Memory corruption

Platform-level Countermeasures

Erik Poll

Digital Security

Radboud University Nijmegen

application-level vs platform-level defences

This week: platform level countermeasures

platform = compiler + OS + CPU

- **programmer does not have to do anything**

Next week: application level countermeasures

- **programmer has to be aware & do something**

Countermeasures can be hunting for bugs
or making things more robust

Trend : more platform level defences

Most platforms have stack canaries, ASLR, and NX (aka DEP or W⊕R)

More advanced features entering mainstream in past years

- Intel introduced CET (Control-flow Enforcement Technology) with shadow stack and IBT (Indirect Branch Tracking) in 2020

  Windows 10 introduced Hardware-enforced Stack Protection with a shadow stack, using Intel CET, in 2020

- ARM introduced PAC (Pointer Authentication Codes) and MTE (Memory Tagging Extension)

  PAC used in iOS since 2018

  MTE used in Google Pixel 8 in 2023 and iPhone 17 in 2025

At of this lecture, the essence of all mechanism should be clear

There are many more non-mainstream solutions, eg. LLVM/clang also has SoftBound+CETS , that I will not try to cover in this lecture

What does "platform" really mean?

To execute, software needs a "platform" that provides

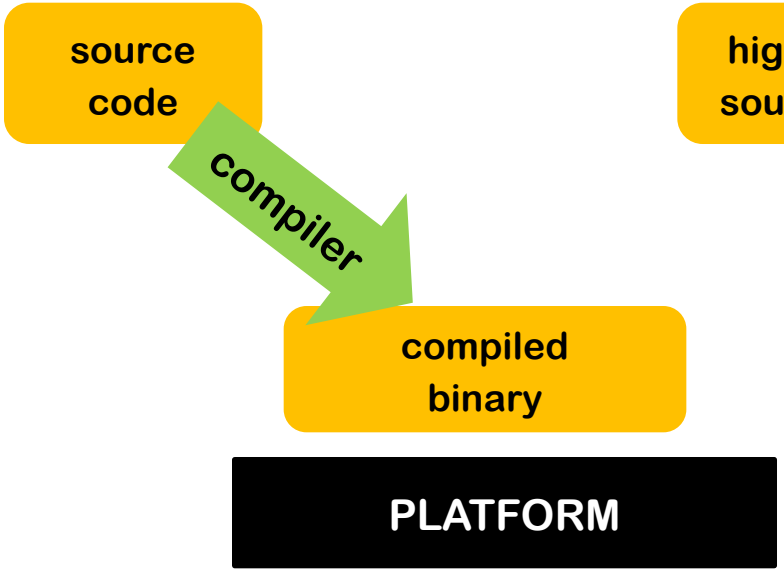
1. execution engine
   a) for (compiled) binary code: CPU
   b) for interpreted languages: virtual machine aka interpreter
2. "system API" to access OS functionality

   E.g.

   - I/O with keyboard, screen, file-system, network
   - `malloc()` & `free()` in C, `new` & `delete` in C++, `new` in Java, C#, JavaScript, `__new__()` in Python
   - `exit()`
   - …

