

# How / When to Instrument

---

- Source / IR Instrumentation
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

# How / When to Instrument

---

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build
- **Static Binary Rewriting**
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior

# How / When to Instrument

---

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build
- **Static Binary Rewriting**
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior
- **Dynamic Binary Instrumentation**
  - Valgrind, Pin, Qemu (& other Vms)
  - Can adapt at runtime, but less info than IR

# Phases of Dynamic Analysis

---

In general, 2-3 phases occur:

## 1) Instrumentation

- Add code to the program for data collection/analysis

# Phases of Dynamic Analysis

---

In general, 2-3 phases occur:

## 1) Instrumentation

- Add code to the program for data collection/analysis

## 2) Execution

- Run the program and analyze its actual behavior

# Phases of Dynamic Analysis

---

In general, 2-3 phases occur:

## 1) Instrumentation

- Add code to the program for data collection/analysis

## 2) Execution

- Run the program and analyze its actual behavior

## 3) (Optional) Postmortem Analysis

- Perform any analysis that can be deferred after termination

# Phases of Dynamic Analysis

---

In general, 2-3 phases occur:

## 1) Instrumentation

- Add code to the program for data collection/analysis

## 2) Execution

- Run the program and analyze its actual behavior

## 3) (Optional) Postmortem Analysis

- Perform any analysis that can be deferred after termination

Very, **very** common mistake to mix 1 & 2.

# Static Instrumentation

---

- 1) Compile whole program to IR
- 2) Instrument / add code directly to the IR
- 3) Generate new program that performs tracing/analysis
- 4) Execute



# Dynamic Binary Instrumentation

- 1) Compile program as usual
- 2) Run program under analysis framework  
(Valgrind, PIN, Qemu, etc)
- 3) Instrument & execute in same command:
  - Fetch & instrument each basic block individually
  - Execute each basic block

```
valgrind --tool=memcheck ./myBuggyProgram
```

# Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the `clang` compiler
- Static instrumentation

# Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the `clang` compiler
- Static instrumentation
- Finds:
  - Use-after-free
  - {heap,stack,global}-buffer overflows

# Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the `clang` compiler
- Static instrumentation
- Finds:
  - Use-after-free
  - {heap,stack,global}-buffer overflows
- Used extensively in Google programs like Chrome

# Example: Address Sanitizer

---

How?

- Replaces `malloc` & `free`

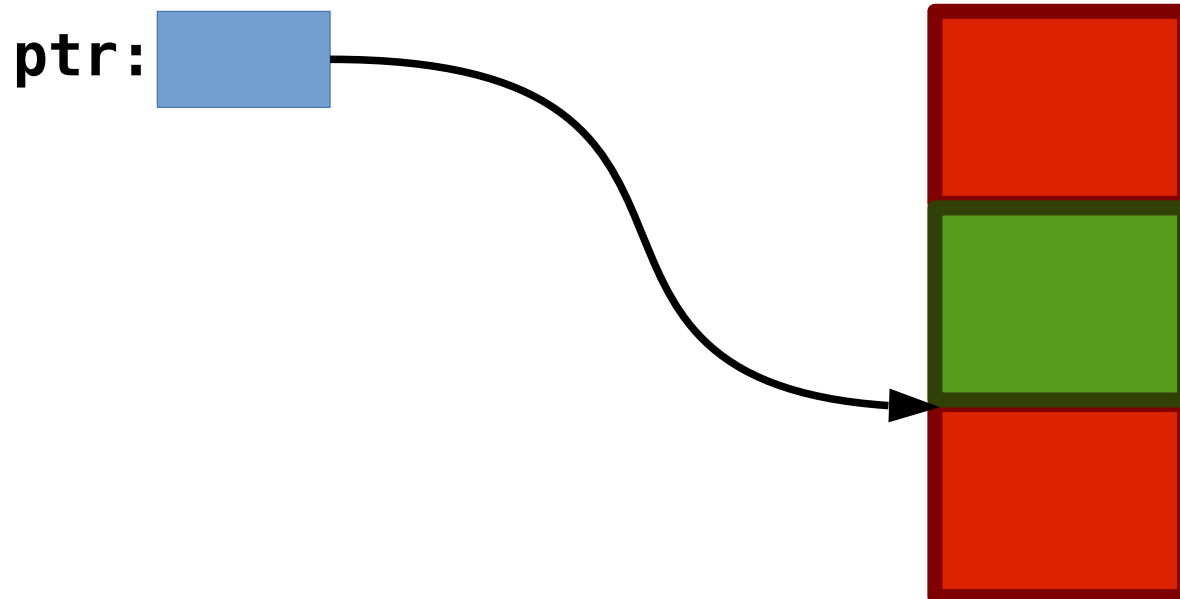
# Example: Address Sanitizer

---

How?

