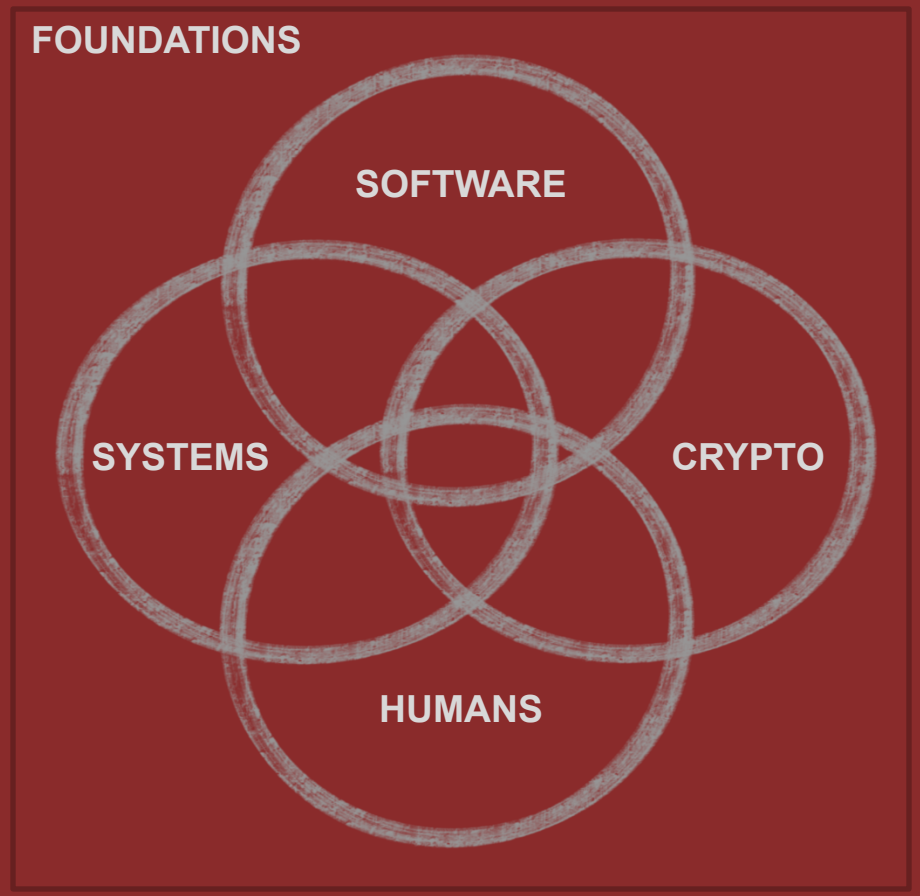
Διάλεξη #7 - Mitigations



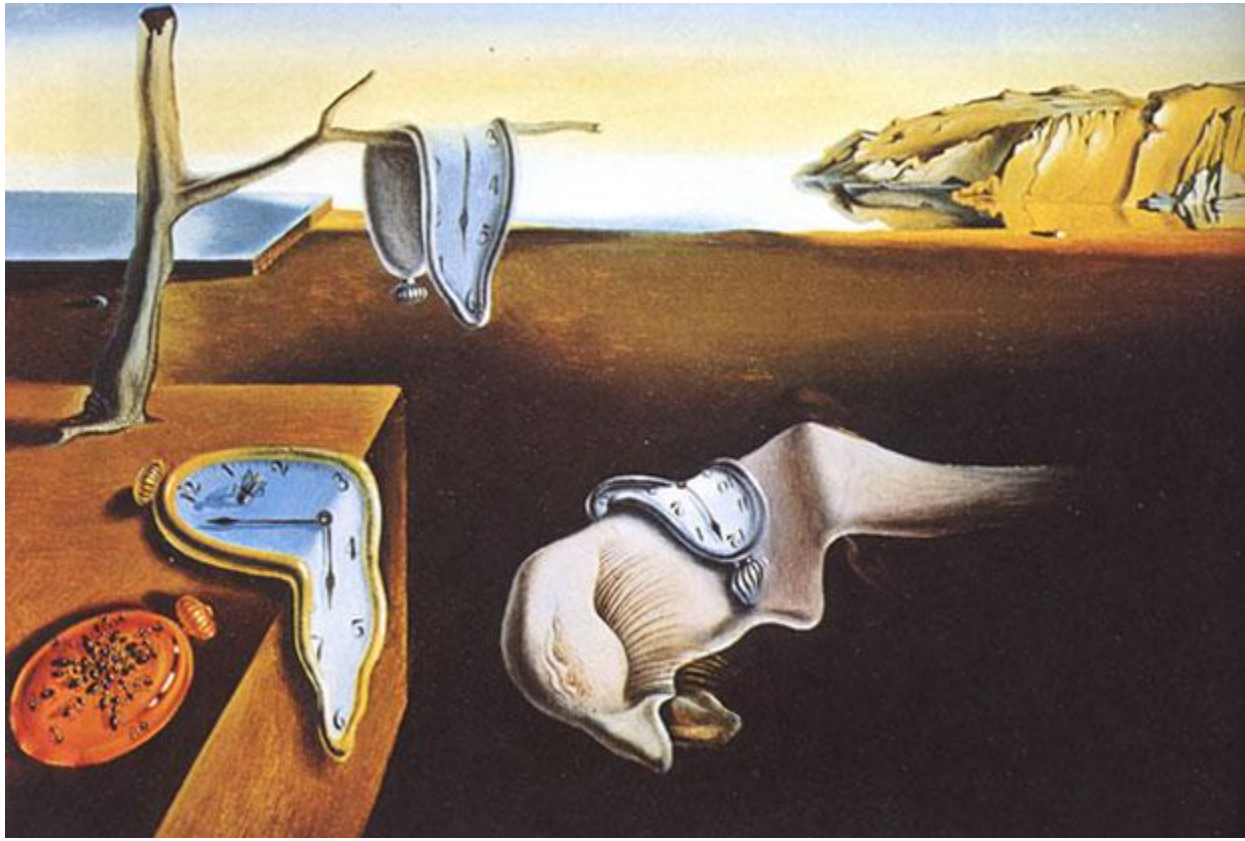
Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class

Ανακοινώσεις / Διευκρινίσεις

- N/A

Την Προηγούμενη Φορά

1. Application Security
2. Format String Attacks and review



Σήμερα

- Control Flow Hijack Mitigations

Control Flow Hijack Defenses / Mitigations

Defenses

1. Canaries
2. DEP (Data Execution Prevention) / NX (No Execute)
3. ASLR (Address Space Layout Randomization)

Defenses we will see today focus on **preventing** control hijacks (Prevention)

Why does this program dump core even though we don't overwrite the return address?

```
int is_good() {
    char * magic = "8675309";
    char buf[32];
    fread(buf, 128, 1, stdin); // BOFs are cool
    if (strncmp(magic, buf, strlen(magic)) == 0) {
        return 1;
    }
    return 0;
}
```

Can we control the instruction pointer in such a program? Yes / No? Why?

