

Address Sanitizer

A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening, Alexander Potapenko,
Dmitry Vyukov

Daejin Lee

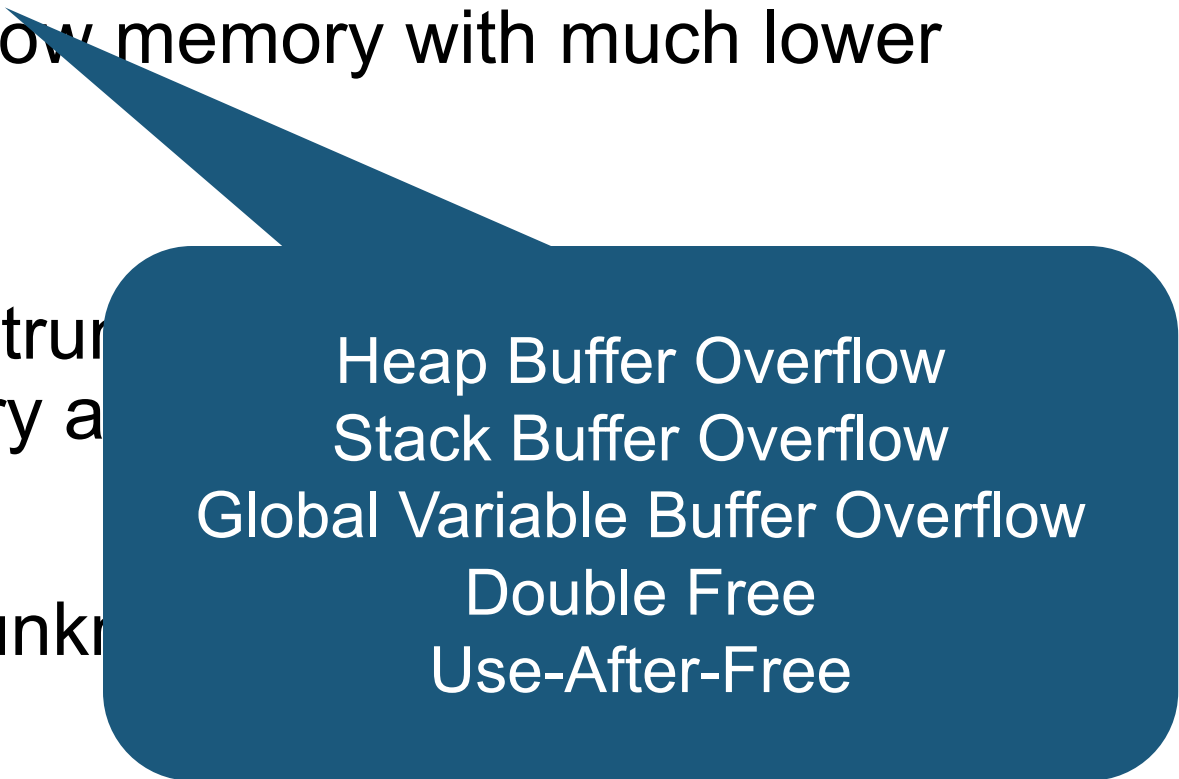
5th October 2020 @ IS893

Introduction

- ASan tries to detect memory bugs by leveraging the comprehensiveness of shadow memory with much lower overhead
- Consists of two parts: an instrumentation module and run-time library to check each memory access(based on LLVM)
- Found over 300 previously unknown bugs in the Chromium browser

Introduction

- ASan tries to detect **memory bugs** by leveraging the comprehensiveness of shadow memory with much lower overhead
- Consists of two parts: an instrumentation library to check each memory access
- Found over 300 previously unknown bugs in Chrome browser



Heap Buffer Overflow
Stack Buffer Overflow
Global Variable Buffer Overflow
Double Free
Use-After-Free