- Replaces `malloc` & `free`
- Memory `around` `malloced` chunks is *poisoned*

```
ptr = malloc(sizeof(MyStruct));
```



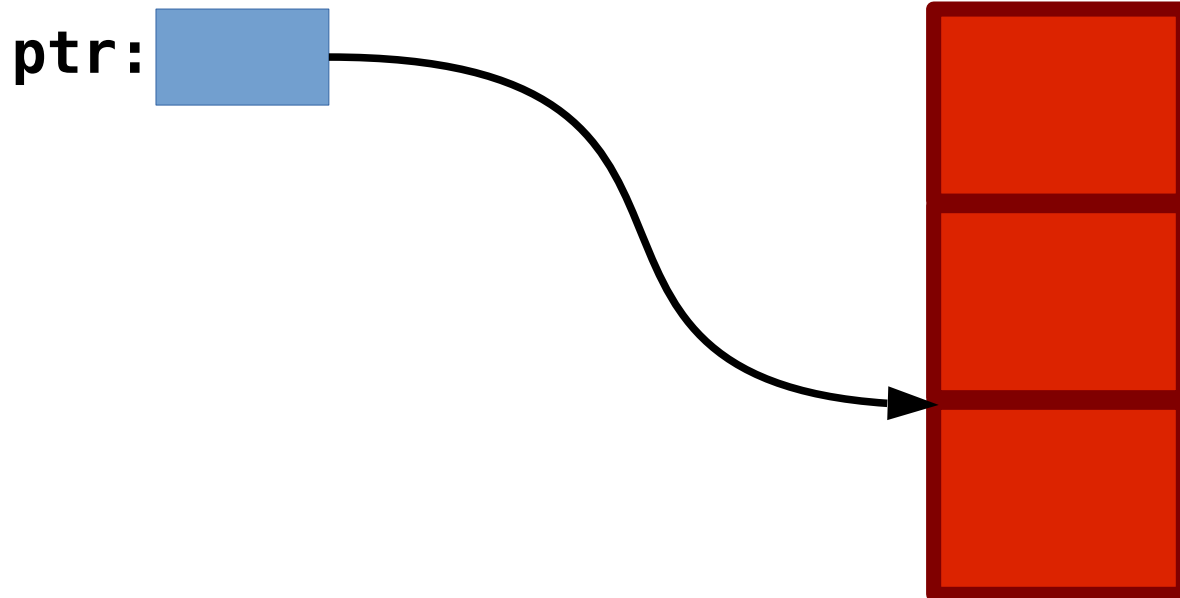
# Example: Address Sanitizer

---

How?

- Replaces `malloc` & `free`
- Memory around malloced chunks is *poisoned*
- **Freed** memory is *poisoned*

```
free(ptr);
```



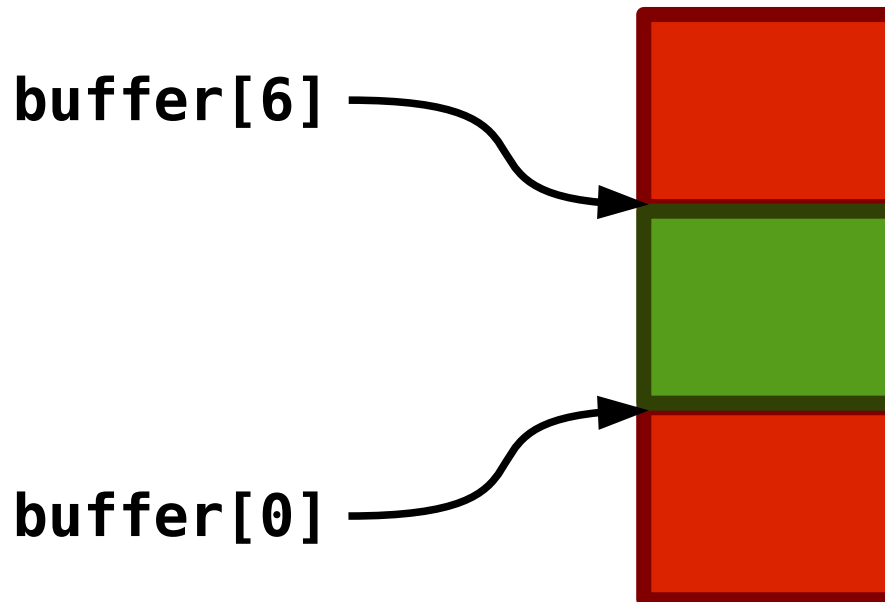
# Example: Address Sanitizer

---

How?