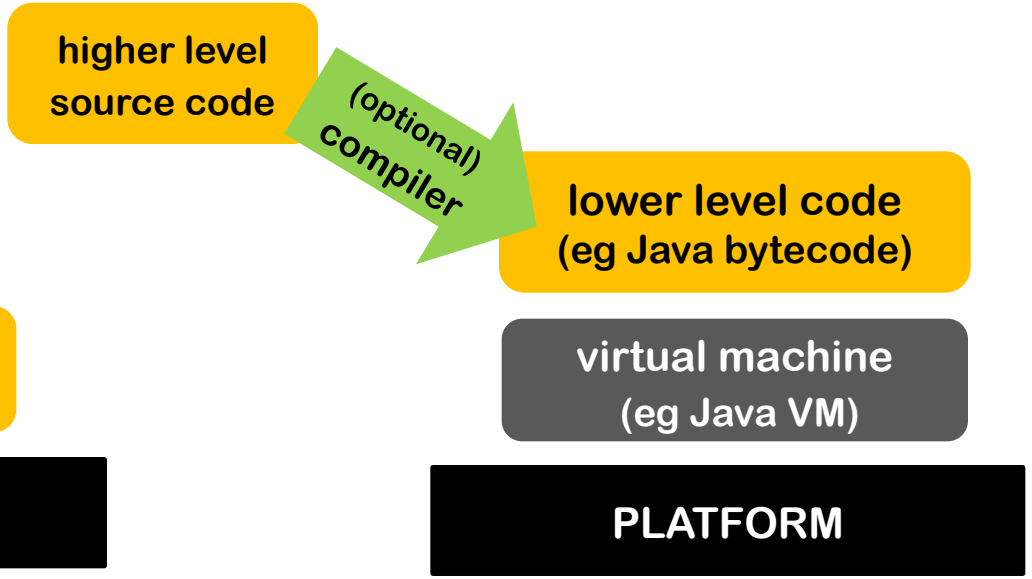
Compiled vs interpreted languages

Compiled binary runs on bare hardware



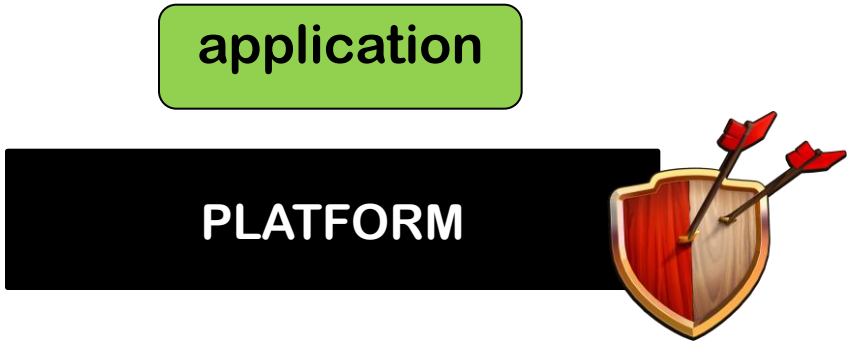
For C/C++, defensive measures have to be compiled into the code or provided by the hardware

Interpreted language runs on virtual machine



Here defensive measures can also be done by virtual machine

Platform-level countermeasures



Compiler and/or hardware insert runtime checks
to prevent certain types of memory corruptions
or mitigate their impact / making them harder to exploit

\+ Programmers do not need to be aware

\- Overhead - in execution time, memory use, and code size

\- May require specific hardware

\- May break binary compatibility

\- Common (inevitable?) residual risk: DoS

Types of memory corruption bugs

Platform defenses usually only protect against some types of memory corruption, eg

Spatial

- reading / writing out-of-bounds
- broken pointer arithmetic
- type confusion
- intra-object corruption
  (for C structs and C++ objects)

Temporal

- use-after-free
- double-free

heap

- *stack*-use-after-return
- reading uninitialised memory
- null pointer dereference
- data race

Last week

Basic platform defenses

1. stack canaries
2. ASLR
3. NX

Limitations

- return-to-libc or ROP attacks can circumvent NX
- data-only attacks are not prevented by NX or stack canaries

This week: more advanced platform defences

We can group them in three categories:

1. protecting memory - spatial, temporal, or initialisation
2. protecting pointers
3. protecting control flow

Some overlap:

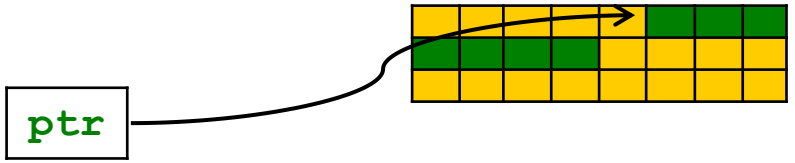
- protecting memory can also protect pointers
- protecting memory or pointers, esp. code pointers, can also protect control flow

More advanced measures (1) : protecting memory

Memory safety checks

ie. additional runtime checks

```
char[7] ptr;
```




```
ptr[i] = ...; // ok if 0 <= i < 7
ptr++; ptr++; ... ; ptr++;
char c = *ptr;  // ok if *ptr still points to a green cell
```

Additional runtime safety check requires some 'meta-data' about pointers and/or memory cells

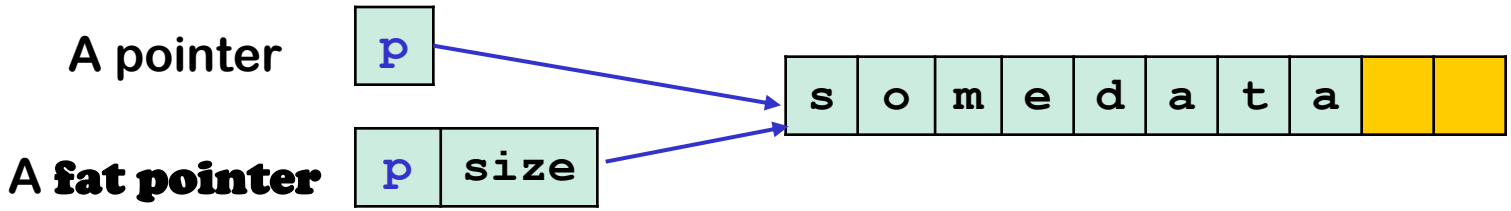
Different places to record the required meta data

