Hardware Support for Memory Safety

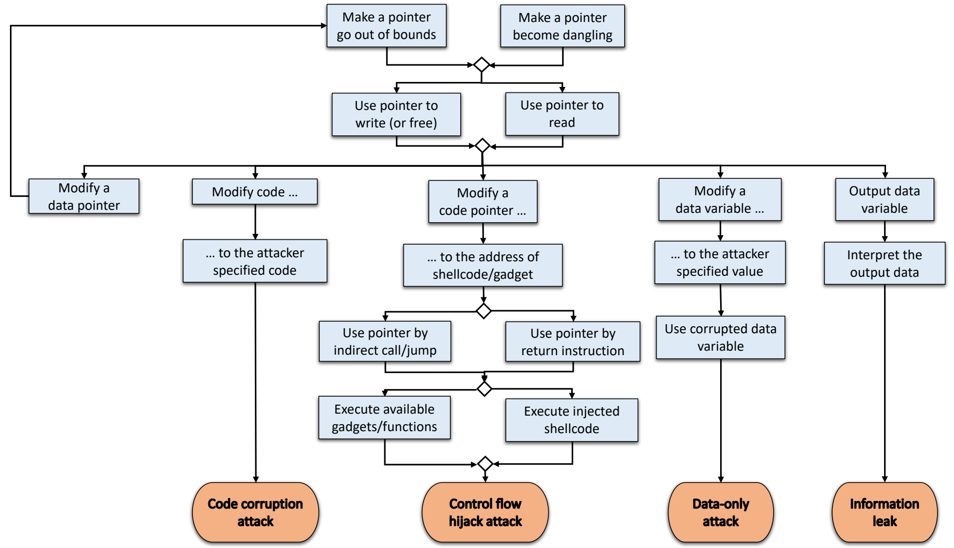
Mengjia Yan
mengjia@csail.mit.edu

6.888 Secure Hardware Design

Overview

- Memory corruption problems and existing mitigations
  - SoK: Eternal War in Memory; Szekeres et al; S&P'13

- Recent progress on hardware defenses



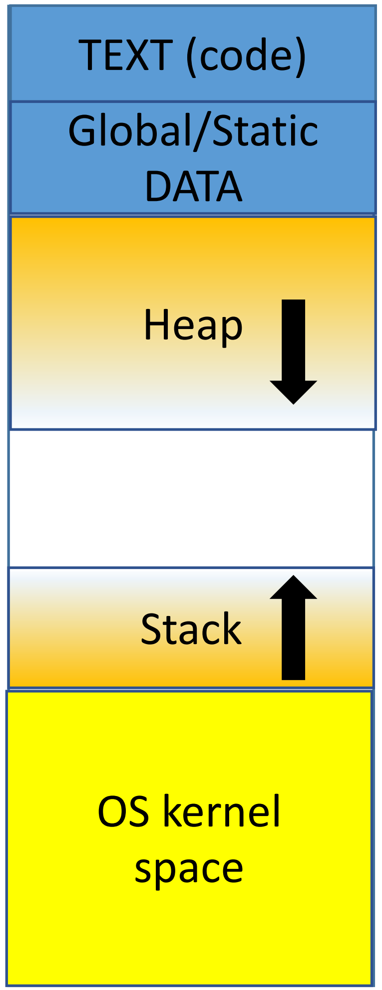
The Problem

- C/C++ is unsafe
- Everybody runs C/C++ code
- They surely have exploitable vulnerabilities

Low-level Language Basics (C/C++/Assembly)

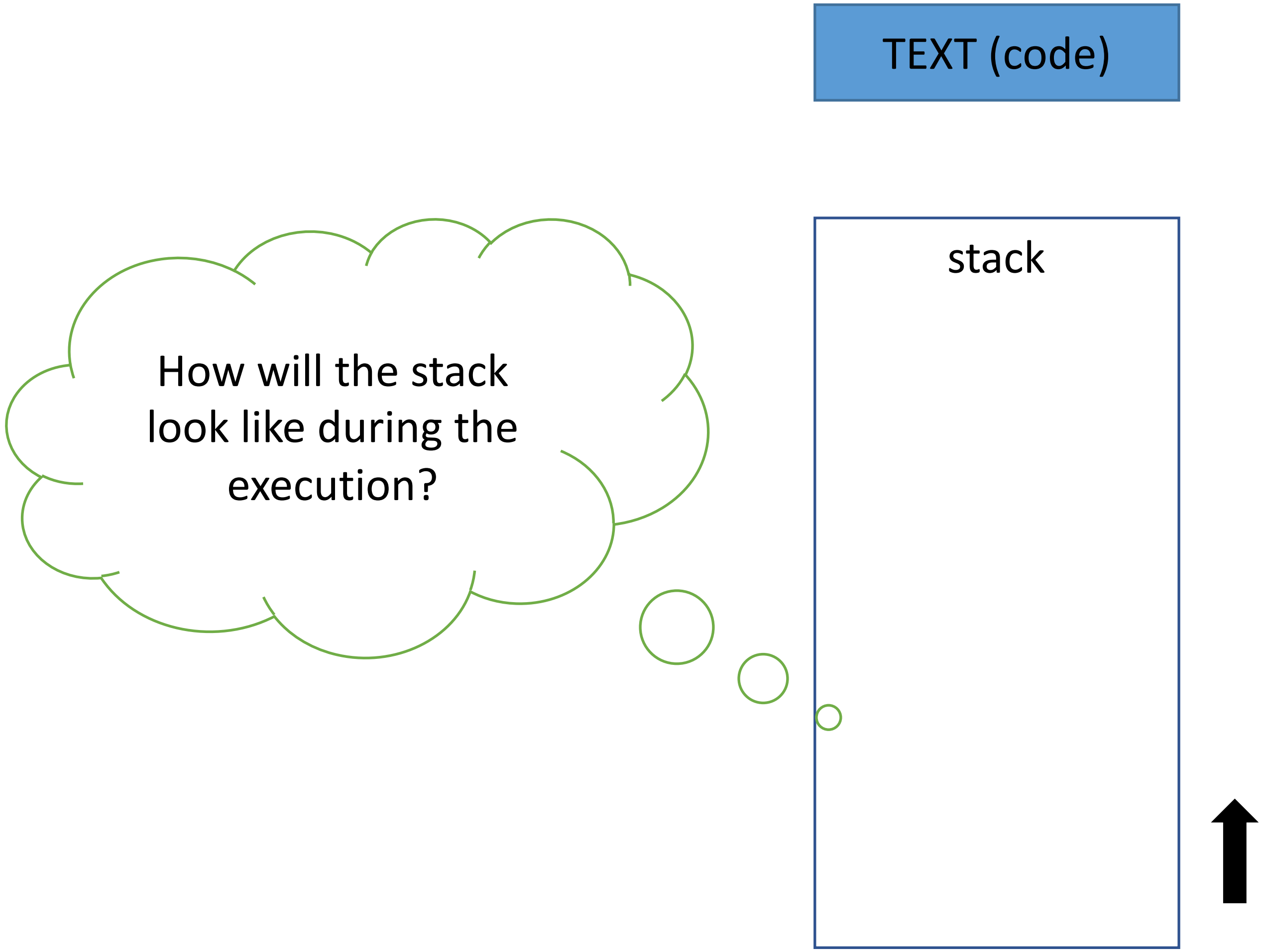
- Programmers have more control
  - + Efficient
  - - Bugs
  - - Programming productivity

- **Pointers**
  - Address of variables: index of memory location where variable is stored
  - It is programmers' responsibility to do **pointer check**, e.g. NULL, out-of-bound, use-after-free



Low-level Language Basics

```c
int hello() {
  int a = 100;
  return a;
}
int main() {
  int a;
  int b = -3;
  int c = 12345;
  int *p = &b;
  int d = hello();
  return 0;
}
```

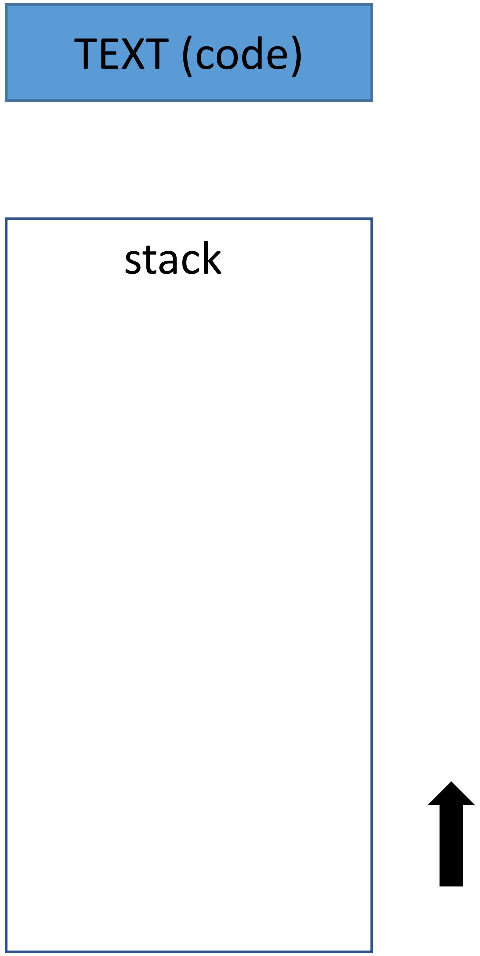


Code Injection Attack Example

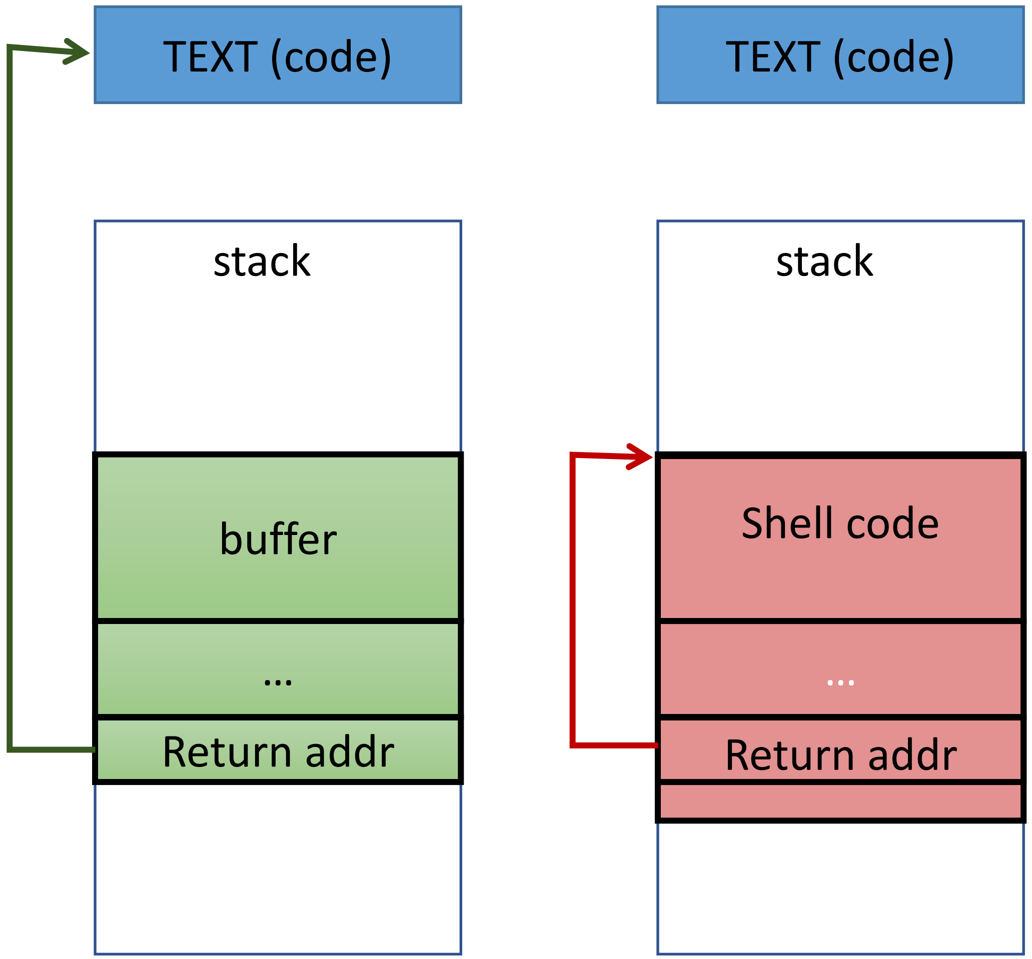
```
int func (char *str) {
        char buffer[12];
        strncpy(buffer, str, len(str));
        return 1;
}

int main() {
        ….
        func (input);
        …
}
```

Shell code:

