

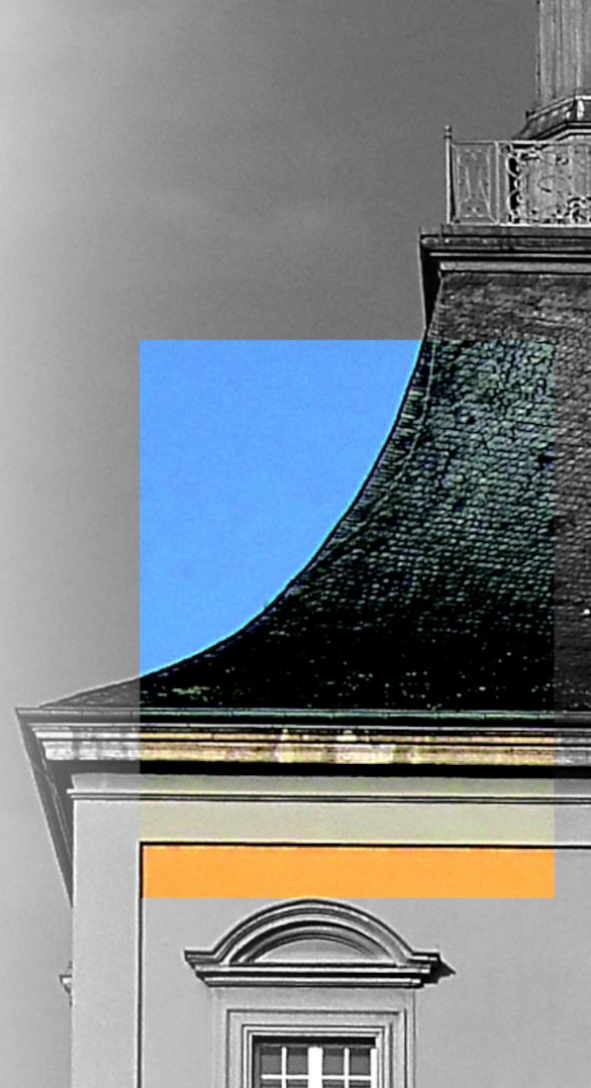
MA-INF 3236

IT Security

Introduction to Fuzzing

Christian Hartlage
s6chhart@uni-bonn.de

Lecture IT Security | Uni Bonn | WT 2024/25



Christian Hartlage



**B.Sc.
Informatik
(2012-2016)**



**M.Sc.
Computer Science
(2017-2020)**



**SHK/WHB
Malware
Analysis
(2017-2019)**



code intelligence

**Consultant
Software
Testing
(2019-2020)**



**Referent
Forschungs-
koordination
(2021-)**



Wir sind das #TeamBSI



Bundesamt
für Sicherheit in der
Informationstechnik

<https://team-bsi.de>



Das BSI als Arbeitgeber



Einstiegsmöglichkeiten

- Ausbildung in der IT und der Verwaltung
- Duales Studium in Kooperation mit der Hochschule des Bundes (DACs)
- Praktika
- Abschlussarbeiten (Master und Bachelor)
- Festanstellung nach Berufseinstieg und mit Berufserfahrung
- Masterförderung



Unsere Werte

Einzigartigkeit
und Sinnhaftigkeit unserer Aufgaben

vielseitige
Karriereoptionen

Job mit
Perspektive

Gestaltungsspielraum

Vielfalt

ausgewogene
Work-Life-Balance

persönliche
Weiterbildung

Today's agenda

- Fuzzing in theory
 - History
 - Concepts
- Fuzzing in practice
 - C/C++
 - AFL++
 - LibFuzzer
 - Go
 - Java

What is fuzzing?

- Software Testing
- Random Testing
- black-/grey-/whitebox
- AI
- language independent (in theory)
- Good at finding bugs!
- A large research area
- Fun





■ functional testing

- make sure usual cases work fine
- `assert(mul(2, 2), 4)`

■ regression testing

- make sure things that broke in the past don't break again
- `assert(div(2, 0), ERROR)`

■ robustness testing

- make sure infinite monkeys on infinite keyboards don't break the system
- `assert(add(, , , , DOES NOT EXPLODE))`

Robustness (Software)
From Wikipedia, the free encyclopedia

ANSI and IEEE have defined robustness as the degree to which a system can function correctly in the presence of invalid inputs.^[1]

Fuzzing: A short history excursion pt.1

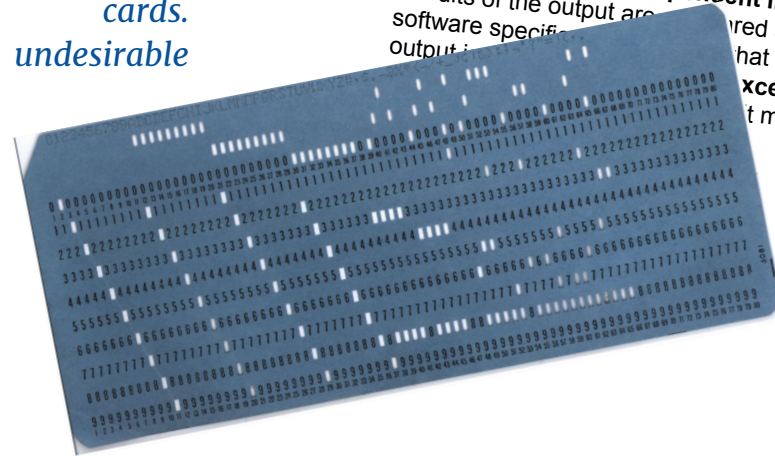
“it was our standard practice to test programs by inputting decks of punch cards taken from the trash. We also used decks of random number punch cards. [...] our random/trash decks often turned up undesirable behavior.”

- Computer scientist Gerald Weinberg (IBM) ^[2]

