Proactive Computer Security

Fuzzing

Ken Friis Larsen
<kflarsen@diku.dk>

Department of Computer Science
University of Copenhagen

May 30, 2016

Many slides by Morten Brøns-Pedersen <f@ntast.dk>

Today's Program

- Introduction
- Fuzzing 101
- Whitebox vs Blackbox fuzzing
- Fuzzers, frameworks and tools
- Case studies
- This week's assignment

What is fuzzing?

- The programs you have been working on so far have been rather simple.
- Still, the exploits you have written are not completely unrealistic.
- That just means that finding the vulnerabilities in “real” software is much harder.
- *Fuzzing* is having a computer push all the buttons until something breaks. Then you go and see if the thing that broke is a vulnerability or just another bug. Rinse, repeat.

Question

How does fuzzing differ from unit testing?

History: The Monkey [Hertzfeld 11]

- One of the first examples of fuzzing (though the term was not coined at the time) was a testing tool developed by Steve Capps at Apple.
- Excerpt from “Revolution in the Valley”:

  > *The Monkey was a small desk accessory that used the journaling hooks to feed random events to the current application, so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon.*

- The thing about pushing buttons until something breaks wasn’t a joke.

History: The Fuzz Generator [Miller 88]

- The term “fuzz” was coined by Barton P. Miller in 1988. The term was used in a project assignment for the course “Advanced Operating Systems” that he was teaching at University of Wisconsin.
- Excerpt from the assignment:

  > *The Fuzz Generator: The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. [. . .] you will build a fuzz generator. This is a program that will output a random character stream.*

- In 1990 “An Empirical Study of the Reliability of UNIX Utilities”[Miller 90] was published, along with open source programs to perform *fuzz testing* (or *fuzzing* in today’s terms).

Fuzzing 101

Targets for fuzzing

In principle, everything that process or carry information can be fuzzed.

File formats Image viewers, word processors, music players, etc. need to parse a variety of different file formats, which can be fuzzed.

Networking protocols Servers (HTTP, FTP, . . .), chat clients, browsers, etc. speak one or more networking protocols, which can be fuzzed.

API's Libraries and frameworks expose API's, which can be fuzzed.

Internal program flow and/or state By acting as a debugger a fuzzing process can attach to another process and fuzz individual functions, write directly to memory, etc.

Other Hardware, thread scheduling (race conditions are nasty), drivers, etc.

Fuzzing Components

To do fuzzing we need two components:

- Data generation
- Crash detection (a monitor)

Data Generation

There are three (two, really) basic data generation strategies:

Random A sequence of random bytes is generated and given to the target program.

Mutation based A sample of real data (an image, a network traffic dump, etc.) is used as a template, and changed slightly between fuzzing runs.

Generation The fuzzer has semantic knowledge about the target, and can generate meaningful data. For example, a generational fuzzer can know that a some part of a file must be gzip'ed data.

Hybrid The fuzzer has semantic knowledge *and* it has a sample database. If the space is very large this may help to get better code coverage.

Question

Why is it usually more effective to generate data that is almost valid than using random data?

Mutation-based Fuzzing

Some common techniques for mutation-based fuzzing are:

- Bit flipping: Random chosen bits of the data are flipped.
- Prepending/Appending: New data is prepended/appended to the given data.
- Repeating: Random chosen parts of the given data are repeated.
- Removal: Random chosen parts of the given data are removed.

Block based data generation

- Many generational fuzzers define data generation in terms of the combination of *blocks*.
- A block is either a primitive supplied by the fuzzing framework, or the combination of other blocks.
- An example of a hybrid (and thus generational) fuzzer is Peach. Peach has many primitive blocks. Some of the most important are:

Raw data	Such as strings, numbers and raw bytes.
Relations	A block describing another block in some way, e.g. it's size, position, offset, etc.
Fixups	Blocks which are generated each run, typically from another block's data, e.g. checksums or random identifiers.
Transforms	A meta-block which transforms another block in some way, e.g. compressing it.