PUSH “/bin/sh”
CALL system

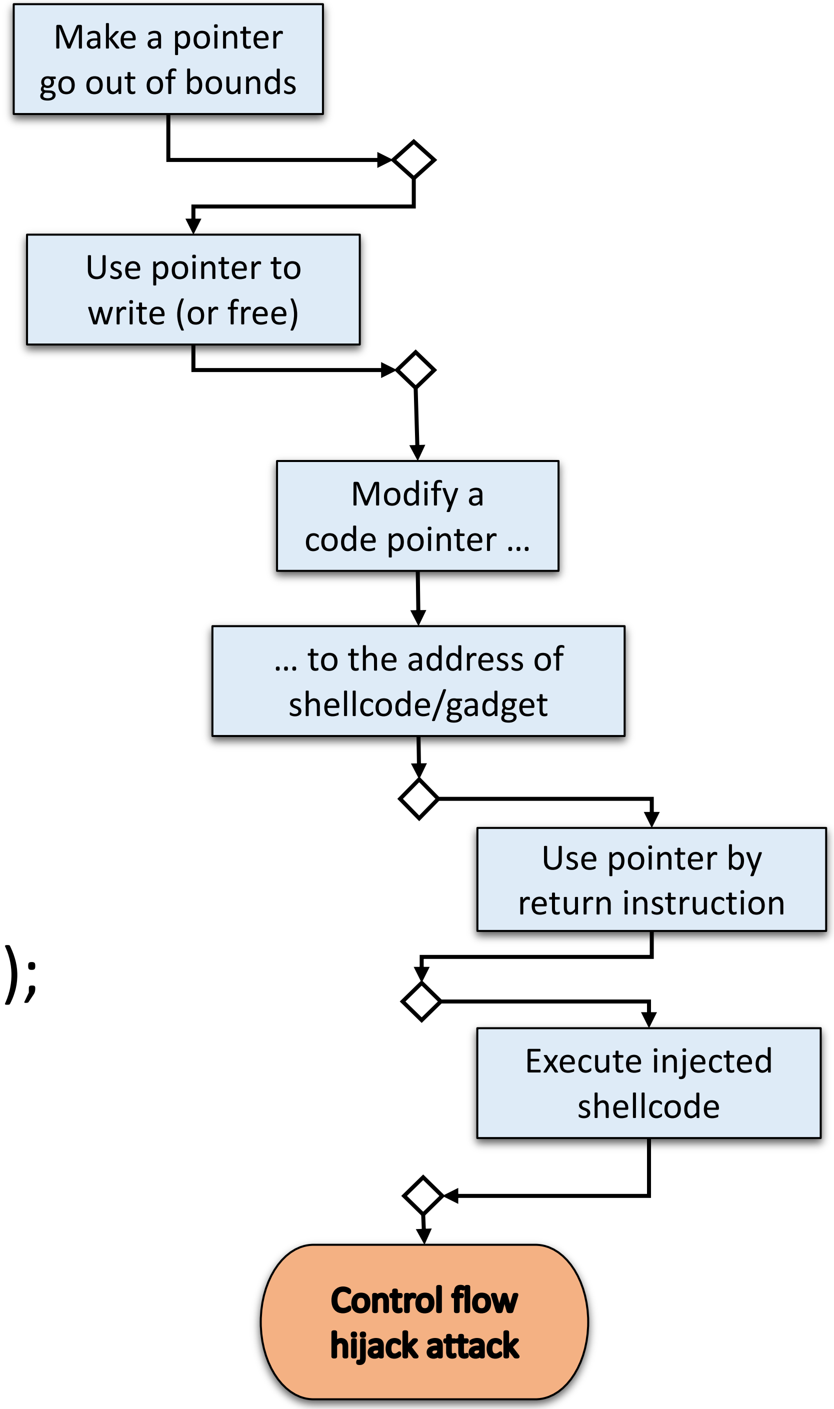


Code Injection Attack

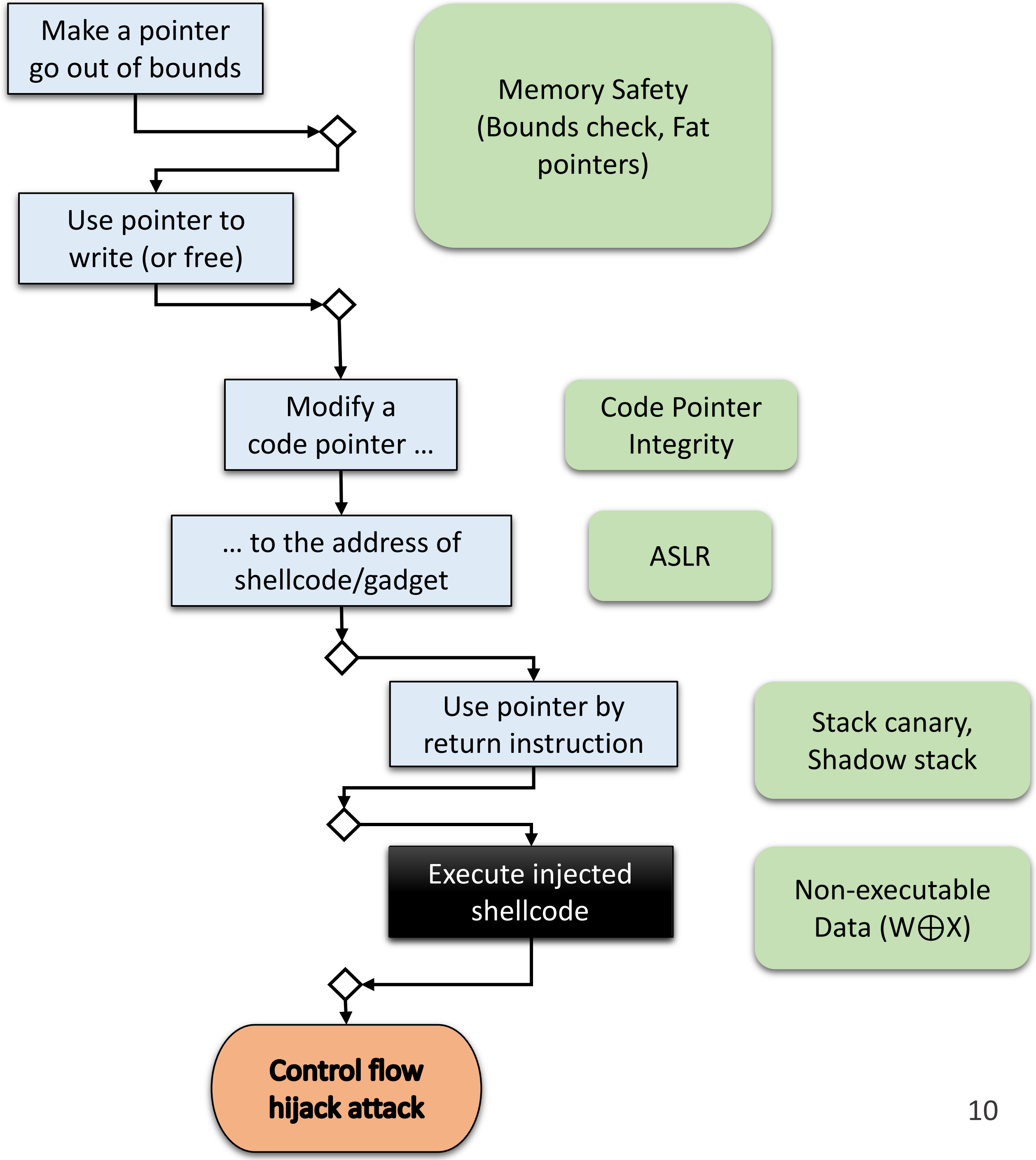


- **Think about mitigations**

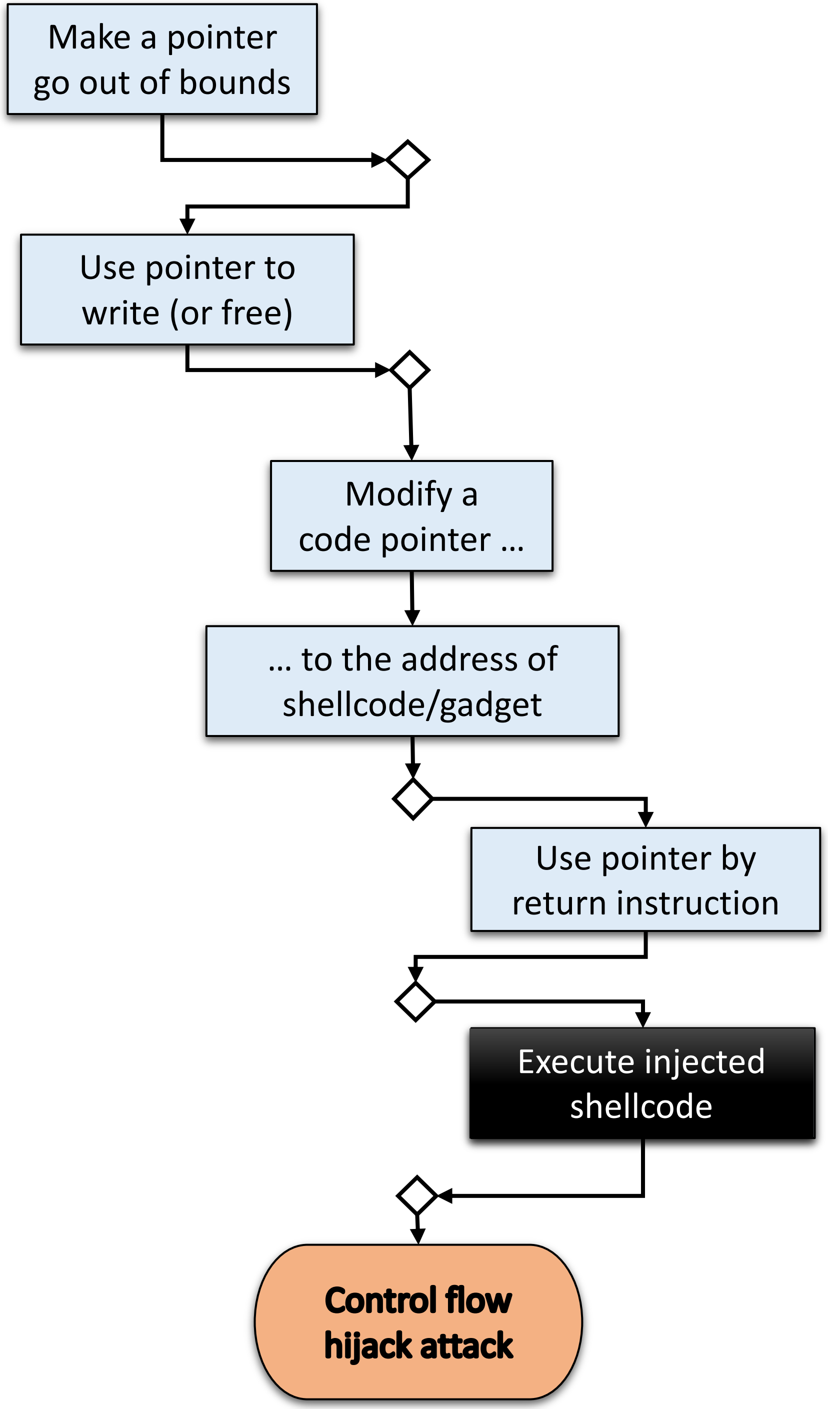
```
int func (char *str) {
        char buffer[12];
        strncpy(buffer, str, len(str));
        return 1;
}
```



