Memory tagging in LLVM and Android

Evgenii Stepanov, Kostya Serebryany, Peter Collingbourne, Mitch Phillips, Vitaly Buka
Google
LLVM Developer Meeting, Oct. 2020

Agenda

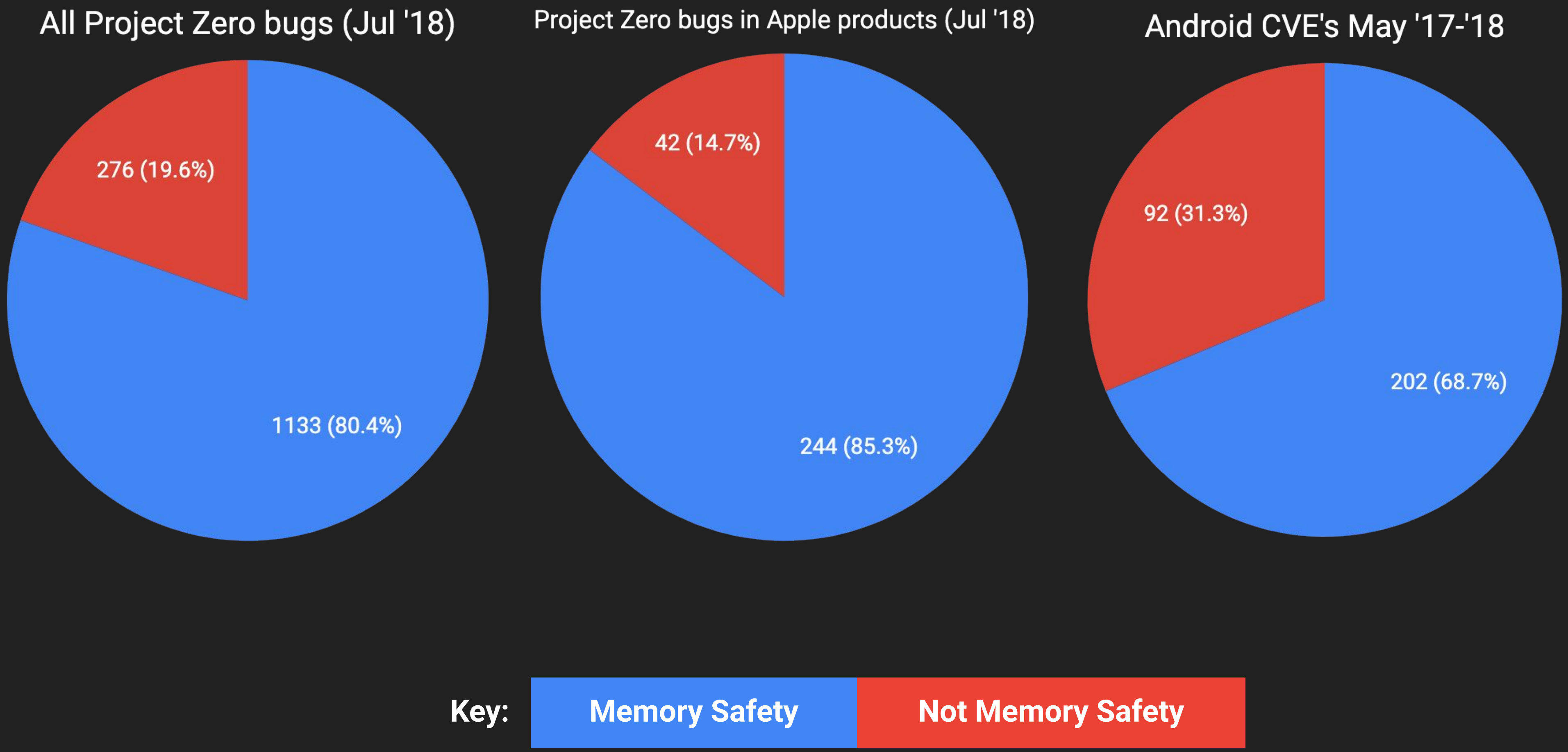
- C(++) memory safety primer
- ARMv9 Memory Tagging Extension
- Implementation Details & Future Work
  - Heap Tagging
  - Stack Tagging
    - Stack Safety Analysis optimizations
  - Globals Tagging
- Expected rollout in Android

C++ Memory Safety

More than 50% of High severity bugs in Android are memory corruption.

Not only security: debugging memory corruption bugs is hard.
AddressSanitizer (ASan and HWASan) helps, but:

- Requires recompilation.
- Slow.
- Can be bypassed (not a security mitigation).


