Advanced Debugging and the Address Sanitizer

Finding your undocumented features

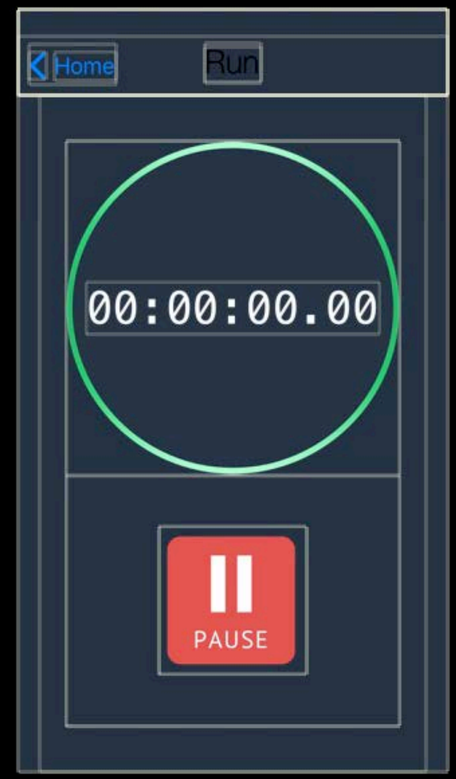
Session 413

Mike Swingler Xcode UI Infrastructure
Anna Zaks LLVM Program Analysis

Overview

Overview

View Debugger



Overview

View Debugger

Advanced Breakpoint Actions

Overview

View Debugger

Advanced Breakpoint Actions

Address Sanitizer

Overview

View Debugger

Advanced Breakpoint Actions

Address Sanitizer

Demo

View Debugger and Advanced Breakpoints

Mike Swingler Xcode UI Infrastructure

Summary

Summary

View Debugger

- Focus on troublesome views
- Visualize your constraints

Summary

View Debugger

- Focus on troublesome views
- Visualize your constraints

Advanced Breakpoint Actions

- Catch exceptions at throw, print message
- Print expressions without adding clutter

Summary

View Debugger

- Focus on troublesome views
- Visualize your constraints

Advanced Breakpoint Actions

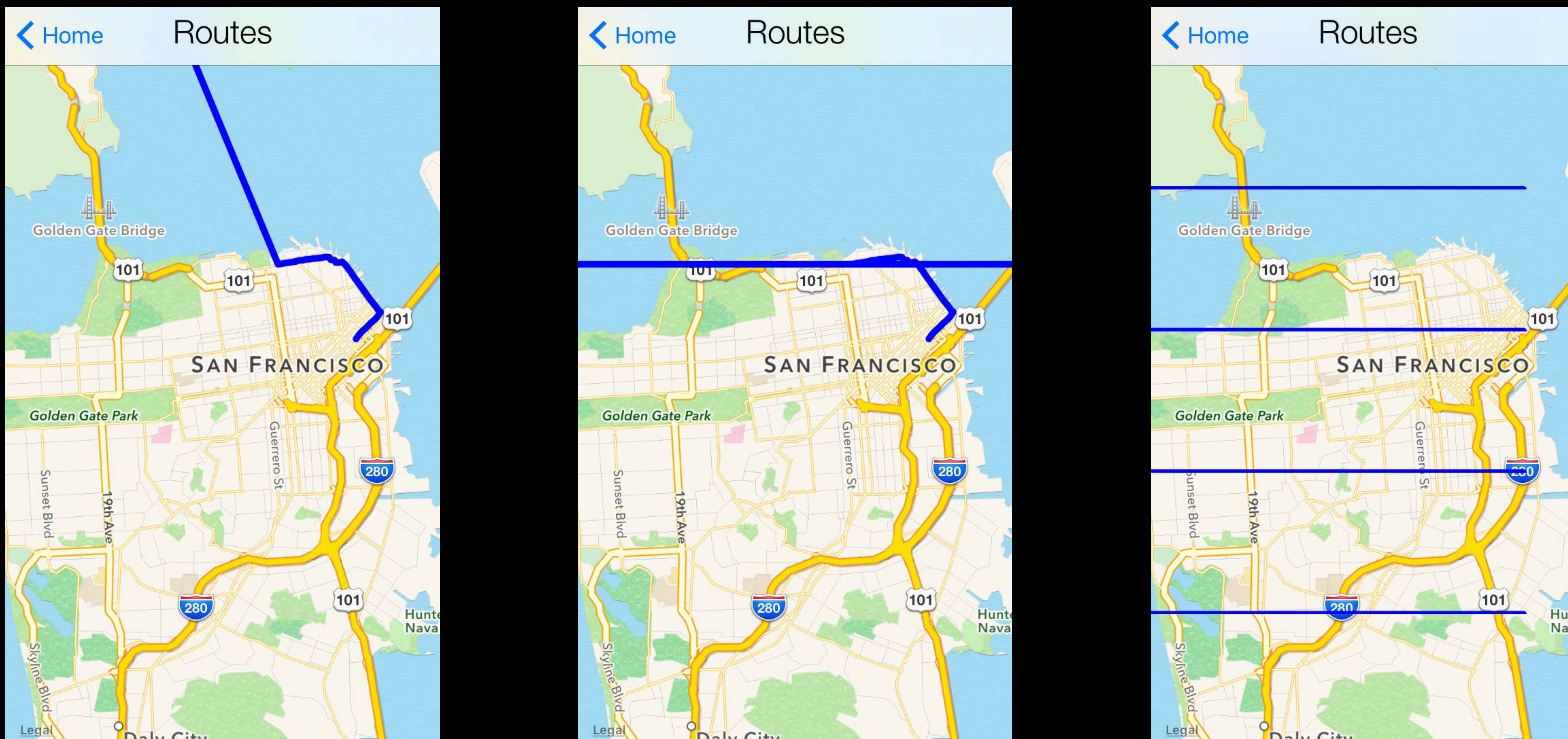
- Catch exceptions at throw, print message
- Print expressions without adding clutter

