzzing

- Mutation-based fuzzers are arguably one of the easier types to create. This technique suites dumb fuzzing but can be used with more intelligent fuzzers as well. With mutation, **samples of valid input are mutated randomly to produce malformed input.**

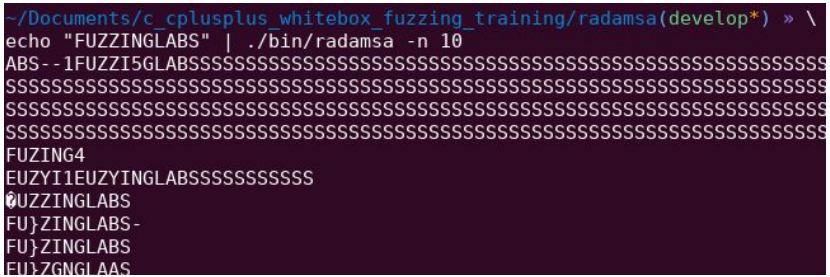
- Further readings:
  - Mutation-Based Fuzzing - link
  - MOpt: Optimized Mutation Scheduling for Fuzzers - link



Radamsa: The most famous mutation-only fuzzer

- Radamsa - gitlab
  - Radamsa is a **test case generator for robustness testing**, a.k.a. a fuzzer. It is typically used to test how well a program can withstand malformed and potentially malicious inputs.
  - **Really simple, easy to use fuzzer** implementing some really efficient mutation algorithms.
  - Clearly, **this tool helped making fuzzing more mainstream.**
  - Installation
    - `git clone https://gitlab.com/akihe/radamsa`
    - `cd radamsa; make`

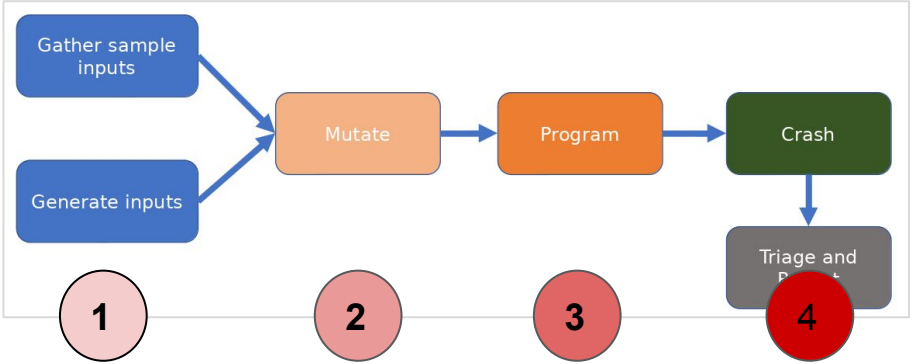
- Run radamsa
  - `echo "FUZZINGLABS" | ./bin/radamsa`



- Further readings:
  - Fuzzing with radamsa - Short - video
  - Fuzzing a simple C program with Radamsa - video
  - **Fuzzing with Radamsa and some thoughts about coverage - link**
  - How To Install and Use Radamsa to Fuzz Test Programs and Network Services on Ubuntu 18.04 - link
  - Radamsa Fuzzer Tutorial | Install and Use on Linux Ubuntu - video
  - pyradamsa: Python bindings for radamsa fuzzing library. - link

EXAMPLE: Fuzzing gzip with Radamsa

1. Create an input file
   - `gzip -c /bin/bash > sample.gz`
2. Mutate this input with radamsa
   - `radamsa sample.gz > fuzzed.gz`
3. Execute the target program
   - `gzip -dc fuzzed.gz > /dev/null`
4. Check the program's return code
   - `echo $?`
5. **Automate the process**
   - Create a simple bash script ⇒




```
gzip -c /bin/bash > sample.gz
while true
 do
   ./bin/radamsa sample.gz > fuzzed.gz
   gzip -dc fuzzed.gz > /dev/null
   test $? -gt 127 && break
 done
```

LABS: Add corpus and Radamsa mutation

- Find all lol script tests and copy them into the corpus folder
  - `find ../lci/test/ -iname '*.lol' -exec md5sum '{}' ';' | while read sum file ; do cp "$file" "corpus/$sum".lol ; done`
- Read corpus folder and pick one sample randomly
  - `glob.glob`
  - `open / read`
  - `random.choice`
- Mutate input with Radamsa (pyradamsa)
  - `pyradamsa.Radamsa()`
  - `rad.fuzz()`

(SOLUTION NEXT SLIDE)

LABS: Add corpus and Radamsa mutation

- Find all lol script tests and copy them into the corpus folder
  - `find ../lci/test/ -iname '*.lol' -exec md5sum '{}' ';' | while read sum file ; do cp "$file" "corpus/$sum".lol ; done`

- Read corpus folder and pick one sample randomly
  - `glob.glob`
  - `open / read`
  - `random.choice`

```
corpus_filenames = glob.glob("corpus/*")  # glob is better b/c full paths
print(corpus_filenames)

# Load the corpus files into memory
corpus = set()  # using set to get rid of aliases/symlinks/dups
for filename in corpus_filenames:
    corpus.add(open(filename, "rb").read())

# Convert the corpus back into a list as we're done with the set for
# deduping inputs which were not unique
corpus = list(map(bytearray, corpus))  # bytearray for in-place mutations
```

- Mutate input with Radamsa (pyradamsa)
  - `pyradamsa.Radamsa()`
  - `rad.fuzz()`

```
rad = pyradamsa.Radamsa()
```

```
# Mutate my input by radamsa
fuzzed = rad.fuzz(inp)
```

- **Solution:**
  - `mutation fuzzing/fuzz.py`
    - `pip3 install pyradamsa`

```
ERR - Exited with 46
ERR - Exited with 154
ERR - Exited with 46
ERR - Exited with 156
ERR - Exited with 24
ERR - Exited with 46
ERR - Exited with 154
CRASH - Exited with -11
```

Blackbox Fuzzing

Improve your fuzzers

How to improve your fuzzers ?

- Some basic fuzzer features
  - Fast random PRNG generation
  - Custom Mutation algorithm
  - Performance measurement / optimization
- Advanced fuzzer features
  - Multithreading
  - Code coverage measurement
  - Crash report generation
  - Snapshot fuzzing
- Further readings
  - Fuzzing Like A Caveman - part #1
    - part #2 - Improving Performance
    - part #3 - Trying to Somewhat Understand The Importance Code Coverage
    - part #4 - Snapshot/Code Coverage Fuzzer!
  - Build simple fuzzer - part#1, part#2, part#3, part#4
  - Resmack: Grammar Fuzzing Thoughts - part #1, part#2

LABS: Improve your fuzzer with extra features

- Add performance statistic (fuzz case/sec)
  - `time.time()`

- Add timeout mechanism
  - `threading.Timer`

- **Solution:**
  - `improve_fuzzing/fuzz.py`

(SOLUTION NEXT SLIDE)

LABS: Improve your fuzzer with extra features

- Add performance statistic (fuzz case/sec)
  - `time.time()`

```python
# Get the time at the start of the fuzzer
start = time.time()

# Total number of fuzz cases
cases = 0
```

```python
# Update number of fuzz cases
cases += 1

# determine the amount of seconds we have been fuzzing for
elapsed = time.time() - start

# determine the number of fuzz cases per second
fcps = float(cases) / elapsed

print(f"[{elapsed:10.4f}] cases {cases:10} | fcps {fcps:10.4f}")
```

- Add timeout mechanism
  - `threading.Timer`

- **Solution:**
  - `improve_fuzzing/fuzz.py`

```
[    0.7553] cases        189 | fcps   250.2306
ERR - Exited with 46
[    0.7581] cases        190 | fcps   250.6184
CRASH - Exited with -11
```

```python
sp = subprocess.Popen(["../lci/lci", tmpfn],
                      stdout=subprocess.DEVNULL,
                      stderr=subprocess.DEVNULL)

kill = lambda process: process.kill()
my_timer = Timer(1, kill, [sp])
try:
    my_timer.start()
    ret = sp.wait()
finally:
    my_timer.cancel()
```

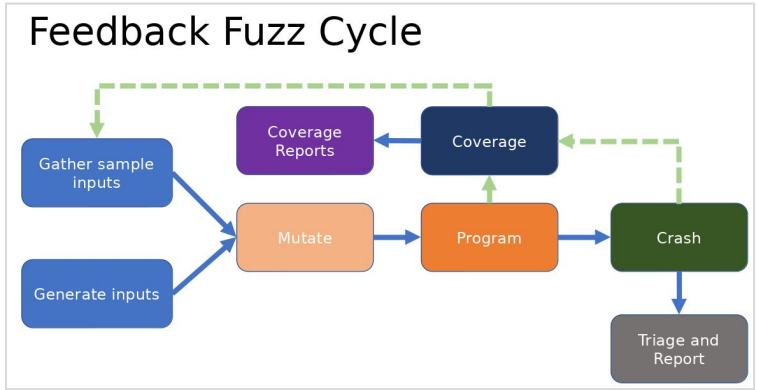
Coverage-guided Fuzzing

What's feedback-driven/coverage-guided fuzzing?

