CS412 Software Security

Program Testing – Sanitization

Mathias Payer

EPFL, Spring 2019

Why testing?

> *Testing is the process of executing a program 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.

Alternative: (purely) static analysis

A static analysis *reasons* about the code instead of executing it.
Several static analysis exist and the line between static and dynamic analysis is blurry.
Generally: a static analysis looks only at the code without keeping track of concrete values during execution.

- Advantage: no concrete input is needed.
- Disadvantage: over-approximation due to imprecision and aliasing, may have large amounts of false positives.

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

Fault detection

How do we detect program faults?

Fault detection

- 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 verified that the policy is not violated.

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

AddressSanitizer

AddressSanitizer (ASan) detects memory errors. It places red zones around objects and checks those objects on trigger events. The tool can detect 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.
Note: some of the ASan slides are inspired by Kostya's presentation

AddressSanitizer

- 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

Idea: store the state of each word in shadow memory (i.e., is it 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 Metadata access

```
shadow = (addr>>3) + shadowbse
```

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 +2
ud2                    # generate SIGILL
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 (example)

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

ASan report (example 2/2)

```
=================================================================
==23666==ERROR: AddressSanitizer: heap-use-after-free on ad
at pc 0x0000004ed7d2 bp 0x7ffc48249450 sp 0x7ffc48249448
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x4ed7d1  (/home/gannimo/a.out+0x4ed7d1)
    #1 0x7f64f2fda2e0  (/lib/x86_64-linux-gnu/libc.so.6+0x2
    #2 0x41b2f9  (/home/gannimo/a.out+0x41b2f9)

0x61400000fe44 is located 4 bytes inside of 400-byte regio
[0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x4eb0d0  (/home/gannimo/a.out+0x4eb0d0)
    #1 0x4ed79e  (/home/gannimo/a.out+0x4ed79e)
    #2 0x7f64f2fda2e0  (/lib/x86_64-linux-gnu/libc.so.6+0x2

previously allocated by thread T0 here:
    #0 0x4eaad0  (/home/gannimo/a.out+0x4eaad0)
    #1 0x4ed793  (/home/gannimo/a.out+0x4ed793)
```

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
- 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.