- **with each pointer, in so-called fat pointers (aka wide pointers)**
- **in shadow memory map**

Fat pointers - meta-data in pointers

The compiler adds code to

- record size information for all pointers
- do runtime checks for pointer arithmetic & array indexing



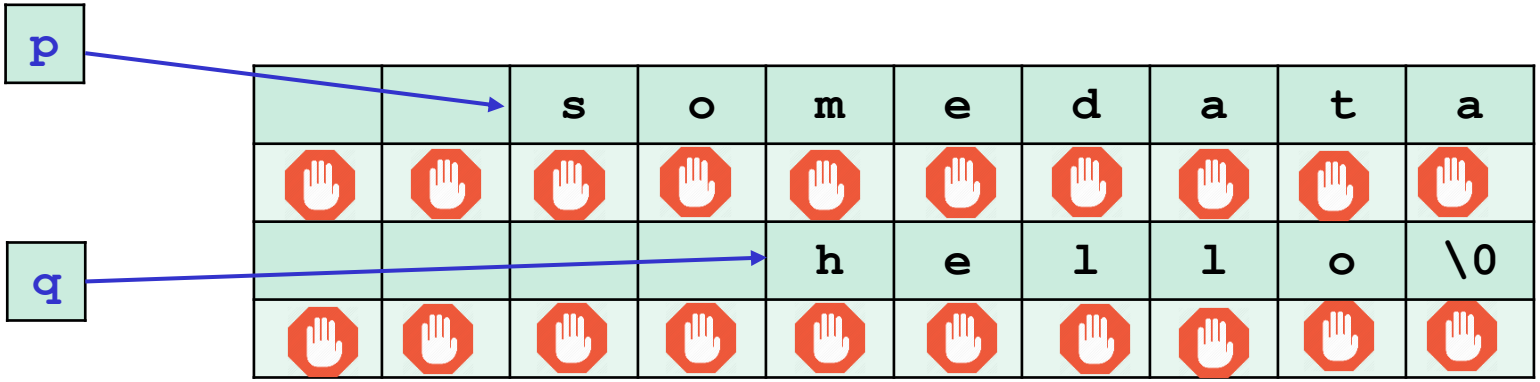
Pros / Cons / Limitations?

- Works for spatial bugs, but not temporal bugs
- Overhead in memory & execution time
- Not binary compatible
- For really weird code this won't work

```
float f = sqrt(2); char* p = (char*) f;
```

Alternative: guard pages

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



Pros/Cons/Limitations?

- No meta-data needed !
- Buffer overwrite/read will cause a memory fault
  but *under*read/write will not
- Smaller execution overhead, but MUCH bigger memory overhead

Alternative: shadow memory map

We record which bytes have been allocated in shadow memory, with one bit per byte, and check for access to unallocated 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

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

Pros/Cons/Limitations?

- This can detect temporal bugs

  But not if stale pointer is used after memory has been re-allocated

- It can also detect spatial bugs, if the allocator leaves unallocated space - a red zone - between allocations

  But not if a buffer overrun is very big

- With second bit per cell, to tell whether it has been initialised, we can also detect reading of uninitialised memory

- Memory overhead of 1 (or 2) bits per 'cell';
  Time overhead 2x on any memory access

ASan & MSan

Testing tools that use shadow memory to find memory corruptions.

But only in testing phase, too slow & big to use in production.

