Automated Test-Case Generation: Address Sanitizer

Testing, Quality Assurance, and Maintenance
Fall 2023

Prof. Arie Gurfinkel

based on
https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

UNIVERSITY OF WATERLOO

Automated Test Case Generation

Test cases can be generated automatically, but…

How to generate interesting test inputs

- Black box – truly random, common / interesting test patterns
- Grey box – guided by coverage, new inputs should cover new code paths
- White box – symbolic reasoning about program code, new inputs are guaranteed to cover new code paths

How to generate automatic / generic test oracles

- do not crash! (easy to check, but often not informative / soon enough)
- do not misuse memory (buffer overflow, use-after-free, …)
- no data races
- user written assertions!
- domain specific specifications and oracles

How to detect bad memory accesses

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

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

Will this program crash?

- depends on the implementation of the memory allocator (`malloc()`)
- If memory for `x` and `y` is allocated next to one another, then `*(x+12)` is the same as `*(y+2)` which is well defined
- otherwise, it might crash

Unpredictable behavior makes it difficult to test and diagnose the problem. Big issue for automatic testing!

Valgrind

An instrumentation framework for dynamic analysis tools

Interprets a program on “synthetic” CPU

Analysis tools inspect CPU instructions and insert additional checks at very low level

Execution of every instruction is interpreted in a sandbox and error report is produced when suspicious behavior is detected

Pros: very detailed analysis

Cons: 10x or more slowdown in performance

UNIVERSITY OF WATERLOO

Address Sanitizer

Compile-time instrumentation

Supported by Clang and GCC

Run-time library (~ 5 KLOC)

Supports {x86, x86_64} x {Linux, Mac, Windows}

Found hundreds of bugs since 2011

- often used in production code
- major part of any automated test-case generation validation

`

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

Key Idea: Instrument all Memory Accesses

The compiler instruments each store and load instruction with a check whether the memory being accessed is accessible (**not poisoned**)

- instrumentation must be very very efficient!
- meta-information about memory (poison/non-poison) must be stored somewhere

Original	Instrumented

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

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

UNIVERSITY OF WATERLOO

Memory Mapping

Virtual memory is divided into two disjoint classes: **Mem** and **Shadow**

- **Mem** is the normal application memory
- **Shadow** is memory that keeps track of meta-data (information) about main memory. For each byte *addr* of Mem, Shadow contains a descriptor *Shadow[addr]*

Poisoning a byte addr of Mem means writing a special value to corresponding place in Shadow

Mem and **Shadow** must be organized in such a way that mapping Mem address to Shadow is super fast

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

UNIVERSITY OF WATERLOO

Memory Alignment

Process memory is divided into 8 byte words, called *QWORDs*

Heap and stack allocation (`malloc`(), `alloca`(), local variables) are allocated at a qword boundary

- i.e., address of an allocated memory is always divisible by 8
- this is called **alignment** (of 8 bytes)
- actual alignment depends on the architecture (4, 8, 16, 128 are possible)
- For simplicity, we fix all alignments at 8 bytes

Depending on the architecture (ARM, Intel, …) unaligned memory accesses are expensive / impossible

- Compilers and runtime allocators optimize the code so that most accesses are aligned

State of an allocated QWORD

AddressSanitizer maps each QWORD of **Mem** into **one** byte of **Shadow**

Each QWORD can be in one of 9 states

- All 8 bytes are accessible (not poisoned). Shadow value is 0
- All 8 bytes are inaccessible (poisoned). Shadow value is negative (< 0)
- First *k* bytes are accessible, the rest *8-k* byes are not, $0 < k < 8$. Shadow is k

No other cases are possible because allocation is aligned at QWORD boundary

- e.g., `malloc(12)` allocated 2 QWORDS
  - all 8 bytes of the first qword are accessible
  - only 4 bytes of the second qword are accessible

New Instrumentation

```
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);
}
```

New Instrumentation (with some bit magic)

```
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 & 7) + (sz - 1)) & 7;
  return (last_accessed_byte >= shadow_value);
}
```