Canary / Stack Cookies

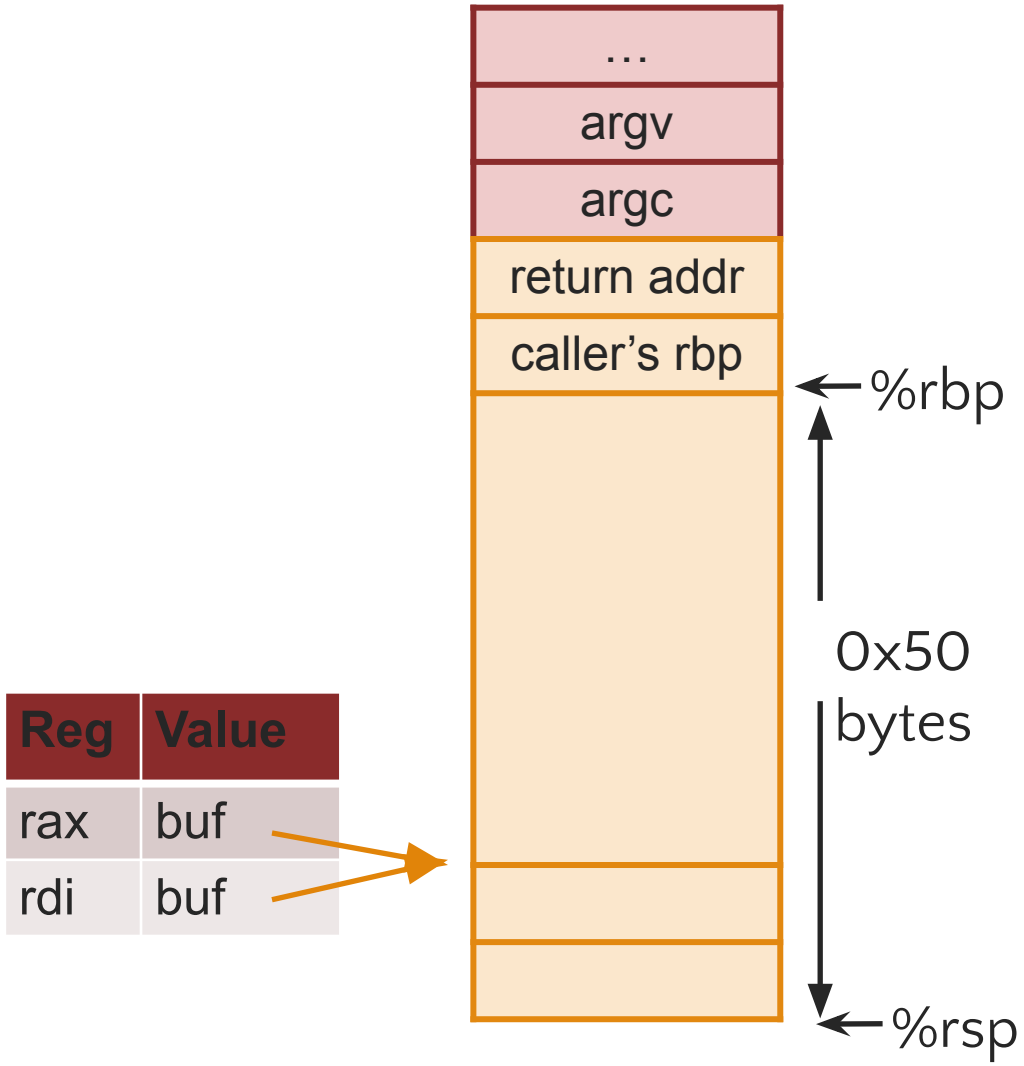
What Is a “Canary”?

Wikipedia: “the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system.”

