Memory Bugs Are Hard

Internal corruption may not be externally visible

Observable symptoms are often delayed and non-deterministic

Testing usually relies on randomly happening to hit visible symptoms

Often remain in shipped products and can show up in customer usage
Outline

● Introduction

● Memory Bugs, Part 1: Bad Pointers
  ○ a.k.a. Unaddressable Accesses

● Memory Bugs, Part 2: Bad Values
  ○ a.k.a. Uninitialized Reads

● Memory Bugs, Part 3: Lost Pointers
  ○ a.k.a. Memory Leaks

● Implementation

● Related Tools

● History
Approach: Look for Known-Bad Behavior

Tracking which pointer corresponds to which variable/object is hard

Thus, knowing where a pointer \textbf{should} point is hard

But, knowing where a pointer \textbf{should not} point is feasible

Probabilistic error detection!
Addressable Memory

Good pointers

exe  lib  heap  stack

address space

Bad pointers!
Stack Layout

int x[16];

... caller's frame
... caller's frame
... caller's frame
... current frame

\textcolor{red}{Bad pointers!}
Heap Layout

int *x = new int;
int *y = new int;
Track Good Versus Bad Memory

allocate
(new, push)

unaddressable

deallocate
(delete, pop)

good
Heap Overflow

86: int *x = new int;
87: int *y = new int;
88: *(x+1) = 42;  Error!

Error #1: UNADDRESSABLE ACCESS
Reading 4 bytes @ 0xa58c4 - 0xa58c8
Next lower object: 0xa58c0 - 0xa58c4
Offending code:
   myapp!main()   myapp.c:88
Freed Memory

```cpp
int *x = new int;
int *y = new int;
delete x;
```

Bad pointers!
Use After Free/Delete

86: int *x = new int;
87: int *y = new int;
88: delete x;
89: *x = 42;  \textcolor{red}{Error!}

Error #1: UNADDRESSABLE ACCESS
Writing 4 bytes @ \textcolor{red}{0xa58c0 - 0xa58c4}
Write overlaps freed \textcolor{red}{0xa58c0 - 0xa58c4}
Offending code:
\texttt{myapp!main()}  \texttt{myapp.c:89}
Redzones

```cpp
int *x = new int;
std::cout << *(x+8);
```

```
x:

hdr    request    pad    hdr    request    pad    hdr    request    pad    hdr    request

x:

hdr    redzone    request    pad    redzone    hdr    request
```

*Bad pointers!*
Delayed Frees

When delete or free is called, do not return the memory for re-use.

Also called “quarantine”.
Outline

- Introduction
- Memory Bugs, Part 1: Bad Pointers
  - a.k.a. Unaddressable Accesses
- Memory Bugs, Part 2: Bad Values
  - a.k.a. Uninitialized Reads
- Memory Bugs, Part 3: Lost Pointers
  - a.k.a. Memory Leaks
- Implementation
- Related Tools
- History

I help to find bad values too!
Stack Layout

```c
int x[16];
```

...
Track Three States of Memory

allocate: mmap, calloc
allocate: new, push
write
deleallocate
deallocate

unaddressable
uninitialized
defined
int *x = new int[3];
x[1] = 42;
class mydata {
    public:
        bool b;
        int i;
};
void foo(mydata d);
mydata x;
x.b = true;
x.i = 42;
foo(x);
Uninitialized Reads Are Everywhere

Copy by value for call to foo()

Initial 1 byte

Is this uninitialized read an error?

Read 4 bytes

No, foo() only uses the first byte.

