

# Software Testing, Quality Assurance & Maintenance—Lecture 7

Patrick Lam  
University of Waterloo

January 26, 2026

The background image shows a dense, misty forest. The trees are tall and thin, heavily covered in bright green moss. Sunlight filters through the canopy, creating a dappled light effect. Ferns and other greenery are visible at the base of the trees.

# Part I

# Fuzzing

# Some JavaScript Code

```
function test() {  
    var f = function g() {  
        if (this != 10) f();  
    };  
    var a = f();  
}  
test();
```

## Huh?

- this code used to crash WebKit  
([https://bugs.webkit.org/show\\_bug.cgi?id=116853](https://bugs.webkit.org/show_bug.cgi?id=116853)).
- automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing effectively finds software bugs, especially security-based bugs (e.g. insufficient input validation.)

# Fuzzing Origin Story

- 1988.
- Prof. Barton Miller was using a modem,  
on a dark and stormy night.
- Line noise caused UNIX utilities to crash!

## Fuzzing Origin Story Part 2

- he got grad students in his Advanced Operating Systems class to write a fuzzer  
(generating unstructured ASCII random inputs)
- result: 25%-33% of UNIX utilities crashed on random inputs<sup>1</sup>

---

<sup>1</sup><http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html>

## (an earlier use of Fuzzing)

- 1983: Apple's "The Monkey"<sup>2</sup>
- Generated random events for MacPaint, MacWrite.
- Found lots of bugs,  
but eventually the monkey hit the Quit command.
- Solution: "MonkeyLives" system flag, ignore Quit.

---

<sup>2</sup>[http://www.folklore.org/StoryView.py?story=Monkey\\_Lives.txt](http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt)

## Experience report: Fuzzinator author

*More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while.*

*As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore.*

*At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.<sup>3</sup>*

---

<sup>3</sup><http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded>

# How Fuzzing Works

Two kinds of fuzzing:

- **mutation-based**: start with existing tests, randomly modify
- **generation-based**: start with grammar, generate inputs

What you do:

- feed randomly-generated inputs to the program;
- look for crashes or assertion errors;
- or run under a dynamic analysis tool (e.g. Valgrind) and observe runtime errors (implicit oracles).

## Level 0 Fuzzing

Generation-based testing for HTML5.

Use the regular expression:

. \*

that is: “any character”, “0 or more times”.

Found a WebKit assertion failure:

[https://bugs.webkit.org/show\\_bug.cgi?id=132179](https://bugs.webkit.org/show_bug.cgi?id=132179).

Process:

- Take the regular expression and generate random strings from it.
- Feed them to the browser and see what happens.
- Find an assertion failure/crash.

## Worked example: fuzzing bc

bc: one of the UNIX calculator utilities.

```
plam@poneke ~> bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
scale=2
4+6+5
15
6*5/3
10.00
4^5/7
146.28
quit
plam@poneke ~> □
```

# Generating random input

```
import random

def fuzzer(max_length: int = 100, char_start: int = 32,
           char_range: int = 32) -> str:
    """A string of up to `max_length` characters
    in the range [ `char_start` , `char_start` + `char_range` ) """
    string_length = random.randrange(0, max_length + 1)
    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range))
    return out
```

I've created a wrapper,  
code/L07/run\_fuzzer.py, that you can just  
run; there are also wrappers for the  
subsequent runs too.

## Roundtripping from disk: roundtrip\_to\_disk.py

```
from fuzzer import *

import os
import tempfile
import random

basename = "input.txt"
tempdir = tempfile.mkdtemp()
FILE = os.path.join(tempdir, basename)
print(FILE)

data = fuzzer()
with open(FILE, "w") as f:
    f.write(data)

contents = open(FILE).read()
print(contents)
assert(contents == data)
```

This writes fuzzer-created input to disk, reads it back, and compares.

(\*Secure temp file creation must be in one step like this.)