Address Sanitizer

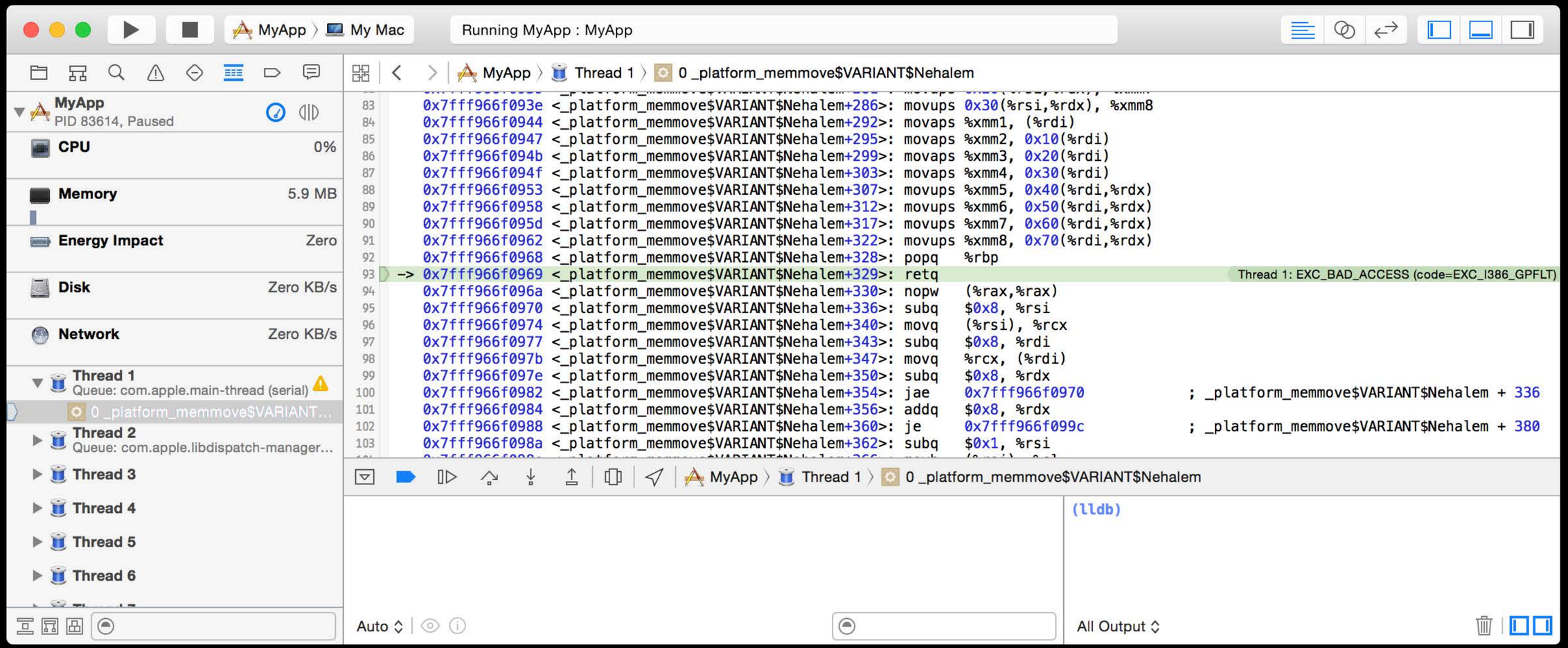
Address Sanitizer

Anna Zaks LLVM Program Analysis

Memory Corruption



Memory Corruption



Memory Corruption Is Hard to Debug

Hard to consistently reproduce

The source of error is often far from its manifestation

Language Memory Safety

Language Memory Safety

Less error prone

- Swift
- Objective-C Automatic Reference Counting

Language Memory Safety

Less error prone

- Swift
- Objective-C Automatic Reference Counting

More susceptible to memory issues

- Direct memory manipulation
- Code that interoperates with C/C++

Language Memory Safety

Less error prone

- Swift
- Objective-C Automatic Reference Counting

More susceptible to memory issues

- Direct memory manipulation
- Code that interoperates with C/C++

What Is Address Sanitizer?

Similar to Guard Malloc and Valgrind

Finds memory corruption at run time

Less overhead

Integrated into Debug Navigator

Works on OS X, iOS (simulator and device)

Analyze Memory Corruption

Use after free

Heap buffer overflow

Stack buffer overflow

Global variable overflow

Overflows in C++ containers

Use after return

Analyze Memory Corruption

Use after free

Heap buffer overflow

Stack buffer overflow

Global variable overflow

Overflows in C++ containers

Use after return

Demo

Using Address Sanitizer from Xcode

Anna Zaks LLVM Program Analysis

Demo Recap

1. Edit Scheme – Diagnostics tab
2. “Enable Address Sanitizer” checkbox
3. Build and Run

When to Use Address Sanitizer

Investigating memory corruption

Manual testing

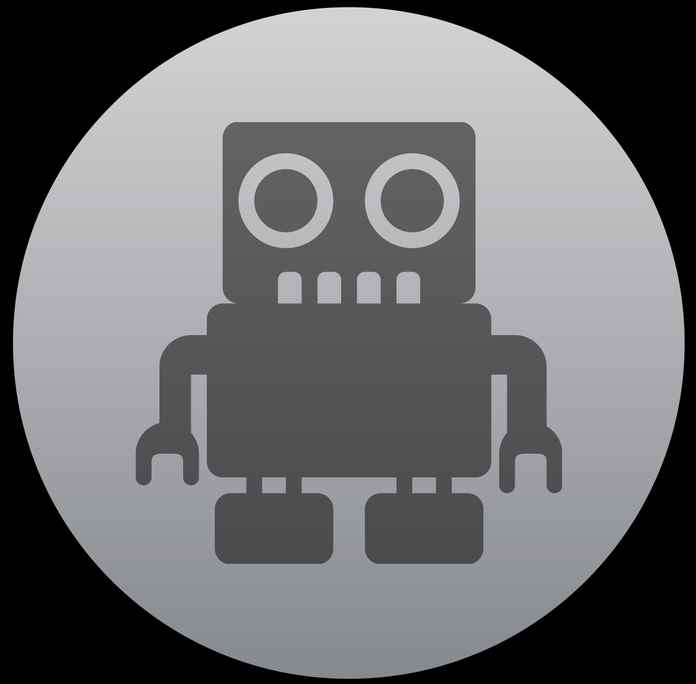
Continuous integration

Continuous Integration

Enable Sanitization in your
non-performance tests

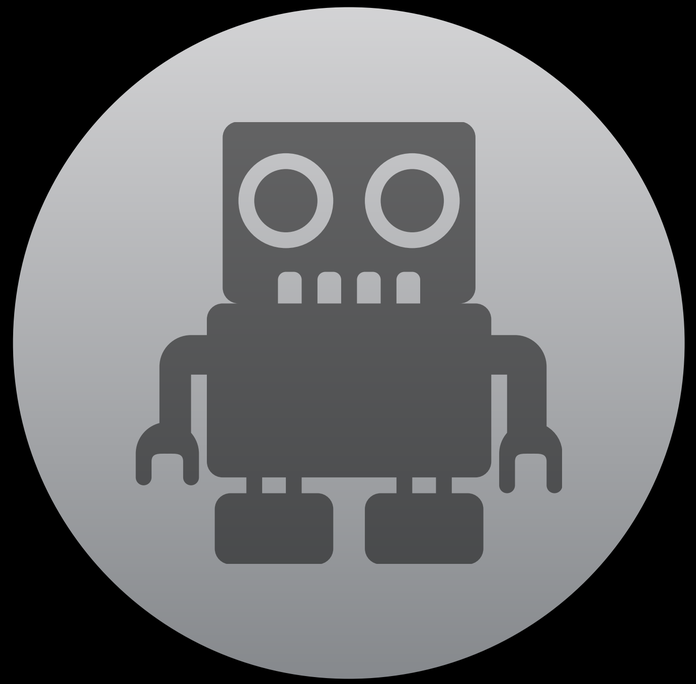
In Xcode

1. Edit Scheme – Test – Diagnostics tab
2. “Enable Address Sanitizer” checkbox
3. Build and Test



Continuous Integration

Enable Sanitization in your
non-performance tests



In Xcode

1. Edit Scheme – Test – Diagnostics tab
2. “Enable Address Sanitizer” checkbox
3. Build and Test

Command Line

```
$ xcodebuild -scheme "Jogr" test -enableAddressSanitizer YES
```