MemToShadow: The big trick

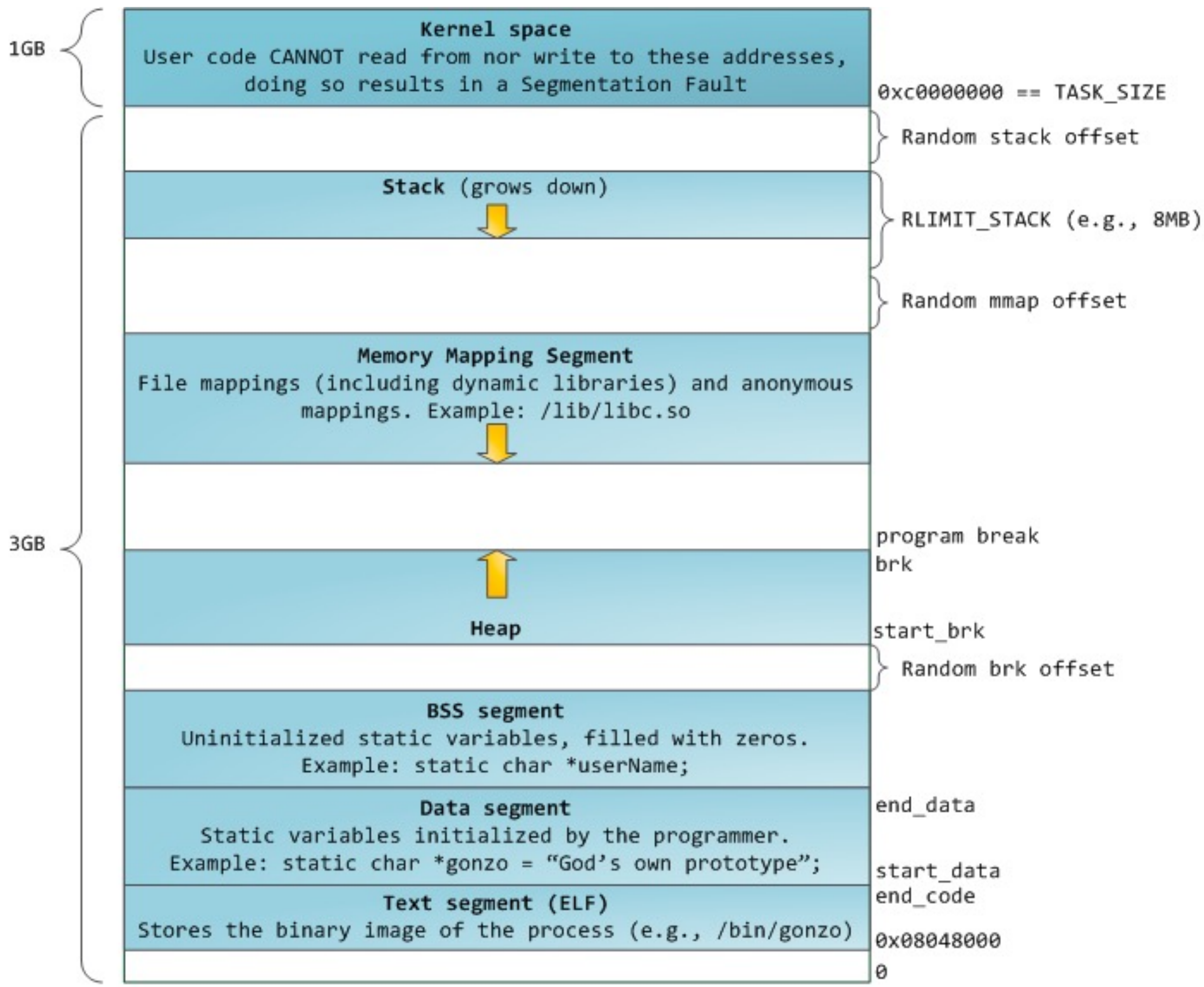
`MemToShadow(addr)` must map each QWORD of application memory Mem to a byte of the shadow memory Shadow