Random testing

From Wikipedia, the free encyclopedia

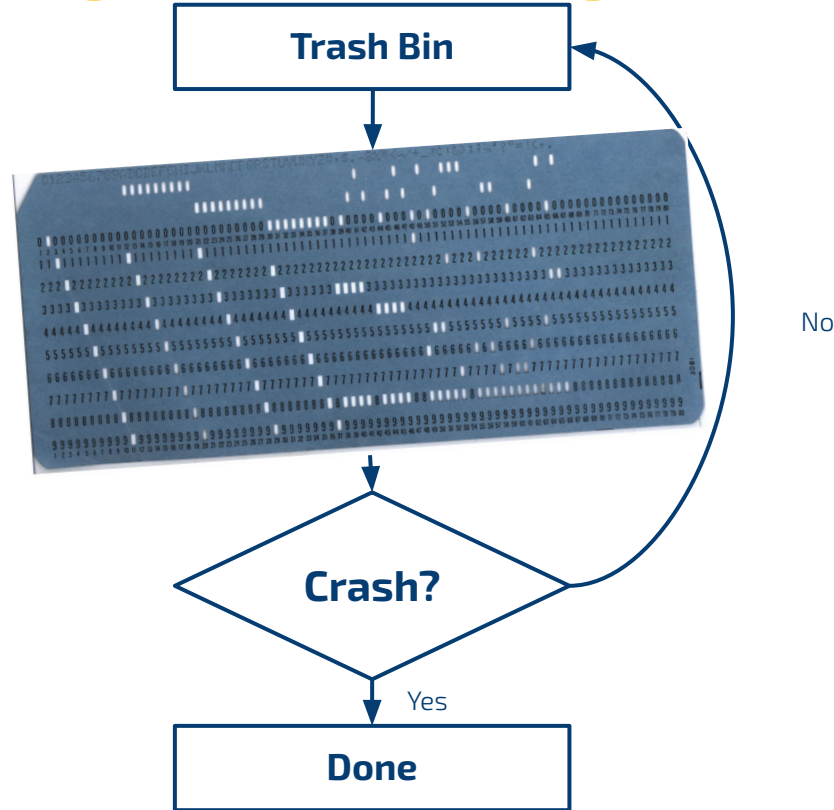
Random testing is a **black-box** software testing technique where programs are tested by generating **random, independent inputs**. Results of the output are compared against software specifications to determine if the test output is correct. An **exception** means there



[1]: [Random testing - Wikipedia](#)

[2]: [Fuzzing: An Old Testing Technique Comes of Age - The New Stack](#)

Fuzzing: Punch Card “Algorithm”

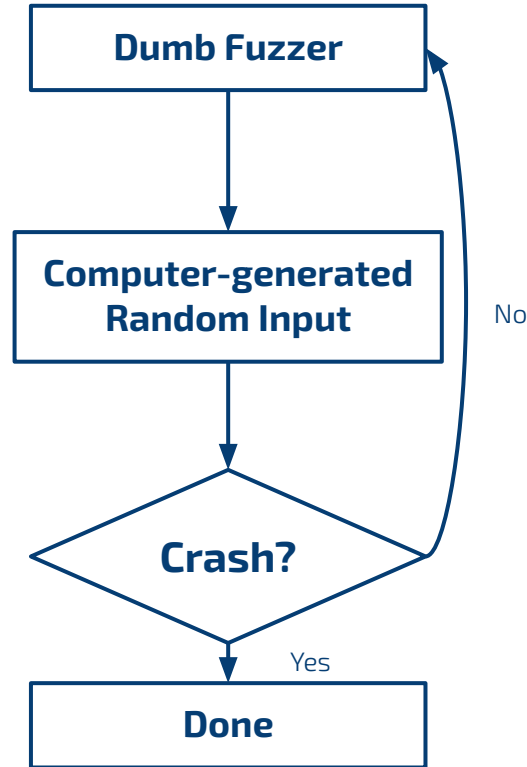


Fuzzing: A short history excursion pt.2

- Software based random testing emerged between 1960 and the 1980s
 - Simply called Random Testing or Monkey Testing
- “Fuzzing” was invented in 1988 by Barton Miller at the University of Wisconsin^[1]
- Miller defined three criteria that make fuzzing more effective than other testing methods:
 1. *The input is random. [...]*
 2. *Our reliability criteria is simple: if the application crashes or hangs, it is considered to fail the test, otherwise it passes.*
 3. *As a result of the first two characteristics, classic fuzz testing can be automated to a high degree and results can be compared across applications, operating systems, and vendors.*

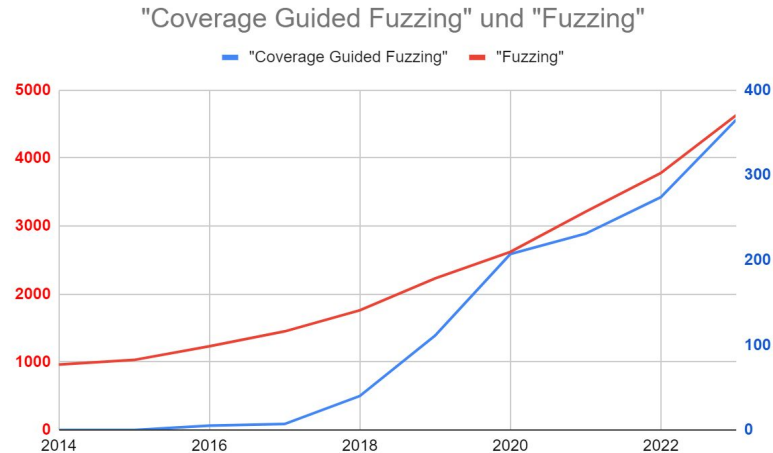
[1]: [Fuzz Testing of Application Reliability - UW Madison](#)

Fuzzing: First Fuzzing Algorithm

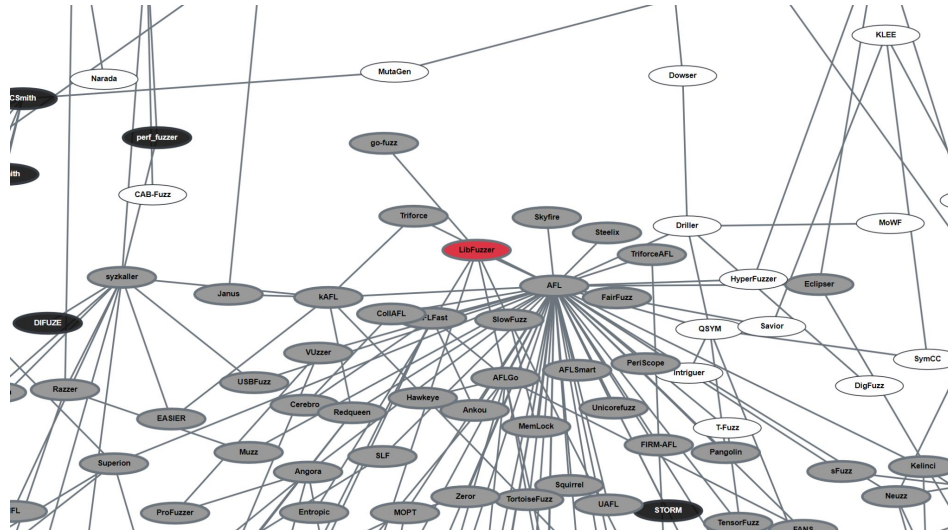


Fuzzing: A short history excursion pt.3

- Many more testing techniques arose in the 2000s
 - whitebox testing
 - symbolic execution
 - **coverage guided fuzzing** (today's lectures focus)



- 1111



Fuzzing: Good at finding bugs

GitLab Acquires Peach Tech and Fuzzit to Expand its DevSecOps Offering

You are here: Press and Logos > Press releases > GitLab Acquires Peach Tech and Fuzzit to Expand its DevSecOps Offering

Acquisitions will make GitLab the first security solution to offer both coverage-guided and behavioral fuzz testing


SAN FRANCISCO, CALIFORNIA — June 11, 2020 - Today [GitLab](#), the single application for the DevOps lifecycle, announced it has acquired Peach Tech, a security software firm specializing in protocol fuzz testing and dynamic application security testing (DAST) API testing, and Fuzzit, a continuous fuzz testing solution providing coverage-guided testing. These acquisitions will add fully-mature testing solutions including protocol fuzzing, API fuzzing, DAST API testing, and coverage-guided fuzz testing. This makes GitLab's DevSecOps offering the first security solution to offer both coverage-guided and behavioral fuzz testing techniques as well as the first true DevSecOps platform to shift fuzz testing left with these new offerings being made available within the GitLab CI/CD environment.