All Project Zero bugs (Jul '18)
276 (19.6%)
1133 (80.4%)
Project Zero bugs in Apple products (Jul '18)
42 (14.7%)
244 (85.3%)
Android CVE's May '17-'18
92 (31.3%)
202 (68.7%)
Key:
Memory Safety
Not Memory Safety

What is the Memory Tagging Extension?

- Optional extension in ARMv9, announced Aug 2018.
- AArch64 only, introduces 2 types of tags:
  - Logical Address Tag - bits 56..59 of the virtual address.
  - Allocation Tag - 4 bits for every 16 bytes of memory, stored separately.
- Load / Store instructions raise an exception if tags differ.
- New instructions to manipulate tags.
- Two modes:
  - Synchronous - process dies immediately with SEGV_MTESERR.
    - hoping for < 20% slowdown (*)
  - Asynchronous - process dies with SEGV_MTEAERR at the nearest context switch.
    - hoping for < 5% slowdown (*)
    - Does not provide fault PC or data address.

** All performance numbers are estimates.*

How to use it?

- Protect heap
  - Randomly tag pointer + memory on allocation
  - Randomly tag memory on deallocation
  - Catches use-after-free, heap-buffer-overflow, double-free with 93% probability
- Protect stack
  - Randomly tag local variables when entering function or scope.
  - Tag local variables to tag(SP) when leaving function or scope.
  - Catches use-after-return, use-after-scope, stack-buffer-overflow with 93% probability
- Protect globals
  - Randomly tag global variables at load time
  - Apply tags to GOT pointers
  - Apply pointer tag when taking address of a local, non-GOT symbol
  - Catches global-buffer-overflow with 93% probability.

Heap tagging example

char *p = new char[20]; // 0xa0000xxxxxxxxxx

-32:-17	-16:-1	0:15	16:31	32:47	47:63

p[32] = …; // CRASH

delete[] p; // 0xa0000xxxxxxxxxx

-32:-17	-16:-1	0:15	16:31	32:47	47:63

p[0] = …; // CRASH

Heap tagging

Implemented in Scudo (default system allocator in Android 11).

Bump minimum alignment to 16.

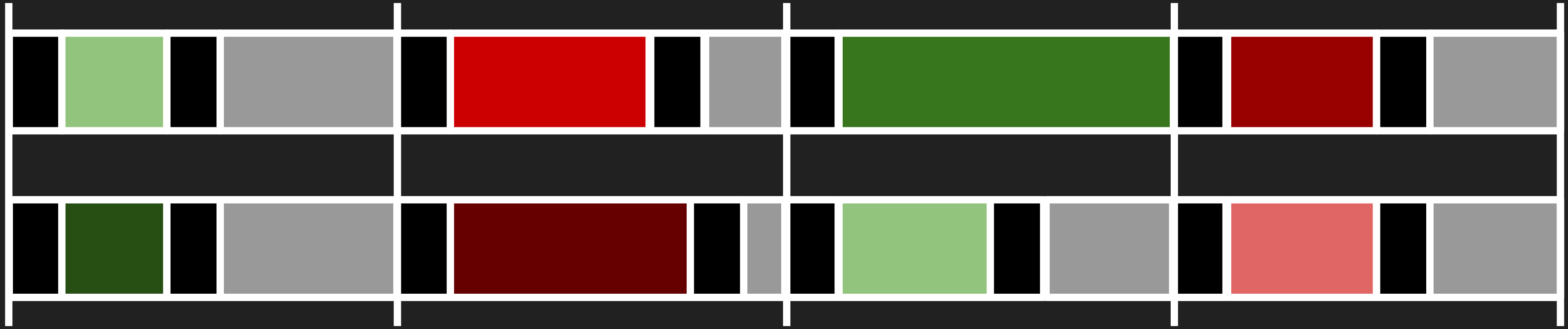
Malloc from mmap: choose a random tag, apply to the pointer and memory.

Free: choose a random tag, apply to memory.

Malloc (memory reuse): load tag from memory, apply to the pointer.

Special cases: memory released to OS loses tag data; size change (within one size class) requires memory tag fixup.

Heap Tagging: implementation details



Zero-tagged chunk header and optional right redzone.

Never reuse the same tag on free.

Spatial vs temporal protection trade-off: odd-even tags in adjacent chunks.

- (+) 100% detection of overflows of up to the entire allocation size
- (-) 87% detection of use-after-free (down from 93%).

