Towards a fully sanitizable C++, with the help from hardware

PLEMM'19

Kostya Serebryany <kcc@google.com>

Agenda

- Memory safety in C++, Sanitizers, “Safe” Languages
- Hardware Memory Tagging vs memory safety
- Other incremental improvements to C++ and hardware

C & C++ memory safety is an industry-wide crisis

- Use-after-free / buffer-overflow / uninitialized memory
- > 50% of High/Critical security bugs in Chrome & Android
- Not only security vulnerabilities
  - crashes, data corruption, developer productivity
- More data: iSecCon'18

Dynamic Tools @ Google

- Sanitizing Google's and everyone's C++ code since 2008
  - Testing: ASan, TSan, MSan, UBSan (also: KASAN for kernel)
  - Fuzzing: libFuzzer, Syzkaller (and more), OSS-Fuzz
  - Hardening in production: LLVM CFI, ShadowCallStack, UBSan
  - Testing in production: GWP-ASan

- Not winning the battle; maybe losing
  - Chrome Release September 10, 2019:
    9 high/critical bugs: 4 use-after-free, 2 buffer overflows
  - Chrome Release September 18, 2019:
    4 high/critical, all are use-after-free

Need more

- Better languages?
- Better testing tools?
- Better hardening?
- Better programmers?
- Better hardware?
- ???

Safe(r) languages?

- Java, Go, C#, …
  - Overhead of garbage collection + races
- Swift
  - Overhead of reference counting + races
- Rust
  - Cognitive overhead

rotten tomatoes are welcome on stage

- All: poor interop with C++, no migration story, have “unsafe”
- We will have to live with C++ for the next decade or two or three

Safety in Rust

Dynamic

- No uses of uninitialized memory:
  - all memory is initialized + optimizations
- No integer overflows
  - Debug: dynamic checking
  - Release: forced wrap around
- No buffer overflows: dynamic checking + optimizations

Static

- No use-after-free: forcing the programmers to write in SSA form
  - OUCH!

Can we get the same in (a dialect of) C++?

- Initialize all memory + optimizations
  - Stack: -ftrivial-auto-var-init=[pattern|zero], thanks JF Bastien
  - Heap: memset in malloc
- Integer overflows
  - -fsanitize=[un]signed-integer-overflow in production
  - Already in your pocket (both iOS and Android)
- Fewer buffer overflows
  - -fsanitize=bounds in production
  - STL container checks in production
  - Reduce (and then ban) pointer arithmetic in new code, cleanup the old code
- Fewer type confusions: ban insecure pointer casts

- Fewer use-after-free: OUCH!

The pesky use-after-free

- C++: no efficient dynamic check *for production*
  - C++ core guidelines: let's sweep them under the rug
- Java, C#, Go, Swift: expensive avoidance
- Rust: torturing the programmer

any more rotten tomatoes?

ASAN is a testing tool

- Finds buffer overflows & use-after-free
- Continuous integration: pre- & post- submit testing
- Continuous automated fuzzing (see Usenix Security'17)
- Responsible for the majority of findings since 2011

ASAN in production is painfully hard

- 2x-3x code size:
  - Explodes the datacenter infrastructure
  - Hard to ship to users
- 2x-3x CPU:
  - Causes cluster management algorithmic problems (lame ducks)
  - More battery use
- **2x-3x RAM:**
  - Different cluster configuration
  - Doesn’t fit on client devices

GWP-ASan: find heap-use-after-free in production

- Allocator *sometimes* uses guard pages
- ~ Zero overhead
- Tiny probability of detecting a bug in one execution
- Huge scale beats tiny probability

Memory Tagging: Arm MTE

Arm Memory Tagging Extension (MTE)

- Announced by Arm on 2018-09-17

- Doesn’t exist in hardware yet
  - Will take several years to appear

- “Hardware-ASAN on steroids”
  - RAM overhead: 3%-5%
  - CPU overhead: (*hoping for)* low-single-digit %

ARM Memory Tagging Extension (MTE)

- 64-bit only
- Two types of tags
  - Every aligned 16 bytes of memory have a 4-bit tag stored separately
  - Every pointer has a 4-bit tag stored in the top byte
- LD/ST instructions check both tags, raise exception on mismatch
- New instructions to manipulate the tags

Allocation: tag the memory & the pointer

- Stack and heap

- Allocation:
  - Align allocations by 16
  - Choose a 4-bit tag (random is ok)
  - Tag the pointer
  - Tag the memory (optionally initialize it at no extra cost)