"We believe GitLab provides best-in-class tools for the complete DevOps lifecycle on a single platform," said Sid Sijbrandij, CEO of GitLab. "Bringing the fuzzing technologies of Peach Tech and Fuzzit into GitLab's security solutions will give our users an even more robust and thorough application security testing experience while enabling them to shift security left. This simultaneously simplifies their workflows and creates collaboration between development, security, and operations teams."

<https://about.gitlab.com/press/releases/2020-06-11-gitlab-acquires-peach-tech-and-fuzzit-to-expand-devsecops-offering.html>

- Software Testing ✓
- Random Testing ✓
- black-/grey-/whitebox ?
- AI ?
- language independent (in theory) ✓
- Good at finding bugs! ✓
- A large research area ✓
- Fun ?

Today's agenda

- Fuzzing in theory
 - History 
 - Concepts
- Fuzzing in practice
 - C/C++
 - AFL++
 - LibFuzzer

- Black-/grey-/whitebox testing
 - Coverage
 - Instrumentation
- AI
 - Seed Files
 - Mutations

Concept: Black-/grey-/whitebox

- Complete Knowledge of the underlying code
- Source is available
- Analytical test methods
(code is usually not executed)

Whitebox

- No Knowledge of the underlying code
- Source is not available
- Dynamic test methods
(code is executed)

Blackbox

Combination of tools from both
methods can be called greybox

Fuzzing: Black-/grey-/whitebox

- Complete Knowledge of the underlying code
- **Source is available**
- Analytical test method
(code is usually not executed)
- **compile time instrumentation**

Whitebox

- No Knowledge of the underlying code
- Source is not available
- **Dynamic test methods
(code is executed)**
- **Dynamic binary instrumentation**

Blackbox

Coverage Guided Fuzzing uses
techniques from white- and blackbox
testing
→ greybox testing

Fuzzing: Code Coverage

- There are different approaches to code coverage
- Line Coverage is one of the most popular metrics
 - has a line been executed → yes / no
 - similar to statement coverage
- More simple metrics include:
 - basic block coverage
 - function coverage
- More complex metrics are:
 - Code coverage metrics concerning the **control flow graph** of a program
 - path coverage
 - edge coverage
 - branch coverage

Code coverage

From Wikipedia, the free encyclopedia

In computer science, test coverage is a percentage measure of the **degree to which the source code of a program is executed** when a particular **test suite is run**. A program with **high test coverage** has **more** of its source code **executed** during testing, which suggests it has a **lower chance** of containing undetected software **bugs** compared to a program with low test coverage.

Fuzzing: Code Coverage - Line coverage

The screenshot illustrates the process of analyzing code coverage in Visual Studio Code. The 'Test' menu is open, showing the option 'Analyze Code Coverage for All Tests'. The 'Test' window displays a list of tests and their durations. The 'Code Coverage Results' window shows a hierarchy of code coverage for the project, with a table of results. The main editor shows the source code of 'SquareRoot' with line coverage indicators (green for covered, red for not covered).

Test Results:

Test	Duration	Traits
MathTests (3)	31 ms	
MathTests (3)	31 ms	
UnitTests (3)	31 ms	
BasicRooterTest	31 ms	
RooterValueRange	< 1 ms	
TestMethod1	< 1 ms	

Code Coverage Results:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (%)
mikejo_MIKEJO_SURFACE 2020-03-31 13...	2	8.70%	21	91.30%
mathTests.dll	0	0.00%	14	100.00%
UnitTests	0	0.00%	14	100.00%
BasicRooterTest()	0	0.00%	4	100.00%
RooterOneValue(MyMath.Ro...	0	0.00%	3	100.00%
RooterValueRange()	0	0.00%	6	100.00%
TestMethod1()	0	0.00%	1	100.00%
mymath.dll	2	22.22%	7	77.78%
MyMath	2	22.22%	7	77.78%

Source Code (SquareRoot):

```

11 public double SquareRoot(double input)
12 {
13     if (input <= 0.0)
14     {
15         throw new ArgumentOutOfRangeException();
16     }
17     double result = input;
18     double previousResult = -input;
19     while (Math.Abs(previousResult - result) > result / 1000)
20     {
21         previousResult = result;
22         result = (result + input / result) / 2;
23         //was: result = result - (result * result - input) / (2*result)
24     }
25     return result;
26 }

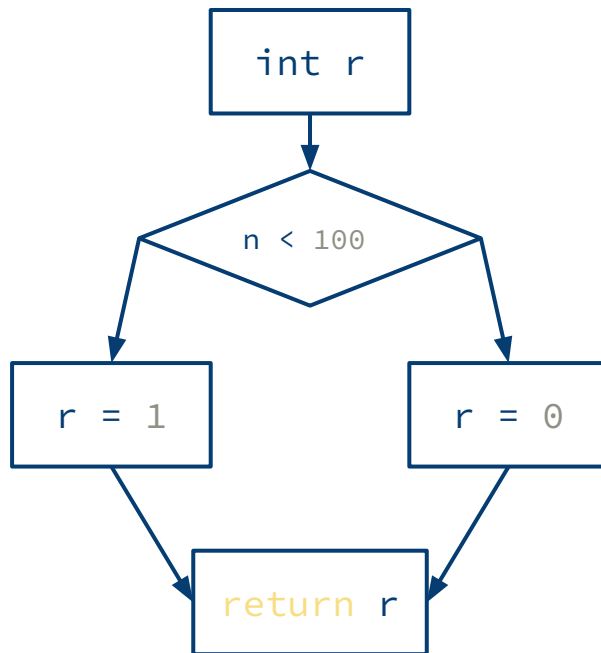
```

An example of line coverage visualization in Visual Studio Code

Fuzzing: Code Coverage - Control Flow Graph

```
int func(int n) {
    int r;
    if (n < 100) {
        r = 1;
    } else {
        r = 0;
    }
    return r;
}
```

