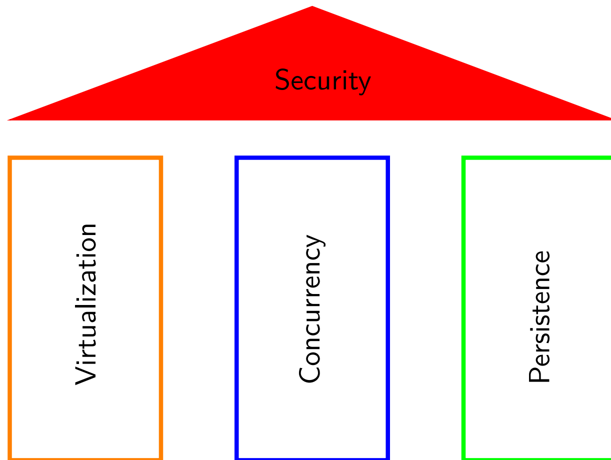
CS323 Operating Systems

Testing (Fuzzing/Sanitization)

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Security



Topics covered in this lecture

- Software testing
- Fuzzing
- Sanitization

This slide deck covers chapter 5.3 in SS3P.

Why testing?

Testing is the process of ***executing code*** *to* ***find errors****.*

An error is a deviation between observed behavior and specified behavior, i.e., a violation of the underlying specification:

- Functional requirements (features a, b, c)
- Operational requirements (performance, usability)
- Security requirements?

Limitations of testing

> *Testing can only show the presence of bugs, never their absence. (Edsger W. Dijkstra)*

A successful test finds a deviation. Testing is a form of dynamic analysis. Code is executed, the testing environment observes the behavior of the code, detecting violations.

- Key advantage: reproducible, generally testing gives you the concrete input for failed test cases.
- Key disadvantage: complete testing of all control-flow/data-flow paths reduces to the halting problem, in practice, testing is hindered due to state explosion.

Forms of testing

- Manual testing
- Fuzz testing
- Symbolic and concolic testing

Forms of testing

- Manual testing
- Fuzz testing
- Symbolic and concolic testing

We focus on *security* testing or testing to find *security* bugs, i.e., bugs that are reachable through attacker-controlled inputs.

Recommended reading: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World

Coverage as completeness metric

Intuition: A software flaw is only detected if the flawed statement is executed. Effectiveness of test suite therefore depends on how many statements are executed.

How to measure code coverage?

Several approaches exist, all rely on instrumentation:

- gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- SanitizerCoverage:
  https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

Sampling may reduce collection cost at slight loss of precision.

Fuzzing

Fuzz testing (fuzzing) is an automated software testing technique. Key idea: execute the target program with an input and check if it crashes. The fuzzing engine automatically generates new inputs based on some criteria:

- Random mutation
- Leveraging input structure
- Leveraging program structure

The inputs are then run on the test program and, if it crashes, a crash report is generated.

Fuzzing effectiveness

- Fuzzing finds bugs effectively (CVEs—unique bug numbers)
- Proactive defense during software development/testing
- Preparing offense, as part of exploit development

Fuzz input generation

Fuzzers generate new input based on generations or mutations.

- ***Generation-based*** input generation produces new input seeds in each round, independent from each other.
- ***Mutation-based*** input generation leverages existing inputs and modifies them based on feedback from previous rounds.

Fuzz input structure awareness

Programs accept some form of *input/output*. Generally, the input/output is structured and follows some form of protocol.

- ***Dumb fuzzing*** is unaware of the underlying structure.
- ***Smart fuzzing*** is aware of the protocol and modifies the input accordingly.

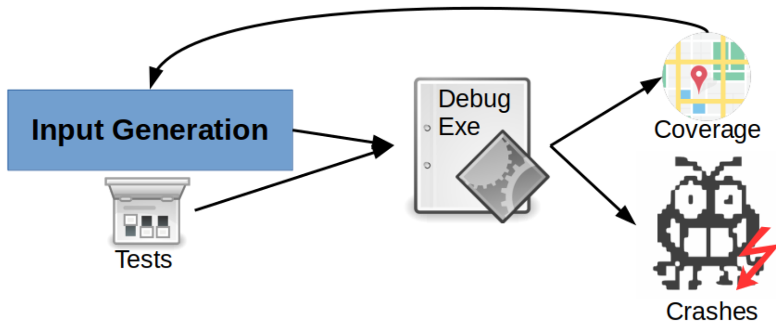
Example: a checksum at the end of the input. A dumb fuzzer will likely fail the checksum.

Fuzz program structure awareness

Input is processed by the program, based on the *program structure* (past executions), input can be adapted to trigger new conditions.