lecture

Reminder; Buffer Overflow

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
```

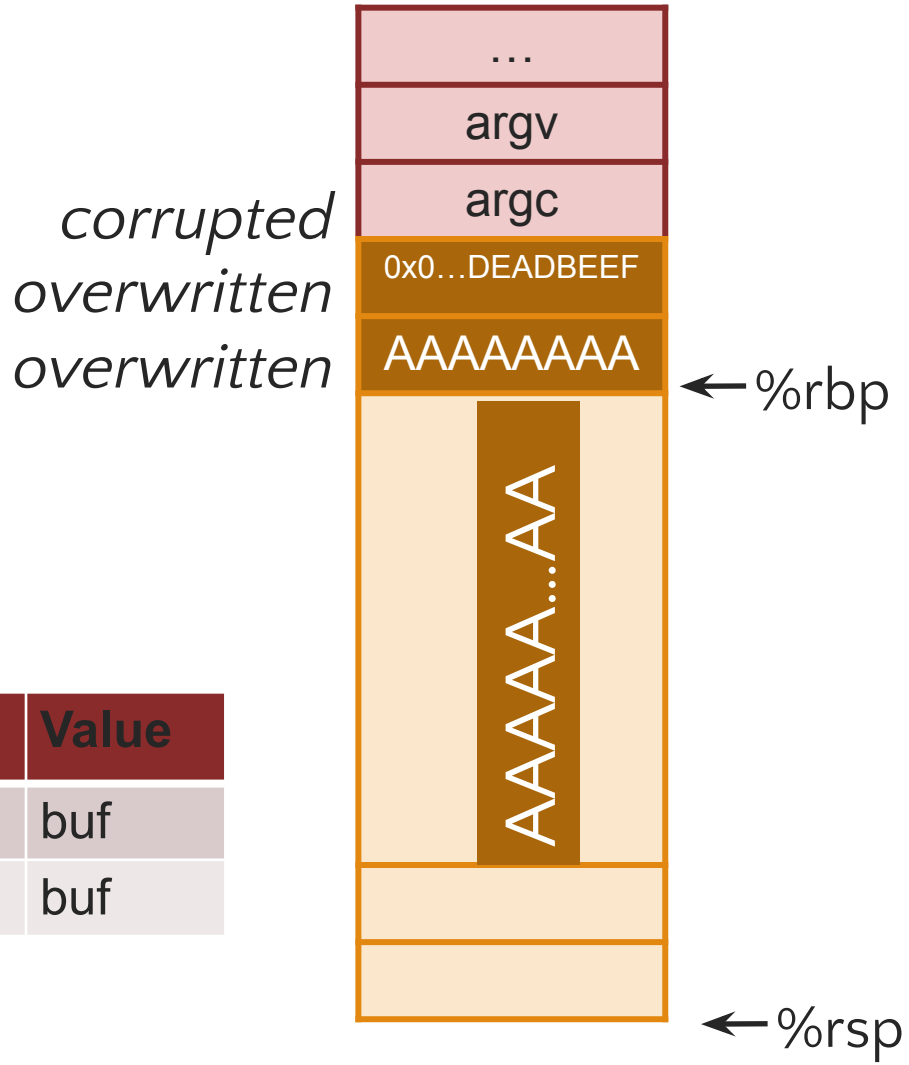


Input

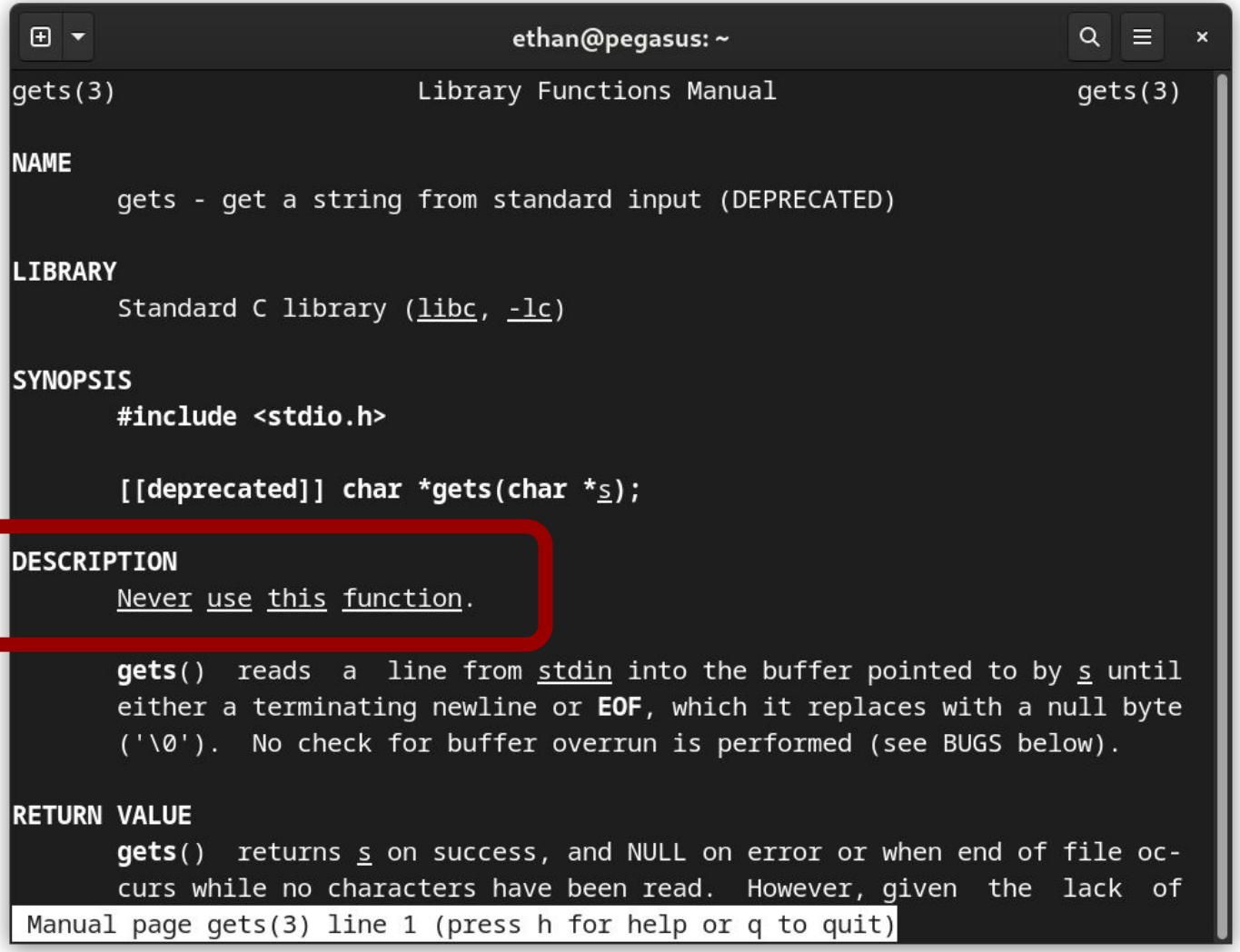
"A"x72 + "\xEF\xBE\xAD\xDE\x00\x00\x00\x00"

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    gets(buf);
}
Dump of assembly code for function main:
  4004fd: push   %rbp
  4004fe: mov    %rsp,%rbp
  400501: sub    $0x50,%rsp
  400505: mov    %rdi,-0x48(%rbp)
  400508: mov    %rsi,-0x50(%rbp)
  40050c: lea    -0x40(%rbp),%rax
  400510: mov    %rax,%rdi
  400518: callq  400400 <gets@plt>
  40051d: leaveq
  40051e: retq
```

