Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 14 — February 27, 2026

Patrick Lam *version 1*

We are now going to spend 3 weeks on fuzzing. This is a technique that works well in practice today, and is particularly useful for discovering security bugs, in conjunction with sanitizers. It tends to find bugs caused, for instance, by a lack of sufficient input validation.

Introduction: Fuzzing

Consider the following JavaScript code[1].

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

Turns out that it can crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`). Plus, it was automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Origin Story. It starts with line noise. In 1988, Prof. Barton Miller was using a 1200-baud dialup modem to communicate with a UNIX system on a dark and stormy night. He found that the random characters inserted by the noisy line would cause his UNIX utilities to crash. He then challenged graduate students in his Advanced Operating Systems class to write a fuzzer—a program which would generate (unstructured ASCII) random inputs for other programs. The result: the students observed that 25%-33% of UNIX utilties crashed on random inputs[2].

(That was not the earliest known example of fuzz testing. Apple implemented "The Monkey" in 1983[3] to generate random events for MacPaint and MacWrite. It found lots of bugs. The limiting factor was that eventually the monkey would hit the Quit command. The solution was to introduce a system flag, "MonkeyLives", and have MacPaint and MacWrite ignore the quit command if MonkeyLives was true.)

How Fuzzing Works. Two kinds of fuzzing: *mutation-based* and *generation-based*. Mutation-based testing starts with existing test cases and randomly modifies them to explore new behaviours. Generation-based testing starts with a grammar and generates inputs that match the grammar.

[1] `http://webkit.sed.hu/blog/20130710/fuzzinator-mutation-and-generation-based-browser-fuzzer`
[2] `http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`
[3] `http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt`

In fuzzing, you feed generated inputs to the program and find crashes, or assertion failures, or you run the program under a dynamic analysis tool such as Valgrind and observe runtime errors.

The Simplest Thing That Could Possibly Work. Consider generation-based testing for HTML5. The simplest grammar—actually a regular expression—that could possibly work[4] is `.*`, where `.` is "any character" and `*` means "0 or more". Indeed, that grammar found the following WebKit assertion failure: `https://bugs.webkit.org/show_bug.cgi?id=132179`.

The process is as described previously. 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 UNIX utility bc

We can demonstrate "any character"-style generation with the UNIX command `bc`. These days, you're unlikely to find anything that way, but we can still demonstrate it. Here is some code that generates some number (default 100) of ASCII characters (between 32 and 64):

```
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
```

We can write it to a temp file. As an example of using assertions, let's read it back and check that we got back the same thing we wrote. This depends on having the `fuzzer()` function, defined above, available.

```
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)
```

[4] `http://trevorjim.com/a-grammar-for-html5/`

```

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

(By the way, this one-step call to `mkdtemp()` is the safe way to create a temporary directory, with `mkdtemp()`[5]. If you create a temporary file or directory in two steps—requesting a name and then creating a file with that name—that can be exploited by an attacker in another process.)

We can run a subprocess in Python, e.g. `bc`, like so:

```
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 3.7

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

This calls `bc` with input “2 + 2”. We expect output “4”.

And we can run `bc` repeatedly, with output from `fuzzer()`. This is a pretty small-scale 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)
```

[5] https://cwe.mitre.org/data/definitions/377.html

```
    result = subprocess.run([program, FILE],
                            stdin=subprocess.DEVNULL,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            universal_newlines=True)  # Will be "text" in Python 3.7
    runs.append((data, result))
```

We can inspect the outcome in various ways:

```
count_no_errors = sum(1 for (data, result) in runs if result.stderr == "")
print (count_no_errors)

errors = [(data, result) for (data, result) in runs if result.stderr != ""]
(first_data, first_result) = errors[0]

print(repr(first_data))
print(first_result.stderr)

print (sum(1 for (data, result) in runs if result.returncode != 0))
```

Causes of problems. Programs should never crash; ideally they might fail in a controlled way (with an error message), and best yet, they produce the right answer. Originally, Miller and his students did find a lot of crashes. C doesn't help. Let's talk about crashes some more.

Consider the following C code and the input "Wednesday":

```
  char weekday[9]; // 8 characters + trailing '\0' terminator
  strcpy (weekday, input);
```

What happens? How bad is it? It turns out that *buffer overflows* are attack vectors, and can be used by attackers to gain control of the executing user's account.

Python doesn't really have buffer overflows, because it checks buffer length and fails fast instead. Rust also has fewer buffer overflows than C++, according to Google[6]. (There still can be buffer overflows in unsafe Rust.)

Another bad thing in C comes from not doing error checks.