A simple function called func



The control flow graph of func

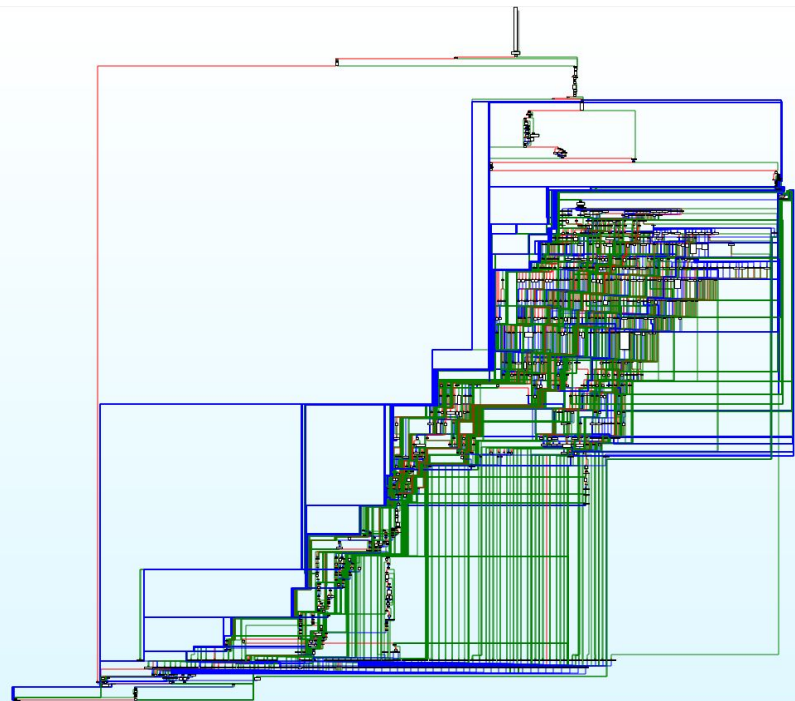
Control Flow Graph

From [GNU Compiler Collection \(GCC\) Internals Manual](#)

A control flow graph (CFG) is a data structure [...] abstracting the control flow behavior of a function that is being compiled.

The CFG is a **directed graph** where the **vertices** represent **basic blocks** and **edges** represent **possible transfer of control flow** from one basic block to another.

Fuzzing: Coverage - Control Flow Graph pt.2

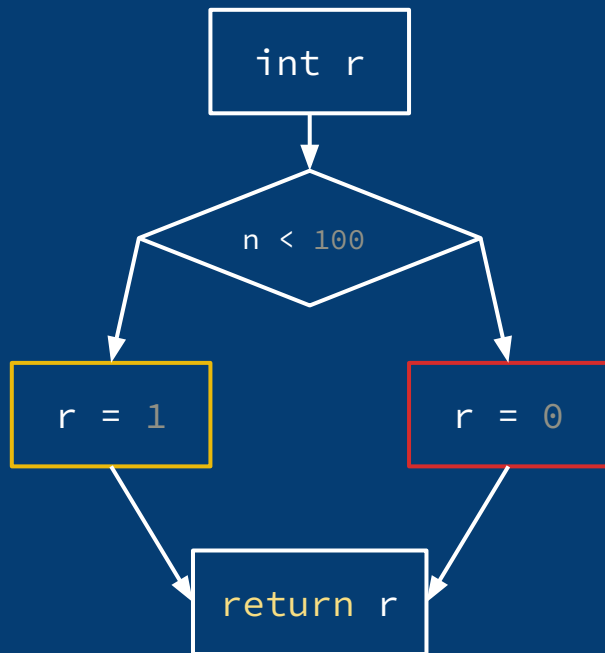


- Control Flow Graph based code coverage helps to answer the question:
“how much of the applications logic has been tested”
- This is what has been done in fuzzing since at least 2016 → Coverage based fuzzing
- This is why the source code is needed → greybox fuzzing

IDA generated Control flow graph representation of a function in ATMF.DLL used by the windows kernel
taken from [Project Zero: One font vulnerability to rule them all #1: Introducing the BLEND vulnerability](#)

Fuzzing: Instrumentation in C/C++

```
int func(int n) {
    int r;
    if (n < 100) {
        r = 1;
    } else {
        r = 0;
    }
    return r;
}
```



```
cmp    edi,0x64
jge    <func+0x1d>
mov     DWORD PTR [rbp-0x8],0x1
jmp     <func+0x24>
mov     DWORD PTR [rbp-0x8],0x0
mov     eax,DWORD PTR [rbp-0x8]
```

LibFuzzer compiler
instrumentation

```
call    <__sanitizer_cov_trace_pc>
cmp     edi,0x64
jge     <func+0x1d>
call    <__sanitizer_cov_trace_pc>
mov     DWORD PTR [rbp-0x8],0x1
jmp     <func+0x24>
call    <__sanitizer_cov_trace_pc>
mov     DWORD PTR [rbp-0x8],0x0
call    <__sanitizer_cov_trace_pc>
mov     eax,DWORD PTR [rbp-0x8]
```

Recap: Black-/grey-/whitebox

- Complete Knowledge of the underlying code
- Source is available
- Analytical test methods
(code is usually not executed)

Whitebox

- No Knowledge of the underlying code
- Source is not available
- Dynamic test methods
(code is executed)

Blackbox

Combination of tools from both
methods can be called greybox

Fuzzing: Black-/grey-/whitebox

- Complete Knowledge of the underlying code
- **Source is available**
- Analytical test method
(code is usually not executed)
- **compile time instrumentation**

Whitebox

- No Knowledge of the underlying code
- Source is not available
- **Dynamic test methods
(code is executed)**
- **Dynamic binary instrumentation**

Blackbox

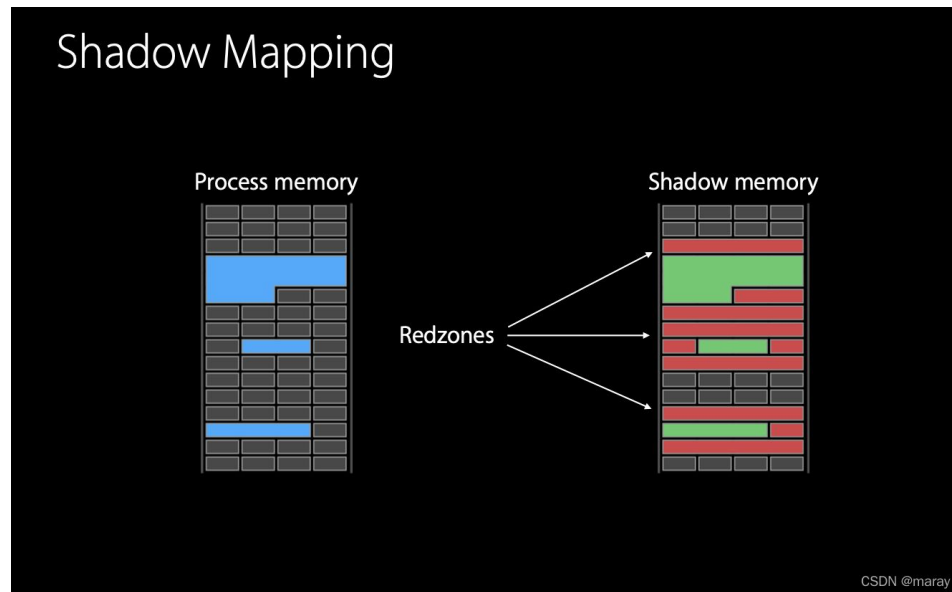
Coverage Guided Fuzzing uses
techniques from white- and blackbox
testing
→ greybox testing

Fuzzing: Address Sanitizer Instrumentation

Address **Sanitizer** is an instrumentation + runtime developed by Google.