- Replaces `malloc` & `free`
- Memory around malloced chunks is *poisoned*
- Freed memory is *poisoned*
- Space **around buffers** is *poisoned*

```
void foo() {  
    int buffer[5];  
}
```





# Example: Address Sanitizer

How?

- Replaces `malloc` & `free`
- Memory around malloced chunks is *poisoned*
- Freed memory is *poisoned*
- Space around buffers is *poisoned*
- Any access of a poisoned value reports an error.

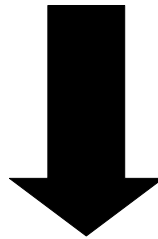
...

# Example: Address Sanitizer

---

How?

```
*address = ...
```



Instrumentation

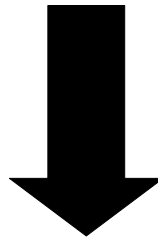
```
if (IsPoisoned(address)) {  
    ReportError(address, kAccessSize, kIsWrite);  
}  
*address = ...;
```

# Example: Address Sanitizer

---

How?

```
*address = ...
```



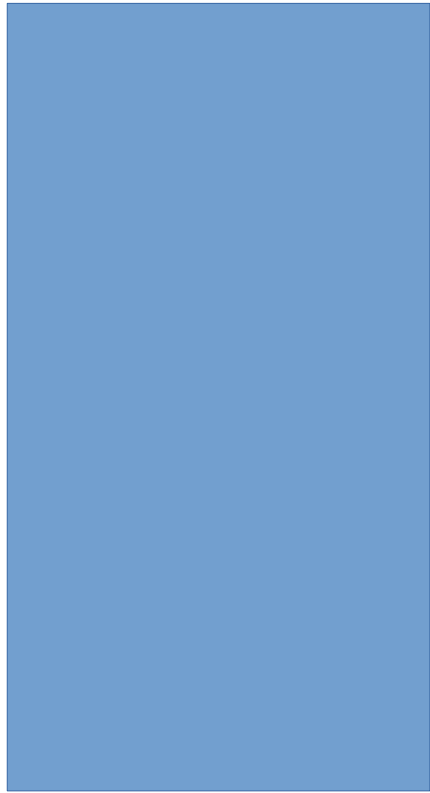
Instrumentation

```
if (IsPoisoned(address)) {  
    ReportError(address, kAccessSize, kIsWrite);  
}  
*address = ...;
```

Difficult! Why?

- Instrumenting every memory access is costly
- Tracking the status of *all memory* is tricky

