[1/2] Fantastic C++ Bugs
and Where to Find Them

[2/2] Find scary C++ bugs before they find you

Konstantin Serebryany, Google
May 2014 @compsciclub.ru

Agenda

- AddressSanitizer (aka ASan)
  - detects use-after-free and buffer overflows (C++)
- ThreadSanitizer (aka TSan)
  - detects data races (C++ & Go)
- MemorySanitizer (aka MSan)
  - detects uninitialized memory reads (C++)
- Related research areas

AddressSanitizer

addressability bugs

AddressSanitizer overview

- Finds
  - buffer overflows (stack, heap, globals)
  - heap-use-after-free, stack-use-after-return
  - some more
- Compiler module (LLVM, GCC)
  - instruments all loads/stores
  - inserts redzones around stack and global Variables
- Run-time library
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages

ASan report example: global-buffer-overflow

```
int global_array[100] = {-1};
int main(int argc, char **argv) {
  return global_array[argc + 100];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc ; ./a.out

==10538== ERROR: AddressSanitizer global-buffer-overflow
READ of size 4 at 0x000000415354 thread T0
    #0 0x402481 in main a.cc:3
    #1 0x7f0a1c295c4d in __libc_start_main ??:0
    #2 0x402379 in _start ??:0
0x000000415354 is located 4 bytes to the right of global
  variable 'global_array' (0x4151c0) of size 400
```

ASan report example: stack-buffer-overflow

```
int main(int argc, char **argv) {
  int stack_array[100];
  stack_array[1] = 0;
  return stack_array[argc + 100];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc; ./a.out

==10589== ERROR: AddressSanitizer stack-buffer-overflow
READ of size 4 at 0x7f5620d981b4 thread T0
    #0 0x4024e8 in main a.cc:4
Address 0x7f5620d981b4 is located at offset 436 in frame
 <main> of T0's stack:
  This frame has 1 object(s):
    [32, 432) 'stack_array'
```

ASan report example: heap-buffer-overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  int res = array[argc + 100];  // BOOM
  delete [] array;
  return res;
}
% clang++ -O1 -fsanitize=address a.cc; ./a.out

==10565== ERROR: AddressSanitizer heap-buffer-overflow
READ of size 4 at 0x7fe4b0c76214 thread T0
    #0 0x40246f in main a.cc:3
0x7fe4b0c76214 is located 4 bytes to the right of 400-
  byte region [0x7fe..., 0x7fe...)
allocated by thread T0 here:
    #0 0x402c36 in operator new[](unsigned long)
    #1 0x402422 in main a.cc:2
```

ASan report example: use-after-free

```
int main(int argc, char **argv) {
   int *array = new int[100];
   delete [] array;
   return array[argc];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte
region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long) _asan_rtl_
   #1 0x4042f3 in main a.cc:2
```

ASan report example: stack-use-after-return

```
int *g;
void LeakLocal() {
  int local;
  g = &local;
}
```

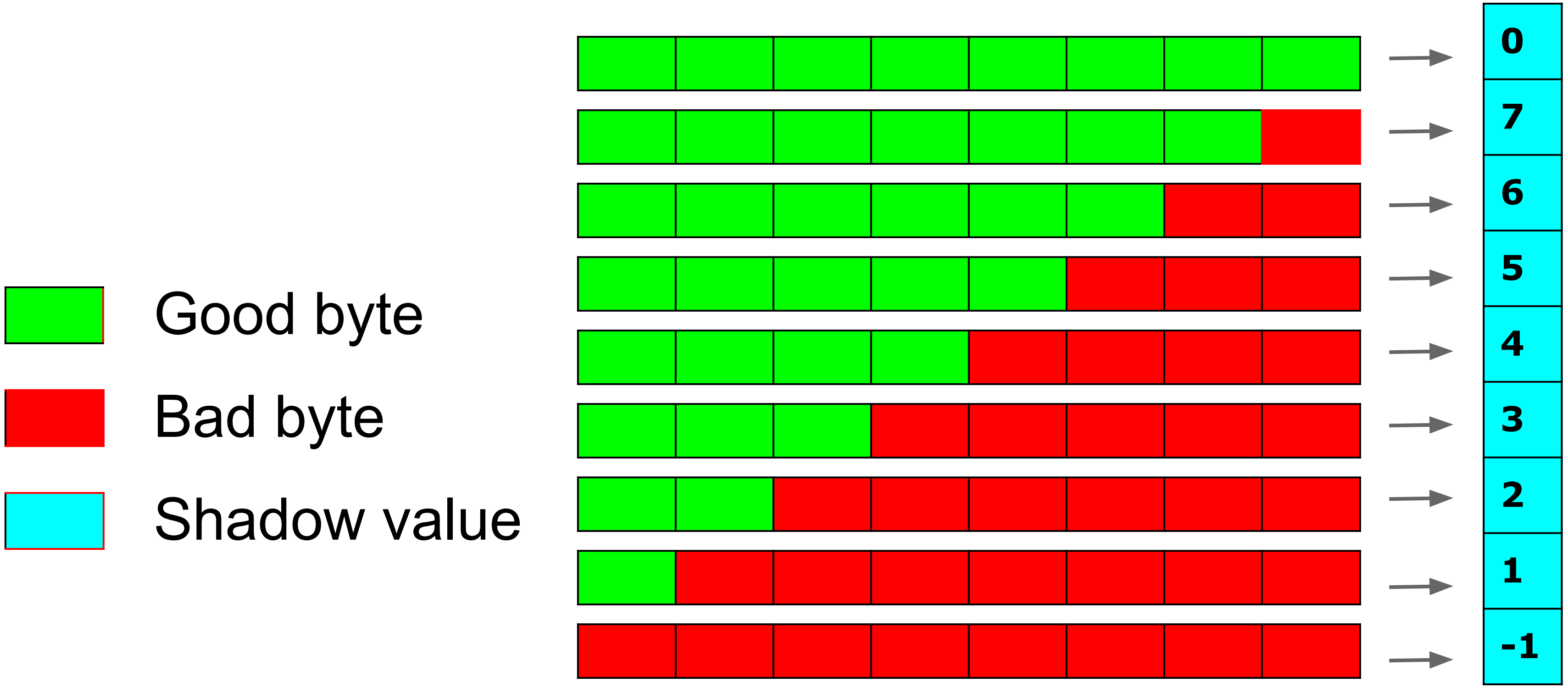
```
int main() {
  LeakLocal();
  return *g;
}
```