Heap tagging: large allocations

Large allocations that are not used immediately, or used sparsely, are expensive to tag up front. Two options:

- Do not tag. Surround with guard pages and never reuse VA (infinite quarantine).
- Use a copy-on-write reference page with a non-zero tag (https://lwn.net/Articles/828828)

Heap tagging: crash reporting

Synchronous mode faults provide PC, data address and register contents. This can be used to implement a lightweight AddressSanitizer-like tool.

A fixed-size ring buffer to store recent alloc/dealloc stack traces. FP-based unwinding.

`__scudo_malloc_set_track_allocation_stacks()`

`__scudo_get_error_info()`

- Provides up to 3 "culprit" alloc/dealloc pairs with the matching address & tag.

Stack tagging

`void f() {`
`  int x = 42;`
`  use(&x);`
`}`
	`str  x30, [sp, #-16]!`
	
	`mov  w8, #42`
	`add  x0, sp, #12`
	`str  w8, [sp, #12]`
	`bl   use`
	
	`ldr  x30, [sp], #16`
	`ret`
		`sub   sp, sp, #32`
		`str   x30, [sp, #16]`
		`irg   x0, sp`
		`mov   w8, #42`
		`stgp  x8, xzr, [x0]`
		
		`bl    use`
		`stg   sp, [sp], #16`
		`ldr   x30, [sp], #16`
		`ret`

```
clang -fsanitize=memtag -march=armv8+memtag
```

