Code: analysis, bugs, and security

supported by Bitdefender

Memory corruption errors and defenses

Marius Minea

marius@cs.upt.ro

15 November 2017

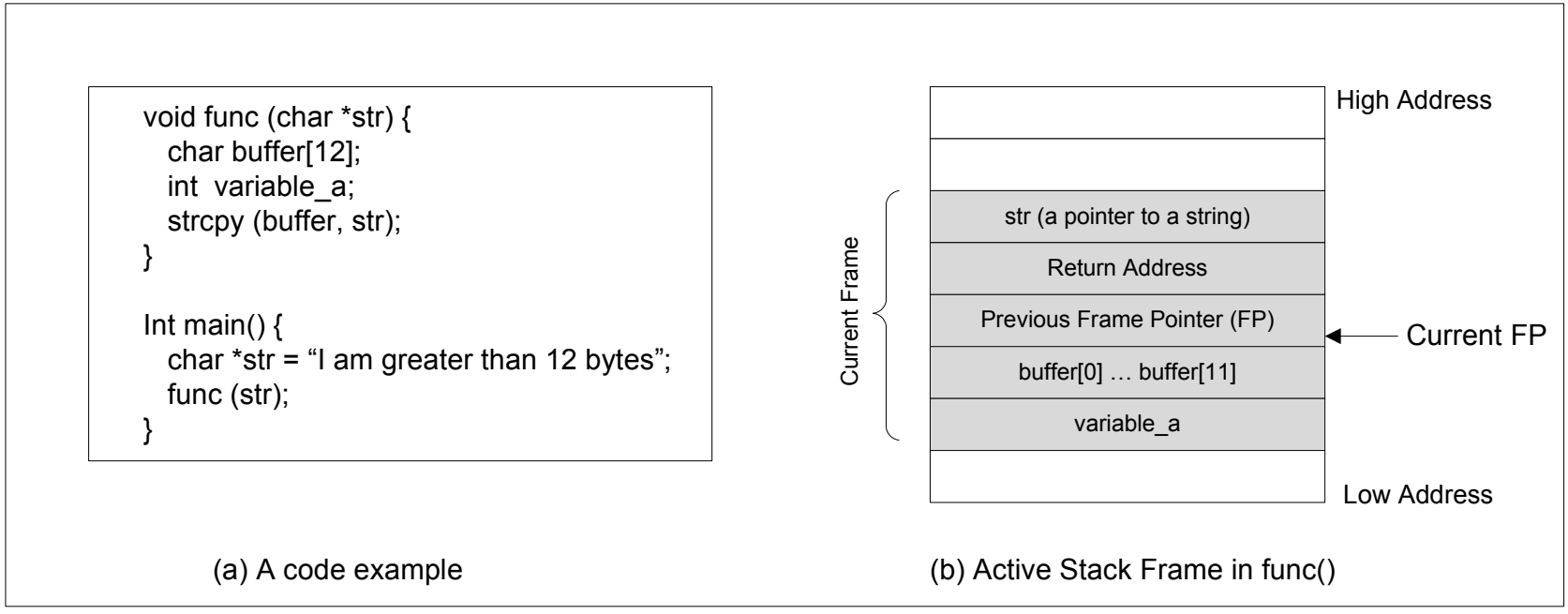
Memory corruption: still a problem

String of vulnerabilities over past 30 years
- still ongoing
- new protection solutions every year

Reason: lack of memory and type safety in low-level languages
- main culprits: C and C++
- (unsafe inputs, pointer arithmetic, unsafe casts)