The instrumentation module creates poisoned redzones around stack and global objects to detect overflows and underflows.

The runtime replaces *malloc*, *free* etc, to create the poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.



A visualisation of redzones in Shadow Memory

Fuzzing: Address Sanitizer Output

SUMMARY: AddressSanitizer: stack-buffer-overflow
(zint/build_libfuzzer/fuzzer+0x50d6a8) in strcat

Shadow bytes around the buggy address:

```

0x100034fa3d10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d50: 00 00 00 00 00 00 00 00 00 00 f2 f2 f2 f2 f2 f2
=>0x100034fa3d60: f2 f2 f2 f2 f2 f2 f2 f2 f2 f2 00 00 04 f3 f3 f3
0x100034fa3d70: f3 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x100034fa3d90: ca ca ca ca 00 07 cb cb cb cb cb cb f1 f1 f1 f1
0x100034fa3da0: f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f2
0x100034fa3db0: f2 f2 f2 f2 f8 f2 00 f3 f3 f3 f3 f3 00 00 00 00

```

Shadow byte legend (one shadow byte represents 8 application bytes):

```

Addressable:                00
Partially addressable:      01 02 03
Heap left redzone:          fa
Freed heap region:          fd
Stack left redzone:         f1
Stack mid redzone:          f2
Stack right redzone:        f3
Stack after return:         f5
Stack use after scope:      f8
Global redzone:             f9
Global init order:          f6
Poisoned by user:           f7
Container overflow:         fc
Array cookie:               ac
Intra object redzone:       bb
ASan internal:              fe
Left alloca redzone:        ca
Right alloca redzone:       cb
Shadow Gap:                 cc

```

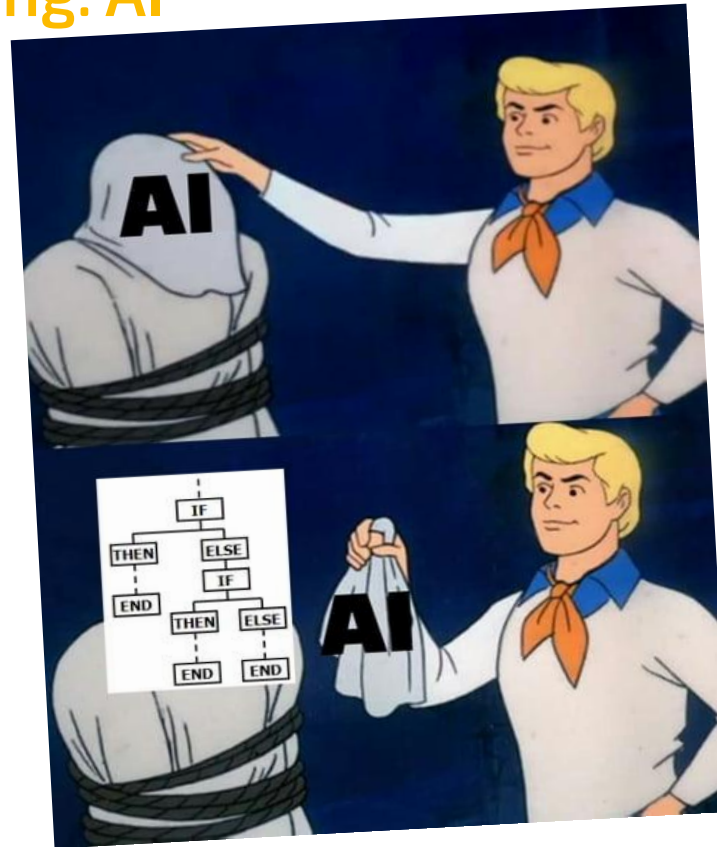
==35439==ABORTING

Fuzzing: Address Sanitizer Instrumentation

- Basically, ASAN makes applications crash more easily in order to discover hidden bugs
- Fuzzing with ASAN greatly improves the fuzzers bug finding abilities
- ASAN significantly slows down fuzzing
 - AFL++ even warns against only using ASAN: [Notes for Asan | AFLplusplus](#)
- Often it makes sense to run a fuzzer instance with ASAN and one without and let them share information (the corpus)

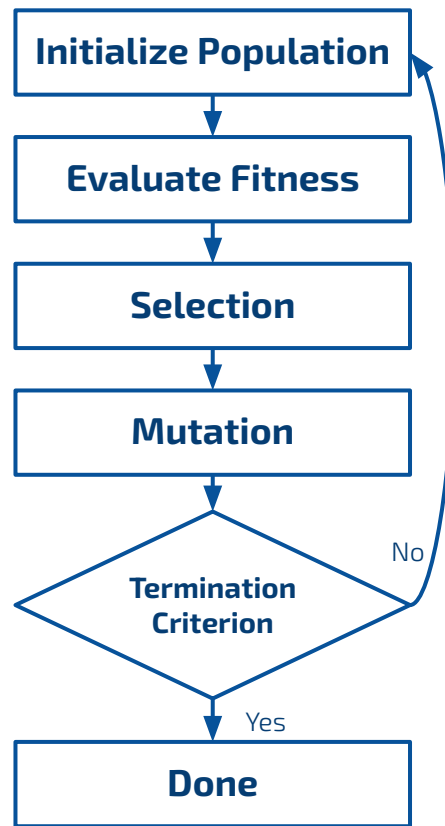
- Black-/grey-/whitebox testing ✓
 - Coverage ✓
 - Instrumentation ✓
- AI
 - Seed Files
 - Mutations