Block based data generation: example

Generate a HTTP GET request with Peach:

```
<DataModel name="HTTP">
  <String value="GET / HTTP/1.1\r\n" />
  <String value="Host" />
  <String value=": " token="true" />
  <String value="localhost" />
  <String value="Content-Length" />
  <String value=": " token="true" />
  <String>
    <Relation type="size" of="Body" />
  </String>
  <String value="\r\n" token="true" />
  <Blob name="Body" />
</DataModel>
```

State

- Interactive programs and networking programs almost always have some form of state.
- When a program is in a particular state it usually accepts data in a format specific to that state. For example, a HTTP server first parses the headers, and then the body.
- To generate (almost) valid data the fuzzer needs to know what state the target program is in.
- Fuzzers usually implement a state machine which emulate the target programs behaviour.
- State transitions can (and should) be fuzzed too.

Monitors

- When a bug (hopefully a vuln) is triggered in the target program we need to know about it.
- On Unix an easy way to look for crashes is to run

  ```
  ulimit -c unlimited
  ```

  before starting the target program. This will make the system produce a *core* file in the event of a crash. The core file holds information about the state of the process (memory, registers, etc.) when it crashed.
- After a crash the core file can be used to inspect the circumstances of the crash:

  ```
  gdb <target program> core
  ```

Advanced monitors

- Many fuzzers and fuzzing frameworks have more advanced monitors.
- One typical example where a more advanced monitor is needed is file format fuzzing; many programs that parse file formats don’t stop execution when they are done parsing. An image viewer is one example. When it is done parsing and showing the image it just sits there waiting for the user to do something.
- If the program gets to the inactive state without crashing, the fuzzing run has been unsuccessful, and the program can be terminated.

Advanced monitors

- A fuzzer for such a program must be able to detect when the target program is idling.
- On Linux, one option is to look in the file

  `/proc/<pid>/status`

  The second line gives the status of the program. In particular "`S (sleeping)`" shows that the program is currently doing nothing.
- This is not always reliable because the program may wake up from time to time to do other things. Or it might yield from time to time to let the system do other things even though the program is not idle.
  In these cases heuristics may be needed.

Blackbox vs Whitebox Fuzzing

Blackbox vs Whitebox Fuzzing

Blackbox

- We only have remote access to a network service
- We have a binary (which we may try to reverse)

Whitebox

- We have a binary (which we may try to instrument)
- We have the source code, which we can compile with instrumentation.

Whitebox Fuzzing

- Idea: mix fuzz testing with dynamic test generation
  1. Symbolic (or monitored) execution
  2. Collect constraints on inputs
  3. Negate those, solve with constraint solver, generate new inputs using DART (directed automated random testing)

SAGE: Whitebox Fuzzing [Godefroid 12]

- SAGE is a whitebox fuzzer for unmodified x86 (windows) binaries
- Start with a well-formed input (not random)
- Combine with a generational search (not DFS): negate collected constraints on a path one-by-one