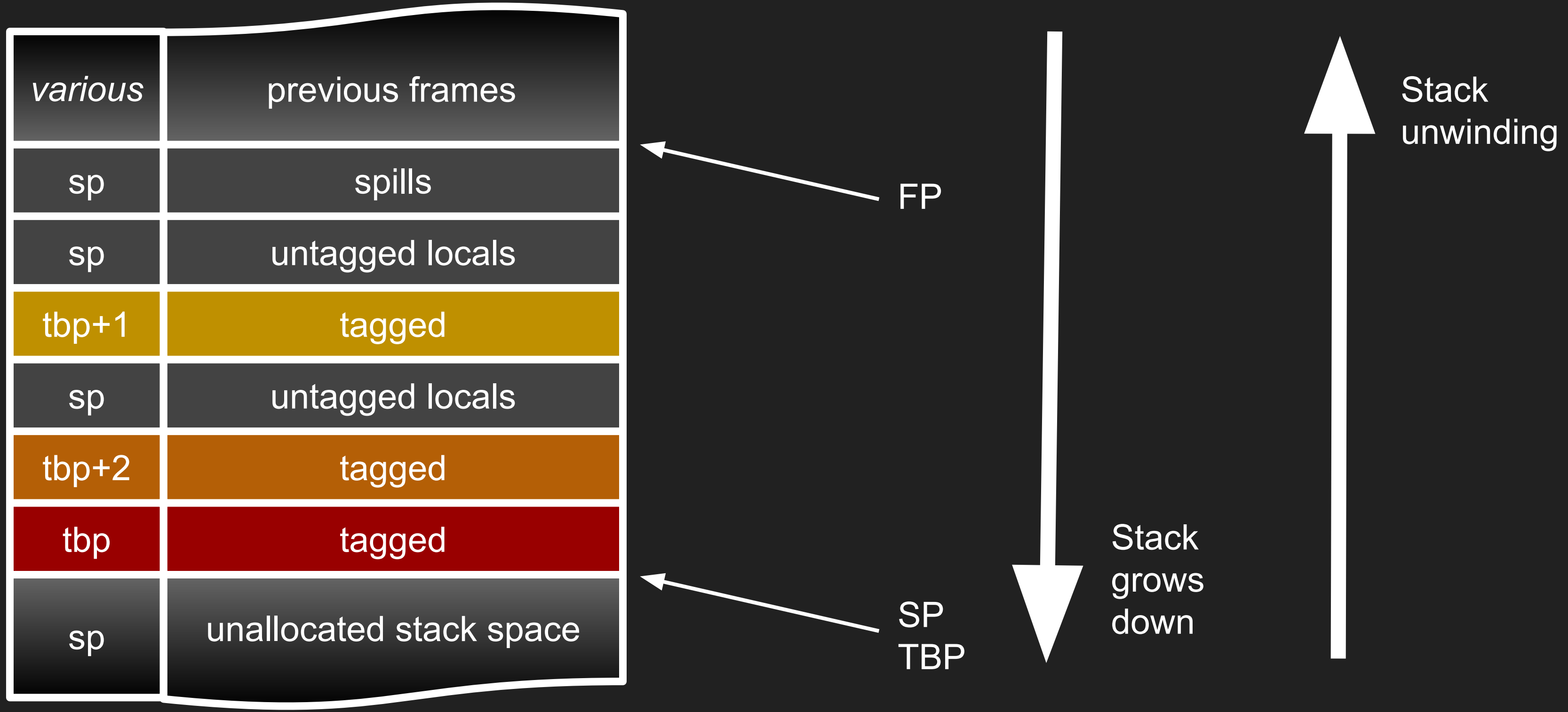
Stack tagging: base pointer

Assigning an independently random tag to each variable requires an extra live register per variable. This does not scale.

A *tagged base pointer* allows addressing variables with (addr offset, tag offset).

`void f() {`
`  int a, b, c;`
`  use(&a);`
`  use(&b);`
`  use(&c);`
`}`
	
	`add   x0, sp, #12`
	`bl    use`
	`add   x0, sp, #8`
	`bl    use`
	`add   x0, sp, #4`
	`bl    use`
		`irg   x19, sp`
		
		`addg  x0, x19, #32, #2`
		`bl    use`
		`addg  x0, x19, #16, #1`
		`bl    use`
		`mov   x0, x19`
		`bl    use`

Tagged stack layout



Stack tagging: optimizations

- Load/Store of [SP+#imm] are unchecked by hardware => no need to materialize a tagged address.
- ST2G sets memory tags 32 bytes at a time => group allocas that leave scope simultaneously, rewrite STG + STG to ST2G.
- Set tag and data simultaneously:

<pre>struct A {
  long a, b, c, d;
};
long f() {
  A a{0, 0, 42,(long)&a};
  use(&a);
  return a.b;
}</pre>
	<pre>irg x0, sp
	mov w8, #42
	stzg x0, [x0]
	stgp x8, x0, [x0, #16]
	bl use
	ldr x0, [sp, #8]
	st2g sp, [sp], #32</pre>

Stack Safety Analysis

Many stack allocations, even address-taken, are trivially safe and do not need protection.

StackSafetyAnalysis finds (min, max) range of offsets that provably covers all memory access of an alloca.

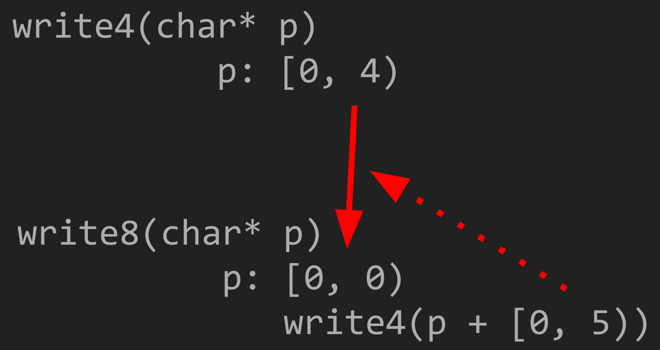
- Conservative: returns full-set if alloca escapes or may be used outside its lifetime.
- Interprocedural, with Thin LTO support.
- Context-insensitive.

Stack Safety: IPO

```
void write4(char *p) {
  memset(p, 0, 4);
}

void write8(char *p) {
  write4(p);
  write4(p + 4);
}

char func() {
  char x[8];
  write8(x);
  return x[2];
}
```

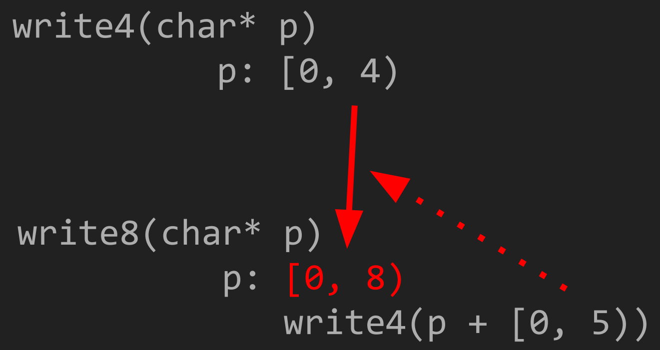


Stack Safety: IPO

```
void write4(char *p) {
  memset(p, 0, 4);
}

void write8(char *p) {
  write4(p);
  write4(p + 4);
}

char func() {
  char x[8];
  write8(x);
  return x[2];
}
```

