“Platform”-level defences

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

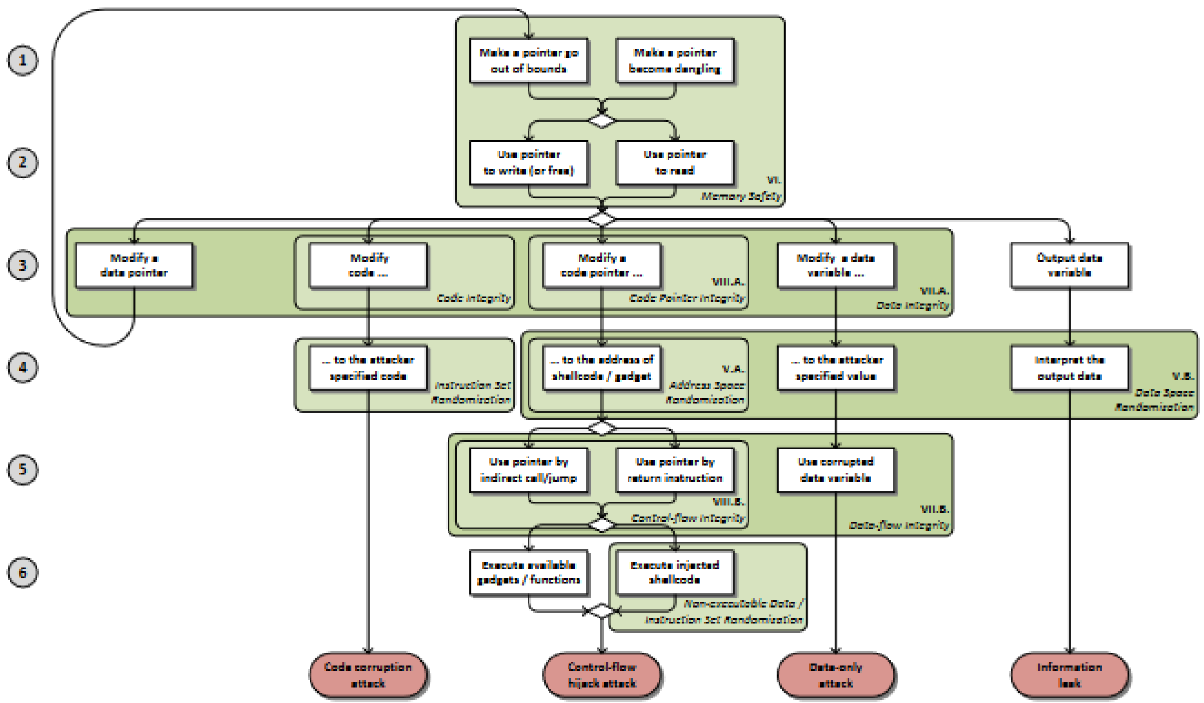


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Platform-level defences

- Defenses the "platform" – ie compiler, hardware, OS, … – can take, without the programmer having to know
- Some defenses need OS & hardware support
- Some defenses cause overhead
  - If this overhead is unacceptable in production code, we can still use it in testing phase
  - Attitudes about how much overhead is acceptable have been changing over time
- Some defenses may break binary compatibility
  - if the compiler adds extra book-keeping & checks, all libraries may need to be re-compiled with that compiler

Platform-level defenses

1. Stack canaries
2. Non-executable memory (NX, W⊕X)
3. Address space layout randomization (ASLR)
4. Various forms of integrity checks on control flow

} now standard on many platforms

More advanced defenses

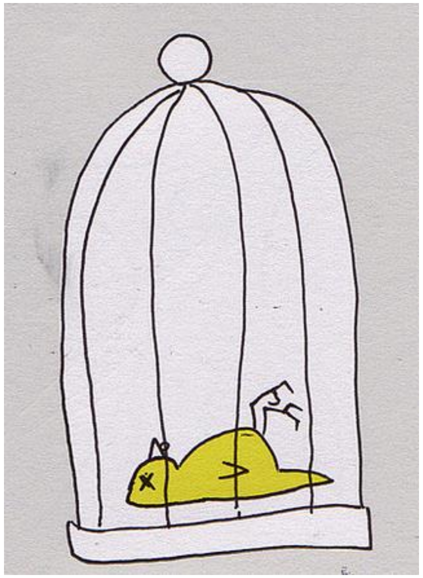
1. More randomisation: eg. pointer & memory encryption
2. More memory safety checks:

   eg. checks on bounds (spatial) or on allocation (temporal)
3. Execution-aware memory protection