- Coverage guided fuzzing uses **program instrumentation to trace the code coverage reached** by each input fed to a fuzz target. Fuzzing engines use this information to make informed decisions about **which inputs** to mutate to **maximize coverage**.

- Coverage guided fuzzing is recommended when target is **self-contained**, **deterministic** and **can be executed really fast** (don't need slow initialisation).

- Feedback can be retrieve at different level:
  - Hardware-based (CPU) coverage-feedback
    - BTS (Branch Trace Store)
    - Intel PT (Processor Tracing)
  - Compile-time instrumentation (clang, gcc, etc.)
    - Function, Edge, Branch, Basic block level



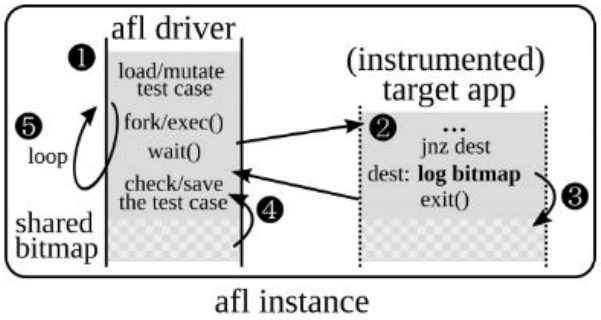
- Further readings:
  - Feedback-driven fuzzing (honggfuzz) - link
  - Study and Comparison of General Purpose Fuzzers - link
  - Full speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing - video

Coverage-guided Fuzzing

AFL / AFL++

American Fuzzy Lop (AFL)

- American fuzzy lop is a security-oriented open source fuzzer
  - Created by Michal Zalewski
  - Official page of the project, Github mirror
- Features
  - **Compile-time instrumentation**
  - Genetic **mutation algorithms**
  - **Generation of test cases** that trigger new internal states
  - Code coverage feedbacks
- In practice:
  - **Compilation**:
    - add instrumentation routines
  - **Execution**:
    - read and mutate inputs (1)
    - launch the instrumented target (2)
    - Execute and record coverage feedbacks (3)
    - Update internal shared bitmap (4)
    - repeat since step (1) or (2)



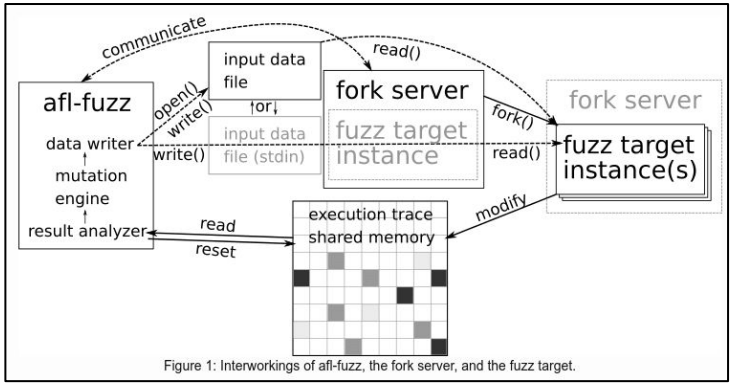
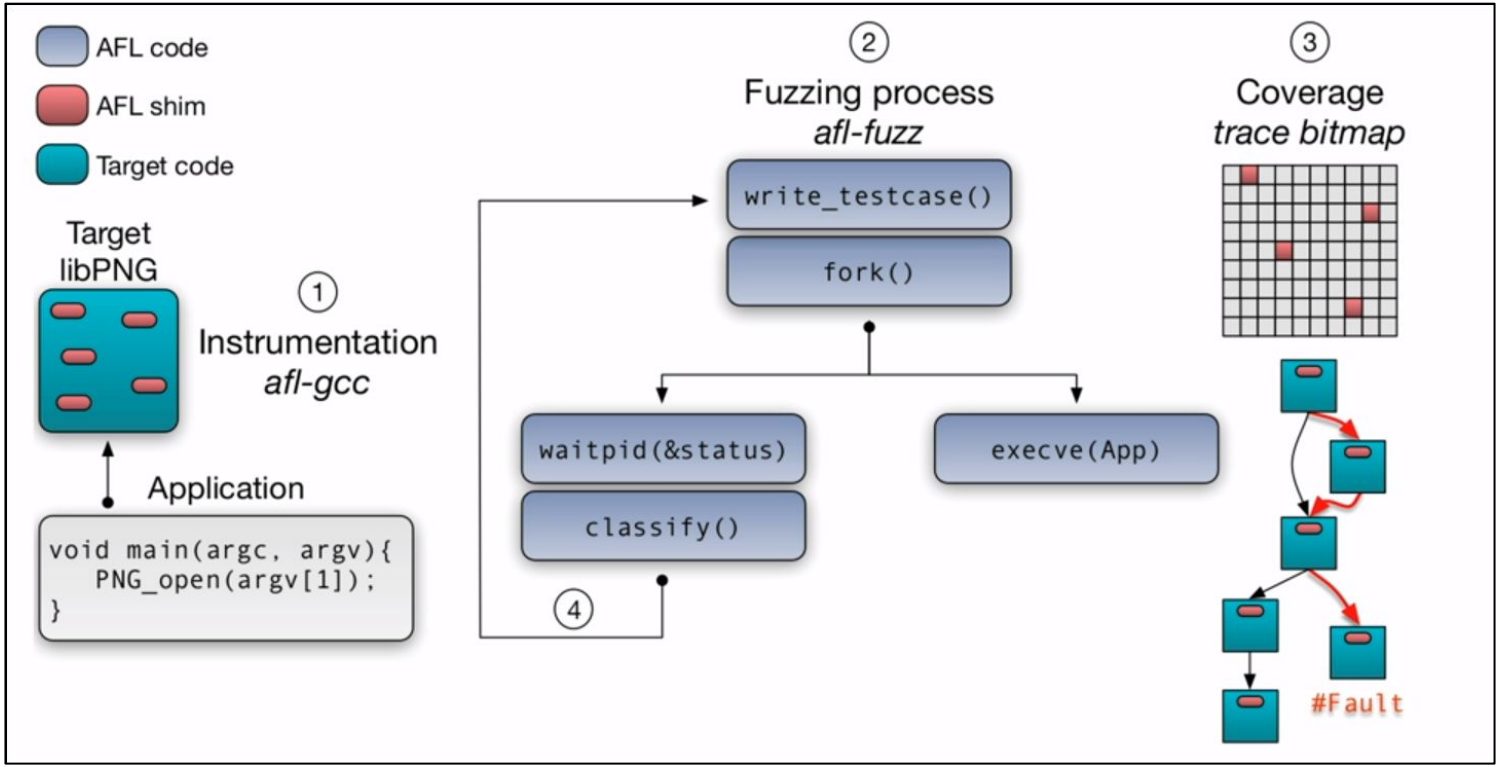


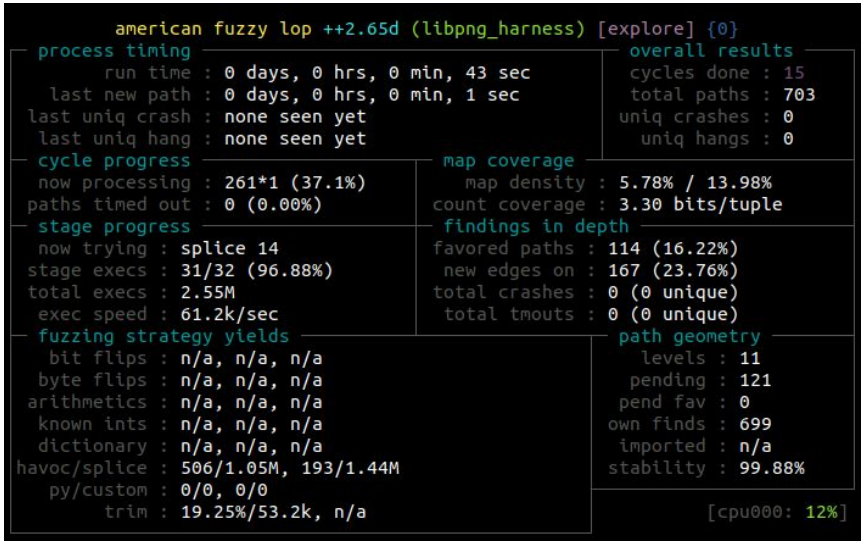
Figure 1: Interworkings of afl-fuzz, the fork server, and the fuzz target.