Reg	Value
rax	buf
rdi	buf



PSA: Avoid "gets"



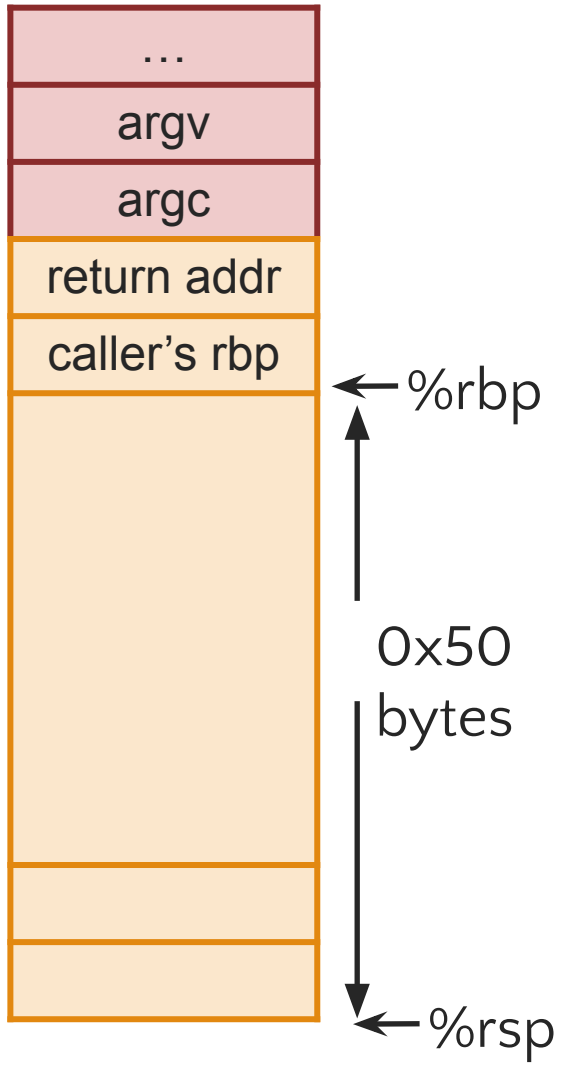
StackGuard

[Cowen et al. 1998]

Idea:

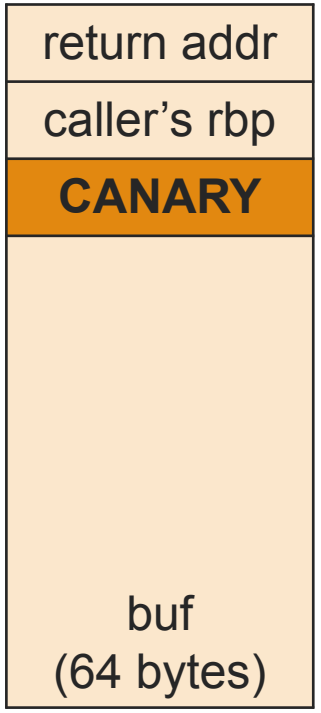
- prologue introduces a ***canary word*** between return addr and locals
- epilogue checks canary before function returns

Wrong canary => Overflow



gcc Stack-Smashing Protector (ProPolice)

```
Dump of assembler code for function main:
   4005a0: sub    $0x58,%rsp
   4005a4: mov    %fs:0x28,%rax
   4005ad: mov    %rax,0x48(%rsp)
   4005b2: xor    %eax,%eax
   4005b4: mov    %rsp,%rdi
   4005b7: callq  4004a0 <gets@plt>
   4005bc: mov    0x48(%rsp),%rdx
   4005c1: xor    %fs:0x28,%rdx
   4005ca: je     4005d1 <main+0x31>
   4005cc: callq  400470 <__stack_chk_fail@plt>
   4005d1: add    $0x58,%rsp
   4005d5: retq
```



Compiled with `**gcc -fstack-protector**`

(you can also use -fstack-protector-all or -fstack-protector-strong)

14

Canary Should Be **HARD** to Forge

- Terminator canary
  - 4 bytes: 0,CR,LF,-1 (low->high)
  - terminate strcpy(), gets(), ...
- Random canary
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness

Proposed Defense Scorecard

Aspect	Defense
Performance	• Smaller impact is better
Deployment	• Can everyone easily use it?
Compatibility	• Doesn’t break libraries
Safety Guarantee	• Completely secure vs. easy to bypass