1. Stack canaries

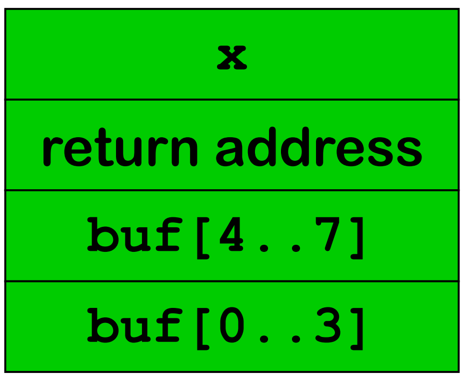
- **Stack canary** aka **stack cookie** is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected

  - first introduced in as StackGuard in gcc
  - only very small runtime overhead

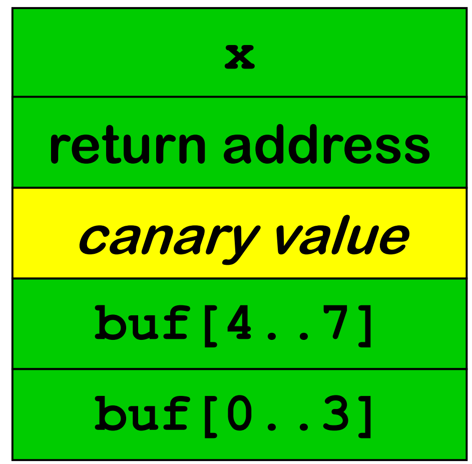


Stack canaries

Stack without canary



Stack with canary



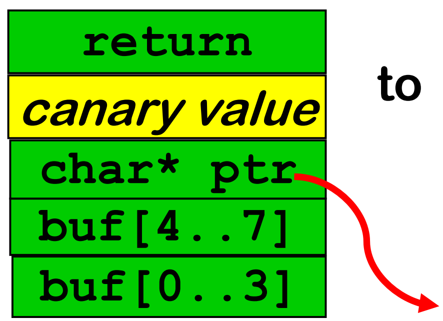
Further improvements

- **More variation in canary values**: eg not a fixed values hardcoded in binary but a random values chosen for each execution
- Better still, **XOR the return address into the canary value**
- **Include a null byte in the canary value**, because C string functions cannot write nulls inside strings

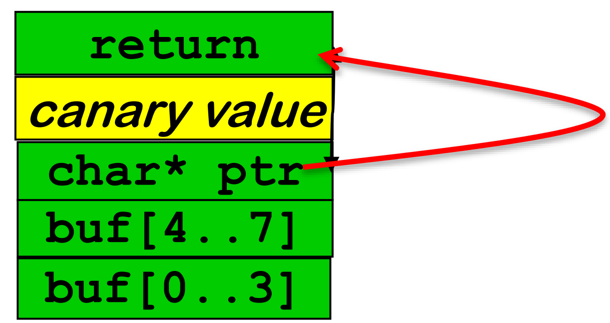
A careful attacker can still defeat canaries, by

- overwriting the canary with the correct value
- corrupting a pointer to point to the return address
  to then change the return address without killing the canary

eg changing



to



Aside: corrupting pointers

Overwriting pointers is especially interesting because subsequent uses of that pointer then read/write data in another place which attacker can choose.

```
100   char* ptr;
101   char[8] buf;
      ...
200   fgets(buf, 12, stdin);  // overflow corrupts ptr,
       // e.g. to point to the position of return address
     ...
210  fgets(ptr, 100, stdin);
           // corrupts any location chosen by the
           // attacker when overflowing buf in line 200
```

Further improvements

- Re-order elements on the stack to reduce the potential impact of overruns
  - swapping parameters `buf` and `fp` on stack changes whether overrunning `buf` can corrupt `fp`
    - which is especially dangerous if `fp` is a function pointer
  - hence it is safer to allocated array buffers 'above' all other local variables

- A separate shadow stack
  - with copies of return addresses, used to check for corrupted return addresses
  - Of course, the attacker should not be able to corrupt the shadow stack

Windows 2003 Stack Protection

Nice example of the ways in which things can go wrong...