American Fuzzy Lop (AFL) - internals



American Fuzzy Lop plus plus (AFL++)

- AFL++ is afl with **community patches**, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn_mode, and a lot more!
  - website, github, install
- Improvement :
  - Speed, Mutations, Custom module support, etc.
  - Instrumentations (QEMU, Unicorn, QBDI)
- Further readings:
  - Fuzzing VIM with AFL++ - link
  - Fuzzing with AFL workshop - link
  - Fuzzing 101 workshop - link
  - Zero bugs found? Hold my beer AFL! - slides
  - Fuzzing TcpDump with AFL on Linux - video
  - Fuzzing FFMpeg using AFL on linux - video
  - How to escape from the fuzz (Exiv2) - link
  - Some afl related tools - link
  - Fuzzing grub: part 1 - link
  - Installing AFLPlusplus and fuzzing a simple C program - link



Coverage-guided Fuzzing

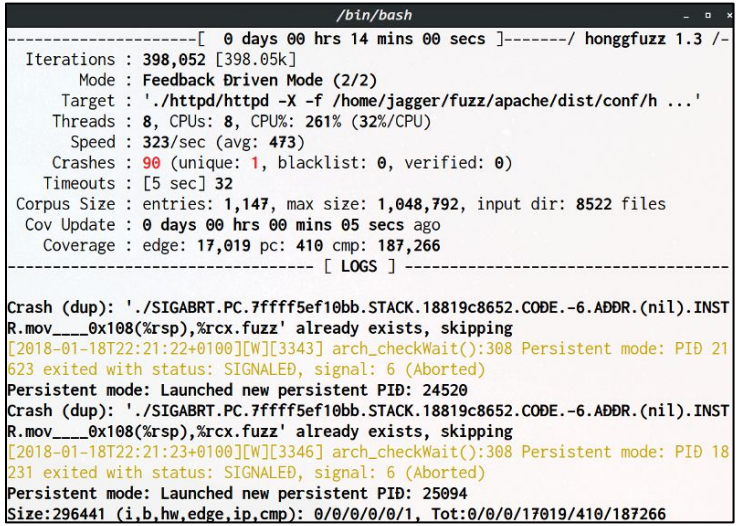
Honggfuzz

Honggfuzz

- Security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options.
  - Created by Robert Swiecki, website, Github
  - Develop by Google and used inside OSS-Fuzz/clusterfuzz

- Features:
  - **Multi-process** and **multi-threaded**
  - Supports several hardware & software-based
    - **feedback-driven fuzzing methods** (Intel PT, ...)
  - Blazingly **fast & Easy-to-use**
  - Works on multiple platforms (**Linux**, **Windows**, **Android**, Mac, ...)
  - Supports the **persistent fuzzing mode**
  - Provides a **corpus minimization** mode.

- Further reading:
  - Double-Free RCE in VLC. A honggfuzz how-to - link
  - Installing honggfuzz and fuzzing simple C program - link
  - Fuzzing tcpdump with honggfuzz - link
  - Honggfuzz:How to build the fuzz environment of openssl - link
  - Fuzzing binaries using Dynamic Instrumentation - link
  - Fuzzing TCP Servers - link



Honggfuzz - Install/tips/options

- Honggfuzz installation
  - `git clone https://github.com/google/honggfuzz`
  - `cd honggfuzz`
  - `make`
- Activate sanitizers at compile time (src)
  - `export HFUZZ_CC_ASAN=true`
  - `export HFUZZ_CC_MSAN=true`
  - `export HFUZZ_CC_UBSAN=false`
- Known issues
  - huge number of HF.sanitizer.log created - link
- Interesting options (help)
  - Use SIGVTALRM to kill timeouting processes (default: use SIGKILL)
    - `--tmout_sigvtalrm | -T`
  - Only generate printable inputs
    - `--only_printable`
  - Verif crashes
    - `--verifier | -V`

```
Usage: ./honggfuzz [options] -- path_to_command [args]
Options:
 --help|-h
       Help plz..
 --input|-i VALUE
       Path to a directory containing initial file corpus
 --output VALUE
       Output data (new dynamic coverage corpus, or the min
 --persistent|-P
       Enable persistent fuzzing (use hfuzz_cc/hfuzz-clang
 --instrument|-z
       *DEFAULT-MODE-BY-DEFAULT* Enable compile-time instru
 --minimize|-M
       Minimize the input corpus. It will most likely delet
 --noinst|-x
       Static mode only, disable any instrumentation (hw/sw
 --keep_output|-Q
       Don't close children's stdin, stdout, stderr; can be
 --timeout|-t VALUE
       Timeout in seconds (default: 10)
 --threads|-n VALUE
       Number of concurrent fuzzing threads (default: numbe
 --stdin_input|-s
       Provide fuzzing input on STDIN, instead of ___FILE__
 --mutations_per_run|-r VALUE
       Maximal number of mutations per one run (default: 6)
 --logfile|-l VALUE
       Log file
 --verbose|-v
       Disable ANSI console; use simple log output
 --verifier|-V
       Enable crashes verifier
```

LABS: Fuzzing lci with Honggfuzz

- lci - a LOLCODE interpreter written in C - github
  - Multiple components: Parser, Interpreter
  - Selected commit
    - 6762b724361a4fb471345961b4750657783aeb3b
  - LOLCODE: An esoteric programming language
    - website, Language specification

- Look at the documentation
  - Really simple cli tool
  - Usage:
    - `./lci FILE`

```
HAI 1.2
  CAN HAS STDIO?
  VISIBLE "HAI WORLD!!!1!"
KTHXBYE
```

- Prepare your target repository
  - `mkdir lci fuzz && cd lci fuzz`
  - `git clone https://github.com/justinmeza/lci`
  - `cd lci && git checkout` 6762b724361a4fb471345961b4750657783aeb3b

(SOLUTION NEXT SLIDE)

LABS: Fuzzing lci with Honggfuzz

- Compile with hfuzz-clang
  - `export CC=/home/fuzzinglabs/Documents/intro fuzzing training/honggfuzz/hfuzz cc/hfuzz-clang`
  - `export CXX=/home/fuzzinglabs/Documents/intro_fuzzing_training/honggfuzz/hfuzz_cc/hfuzz-clang++`
  - **`export HFUZZ_CC_ASAN=true`**
  - `cmake . && make`