Write 4 bytes

Copy by value for call to foo()

foo() { ... }

19
Solution: Delayed Error Reporting

Report uninitialized read errors on “meaningful” reads only

- Conditional branch
- Pointer
- System call

Requires propagating state as data flows through the processor

- Expensive: now we need to track colors inside the processor, not just in memory!
Outline

● Introduction
● Memory Bugs, Part 1: Bad Pointers
  ○ a.k.a. Unaddressable Accesses
● Memory Bugs, Part 2: Bad Values
  ○ a.k.a. Uninitialized Reads
● Memory Bugs, Part 3: Lost Pointers
  ○ a.k.a. Memory Leaks
● Implementation
● Related Tools
● History
Memory Leaks Are Lost Pointers

Reachability-based leak detection: a leak is memory that is no longer reachable by the application

Global memory that is never freed is not considered a leak
- Acceptable to not free memory whose lifetime matches process lifetime
Scanning Memory

Search for initialized pointers
Reachable == Not A Leak

0xa58c0:
Unreachable == A Leak

0xa58c0:

header padding header padding

no pointer found to any part of data!
Possibly Reachable Memory

0xa58c0:

header | padding | header | 0xa58c8 | padding

Suspicious!
Large integer that just looks like a pointer?!
Eliminating False Positives: new[]

C++ arrays allocated via new[] whose elements have destructors

- new[] adds header and returns to caller address past header

0xa58c0:

header  size  new[]  padding  header
Eliminating False Positives: std::string

std::string points to char[] in middle of allocation
Eliminating False Positives: Multiple Inheritance

A pointer to a class with multiple inheritance that is cast to one of the parents

○ Points to the sub-object representation in the middle of the allocation
Outline

- Introduction
- Memory Bugs, Part 1: Bad Pointers
  - a.k.a. Unaddressable Accesses
- Memory Bugs, Part 2: Bad Values
  - a.k.a. Uninitialized Reads
- Memory Bugs, Part 3: Lost Pointers
  - a.k.a. Memory Leaks
- Implementation
- Related Tools
- History

Wow, it must be really complicated?
Implementation

Monitor every action taken by the application

○ Not just memory reads or write: delayed uninitialized read reporting requires monitoring every instruction

Replace heap allocator

○ Insert redzones and delay frees
## Dr. Memory Actions

<table>
<thead>
<tr>
<th>Category</th>
<th>Application Action</th>
<th>Corresponding Tool Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>library call</td>
<td>new, new[], malloc, HeapAlloc</td>
<td>add redzones, mark between as uninitialized</td>
</tr>
<tr>
<td>library call</td>
<td>realloc, HeapReAlloc</td>
<td>add redzones, copy old shadow, mark rest as uninitialized</td>
</tr>
<tr>
<td>library call</td>
<td>calloc, HeapAlloc(HEAP_ZERO_MEMORY)</td>
<td>add redzones, mark between as defined</td>
</tr>
<tr>
<td>library call</td>
<td>delete, delete[], free, HeapFree</td>
<td>mark unaddressable and delay any re-use by malloc</td>
</tr>
<tr>
<td>system call</td>
<td>file or anonymous memory map</td>
<td>mark as defined</td>
</tr>
<tr>
<td>system call</td>
<td>memory unmap</td>
<td>mark as unaddressable</td>
</tr>
<tr>
<td>system call</td>
<td>pass input parameter to system call</td>
<td>report error if any part of parameter is not defined</td>
</tr>
<tr>
<td>system call</td>
<td>pass output parameter to system call</td>
<td>report error if any part of parameter is unaddressable; if call succeeds, mark memory written by kernel as defined</td>
</tr>
<tr>
<td>instruction</td>
<td>decrease stack pointer register</td>
<td>mark new portion of stack as uninitialized</td>
</tr>
<tr>
<td>instruction</td>
<td>increase stack pointer register</td>
<td>mark de-allocated portion of stack as unaddressable</td>
</tr>
<tr>
<td>instruction</td>
<td>copy from immediate</td>
<td>mark target as defined</td>
</tr>
<tr>
<td>instruction</td>
<td>copy from register or memory</td>
<td>copy source shadow to target shadow</td>
</tr>
<tr>
<td>instruction</td>
<td>combine 2 sources (arithmetic, logical, etc. operation)</td>
<td>combine source shadows, mirroring application operation, and copy result to target shadow</td>
</tr>
<tr>
<td>instruction</td>
<td>access memory via base and/or index register</td>
<td>report error if addressing register is uninitialized</td>
</tr>
<tr>
<td>instruction</td>
<td>access memory</td>
<td>report error if memory is unaddressable</td>
</tr>
<tr>
<td>instruction</td>
<td>comparison instruction</td>
<td>report error if any source is uninitialized</td>
</tr>
</tbody>
</table>
Instrumentation Overhead

```
lea rdx, [rdi]
cmp word ptr [gs:0x000000fe], 0x0000
jnz 0x00007fb3320c7960
test dl, 0x03
jnz 0x00007fb3320c7960
and rdx, qword ptr [0x00007fb3b2374ec0]
add rdx, qword ptr [0x00007fb3b2374eb8]
shr rdx, 0x02
movzx rcx, word ptr [rdx]
test cx, cx
jnz 0x00007fb3320c7960
mov word ptr [gs:0x000000f0], 0x0000
jmp 0x00007fb3320c6338
mov rdx, 0x00007fb3c6035491
mov rcx, 0x00007fb3320c87f8
jmp 0x00007fb3b2434cf1
mov rax, qword ptr [rdi]
```
Instrumentation Platform: DynamoRIO