- **/GS command line option in Visual Studio add stack canaries**
- **When canary is corrupted, control is transferred to an exception handler**
- **Exception handler information is stored ...**

  on the stack!
- **Attacker can corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection transfer control to a malicious exception handler**

  [http://www.securityfocus.com/bid/8522/info]

- **Countermeasure: only allow transfer of control to registered exception handlers**

2. ASLR (Address Space Layout Randomisation)

- Attacker needs detailed info about memory layout
  - eg to jump to specific piece of code
  - or to corrupt a pointer at known position on the stack
- Attacks become harder if we randomise the memory layout every time we start a program
  - ie. change the offset of the heap, stack, etc, in memory by some random value

- Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim's machine to carry out an attack

- NB security by obscurity, despite its bad reputation, is a really great defense mechanism to annoy attackers!
- Once the offset leaks, we're back to square one…

3. Non-eXecutable memory (NX, aka W⊕X, W^X, DEP)

Distinguish

- X: executable memory (for storing code)
- W: writeable, non-executable memory (for storing data)

and let processor refuse to execute non-executable code

Attackers can then no longer jump to their own attack code, as any input provide as attack code will be non-executable

aka DEP (Data Execution Prevention).

Intel calls it eXecute-Disable (XD)

AMD calls it Enhanced Virus Protection

Limitation:

this technique does not work for JIT (Just In Time) compilation, where e.g. JavaScript is compiled to machine code at run time.

Defeating NX: return-to-libc attacks

With NX, code *injection* attacks no longer possible,
but code *reuse* attacks still are...

- **Attackers can no longer corrupt code or insert their own code, but can still corrupt code pointers**
- **Called control-flow hijack in SoK paper**

So instead of jumping to own attack code
corrupt return address to jump to existing code
esp. library code in `libc`

`libc` is a rich library that offers lots of functionality,
eg. `system()`, `exec()`,
which provides attackers with all they need...

reTURN oriented programMing (ROP)

Next stage in evolution of attacks, as people removed or protected dangerous `libc` calls such as `system()`

Instead of using a library call, attackers can

- look for gadgets, small snippets of code which end with a return, in the existing code base

```
...; ins1 ; ins2 ; ins3 ; ret
```

- chain these gadgets together as subroutines to form a program that does what they want