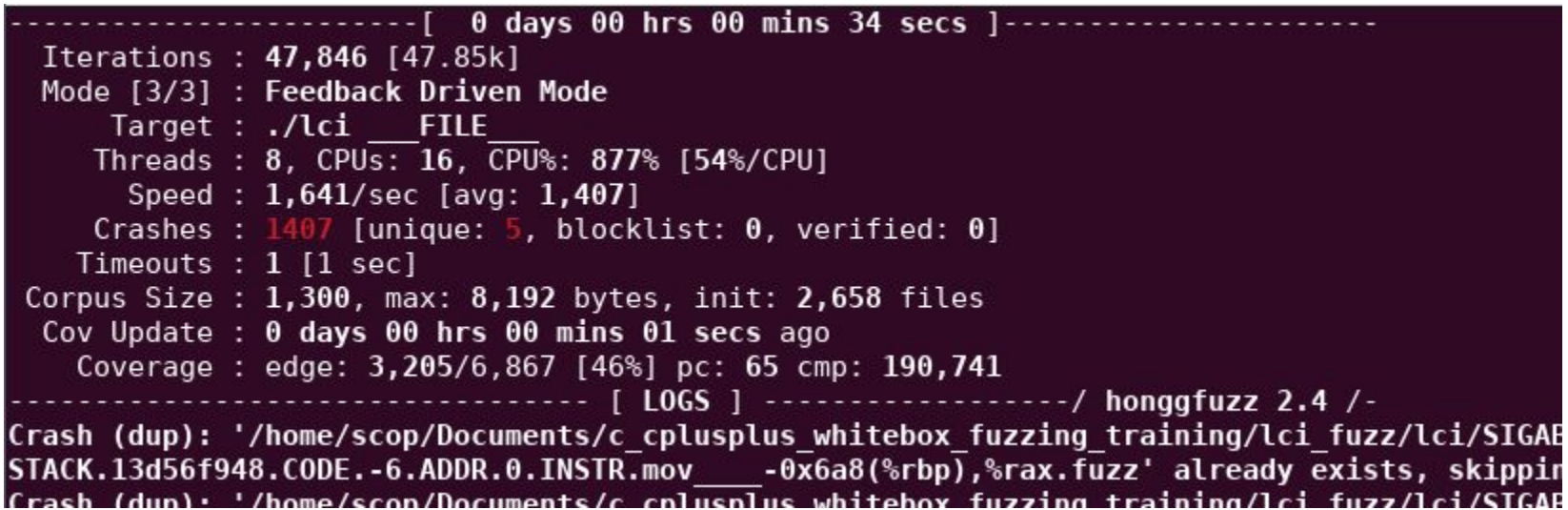
```
000000000040a7d0 T hfuzzInstrumentInit
0000000000449a50 T hfuzz_trace_cmp1
0000000000448040 t hfuzz_trace_cmp1_internal
0000000000449a60 T hfuzz_trace_cmp2
00000000004480b0 t hfuzz_trace_cmp2_internal
0000000000449a70 T hfuzz_trace_cmp4
0000000000448120 t hfuzz_trace_cmp4_internal
0000000000449a80 T hfuzz_trace_cmp8
0000000000448180 t hfuzz_trace_cmp8_internal
00000000004483c0 T hfuzz_trace_pc
```

- **Validate** your compilation
  - honggfuzz symbols: `nm ./lci | grep hfuzz`
  - sanitizers symbols: `nm ./lci | grep asan`

- Prepare fuzzing
  - Create corpora directory
    - `mkdir input`
  - Find all tests and copy them into the input folder
    - `find test/ -iname '*.lol' -exec md5sum '{}' ';' | while read sum file ; do cp "$file" "input/$sum".lol ; done`

- **Start fuzzing**
  - `../../honggfuzz/honggfuzz -i input -- ./lci ___FILE___`

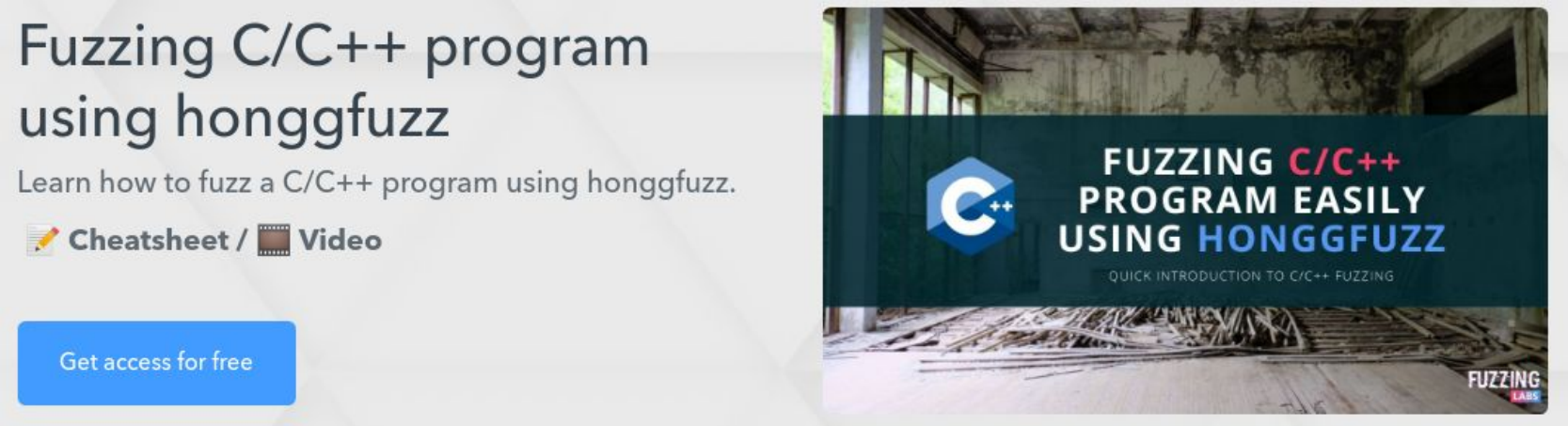
LABS: Fuzzing lci with Honggfuzz



- **TIPS:**
  - Only generate printable input: `--only printable`
  - Enable crashes verifier: `--verifier|-V`
  - Number of concurrent fuzzing threads (default: number of CPUs / 2): `--threads|-n VALUE`

EXTRA: Fuzzing binutils/readelf tutorial

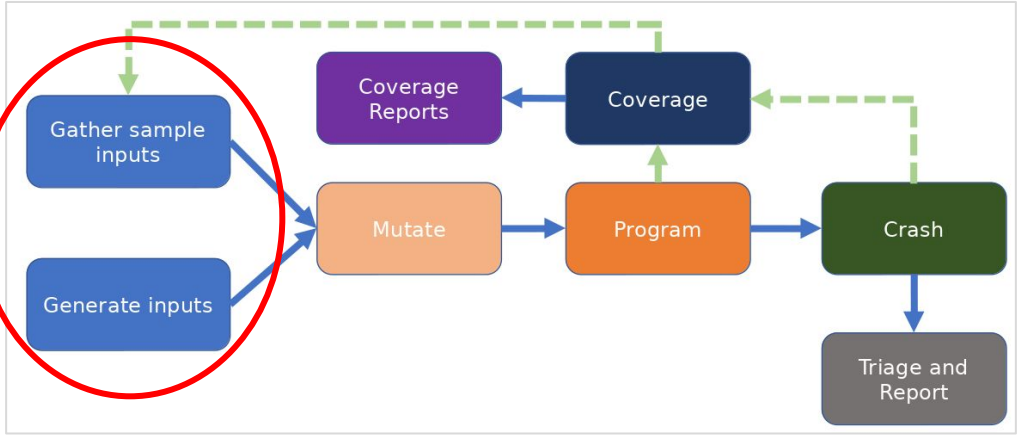
- Target: binutils/readelf
- Link: https://academy.fuzzinglabs.com/fuzzing-c-program-honggfuzz



Fuzzing workflow

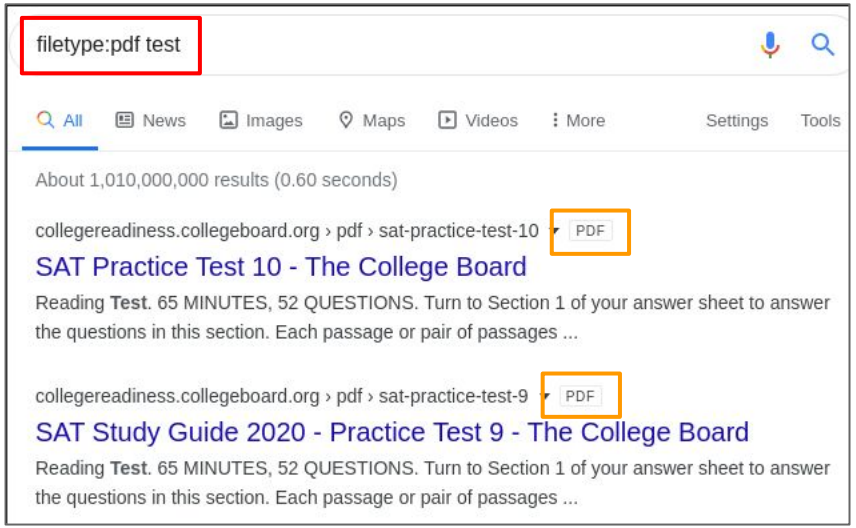
Fuzzing workflow

Corpus/Input Collection



Corpus collection: Gather valid inputs/seeds

- Google dork - Google Search to find public files
  - Work on other search engines as well like Bing
  - **Direct download links**
    - `filetype:pdf test`