Related Work

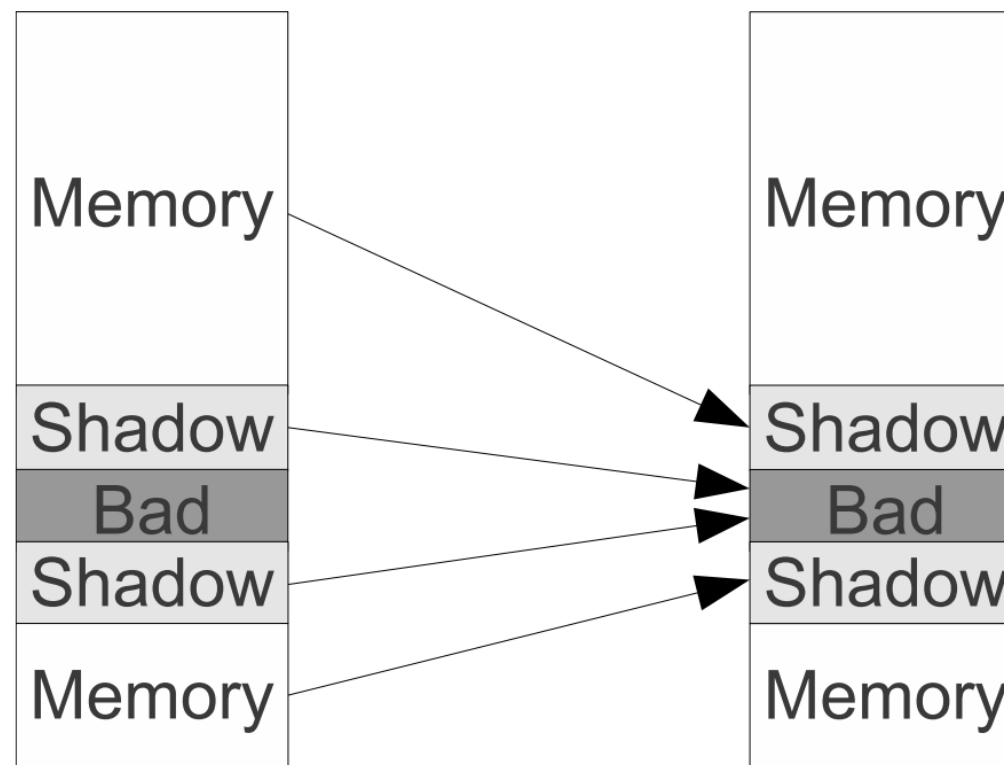
- Valgrind(VEE '07), Dr.Memory(CGO '11)
 - Multi-level translation schemes.
- Mudflap(RedHat Inc.)
 - Compile-time Instrumentation
- GuardMalloc(Mac OSX Lib), Page Heap(Windows Lib)
 - CPU page protection

Shadow Byte: Memory State



Shadow Memory

- Asan uses one-eighth of the virtual address space to its shadow memory.
 - Address Compute:
 $(Addr \gg Scale) + Offset$
 - With PIE enabled, a zero offset can be used to simplify instrumentation



Instrumentation – 8 bytes access

- ASan computes the address of the corresponding shadow byte, loads that byte, and checks whether it is zero

```
ShadowAddr = (Addr >> 3) + Offset;  
if (*ShadowAddr != 0)  
    ReportAndCrash(Addr);
```

Instrumentation – 1,2,4 bytes access

- If the shadow value is positive, we need to compare the 3 last bits of the address with k

```
ShadowAddr = (Addr >> 3) + Offset;  
k = *ShadowAddr;  
if (k != 0 && ((Addr & 7) + AccessSize > k))  
    ReportAndCrash(Addr);
```


Runtime-Library

- Initializes shadow memory at startup
- Malloc/free are replaced with a custom implementation
 - Allocate extra memory, the redzone(unaddressable), around the returned region
 - Record current call stack
 - Free function poisons the entire memory region(And prohibit not to be called by malloc any time soon)

Report Example – use-after-free

ERROR: AddressSanitizer heap-use-after-free on address 0x7fe8740a6214

at pc 0x40246f bp 0x7fffe5e463e0 sp 0x7fffe5e463d8

READ of size 4 at 0x7fe8740a6214 thread T0

#0 0x40246f in main example_UseAfterFree.cc:4

#1 0x7fe8740e4c4d in __libc_start_main ??:0

0x7fe8740a6214 is located 4 bytes inside of 400-byte region

freed by thread T0 here:

#0 0x4028f4 in operator delete[](void*) _asan_rtl_

#1 0x402433 in main example_UseAfterFree.cc:4

previously allocated by thread T0 here:

#0 0x402c36 in operator new[](unsigned long) _asan_rtl_

#1 0x402423 in main example_UseAfterFree.cc:2

Stack and Globals

- For stack objects, the red-zones are created and poisoned at run-time.
- For globals, the redzones are created at compile time and the addresses of the redzones are passed to the runtime library at startup
- They used 32 bytes red-zones(plus up to 31 bytes for alignment).