Example Program

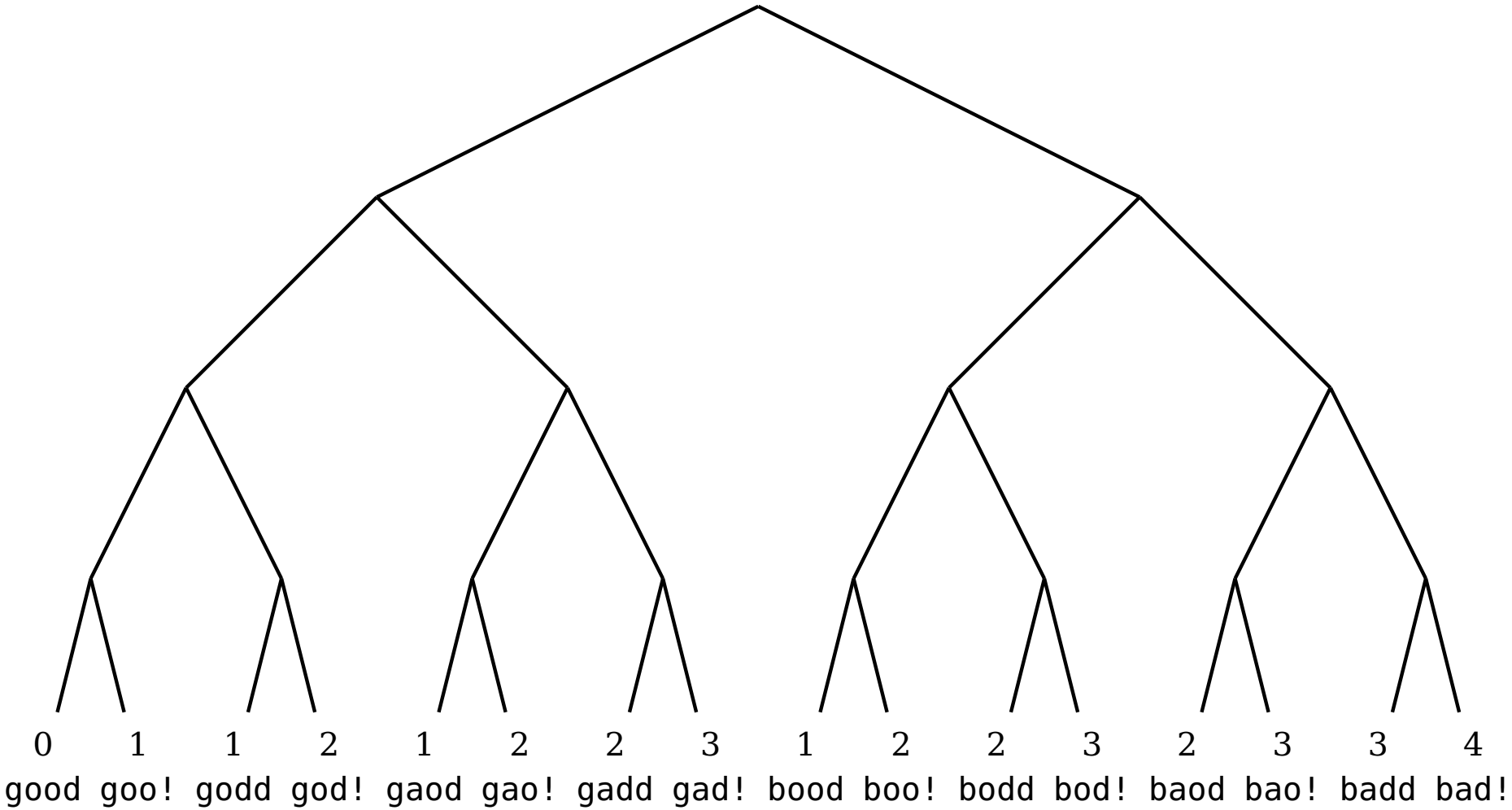
```
void tiptop(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

Input: "good"

Path constrains:

$$I_0 \neq \texttt{'b'}$$
$$I_1 \neq \texttt{'a'}$$
$$I_2 \neq \texttt{'d'}$$
$$I_3 \neq \texttt{'!'}$$

Search Space



SAGE Results

- Huge success internally at Microsoft
- Example: 1/3 of all security (internally) bugs on Win 7 was found by SAGE
- Running 24/7 on 100s of machines

Other Whitebox Fuzzers

- KLEE
- Angr

American Fuzzing Lob

- Mutation-based instrumentation-guided fuzzer
- Efficient compile-time, or binary-only, instrumentation
- Remarkably solid
- **Demo time**

Demo Overview

- Compile with instrumentation:

```
afl-gcc -std=c99 -o notbad notbad.c
```

- Set up directories

```
mkdir -p input && rm -rf output && mkdir -p output
```

- Create valid input

```
echo good > input/goodstuff
```

- Start fuzzing

```
afl-fuzz -i input -o output ./notbad
```

AFL Tips

- You might want to put your input and output directories on a RAM disk
- To read from a file rather than stdin, use `@@`