- **Webcrawler**
  - Flounder - corpus collector using Bing API - github
  - Scrapy: framework in Python - link

- **Github** is your friend ;)
  - Fuzzing corpora of multiple files formats
    - **Corpus of go-fuzz - link**
    - Fuzzing corpus of multiple file format - link
    - **fuzzdata by Mozilla - link**
    - Corpus of crypto formats - link
    - Seed Corpus for clamav-devel oss-fuzz integration - link
    - small: Smallest possible syntactically valid files of different types
  - (unit)test suite files of big projects

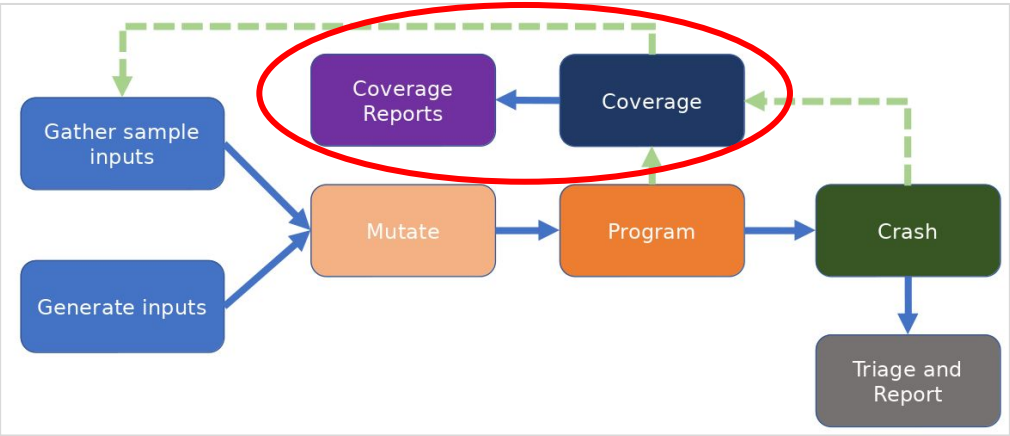
Corpus collection: Generate valid inputs/seeds

- **Generates testcases from scratch**
  - According to some rules or grammar.
  - Convert file to another format
    - ex: ImageMagick convert tool
- Grammarinator - github, slides, paper
  - ANTLRv4 grammar-based test generator
  - Creates test cases according to an input ANTLR v4 grammar.
  - grammars-v4: collection of Antlr4 grammars.
  - Installation:
    - `pip3 install grammarinator`
- Further readings
  - Libfuzzer documentation (corpus) - link
  - Optimizing Seed Selection for Fuzzing - paper
  - Skyfire: Data-Driven Seed Generation for Fuzzing - paper
  - SmartSeed: Smart Seed Generation for Efficient Fuzzing - paper
  - SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing - paper

```
~ » convert -list format
  Format  Module    Mode  Description
---------------------------------------------------------------------
     3FR  DNG       r--   Hasselblad CFV/H3D39II
     AAI* AAI       rw+   AAI Dune image
      AI  PDF       rw-   Adobe Illustrator CS2
     ART* ART       rw-   PFS: 1st Publisher Clip Art
     ARW  DNG       r--   Sony Alpha Raw Image Format
     AVI  MPEG      r--   Microsoft Audio/Visual Interleaved
     AVS* AVS       rw+   AVS X image
     BGR* BGR       rw+   Raw blue, green, and red samples
    BGRA* BGR       rw+   Raw blue, green, red, and alpha samples
    BGRO* BGR       rw+   Raw blue, green, red, and opacity sample
     BIE* JBIG      rw-   Joint Bi-level Image experts Group inte
     BMP* BMP       rw-   Microsoft Windows bitmap image
    BMP2* BMP       -w-   Microsoft Windows bitmap image (V2)
    BMP3* BMP       -w-   Microsoft Windows bitmap image (V3)
     BRF* BRAILLE   -w-   BRF ASCII Braille format
     CAL* CALS      rw-   Continuous Acquisition and Life-cycle Su
          Specified in MIL-R-28002 and MIL-PRF-28002
    CALS* CALS      rw-   Continuous Acquisition and Life-cycle Su
          Specified in MIL-R-28002 and MIL-PRF-28002
  CANVAS* XC        r--   Constant image uniform color
 CAPTION* CAPTION   r--   Caption
     CIN* CIN       rw-   Cineon Image File
     CIP* CIP       -w-   Cisco IP phone image format
    CLIP* CLIP      rw+   Image Clip Mask
    CMYK* CMYK      rw+   Raw cyan, magenta, yellow, and black sa
   CMYKA* CMYK      rw+   Raw cyan, magenta, yellow, black, and a
```

Fuzzing workflow

Code coverage



What’s code coverage?

- Test coverage is a **measure** used to describe the degree to **which the source code** of a program **is executed** when a particular test suite runs. - wikipedia

- **lcov/gcov** - website, man page,
  - **Gcov** is a **source code coverage analysis and statement-by-statement profiling tool.** Generates exact counts of the number of times each statement in a program is executed and annotates source code to add instrumentation - wikipedia**. LCOV** is a graphical front-end for GCC's coverage testing tool gcov. It **collects gcov data** for multiple source files and **creates HTML pages** containing the source code annotated **with coverage information**. - website

- **Kcov** - website, github, installation, video
  - **Code coverage tester for compiled programs**. It allows **collecting code coverage information** from executables w/o special command-line arguments, continuously produces output from applications.

- Further readings:
  - afl-cov: Produce code coverage results with gcov from afl-fuzz test cases - link
  - How to check code coverage on Linux with gcov, lcov and gcovr - video
  - Generating Code Coverage Report Using GNU Gcov & Lcov. - link
  - Coverage testing with gcov - link

EXAMPLE: Code coverage of **Gzip** with Kcov

Coverage Report

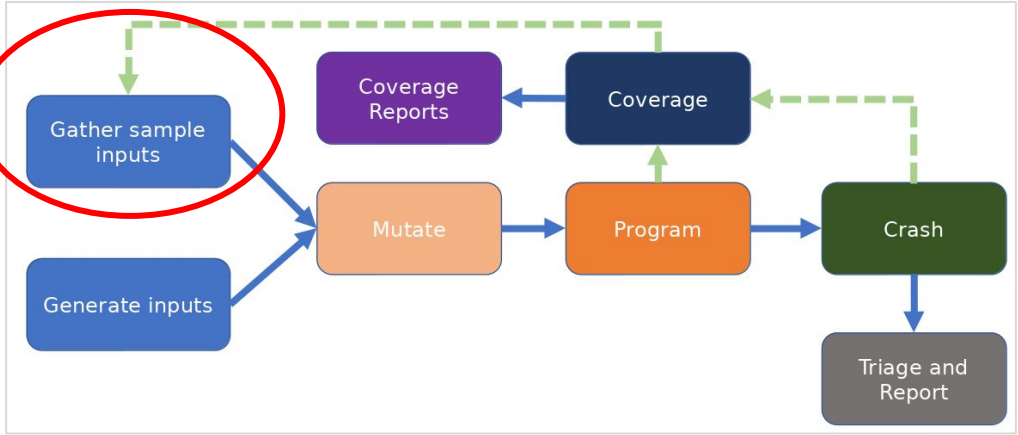
Command:
Date: 2021-03-05 09:47:38
Code covered: 29.0%