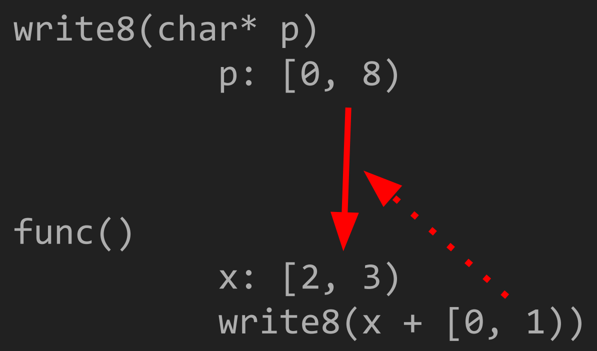


Stack Safety: local analysis

```
void write4(char *p) {
  memset(p, 0, 4);
}

void write8(char *p) {
  write4(p);
  write4(p + 4);
}

char func() {
  char x[8];
  write8(x);
  return x[2];
}
```

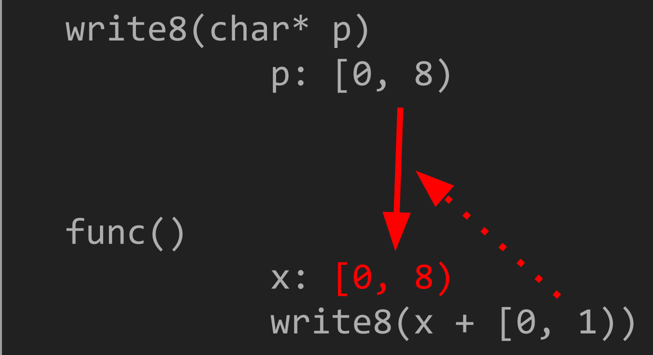


Stack Safety: local analysis

```c
void write4(char *p) {
  memset(p, 0, 4);
}

void write8(char *p) {
  write4(p);
  write4(p + 4);
}

char func() {
  char x[8];
  write8(x);
  return x[2];
}
```



Stack Safety

Runs until fixed point.

Unbounded recursion? Relax offset ranges to full-set after a number of steps.

Using Chromium as a benchmark:

- 25% allocas proven safe in separate compilation
- 60% allocas proven safe with LTO

Globals Tagging

- Dynamic symbols (`int f; extern int f;`)
  - Mark dynamic symbol table with st_other.STO_TAGGED
  - Teach the loader to read entire symbol table at startup and assign memory tags.
- Local symbols (`static int g;` or `-fvisibility=hidden`)
  - Create a segment containing { &global, sizeof(global } pairs for each global. Place this table's address in the .dynamic section under a new tag DT_MTEGLOBTAB.
  - Teach the loader to read this table and assign a random memory tag to each global.
  - Address-taken sequences (`&g`) insert the tag via `ldg`.
- All globals:
  - Realign to granule size (16 bytes), resize to multiple of granule size (e.g. 40B -> 48B).
  - Ensure non-executable segments are mapped `MAP_ANONYMOUS` and `PROT_MTE` (file-based mappings aren't necessarily backed by tag-capable memory)
  - Ban data folding (except where contents **and** size are same, no tail merging)

Globals Tagging (Relocations)

- GLOB_DAT, ABS64 need to insert memory tag into relocated value (via `ldg`).
  - dlsym() needs to do the same thing.
- RELATIVE relocations need to append memory tag, but...

```
static int array[] = { 1, 2, 3, 4 };
// array_end must have the same tag as array[]. array_end is out of
// bounds w.r.t. array, and may point to a completely different global.
int *array_end = &array[4];
```

- Introduce RELATIVE_TAGGED
  - Place (*r_offset) stores where the tag should be derived from
  - Addend (r_addend) contains the untagged value to be relocated.
  - XOR the addend and the tag to get the tagged value, and store that in the place.
  - Zero addend means tag is derived from the place, and can be RELR-style compressed.

Android

Experimental implementation available in AOSP(*) now.

- Async heap tagging in the system apps on by default.
- User apps can opt-in via manifest.
- An API to enable Sync mode and allocator debugging features.
- Stack + globals tagging requires incompatible code instrumentation.
  - Can be shipped in non-updatable platform binaries only.
  - Can be used for local debugging.
  - In the distant future, a new application ABI will include hardware MTE support.

(*) `https://cs.android.com/android/platform/superproject/+/master:device/generic/goldfish/fvpbase/README.md`

Thank you for listening!

Questions?