- ASan (AddressSanitiser) uses shadow memory with red-zones to detect spatial & temporal memory bugs
  - Not all of them: - not if spatial bug jumps the red-zone
    - not if temporal bug hits re-allocated memory
  - Freed memory is kept in quarantine for a period to improve detection of temporal bugs
  - Typical slow-down: 2x
- MSan (MemorySanitiser) uses shadow memory to detect access to uninitialised memory
  - Typical slow-down: 3x

ASan supported by clang (2012), gcc (2013) and mscv (2019)

Nice overview of checks & examples at https://learn.microsoft.com/en-us/cpp/sanitizers/asan

[Aside: Other sanitisers: UBSan, TSan, LSan]

Complimentary tools, with more ad-hoc checks

unrelated to how ASan & MSan work using shadow memory

- **LSan (LeakSanitiser) checks for memory leaks**

- **UBSan (UndefinedBehaviourSanitiser) checks for a.o.**
  - **integer overflow and**
  - **array bounds checks if bounds can be statically determined**

  For full list, see https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

- **TSan (ThreadSanitiser) checks for data races**

Can shadow memory catch a typical use-after-free exploit?

```
// Vulnerable code
File f = new File(…);
…
delete f;
…     // Attacker-induced behaviour,
      //   assuming File objects are 1024 bytes
      char* b_1 = new char [1024];
      fgets(b_1, 1024, payload);   // attacker-supplied exploit
      …
      char* b_n = new char[1024];
       fgets(b_n, 1024, payload);
      // attacker hopes that one of the b_i will be aliased with f
…
f->close(); // use-after-free
```

No, none of the checks discussed so far will catch this.

See (fake) OO examples in Brightspace to understand how use-after-free exploit works

Memory tagging

-

a variant of shadow memory

Detecting this attack using *memory tagging*

```
// Vulnerable code
File f = new File(…);
…
delete f;
…  // Attacker-induced behaviour,
   //  assuming File objects are 1024 bytes
   char* b_1 = new char [1024];
   fgets(b_1, 1024, payload);
   …
   char* b_n = new char[1024];
    fgets(b_n, 1024, payload);
   // attacker hopes that one of the b_i will
…
f->close(); // use-after-free
```

When we allocate memory, both the pointer & the memory are tagged (or coloured) with a random tag.

When a pointer is used, we check if the tag of the pointer matches the tag of the memory

- So f and all the b_i would have different tags
- Note: *ALL* bytes in b_i have to be tagged! So quite some overhead at allocation.
  But: initialisation of newly allocated memory is then 'for free'

Alternatives to memory tagging?

```
// Vulnerable code
File f = new File(…);
…
delete f;
…     // Attacker-induced behaviour,
      //        assuming File objects ar
      char* b_1 = new char [1024];
      fgets(b_1, 1024, payload); t
      …
      char* b_n = new char[1024];
       fgets(b_n, 1024, payload);
      // attacker hopes that one of the
…
f->close(); // use-after-free
```

Suspicious behaviour:

- **f should point to a File object, not a byte array**
- **f->close() should be a code pointer to a void function without any parameters, but points to some random bytes.**

This suggests alternatives:

1. **Different custom allocators, one for Objects and one for byte arrays, so that objects and byte arrays are never aliased.**
   - **This will not prevent use-after-free bugs involving just objects, unlike memory tagging will**

   Custom allocators could be used as application level countermeasure
2. **We could have special measures to protect function pointers, to prevent random address from being used as function pointer – more on that later**

Recap: Memory tagging (aka Memory colouring)

For each memory allocation we choose a tag (or colour), and we colour both pointers and memory cells, e.g.

```
char* p = new char [1024];  // p and all allocated chars marked as purple
```

We keep track of these colours & check if pointers always point to cells of the right color.

When we de-allocate memory, we reset the color of the cells.

Pros & Cons?

\+ This detects more use-after-free bugs.
With enough colors, maybe we can catch all memory corruptions?

\- More overhead

- multiple bits for colour instead of single bit for allocated or not
- for both pointers and memory cells:
  we need both fat (coloured) pointers and shadow memory

Very expensive, but… ARM Top-Byte-Ignore to the rescue!

Memory tagging with ARM MTE

ARM Top-Byte-Ignore (TBI) leaves top byte in 64 bit address unused,

ARM Memory Tagging Extension (MTE) uses this byte to record tags

- MTE uses 4 bit tag for every 16 byte granule of memory
  - detects temporal and spatial heap bugs with 15/16 chance;
    also prevents access to unitialised memory
  - adapted allocator needed to set tags in malloc() and free()
  - 3.25% overhead on memory usage
    2x overhead on execution (for tags checks & initialisation of tags)

To read: Kosty Serebryany, ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety, Usenix login magazine, 2019

Uses of ARM MTE

- HWASan (Hardware-Assisted ASan)
  - uses memory tagging instead of ASan's bitmap with red-zones
  - with LLVM enhancement of MTE to also tag stack memory
    |and detect temporal stack bugs (i.e. use-after-return/scope)
- Apple uses MTE, with some additional protections,
  under name Memory Integrity Enforcement (MIE)