This turns out to be doable

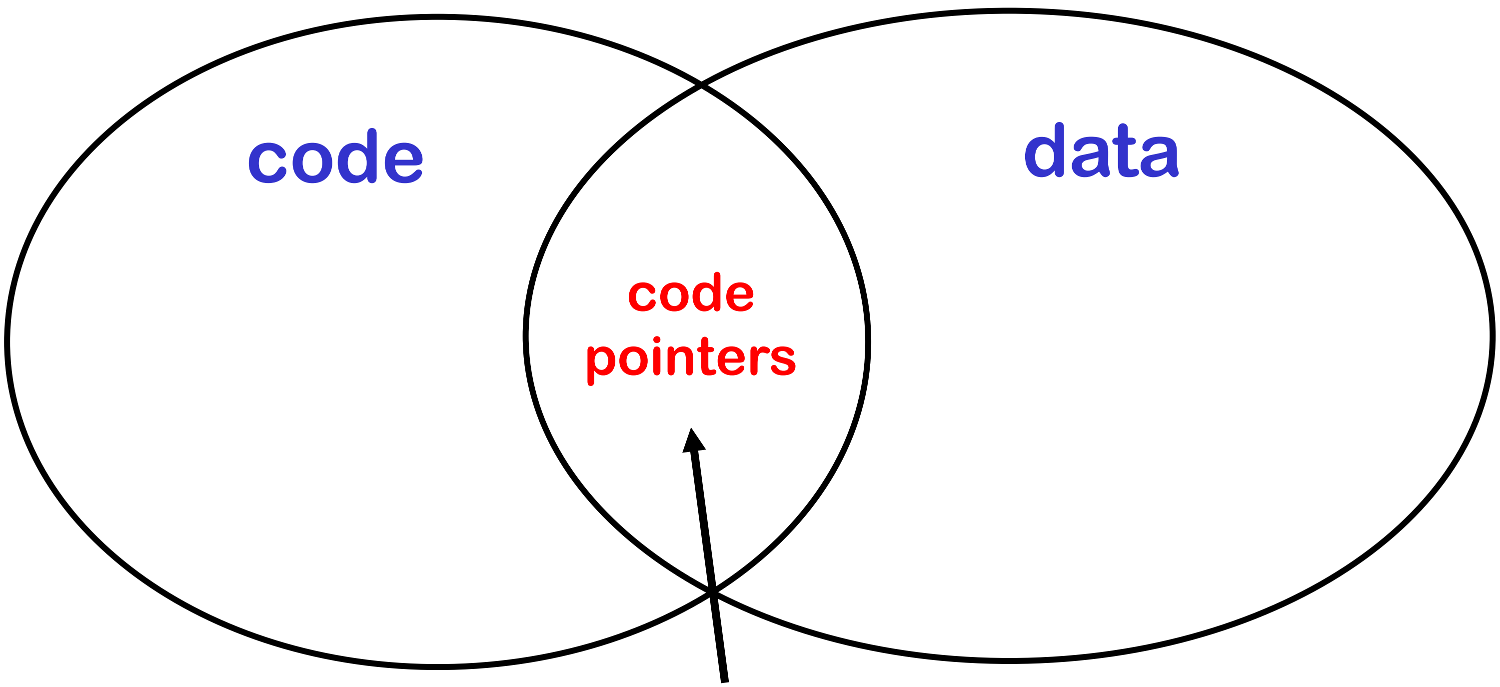
- Most libraries contain enough gadgets to provide a Turing complete programming language
- ROP compilers can then translate arbitrary code to a sequence of these gadgets

More advanced defences

[See SoK Eternal War in Memory paper]

Memory corruption attacks

- Attacks can target code or data
- Attacks can compromise integrity or confidentiality



The biggest disasters happen here

Types of (building blocks for) attacks

1. **Code corruption attack**

   Overwrite the original program code in memory

   Impossible with W⊕X

2. **Control-flow hijack attack**

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

3. **Data-only attack**

   Overwrite some data, eg `bool isAdmin;`

4. **Information leak**

   Only reading some data; e.g. 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` typically 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 Eternal War in Memory]

- Probabilistic methods

  Basic idea: add randomness to make attacks harder

  – 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 check to prevent strange control flow

More randomness: Pointer Encryption (PointGuard)

- Many buffer overflow attacks involve corrupting pointers,
  pointers to data or code pointers
- To complicate this: store pointers encrypted in main memory, unencrypted in registers
  - simple & fast encryption scheme: eg. XOR with a fixed value, randomly chosen when a process starts
- Attacker can still corrupt encrypted pointers in memory, but these will not decrypt to predictable values
  - 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
  - Some AMD chips support this under name SME (Secure Memory Encryption) that uses AES

Recent trends on pointer encryption/authentication

- **Pointer Authentication on Qualcomm ARMv8.3**

  if not all 64 bits are needed for pointers, remaining bits can be used for a PAC (Pointer Authentication Code)

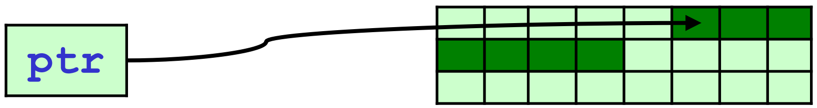
  - 3 – 24 bits PACs using fast QARMA cipher

- Joan Daemen's PhD student Yanis Belkheyar in our group works on lightweight ciphers suitable for pointer encryption for Intel's Cryptographic Capability Computing (C^3)

  - Lightweight can be lightweight in 1) power consumption, 2) surface area of hardware implementation, or 3) time. For pointer encryption/authentication, time (aka latency) is crucial.