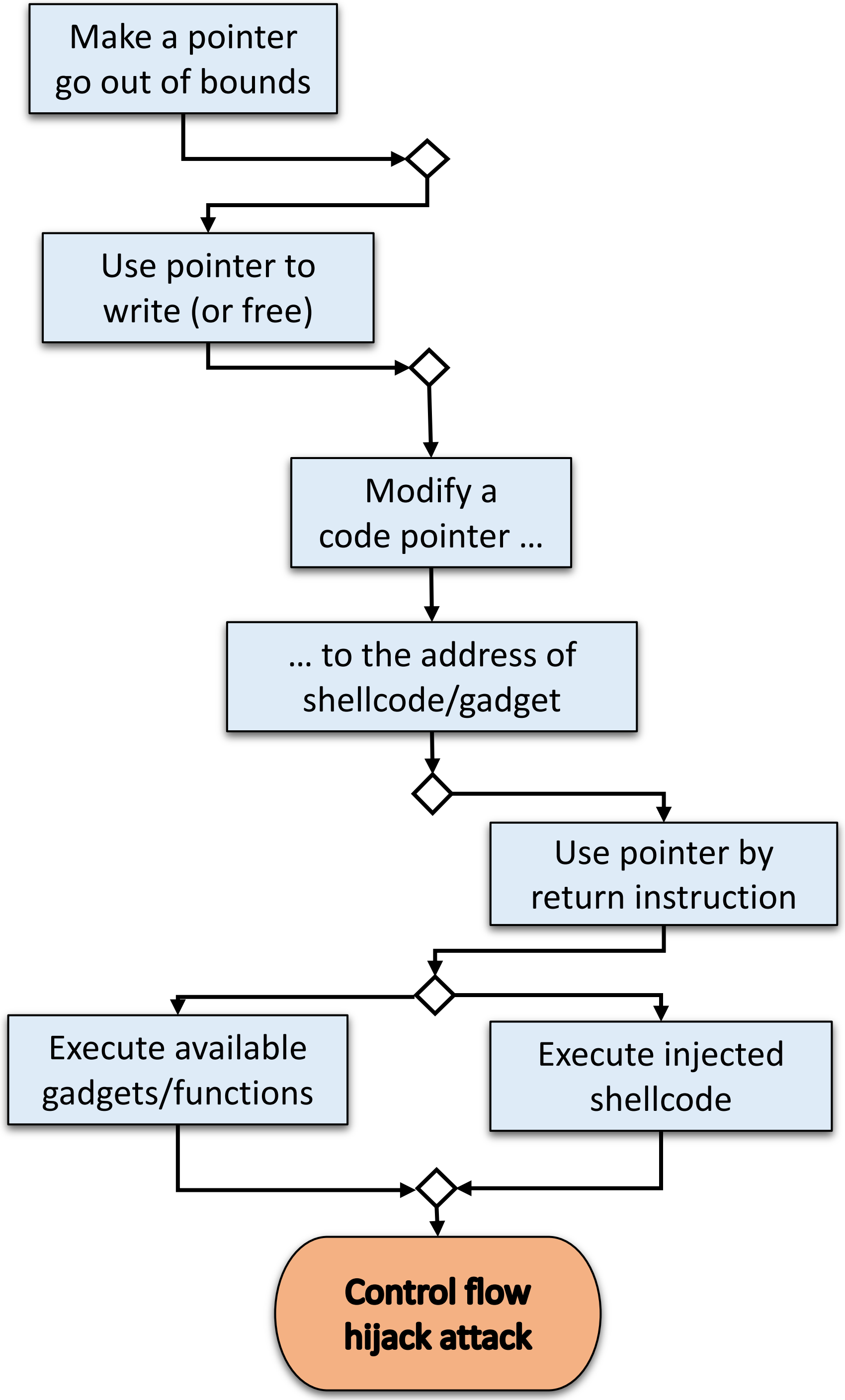
Mitigations



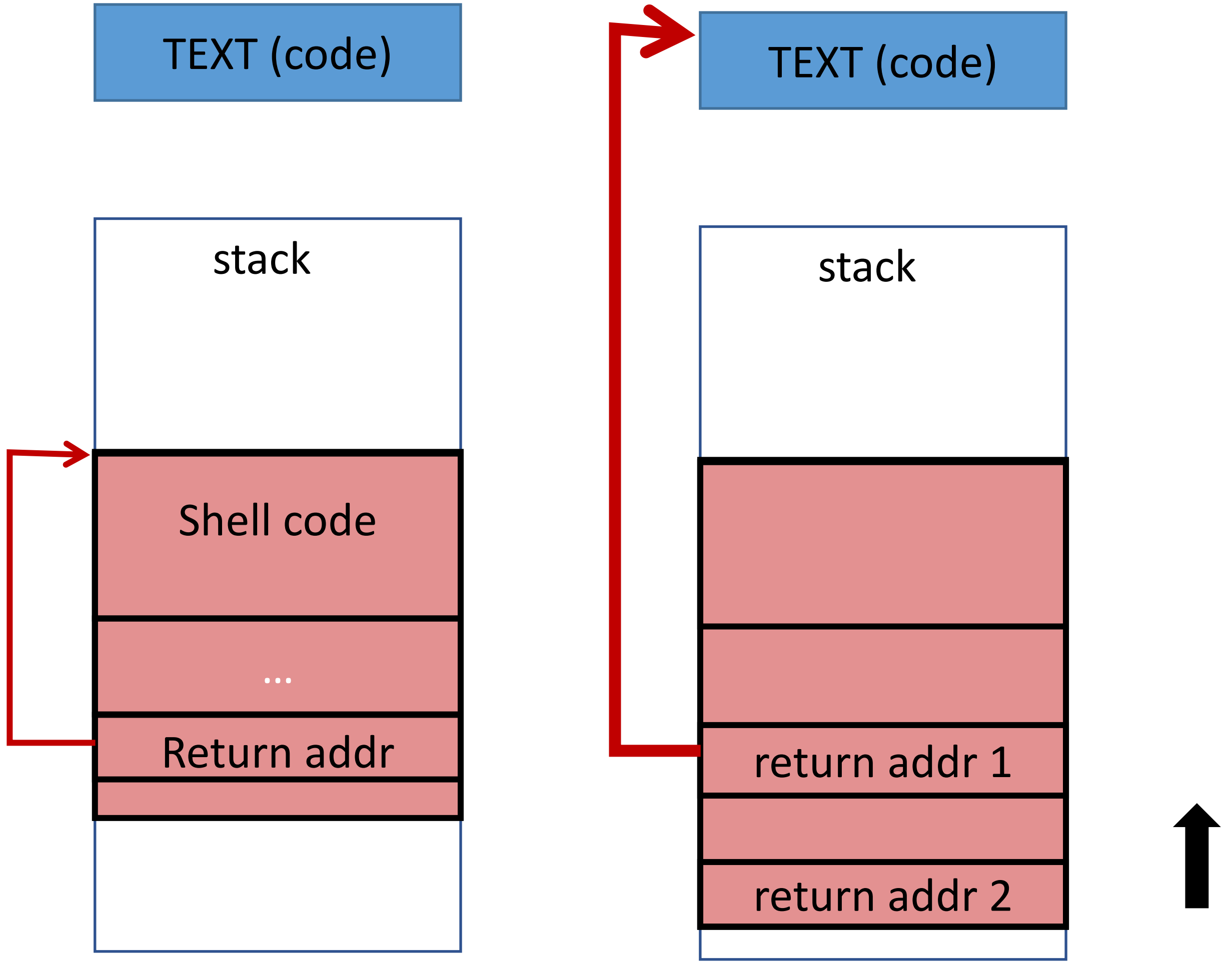
- **Think about attack variations**



Attack Variations



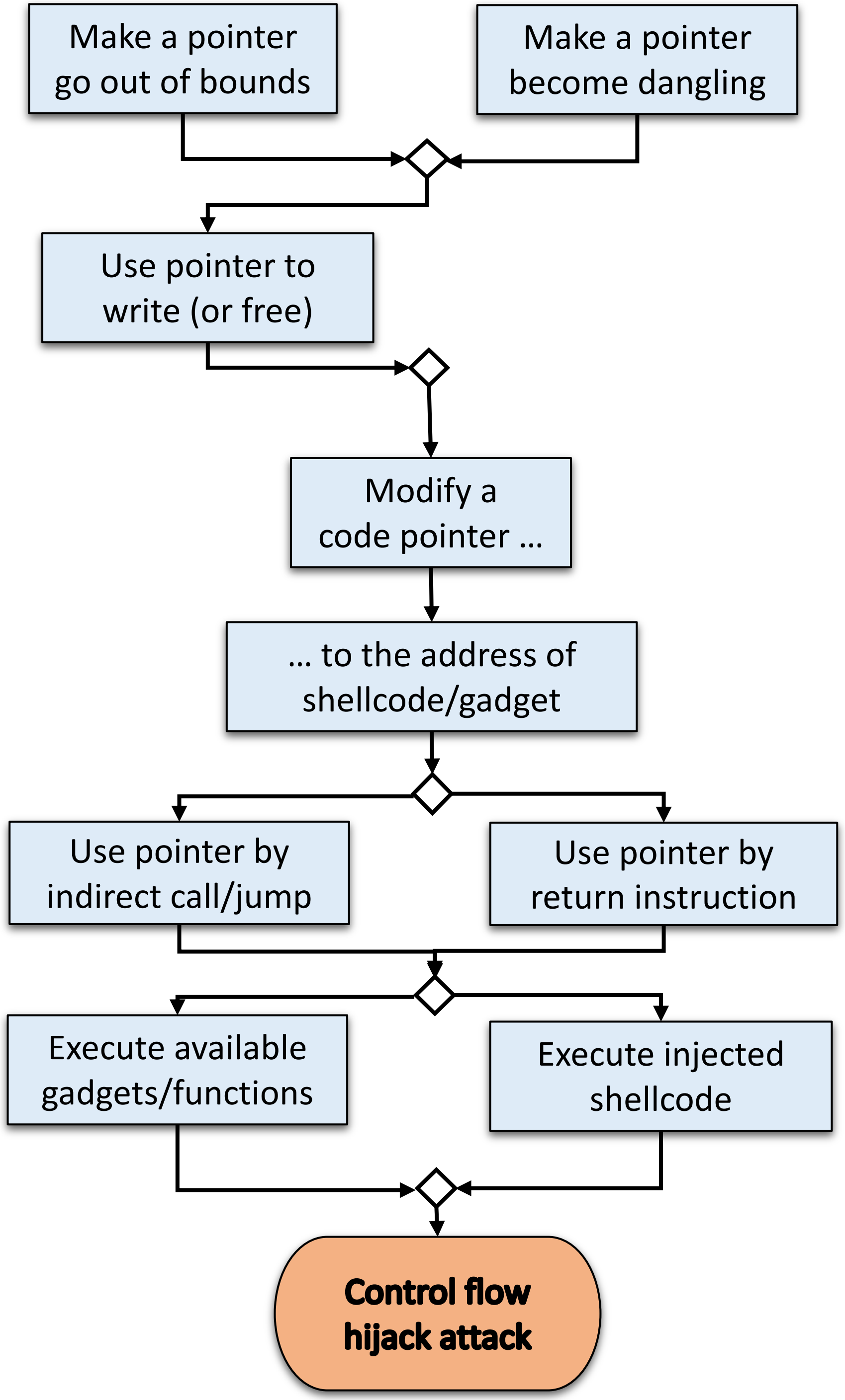
Return-Oriented Programming (ROP)



Gadget example:

pop rdi
ret

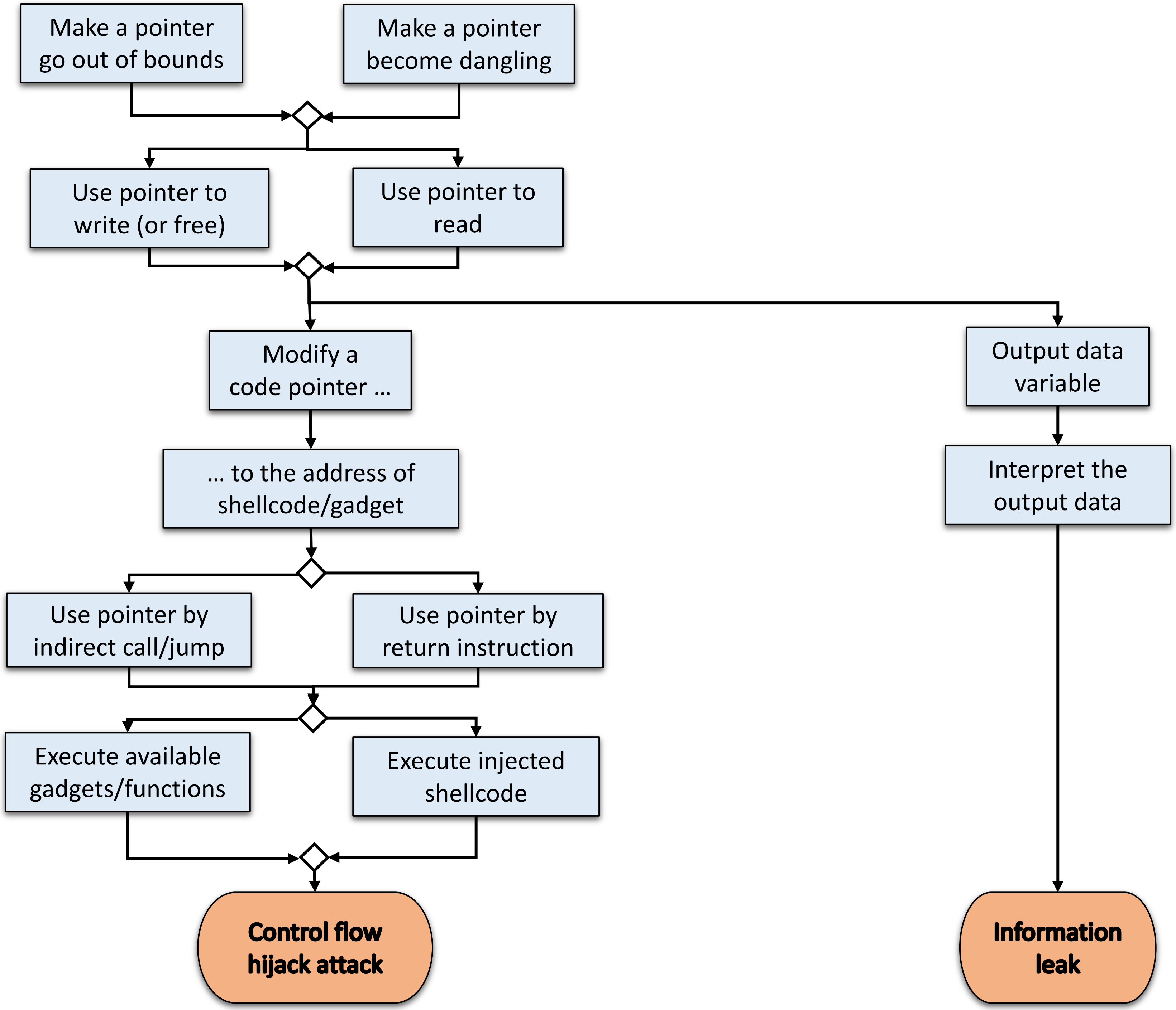
Attack Variations



Use-After-Free

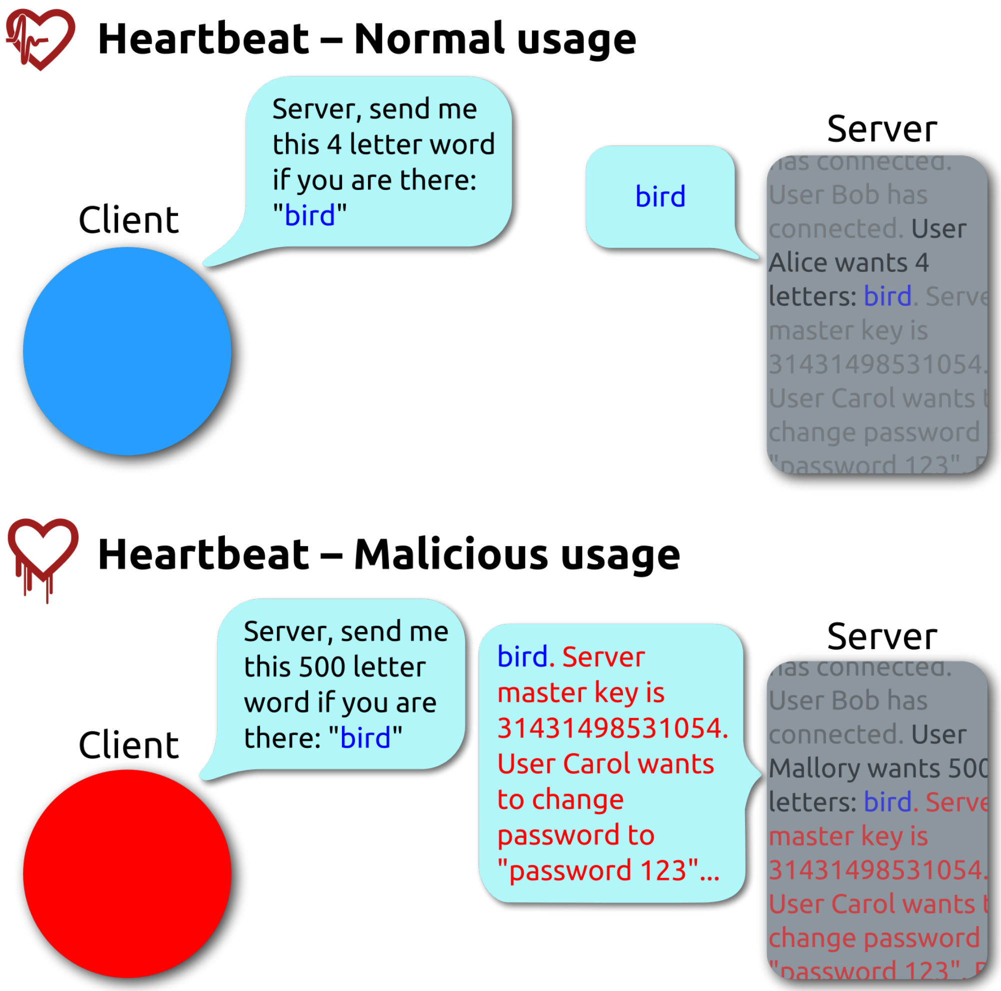
- Example
- How to check?

- **More Variations**

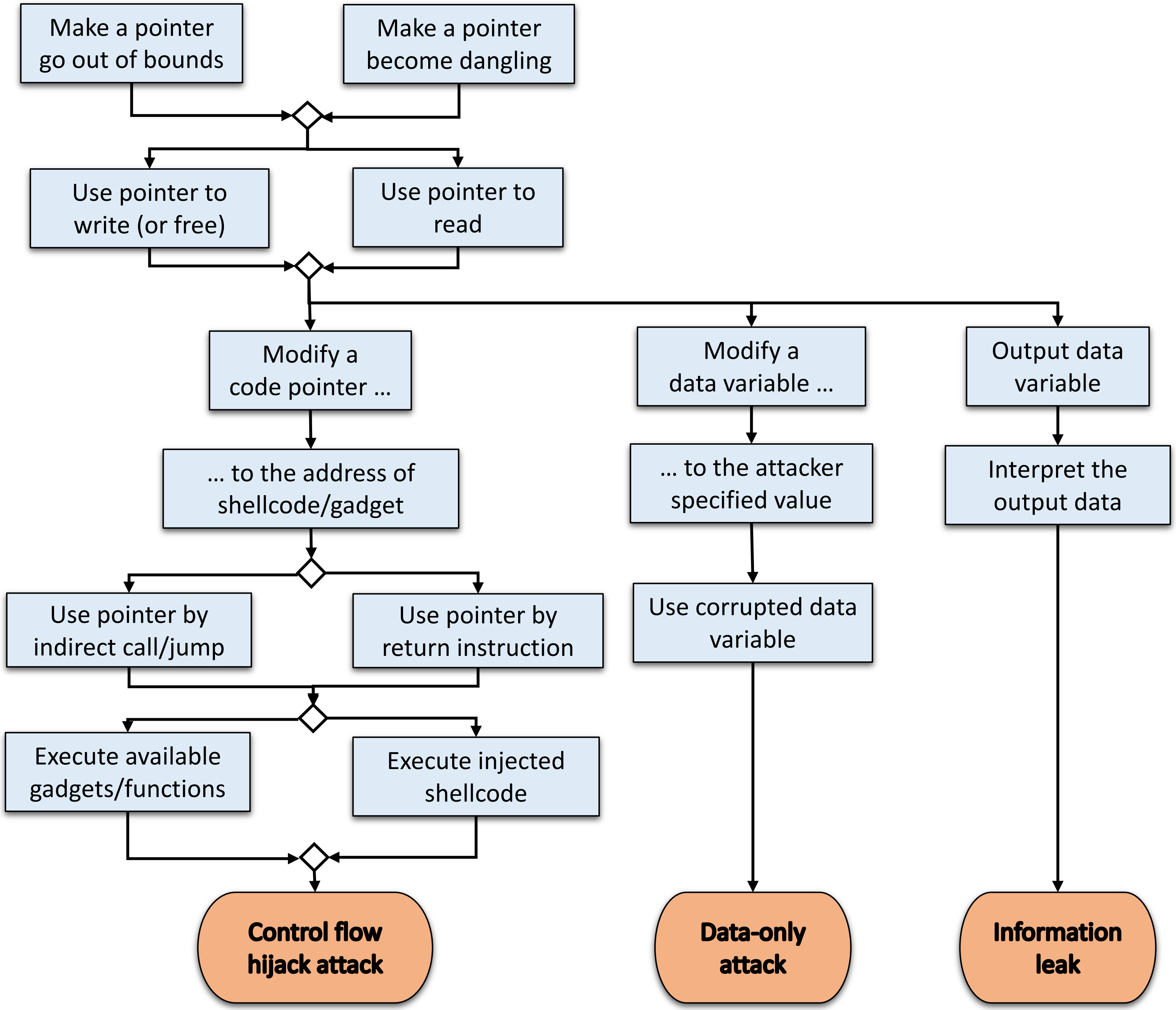


HeartBleed Vulnerability

- Publicly disclosed in April 2014
- Missing a bound check
- Bug in the OpenSSL cryptographic software library heartbeat extension