Fuzzing: AI

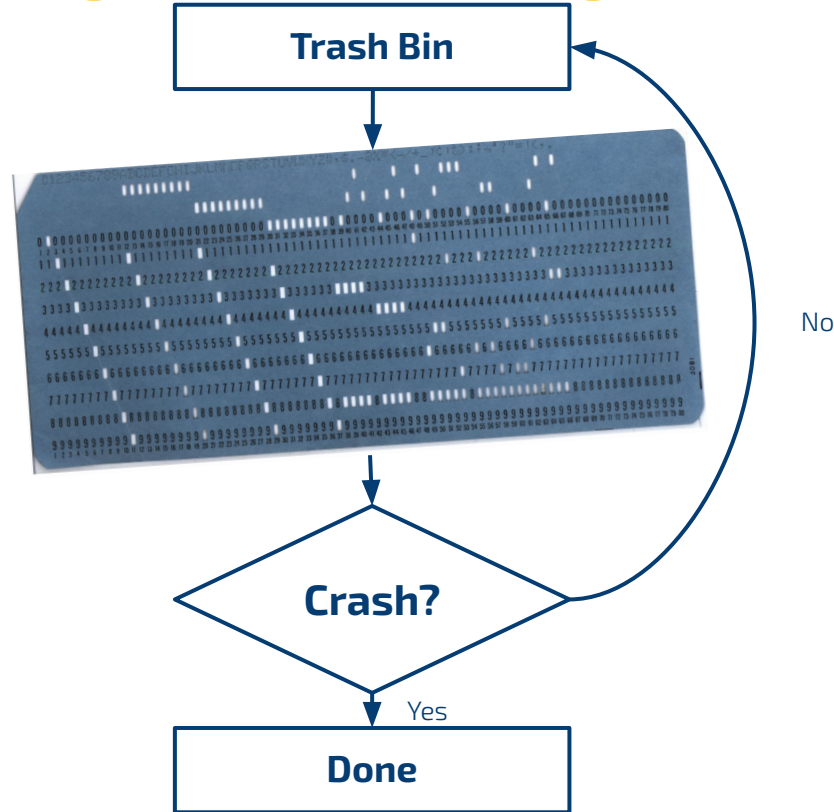


Genetic Algorithm / Evolutionary Algorithm

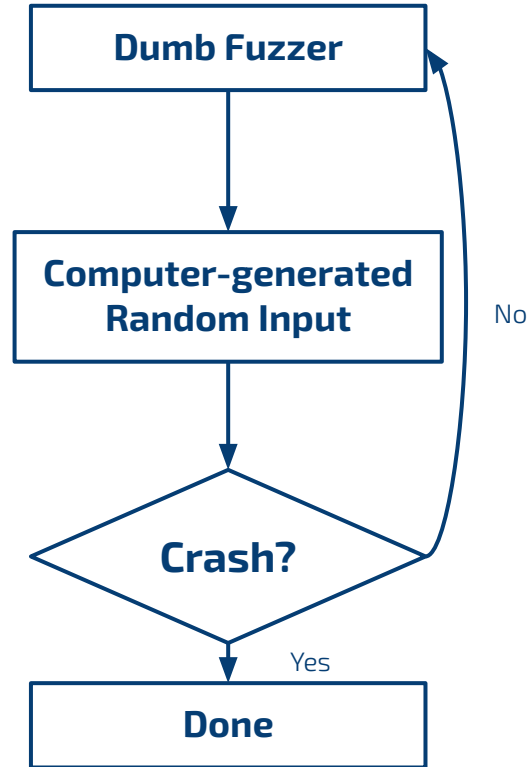
- 1 Initial Population = Seed Files
- 2 Fitness = (Path) Coverage
- 3 Take fittest samples = highest coverage
- 4 Mutate = Many different Mutation strategies
- 5 Termination ???
 - On crash
 - On memory access violation (ASAN)
 - On Timeout
 - After n runs



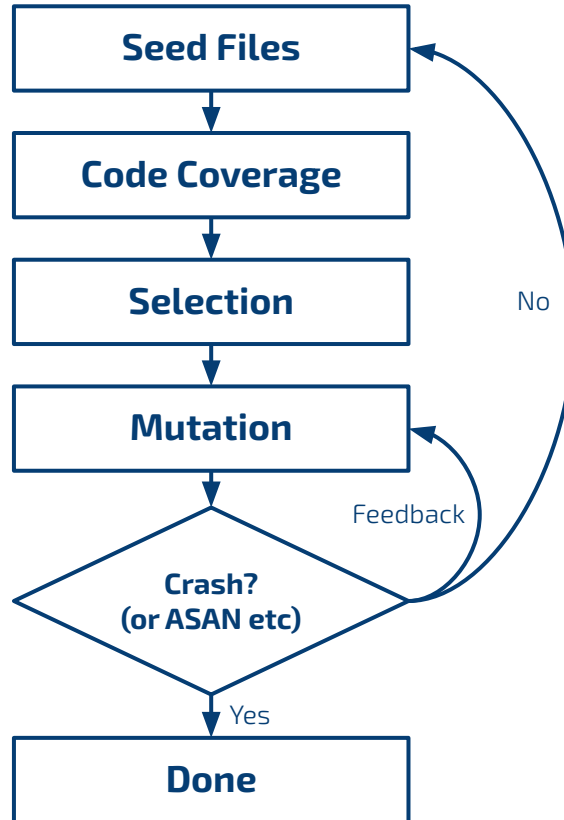
Fuzzing: Punch Card “Algorithm”



Fuzzing: First Fuzzing Algorithm



Fuzzing: Coverage Guided Fuzzing Algorithm



Fuzzing: Seed Files

- Providing Seed files greatly increases the performance of a fuzzer
- The seed file should be a valid input file for the target application. While invalid input can sometimes yield interesting bugs and crashes, valid input will find more paths, sooner.
- To fuzz a image parser you would provide a set of valid images
 - The fuzzer uses that set and mutates it into (invalid) inputs that will hopefully crash the parser
- To fuzz a json parser you would provide a set of valid json files
- To fuzz a **text-to-barcode generator** you would provide a set of valid input strings
- As it usually take less time for an program to process smaller input files, the seed file should be small — under 1 KB. This will result in more executions per second.
- Existing unit test suites often include input files — these can be useful seed files. ¹

[1]: [Fuzz Testing: Choosing a Seed File for AFL](#) | by David Moore

Fuzzing: Mutation



- One of the largest research areas in fuzzing
- Basic mutation algorithms:
 - bitflips
 - byteshuffling
 - string randomization
- More complexity is required for some applications
 - Network applications, require correct protocol usage and state tracking
 - Fuzzing application logic behind a REST-API
 - We don't want to fuzz the JSON parser here but the logic behind
 - All inputs must be generated as valid JSON, but with mutated payload
- Grammar based fuzzing can generate structured input



AFL's mutation of a JPEG

- Black-/grey-/whitebox testing ✓
 - Coverage ✓
 - Instrumentation ✓
- AI
 - Seed Files ✓
 - Mutations ✓

Today's agenda

- Fuzzing in theory
 - History 
 - Concepts 
- Fuzzing in practice
 - C/C++
 - AFL++
 - LibFuzzer

How to fuzz a project

1. Identify a project
2. Prepare the fuzzing
3. Run the Fuzzer


Fuzzing in practice: Identifying a project

- If you are a cybersecurity student:
 - The lecturer will probably tell you
- If you are a cybersecurity consultant:
 - The customer will tell you.
 - Also the customer is always right. Even it that means fuzzing a UI buttons callback function.
- If you are a trying to find bugs in OSS:
 - Your motivations might be bug bounties, securing your own projects dependencies or just helping an OSS-project
 - Software that **parses, generates, converts, encodes** or **decodes**
 - Software that processes user input of any kind
 - Software that reads from STDIN
 - Functions that accept arrays of bytes

Fuzzing in practice: Identifying a project

- If you are a cybersecurity student:
 - The lecturer will probably tell you
- Let's take a look at **yaml-cpp**
 - <https://github.com/jbeder/yaml-cpp>
 - “A YAML parser and emitter in C++”
 - Not (yet) in OSS-Fuzz
 - Googling yaml-cpp and fuzzing returns some results, so others have tried already. Still might be worth it