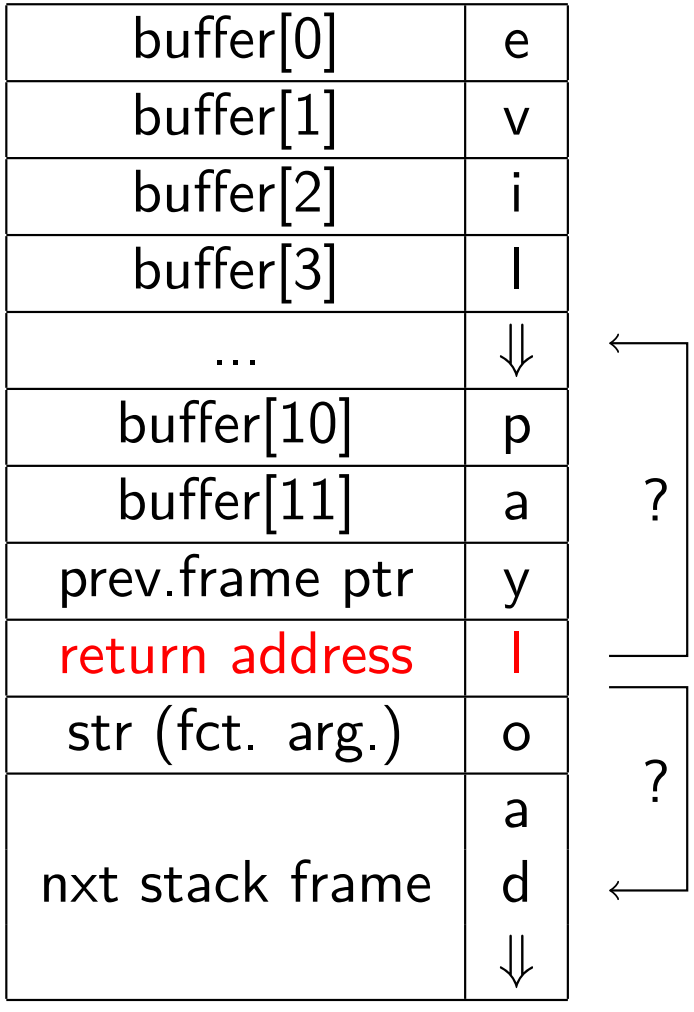
A classic paper:
Aleph One, Smashing the stack for fun and profit
Phrack magazine 7(49), 1997