## Running bc once: run\_bc\_once.py

```
import os
import tempfile
import subprocess

program = "bc"
basename = "input.txt"
tempdir = tempfile.mkdtemp()
FILE = os.path.join(tempdir, basename)
print(FILE)
with open(FILE, "w") as f:
    f.write("2+2\n")
result = subprocess.run([program, FILE],
                      stdin=subprocess.DEVNULL,
                      stdout=subprocess.PIPE,
                      stderr=subprocess.PIPE,
                      universal_newlines=True) # Will be "text" in Python

print(result.stdout)
print(result.returncode)
```

# Our Very Own Fuzzing Campaign: fuzzing\_campaign.py

```
from fuzzer import *
import os
import tempfile
import subprocess

def fuzzing_campaign():
    trials = 100
    program = "bc"
    runs = []
    basename = "input.txt"
    tempdir = tempfile.mkdtemp()
    FILE = os.path.join(tempdir, basename)

    for i in range(trials):
        data = fuzzer()
        with open(FILE, "w") as f:
            f.write(data)
        result = subprocess.run([program, FILE],
                               stdin=subprocess.DEVNULL,
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE,
                               universal_newlines=True)
        runs.append((data, result))

    return runs
```

## Queries: run\_fuzzing\_campaign.py

```
from fuzzing_campaign import *

if __name__ == "__main__":
    runs = fuzzing_campaign()

    # how many runs don't result in error messages?
    count_no_errors = sum(1 for (data, result) in runs if result.stderr == "")
    print ("#_runs:{}".format(len(runs)))
    print ("#_no-errors:{}".format(count_no_errors))

    # let's look at the first error message
    errors = [(data, result) for (data, result) in runs if result.stderr != ""]
    (first_data, first_result) = errors[0]

    print ("first_error_input:{}".format(repr(first_data)))
    print ("stderr_output_for_first_error_input:{}".format(first_result.stderr))

    # are there any non-error outputs?
    non_errors = [result.stdout for (data, result) in runs if
        "illegal_character" not in result.stderr
        and "parse_error" not in result.stderr
        and "syntax_error" not in result.stderr]
    print ("stdout_on_non-error_runs:{}".format(non_errors))
```

# More on crashing: buffer overflows

We saw this recently:

```
plam@poneke ~/c/s/n/c/L05 (main)> cat segfault.c
#include <stdlib.h>

int main()
{
    int * q = malloc(5*sizeof(int));
    q[2000000] = 4;
}
plam@poneke ~/c/s/n/c/L05 (main)> gcc segfault.c -o segfault
plam@poneke ~/c/s/n/c/L05 (main)> ./segfault
fish: Job 5, './segfault' terminated by signal SIGSEGV (Address boundary error)
plam@poneke ~/c/s/n/c/L05 (main) [SIGSEGV]> █
```

That's bad. So is this:

```
// input = "Wednesday"
char weekday[9];
strcpy(weekday, input);
```

Buffer overflows are always wrong.

## Choose your language

C/C++ are rife with buffer overflows.

Safe Rust, Python don't have buffer overflows.

They will fail fast instead.

## Failing to check errors

```
while (getchar() != '■') /* ... */ ;
```

Turns out that `getchar()` might return EOF—especially likely for fuzz-generated inputs.

If so, this code causes an infinite loop;  
can detect with timeouts.

Not just a C problem.

## Other unexpected unsanitized inputs

```
char *read_input() {
    size_t size = read_buffer_size(); //<--
    char *buffer = (char *)malloc(size);
    // fill buffer
    return (buffer);
}
```

size may not be a valid size:

too big, too small;

conceptually, negative (but `size_t` can't be).

## Fuzzing Summary

Fuzzing finds interesting test cases.

Works best at interfaces between components  
(including system/user interface).

Advantages: it runs automatically and really works.

Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

# Part II

## Address Sanitizer

## Buffer Overflow On Demand

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv) {
    /* Create an array with 100 bytes, initialized with 42 */
    char *buf = malloc(100);
    memset(buf, 42, 100);

    /* Read the N-th element, with N being the first argument */
    int index = atoi(argv[1]);
    char val = buf[index];

    /* Clean up memory so we don't leak */
    free(buf);
    return val;
}
```

## About that `free()`

In this context, `buf` would be freed on exit anyway.

Relying on free-upon-exit is less tidy, and it would be a problem in `not-main()`.

# The Demand

```
$ clang -fsanitize=address -g buffer-overflow-on-demand.c
$ ./a.out 5000
=====
==638062==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7bbdd
READ of size 1 at 0x7c21313e13c8 thread T0
#0 0x55e13d86cb4f in main /home/plam/courses/stqam-2026-working-notes/
#1 0x7f7132029f74  (/usr/lib/x86_64-linux-gnu/libc.so.6+0x29f74) (Build
#2 0x7f713202a026 in __libc_start_main (/usr/lib/x86_64-linux-gnu/libc.
#3 0x55e13d783360 in _start (/home/plam/courses/stqam-2026-working-not
```

Address 0x7c21313e13c8 is a wild pointer inside of access range of size 0x
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/plam/courses/stqam-2
Shadow bytes around the buggy address:

```
0x7c21313e1100: fa fa
0x7c21313e1180: fa fa
0x7c21313e1200: fa fa
0x7c21313e1280: fa fa
0x7c21313e1300: fa fa
=>0x7c21313e1380: fa fa fa fa fa fa fa fa[fa]fa fa fa fa fa fa fa fa
0x7c21313e1400: fa fa
0x7c21313e1480: fa fa
0x7c21313e1500: fa fa
0x7c21313e1580: fa fa
0x7c21313e1600: fa fa
```

## Fuzzing & AddressSanitizer



Wikimedia Commons, Evan-Amos, public domain

Two Great Tastes...

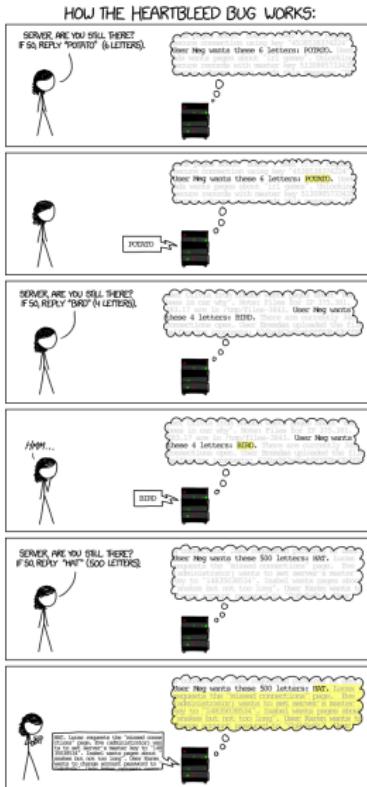
## Fuzzing & AddressSanitizer

Fuzzing causes (memory and other) problems;  
AddressSanitizer detects them.

The program runs like  $2\times$  slower in  
AddressSanitizer.

Or, choose Valgrind: find more bugs, program  
runs like  $100\times$  slower.

## Found w/sanitizer: Heartbleed (thx xkcd)



## How Found?

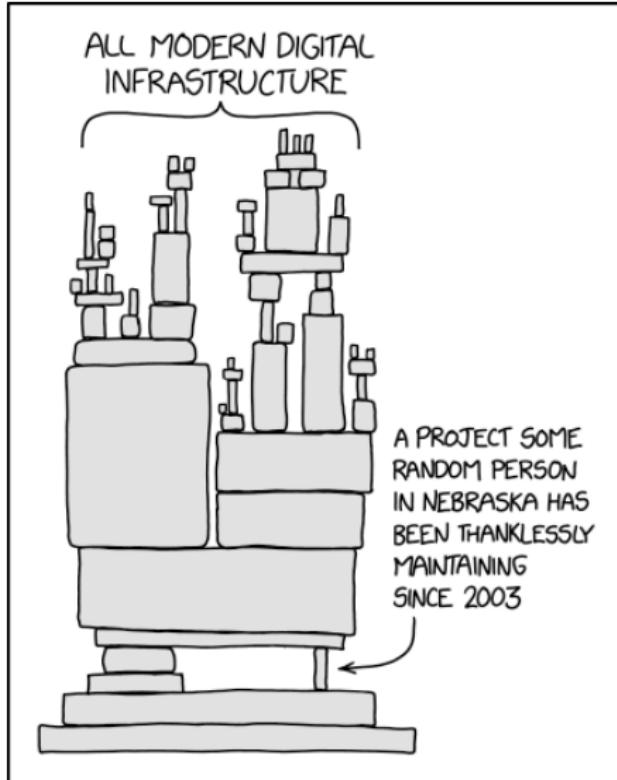
Researchers at Google and Codenomicon found Heartbleed via memory sanitizer.

Fed fuzzed inputs to OpenSSL.

Sanitizer found illegal accesses; easy to fix.

(Note: use responsible disclosure if you find something.)

## Side note: on Open Source



but note also <https://cryptography.io/en/latest/statements/state-of-openssl/>