Compiler Optimization Level

None [`-O0`] is recommended

Fast [`-O1`] is supported

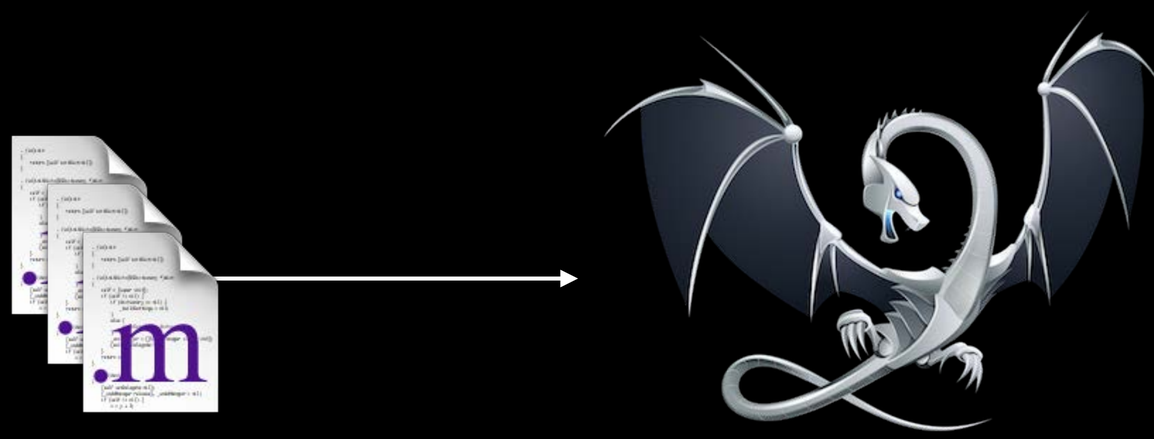
Higher optimization is not supported

Under the Hood

How Address Sanitizer works

How Address Sanitizer Works

How Address Sanitizer Works

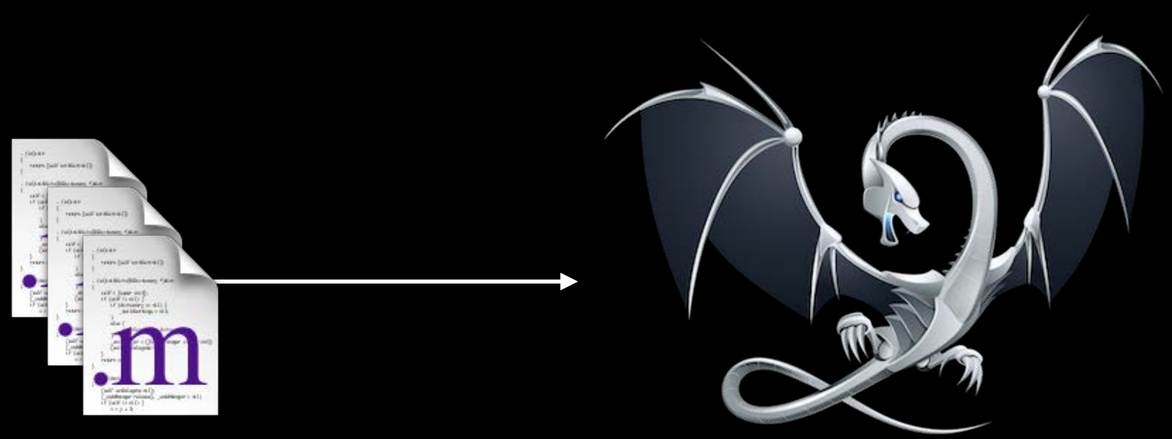


clang

How Address Sanitizer Works

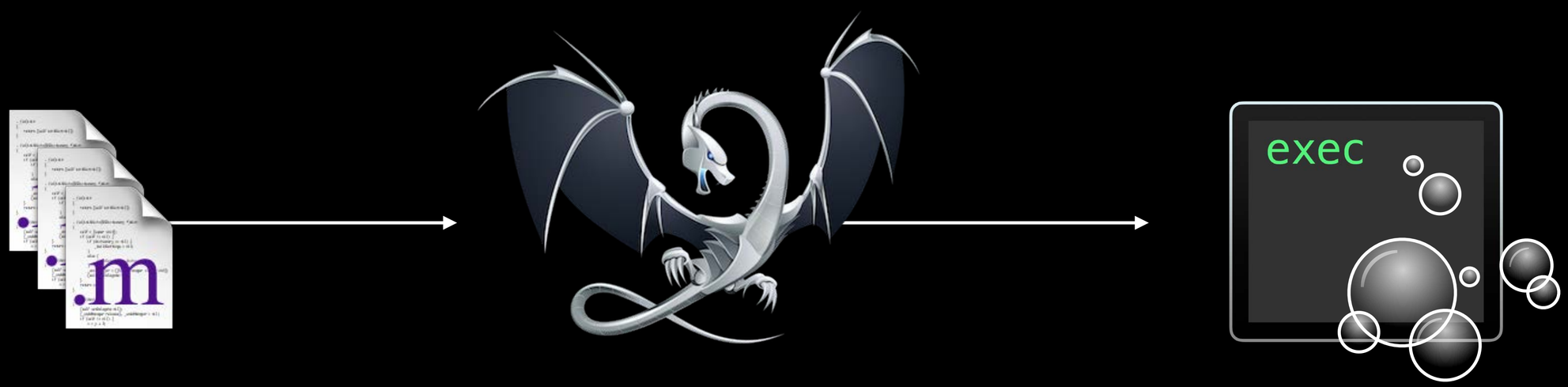
clang

How Address Sanitizer Works



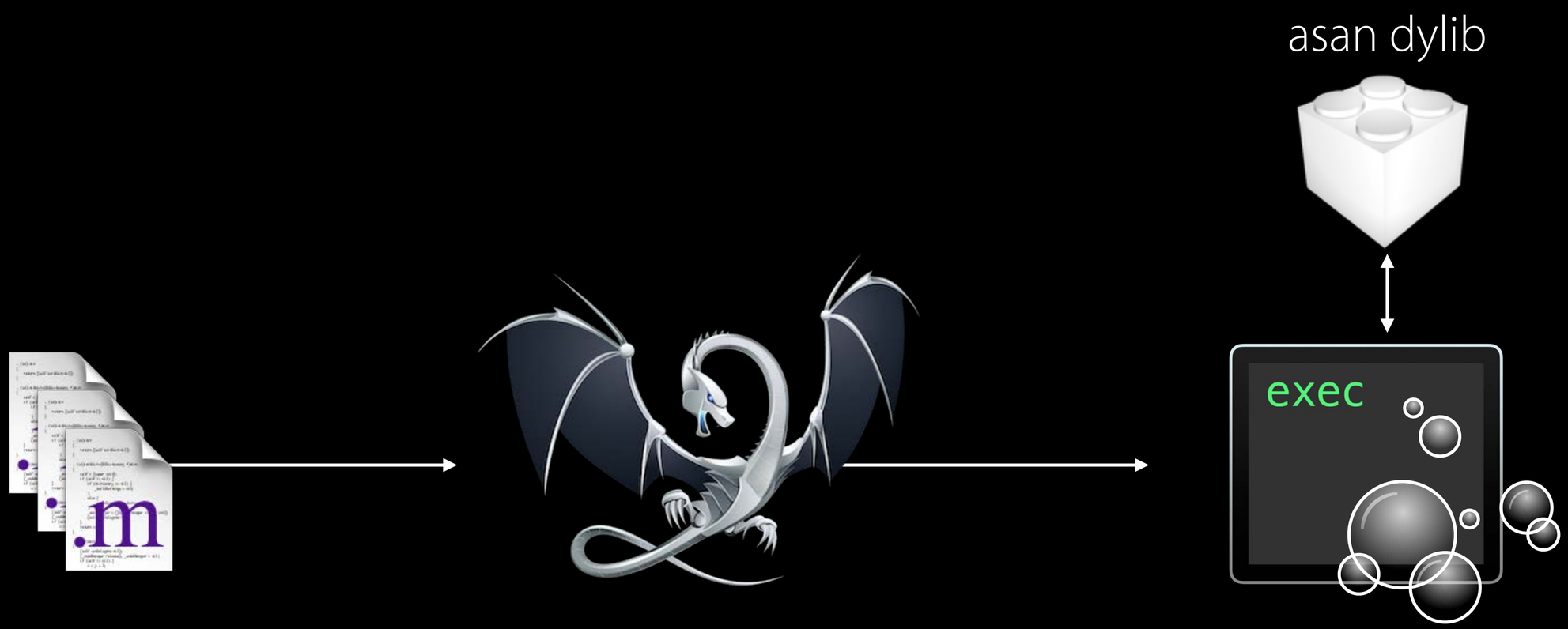
```
clang
-fsanitize=address
```

How Address Sanitizer Works




```
clang
-fsanitize=address
```

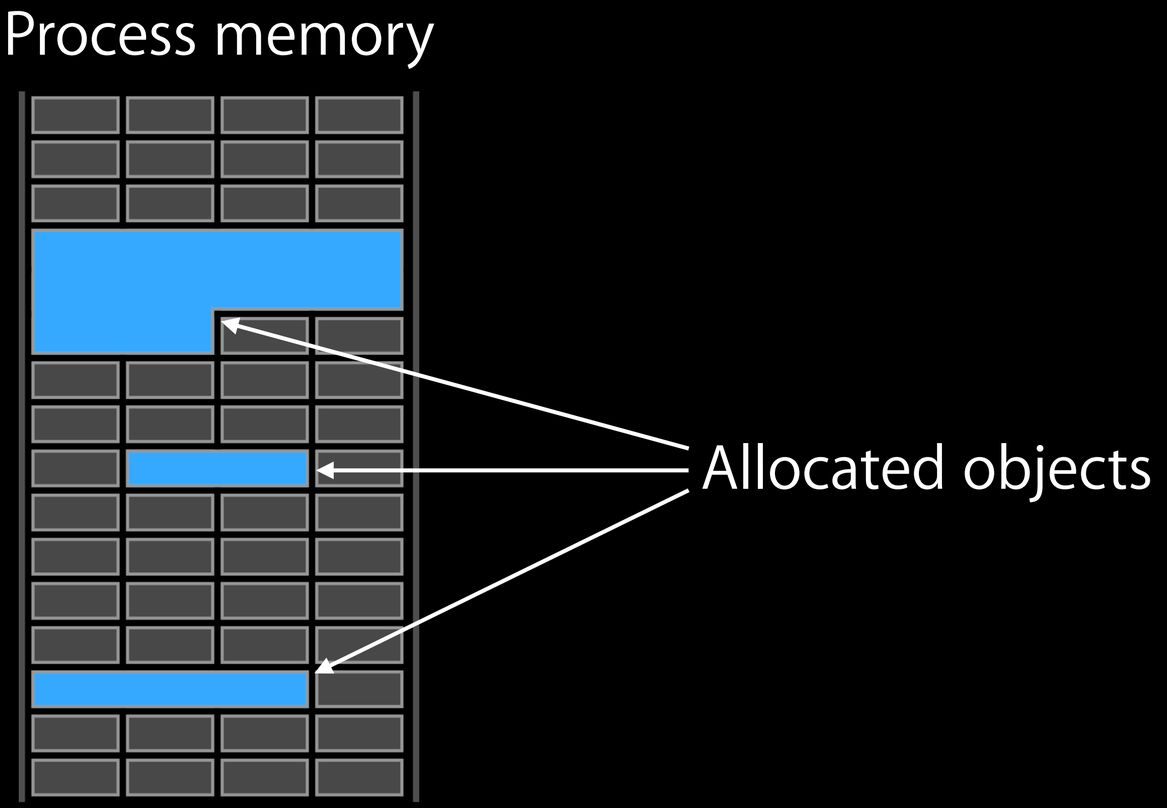
How Address Sanitizer Works




```
clang
-fsanitize=address
```