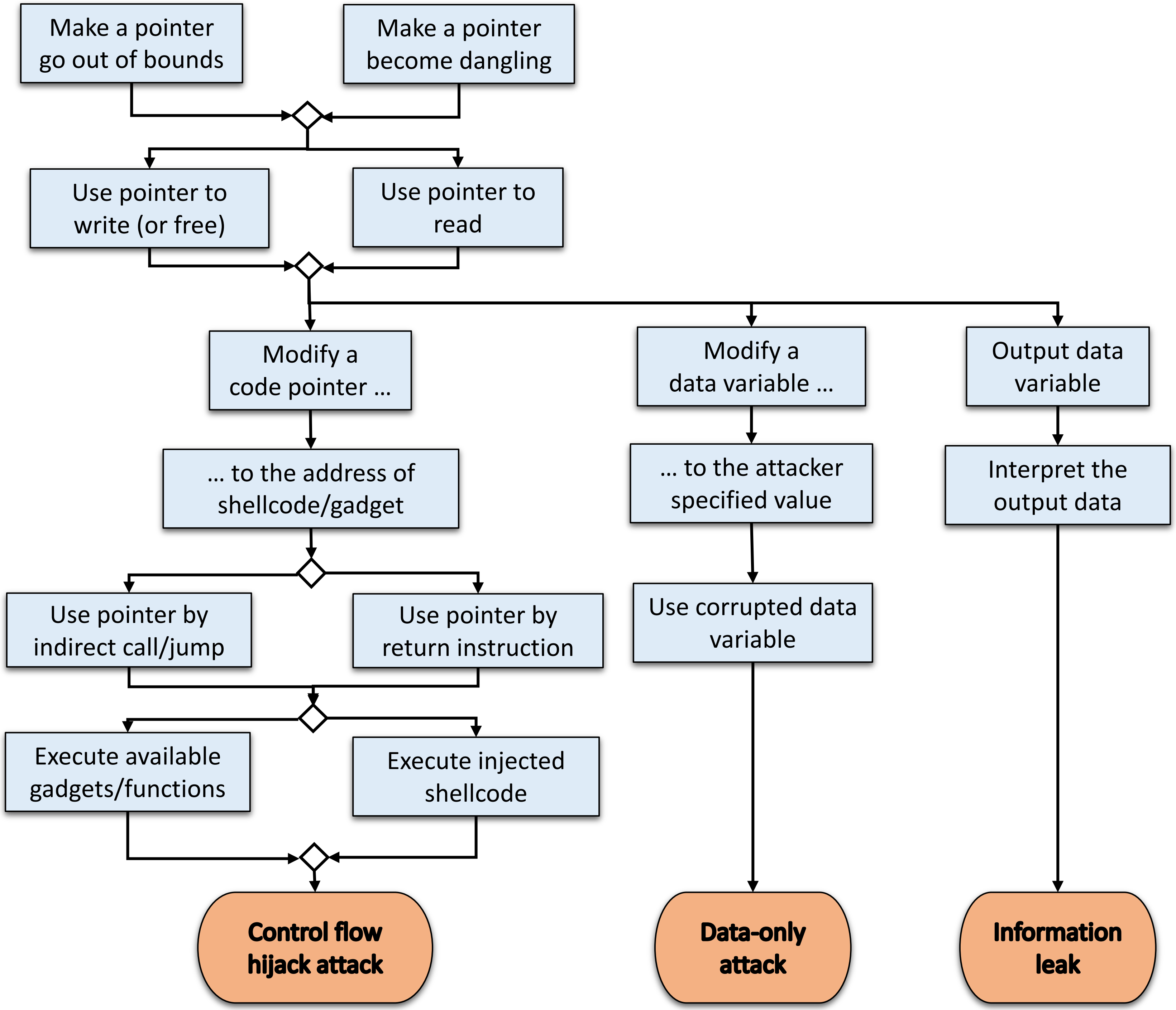


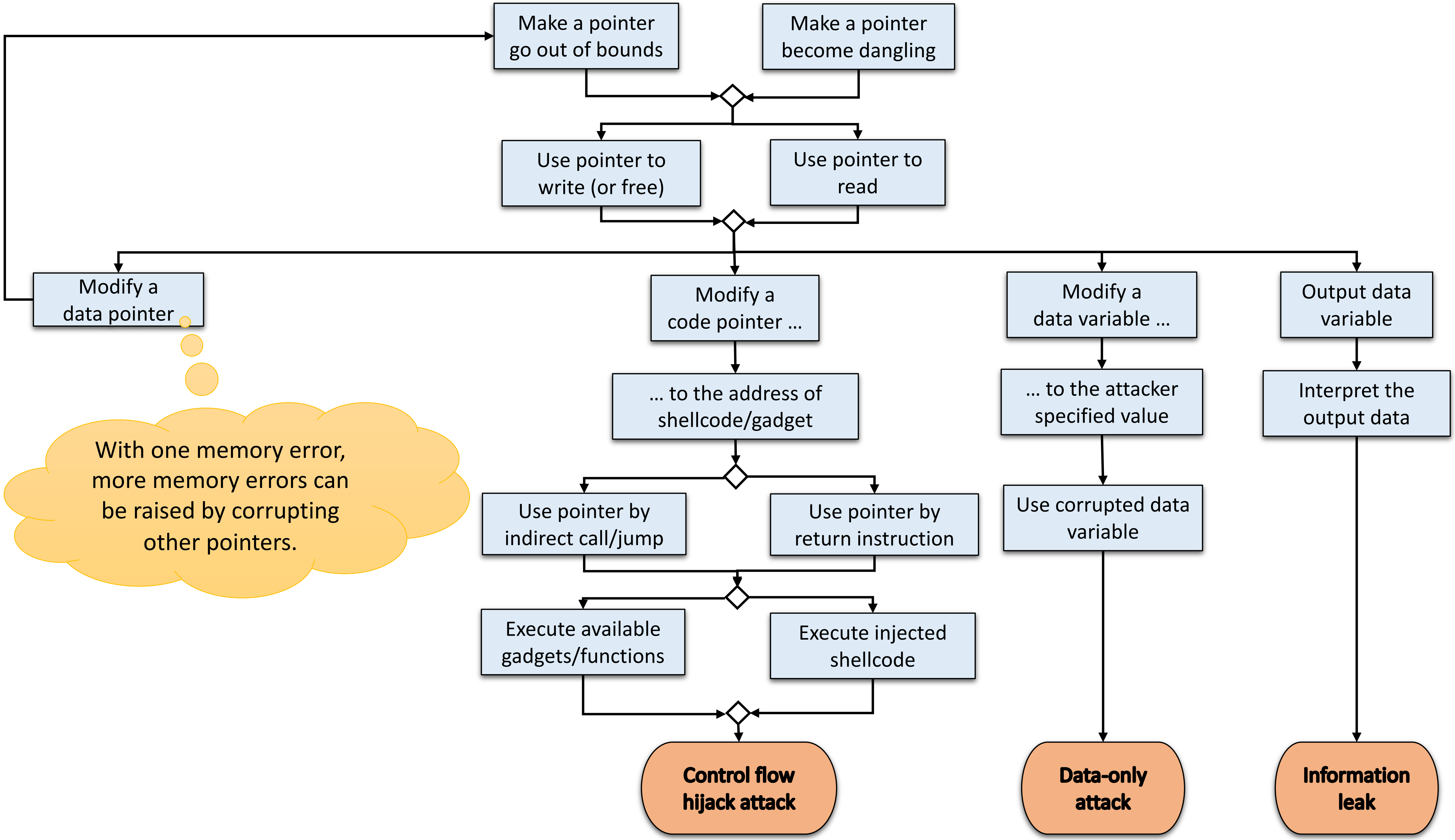
https://heartbleed.com/

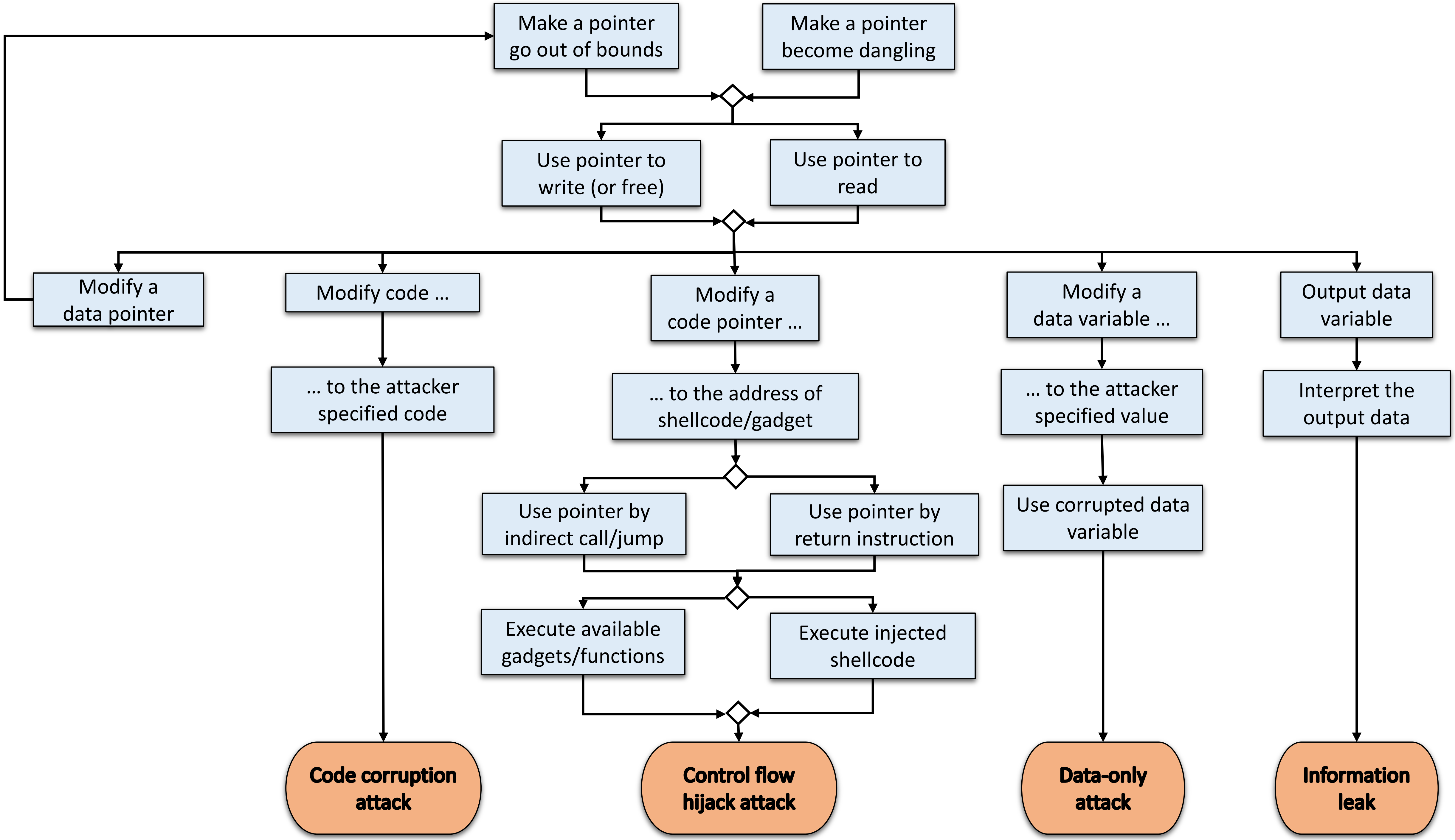

Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a code pointer …
… to the address of shellcode/gadget
Use pointer by indirect call/jump
Use pointer by return instruction
Execute available gadgets/functions
Execute injected shellcode
Control flow hijack attack
Modify a data variable …
… to the attacker specified value
Use corrupted data variable
Data-only attack
Output data variable
Interpret the output data
Information leak

Data-only attack example:

```
bool isAdmin = false;
...
if (isAdmin)
// do privileged operations
```



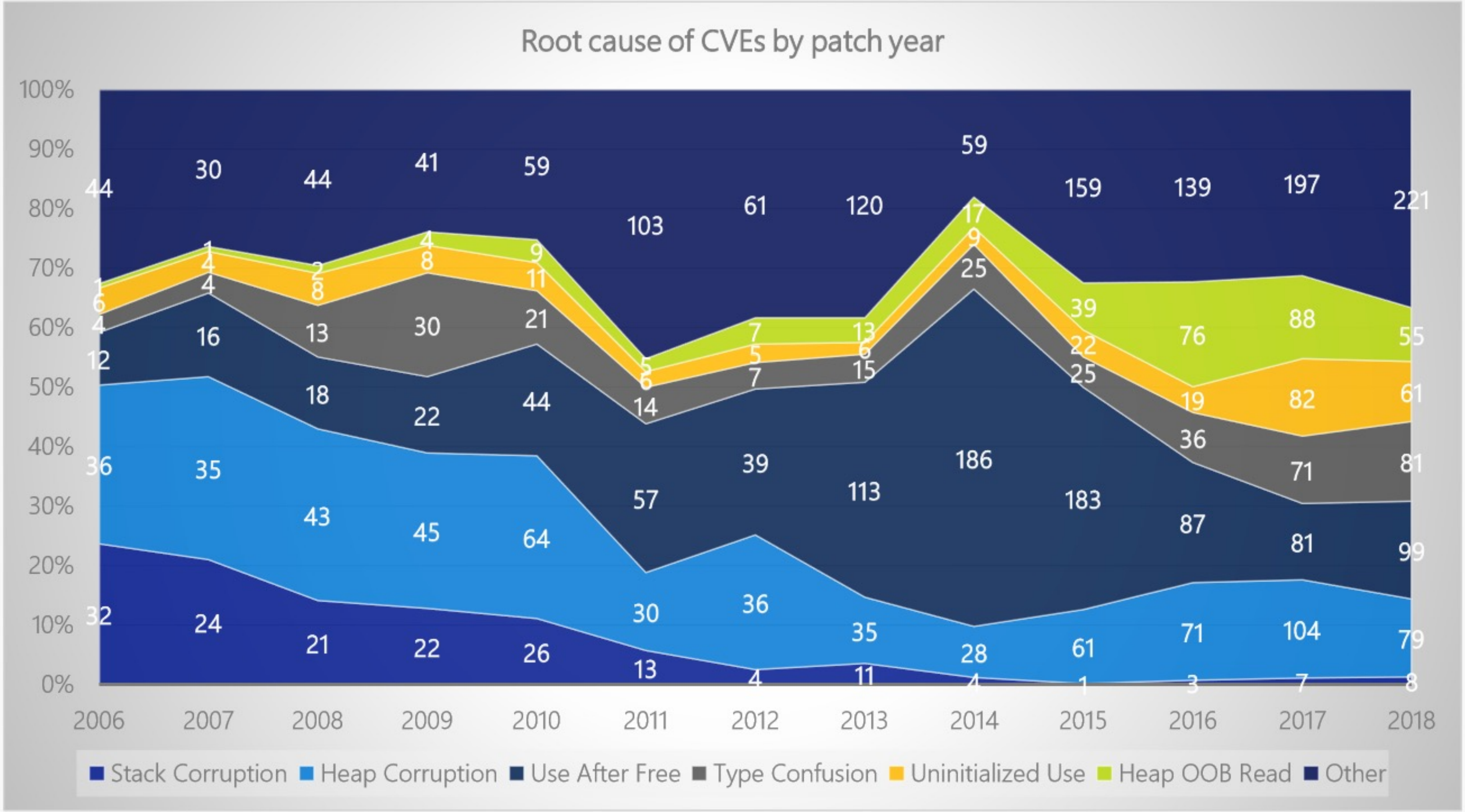

Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a data pointer
Modify a code pointer …
Modify a data variable …
Output data variable
… to the address of shellcode/gadget
… to the attacker specified value
Interpret the output data
With one memory error, more memory errors can be raised by corrupting other pointers.
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Execute available gadgets/functions
Execute injected shellcode
Control flow hijack attack
Data-only attack
Information leak


Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a data pointer
Modify code …
Modify a code pointer …
Modify a data variable …
Output data variable
… to the attacker specified code
… to the address of shellcode/gadget
… to the attacker specified value
Interpret the output data
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Execute available gadgets/functions
Execute injected shellcode
Code corruption attack
Control flow hijack attack
Data-only attack
Information leak

Drilling down into root causes

Trend reported by Microsoft

https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Why not High-level Language?

- Benefits:
  - Easier to program
  - Simpler concurrency with GC
  - Prevents classes of kernel bugs

- Downsides (performance):
  - Safety tax: Bounds, cast, null-pointer checks
  - Garbage collection: CPU and memory overhead, pause time
  - Feasibility?