Classic buffer overflow



http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/Buffer_Overflow.pdf

What happens on overflow



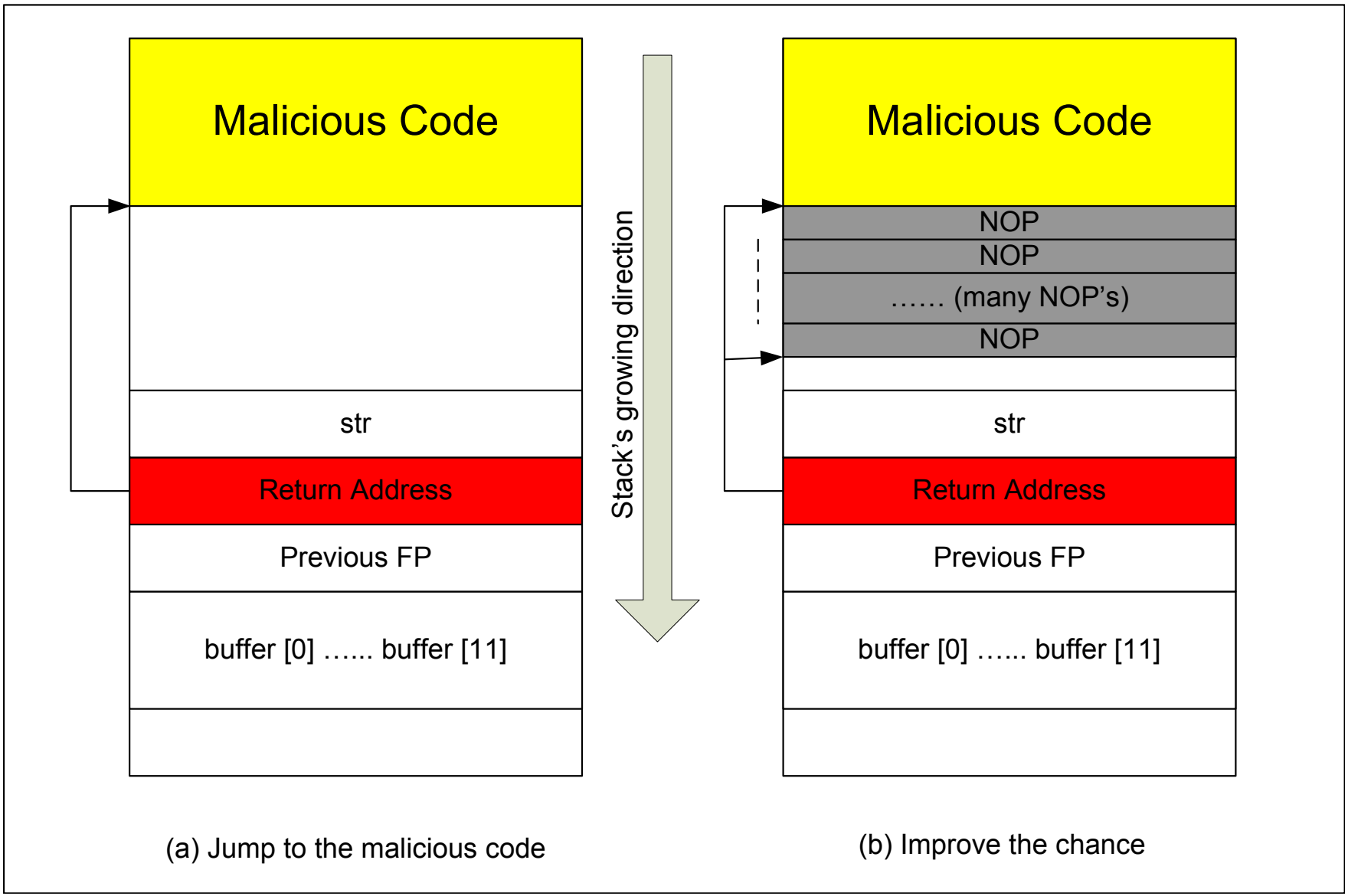
return address slot overwritten

on function return, execution jumps wherever that points to

For *successful* exploit, must know:
1) position of return address slot *relative* to buffer start:
i.e., buffer size and stack layout (calling convention)

2) *absolute* memory address of buffer (to fill in proper payload address)

Exploit: getting the address right



http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/Buffer_Overflow.pdf

Steps to successful exploit

Let's revisit exploit assumptions:

- can determine *where* to inject payload (*address*)
- can *overwrite* return address
- tampering is *not detected*
- can *execute* payload code

Variants:
a) overwrite base pointer rather than return address
   returns into attacker-crafted stack frame $\Rightarrow$ then into exploit
b) overwrite C++ exception handling pointers (stored on stack), and cause exception

How to protect? (1)

Option 1: detect change

check *before* function return if RET address altered

Two basic ideas:

Check return address itself ⇒ need copy of correct value

Check bytes next to (before) ret address ⇒ canaries

terminator canary: 0, CR, LF, EOF

random canary (created at process startup time)

don't know ⇒ can't put back)

random XOR canary (XOR'ed with protected/control data – if it changes, canary will be wrong)

Checks inserted by compiler where needed

How to protect? (2)

