More advanced defences against memory corruption

[See SoK Eternal War in Memory paper]

Last week

Big security worries in C/C++ code

- Memory corruption
- Integer overflow
- Format String attacks

Standard, basic defences against memory corruption

- stack canaries – to detect some problems
- ASLR – randomness/noise to make exploiting harder
- Non-Executable memory W⊕X – to prevent some exploit

Some cheap / insecurely built devices still do not use these basic defences mechanisms!

This week: more advanced defences

Types of (building blocks for) attacks

- Code corruption attack

  Overwrite the original program code in memory; impossible with W⊕X

- Control-flow hijack attack

  Overwrite a code pointer, eg return address, jump address, function pointer, or pointer in `vtable` of C++ object

- Data-only attack

  Overwrite some data, eg `bool isAdmin;`

- Information leak

  Only reading some data; recall Heartbleed attack on TLS

Control flow hijack via code pointers

- A compiler translates function calls in source code to `call` `<address>` or `JSR` `<address>` in machine code where `<address>` is the location of the code for the function.

- For a function call `f(...)` in C a static address (or offset) of the code for `f` may be known at compile time.

  If compiler can hard-code this static address in the binary, W⊕X can prevent attackers from corrupting this address

- For a virtual function call `o.m(...)` in C++ the address of the code for `m` usually has to be determined at runtime, by inspecting the virtual function table (`vtable`)

  W⊕X does not prevent attackers from corrupting code pointers in these tables

Classification of defences [SoK paper]

- Probabilistic methods

  Basic idea: add randomness to make attacks harder

  - randomness in location where certain data is located (eg ASLR)
    or in the way data is represented in memory (eg pointer encryption)

- Memory Safety

  Basic idea: do additional bookkeeping & add runtime checks to prevent some illegal memory access

- Control-Flow Hijack Defenses

  Basic idea: do additional bookkeeping & add runtime checks to prevent strange control flow

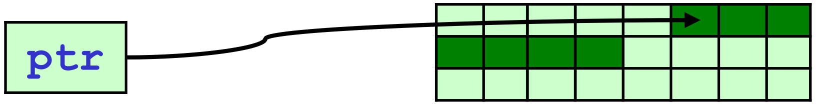
More randomness: Pointer Encryption (PointGuard)

- Many buffer overflow attacks involve corrupting pointers: pointers to data or code pointers
- To make this harder: store pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: eg. XOR with a fixed value that is randomly chosen when a process starts
- Attacker can still corrupt encrypted pointers in memory, but these will not decrypt to predictable values
  - Beware: this uses *encryption* to ensure *integrity*. Normally NOT a good idea, but here it works.

- More extreme variant: Data Space Randomisation (DSR)
  - store not just pointers encrypted in main memory, but store all data encrypted in memory

More memory safety

Additional book-keeping of meta-data
& extra runtime checks to prevent illegal memory access



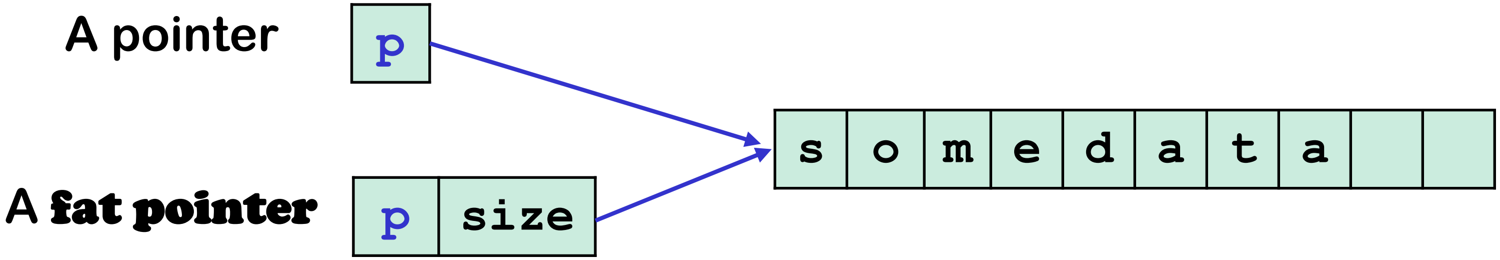
Different possibilities

- **add information to pointer about size of memory chunks it points to (fat pointers)**
- **add information to memory chunks about their size (Spatial safety with object bounds)**
- ...

Fat pointers

The compiler

- records size information for all pointers
- adds runtime checks for pointer arithmetic & array indexing



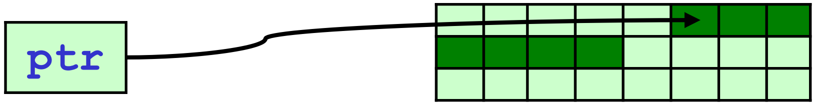
Downsides?