Must be very very very efficient

- as few CPU instructions as possible

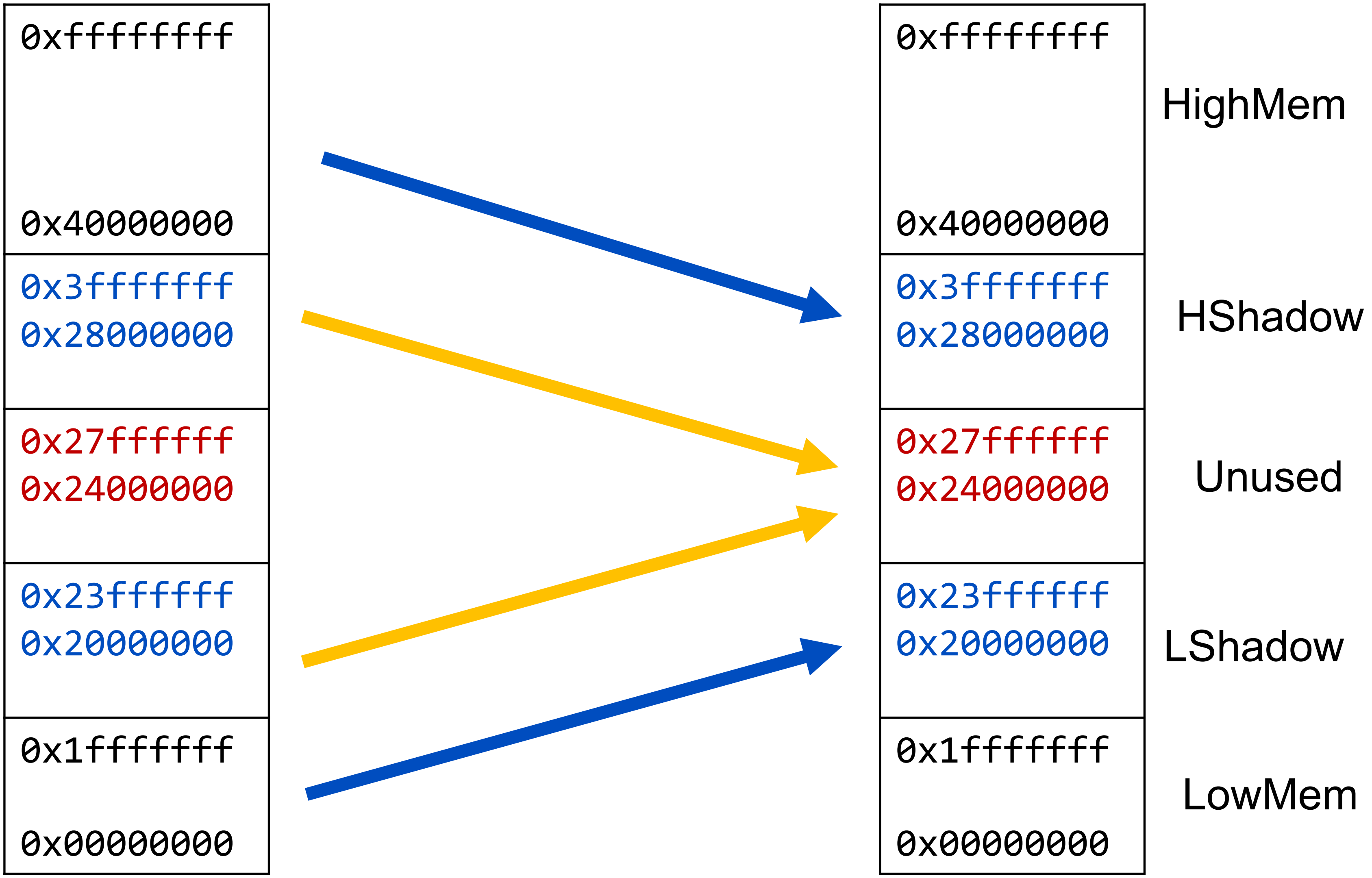
Exploits the physical layout of process memory

Process Address Space Layout



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

Mapping: Shadow = (Mem >> 3) + 0x20000000



Final Instrumentation (with all the magic)

```
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);
}
```

UNIVERSITY OF WATERLOO

But does this work for our original example?

