Software Security

Memory corruption

Countermeasures

Erik Poll

Digital Security

Radboud University Nijmegen

Overview

Last week

- **Memory corruption**
  - **temporal & spatial**
  - **accessing uninitialised memory**
- **Stack canaries**

  to detect memory corruption attacks that corrupt control flow (by corrupting return addresses on the stack)

Today

- **more runtime countermeasures**
- **static analysis (SAST)**

ASLR

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**
- **This is security by obscurity. Despite its bad reputation, this increases the cost & efforts for attackers**
- **Once the offset leaks, we're back to square one…**

Non-executable Memory

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

- JIT (Just In Time) compilation, where e.g. JavaScript is compiled to machine code at run time, introduces a complication. *Why?*

  Data *written* by JIT compiler to be *exectuted*

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 orienteded 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, attacker 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

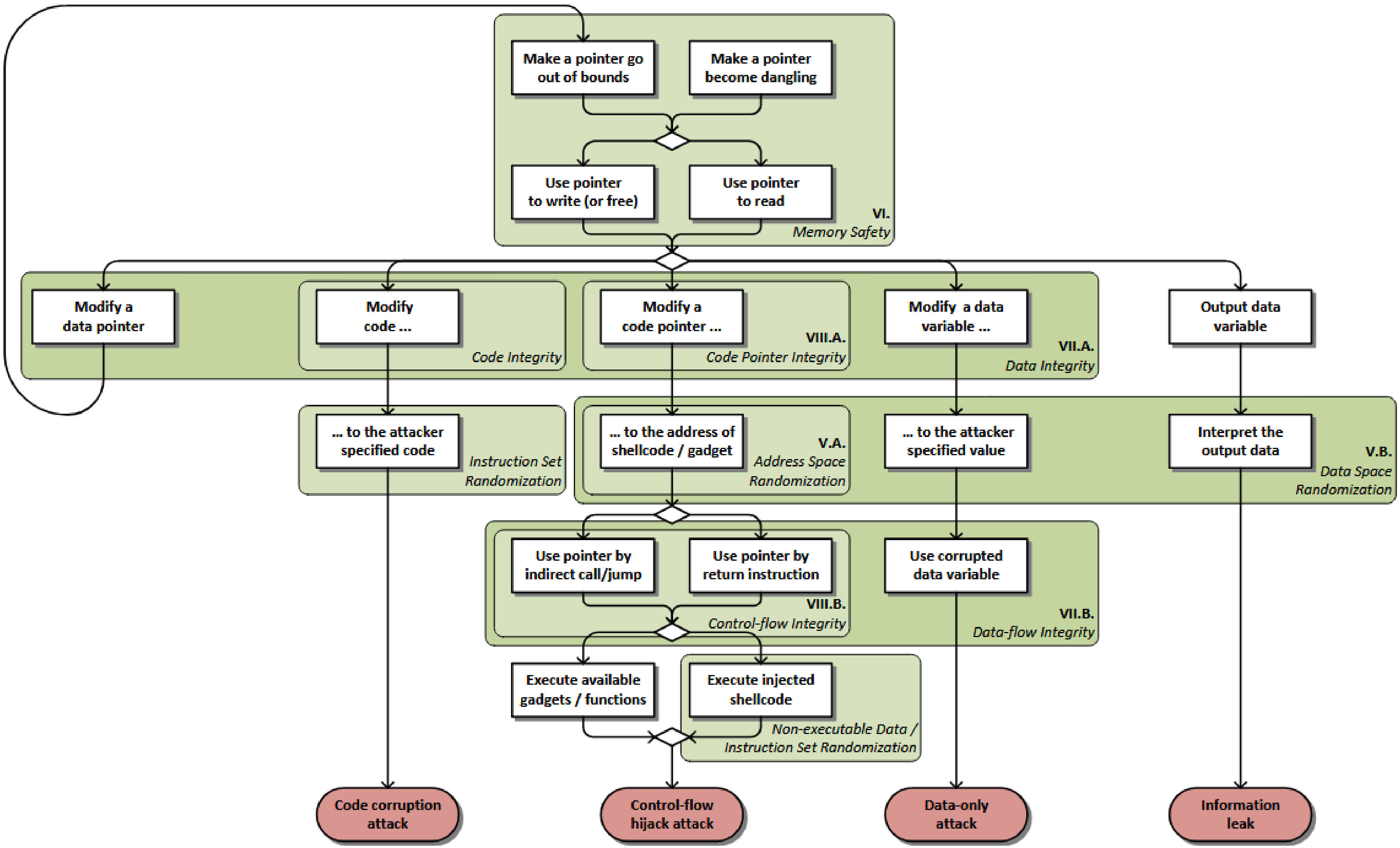
This turns out to be doable

- 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

[SoK Eternal War in Memory paper]

SoK: Eternal War in Memory



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; 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` is known at compile time