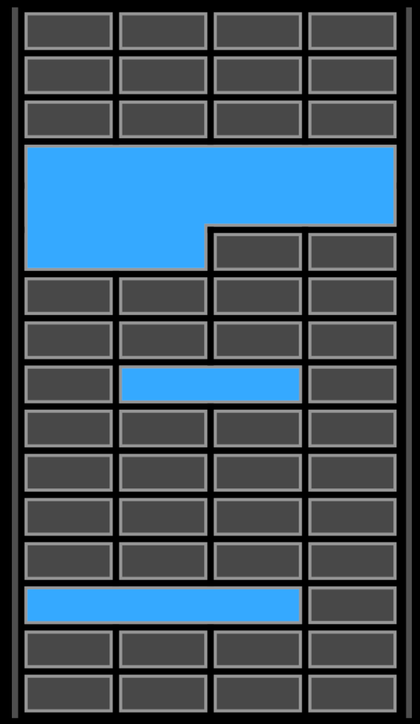
Shadow Mapping

Shadow Mapping

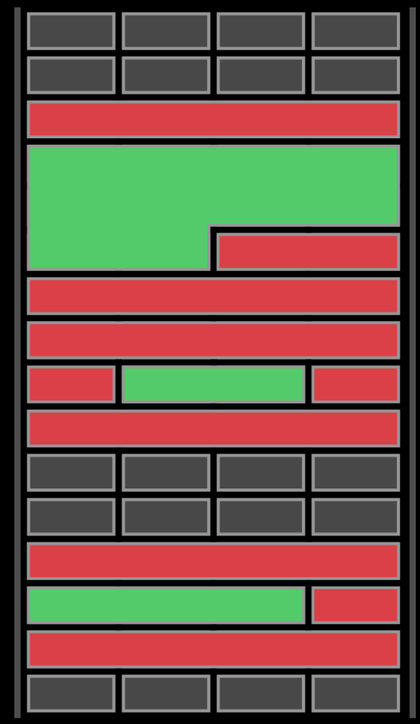


Shadow Mapping

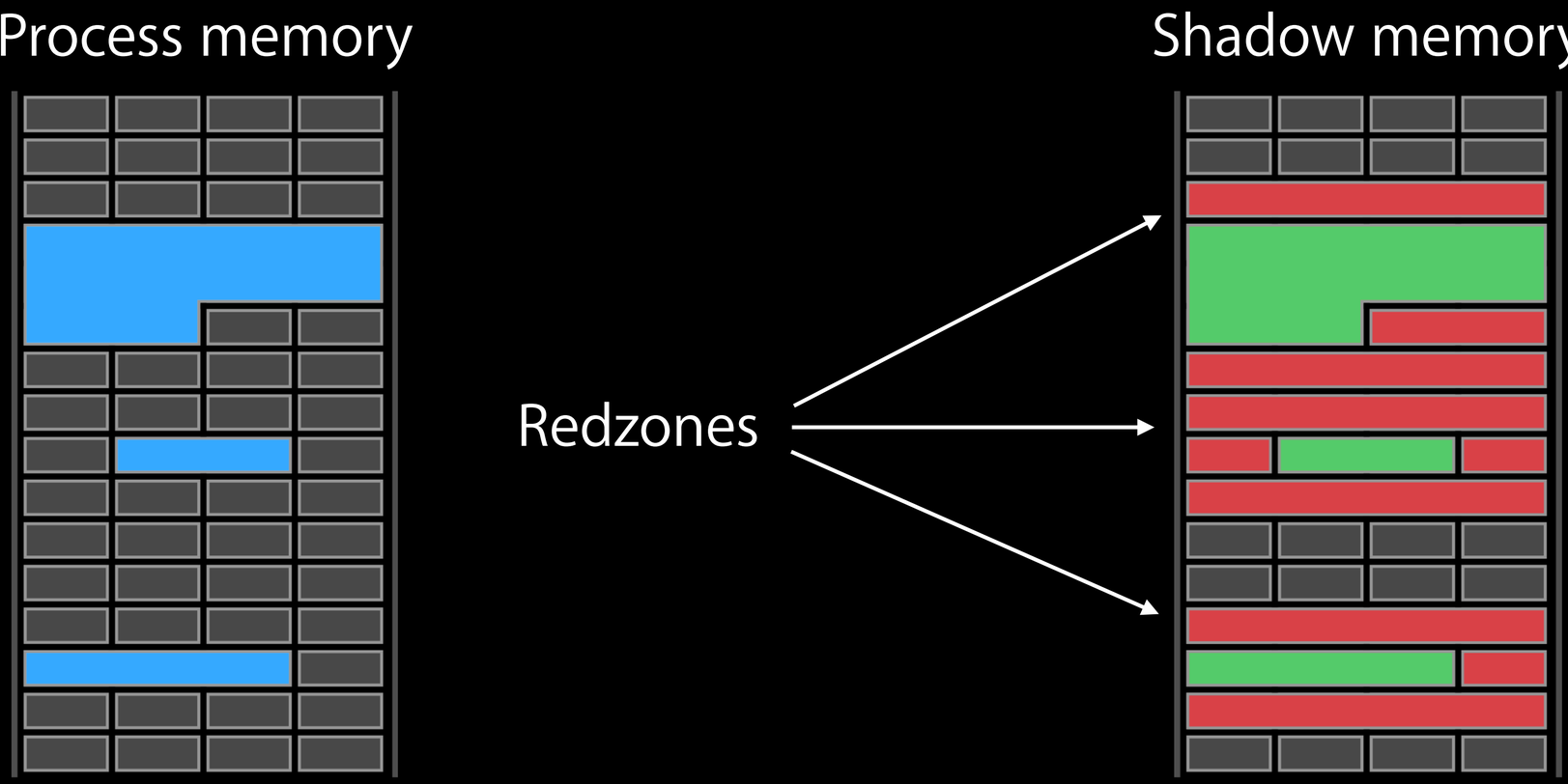
Process memory



Shadow memory



Shadow Mapping



Shadow Mapping

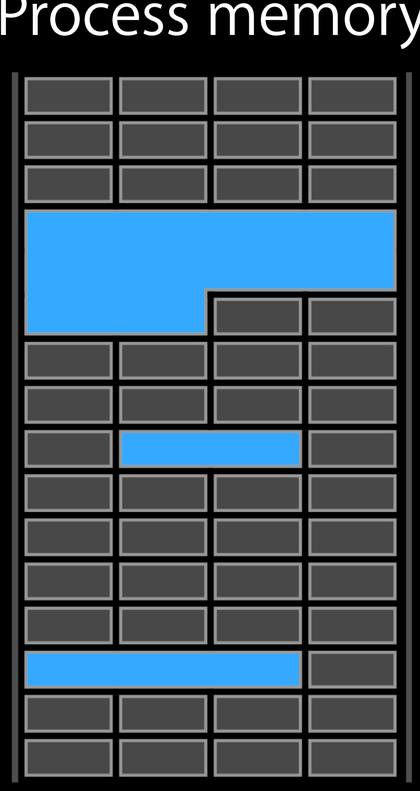
```
*p = 0xb00;
```

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

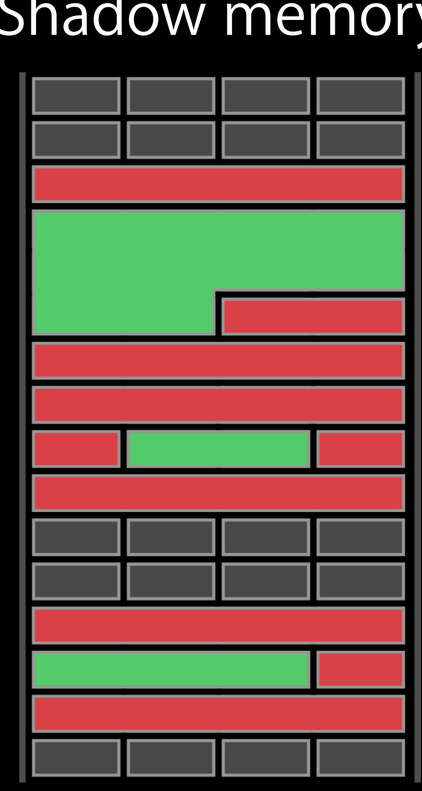
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