Instrumented lines: 3193
Executed lines: 925

Filename	Coverage percent	Covered lines	Uncovered lines	Executable lines
[...]/gzip_fuzz/gzip/unlzh.c	100.0%	106	0	106
[...]/gzip_fuzz/gzip/inflate.c	97.8%	222	5	227
[...]/gzip_fuzz/gzip/unpack.c	96.2%	50	2	52
[...]/gzip_fuzz/gzip/unzip.c	93.8%	60	4	64
[...]/gzip_fuzz/gzip/unlzw.c	85.3%	64	11	75
[...]/gzip_fuzz/gzip/gnulib/lib/basename-lgpl.c	62.5%	5	3	8
[...]/gzip_fuzz/gzip/gnulib/lib/open-safer.c	60.0%	3	2	5
[...]/gzip_fuzz/gzip/gnulib/lib/openat-safer.c	60.0%	3	2	5
[...]/gzip_fuzz/gzip/gnulib/lib/fprintf.c	58.8%	10	7	17
[...]/gzip_fuzz/gzip/lib/xsize.h	50.0%	1	1	2
[...]/gzip_fuzz/gzip/lib/printf-parse.c	45.1%	55	67	122
[...]/gzip_fuzz/gzip/gnulib/lib/fd-safer.c	42.9%	3	4	7
[...]/gzip_fuzz/gzip/util.c	39.0%	60	94	154
[...]/gzip_fuzz/gzip/gzip.c	33.0%	205	417	622

Fuzzing workflow

Corpus minimization

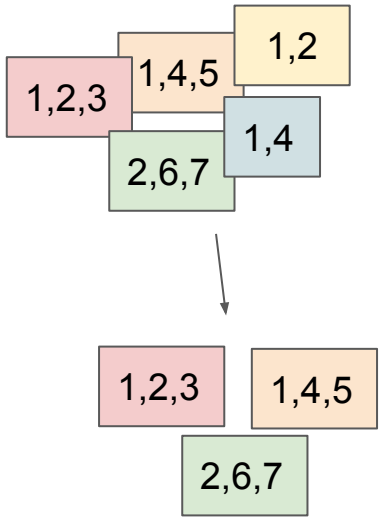


What’s corpus minimization ?

- Corpus minimization tools **tries to find the smallest subset of files** in the input directory **that still trigger the full range of instrumentation data points** seen in the starting corpus i.e. **same code coverage**.
  - Other names: corpus distillation / corpus pruning
  - **CAUTION**: Corpus minimization **doesn't mean modifying/reducing your corpus individual files.**
    - If you are looking for that, take a look to crashes minimization.

- C/C++ fuzzers tools & options:
  - **afl-cmin**
    - Corpus minimization tool for afl-fuzz - original, aflplusplus
    - AFLplusplus: Making the input corpus unique - link
  - **honggfuzz**
    - Minimize the input corpus - link: `--minimize | -M`
  - **libfuzzer**:
    - Done automatically by default at runtime: `-reduce_inputs`

- Further readings:
  - Corpus distillation & fuzzing - link, video
  - afl-kit: Reimplement afl-cmin in python using less memory, less disk space, and faster
  - MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation - paper



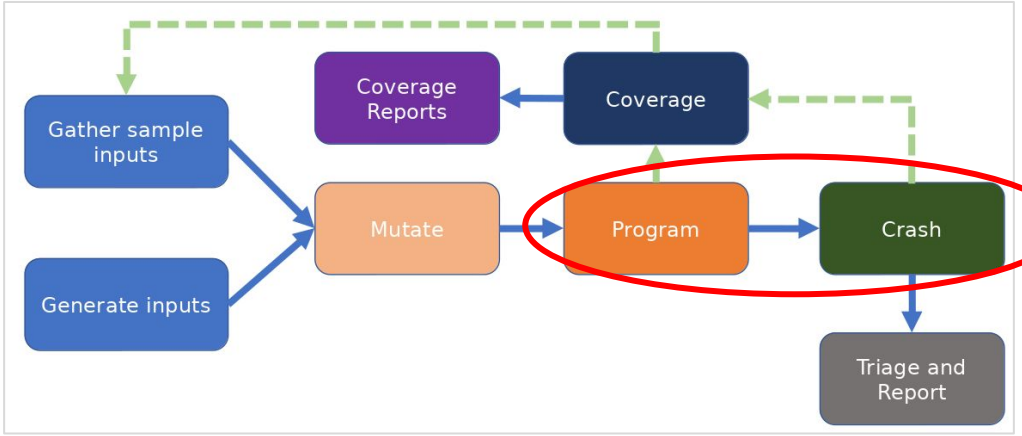
LABS: Minimized **lci** corpus with honggfuzz

- Minimized your corpora using honggfuzz
  - `../honggfuzz/honggfuzz -i input -M -- ./lci ___FILE___ > /dev/null`
- Verify some files have been removed
  - `ls input | wc`
  - `1292`

```
---------------------------------- [ LOGS ] ------------------/ honggfuzz 2.4 /-
Removed unnecessary 'ec07590089ec837f12475d93e300483f.00000010.honggfuzz.cov'
Removed unnecessary 'a615378aa4d70c7b430e2ec37071e533.00000049.honggfuzz.cov'
Removed unnecessary 'ba498eca5c8059756433b36c5d76422c.00002000.honggfuzz.cov'
Removed unnecessary '0eccc2614a7d60609214c71c413df087.0000006d.honggfuzz.cov'
Removed unnecessary '6cd742f339d1a9631750ca7908083640.0000000f.honggfuzz.cov'
Removed unnecessary '97412dd702f9dbdb3cbeda08e0d3ba97.00000040.honggfuzz.cov'
```

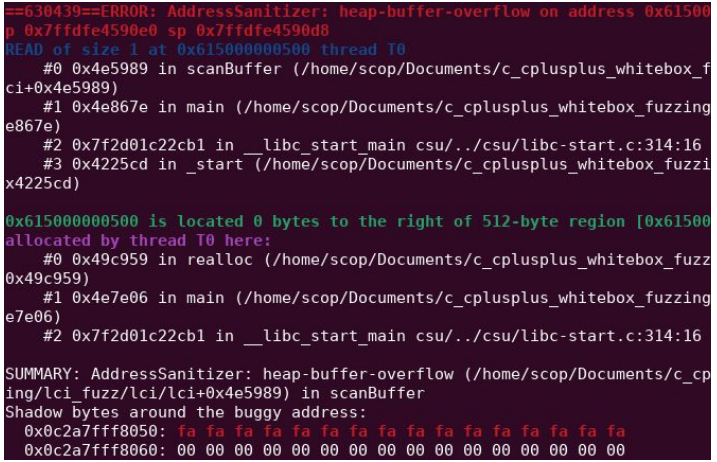
Fuzzing workflow

Sanitizers



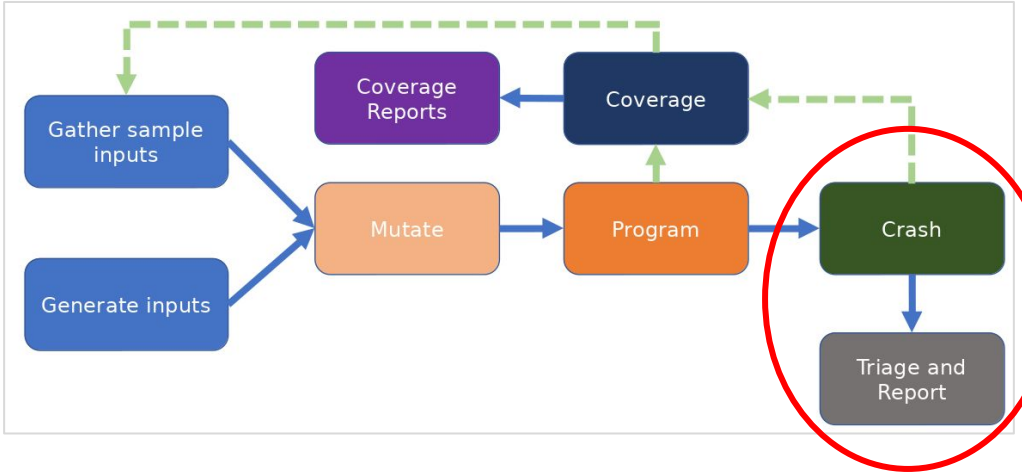
Sanitizers (ASAN, MSAN, …)