Stack and Globals: Instrumentation

```
void foo() {  
    char a[10];  
    <function body>  
}
```

Stack and Globals: Instrumentation

```
void foo() {  
    char rz1[32]  
    char arr[10];  
    char rz2[32-10+32];  
    unsigned int *shadow = (unsigned*)((long)rz1>>8)+Offset);  
    shadow[0] = 0xffffffff; // rz1  
    shadow[1] = 0xffff0200; // arr and rz2  
    shadow[2] = 0xffffffff; // rz2  
    <function body> // un-poison all.  
    shadow[0] = shadow[1] = shadow[2] = 0;  
}
```

False Negative

// 8-aligned

```
int *a = new int[2];
```

// Fetch memory range [6-9]

```
int *u = (int*)((char*)a + 6);
```

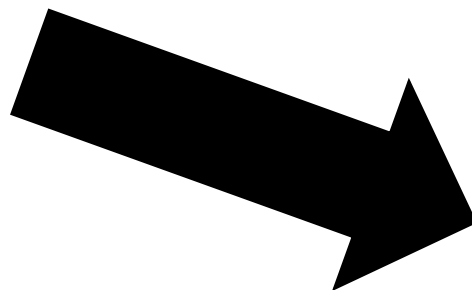
// Access to range [6-9]

```
*u = 1;
```

False Negative

```
// 8-aligned
```

```
int *a = new int[2];
```



4 bytes

4 bytes

```
// Fetch memory range [6-9]
```

```
int *u = (int*)((char*)a + 6);
```

```
// Access to range [6-9]
```

```
*u = 1;
```

False Negative

// 8-aligned

```
int *a = new int[2];
```

// Fetch memory range [6-9]

```
int *u = (int*)((char*)a + 6);
```

// Access to range [6-9]

```
*u = 1;
```



False Negative

// 8-aligned

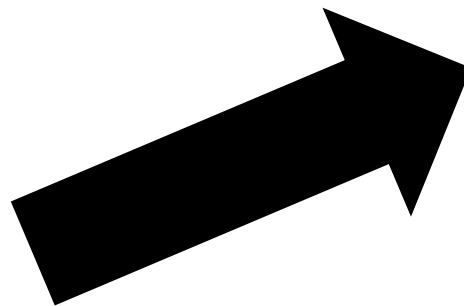
```
int *a = new int[2];
```

// Fetch memory range [6-9]

```
int *u = (int*)((char*)a + 6);
```

// Access to range [6-9]

```
*u = 1;
```



False Positive: Load Widening

```
struct X {  
    char a, b, c;  
};  
  
void foo() {  
    X x;  
    ... .. = x.a + x.c;  
}
```

False Positive: Load Widening

```
struct X {  
    char a, b, c;  
};
```



3 bytes size structure

```
void foo() {  
    X x;  
    ... .. = x.a + x.c;  
}
```

False Positive: Load Widening

```
struct X {  
    char a, b, c;  
};
```

```
void foo() {  
    X x;  
    ... .. = x.a + x.c;  
}
```



Load Widening
transforms `x.a + x.c` into
one 4-byte Load

Evaluation – Setup

- HP Z600 Machine 2 quad-core Intel Xeon E5620 CPUs and 24GB RAM
- Compared instrumented binaries with the binaries build using the regular LLVM compiler

1.73x slowdown (reads & writes)

1.26x slowdown (writes only)