Original Code (never run directly)

Dr. Memory

DynamoRIO Binary Translator

Dr. Memory

Shared Instrumentation + Callouts

Code Cache
Outline

- Introduction
- Memory Bugs, Part 1: Bad Pointers
  - a.k.a. Unaddressable Accesses
- Memory Bugs, Part 2: Bad Values
  - a.k.a. Uninitialized Reads
- Memory Bugs, Part 3: Lost Pointers
  - a.k.a. Memory Leaks
- Implementation
- Related Tools
- History

Remember, kids, look both ways before crossing the street!
Valgrind Memcheck

Similar system in errors found and deployment

Dr. Memory runs the application natively, with instrumentation inserted as inlined fastpaths and callouts to slowpaths

Valgrind runs instrumentation natively, and emulates the application

Dr. Memory is 2x faster

Dr. Memory supports Windows
Performance Comparison With Valgrind

The chart compares the performance of Valgrind Memcheck and Dr. Memory across various benchmarks. The x-axis represents different benchmarks such as gcc, mcf, sjeng, h264ref, masmempp, mcf, bwaves, milc, memcheck, and cactusADM. The y-axis shows the relative time compared to the native execution, with values ranging from 0x to 45x.

Key observations:
- Valgrind Memcheck is marked with blue bars, while Dr. Memory is marked with yellow bars.
- Some benchmarks, indicated by "Valgrind failed" labels, show significant performance issues with Valgrind.
- The yellow bars for Dr. Memory indicate where it outperforms Valgrind in terms of time saved.

Overall, the chart highlights the varying performance impacts across different benchmarks, with some showing significant improvements with Dr. Memory compared to Valgrind.
AddressSanitizer

Implements unaddressable checking and leak checking in the compiler

- No uninitialized read detection
  - MemorySanitizer
- Only detects bugs in recompiled code
  - Also intercepts common libc and libc++ function calls
- Binary pays cost of checks on every run, so a separate dedicated build is required
AddressSanitizer Performance

Faster (2x vs native) than Dr. Memory (10x) or Valgrind (20x)