  See https://security.apple.com/blog/memory-integrity-enforcement/
- Android's default Scudo allocator can use MTE
  but not for large allocations to reduce overhead to set tags
- GLibc (the GNU C library) supports MTE
- PartitionAlloc allocator used by Chrome uses MTE
- Linux kernel can use MTE in its Kernel Address Sanitizer (KASAN)
  for one of the Linux allocators, namely the SLUB allocator

Other security features in custom allocators

Apart from inserting tags, custom allocators can also improve security by

- erasing freed memory
- keeping deallocated memory in quarantine for a while
- surrounding memory chunks with guard pages
- preventing type-confusion bugs by using different partitions for different types
- …

More advanced measures (2) :
Protecting pointers

Protecting pointers

Memory tagging also protects pointers, as the tags prevent a pointer from pointing to memory it should not be pointing to.

Alternative – or additional – ways to protect pointers:

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

2. adding integrity checks to detect corrupt (use of) pointer:
   pointer *authentication*

Pointer Encryption (eg. PointGuard)

We store pointers encrypted in main memory,
but have them unencrypted in registers

- with a very 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.

  Why would we use encryption and not authentication?

  - *Encrypted 64-bit pointer is 64-bits, so no memory overhead and – more importantly - minimal changes to code and memory layout*

PointGuard has 2% performance overhead and no memory overhead

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

We add a cryptographic checksum (MAC) to pointers and check this whenever we use a pointer

- *Downside compared to pointer encryption?*

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

  But: spare bits in 64 bit addresses may come to the rescue, as we saw with memory tagging

ARM Pointer Authentication Codes (PAC)

- Adds 3 to 24 bits MAC using the fast QARMA cipher

  3 bits security is unacceptable for most purposes,
  but here it helps make life for attackers much harder

- Used for return addresses and C++ methods calls

  For method calls, hash of method name and parameter types is thrown in the computation of the MAC (as so-called discriminator)

- ***Not*** used for data pointers or C function pointers

  But: programmers can add pointer authentication to any pointer with

```
sign (raw_pointer, key, discriminator)
auth (signed_pointer, key, discriminator)
```

  to sign and authenticate it. The `discriminator` argument can be used to different types of pointers.

See https://clang.llvm.org/docs/PointerAuthentication.html

Memory tagging vs pointer authentication

Memory tagging and pointer authentication both protect pointers.

Can you think of examples to show ARM MTE & PAC provide different protection for pointers?

- **PAC can be better than MTE:**

  MTE does not prevent an overflowing byte array field in a struct from corrupting a function pointer in that struct (because all these fields will have the same tag).
  PAC will, as this is unlikely to produce a properly authenticated pointer

- **MTE can be better than PAC:**

  PAC does not prevent attacks that replace an authenticated pointer with another autenticated pointer (if it has the same discriminator)

  E.g. `dog->eat()` can still be corrupted to point to `cat->eat()`

Of course, MTE also protects against many forms of memory corruption where PAC won't help at all because it does not involve corrupting pointers.

More advanced measures (3) :
Control Flow Integrity (CFI)

Protecting control flow

NX makes it impossible for attackers to inject or change code,
but attacker can still corrupt control flow - aka control flow hijacking

This can be done by

1. corrupting return addresses on the stack
2. corrupting function pointers
3. corrupting vtable entries for C++ method calls (special case of 2)

Some of the earlier mechanisms offer some protection

- ASLR makes it harder to corrupt addresses in predictable way
- memory tagging and pointer encryption & authentication can constrain possibilities for corrupting function pointers

Control Flow Integrity (CFI) is another set of protection mechanisms

Background: compilation of function calls

- A compiler translates function calls in C/C++ 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 the address of the code for `f` is known at compile time, except when it is a function pointer

  Compiler can hard-code this in the binary, and then NX 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`)

  NX does not prevent attackers from corrupting code pointers in these tables

Background: Control flow basics

- Control flow changes happen through branches
  eg. due to if-then-else, goto/jump or function call
- Function calls are special branches because they involve forward edge (ie. the call) and backward edge (i.e the return)
- Branches can be
  - direct: destination address hard-coded/fixed at compile-time
    Eg. `printf()`
  - indirect: destination address determined at runtime
    Eg. `Animal->eat()`

  Indirect branches are usually function calls, but CPUs can also have jump instructions that are indirect.