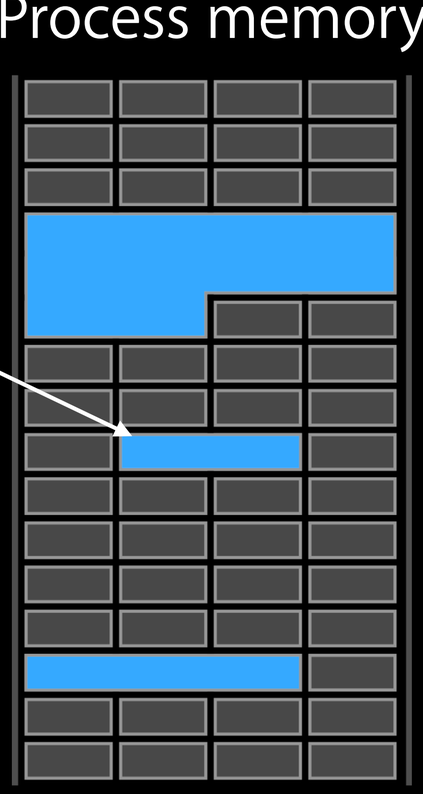
Shadow memory



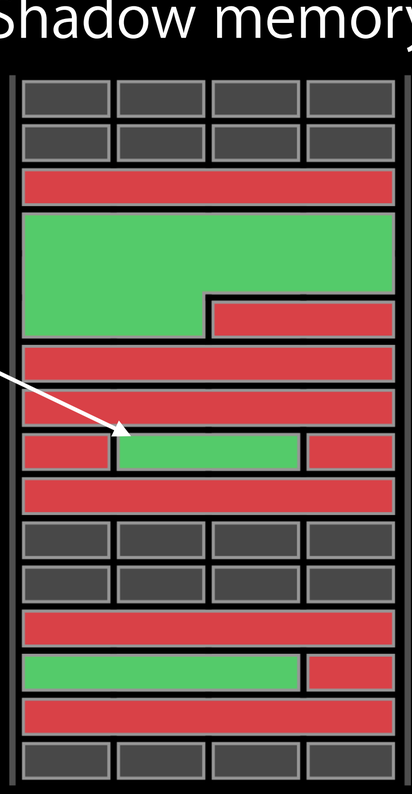
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



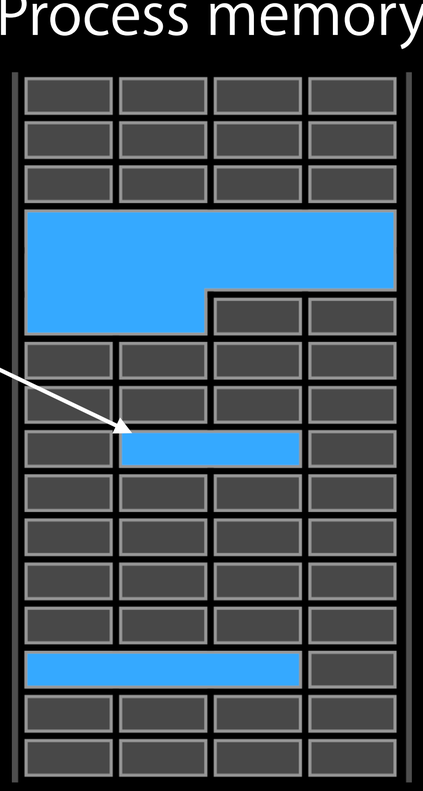
Shadow memory



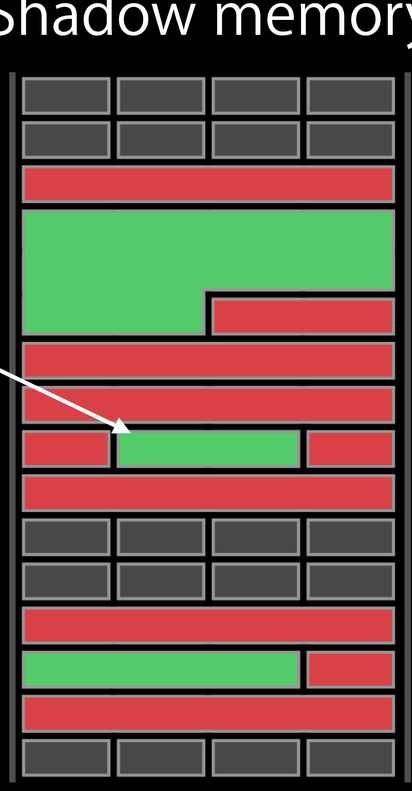
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



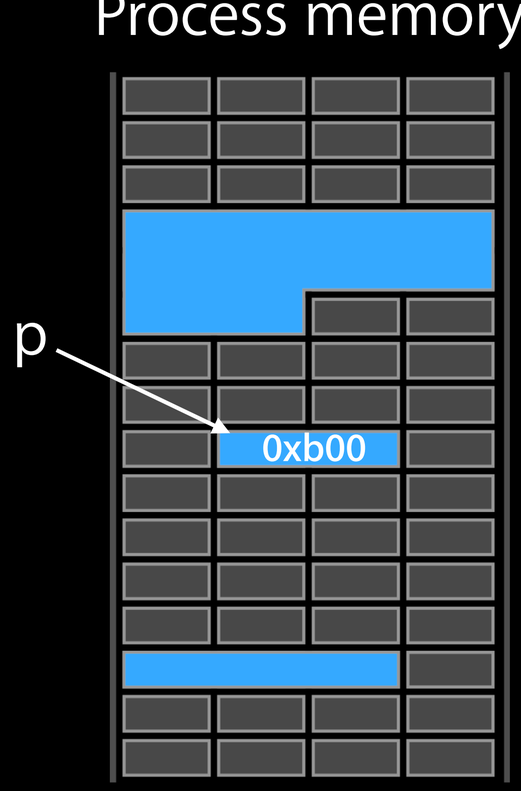
Shadow memory



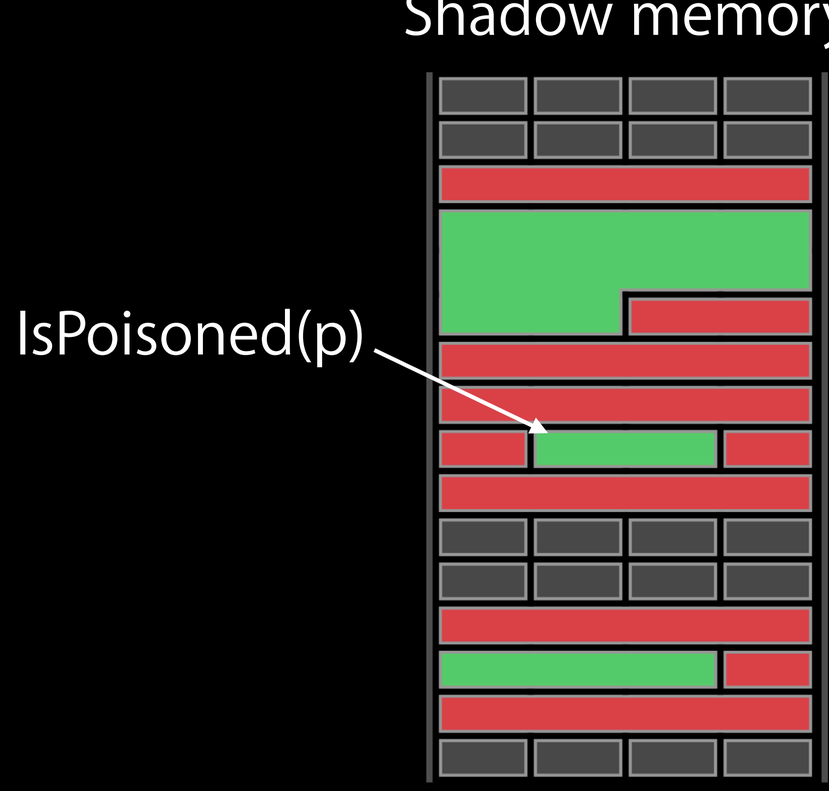
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