- Sanitizers are compiler instrumentation LLVM modules designed to **detect security issues.** Some assertions will be injected in the code making the program to crash when a failure is detected **at runtime.**
  - **AddressSanitizer (ASan) - Memory error detector** (2x slowdown)
    - Out-of-bounds accesses to heap, stack and globals, Use-after-free, Double-free, invalid free
    - `CCFLAGS="-fsanitize=address"`
  - LeakSanitizer (LSan) - Memory leak detector
    - `ASAN_OPTIONS=detect_leaks=1`
  - MemorySanitizer (MSan)- Detector of uninitialized reads
    - `CCFLAGS="-fsanitize=memory"`
  - ThreadSanitizer (TSan) - Data race detector
    - `CCFLAGS="-fsanitize=thread"`
  - UndefinedBehaviorSanitizer (UBSan)
    - undefined behavior detector
    - `CCFLAGS="-fsanitize=undefined"`



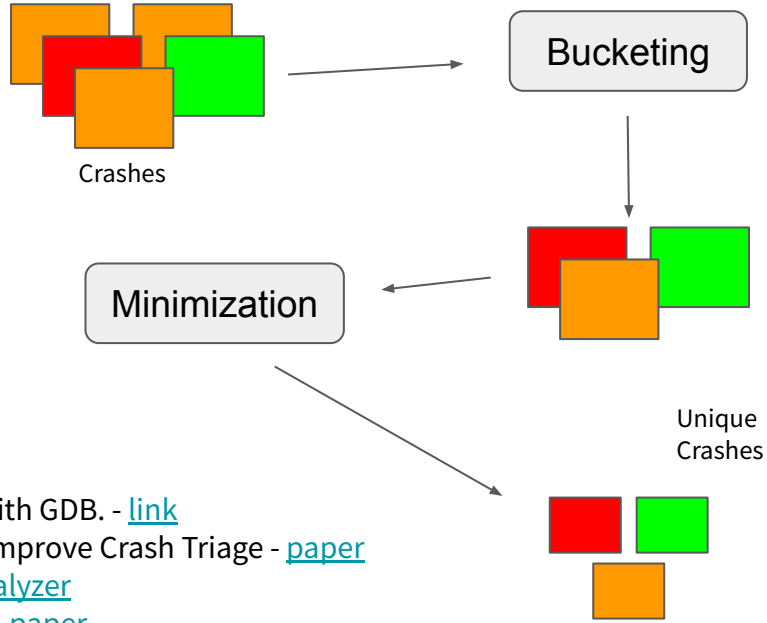
- Further readings:
  - AddressSanitizer: A Fast Address Sanity Checker - paper
  - Sanitize, Fuzz, and Harden Your C++ Code - link

Crashes Triaging



Crash Triaging Process

- Crash triaging is the **process of going through a list of bugs to analyze and report test cases** that cause policy violations and/or **crashes**.
- Steps overview:
  - **Bucketing**
    - De-duplicate bugs
  - **Crashes minimization**
    - Reduce size of each individual crashing sample
  - **Debugging / Root cause analysis (RCA)**
    - Manual debugging and crash analysis
    - Automated bugs analysis

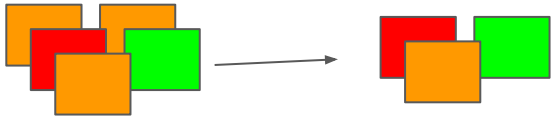


- Further readings:
  - Introduction to Triaging Fuzzer-Generated Crashes - link
  - Crash Triage Process - link
  - Triaging crashes with crashwalk and root cause analysis with GDB. - link
  - Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage - paper
  - Crash triage (AFL) - link, another script for afl, afl-crash-analyzer
  - Igor: Crash Deduplication Through Root-Cause Clustering - paper

Crashes Triaging

Bucketing

Bucketing



- Bucketing consist of **removing any duplicate** input files **that trigger the same bug**. Rather than having multiple input files that trigger three bugs, this step filters out duplicate input files such that **only one file per bug remains.**

- crashwalk - Bucket and triage on-disk crashes for OSX and Linux.
  - Debugs the target program while it processes each input file and analyzes the program state using the debugger.
  - Buckets each input file based on a **hash of the program's backtrace.**

- afl-cmin - corpus minimization tool of afl
  - Track the execution path a program takes when processing an input.
  - Typically used to reduce the size of a fuzzing corpus
    - **can also be used for bucketing crashing inputs with -C flag.**
    - Compares those paths for all of the inputs, and **saves the smallest file** that traverses each of the unique paths.

- Further readings:
  - Semantic Crash Bucketing - paper, github
  - ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity - link
  - Crash Buckets (FuzzingBook) - link
  - DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging - paper, website

LABS: Bucketing lci crashes manually

1. Remove duplicated file (exact same contents/bytes) using fdupes
   - `cd lci_fuzz/ && cp -r crash_to_bucket uniq_crashes && cd uniq_crashes`
   - `fdupes -r -PdN .`

```
~/Documents/c_cplusplus_whitebox_fuzzing_training/lci_fuzz/unique_crash(master*) » fdupes -r -PdN .

   [+] ./SIGSEGV.PC.4963be.STACK.dae19f5ae.CODE.1.ADDR.0.INSTR.mov____(%rax),%ecx.fuzz
   [-] ./SIGSEGV.PC.54510d.STACK.d03026bf9.CODE.1.ADDR.0.INSTR.mov____(%rax),%ecx.fuzz


   [+] ./SIGSEGV.PC.482704.STACK.16fc78d081.CODE.1.ADDR.9e9c000.INSTR.movsbl_(%rax),%ecx.fuzz
   [-] ./SIGABRT.PC.52ed8c.STACK.ca921d128.CODE.-6.ADDR.0.INSTR.mov____-0x368(%rbp),%rax.fuzz
```

2. Process all files to the target and compare errors generated
   - `for i in $(ls); do echo; ll $i; ../lci/lci $i 2>&1 | grep 'pc\|SUMMARY'; done`
   - Multiple indicators can be useful for triaging
     - **pc: Program counter** (where the crash occurs)
     - **SUMMARY** (sanitizer information, signal, crashing method, etc.)
   - Keep the smaller ones and rename them

```
~/Documents/c_cplusplus_whitebox_fuzzing_training/lci_fuzz/unique_c
heap-buffer-overflow-0x0000004e598a_scanBuffer.input
heap-buffer-overflow-0x0000004e81aa-main.input
SEGV-0x0000004eea2a-nextToken.input
stack-buffer-overflow-0x000000502376-convertCodePointToUTF8.input
```

3. **EXTRA EXERCISE**
   - Automate the process with your own script

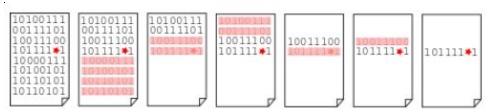
Crashes Triaging

Crashes minimization

Crashes minimization/reduction

- Crashes minimization (or delta debugging) tools takes an input file and **tries to remove as much data as possible** while keeping the **binary in a crashing state** and/or **maintaining the same coverage** observed.
  - Reduce input file size before fuzzing (fuzzer works better with small files).
  - Reduce crash file before analysis to only keep crashing bytes.

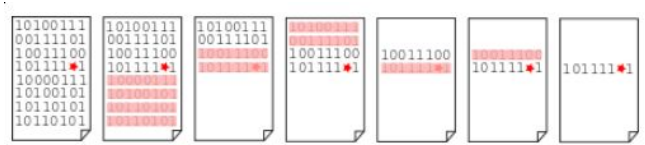
- C/C++ fuzzers tools & options:
  - **afl-tmin**
    - Simple test case minimizer that takes an input file and **tries to remove as much data as possible** while **keeping the binary in a crashing state** or producing consistent instrumentation output - link
    - Other test case minimizer based on afl-fuzz: afl-pytmin, afl-ddmin-mod
  - **libfuzzer**: `-minimize_crash`
    - Minify failing input to the smallest input that causes failure - link
  - **halfempty**
    - Testcase minimization tool, designed with parallelization in mind.