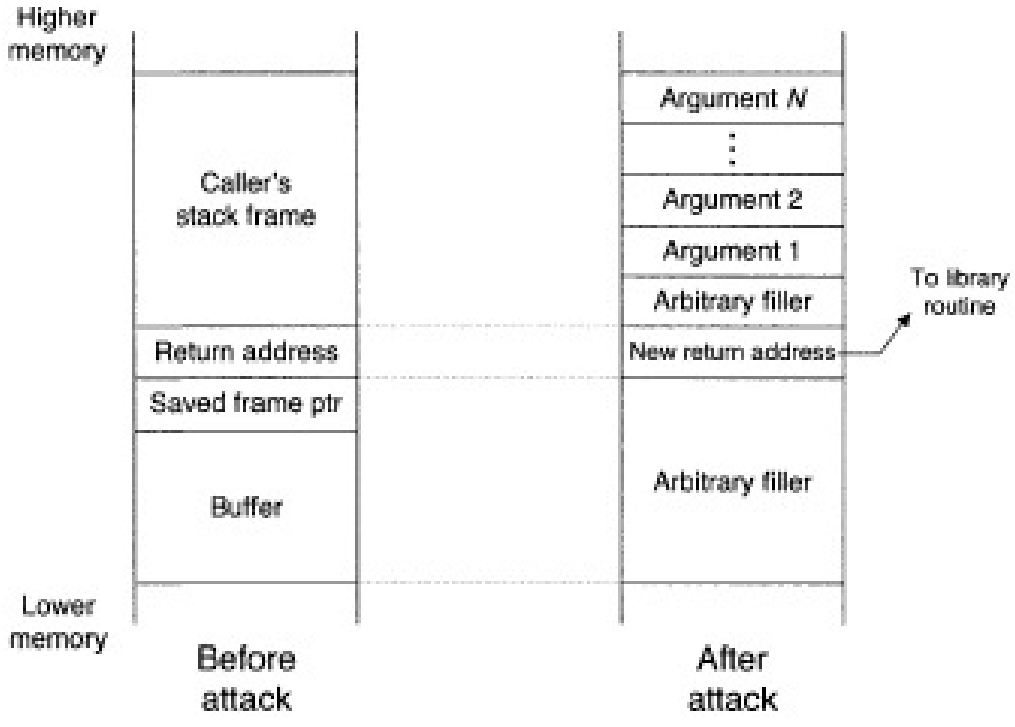
Option 2: hamper execution

Attacker must execute injected code:
- Non-executable stack / write XOR execute (operating system support)

Attacker must know *what address* to jump to:
- Address Space Layout Randomization
  - good, but ineffective against brute force

Return-to-libc attack

Typical attack is to call `exec` or some other library function
$\Rightarrow$ instead of *executing code* (`call exec`),
put address (and parameters) of libc function on stack,
in place of normal ret address



`http://geekscomputer.blogspot.ro/2008/12/buffer-overflows.html`

Can chain calls – put multiple library addresses on stack

Attack: Overwriting a pointer

Function pointers (denote code)
- pointers from `longjmp`
- pointers to user functions
- pointers to library functions (PLT: procedure linkage table)
- pointers to virtual method (C++ vtable)

or usual pointers to data

Attacks has two steps:
- a buffer overflow overwrites a pointer (to desired address)
- in later code, this is used to overwrite critical area
  - ret address, PLT, etc.

Return-oriented programming (ROP)

(H. Shacham, ACM CCS 2007)
Generalizes return-to-libc by chaining returns

Stack overwritten so returns go from a piece libc code to another
control flow given by stack contents

Pieces ("gadgets") chosen with useful instructions

`libc` contains enough gadgets for arbitrary programs
(Turing-complete)