```
  while (getchar() != ' ');
```

If there is an end-of-file on standard input, then `getchar()` returns the value `EOF`, which is never a space. This implies an *infinite loop*. This sort of infinite loop is also possible in other languages, and one can use a timeout to guess whether or not the code is looping.

Unexpected unsanitized inputs can also cause havoc:

[6] `https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html`

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

The function `read_buffer_size()` might return something that is *not a valid size*: the number of requested bytes may be too big; or it might be smaller than the number of bytes that actually get filled (another buffer overflow). The Fuzzing Book talks about `size` being negative, but that's not allowed by spec, since `size_t` is an unsigned type.

Finding problems. As I've explicitly said, crashes are always bad. I alluded to sanitizers above also. Let's talk about them in more detail.

Here is a C program that can trigger a buffer overflow on demand.

```c
#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 command-line
        argument */
    int index = atoi(argv[1]);
    char val = buf[index];

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

(How nice of it to clean up memory; it's fine to rely on `buf` being freed upon exit, but less tidy, and potentially problematic if it was instead a function that could be called arbitrarily many times.)

If you compile it like this:

```
clang -fsanitize=address -g buffer-overflow.c
```

and you invoke it with a sufficiently large argument, then you'll get an AddressSanitizer error telling you the program did something bad.

Combine this with fuzzing and you can detect some memory errors in programs. The program will run like 2× slower with AddressSanitizer. Another option is valgrind, which is more picky but also runs like 100× slower.

A problem found using AddressSanitizer: Heartbleed. xkcd has a good explainer (in pictures): `https://xkcd.com/1354/`

In words: the OpenSSL implementation allowed the client to say how many bytes it was supposed to return when requesting a heartbeat. The number of bytes asked-for could exceed the number of bytes specified, thus returning bytes to the client (or server) that it shouldn't be allowed to see. There is also a good Wikipedia page: `https://en.wikipedia.org/wiki/Heartbleed`.

We're talking about this now because researchers at Google and Codenomicon discovered Heartbleed using a memory sanitizer, feeding fuzzed inputs to OpenSSL. The sanitizer found the illegal accesses, which then were easy enough to fix (although when something like this turns out, it's important to follow responsible disclosure practices.)

Fuzzing Summary. Fuzzing is a useful technique for finding interesting test cases. It works best at interfaces between components. Advantages: it runs automatically and really works. Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

How Address Sanitizer Works

It's good to know how the tools that you use work. So, we'll look into Address Sanitizer[7], which can be used with clang and gcc to provide compile-time instrumentation using a small (5kLOC) library for x86 and x86_64 on Linux, Mac, and Windows. This tool has found thousands of bugs in production code. Consider this C function.

```
void foo() {
  int *x = malloc (10*sizeof(int));
  int *y = malloc (5*sizeof(int));

  y[0] = x[12];
}
```

What happens when you call it?

Probably, in practice, you observe nothing weird. The array that `x` points to might have extra bytes allocated at the end, or maybe `x[12]` could happen to point to the memory allocated for `x`, and the program continues to execute. `y[0]` gets some value.

But this is undefined behaviour. You're not supposed to write this code, per the C standard: you're not allowed to read past the end of the allocation. C says that anything is allowed to happen after calling `foo()`. Could be a crash, could be nothing, could get an arbitrary value for `y[0]`.

Can we detect this undefined behaviour by writing a test case? I wouldn't bet on it.

Demo. From the Internet:

`https://github.com/dutor/asan-demo`

[7]More information: `https://llvm.org/devmtg/2011-11/Serebryany_FindingRacesMemoryErrors.pdf`

Instrumenting the code. It's a compiler, right, so it can generate whatever code it wants. Address Sanitizer generates instrumented code at loads and stores, and tracks meta-information about memory.

```
*addr = e
```

```
if (IsPoisoned(addr))
  ReportError(addr, sz, true);
*addr = e;
```

```
e = *addr
```

```
if (IsPoisoned(addr))
  ReportError(addr, sz, false);
e = *addr;
```

How do these functions work? How to make them fast?