- Big execution time overhead
- Small size overhead
- Not binary compatible – ie all code needs to be compiled to add this book-keeping for all pointers

More memory safety

Additional book keeping of meta-data
& extra runtime checks to prevent illegal memory access

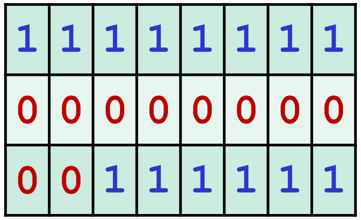
Different possibilities



- add information to pointer about size of memory chunks it points to (fat pointers)
- add information to memory chunks about their size (Spatial safety with object bounds)
- keep a shadow administration of this meta-data, separate from the pointers & the existing memory (SoftBounds)
- keep a shadow administration of which memory cells have been allocated (Valgrind, Memcheck, AddressSanitizer or ASan)
  - to also spot temporal bugs, ie. malloc/free bugs

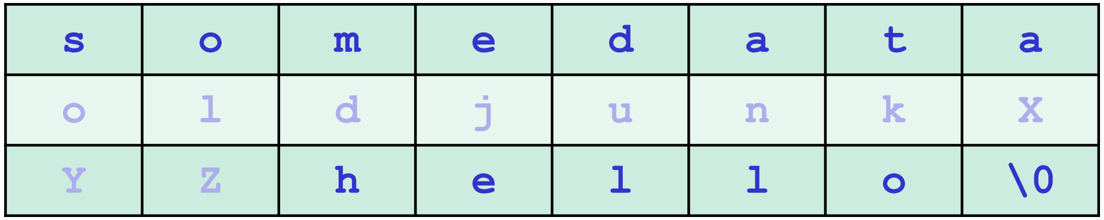
Object-based temporal safety (Valgrind, Memcheck, ASan)

Shadow admin



1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1

of allocated memory



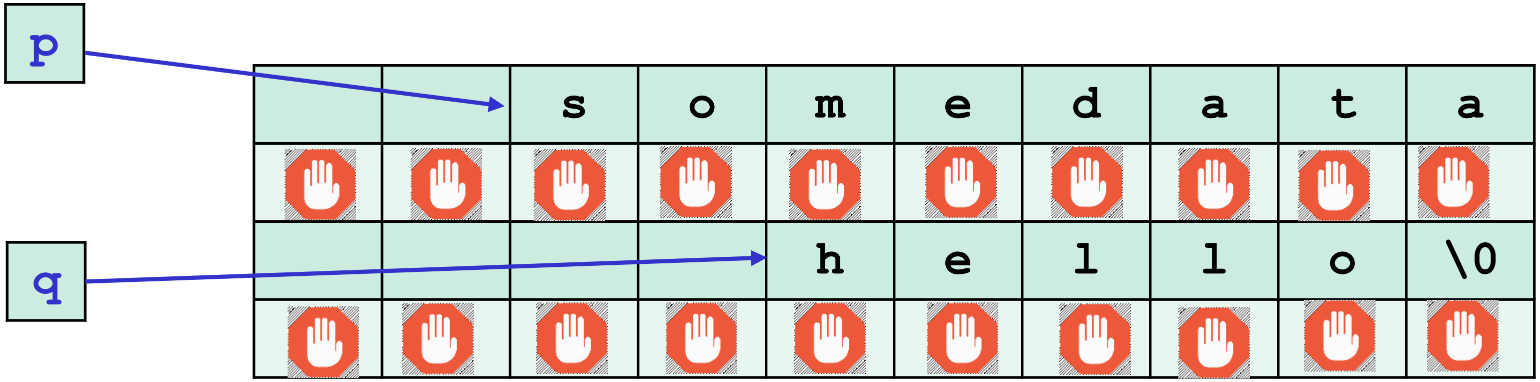
s	o	m	e	d	a	t	a
o	l	d	j	u	n	k	X
Y	Z	h	e	l	l	o	\0

to keep track of which memory is allocated, to generate runtime error when code tries to read/write unallocated memory

- Can also catch spatial bugs by always keeping empty space between allocated chunks
  - Small buffer overrun will end up in this unallocated space, but a big buffer overrun may end up in the next allocated chunk
- Cannot spot illegal access via a stale pointer if the data chunk it points to has been re-allocated
  - Eg the last bug, line 3004, on slide 15

Guard pages to improve memory safety

Allocate chunks with the end at a page boundary with a non-readable, non-writeable page 🛑 between them



Buffer overwrite or overread will cause a memory fault.