- ***White-box*** fuzzing leverages (expensive) semantic program analysis to mutate input; often does not scale
- ***Grey-box*** leverages program instrumentation based on previous inputs; light runtime cost, scales to large programs
- ***Black-box*** fuzzing is unaware of the program structure; often cannot explore beyond simple/early functionality

Coverage-guided grey-box fuzzing



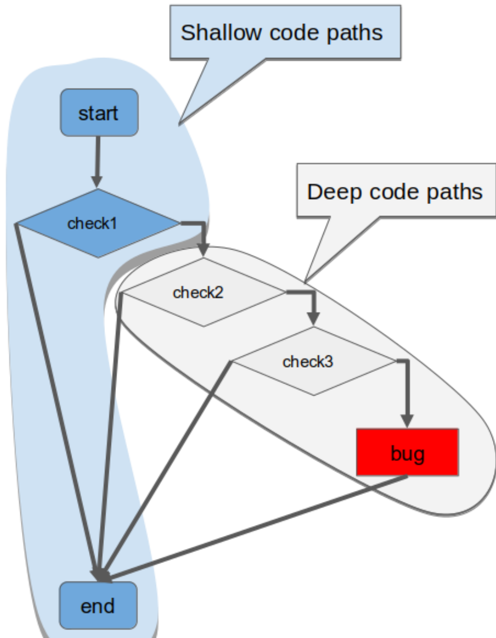
American Fuzzy Lop (++)

- AFL++ is a well-established fuzzer
- AFL uses grey-box instrumentation to track branch coverage and mutate fuzzing seeds based on previous branch coverage
- Branch coverage tracks frequency of executed edges between basic blocks
- Global coverage map keeps track of “power of two” changes
- AFL: http://lcamtuf.coredump.cx/afl/
- AFL++ https://aflplus.plus/

Fuzzer challenges: coverage wall

- After certain iterations the fuzzer no longer makes progress
- Hard to satisfy checks
- Chains of checks
- Leaps in input changes

Fuzzer challenges: coverage wall



Fuzzer challenges: coverage wall

Bypassing the coverage wall is hard, the following lists some approaches:

- Better input (seeds) can mitigate the coverage wall
- Fuzz individual components by writing fuzzer stubs (LibFuzzer)
- Better mutation operators (help the fuzzer guide exploration)
- Stateful fuzzing (teach fuzzer about different program states)
- Grammar-aware fuzzing (teach fuzzer about input grammar)

Fault detection

- How do we detect program faults?

Fault detection

- How do we detect program faults?
- Test cases detect bugs through
  - Assertions (`assert(var != 0x23 && "var has illegal value");`) detect violations
  - Segmentation faults
  - Division by zero traps
  - Uncaught exceptions
  - Mitigations triggering termination
- How can you increase the chances of detecting a bug?

Sanitization

Sanitizers enforce a given policy, detect bugs earlier and increase effectiveness of testing. Most sanitizers rely on a combination of static analysis, instrumentation, and dynamic analysis.

- The program is analyzed during compilation (e.g., to learn properties such as type graphs or to enable optimizations)
- The program is instrumented, often to record metadata at certain places and to enforce metadata checks at other places.
- At runtime, the instrumentation constantly verifies that the policy is not violated.

What policies are interesting? What metadata do you need? Where would you check?

AddressSanitizer (1/2)

AddressSanitizer (ASan) detects memory errors. It places red zones around objects and checks those objects on trigger events. ASan detects the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (configurable)
- Use-after-scope (configurable)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is 2x.

AddressSanitizer (2/2)

Goal: detect memory safety violations (both spatial and temporal)

Key idea: allocate redzones (prohibited area around memory objects), check each memory access if it targets a redzone.

- What kind of metadata would you record? Where?
- What kind of operations would you instrument?
- What kind of optimizations could you think of?

ASan Metadata

Record live objects, guard them by placing redzones around them.

ASan uses a table that maps each 8-byte word in memory to one byte in the table. Advantage: simple address calculation (`offset+address>>3`); disadvantage: memory overhead.

ASan stores accessibility of each word as metadata (i.e., is a given address accessible or not).

- An 8-byte aligned word of memory has only 9 states
- First N-bytes are accessible, 8-N are not
    - 0: all accessible
    - 7: first 7 accessible
    - 6: first 6 accessible
    - ...
    - 1: first byte accessible
    - -1: no byte accessible
- Trade-off between alignment and encoding (extreme: 128 byte alignment, per byte)

ASan instrumentation: memory access

```
long *addr = getAddr();
long val;
char *shadow = (addr>>3) + shadowbse;

// 8-byte access (read/write)
if (*shadow)
  ReportError(a);
val = *addr;

// N-byte access instead:
if (*shadow && *shadow <= ((addr&7)+N-1))
```

ASan instrumentation: asm

```
shr $0x3,%rax          # shift by 3
mov $0x100000000000,%rcx
or %rax,%rcx           # add offset
cmpb $0x0,(%rcx)       # load shadow
je .out
ud2                    # generate SIGILL
.out:
movq $0x1234,(%rdi) # original store
```