How to fuzz a project

1. Identify a project  yaml-cpp
2. Prepare the fuzzing
3. Run the Fuzzer

Fuzzing in practice: Preparing a project

LibFuzzer approach

1. Identify a function that can be used as an entry point
2. Instrument the library for fuzzing with libfuzzer
3. Write a Fuzz Target for the function

AFL++ approach

1. Check whether there is a CLI that reads either from STDIN or from a file
2. Instrument the library for fuzzing with AFL++

OR

1. Identify a function that can be used as an entry point
2. Instrument the library for fuzzing with AFL
3. Write a Fuzz Target for the function

Fuzzing in practice: Looking at yaml-cpp

LibFuzzer approach

The documentation suggests using
`YAML::Load()`

AFL++ approach

The library ships with some utils.
One of them is called `parse` and it reads STDIN
into `YAML::Load()`

Fuzzing in practice: Instrumenting a project

LibFuzzer approach

- Use a modern `clang` (≥ 5) compiler
- compile the whole project with
`-fsanitize=fuzzer-no-link`
and your sanitizer of choice eg.
`-fsanitize=address, fuzzer-no-link`

AFL++ approach

- AFL ships its own compilers:
 - `afl-clang` / `afl-clang++` / **`afl-clang-fast`** / **`afl-clang-fast++`**
 - no compiler flags necessary
- compile the whole project with `afl-clang`

There are several ways to instrument whole projects:

- Modifying the projects build scripts ([MAKEFILE](#), [CMakeLists.txt](#), [Autoreconf](#))
- Modifying environment variables: [CFLAGS/CXXFLAGS/CPPFLAGS](#),
- This can be the hardest part about fuzzing a project

Fuzzing in practice: Instrumenting yaml-cpp

yaml-cpp uses CMake, so instrumenting the library will be fairly easy, as CMake *usually* respects CFLAGS and friends

LibFuzzer approach

```
mkdir build
cd build
export CXX=clang++
export CXXFLAGS="-g -fsanitize=fuzzer-no-link,address"
cmake ..
make -j
```

AFL++ approach

```
mkdir build
cd build
export CXX=afl-clang-fast++
cmake ..
make -j
```

Fuzzing in practice: Writing a fuzz target

Writing a good fuzz target is a science of its own. Some key points¹:

- The fuzzing engine will execute it many times with different inputs in the same process.
- It must be as deterministic as possible. Non-determinism (e.g. random decisions not based on the input bytes) will make fuzzing inefficient.
- It must be fast. Try avoiding cubic or greater complexity.
- It should accept an array of bytes as the input. If it does not, you have to write some functionality to convert an array of bytes to arbitrary data types.

There is lots of examples on [OSS-Fuzz](#), [Fuzzbench](#) or the older [Fuzzer-Test-Suite](#)

[1]: [fuzzing/good-fuzz-target.md at master · google/fuzzing · GitHub](#)

Fuzzing in practice: A fuzz target for yaml-cpp

```
#include <string>
#include <stdint.h>
#include <iostream>
#include "yaml-cpp/yaml.h"

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    try {
        if (Size > 0) {
            YAML::Node doc = YAML::Load(std::string(reinterpret_cast<const char *>(Data), Size));
        }
    } catch (const YAML::Exception& e) {
    } catch (const std::exception& e) { }
    return 0; // Non-zero return values are reserved for future use.
}
```

yaml-fuzzer.cpp, A fuzz target for yaml-cpp

Fuzzing in practice: compiling the fuzz target

Compiling the fuzz target can be a bit tricky. Some things to consider:

- The fuzz target has to be correct. Check if you are inputting data in the correct way.
- Make sure the including the headers is working
 - use `-I <directory>` ← this is an uppercase i
 - `-I` adds the specified directory to the search path for include files
- Make sure you are linking against the instrumented library
 - check the library with `nm`, to see if it is instrumented
 - use `-L <directory>`
 - `-L` adds the specified directory to the search path for libraries
 - use `-l<libraryname>` ← this is an lowercase L
 - `-l` adds the library to the libraries to be linked against

Fuzzing in practice: compiling yaml-fuzzer.cpp

- Use `-fsanitize=fuzzer` and not `-fsanitize=fuzzer-no-link` when building the fuzz target
- Make sure to also use `-fsanitize=address` if you have used it when building the library



```
bash-5.1$ nm libyaml-cpp.a | grep sanitizer_cov
          U __sanitizer_cov_8bit_counters_init
          U __sanitizer_cov_pcs_init
          [...]
```

using `nm` to check whether `libyaml-cpp.a` is instrumented for fuzzing

```
bash-5.1$ clang++ yaml-fuzzer.cpp -o yaml-fuzzer -fsanitize=fuzzer,address -L. -lyaml-cpp
-I../include
```

compiling `yaml-fuzzer.cpp` with `clang++`

How to fuzz a project

1. Identify a project  yaml-cpp
2. Prepare the fuzzing 
3. Run the Fuzzer

LibFuzzer will produce a fuzzer executable that you can run like any other executable:

```
bash-5.1$ ./yaml-fuzzer
INFO: Seed: 2750339890
INFO: Loaded 1 modules (5144 inline 8-bit counters): 5144 [0x68ce40, 0x68e258),
INFO: Loaded 1 PC tables (5144 PCs): 5144 [0x68e258,0x6a23d8),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 967 ft: 968 corp: 1/1b exec/s: 0 rss: 32Mb
      NEW_FUNC[1/424]: 0x5645e0 in YAML::Mark::null_mark()
(/yaml-cpp/build_libfuzzer/fuzzer+0x5645e0)
```

If you linked the fuzzer against a dynamic library/shared object you will have to specify the directory containing the .so file with the `LD_LIBRARY_PATH1` environment variable:

```
bash-5.1$ LD_LIBRARY_PATH=. ./yaml-fuzzer
INFO: Seed: 2750339890
```

[1] <https://stackoverflow.com/a/4250666/2899746>

Fuzzing in practice: Running AFL++

In order to fuzz with AFL++ you call the standalone fuzzer afl-fuzz and tell it how to run the instrumented executable. AFL++ will not start without a directory with initial seed files and an output directory.

```
bash-5.1$ mkdir seeds out
bash-5.1$ echo 'a: ["valid", "yaml", 3]' > seeds/input
bash-5.1$ afl-fuzz -i seeds -o out -- ./util/parse
```

If the target executable gets its input via file(-name):

```
bash-5.1$ afl-fuzz -i seeds -o out -- ./util/parse @@
```