The benefits and costs of writing a POSIX kernel in a high-level language; Cutler et al (OSDI'18)

BISCUIT: new x86-64 Go kernel

No fundamental challenges due to HLL

But many implementation puzzles

- Interrupts
- Kernel threads are lightweight
- Runtime on bare-metal

  ...

Surprising puzzle: heap exhaustion

Why not Rust (no GC)?

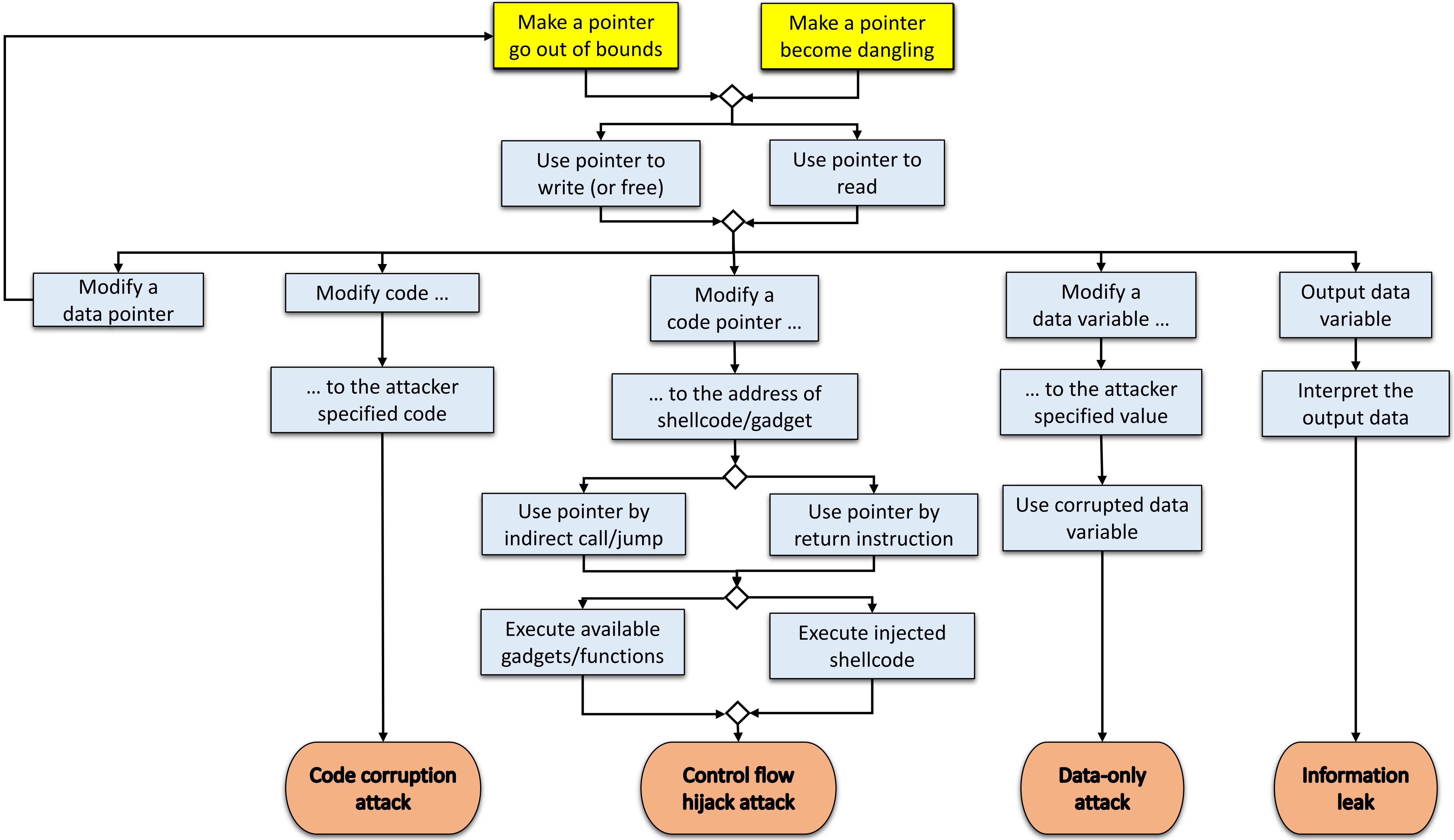
Rust compiler analyzes the program to partially automate freeing of memory.
This approach can make sharing data among multiple threads or closures awkward

HW Support for Memory Safety

- Spatial safety (bound information)
- Temporal safety (allocation/de-allocation information)

- Low-level reference monitor
  - SW approach: add checks → performance overhead (e.g., SoftBound)
  - Execution time: Extra instructions to perform the check
  - Memory: Maintain extra meta data (in shadow memory)

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C; Nagarakatte et al; PLDI'09



Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a data pointer
Modify code …
Modify a code pointer …
Modify a data variable …
Output data variable
… to the attacker specified code
… to the address of shellcode/gadget
… to the attacker specified value
Interpret the output data
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Execute available gadgets/functions
Execute injected shellcode
Code corruption attack
Control flow hijack attack
Data-only attack
Information leak

Intel MPX (Memory Protection Extension)

4 bound registers (bnd0-3)

- Bndmk: create base and bound metadata
- Bndldx/bndstx: load/store metadata from/to bound tables
- Bndcl/bndcu: check pointer with lower and upper bounds

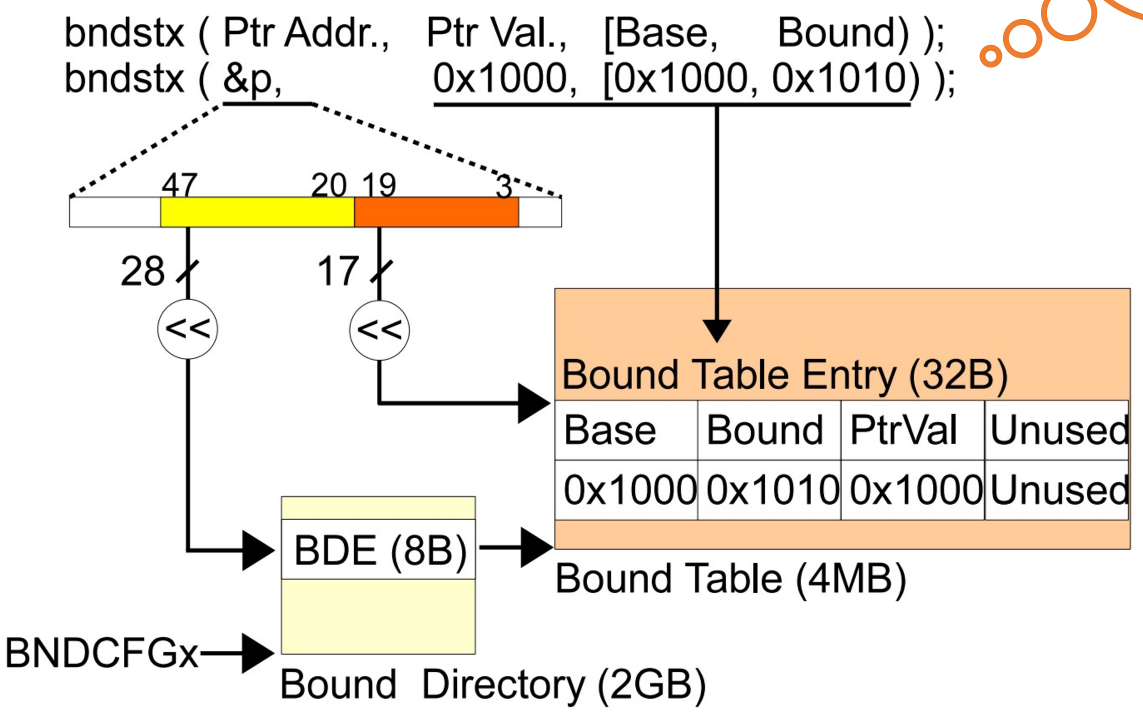
Any problem?

Original Program

```
p=malloc(16);
… // p = p + 4;
*p = 'a';
```

Instrumented Program

```
p=malloc(16);
bnd0 = bndmk(p,16);
bndstx (&p,p,bnd0);
… // p = p + 4;
bnd1 = bndldx(&p,p);
bndcl (&p, bnd1);
bndcu (&p, bnd1);
*p = 'a';
```



Analysis of Intel MPX

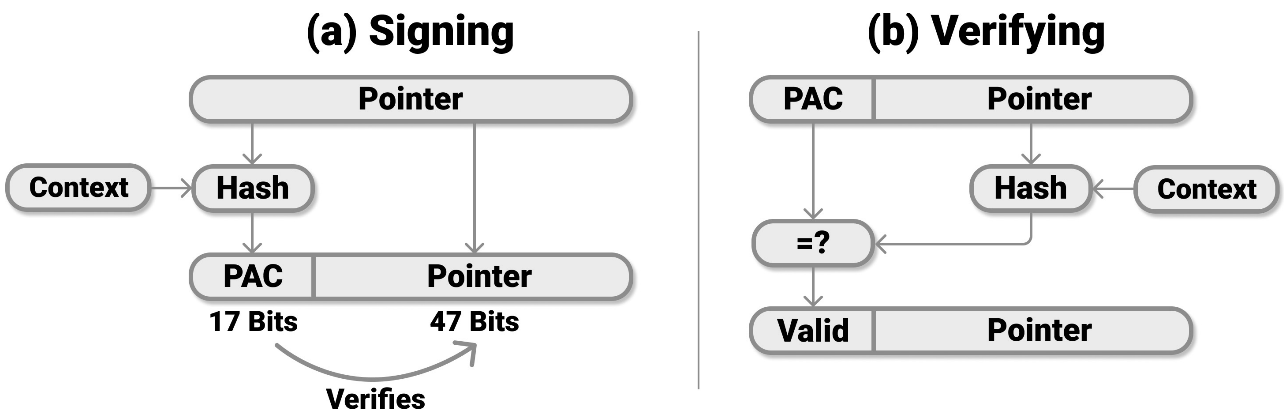
Intel MPX is **impractical** for fine-grained memory safety

- High overheads
  - Check is sequential
  - loading/storing bounds registers involves two-level address translation
- Does not provide temporal safety
- Does not support multithreading transparently
- Meltdown? Bound Range Exceeded (#BR) hardware exception

Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack; OLEKSENKO et al; SIGMETRICS'18

ARM PA (Pointer Authentication)

- Widely used in Apple processors
- Motivation:
  - 64-bit pointer, but 48-bit virtual address space
  - Unused high bits
- Hash:
  - A tweakable message authentication code (MAC)
  - ARM calls it **PAC (pointer authentication code)**
- Context:
  - secret key
  - salt (could be the stack pointer)