- Further readings:
  - Delta Debugging (by GRIMM) - link
  - Lithium: Lithium is an automated testcase reduction tool
  - Creating your own interestingness tests for Lithium - link

halfempty

- **Testcase minimization** tool, designed **with parallelization** in mind. Halfempty was built to use strategies and techniques that dramatically speed up the minimization process.
  - Created by Tavis Ormandy from Google Project Zero
  - Command
    - `halfempty test.sh target.bin`
  - Script examples: here or here



- Interesting options
  - `--num-threads=threads`
    - Halfempty will default to using all available cores, but you can tweak this if you prefer.
  - `--zero-char=byte`
    - Tries to simplify files by overwriting data with the given byte value.
  - `--monitor`
    - If you have the graphviz package installed, halfempty can generate graphs so you watch the progress.

- Sometimes your target program might crash with a different crash accidentally found during minimization. One solution might be to use gdb to verify the crash site - link

LABS: Reduce **lci** crashes with halfempty

1. Copy your unique files
   - `cp -r unique_crashes minimize_crashes`
2. Create a halfempty script
   - `touch halfempty_lci.sh`
   - `chmod +x halfempty_lci.sh`
3. Verify the script is working
   - `cat input.bin | ./halfempty_lci.sh || echo failed`
4. Run halfempty
   - `halfempty ./halfempty_lci_grep.sh input.bin`
5. Verify the result
   - `../lci/lci halfempty.out`

```
#!/bin/sh

# You need to change the program counter for each target
expected_pc="0x000000502376"

tempfile=`mktemp` && cat > ${tempfile}
result=1

trap 'rm -f ${tempfile}; exit ${result}' EXIT TERM ALRM

# Command to be tested
../lci/lci ${tempfile} 2>&1 | grep -q "pc $expected_pc"

# Check if we were killed with SIGSEGV
if test $? -eq 0; then
    result=0 # We want this input
fi
```

```
── halfempty ──────────────────────────────────────────── v0.40 ──
 16 A fast, parallel testcase minimization tool
──────────────────────────────────────────────────────── by @taviso ──

Input file "stack-buffer-overflow-0x000000502376-convertCodePointToUTF8.input" i
 strategy "bisect"...
Verifying the original input executes successfully... (skip with --noverify)
The original input file succeeded after 0.0 seconds.
New finalized size: 83 (depth=2) real=0.0s, user=0.0s, speedup=~-0.0s
New finalized size: 63 (depth=7) real=0.1s, user=0.2s, speedup=~0.2s
New finalized size: 43 (depth=8) real=0.1s, user=0.3s, speedup=~0.2s
New finalized size: 41 (depth=24) real=0.2s, user=1.1s, speedup=~0.9s
New finalized size: 39 (depth=31) real=0.3s, user=1.5s, speedup=~1.2s
```

Crashes Triaging

Debugging / Root cause analysis

Debugging - Symbols & Debuggers

- Compile with debugging symbols
  - `CFLAGS="-g -O0 -ggdb" CXXFLAGS="-g -O0 -ggdb" make`
  - GCC - Options for Debugging Your Program - link
- Debuggers
  - **gdb** - GDB (the GNU Project Debugger).
    - pwndbg, GEF, and PEDA - projects addings security/analysis features to GDB.
    - **GEF** - GDB Enhanced Features for exploit devs & reversers
      - **Improve readability**, add some **great variety of commands**
  - **lldb** - LLDB Debugger (LLVM project) - website, wiki, docs, tutorial
    - next generation, high-performance debugger.
- Further readings:
  - **The Debugging Book - Tools and Techniques for Automated Software Debugging**
  - GDB to LLDB command map - link
  - CS107 GDB and Debugging - link
  - Understanding, Scripting and Extending GDB - link
  - Advanced Debugging — Part 1: LLDB Console - link
  - Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries - paper

Valgrind

- Valgrind was originally designed to be a free **memory debugging tool** for Linux/x86, but evolved to become a **generic Instrumentation framework** for building dynamic analysis tools such as checkers and profilers.
  - Machine-code interpreter: just-in-time (JIT) instruction recompilation.
  - Useful to determine **memory leak** or other kind of **memory errors**.

- Valgrind tools:
  - Dynamic Heap Analysis: a dynamic heap analysis tool
    - detect short-lived allocation
    - `valgrind --tool=dhat mytarget`
  - Massif: a heap profiler
    - `valgrind --tool=massif mytarget`
  - **Memcheck: a memory error detector**

```
~/Documents/c_cplusplus_whitebox_fuzzing_training/lci_fuzz/lci_debug(master) » valgrind
e_crashes/stack-buffer-overflow-0x000000502376-convertCodePointToUTF8.min
==637311== Memcheck, a memory error detector
==637311== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==637311== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==637311== Command: ./lci ../minimize_crashes/stack-buffer-overflow-0x000000502376-conve
8.min
==637311==
*** stack smashing detected ***: terminated
==637311==
==637311== Process terminating with default action of signal 6 (SIGABRT)
==637311==    at 0x49F88CB: raise (raise.c:50)
==637311==    by 0x49DD863: abort (abort.c:79)
==637311==    by 0x4A40AF5: __libc_message (libc_fatal.c:155)
==637311==    by 0x4AE0799: __fortify_fail (fortify_fail.c:26)
==637311==    by 0x4AE0765: __stack_chk_fail (stack_chk_fail.c:24)
==637311==    by 0x191139: castStringExplicit (in /home/scop/Documents/c_cplusplus_whiteb
ning/lci_fuzz/lci_debug/lci)
==637311==    by 0x18FF98: castStringImplicit (in /home/scop/Documents/c_cplusplus_whiteb
ning/lci_fuzz/lci_debug/lci)
```

- Further readings:
  - Valgrind Quick Start Guide - link
  - CS107 Valgrind Memcheck - link
  - Fuzzgrind: an automatic fuzzing tool - link
  - Using Valgrind to Find Memory Leaks and Invalid Memory Use - link

LABS: Analyze **lci** crashes

1. Compile multiple versions of the target
   - One with **debug**/symbols only
     - `git clone https://github.com/justinmeza/lci lci_debug`
     - `export CCFLAGS="-g -O0 -ggdb"`
     - `cmake . && make`
   - One for each sanitizers
2. Debug with **GDB/GEF**
   - `gdb --args ./lci input.bin`
   - `break main`
   - `heap-analysis` (heap-analysis-helper command)
   - `run`
   - `bt # backtrace`
3. Analyze **sanitizers** stacktraces/logs
   - ASAN, LSan, MSan, TSan, UBSan
4. Run **Valgrind**
   - Only working on the debug version

```
    0x4eea1d <nextToken+205>  mov    rdi, QWORD PTR [rbp-0x50]
    0x4eea21 <nextToken+209>  call   0x4a2960 <__asan_report_load4>
    0x4eea26 <nextToken+214>  mov    rax, QWORD PTR [rbp-0x50]
→   0x4eea2a <nextToken+218>  mov    ecx, DWORD PTR [rax]
    0x4eea2c <nextToken+220>  mov    edx, DWORD PTR [rbp-0x14]
    0x4eea2f <nextToken+223>  mov    edi, ecx
    0x4eea31 <nextToken+225>  mov    esi, edx
    0x4eea33 <nextToken+227>  mov    DWORD PTR [rbp-0x58], ecx
    0x4eea36 <nextToken+230>  mov    DWORD PTR [rbp-0x5c], edx
[#0] Id 1, Name: "lci", stopped 0x4eea2a in nextToken (), reason: SIGSEGV
[#0] 0x4eea2a → nextToken()
[#1] 0x4f9e4d → parseLoopStmtNode()
[#2] 0x4fd5aa → parseStmtNode()
[#3] 0x4f6e93 → parseBlockNode()
[#4] 0x4fdbd7 → parseMainNode()
[#5] 0x4e8727 → main()
```

```
[*] Heap-Analysis
Possible Use-after-Free in '/home/scop/Documents/c_cplusplus_whitebox_fuzzing_training/lg/lci': pointer 0x5555556d99f0 was freed, but is attempted to be used at 0x7ffff7d0511e
0x7ffff7d0511e   punpcklqdq xmm0, xmm1
```

End of the day ;)
Questions?

Thanks & Question

- Patrick Ventuzelo / @Pat_Ventuzelo / patrick@fuzzinglabs.com
- If you are interested about **fuzzing, vulnerability research and/or blockchain security**, contact me ;)