```
% clang -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_stack_use_after_return=1 ./a.out

==19177==ERROR: AddressSanitizer: stack-use-after-return
READ of size 4 at 0x7f473d0000a0 thread T0
    #0 0x461ccf in main   a.cc:8

Address is located in stack of thread T0 at offset 32 in frame
    #0 0x461a5f in LeakLocal()  a.cc:2
  This frame has 1 object(s):
    [32, 36) 'local' <== Memory access at offset 32
```

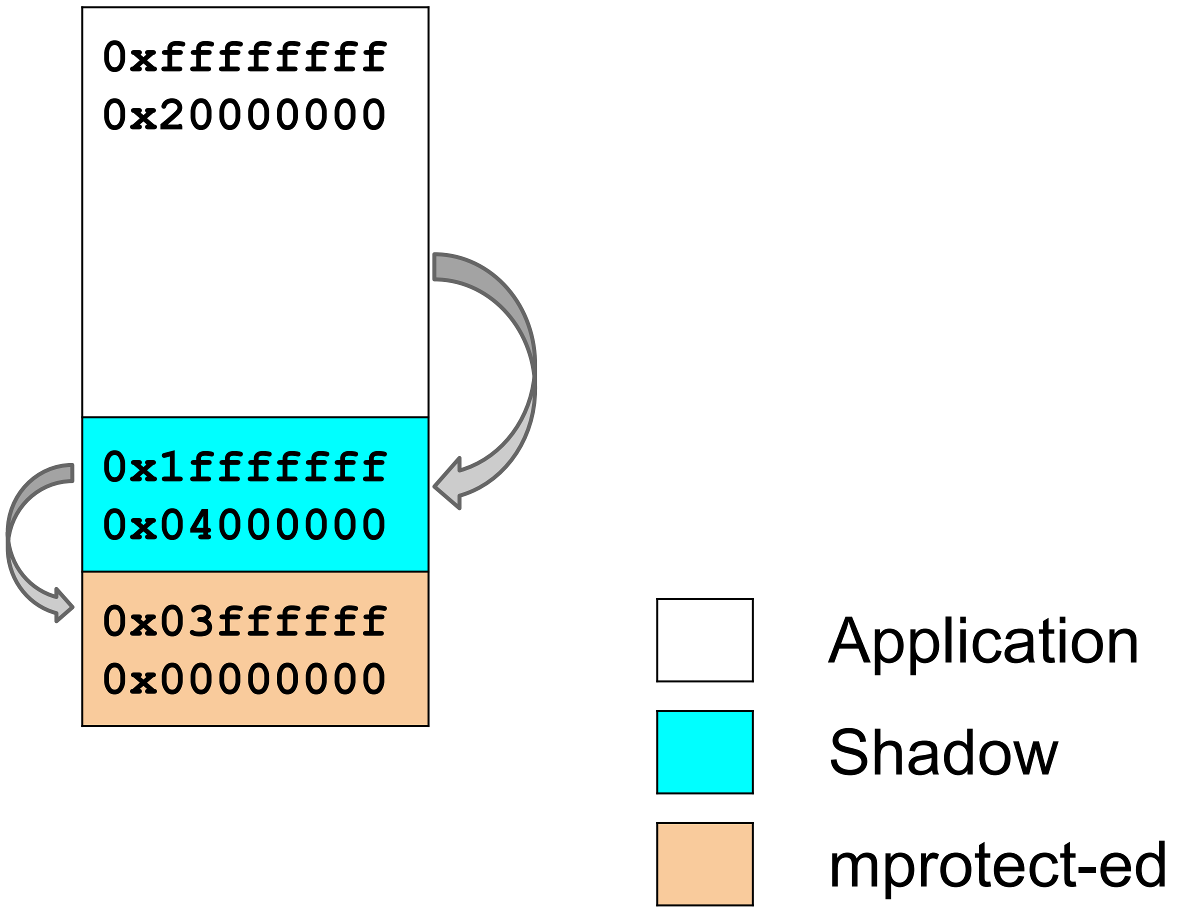
ASan shadow byte

Any aligned 8 bytes may have 9 states:

N good bytes and 8 - N bad (0<=N<=8)



ASan virtual address space



ASan instrumentation: 8-byte access