ASan instrumentation: stack

Insert red zones around objects on stack, poison them when entering stack frames.

```
void foo() {
  char rz1[32]; // 32-byte aligned
  char a[8];
  char rz2[24];
  char rz3[32];
  int *shadow = (&rz1 >> 3) + kOffset;
  shadow[0] = 0xffffffff; // poison rz1
  shadow[1] = 0xffffff00; // poison rz2
  shadow[2] = 0xffffffff; // poison rz3
  <------------- CODE ------------->
  shadow[0] = shadow[1] = shadow[2] = 0;
}
```

ASan instrumentation: globals

Insert red zone after global object, poison in init.

```
int a;
// translates to
struct {
  int original;
  char redzone[60];
} a; // again, 32-byte aligned
```

ASan runtime library

- Initializes shadow map at startup
- Replaces malloc/free to update metadata (and pad allocations with redzones)
- Intercepts special functions such as `memset`

ASan report (1/2)

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

ASan report (2/2)

```
gannimo@wasp  ~  clang++ -fsanitize=address -g test.cpp
gannimo@wasp  ~  ./a.out test
=================================================================
==7862==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe48 at pc 0x0000004ed812 bp 0x7ffdf6c39de0 sp 0x7ffdf6c39dd8
READ of size 4 at 0x61400000fe48 thread T0
    #0 0x4ed811  (/home/gannimo/a.out+0x4ed811)
    #1 0x7f3686d382e0  (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)
    #2 0x41b2f9  (/home/gannimo/a.out+0x41b2f9)

0x61400000fe48 is located 8 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x4eb0d0  (/home/gannimo/a.out+0x4eb0d0)
    #1 0x4ed7c6  (/home/gannimo/a.out+0x4ed7c6)
    #2 0x7f3686d382e0  (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)

previously allocated by thread T0 here:
    #0 0x4eaad0  (/home/gannimo/a.out+0x4eaad0)
    #1 0x4ed7a4  (/home/gannimo/a.out+0x4ed7a4)
    #2 0x7f3686d382e0  (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)

SUMMARY: AddressSanitizer: heap-use-after-free (/home/gannimo/a.out+0x4ed811)
Shadow bytes around the buggy address:
  0x0c287fff9f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c287fff9fc0: fa fa fa fa fa fa fa fa fd[fd]fd fd fd fd fd fd
  0x0c287fff9fd0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9fe0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9ff0: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
```

ASan policy

- Instrument every single access, check for poison value in shadow table
- Advantage: fast checks
- Disadvantage: large memory overhead (especially on 64 bit), still slow (2x)

LeakSanitizer

LeakSanitizer detects run-time memory leaks. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode.

LSan adds almost no performance overhead until process termination, when the extra leak detection phase runs.

MemorySanitizer

MemorySanitizer detects uninitialized reads. Memory allocations are tagged and uninitialized reads are flagged.

Typical slowdown of MemorySanitizer is 3x.

Note: do not confuse MemorySanitizer and AddressSanitizer.

UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (UBSan) detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs. Detectable errors are:

- Unsigned/misaligned pointers
- Signed integer overflow
- Conversion between floating point types leading to overflow
- Illegal use of NULL pointers
- Illegal pointer arithmetic
- ...

Slowdown depends on the amount and frequency of checks. This is the only sanitizer that can be used in production. For production use, a special minimal runtime library is used with minimal attack surface.

ThreadSanitizer

ThreadSanitizer detects data races between threads. It instruments writes to global and heap variables and records which thread wrote the value last, allowing detecting of WAW, RAW, WAR data races.

Typical slowdown is 5-15x with 5-15x memory overhead.

HexType

HexType detects type safety violations. It records the true type of allocated objects and makes all type casts explicit.

Typical slowdown is 0.5x.

Sanitizers

- AddressSanitizer: https://clang.llvm.org/docs/AddressSanitizer.html
- LeakSanitizer: https://clang.llvm.org/docs/LeakSanitizer.html
- MemorySanitizer: https://clang.llvm.org/docs/MemorySanitizer.html
- UndefinedBehaviorSanitizer: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- ThreadSanitizer: https://clang.llvm.org/docs/ThreadSanitizer.html
- HexType: https://github.com/HexHive/HexType

Use sanitizers to test your code. More sanitizers are in development.

Summary and conclusion

- Software testing finds bugs before an attacker can exploit them
- Manual testing: write test cases to trigger exceptions
- Fuzz testing automates and randomizes testing
- Sanitizers allow early bug detection, not just on exceptions
- AddressSanitizer is the most commonly used sanitizer and enforces probabilistic memory safety by recording metadata for every allocated object and checking every memory read/write.

Don't forget the Moodle quiz!