More memory safety

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



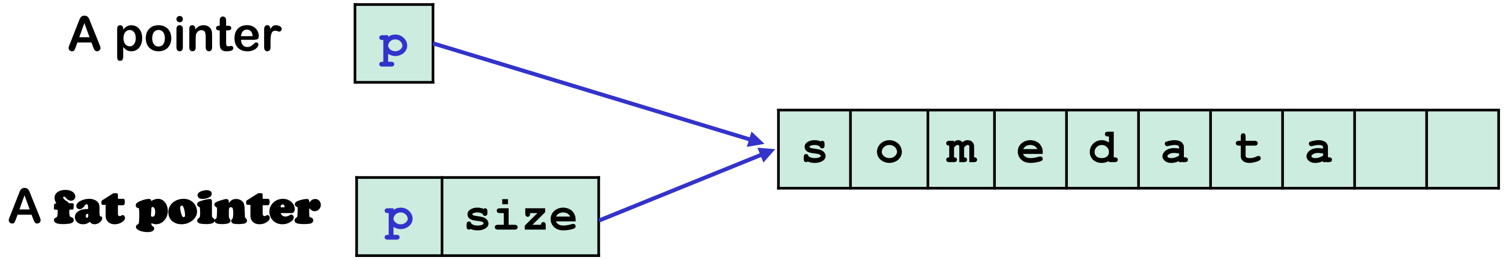
Different possibilities

- **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

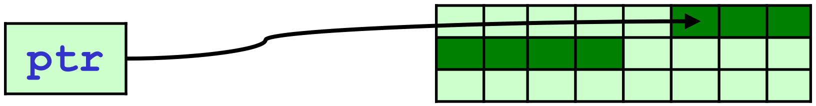


Downsides

- Considerable execution time 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

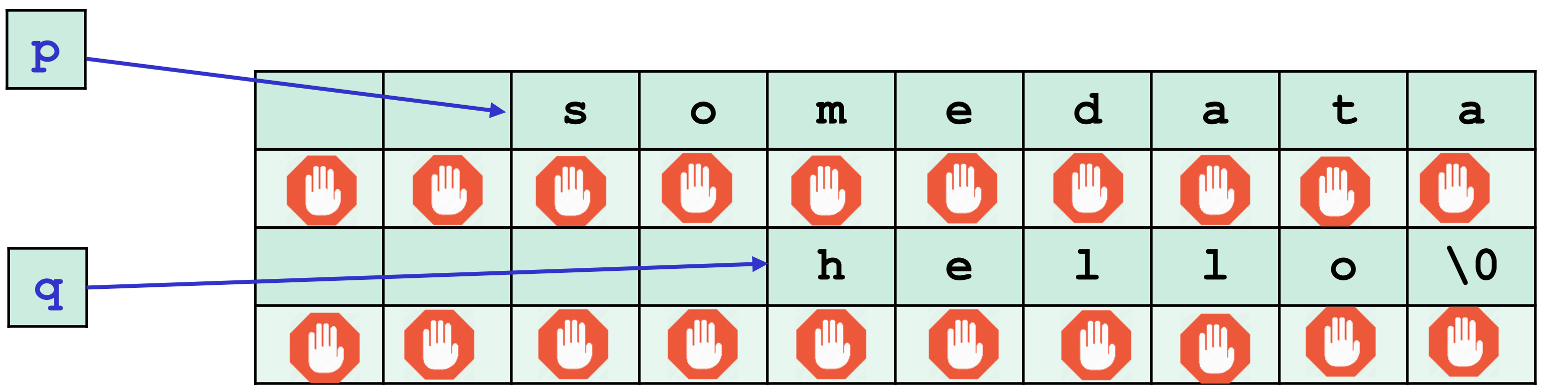
s	o	m	e	d	a	t	a
o	l	d	j	u	n	k	X
Y	Z	h	e	l	l	o	\0

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, ie. small buffer overruns, by keeping empty space between allocated chunks (unless overrun is huge)
  - small overrun will end up in this unallocated space
- 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 19

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.