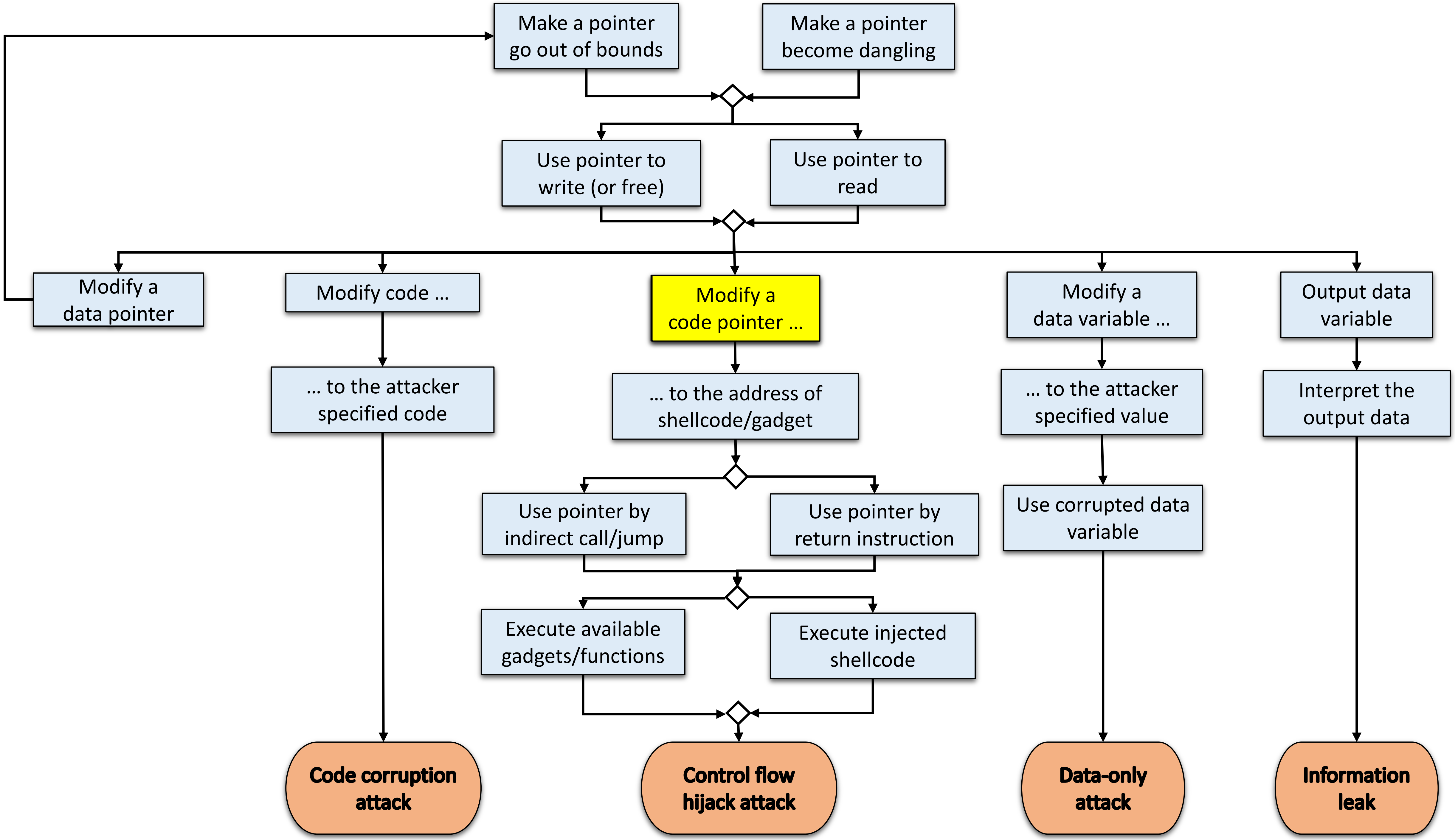


Before function call

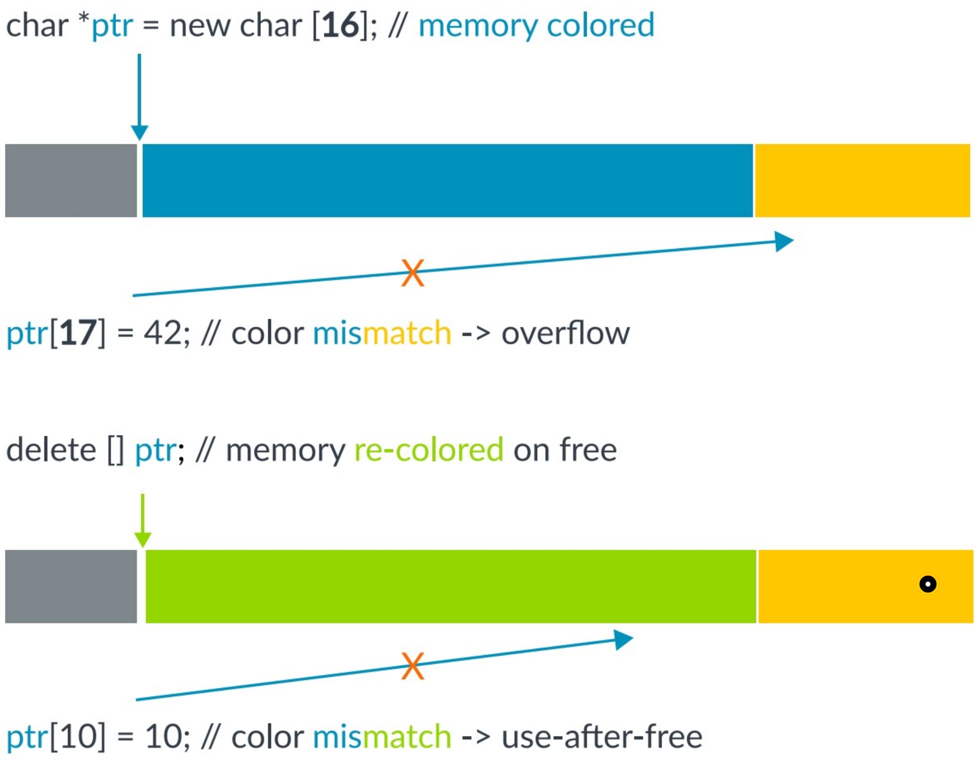
```
pacia lr, sp
sub   sp, sp, #0x40
str   lr, [sp, #0x30]
...
```

Before function return

```
ldr   lr, [sp, #0x30]
add   sp, sp, #0x40
autia lr, sp
ret
```

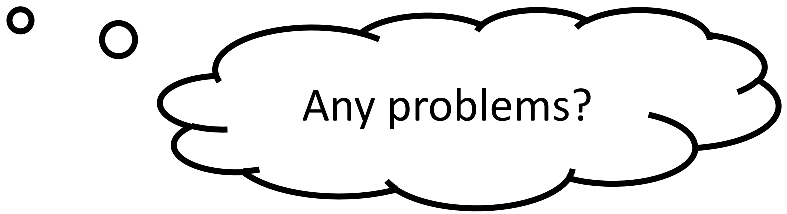

Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a data pointer
Modify code …
Modify a code pointer …
Modify a data variable …
Output data variable
… to the attacker specified code
… to the address of shellcode/gadget
… to the attacker specified value
Interpret the output data
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Execute available gadgets/functions
Execute injected shellcode
Code corruption attack
Control flow hijack attack
Data-only attack
Information leak

ARM MTE/Intel MPK

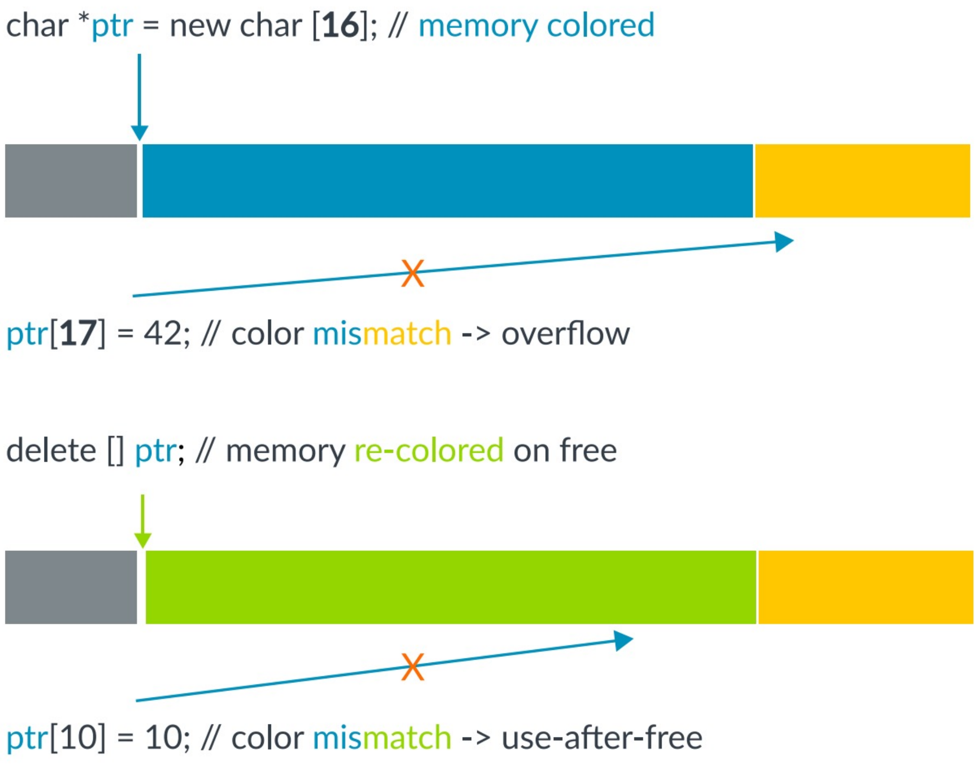


Armv8.5-A Memory Tagging Extension White paper
https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

- 2019, Google announced that it is adopting Arm’s MTE in Android
- Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory
- Where to store tags?
  - Pointer tag is stored in top byte of the pointer
  - Physical memory tag is stored in hardware

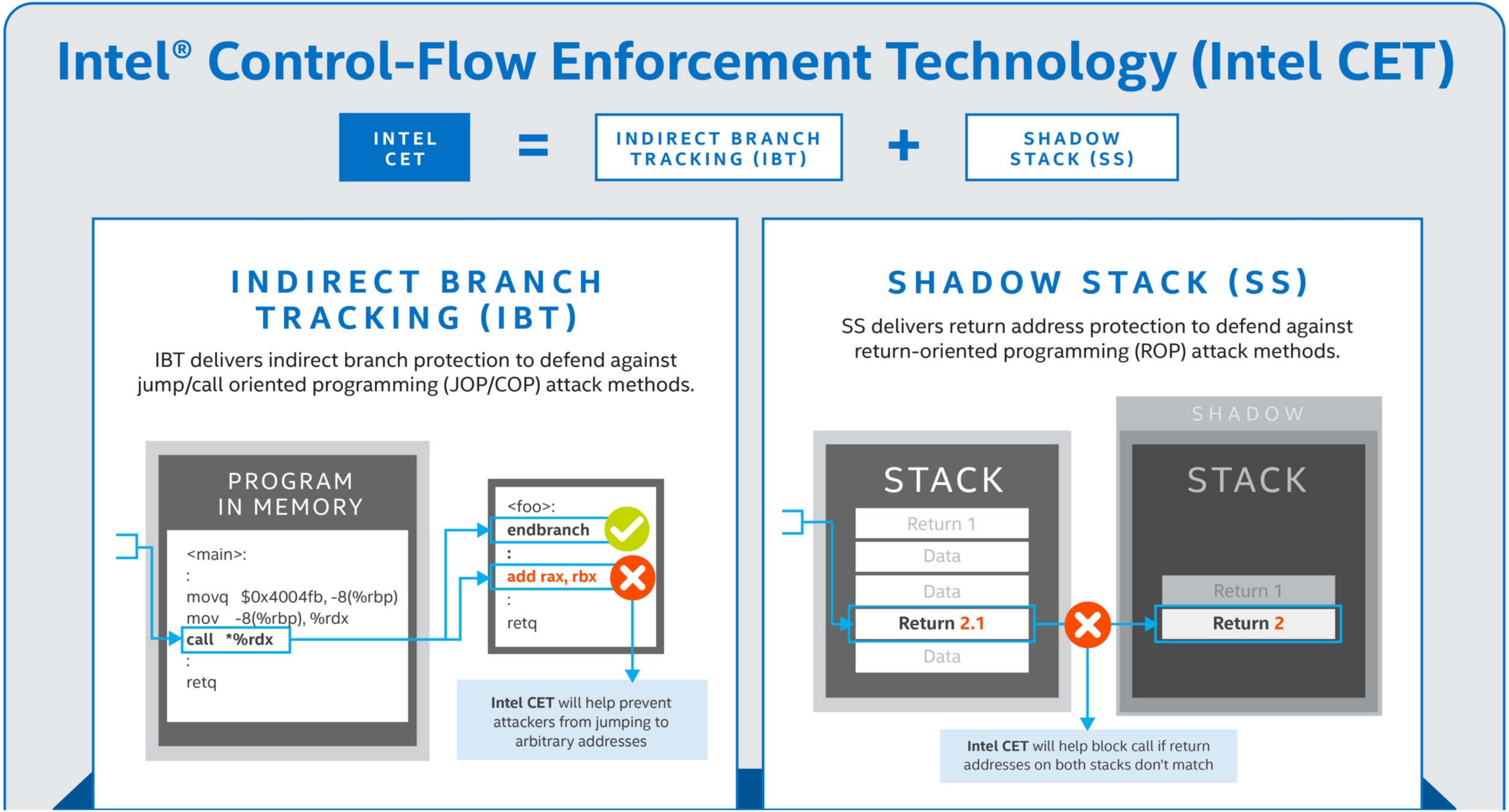


ARM MTE/Intel MPK



Armv8.5-A Memory Tagging Extension White paper
https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

- 2019, Google announced that it is adopting Arm's MTE in Android
- Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory
- Where to store tags?
  - Pointer tag is stored in top byte of the pointer
  - Physical memory tag is stored in hardware
- Limited tag bits
  - Cannot ensure two allocations have different colors
  - But can ensure that the tags of sequential allocations are always different