# Example: Address Sanitizer



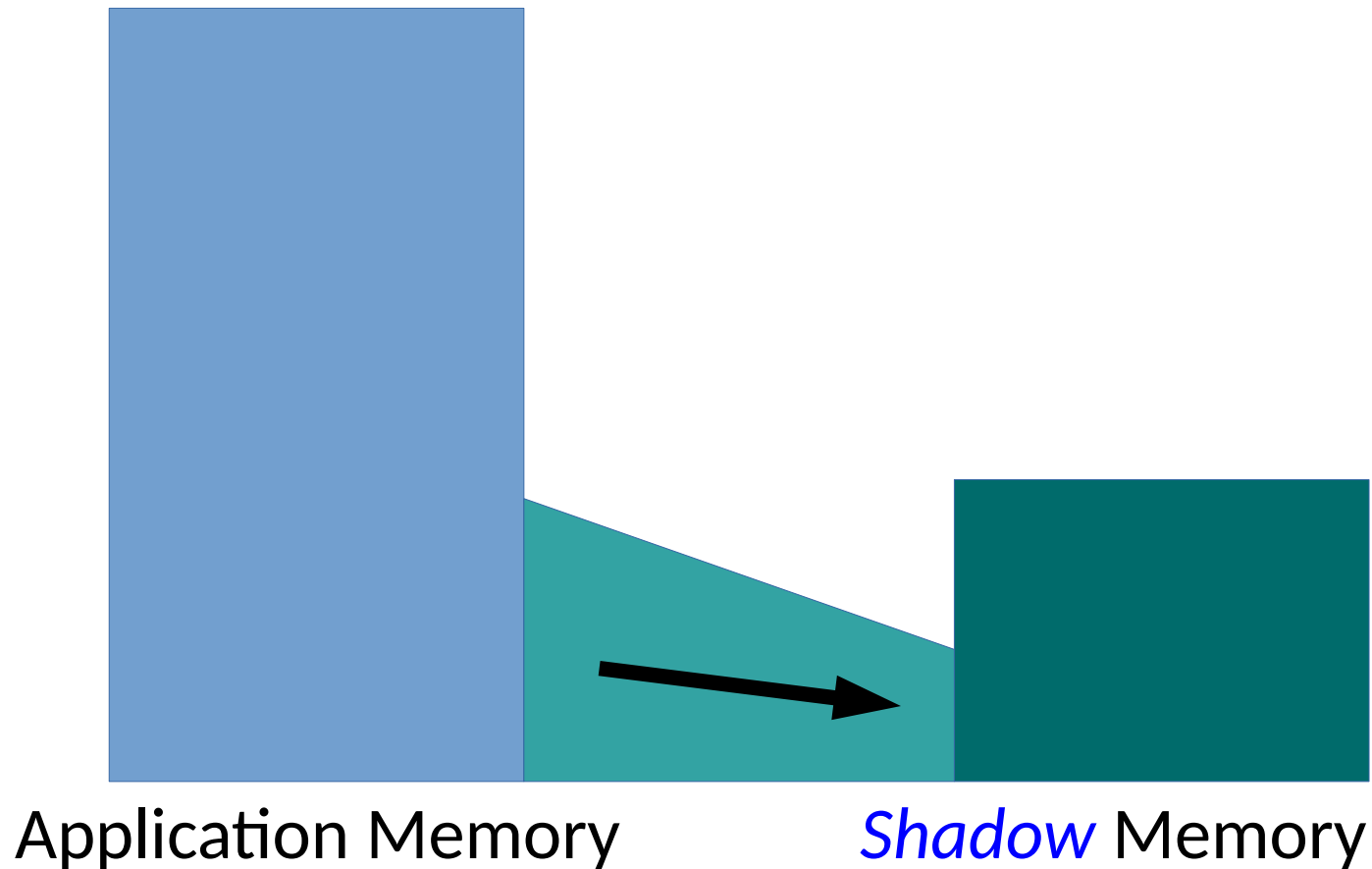
Application Memory

Need to know whether *any byte* of application memory is poisoned.

# Example: Address Sanitizer

---

- Maintain 2 views on memory:



# Example: Address Sanitizer

---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table

# Example: Address Sanitizer

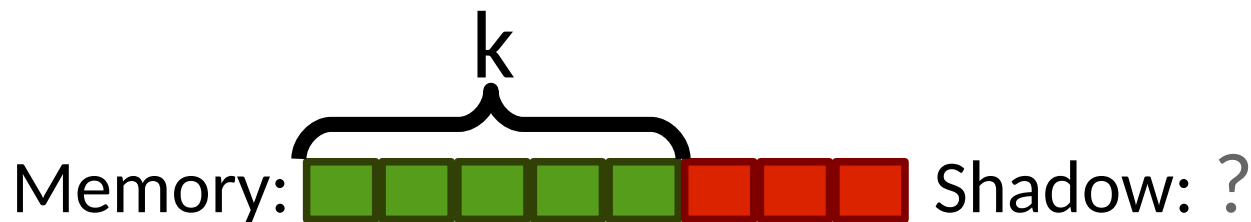
---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables

# Example: Address Sanitizer

---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
- In Asan:
  - In an 8 byte chunk, only first k may be addressable





# Example: Address Sanitizer

---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.

Memory:  Shadow: 0

# Example: Address Sanitizer

---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.
  - All 8 bytes poisoned: shadow value is negative.

Memory:  Shadow: -1

# Example: Address Sanitizer

---

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.
  - All 8 bytes poisoned: shadow value is negative.
  - First k bytes are unpoisoned: shadow value is k.

Memory:  Shadow: 5

# Example: Address Sanitizer

---

- (64bit) Shadow Mapping:
  - Preallocate large block of memory
  - $\text{Shadow} = (\text{Mem} \gg 3) + 0x7fff8000;$

# Example: Address Sanitizer

---

- (64bit) Shadow Mapping:
  - Preallocate large block of memory
  - $\text{Shadow} = (\text{Mem} \gg 3) + 0x7fff8000;$
- The shadow memory itself must also be considered poisoned.

Why?!