Canary Scorecard

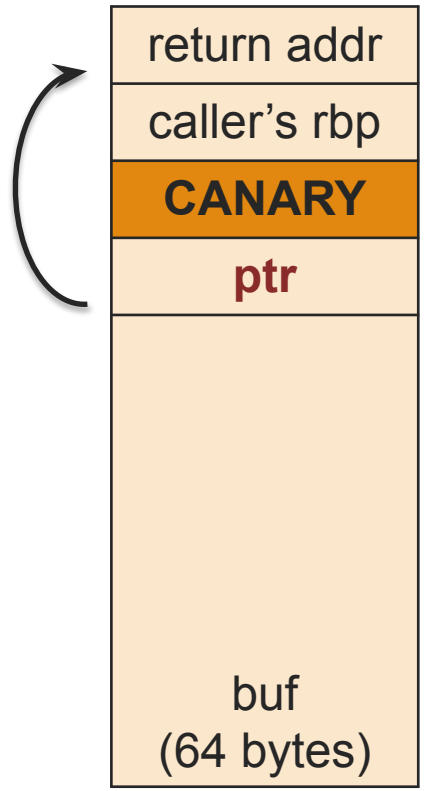
Aspect	Canary
Performance	• several instructions per function • time: a few percent on average • size: can optimize away in safe functions (but see MS08-067 *)
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *not really…*

Shadow stack and canaries performance

Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*…

```
int *ptr;
char buf[64];
memcpy(buf, user1, large);
*ptr = user2;
```



Bypass: Combine with a memory leak

Print out canary value first and use it in overwrite!

```
int *ptr;
char buf[64];
printf(user2);
memcpy(buf, user1, large);
```

Canary Weakness

Check does ***not*** happen until epilogue...

- func ptr subterfuge
- C++ vtable hijack
- exception handler hijack
- ...

PointGuard

SafeSEH
SEHOP

ProPolice
puts arrays
above others
when
possible

struct is fixed;
& what about heap?

Quiz

Which attack would be ***MOST*** effective at **hijacking control on a canary-protected machine**?

A. Using a **single** `memcpy`-based buffer overflow of a local variable

B. Using a format-string vulnerability **and** the "%n" specifier

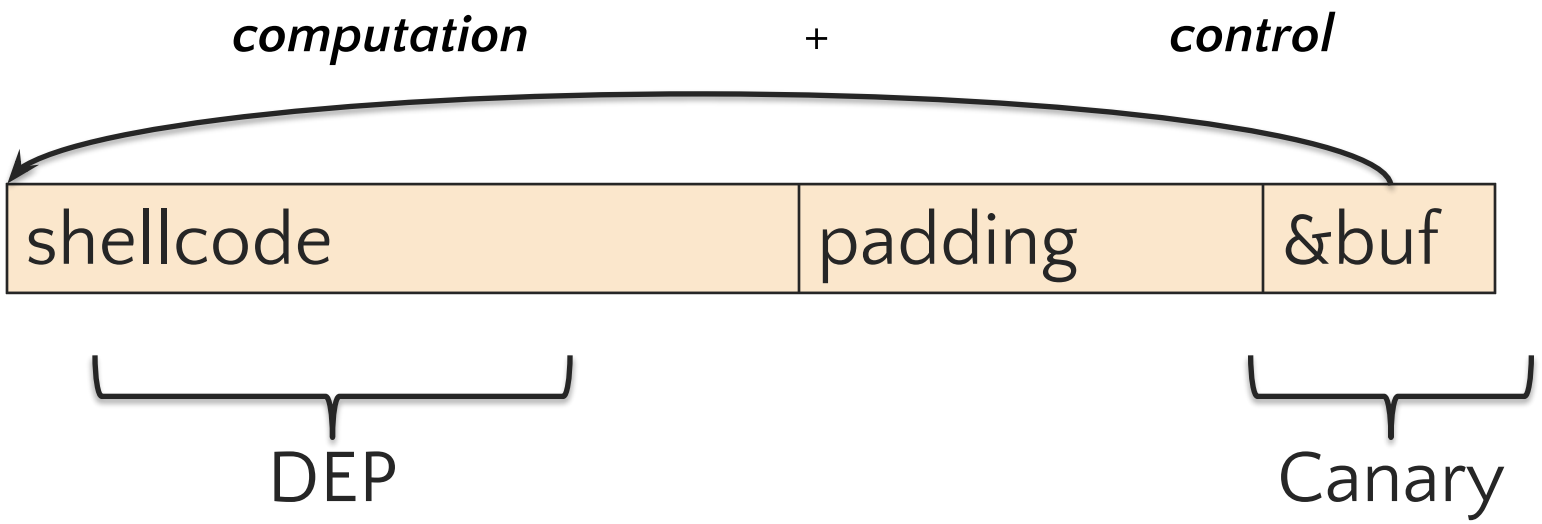
C. Using a format-string vulnerability **and** a targeted address specifier (e.g., **"%9$sBBBB\x47\xf7\xff\xff"**)