Small execution overhead, but big memory overhead

Control Flow Integrity (CFI)

Extra bookkeeping & checks to spot unexpected control flow

- **Dynamic return integrity**

  Stack canaries, or shadow stack that keeps copies of all return addresses, providing extra check against corruption of return addresses

- **Static control flow integrity**

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

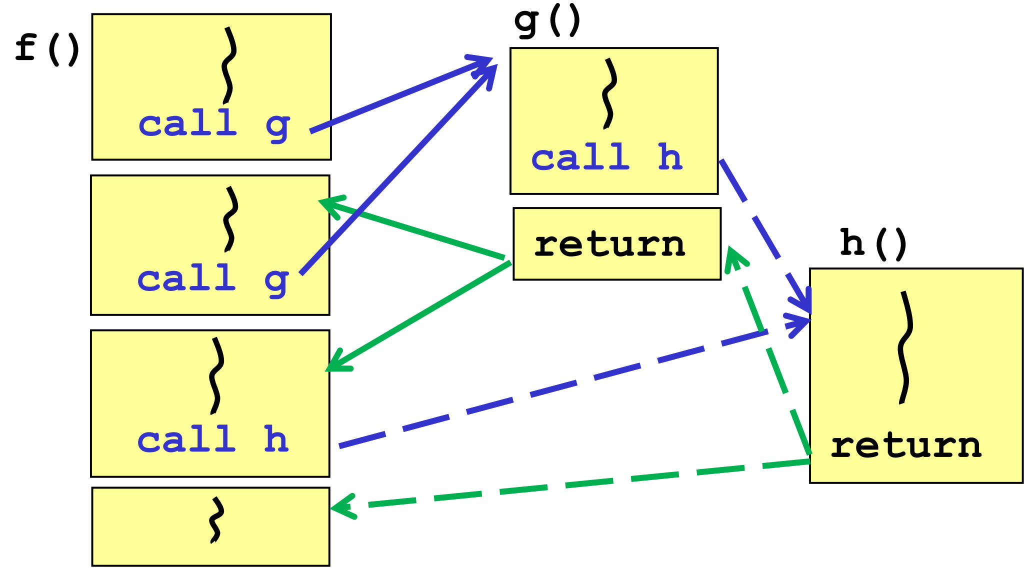
  > **If `f()` never calls `g()`,**
  > **because `g()` does not even occur in the code of `f()`,**
  > **then call from `f()` to `g()` is suspicious,**
  > **as is a return from `g()` to `f()`**

  Interrupting execution when this happens prevents (some) attacks.

  This can detect some 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 return jumps still allowed; eg if we call `h()` from `g()`, and return to `f()` 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: we'd need
  - a points-to analysis to determine where such code pointers can point to

    eg in C++, if `Animal.eat()` can resolve to `Cat.eat()` or `Dog.eat()`, so both these addresses are valid targets for transferring control
  - or: simply allow transfer to any function entry point

  Microsoft Control Flow Guard (CFG) performs such checks

New(er) features of modern OS

Stack canaries, ASLR, and NX are standard, except on very cheap devices (eg in IoT).

Some fancier features are slowly becoming used:

- 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

The big & tricky design question

Is the extra overhead of some protection mechanism worth the extra protection?

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, ASLR, non-executable memory, CFI, bounds checkers, pointer encryption - work**
- **Explain why they might not always work**

Evolution of CFI at Microsoft (*not* exam material)

If you're curious to know how usage of CFI in Windows has evolved (up to 2018), watch the talk by Joe Bialek at OffensiveCON 18

The Evolution of CFI Attacks and Defenses

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

Recent developments at Apple (*not* exam material)

Apple has started to leave runtime checks for bounds safety in production code, to prevent (some) spatial bugs, but not temporal bugs.

See Yeoaul Na's keynote talk at LLVM'23

"-fbounds-safety": Enforcing bounds safety for production C code

https://www.youtube.com/watch?v=RK9bfrsMdAM