Memory Layout. We make two disjoint areas of memory: `Mem` and `Shadow`. `Mem` is normal memory. `Shadow` tracks meta-data about the memory; put information about address `addr` in `Mem` at location `MemToShadow(addr)` in `Shadow`. We need `MemToShadow` to be a fast computation.

The shadow memory may then say that `addr` is normal memory, or it may be poisoned in some way.

In a bit more detail, here's what an access could look like:

```
shadow_addr = MemToShadow(addr);
if (ShadowIsPoisoned(shadow_addr)) {
  ReportError(addr, sz, kIsWrite);
}
```

You've seen the concept of memory alignment in ECE 222. For our purposes, we'll assume that memory is allocated such that its address is always divisible by 8 (QWORD-aligned). So, we can use one byte of `Shadow` for one QWORD of `Mem`, to store the poisoning state of that byte.

There are a lot of poisoning states in the actual AddressSanitizer implementation, but we'll talk about the simple case. Here are the possibilities:

- all 8 bytes are accessible (not poisoned); shadow value 0.
- all 8 bytes are inaccessible (poisoned); shadow value < 0 (negative).
- first k bytes are accessible, the next $8 - k$ bytes are not, where $0 < k < 8$; shadow value is k.

These possibilities are exhaustive due to allocations being aligned at QWORD boundaries; that is, it's impossible that an allocation starts in the middle of a QWORD, or that there is something like 11001111 where 1 is accessible and 0 is not. However, Address Sanitizer may store other metadata in addition to accessibility.

Let's look at some implementation:

```
byte *shadow_addr = MemToShadow(addr);
byte shadow_value = *shadow_addr;
if (shadow_value < 0) { ReportError(addr, sz, kIsWrite); }
else if (shadow_value) {
  if (SlowPathCheck(shadow_value, addr, sz)) {
    ReportError(addr, sz, kIsWrite);
  }
}

bool SlowPathCheck(shadow_value, addr, sz) {
  last_accessed_byte = (addr + sz - 1) % 8;
  return (last_accessed_byte >= shadow_value);
}
```

Using bit magic, the compiler will automatically transform

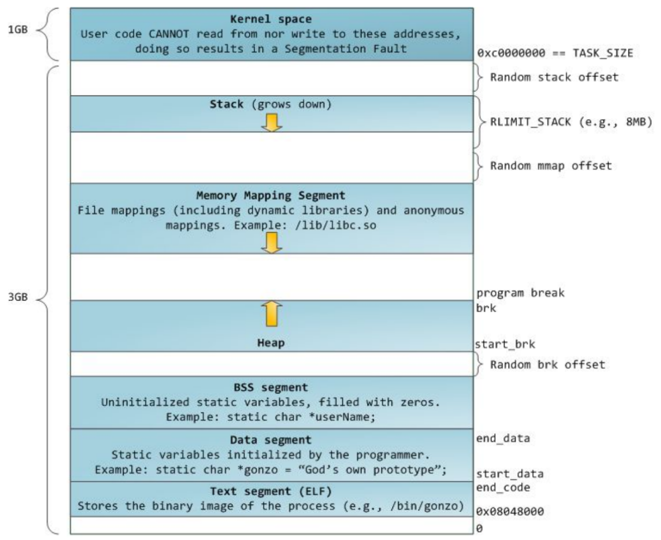
```
last_accessed_byte = (addr + sz - 1) % 8;
```

into

```
last_accessed_byte = (addr & 7) +  (sz - 1)) & 7;
```

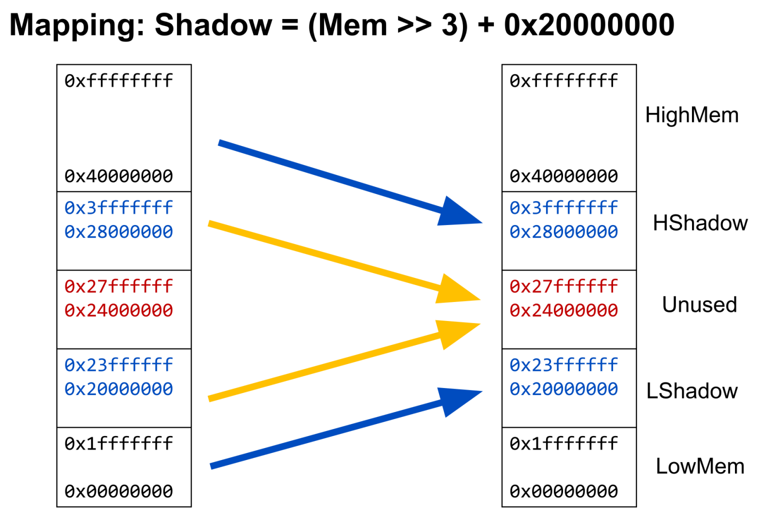
because bitwise AND is faster than modulo; convince yourself that the transformation is correct.