```
*a = ...
```

↓

```
char *shadow = a >> 3;
if (*shadow)
  ReportError(a);
*a = ...
```

ASan instrumentation: N-byte access (1, 2, 4)

```
*a = ...
```

```
char *shadow = a >> 3;
if (*shadow &&
    *shadow <= ((a&7)+N-1))
  ReportError(a);
*a = ...
```

Instrumentation example (x86_64)

```
mov     %rdi,%rax
shr     $0x3,%rax              # shift by 3
cmpb    $0x0,0x7fff8000(%rax) # load shadow
je 1f   <foo+0x1f>
ud2a                           # generate SIGILL*
movq    $0x1234,(%rdi)         # original store
```

*** May use call instead of UD2**

Instrumenting stack frames

```
void foo() {
  char a[328];



  <------------- CODE ------------->
}
```

Instrumenting stack frames

```
void foo() {
  char rz1[32];  // 32-byte aligned
  char a[328];
  char rz2[24];
  char rz3[32];
  int  *shadow = (&rz1 >> 3) + kOffset;
  shadow[0] = 0xffffffff;    // poison rz1

  shadow[11] = 0xffffff00;  // poison rz2
  shadow[12] = 0xffffffff;  // poison rz3
  <------------- CODE ------------->
  shadow[0] = shadow[11] = shadow[12] = 0;
}
```

Instrumenting globals

```
int a;
```

```
struct {
  int original;
  char redzone[60];
} a;  // 32-aligned
```

Malloc replacement

- Insert redzones around every allocation
  - poison redzones on malloc
- Delay the reuse of freed memory
  - poison entire memory region on free
- Collect stack traces for every malloc/free

ASan *marketing* slide

- 2x slowdown (Valgrind: 20x and more)
- 1.5x-3x memory overhead

- 2000+ bugs found in Chrome in 3 years

- 2000+ bugs found in Google server software

- 1000+ bugs everywhere else
  - Firefox, FreeType, FFmpeg, WebRTC, libjpeg-turbo, Perl, Vim, LLVM, GCC, MySQL

ASan and Chrome

- Chrome was the first ASan user (May 2011)
- Now all existing tests are running with ASan
- Fuzzing at massive scale (ClusterFuzz), 2000+ cores
  - Generate test cases, minimize, de-duplicate
  - Find regression ranges, verify fixes
- Over 2000 security bugs found in 2.5 years
  - External researchers found 100+ bugs
  - Most active: Oulu University (Finland)

ThreadSanitizer

data races

ThreadSanitizer

- Detects data races
- Compile-time instrumentation (LLVM, GCC)
  - Intercepts all reads/writes
- Run-time library
  - Malloc replacement
  - Intercepts all synchronization
  - Handles reads/writes

TSan report example: data race

```
void Thread1() { Global = 42; }
int main() {
  pthread_create(&t, 0, Thread1, 0);
  Global = 43;
  ...
% clang -fsanitize=thread -g a.c && ./a.out

WARNING: ThreadSanitizer: data race (pid=20373)
  Write of size 4 at 0x7f... by thread 1:
    #0 Thread1 a.c:1
  Previous write of size 4 at 0x7f... by main thread:
    #0 main a.c:4
  Thread 1 (tid=20374, running) created at:
    #0 pthread_create ??:0
    #1 main a.c:3
```

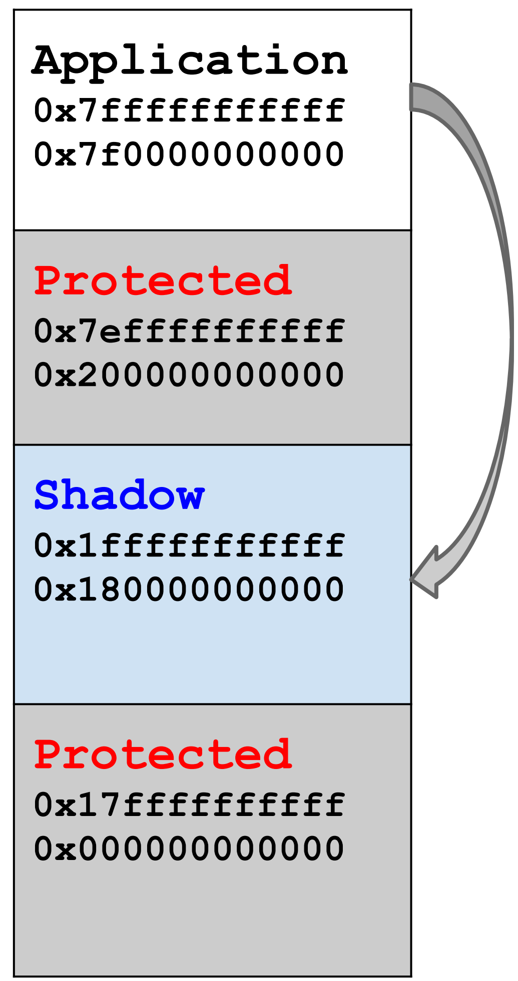
Compiler instrumentation