```
afl-fuzz -i input -o output ./notbadfile @@
```

- To fuzz network services you might want to try Preeny, see `https://github.com/zardus/preeny`.
- Make sure the read the AFL `README` carefully.
- Get AFL from `http://lcamtuf.coredump.cx/afl/`

Fuzzers, frameworks and tools

Fuzzer: zzuf

An easy to use mutation-based fuzzer. It works by intercepting file and network operations and changing random bits in the program's input. zzuf's behaviour is deterministic, making it easy to reproduce bugs.

Born out of the VLC project.

Get it from: `http://caca.zoy.org/wiki/zzuf` (also packed for most Linux distributions and for OS X via homebrew).

Fuzzer: Peach

Peach is a hybrid (both mutating and generational) fuzzer, providing many advanced features.

It uses an XML based format called Pit files to configure fuzzers. Pit files describe data generation, state models, monitors, etc.

It is possible to extend the functionality of Peach but that is only recommended for advanced users

Peach used to be written in Python, but as of version 3 it has changed to C#. It still runs under both Windows and Linux (using Mono).
Get it from: `http://peachfuzzer.com`

Fuzzing framework: Sully

Sully is a fuzzing framework written in Python. That means that the fuzzers you write with it will be written in python as well.

This makes it easier to bridge the gaps where the framework doesn't have that particular feature that you need.

Sully only supports writing generational fuzzers, but apart from that it has many of the same features as Peach, including state machines and monitors.

Sully is developed on Windows but runs on Unix as well, though maybe with a bit more quirks.
Get it from: `https://github.com/OpenRCE/sulley`

Fuzzing framework: Tavor

Tavor is a fuzzing framework written in Go. That means that the fuzzers you write with it will be written in Go as well. Alternatively, it can be used purely as a data generator with a bit of scripting.

Out of the box Tavor comes with its own format using a EBNF-like notation that allows you to define file formats, protocols, and other structured data without the need to write source code.

Get it from: `https://github.com/zimmski/tavor`

Fuzzer: Autodafé [Vuagnoux 06]

Autodafé is a block based generational fuzzer with a twist: Using markers and a tracer (a kind of debugger which monitor the target program) it is able to determine which parts of a protocol or file format is most likely to trigger the use of unsafe functions (`strcpy`, `gets`, etc.)

Using this knowledge it is able to fuzz more effectively.

Autodafé uses a custom description language to create fuzzers. Apart from the tracer it does not support monitors.

Get it from: `http://autodafe.sourceforge.net`

Fuzzer: Grinder

Grinder is a distributed fuzzer targeted at web browsers. It is designed to handle and categorize large amounts of crashes.

Get it from: `https://github.com/stephenfewer/grinder`

Framework: DynamoRIO

A framework for during runtime code manipulation and analysis.
Especially useful for in-memory fuzzing.

DynamoRIO uses a JIT-compiler to be able to control every instruction executed.

Get it from: `http://dynamorio.org`

Framework: Pin

Another framework for dynamic code manipulation. Like DynamoRIO, Pin also uses a JIT-compiler.

An example of a tool which uses Pin, which is relevant to next week's material, is ROPdefender[Davi 10].

ROPdefender uses Pin to mitigate ROP based exploits by halting the program on all `ret` instructions and checking the return address.

Pin is developed by Intel. Get it from: `http://pintool.org`

Framework and tools: Valgrind

Yet another JIT-based framework for dynamic instrumentation.

Tools included with Valgrind include memory leak detector, thread error detector, profiler, and call graph generator.

Get it from: `http://valgrind.org`

Case studies

PDF Fuzzing

Mateusz “j00ru” Jurczyk and Gynvael Coldwind from Google have found more than 50 vulnerabilities in Chrome’s PDF reader, and around 60 in Adobe Reader.

They used custom software to produce a large corpus of PDF files aimed at optimal code coverage.

Based on this corpus they wrote several fuzzers ranging from simple bit-flippers to fuzzers with semantic knowledge of the PDF format.

Read more: `http://j00ru.vexillium.org/?p=1507`

Flash Fuzzing

Chris Evans, Matt Moore and Tavis Ormandy, also from Google used the same technique to generate a corpus of Flash files.

Using this corpus and mutating fuzzers they were able to find more than 100 security bugs.

Read more:
`http://googleonlinesecurity.blogspot.ch/2011/08/fuzzing-at-scale.html`

USB Stick of Death

Gynvael Coldwind found a bug in `ntfs.sys` – Microsoft's NTFS filesystem – using a custom fuzzer. The bug is triggered by inserting a specially crafted USB stick into the target computer, which will happily mount the filesystem thus triggering the bug.

The bug was exploited by Mateusz "j00ru" Jurczyk, giving local privilege escalation.

Read more: `http://j00ru.vexillium.org/?p=1272`

Injecting SMS Messages into Smart Phones for Security Analysis

In 2009 Collin Mulliner and Charlie Miller used Sully to generate SMS messages and inject them into Android and iPhone phones.

They found that they where able to generate SMS messages that would crash the phone.

Read more: `https://www.usenix.org/legacy/event/woot09/tech/full_papers/mulliner.pdf` or `https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf`

This week’s assignment

Yet Another Internet Relay Chat

- This week your assignment is to fuzz a chat server and find a bug in it.
- While the chat server is obviously not as advanced as “real” software, we have tried to make it look as much like it as possible.
- It speaks a custom protocol, which requires you to hold some state while fuzzing.

Your fuzzer

As it says in the assignment, it is up to you if you want to use an existing fuzzer or framework, or write your own from scratch.

If you decide to write your own it can be as simple as this (Python code):