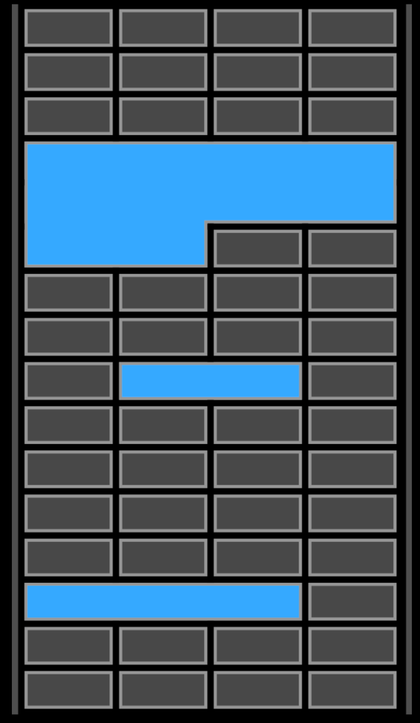
Shadow memory



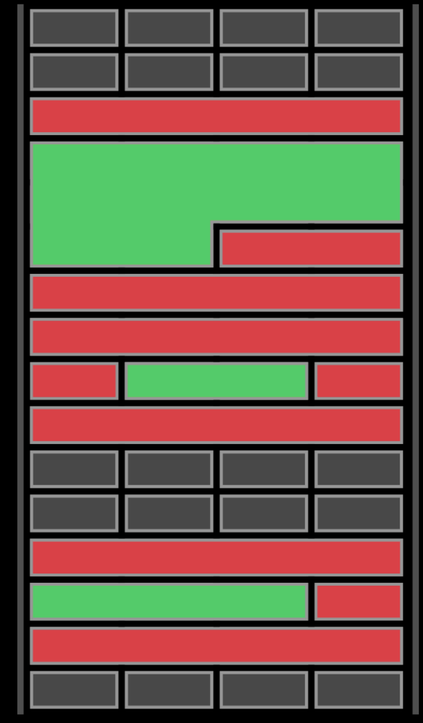
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



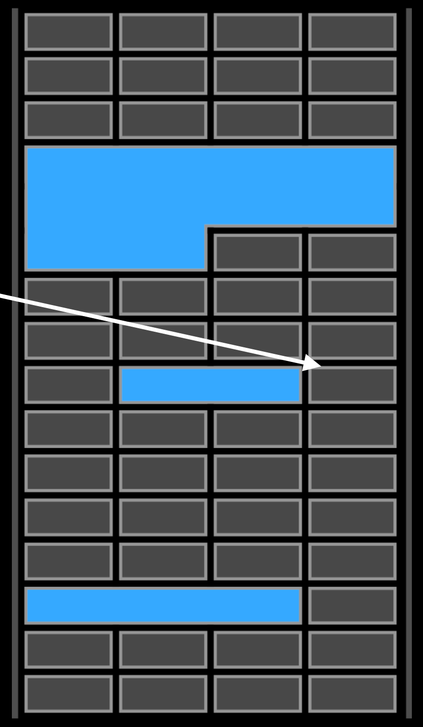
Shadow memory



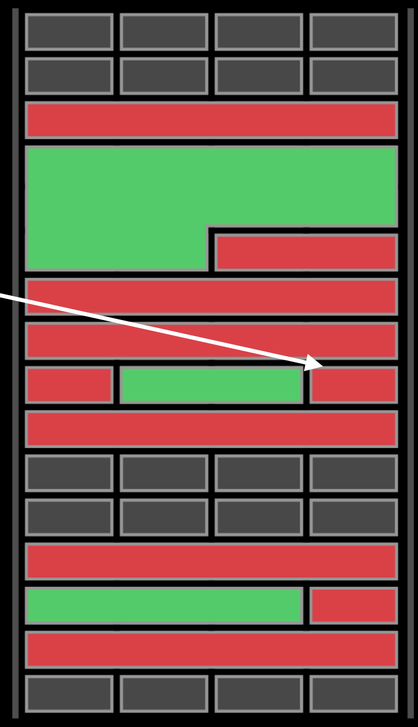
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



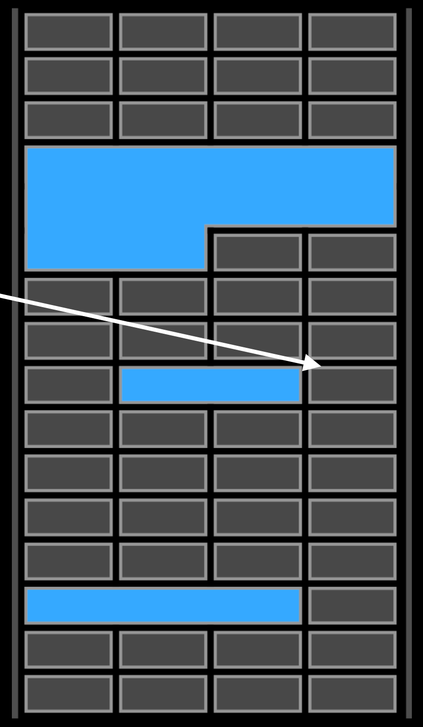
Shadow memory



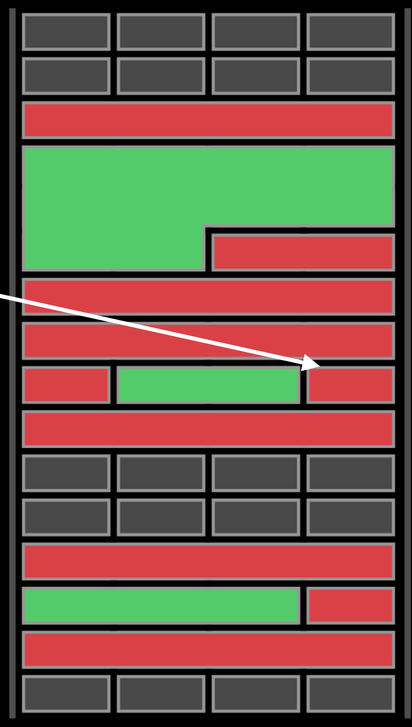
Shadow Mapping

```
if (IsPoisoned(p))
    Crash();
*p = 0xb00;
```

Process memory



Shadow memory



Shadow Mapping

IsPoisoned needs to be fast

1/8 of the address space

mmap'd at launch

Shadow Mapping

IsPoisoned needs to be fast

1/8 of the address space

mmap'd at launch

```
bool IsPoisoned(Addr) {
  Shadow = Addr >> 3 + Offset
  return (*Shadow) != 0
}
```