- Deallocation:
  - Re-tag the memory with a different tag

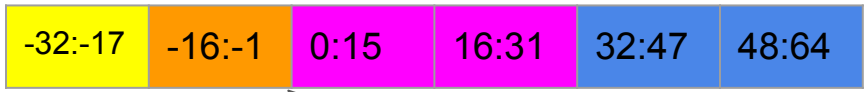
Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```

Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```


```
delete [] p;  // Memory is retagged ⬛ ⇒ ⬛
```




```
p[0] = ...  // heap-use-after-free ⬛ ≠ ⬛
```

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```


```
p[32] = ... // heap-buffer-overflow ⬛ ≠ ⬛
```

Probabilities of bug detection

```
int *p = new char[20];

p[20]                 // undetected, same granule (*)

p[32], p[-1]          // 93%-100% (15/16 or 1)

p[100500]             // 93% (15/16)

delete [] p; p[0] // 93% (15/16)
```

Buffer overflows within a 16-byte granule

- Typically, not security bugs if heap/stack is 16-byte aligned in production

- Still, logical bugs

- Only so-so solutions for testing:
  - Malloc may optionally align right (tricky on ARM, more tricky on x86_64)
  - Put magic value on malloc, check on free (detects only overwrites, with delay)
  - Tag the last granule with a different tag, handle in the signal handler (SLOW)

MTE Overhead

- RAM: 3% - 5% (measured)
- Code Size: 2%-4% (measured)
- CPU: 0% - 5% (*estimated*)
- Power: ?

MTE Usage Models

- Testing in lab
  - Better & cheaper than ASAN
- **Testing in production** aka crowdsourced bug detection
  - possibly with per-process or per-allocation sampling
  - actionable deduplicated bug reports
- Always-on **security mitigation**
  - with per-process knobs

Is probabilistic detection OK for security mitigation?

- Enough retries may allow an MTE bypass in some cases (e.g. UAF)

- BUT:
  - Software could block the restarts on first MTE report (i.e. no retries)
  - The vendors gets actionable bug report on first failed attempt

Legacy code

- MTE will work on legacy code w/o recompilation
  - Libc-only change
  - Will find and mitigate heap OOB & UAF (~90% of all bugs)

No more uses of uninitialized memory

- Tagging the memory during allocation also initializes it
  - MTE always-on => no more uninitialized memory
  - MTE only during testing => uninitialized memory remains

- Can initialize all memory today, at ~ the same cost as full MTE

While we are waiting for Arm MTE

- AArch64: HWAsan - software implementation of memory tagging
  - Just like ASan, but with < 10% RAM overhead
- SPARC ADI: shipped since ~2016, but only SPARC M7/M8 :(
- X86_64, RISC-V: ask your favourite CPU vendor to support it!!

Remaining bug class #1: intra-object buffer overflow

```
struct S {
  int array[5];
  int another_field;
};

int GetInt(int *p, size_t idx) {
  return p[idx];
}
int Foo(S *s) {
  // return s->array[five];
  return GetInt(s->array, 5);
}
```

Solution:

- **Dynamic checks** when the type is known
- **BAN constructs that lose the type information** (yes, language dialect)

Remaining bug class #2: type confusion

```
struct Image {
  int pixels[100];
};

struct Secret {
  int sensitive_data[200];
};

Secret *secret = new Secret;
...
DrawOnScreen((Image*) secret);
// Checked if types are polymorphic
```

Solution:

- **Dynamic checks** for polymorphic types
- **BAN pointer casts for non-polymorphic types** (yes, language dialect)

Next hardware ask (beyond memory tagging)

- **Cheaper run-time checks**:
  - Bounds
  - Integer overflows
  - User asserts

- Possibly asynchronous:
  - Special instructions check the conditions and set a “sticky flag”, w/o blocking other instructions
  - The flag is checked periodically, e.g. during a context switch

None of what I said covers data races (directly)...

- Safe Rust solves races
  - But heavily threaded Rust code is often unsafe
  - Anecdotal evidence; would like to see more research
- Races often cause memory safety bugs (use-after-free)
  - I.e. MTE will indirectly help with many races
- MTE will allow to implement efficient race detection
  - Similar to DataCollider

Kostya’s (*) strategy for C++ 2020-2030

- Memory Tagging everywhere once available
  - Until then: Sanitizers for testing; GWP-ASan & initialize memory in production
- More run-time checking in production
  - fsanitize=bounds,[un]signed-integer-overflow, STL container checks
  - Software Control Flow Integrity: LLVM CFI and ShadowCallStack
  - Hardware Control Flow Integrity: Intel CET, Arm PAC
- Ban unsafe constructs, incrementally cleanup existing code
  - pointer arithmetic
  - pointer casts for non-polymorphic types
- Make fuzzing the first class citizen (CppCon’17)

(*) not necessarily anybody else’s strategy

Q&A