- Not propagating values for uninitialized reads
- Ignores compiler “glue code”
- Register allocation and optimizations integrated with application
## Bug Coverage Comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Bugs in entire program and libraries</th>
<th>Use-after-free</th>
<th>Heap over/under flow</th>
<th>Stack var over/under flow</th>
<th>Global var over/under flow</th>
<th>Uninitialized reads</th>
<th>Leaks with no stale pointers</th>
<th>Leaks with stale pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Valgrind</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Address Sanitizer</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Example Bad Pointer Missed by AddressSanitizer

```cpp
#include <pthread.h>
#include <iostream>
void *func(void *) { return nullptr; }
int main() {
    pthread_t *p = new pthread_t;
delete p;
pthread_create(p, nullptr, func,
    nullptr);
std::cout << "All good\n";
return 0;
}
```

```
$ clang++ -fsanitize=address -g noasan2.cpp -lpthread && ./a.out
All good
```

```
$ clang++ -g noasan2.cpp -lpthread && ~/DrMemory-Linux-2.3.0-1/bin64/drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.3.0
~~Dr.M~~ Error #1: UNADDRESSABLE ACCESS of freed memory: writing
0x41eb30-0x41eb38 8 byte(s)
~~Dr.M~~ # 0 libpthread.so.0!__pthread_create_2_1
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:01.054 in thread 1759547
~~Dr.M~~ Note: next higher malloc: 0x41eb90-0x41ecb0
~~Dr.M~~ Note: 0x41eb30-0x41eb38 overlaps memory 0x41eb30-0x41eb70 that was
freed here:
~~Dr.M~~ Note: # 0 replace_operator_delete_array [.../alloc_replace.c:2999]
~~Dr.M~~ Note: # 1 main
~~Dr.M~~ Note: instruction: mov %rbx -> (%rax)
All good
```
# Example Leak Missed by AddressSanitizer

```cpp
#include <iostream>

void func1() {
    char buf1[1024];
    int *ptr = new int[4];
    std::cout<<"ptr=\"<<std::hex<<ptr<<\"\n";
    char buf2[1024];
    buf1[0] = 'a';
    buf2[0] = 'b';
}

void func2() {
    char buf1[1024 + sizeof(int*)];
    char buf2[1024];
    exit(0);
}

int main() {
    func1();
    func2();
    return 0;
}

$ clang++ -fsanitize=address -g noasan.cpp &&
    ASAN_OPTIONS="detect_leaks=1" ./a.out
    ptr=0x602000000010

$ clang++ -fsanitize=address -g noasan.cpp &&
    ASAN_OPTIONS="detect_leaks=1:detect_stack_use_after_return=1" ./a.out
    ptr=0x602000000010

$ clang++ -g noasan.cpp && ~/DrMemory-Linux-2.3.0-1/bin64/drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.3.0
    ptr=0x41e2e0
~~Dr.M~~
~~Dr.M~~ Error #1: LEAK 16 direct bytes 0x41e2e0-0x41e2f0 + 0 indirect bytes
~~Dr.M~~ # 0 replace_operator_new_array [../../alloc_replace.c:2929]
~~Dr.M~~ # 1 func1 [../../noasan.cpp:4]
~~Dr.M~~ # 2 main [../../noasan.cpp:16]
```
Outline

- Introduction
- Memory Bugs, Part 1: Bad Pointers
  - a.k.a. Unaddressable Accesses
- Memory Bugs, Part 2: Bad Values
  - a.k.a. Uninitialized Reads
- Memory Bugs, Part 3: Lost Pointers
  - a.k.a. Memory Leaks
- Implementation
- Related Tools
- History

You could build cool stuff too!
DynamoRIO

Dynamo
@HP Labs
on PA-RISC
late 1990's

Dynamo
@HP Labs
on x86
2000

RIO @MIT
(Runtime Introspection and Optimization)
1999

Dynamo + RIO →
DynamoRIO
2001
Graduate School + Industry History

Original Code (never run directly)

Dr. Memory

DynamoRIO Binary Translator

Code Cache

Dr. Memory Shared Instrumentation + Callouts
Security Startup

Original Code (never run directly)

DynamoRIO Binary Translator

Security Checks

Code Cache

Security Shared Instrumentation + Callouts
Dr. Memory in the Real World

Used by the Chrome developers for several years

- Found several hundred bugs in Chrome

Open-source

- Contributions welcome
- Google Summer of Code participant in the past
- RCOS project possibilities
The End

- Introduction
- Memory Bugs, Part 1: Bad Pointers
  - a.k.a. Unaddressable Accesses
- Memory Bugs, Part 2: Bad Values
  - a.k.a. Uninitialized Reads
- Memory Bugs, Part 3: Lost Pointers
  - a.k.a. Memory Leaks
- Implementation
- Related Tools
- History

That's all, folks! Thanks for listening.