Default Malloc Implementation

Default Malloc Implementation

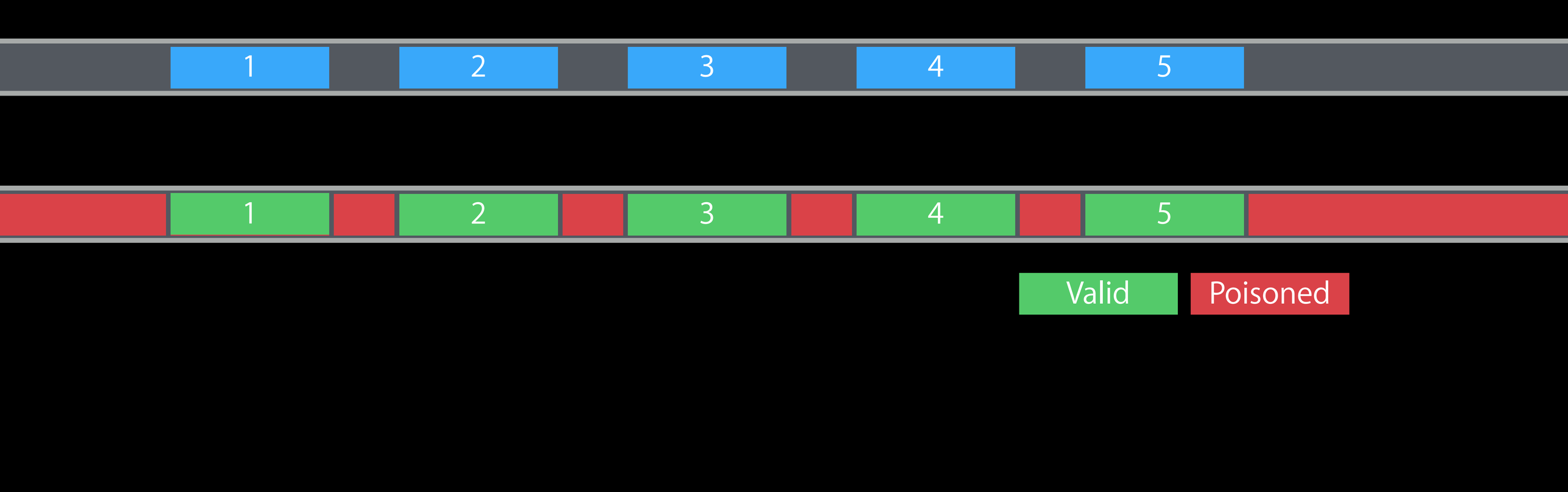
allocations

Default Malloc Implementation

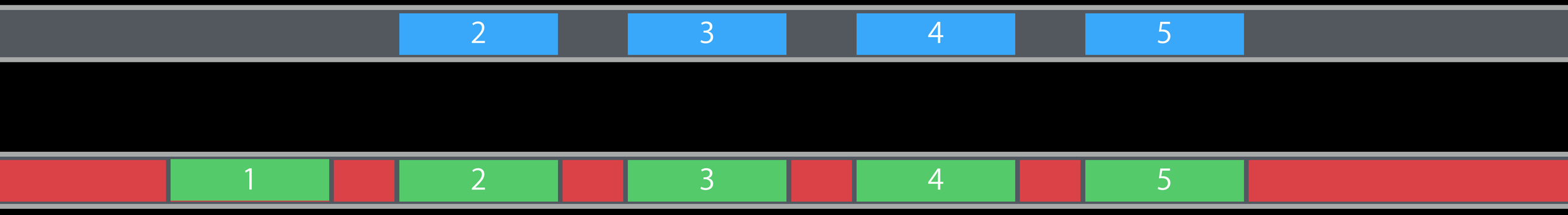
allocations

Custom Malloc Implementation

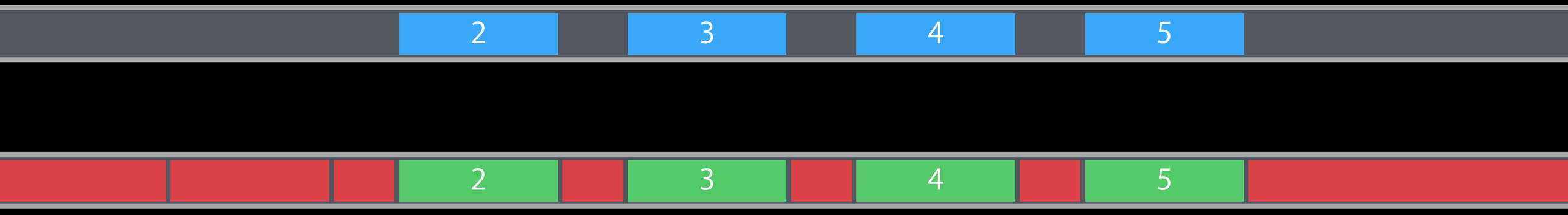
Custom Malloc Implementation



Custom Malloc Implementation



Custom Malloc Implementation



Custom Malloc Implementation

Custom Malloc Implementation

Inserts poisoned “red zones” around allocations

- Heap underflows/overflows

Custom Malloc Implementation

Inserts poisoned “red zones” around allocations

- Heap underflows/overflows

Delays reuse of freed memory

- Use-after-free, double free

Custom Malloc Implementation

Inserts poisoned “red zones” around allocations

- Heap underflows/overflows

Delays reuse of freed memory

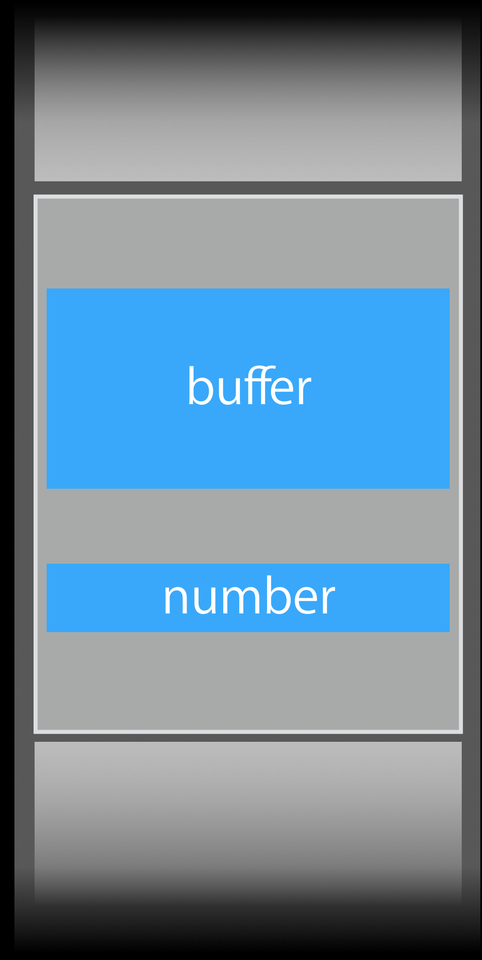
- Use-after-free, double free

Collects stack traces for allocations and frees

- Comprehensive error reports

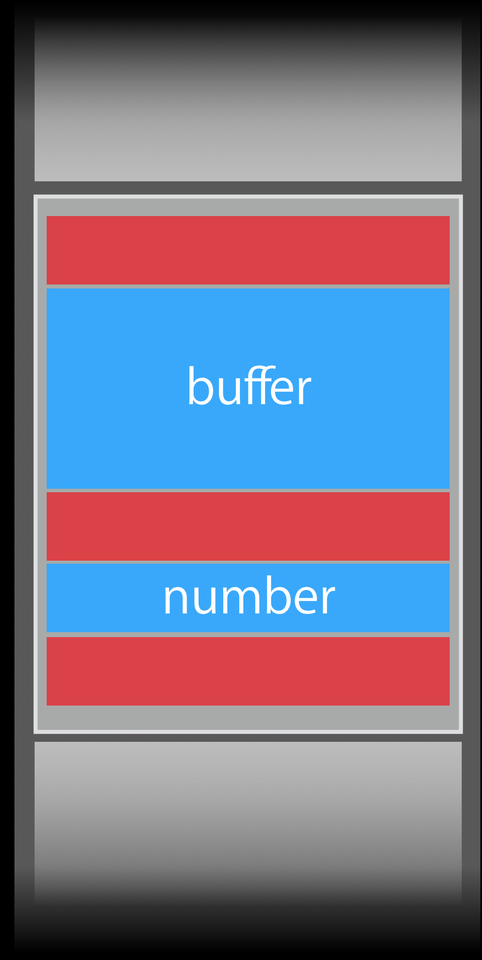
Compiler Instrumentation of the Stack

```
void foo() {
  char buffer[16];
  int number;

  …

  buffer[16] = ‘\0’;
}
```



Compiler Instrumentation of the Stack

```
void foo() {
  char buffer[16];
  int number;
  if (IsPoisoned(&buffer[16]))
    Crash();
  buffer[16] = ‘\0’;
}
```



Compiler Instrumentation of Globals

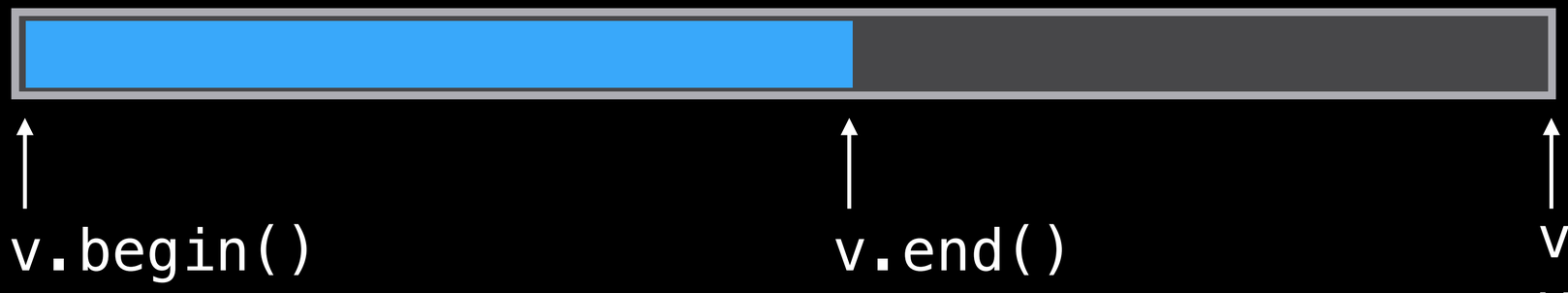
```
int array[] = {1, 2, 3};

void foo() {


  int x = array[3];
}
```

Compiler Instrumentation of Globals

```
char poisoned_redzone1[16];
int array[] = {1, 2, 3};
char poisoned_redzone2[16];
void foo() {
  if (IsPoisoned(&array[3]))
    Crash();
  int x = array[3];
}
```