  - NX prevents attackers from corrupting direct branches
  - For programs with only direct branches we can determine the exact control flow graph (cfg) at compile time. For programs with indirect branches we can only over-approximate it.

CFI: Protecting *backward* edges (ie. returns)

- Stack canaries

- Shadow stack that keeps copies of all return addresses, providing extra check against corruption of return addresses

Most compilers can generate code for stack canaries.
A lot of modern hardware provides a shadow stack.

Compiler flags in gcc and clang for this include

```
-mshstk, -fstack-protector, -fstack-protector-strong, ...
```

CFI: Protecting *forward* edges

Jumping to the middle of a function, as in ROP attack, is suspicious.
Can we detect this?

We know valid start addresses of functions at compile-time, so we can check that indirect branches only jump to these locations:

1. make a list of all legal addresses and check *before* a call that we jump to an address on that list

   Used by Microsoft Control Flow Guard (CFG), introduced in Windows 8

2. mark start of a function with dummy instruction `ENDBR` and check that we land on that instruction *after* any indirect branch

   ARM calls it Branch Target Identification (BTI)
   Intel calls it Indirect Branch Tracking (IBT); it is as part of Intel Control-Flow Enforcement Technology (CET)

Does this prevent <u>all</u> control flow hijack attacks?

No: attacker can corrupt of function pointer to point to a valid, but the wrong functions (as in return-to-libc attack).

More *fine-grained* forward edge protection

- For method calls, the compiler has extra knowledge

  eg. Animal->eat() can resolve to Dog->eat() or Cat->eat()
  but not to execve() or Circle->area()

  Clang/LLVM can insert extra checks for this on method calls,
  with `clang -fsanitize=cfi*`

- For function pointers this is trickier
  - static analysis could tell the compiler the legal values of some function pointer, but that requires complex whole program analysis

- And… recall ARM Pointer Authentication Codes (PAC)

  Including hash of parameter types (and for methods, method name) in PAC authentication code constains the attacker

  For function pointers including name makes no sense,
  but including parameter types does.

  Microsoft proposed a similar mechanism, eXtended Flow Guard (XFG), for Windows 10, but abandoned it

Overview: platform-level countermeasures

- ASLR
- NX
- protecting memory - spatial, temporal, and/or initialisation
  a) fat pointers
  b) guard pages
  c) shadow memory map, eg. ASan and MSan
  d) memory tagging, eg. ARM MTE
- protecting pointers
  a) pointer encryption
  b) pointer authentication, eg. ARM PAC
- protecting control flow
  a) basic: stack canaries & shadow stacks
  b) more advanced:
     - coarse forward edge protection, eg. BTI and Windows CFG
     - finer-grained forward edge protection, eg. LLVM CFI or ARM PAC

So, problem solved?

Data Oriented Attacks

Attacker can still mess with data

Eg corrupting some important boolean flag somewhere

Any access control implemented *inside* an application can be by-passed by corrupting data in a *privilege escalation* attack

Eg. Shuo Chen et al., Non-control data attacks are realistic threats, USENIX Security, 2005

In 2016, people discovered that you can pervert many programs in weird ways just by corrupting data

Hong Hu et al. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks, IEEE S&P 2016

DOP (Data Oriented Programming)

In a DOP attack the attacker repeatedly

- corrupts some data
- then triggers execution of existing code (DOP gadgets), *without violating CFI*, to do something weird

As with ROP, many binaries offer enough DOP gadgets to provide attacker with a Turing-complete platform…

Tools like STEROIDS and Sploit can identify DOP gadgets in binaries.

Sploit comes with a compiler for an exploit programming language to automate use of identified gadget, using a variant of DOP called BOP (Block Oriented Programming)

Beware of confusion: Data-Oriented Programming is also the name of a benign programming paradigm

The never ending arms race…

1. protect return addresses with canary
   → by-pass canary by using corrupted pointers to corrupt return address
2. use ASLR
   → add NOP sledge or try to leak offset
3. make stack non-executable
   → shell code on heap
4. NX: make all writeable data, on both stack and heap, non-executable
   → return-to-libc
5. remove or protect dangerous library calls to prevent return-to-libc
   → ROP
6. Control Flow Integrity (CFI) to prevent ROP
   → DOP

Homework

- To read:

  Kosty Serebryany, ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety, Usenix login magazine, 2019

- To do:

  Use a compiler of your choice to analyse the sample buggy C programs, and then compare it with cppcheck

  Due next week before the lecture

- Also, check out the code samples demonstrating the corruption of function pointers