Intel® Control-Flow Enforcement Technology (Intel CET)
INTEL CET
=
INDIRECT BRANCH TRACKING (IBT)
+
SHADOW STACK (SS)
INDIRECT BRANCH TRACKING (IBT)
IBT delivers indirect branch protection to defend against jump/call oriented programming (JOP/COP) attack methods.
PROGRAM IN MEMORY
<main>:
:
movq $0x4004fb, -8(%rbp)
mov -8(%rbp), %rdx
call *%rdx
:
retq
<foo>:
endbranch
:
add rax, rbx
:
retq
Intel CET will help prevent attackers from jumping to arbitrary addresses
SHADOW STACK (SS)
SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.
STACK
Return 1
Data
Data
Return 2.1
Data
SHADOW
STACK
Return 1
Return 2
Intel CET will help block call if return addresses on both stacks don't match

Control Flow Integrity (CFI)

```
p = &f                  f() {
jmp p                     …
                        }
```

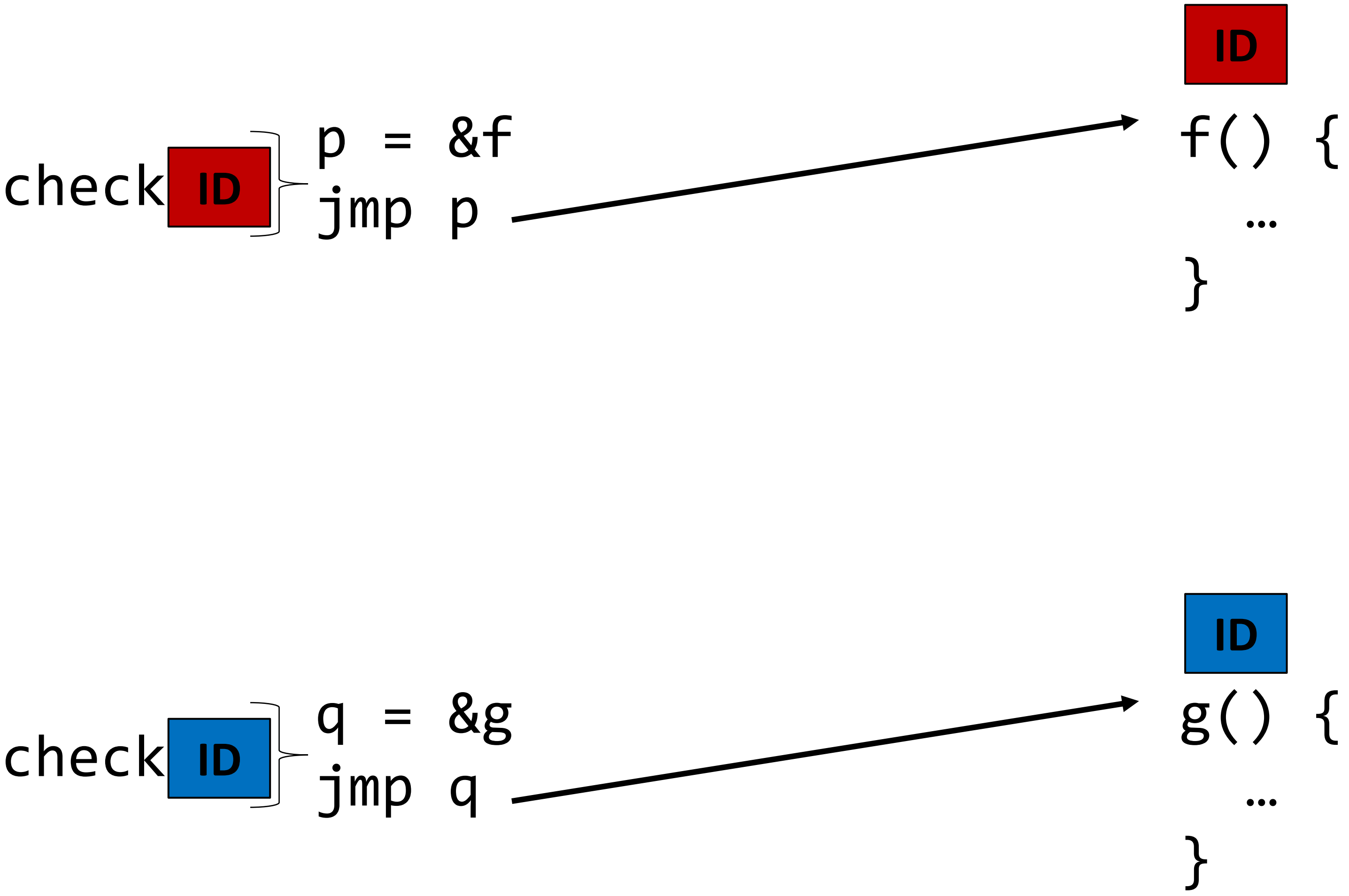
```
q = &g                  g() {
jmp q                     …
                        }
```

Control Flow Integrity (CFI)

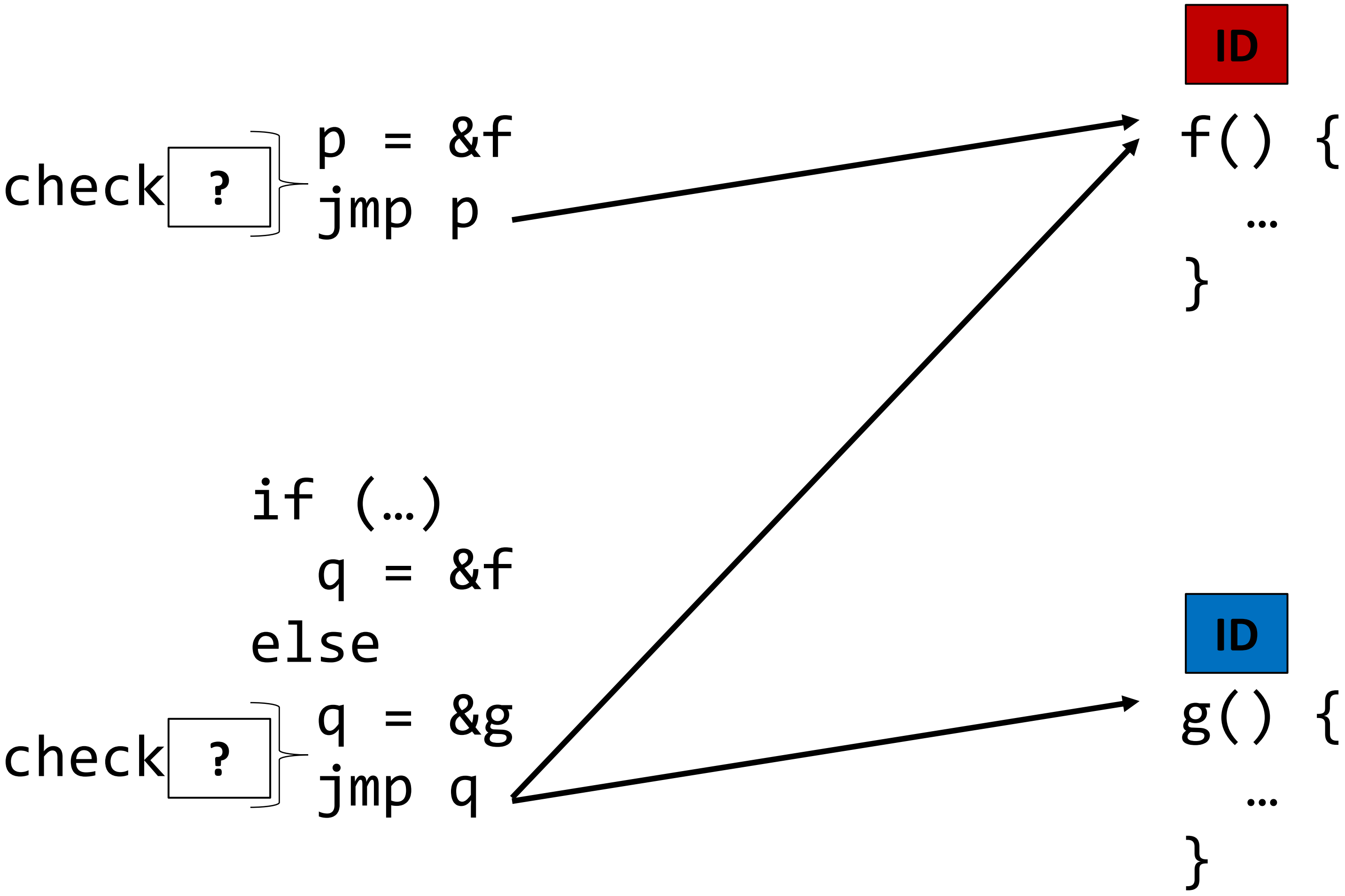
```
p = &f          ID
jmp p           f() {
                  …
                }
```

```
q = &g          ID
jmp q           g() {
                  …
                }
```

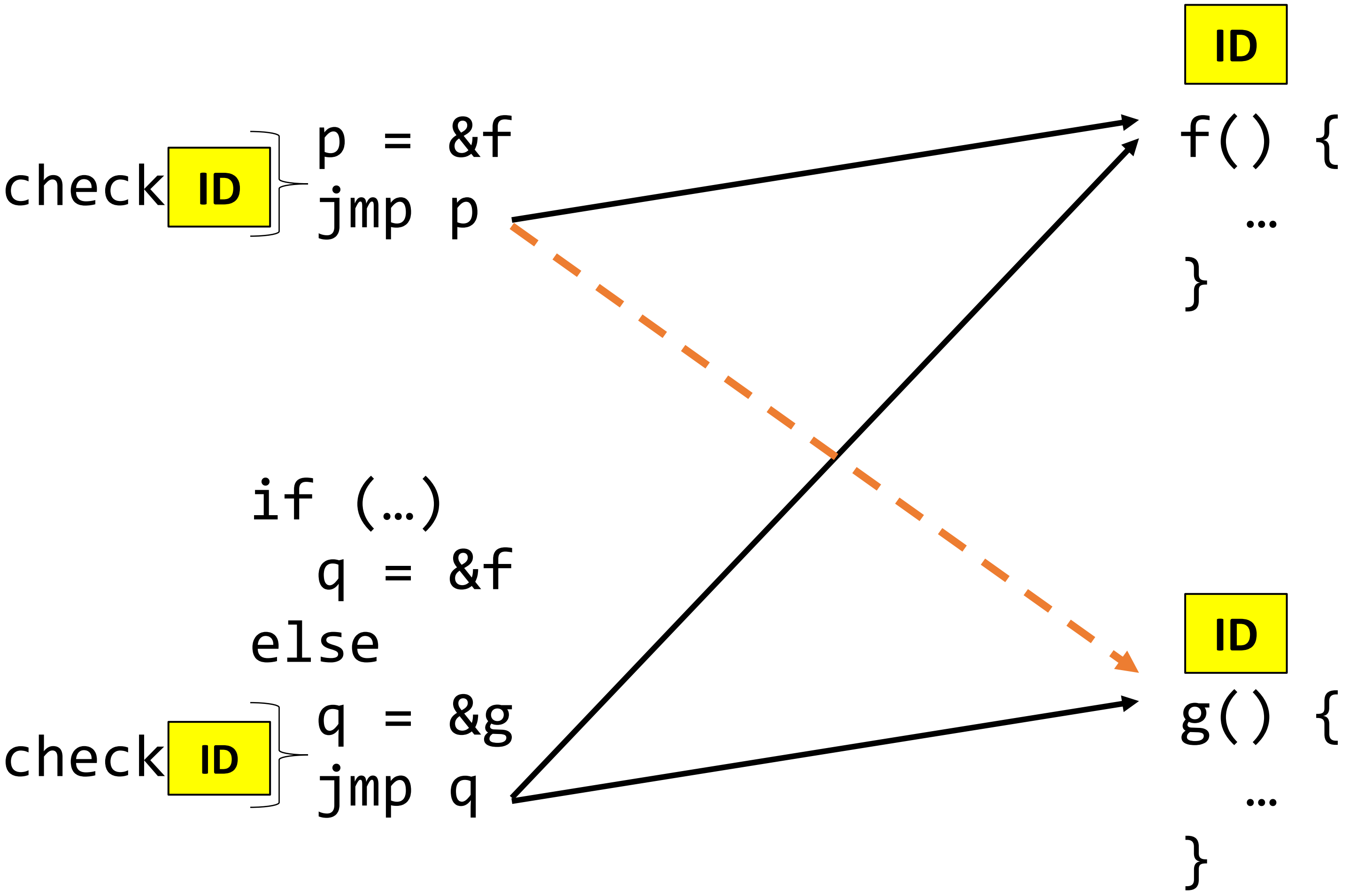
Control Flow Integrity (CFI)



Control Flow Integrity (CFI)



Over-approximation Problem



Summary

- Memory corruption problems: An eternal war
- Attack variations and mitigations
- Recent hardware support