```
void foo(int *p) {
  *p = 42;
}
```

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

Direct shadow mapping (64-bit Linux)

```
Shadow = 4 * (Addr & kMask);
```

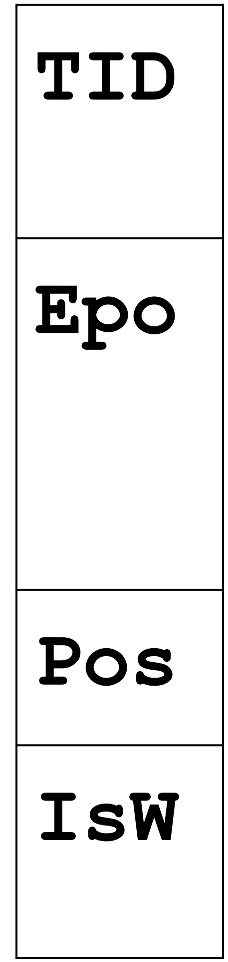


Shadow cell

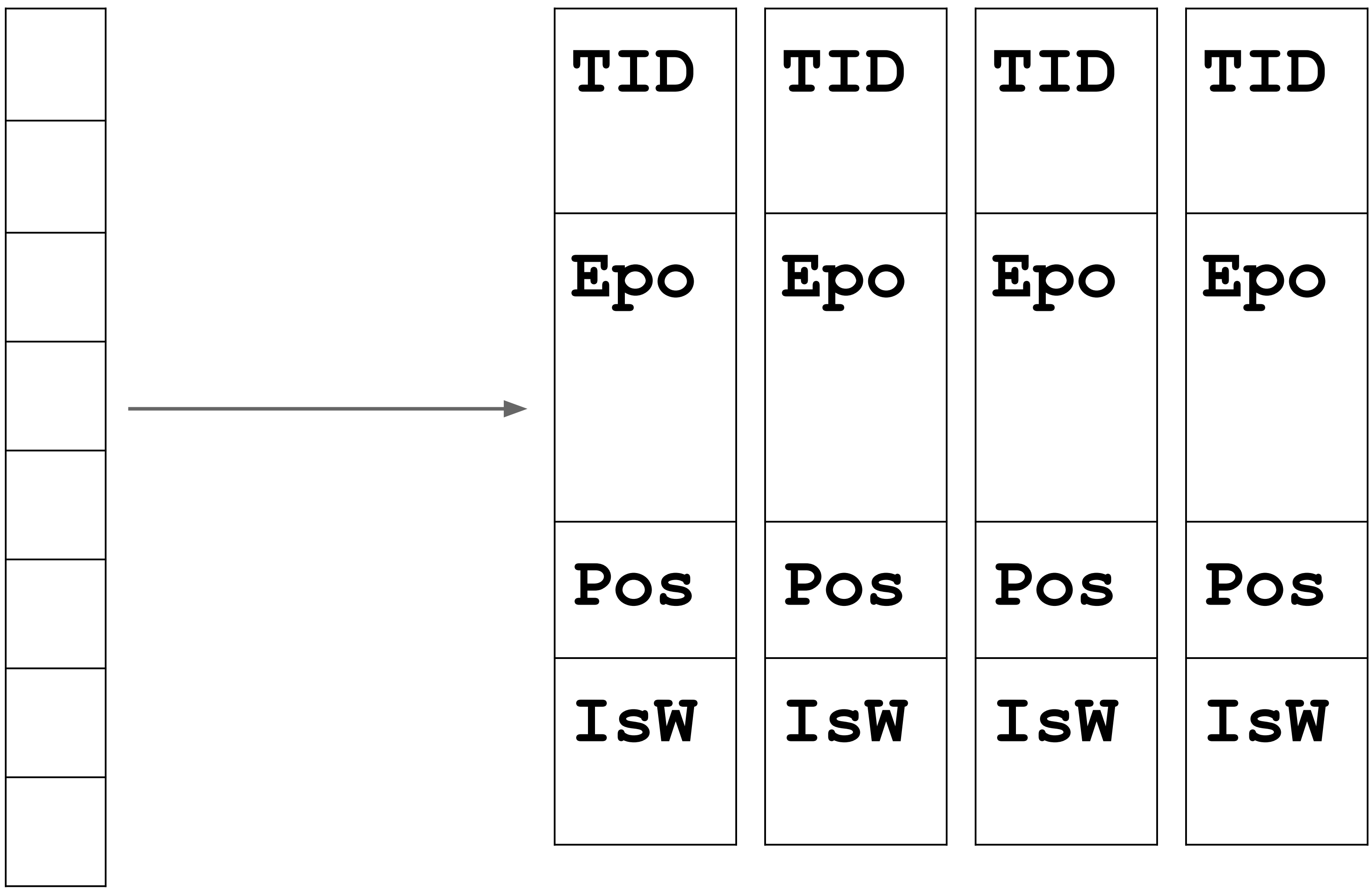
An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

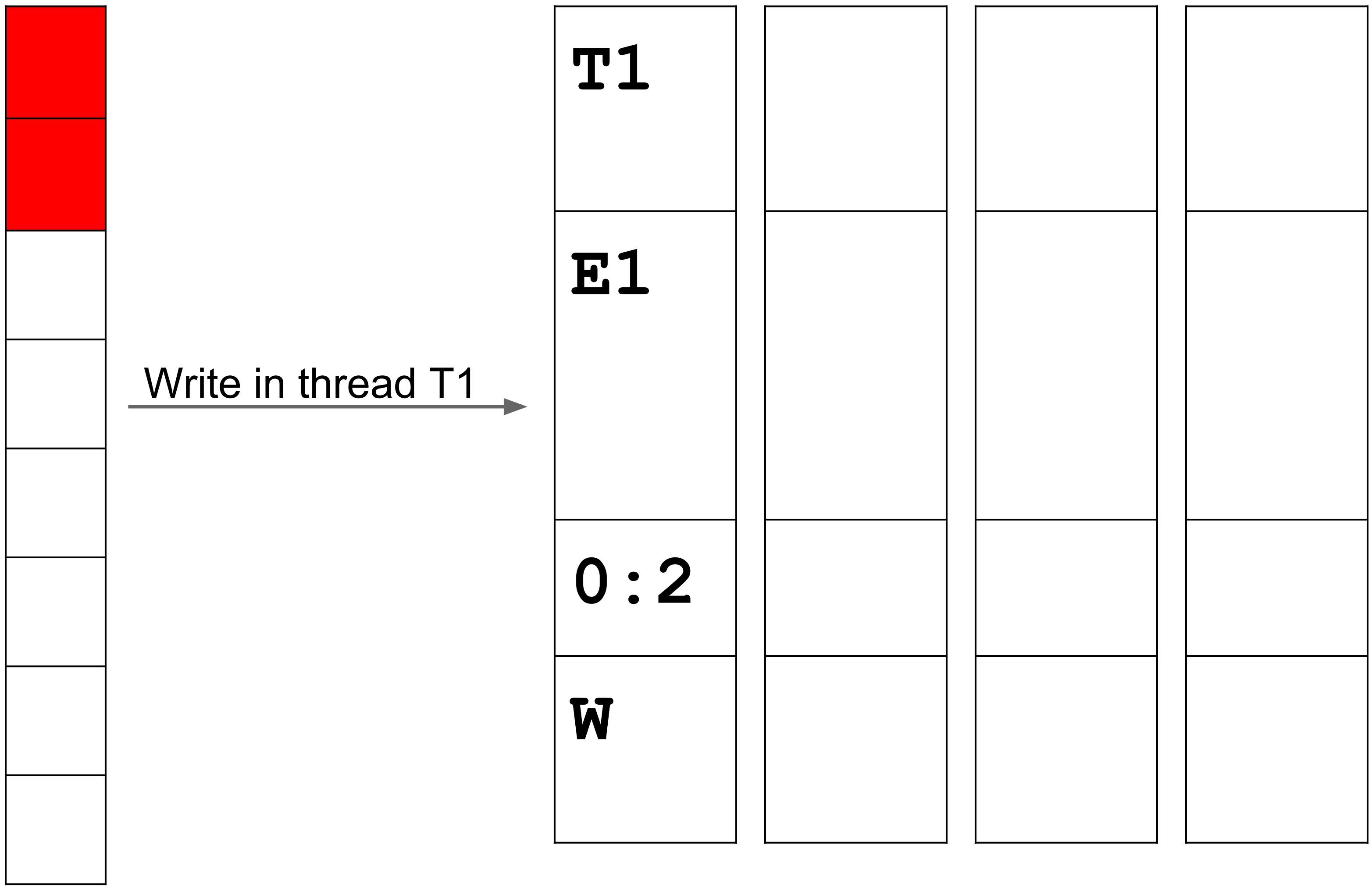
Full information (no more dereferences)