@@ will be replaced with a temp file containing the fuzzers input

Fuzzing in practice: Running AFL++

```
american fuzzy lop ++4.01a {default} (./util/parse) [fast]

process timing ----- overall results -----
  run time   : 0 days, 0 hrs, 4 min, 54 sec      cycles done : 0
  last new find : 0 days, 0 hrs, 0 min, 0 sec    corpus count : 508
last saved crash : none seen yet                saved crashes : 0
last saved hang  : none seen yet                saved hangs  : 0

cycle progress ----- map coverage -----
now processing   : 0.0 (0.0%)                    map density  : 16.77% / 32.81%
runs timed out  : 0 (0.00%)                     count coverage : 3.36 bits/tuple

stage progress ----- findings in depth -----
now trying      : splice 11                      favored items : 1 (0.20%)
stage execs     : 1652/2048 (80.66%)              new edges on  : 174 (34.25%)
total execs     : 22.4k                          total crashes : 0 (0 saved)
exec speed      : 72.11/sec (slow!)              total tmouts  : 1 (1 saved)

fuzzing strategy yields ----- item geometry -----
  bit flips    : disabled (default, enable with -D) levels      : 2
  byte flips   : disabled (default, enable with -D) pending     : 508
  arithmetics   : disabled (default, enable with -D) pend fav    : 1
  known ints    : disabled (default, enable with -D) own finds   : 507
  dictionary    : n/a                               imported    : 0
havoc/splice    : 482/16.4k, 4/256                  stability   : 100.00%
py/custom/rq    : unused, unused, unused, unused
trim/eff        : 0.00%/5, disabled

[cpu000: 83%]
```

How to fuzz a project

1. Identify a project  yaml-cpp
2. Prepare the fuzzing 
3. Run the Fuzzer 

Fuzzing Exercise due until 12.07.2022 23:49

Find the exercise sheet here: [IT-Security Exercise Fuzzing](#)

The sheet is designed around finding a real bug in a library that generates barcodes from text. Find a vulnerable version of the library “libzint” here: [GitHub - dende/zint](#)

A docker image with LLVM/clang and AFL++ already installed can be found here: [Docker Hub - hartlage/unibn-macs-itsec-fuzzing](#). Find an explanation on the sheet.

■ 5 tasks

- 1 Research task - name existing fuzzers. You could use resources provided in the slide
- 1 task to repeat instrumenting and compiling for LibFuzzer / AFL++
- 1 Task to instrument a real library for LibFuzzer and write a fuzztarget for it
- 1 Task to instrument the same library for AFL++
- 1 Task designed to prove that you found a bug in the library with fuzzing (either Libfuzzer or AFL++)
 - If you really hate fuzzing you can also do manual pentesting, find the bug and provide the solution

Fuzzing: Useful Links

- Contacting me:
 - c@dende.de
- General Fuzzing:
 - [Google's tutorials, examples, discussions, research proposals, and more related to fuzzing](#)
 - [Tut10-1: Fuzzing - CS6265: Information Security Lab](#)
 - [Google's OSS-Fuzz: Ideal Fuzz Target Integration](#)
 - [Fuzzing Survey - an overview over existing Fuzzing Frameworks](#)
- Libfuzzer:
 - [The LLVM LibFuzzer Documentation](#)
 - [Googles LibFuzzer Tutorial](#)
- AFL/AFL++:
 - [The AFL++ Documentation](#)
 - [AFL training by Michael Macnair](#)



Lightning
Surveys 

- Go is a compiled language.
- For Go \leq 1.17 go-fuzz-build, a third party tool had to be used for instrumentation
 - [GitHub - dvyukov/go-fuzz: Randomized testing for Go](#)
 - Fuzzing was performed the third party tool go-fuzz
- Go \geq 1.18 comes with a native fuzzing toolchain called gofuzz
- Here's the [Documentation](#) and a short [tutorial](#)

```
root@6dfd716a7df8:~/gofuzz-test# go test -fuzz=Fuzz
fuzz: elapsed: 0s, gathering baseline coverage: 0/33 completed
fuzz: elapsed: 0s, gathering baseline coverage: 33/33 completed, now fuzzing with 6 workers
fuzz: elapsed: 3s, execs: 55307 (18433/sec), new interesting: 4 (total: 37)
fuzz: elapsed: 6s, execs: 117223 (20640/sec), new interesting: 4 (total: 37)
```

The output of gofuzz, Go's native fuzzing framework

Fuzzing: a Go Fuzz Target

```
package fuzz
import (
    "bytes"
    "fmt"
    "github.com/filecoin-project/lotus/chain/types"
    "github.com/google/go-cmp/cmp"
    gfuzz "github.com/google/gofuzz"
)

func FuzzBlockMsg(data []byte) int {
    msg, err := types.DecodeBlockMsg(data)
    if err != nil {
        return 0
    }
    encodedMsg, err := msg.Serialize()
    if err != nil {
        panic(fmt.Sprintf("Error in serializing BlockMsg: %v", err))
    }
    // Checks if the encoded message is different to the fuzz data.
    if !bytes.Equal(encodedMsg, data) {
        panic(fmt.Sprintf("Fuzz data and serialized data are not equal: %v", err))
    }
    return 1
}
```

[filecoin-project/fuzzing-lotus · GitHub](https://github.com/filecoin-project/fuzzing-lotus)

Fuzzing: Java with Jazzer

- Java is a compiled language. Or is it?
 - Java can be considered both a compiled and an interpreted language, because the compiled byte-code runs on the Java Virtual Machine (JVM), a software based interpreter
- Jazzer is a coverage-guided, in-process fuzzer for the JVM platform developed by Code Intelligence
 - [Jazzer - Coverage-guided, in-process fuzzing for the JVM](#)
 - based on LibFuzzer
 - Joint effort with google to support Jazzer in OSS-Fuzz: [Google Online Security Blog: Fuzzing Java in OSS-Fuzz](#)

```
INFO: Loaded 1 hooks from com.example.ExampleFuzzerHooks
INFO: Instrumented com.example.ExampleFuzzer (took 81 ms, size +83%)
INFO: libFuzzer ignores flags that start with '--'
INFO: Seed: 2735196724
INFO: Loaded 1 modules (65536 inline 8-bit counters): 65536 [0xe387b0, 0xe487b0),
INFO: Loaded 1 PC tables (65536 PCs): 65536 [0x7f9353eff010,0x7f9353fff010),
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 94Mb
#1562   NEW    cov: 4 ft: 4 corp: 2/14b lim: 17 exec/s: 0 rss: 98Mb L: 13/13 MS: 5 CrossOver-ShuffleBytes-CMP- DE:
"magicsring4"
```

The output of Jazzer, Code Intelligence's JVM fuzzing framework

Fuzzing: Java with Jazzer

```
class ParserTests {  
    @Test  
    void unitTest() {  
        assertEquals("foobar", SomeScheme.decode(SomeScheme.encode("foobar")));  
    }  
  
    @FuzzTest  
    void fuzzTest(FuzzedDataProvider data) {  
        String input = data.consumeRemainingAsString();  
        assertEquals(input, SomeScheme.decode(SomeScheme.encode(input)));  
    }  
}
```

A simple property-based fuzz test in Java (<https://github.com/CodeIntelligenceTesting/jazzer>)