Again, a really big overrun may not be caught as it falls in the next page

Small execution overhead, but **big** memory overhead

Control Flow Integrity (CFI)

Extra bookkeeping & checks to spot unexpected control flow

- Dynamic return integrity

  Stack canaries are a way to provide dynamic return integrity, ie. provide check against corruption of return addresses.

  A shadow stack is an alternative mechanism for this.

- Static control flow integrity

  Idea: determine the control flow graph (cfg) at compile-time and monitor jumps in the control flow to spot deviant behavior

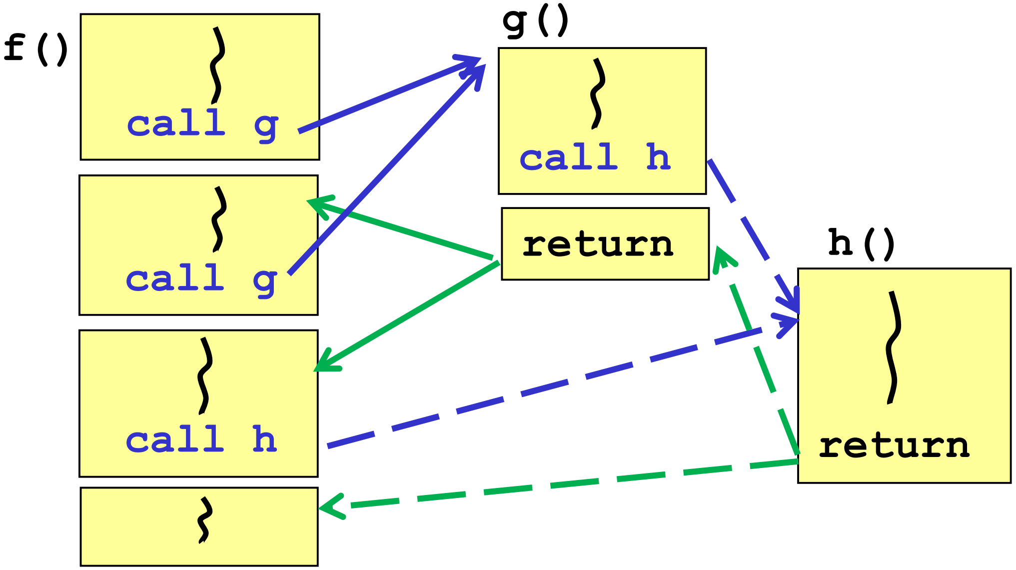
  If `f()` never calls `g()`,
  because `g()` does not even occur in the code of `f()`,
  then call from `f()` to `g()` is suspicious,
  as is a return from `g()` to `f()`

  We could interrupt execution when this happens.

  This can detect Return-to-libc and ROP attacks!

Static control flow integrity: example code & CFG

```
void f()  {
     ... ; g();
     ... ; g();
     ... ; h();
     ...
}
void g(){ ..h();}
void h(){ ... }
```



Before and/or after every control transfer (function call or return) we could check if it is legal – ie. allowed by the cfg

Some weird returns would still be allowed

- eg if we call `h()` from `g()`, and the return is to `f()`, this would be allowed by the static cfg
- Additional *dynamic* return integrity check can narrow this down to actual call site – using recorded call site on shadow stack

Downsides of static control flow integrity checks

- Requires a whole program analysis

- Use of function pointers in C or virtual functions in C++ (that both result in so-called indirect control transfers) complicate compile-time analysis of the cfg

  For example, in C++, `Animal.eat()` can resolve to `Cat.eat()` or `Dog.eat()`, so both these addresses are valid targets for transferring control

  Solutions:

  - a points-to analysis to determine where such code pointers can point to
  - simply allow transfer of control to any function entry point for virtual calls that can not be resolved at compile time

Are people actually using these fancier mechanisms?

- Pointer encryption in iOS (2018)
- Hardware-enforced Stack Protection in Windows 10 (2020)

  with a shadow stack, using Intel Control-flow Enforcement Technology (CET)

  https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815
- Evolution of CFI at Microsoft discussed by Joe Bialek

  https://www.youtube.com/watch?v=oOqpl-2rMTw

  The Evolution of CFI Attacks and Defenses @ OffensiveCON 18
- In testing phase, many of the instrumentation-based approaches can be really useful, even in the overhead is unacceptable in real use. More on that next week

Exam questions: you should be able to

- Explain how simple buffer overflows work & what root causes are
- Spot a *simple* buffer overflow, memory-allocation problem, format string attack, or integer overflow in some C code
- Explain how countermeasures - such as stack canaries, non-executable memory, ASLR, CFI, bounds checkers, pointer encryption, guards pages, etc … - work
- Explain why they might not always work