  Compiler can hard-code this static address in the binary, and then W⊕X prevents attackers from corrupting it

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

  by looking it up in the virtual function table (`vtable`)

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

Classification of defences

- **Probabilistic methods**

  Basic idea: add randomness to make attacks harder

- **Memory safety checks**

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

- **Control-Flow hijack checks**

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

Defenses can have overhead in time or space

Hardware support may be needed or reduce overhead

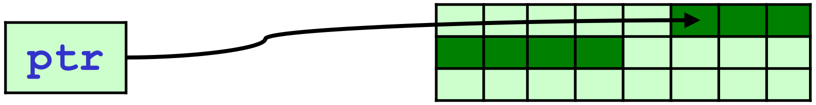
Defenses can break binary compatibility

if compiler adds extra book-keeping & checks,
all libraries may need to be re-compiled with that compiler

Memory safety checks

More memory safety

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



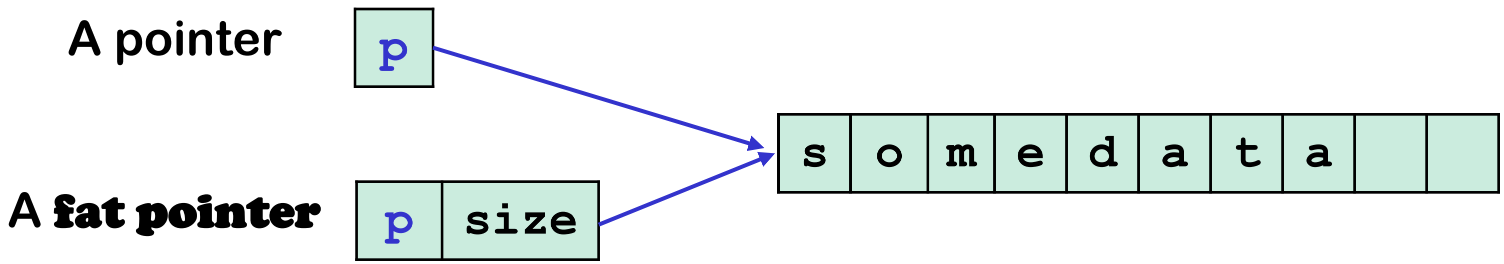
Different possibilities

- 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

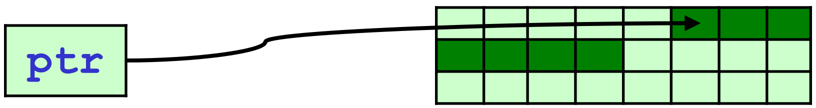


Downsides

- 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

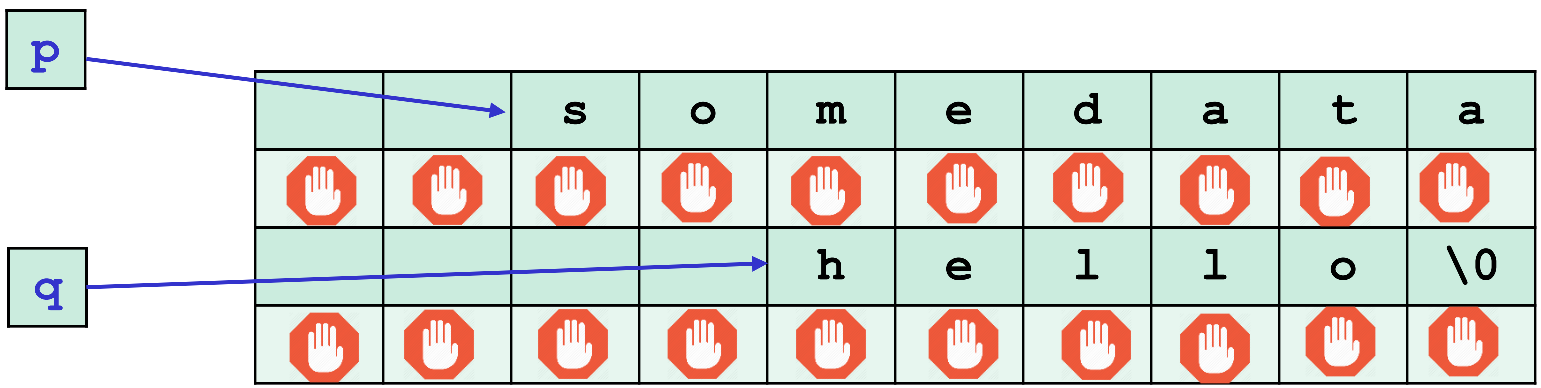
to generate runtime error when code tries to read/write unallocated memory

- This can also detect spatial bugs, ie. buffer overruns, by keeping empty space (aka red zones) between allocated chunks
  - unless overrun is very big; small overruns access unallocated memory
- This cannot spot illegal access via a stale pointer if the data it points to has been re-allocated