OK, but what does `MemToShadow()` do? Here's a picture of address space layout for a 32-bit process in Linux.


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

There's room to put shadow memory in the middle there. In particular, we can do this:



It's then a quick calculation to convert a main-memory address to the corresponding location in shadow memory, that is:

```
byte *shadow_addr = addr >> 3 + 0x20000000;
byte shadow_value = *shadow_addr;
if (shadow_value < 0) { ReportError(addr, sz, kIsWrite); }
else if (shadow_value) {
  if (SlowPathCheck(shadow_value, addr, sz)) {
    ReportError(addr, sz, kIsWrite);
  }
}

bool SlowPathCheck(shadow_value, addr, sz) {
  last_accessed_byte = ((addr & 7) + (sz - 1)) & 7;
  return (last_accessed_byte >= shadow_value);
}
```

But remember our original example, with the allocation of 10 ints perhaps next to 5 ints. The read from `x[12]` might go right into valid memory allocated for `y`. One way is to remember the size of `x` and do a bounds check. But Address Sanitizerr instead uses redzones around allocated memory. It's not perfect (you can overshoot), but it's faster.

Replace calls to `malloc()` by calls to `__asan_malloc()`. Here's a schematic implementation of that function:

```
void *__asan_malloc(size_t sz) {
  void *rz = malloc(RED_SZ);
  Poison(rz, RED_SZ);

  void *addr = malloc(sz); // assuming sequential allocation
  UnPoison(addr, sz);

  rz = malloc(RED_SZ);
  Poison(rz, RED_SZ);
```

```
    return addr;
  }
```

You can run a program compiled with Address Sanitizer that does an out-of-bounds access and it'll show the memory as well as the redzones surrounding the memory.

The Stack. OK, that works for heap allocations. Stack memory is not a result of `malloc()` calls and can't be intercepted that way. But, we still can use the compiler to rewrite the code:

```
void foo() {
  char redzone1[32];  // assuming QWORD alignment
  char a[8];          // another QWORD start
  char redzone2[24];
  char redzone3[32];  // third QWORD start
  int * shadow_base = MemToShadow(redzone1);
  shadow_base[0] = 0xffffffff; // poison redzone1
  shadow_base[1] = 0xffffff00; // poison redzone2 and unpoison 'a'
  shadow_base[2] = 0xffffffff; // poison redzone3

  // ...

  shadow_base[0] = shadow_base[1] = shadow_base[2] = 0; //
      unpoison all
  return;
}
```

The slides also show x86 assembly from compiling these functions:

```
long load8(long *a) { return *a; }
int load4(int *a) { return *a; }
```

but I won't include that in these notes; use `godbolt.org` to see for yourself.

Other sanitizers. It's not just AddressSanitizer. There are other dynamic analyses that you can compile into your program.

- ThreadSafetySanitizers: detect race conditions, i.e. accesses to the same memory by two threads where one access is a write, unprotected by a lock.
- MemorySanitizer: detects uninitialized reads (at 3× slowdown); requires all code to be instrumented.
- Undefined Behavior Sanitizer (ubsan): detects many types of undefined behaviour e.g. signed integer overflow, null pointer dereferences, etc.
- DataFlowSanitizer: lets you write your own data-flow dynamic sanitizers
- LeakSanitizer: detects memory leaks; no performance overhead.

Valgrind. An alternatve to Address Sanitizer is Valgrind. It is basically a just-in-time compiler from x86 to x86, and returns more detailed information than asan, but runs more slowly. It does not require compile-time instrumentation—it can just run code. We said that Address Sanitizer marks up shadow memory. Valgrind instead remembers the size of allocations and where they come from. Running programs compiled with debug information yields better errors.