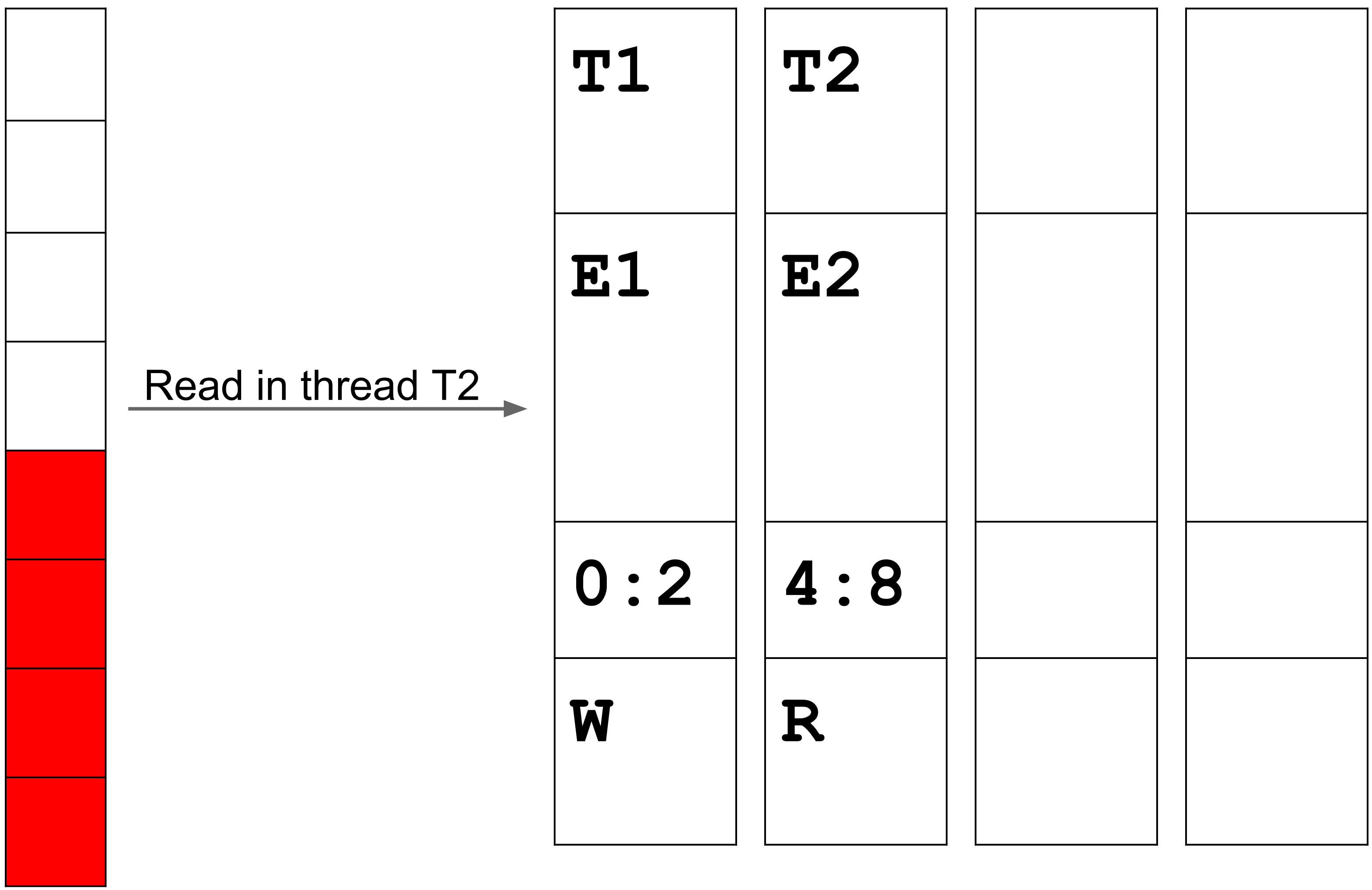
4 shadow cells per 8 app. bytes



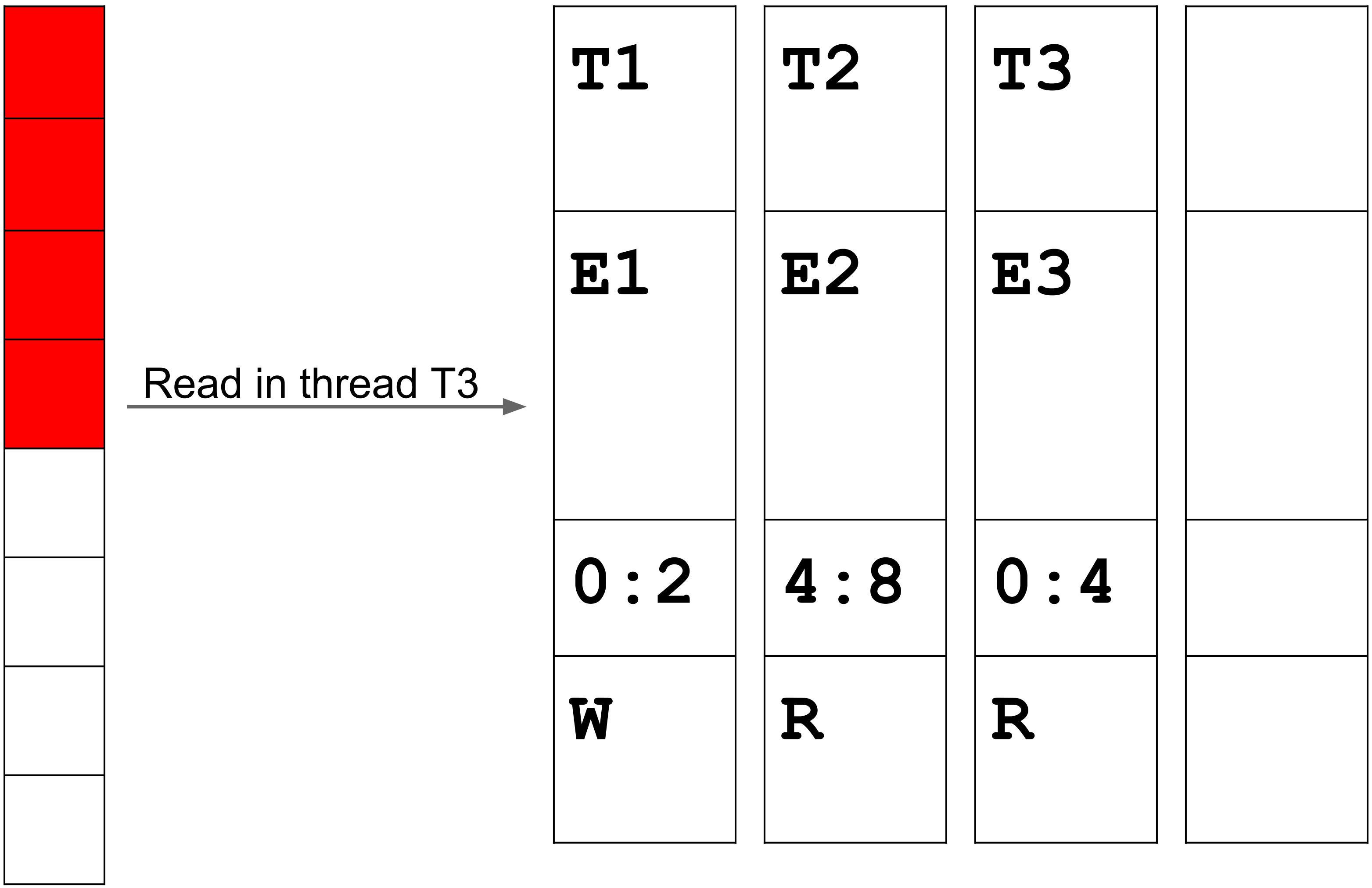
Example: first access



Example: second access



Example: third access



Example: race?

Race if **E1** does not "happen-before" **E3**

T1	T2	T3	
E1	E2	E3	
0:2	4:8	0:4	
W	R	R	

Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison
- Somewhat similar to FastTrack (PLDI'09)

Stack trace for previous access

- Important to understand the report

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable

- Replay the event buffer on report
  - Unlimited number of frames

TSan overhead

- CPU: 4x-10x
- RAM: 5x-8x

Trophies

- 500+ races in Google server-side apps (C++)
  - Scales to huge apps
- 100+ races in Go programs
  - 25+ bugs in Go stdlib
- 100+ races in Chrome

Key advantages

- Speed
  - > 10x faster than other tools
- Native support for atomics
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)

Limitations

- Only 64-bit Linux
  - Relies on atomic 64-bit load/store
  - Requires lots of RAM
- Does not instrument (yet):
  - pre-built libraries
  - inline assembly

MemorySanitizer
uninitialized memory reads (UMR)

MSan report example: UMR