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

  *y = *(x + 12);
}
```

Will this program crash?

- depends on the implementation of the memory allocator (`malloc()`)
- If memory for `x` and `y` is allocated next to one another, then `*(x+12)` is the same as `*(y+2)` which is well defined
- otherwise, it might crash

Unpredictable behavior makes it difficult to test and diagnose the problem. Big issue for automatic testing!

Marking Allocation boundaries with redzones

Change heap allocator to mark boundaries of allocated segments

- The markers are called ***redzones***
- All calls to `malloc()` are replaced with calls to `__asan_malloc()`

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

  void *addr = malloc(sz);
  UnPoison(addr, sz);

  rz = malloc(RED_SZ);
  Poison(rz, RED_SZ);
  return addr;
}
```

```
/t/asan-demo
File  Edit  View  Search  Terminal  Help
SUMMARY: AddressSanitizer: container-overflow src/main.cpp:85:10 in VectorOverflow()
Shadow bytes around the buggy address:
  0x0c327fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c327fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c327fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c327fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c327fff8000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c327fff8010:[01]fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
  0x0c327fff8020: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
  0x0c327fff8030: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
  0x0c327fff8040: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
  0x0c327fff8050: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
  0x0c327fff8060: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
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
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==357000==ABORTING
plam@amqui /t/a/build> ls
bin/  CMakeCache.txt  CMakeFiles/  cmake_install.cmake  CTestTestfile.cmake  Makefile  src/
plam@amqui /t/a/build> bin/init-order-fiasco
```

What about the Stack

```
void foo() {
  char a[8];

  ...

  return;
}
```

No explicit allocation

Need to ensure proper alignment

Need to insert redzones

Instrumented Stack Example

```
void foo() {
  char redzone1[32];  // 32-byte aligned
  char a[8];          // 32-byte aligned
  char redzone2[24];
  char redzone3[32];  // 32-byte aligned
  int  *shadow_base = MemToShadow(redzone1);
  shadow_base[0] = 0xffffffff;  // poison redzone1
  shadow_base[1] = 0xffffff00;  // poison redzone2, unpoison 'a'
  shadow_base[2] = 0xffffffff;  // poison redzone3



  ...


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

Instrumentation in X86 ASM

```
# long load8(long *a) { return *a; }

0000000000000030 <load8>:
  30:   48 89 f8                mov    %rdi,%rax
  33:   48 c1 e8 03             shr    $0x3,%rax
  37:   80 b8 00 80 ff 7f 00    cmpb   $0x0,0x7fff8000(%rax)
  3e:   75 04                   jne    44 <load8+0x14>
  40:   48 8b 07                mov    (%rdi),%rax   <<<<<< original load
  43:   c3                      retq
  44:   52                      push   %rdx
  45:   e8 00 00 00 00          callq  __asan_report_load8
```

Instrumentation in X86 ASM

```
# int  load4(int *a)  { return *a; }

  0000000000000000 <load4>:
  0:    48 89 f8               mov    %rdi,%rax
  3:    48 89 fa               mov    %rdi,%rdx
  6:    48 c1 e8 03            shr    $0x3,%rax
  a:    83 e2 07               and    $0x7,%edx
  d:    0f b6 80 00 80 ff 7f   movzbl 0x7fff8000(%rax),%eax
  14:   83 c2 03               add    $0x3,%edx
  17:   38 c2                  cmp    %al,%dl
  19:   7d 03                  jge    1e <load4+0x1e>
  1b:   8b 07                  mov    (%rdi),%eax    <<<<<< original load
  1d:   c3                     retq
  1e:   84 c0                  test   %al,%al
  20:   74 f9                  je     1b <load4+0x1b>
  22:   50                     push   %rax
  23:   e8 00 00 00 00         callq  __asan_report_load4
```

Other Available Sanitizers (in Clang)

ThreadSafetySanitizers

- race conditions. Is a variable being modified/accessed by two threads without being protected by a lock

MemorySanitizer

- uninitialized reads. 3x slow-down
- requires **ALL** code to be instrumented

Undefined Behavior Sanitizer (ubsan)

- many checks for undefined behaviors such as integer overflow, nullptr, etc.

DataFlowSanitizer

- a framework to write data-flow dynamic sanitizers
- CREATE YOUR OWN!

Leak Sanitizer

- detects memory leaks
- no performance overhead