Catching C++ Container Overflows

```
std::vector<T> v;
```



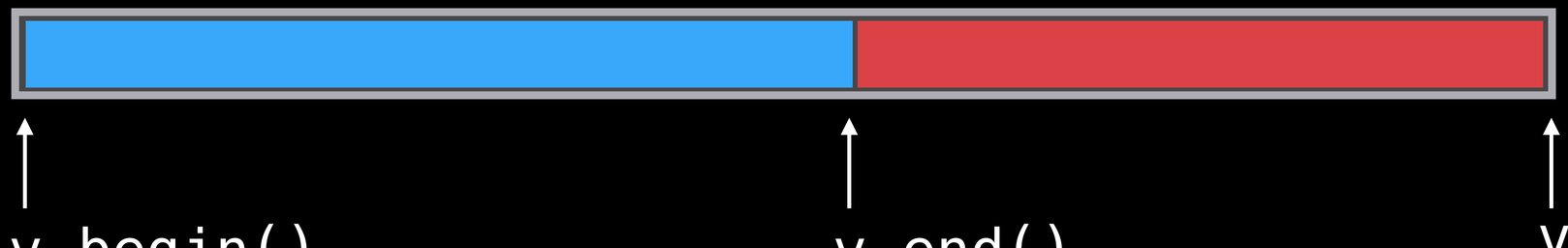
```
v.begin()
```

```
v.end()
```

```
v.begin() +
v.capacity()
```

Catching C++ Container Overflows

```
std::vector<T> v;
```



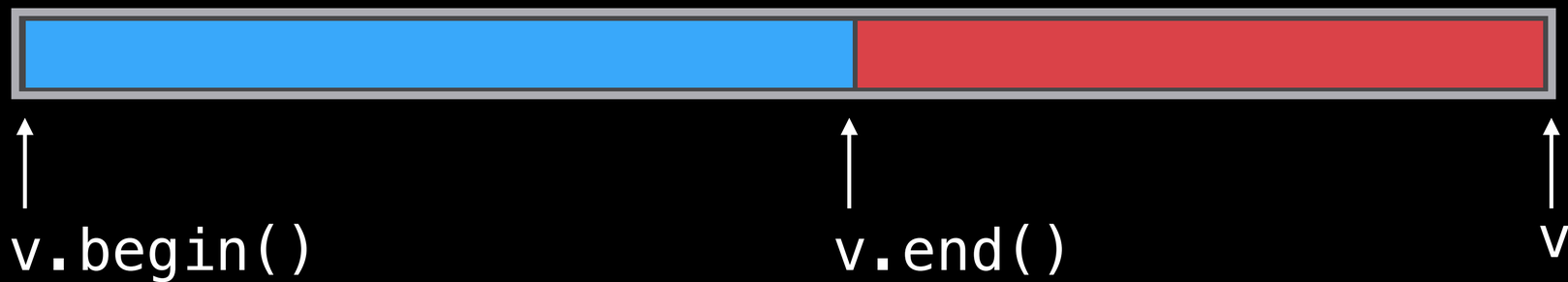
```
v.begin()
```

```
v.end()
```

```
v.begin() +
v.capacity()
```

Catching C++ Container Overflows

```
std::vector<T> v;
```



```
v.begin()
```

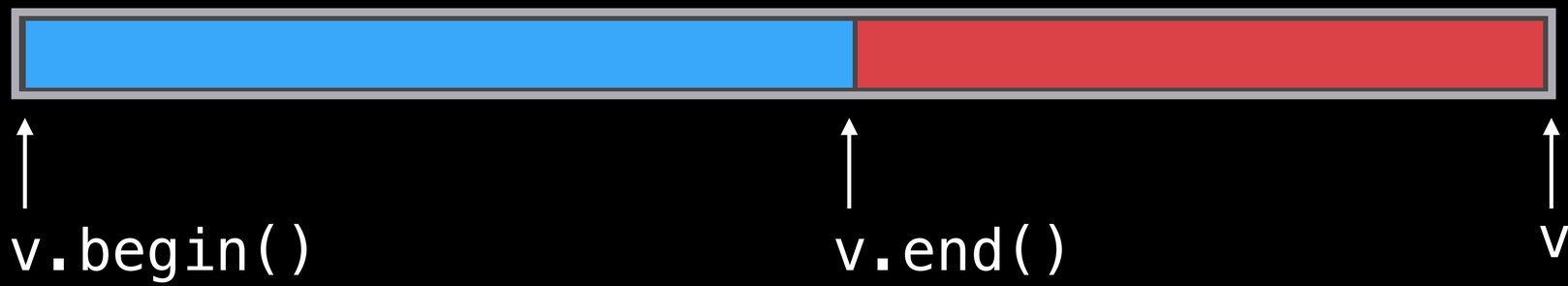
```
v.end()
```

```
v.begin() +
v.capacity()
```

```
std::vector<int> V(8);
V.resize(5);
return V.data()[5];
```

Catching C++ Container Overflows

```
std::vector<T> v;
```



```
v.begin()
```

```
v.end()
```

```
v.begin() +
v.capacity()
```

```
std::vector<int> V(8);
V.resize(5);
return V.data()[5];
```

container-overflow

Runtime Function Interposition

Runtime Function Interposition

Wraps `memcpy`, `memset`, `strcpy`, `strlen`, `fwrite`, `printf`, `getline`, . . .

Extended with extra memory checks

These checks work even in non-instrumented code

Runtime Function Interposition

Wraps `memcpy`, `memset`, `strcpy`, `strlen`, `fwrite`, `printf`, `getline`, . . .

Extended with extra memory checks

These checks work even in non-instrumented code

```
wrap_memcpy(dest, src, n) {
  ASSERT_MEMORY_READABLE(src, n)
  ASSERT_MEMORY_WRITABLE(dest, n)
  return orig_memcpy(dest, src, n)
}
```

Small Performance Overhead

Small Performance Overhead

CPU slowdown usually between 2x–5x

Small Performance Overhead

CPU slowdown usually between 2x–5x

Memory overhead 2x–3x

Small Performance Overhead

CPU slowdown usually between 2x–5x

Memory overhead 2x–3x

Address Sanitizer

Complementary Tools

Guard Malloc

Finds heap overruns and use-after-free

Adds guard pages before and after allocations

Does not require recompilation

Supported on OS X and in iOS simulator

Misses some bugs that Address Sanitizer finds

Complementary Tools

NSZombie

Catches Objective-C object over-releases

Replaces deallocated objects with “zombie” objects that trap

“Enable Zombie Objects” in Xcode

Zombies Instrument

Complementary Tools

Malloc Scribble

Helps detecting uninitialized variables

Fills allocated memory with 0xAA

Fills deallocated memory with 0x55

Complementary Tools

Leaks Instrument

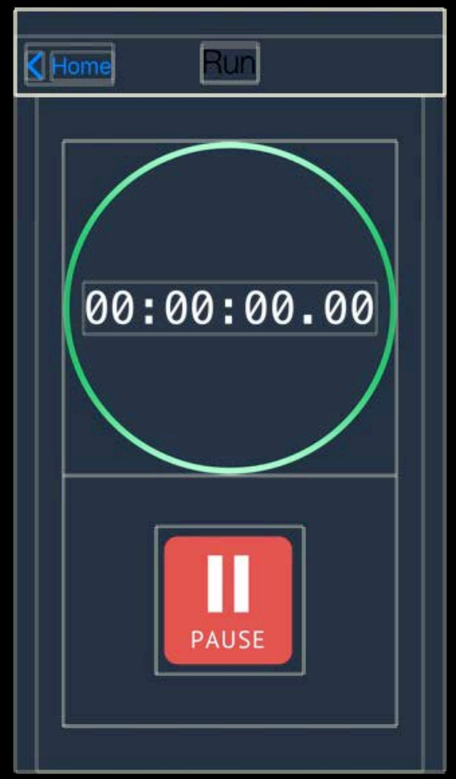
Helps detecting leaks

- Retain cycles
- Abandoned memory

Summary

Summary

View Debugger



Summary

View Debugger

Advanced Breakpoint Actions

Summary

View Debugger

Advanced Breakpoint Actions

Address Sanitizer

More Information

Documentation

Xcode Debugging

http://developer.apple.com/support/technical/debugging/

Address Sanitizer

http://clang.llvm.org/docs/AddressSanitizer.html

Apple Developer Forums

developer.apple.com/forums

Stefan Lesser

Developer Technologies Evangelist

slesser@apple.com

Related Sessions

What's New in LLDB	Nob Hill	Tuesday 2:30PM
UI Testing in Xcode	Nob Hill	Wednesday 11:00AM
Implementing UI Designs in Interface Builder	Pacific Hights	Wednesday 1:30PM
Continuous Integration and Code Coverage in Xcode	Presidio	Thursday 10:00AM
Profiling in Depth	Mission	Thursday 3:30PM

Labs

Instruments and Debugging	Developer Tools Lab B	Friday 9:00AM
Xcode Open Hours	Developer Tools Lab B	Friday 1:00PM

WWDC15