```
int main(int argc, char **argv) {
  int x[10];
  x[0] = 1;
  if (x[argc]) return 1;
  ...
% clang -fsanitize=memory a.c -g; ./a.out

WARNING: MemorySanitizer: UMR (uninitialized-memory-read)
    #0 0x7ff6b05d9ca7 in main stack_umr.c:4
  ORIGIN: stack allocation: x@main
```

Shadow memory

- Bit to bit shadow mapping
  - 1 means 'poisoned' (uninitialized)
- Uninitialized memory:
  - Returned by malloc
  - Local stack objects (poisoned at function entry)
- Shadow is unpoisoned when constants are stored

Shadow propagation

Reporting every load of uninitialized data is too noisy.

```
struct {
  char x;
  // 3-byte padding
  int y;
}
```

It's OK to copy uninitialized data around.

Uninit calculations are OK, too, as long as the result is discarded. People do it.

Shadow propagation

$A = B \ll C$: $A' = B' \ll C$

$A = B \,\&\, C$: $A' = (B' \,\&\, C') \mid (B \,\&\, C') \mid (B' \,\&\, C)$

$A = B + C$: $A' = B' \mid C'$ *(approx.)*

Report errors only on some uses: conditional branch, syscall argument (visible side-effect).

Tracking origins

- Where was the poisoned memory allocated?

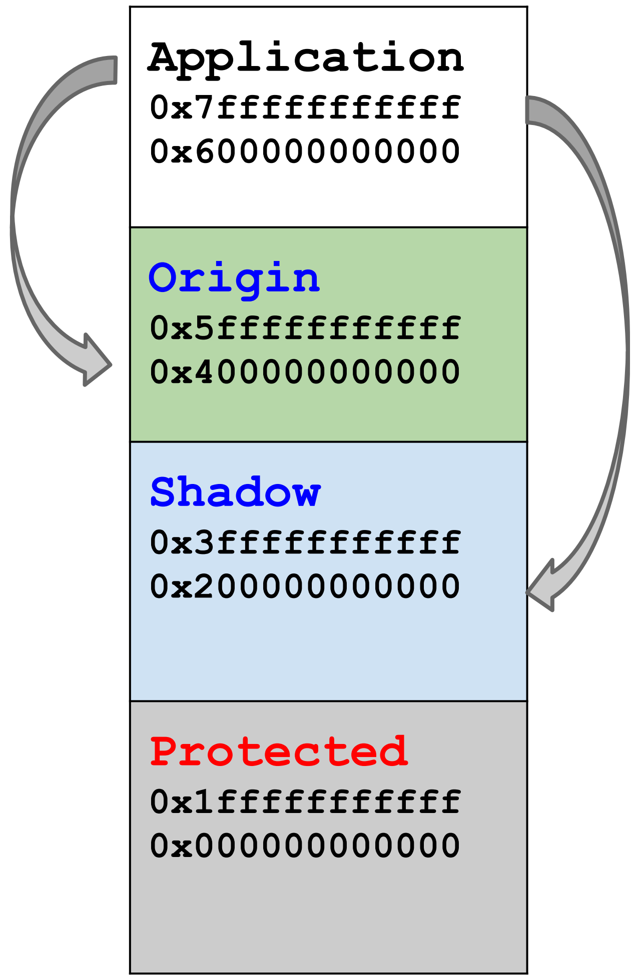
```
a = malloc() ...
b = malloc() ...
c = *a + *b  ...
if (c) ...  // UMR. Is 'a' guilty or 'b'?
```

- Valgrind **`--track-origins`**: propagate the origin of the poisoned memory alongside the shadow

- MemorySanitizer: secondary shadow
  - Origin-ID is 4 bytes, 1:1 mapping
  - 2x additional slowdown

Shadow mapping

```
Shadow = Addr - 0x400000000000;
Origin = Addr - 0x200000000000;
```



MSan overhead

- Without origins:
  - CPU: 3x
  - RAM: 2x
- With origins:
  - CPU: 6x
  - RAM: 3x

Tricky part :(

Missing any write causes false reports.

- Libc
  - Solution: function wrappers

- Inline assembly
  - Openssl, libjpeg_turbo, etc

- JITs (e.g. V8)

MSan trophies

- Proprietary console app, 1.3 MLOC in C++
  - Not tested with Valgrind previously
  - 20+ unique bugs in < 2 hours
  - Valgrind finds the same bugs in 24+ hours
  - MSan gives better reports for stack memory
- 20+ in LLVM
  - Regressions caught by regular LLVM bootstrap
- 300+ bugs in Google server-side code

What’s next?

You can help

Faster

- Use hardware features
  - Or even create them (!)
- Static analysis: eliminate redundant checks
  - Many attempts were made; not trivial!
  - How to test it??

More bugs

- Instrument assembler & binaries
  - SyzyASAN: instruments binaries statically, Win32
- Instrument JIT-ed code & JIT's heap
- More types of bugs
  - Intra-object overflows
  - Annotations in STL, e.g. std::vector<>
- Intel MPX
- Other languages (e.g. races in Java)

More environments

- Microsoft Windows
- Mobile, embedded
- OS Kernel (Linux and others)
- Production
  - Crowdsourcing bug detection?

Q&A

http://code.google.com/p/address-sanitizer/

http://code.google.com/p/thread-sanitizer/

http://code.google.com/p/memory-sanitizer/

Supported platforms

- AddressSanitizer (memory corruption)
  - Linux, OSX, CrOS, Android, iOS
  - i386, x86_64, ARM, PowerPC
  - WIP: Windows, *BSD (?)
  - Clang 3.1+ and GCC 4.8+
- ThreadSanitizer (races)
  - A "must use" if you have threads (C++, Go)
  - Only x86_64 Linux; Clang 3.2+ and GCC 4.8+
- MemorySanitizer (uses of uninitialized data)
  - WIP, usable for "console" apps (C++)
  - Only x86_64 Linux; Clang 3.3

ASan/MSan vs Valgrind (Memcheck)

	Valgrind	ASan	MSan
Heap out-of-bounds	YES	YES	NO
Stack out-of-bounds	NO	YES	NO
Global out-of-bounds	NO	YES	NO
Use-after-free	YES	YES	NO
Use-after-return	NO	Sometimes	NO
Uninitialized reads	YES	NO	YES
CPU Overhead	10x-300x	1.5x-3x	3x

Why not a single tool?

- Slowdowns will add up
    - Bad for interactive or network apps
- Memory overheads will multiply
    - ASan redzone vs TSan/MSan large shadow
- Not trivial to implement