```
import os
while not os.path.isfile('core'):
    # your fuzzer here
```

Take a look at `fuzzer.py` for a starting point for the `your fuzzer here` -part.

Starting the server

Open a new terminal for the server to run in, and navigate to its folder.

Remember to enable core dumps:

```
ulimit -c unlimited
```

Then start the server:

```
./yairc
```

Connecting to the server

Open another terminal for the client.

Connect to the server using netcat:

```
nc localhost 4242
```

Talk to the server:

```
> HELO SERVER
  HELO CLIENT
> NICK br0ns
  NICK OK (br0ns)
> JOINING #pcs2016
  TOKEN plyw00dhe4dywh41in9tran5mis5ion
  JOIN plyw00dhe4dywh41in9tran5mis5ion 2016-05-25 23:05:33 br0ns
> MSG plyw00dhe4dywh41in9tran5mis5ion a/s/l?
  MSG plyw00dhe4dywh41in9tran5mis5ion 2016-05-25 23:06:00 br0ns a/s/l?
```

Attaching a debugger

The server forks when it receives a connection, so after connecting you will see a new process in the process list:

```
> ps f | grep yairc
  11832 pts/4    S+     0:00  \_ ./yairc
  11937 pts/4    S+     0:00      \_ ./yairc
```

Here we are interested in the process with the PID 11937. Use GDB to attach to it:

```
gdb yairc 11937
```

Then continue the process:

```
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Attaching a debugger: Tip

- If you are writing an exploit and need to debug it, you can make it wait just after it connects, but before it sends any data. When it pauses you can attach a debugger, and then resume the exploit. This will let you continue the process in your debugger just after the first call to `recv(2)`.
- When the process continues you will be deep inside a library call. Type `finish` a few times to return to the program, or place a breakpoint and run to it. Notice that the breakpoint must to be placed *after* the first `recv(2)` call.

Attaching a debugger: Tip

In Python:

```
import socket

sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
sock.connect((host, port))
raw_input('Hit <enter>') # pause exploit
sock.send('...')
```

Arbitrary code execution

- Just because the assignment only calls for a crash it doesn't mean you shouldn't write an exploit.
- It is possible to gain arbitrary code execution. The server is running on `163.172.148.91`. Go steal our files!
- In order to use arbitrary code execution you must know about networking shellcode. This is shellcode that gives you control across the network. We'll put up a short note on Absalon.

Summary

- Fuzzing is about generating random input to programs to try to crash them.
- To generate data we either mutate known valid input, or we generate it from some description of the input.
- Blackbox fuzzing is lightweight, easy to implement, and fast. But may give poor coverage.
- Whitebox fuzzing gives better coverage, but is complex to implement and slower.
- **Remember to participate in the course and teachers evaluations**

References I

Lucas Davi, Ahmad-Reza Sadeghi & Marcel Winandy.
ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks.
Rapport technique, Horst Görtz Institute for IT Security, Ruhr-University Bochum, March 2010.

Patrice Godefroid, Michael Y. Levin & David Molnar.
SAGE: Whitebox Fuzzing for Security Testing.
Queue, vol. 10, no. 1, pages 20:20–20:27, January 2012.
Available from: `http://doi.acm.org/10.1145/2090147.2094081`.

Andy Hertzfeld.
Revolution in the valley: The insanely great story of how the mac was made.
O'Reilly Media, 2011.

References II

Barton P. Miller.
CS736: Advanced Operating Systems, Project List.
1988.
Available from: `http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf` [cited 2 June 2013].

Barton P. Miller, Louis Fredriksen & Bryan So.
An empirical study of the reliability of UNIX utilities.
Commun. ACM, vol. 33, no. 12, pages 32–44, December 1990.
Available from: `http://doi.acm.org/10.1145/96267.96279`.

Martin Vuagnoux.
Autodafé: an act of software torture.
Rapport technique, Swiss Federal Institute of Technology (EPFL), Cryptograhy and Security Laboratory (LASEC, 2006.