Eg the last bug, line 3004, on slide 20 from last week

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

Protecting Pointers

with *randomness* or *checks*

Protecting pointers

Many buffer overflow attacks involve corrupting pointers,

pointers to data or code pointers

To complicate this:

1. **we can add noise, to make it hard to corrupt pointers in predictable way,**
   a) **with ASLR**
   b) **with pointer *encryption***

2. **we can add checks, to detect corrupt pointers**
   a) **with pointer *authentication* or pointer *tagging***

Pointer Encryption (eg. PointGuard)

To complicate corruption of pointers:

store pointers encrypted in main memory, unencrypted in registers

- **requires a very 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.**

- **PointGuard has 2% performance overhead and no memory overhead**

- **Joan Daemen's PhD student Yanis Belkheyar in our group works on lightweight ciphers for pointer encryption for Intel Cryptographic Capability Computing (C^3). Yanis defends his PhD thesis on 18 November**

More extreme variant: encrypt all data

Data Space Randomisation (DSR)

- not only store pointers encrypted in main memory,
  but store all data encrypted in main memory
- Some AMD chips support this under name SME (Secure Memory Encryption)

Pointer Authentication (or pointer tagging)

Instead of encrypting pointers,

we can add an integrity check to pointers

- **a cryptographic checksum or just a tag**

Downside compared to pointer encryption?

Not only *runtime overhead*, but also *memory overhead*

Ideally, we can use spare bits in 64 bit words that not needed to address all memory of a process

- **ARM Memory Tagging Extension (MTE) adds 4 bit tag to pointers**
- **ARMv8.3 Pointer Authentication adds 3 to 24 bits Pointer Authentication Codes (PACs) to pointers using fast QARMA cipher**

Protecting Control Flow

with *randomness* or *checks*

Protecting control

We can protect control flow

1. **by adding noise, to make it hard to corrupt pointers in predictable way,**
   - **eg. with ASLR**

2. **by adding checks, to detect corrupt control flow**

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

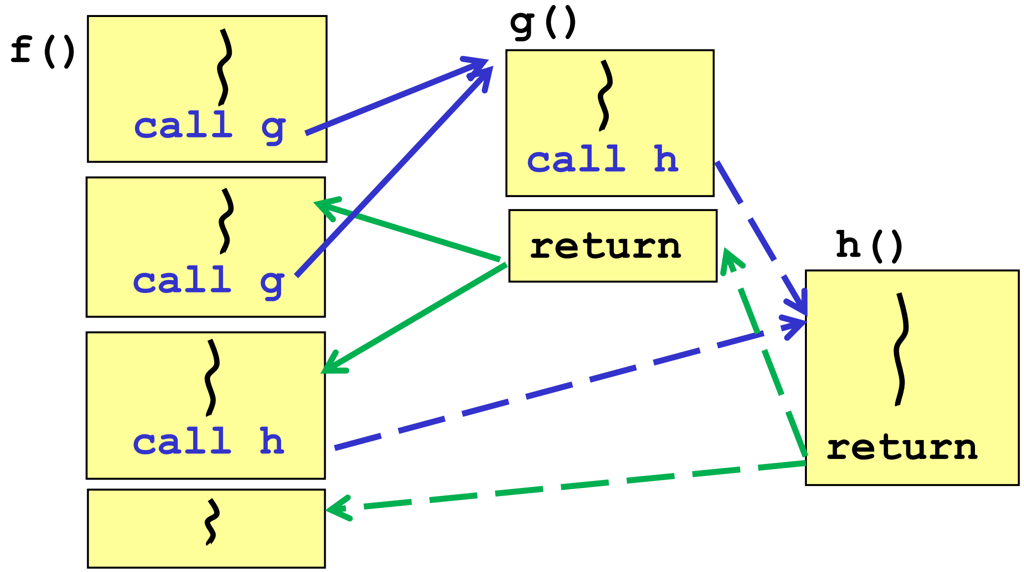
  If `f()` never calls `g()`,
  because `g()` does not even occur in the code of `f()`,
  then call from `f()` to `g()` is suspicious,
  as is a return from `g()` to `f()`

  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 each transfer of control (function call or return) the compiler can insert 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: we would 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()`, both these addresses are valid targets for transferring control
  - or: simply allow transfer to any function entry point

State of the art in memory protection

Stack canaries, ASLR and NX are standard

except on very cheap devices (eg in IoT)

Fancier protection mechanism are becoming more widely used:

- **Intel Memory Protection Extensions (MPX) introduces bound checks for pointers** (in 2015 Skylake architecture, discontinued in 2019)
- **Pointer encryption in iOS** (2018)
- **Hardware-enforced Stack Protection in Windows 10** (2020)
  with shadow stack, using Intel Control-flow Enforcement Technology
- **ARM Enhanced Memory Tagging Extension** (2022)
- **Apple Memory Integrity Enforcement (MIE) announced Sept 2025**

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