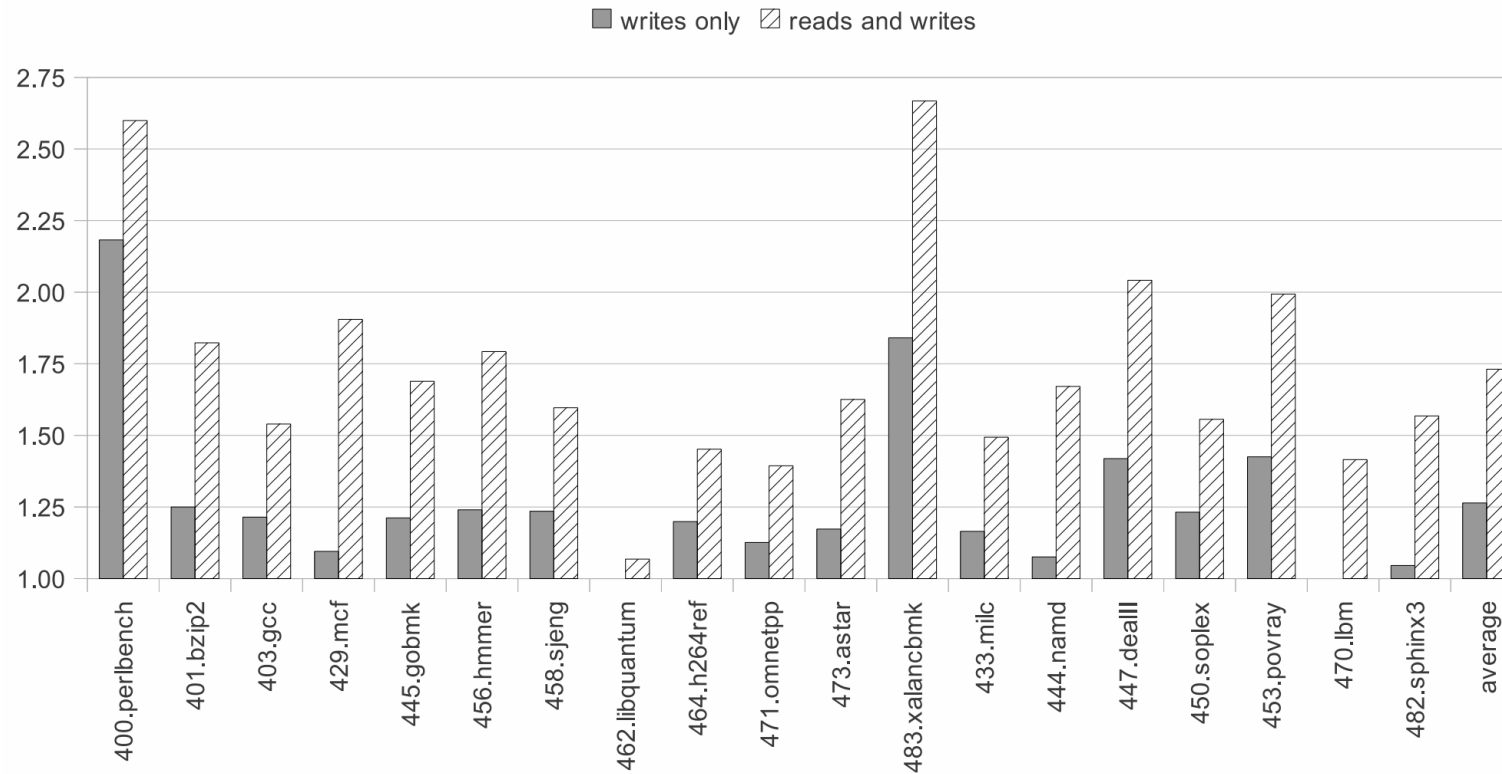


Figure 2: The average slowdown on SPEC CPU2006 on 64-bit Linux.

3.37x Heap Memory overhead

Table 1: Memory usage with AddressSanitizer (MB)

Benchmark	Original	Instrumented	Increase
400.perlbench	670	2168	3.64x
401.bzip2	858	1618	2.12x
403.gcc	893	4133	5.21x
429.mcf	1684	2098	1.40x
445.gobmk	37	369	11.22x
456.hmmer	33	582	19.84x
458.sjeng	180	249	1.56x
462.libquantum	104	930	10.06x
464.h264ref	72	439	6.86x
471.omnetpp	181	787	4.89x
473.astar	343	1214	3.98x
483.xalancbmk	434	1688	4.38x
433.milc	694	1618	2.62x
444.namd	58	146	2.83x
447.dealII	807	2602	3.63x
450.soplex	637	2479	4.38x
453.povray	17	371	24.55x
470.lbm	417	550	1.48x
482.sphinx3	52	426	9.22x
total	8171	24467	3.37x

2.5x Stack Memory Overhead

Table 2: Stack increase with AddressSanitizer (KB)

Benchmark	Original	Instrumented	Increase
400.perlbench	568	1740	3.06x
445.gobmk	184	264	1.43x
458.sjeng	828	848	1.02x
483.xalancbmk	2116	4720	2.23x
453.povray	88	96	1.09x
482.sphinx3	248	252	1.02x

Comparison

- Valgrind, Dr.Memory incur 20x and 10x slowdowns on CPU2006 benchmark.
 - But these tools also detect uninitialized read, Memory leaks including out-of-bounds and use-after-free)

Questions? 😊