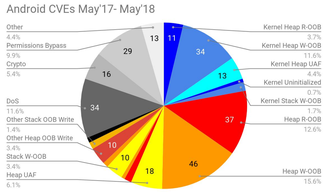
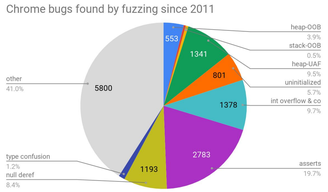
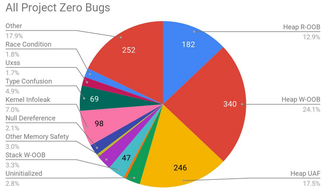
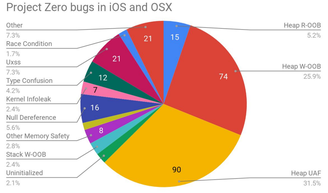
Hardware Memory Tagging makes C++ Memory-Safer

C/C++ Memory Safety is a mess:









> 20% of all bugs and
> 50% of all vulnerabilities are buffer-overflow (heap,stack,globals) or use-after-free

AddressSanitizer (ASAN) is not enough

- Hard to use in production: 2x overhead in CPU, RAM, Code Size
- Weak as a mitigation: attacker can easily bypass the redzones & quarantine

Kostya Serebryany kcc@google.com kayseesee@
Evgenii Stepanov eugenis@google.com
Vlad Tsyrklevich vtsyrklevich@google.com vlad902@

See also: https://arxiv.org/pdf/1802.09517.pdf

Memory Tagging Improves Memory Safety

- Every aligned 16 bytes have a 4-bit tag (extra metadata bits!)
- Every pointer has a tag in the top byte (64-bit only!)
- malloc tags memory & pointers with the same tag, same for stack
- Loads/stores fail on tag mismatch
- Detects use-after-free and buffer-overflow (heap, stack, globals)
  - Probabilistic (> 90%) bug detection
- Other tagging granularities (TG) and tag sizes (TS) are possible

Heap-buffer-overflow (TG=16, TS=4)

```
char *p = new char[20]; // 0xa0007fffffff1240
```

-32-17 -16-1 0 15 16 31 32-47 48-64

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

Heap-use-after-free (TG=16, TS=4)

```
char *p = new char[20]; // 0xa0007fffffff1240
```

-32-17 -16-1 0 15 16 31 32-47 48-64

```
delete [] p;  // ■ ⇒ ■
```

-32-17 -16-1 0 15 16 31 32-47 48-64

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

Better than ASAN:

- Smaller RAM overhead (3% vs 2x)
- Detects buffer overflows far from bounds
- Detects use-after-free long after deallocation

Usage Models:

- Regular testing/fuzzing: just like ASAN, but with less RAM
- Testing in production
  - Vendor gets actionable and bucketizable reports
  - Easy to turn on/off
- Always-on security mitigation
  - Unstable exploits typically discourage attackers
  - First failed attack alerts the user and the vendor

Homework:

Analyze you favourite exploit: is it preventable by MT?
Ask your CPU vendor to implement MT

Existing Implementations

SPARC ADI: TG=64;TS=4
Available in *hardware* since ~2016, SPARC M7/M8
Heap-only; 5%-20% RAM overhead due to alignment

HWASAN (Hardware ASAN): TG=16; TS=8
Software (compiler) instrumentation, similar to ASAN
Uses AArch64-only feature “top-byte-ignore”
Limited applicability on x86_64 (needs to see all loads/stores)

```
// int foo(int *a) { return *a; }
// clang -O2 --target=aarch64-linux -fsanitize=hwaddress -c load.c
     0:       08 dc 44 d3     ubfx    x8, x0, #4, #52  // shadow address
     4:       08 01 40 39     ldrb    w8, [x8]         // load shadow
     8:       09 fc 78 d3     lsr     x9, x0, #56      // address tag
     c:       3f 01 08 6b     cmp     w9, w8           // compare tags
    10:       61 00 00 54     b.ne    #12              // jump on mismatch
    14:       00 00 40 b9     ldr     w0, [x0]         // original load
    18:       c0 03 5f d6     ret
    1c:       40 20 21 d4     brk     #0x902           // trap
```

ARM v8.5: Memory Tagging Extension NEW!

- TG=16;TS=4 (good!)
- Announced on September 17 2018
- Does not exist in hardware yet :(

Hardware/CPU challenges:

- Tags and data need to be in the same caches (?)
- Where to store the tags? ECC bits? Separate RAM region?

Software/Compiler challenges

- Efficient instrumentation to tag objects on stack
- Statically prove safety for some stack variables
- Malloc now zero-fills - use this to remove stores
- Malloc is now more expensive: eliminate it harder