ROP "compilers" can produce arbitrary code

Return-oriented programming

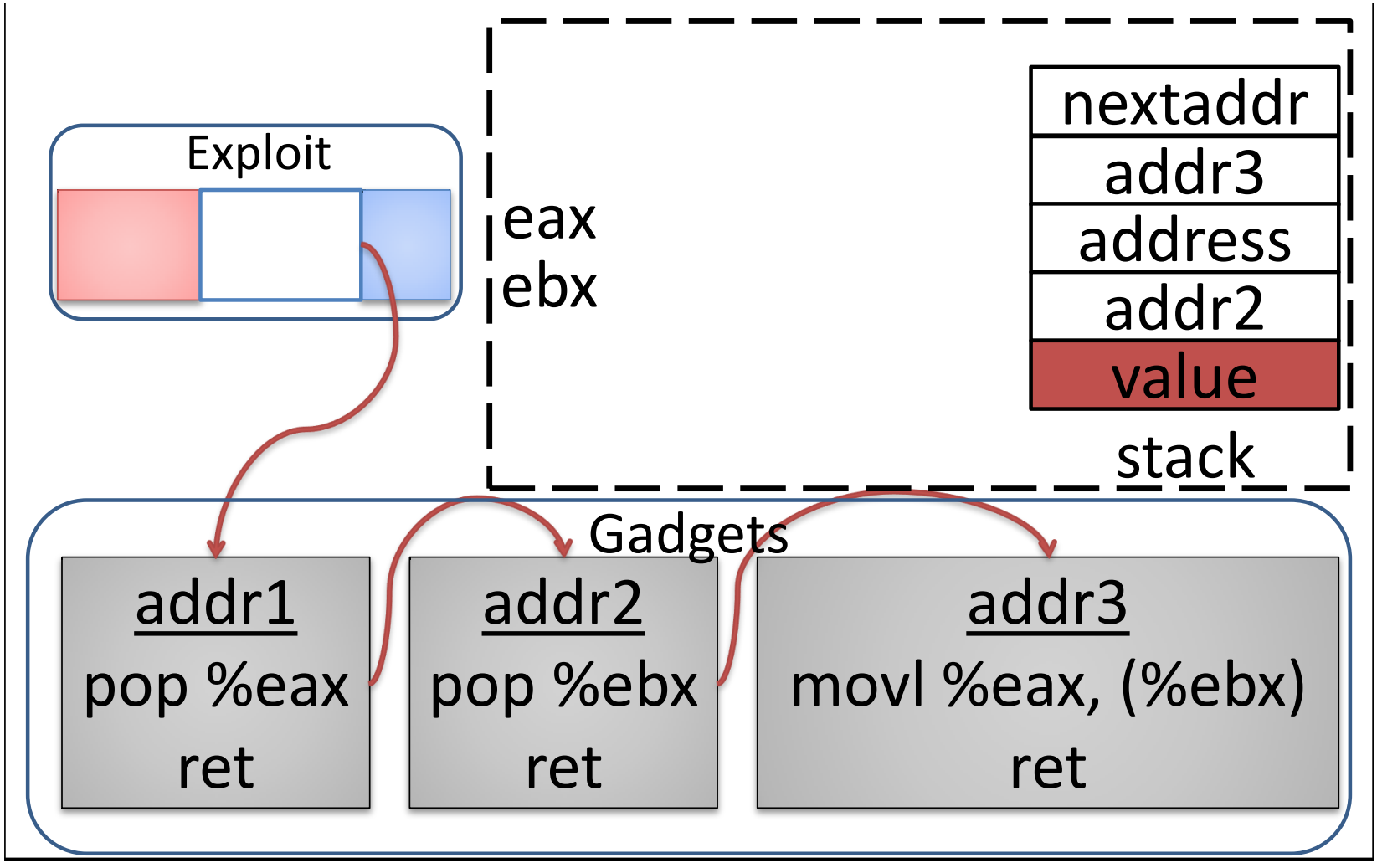


figure: Schwartz, Avgerinos, Brumley, USENIX 2011

Taxonomy of attacks

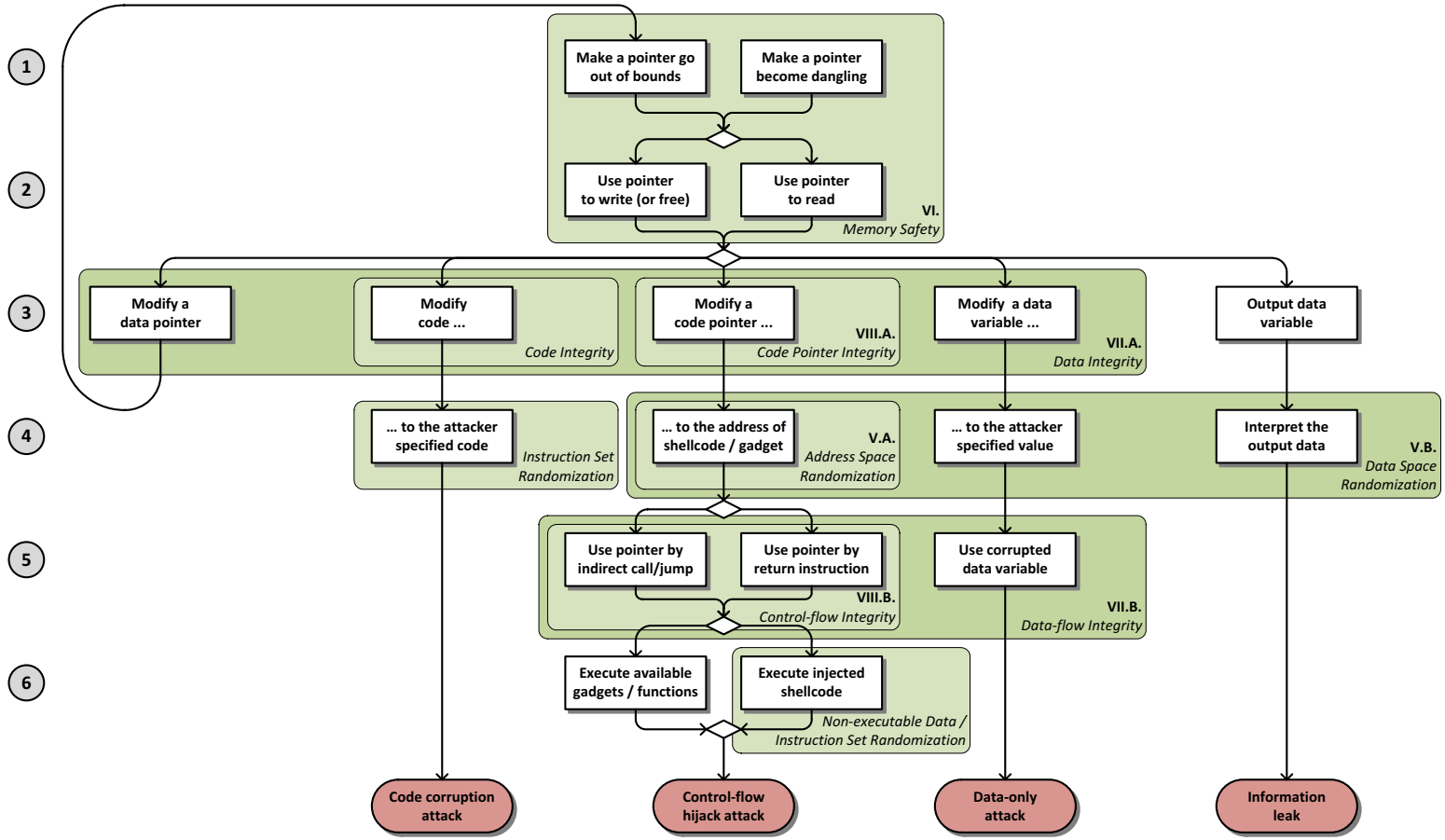


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Szekeres et al., SoK: Eternal War in Memory, S&P'13

Control Flow Integrity

(Abadi, Budiu, Erlingsson, Ligatti, ACM CCS 2005)

Fundamental idea: exploits deviate from normal program execution
(regardless whether in injected code or libc gadgets)

Capture legal control flow graph (at compile time)
Ensure execution never leaves the CFG
true for direct calls/jumps (if code not modifiable)
checked at runtime for indirect calls/jumps (add instrumentation before these instructions)

Tradeoff between precision/safety and performance overhead

Recently: Code Pointer Integrity: protect only code pointers, stored in a separate memory area.

Protecting data

One main problem is unrestricted pointer use
  any pointer arithmetic / pointer creation must be checked

Techniques:
encrypt pointers (in theory, they are abstract values, even in C)
  but much nonportable code relies on pointer representation
fat pointer:
  pointer is not just address word but has info on base + block size
  incompatible with standard libraries (must be recompiled)

store metadata separately from pointers
  use (hash)map from pointers to metadata
  still must update metadata on library call

store address ranges of live objects in a global table
  [Jones & Kelly], first implementation in GCC, high overhead

An escalating race: where to?

Many solutions, not all adopted (cf. Szekeres et al. '13)

- *performance overhead* outweighs potential benefit/safety
- not *compatible* with legacy programs
- lack of *robustness* / incomplete
  - attacks often discovered soon after protection proposed
- toolchain *dependence* blocks adoption

Trends:

use higher-level languages with better safety guarantees

- but: their implementation may still have low-level bugs

implement stronger safety policies

- (control flow integrity, data integrity)