D. Using a format-string overflow of a local variable (e.g., **"%80u\x3c\xd3\xff\xff"**)

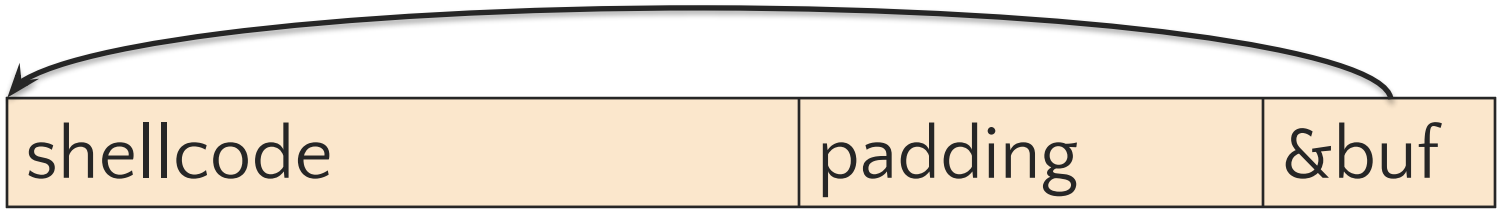
Data Execution Prevention (DEP) / No eXecute (NX)

Idea: maybe we shouldn't allow data to be executable

How to Defeat Exploits?



Data Execution Prevention



Mark stack as non-executable using NX bit

(still a Denial-of-Service attack!)

DEP prevents injected code on the stack from executing

DEP Scorecard

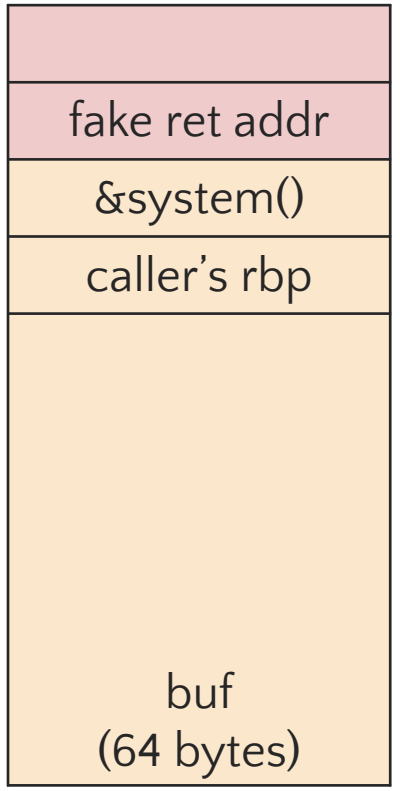
Aspect	Data Execution Prevention
Performance	• with hardware support: no impact • otherwise: reported to be <1% in PaX
Deployment	• kernel support (common on all platforms) • modules opt-in (less frequent in Windows)
Compatibility	• can break legitimate programs – Just-In-Time compilers – unpackers
Safety Guarantee	• code injected to NX pages never execute • *but code injection may not be necessary…*

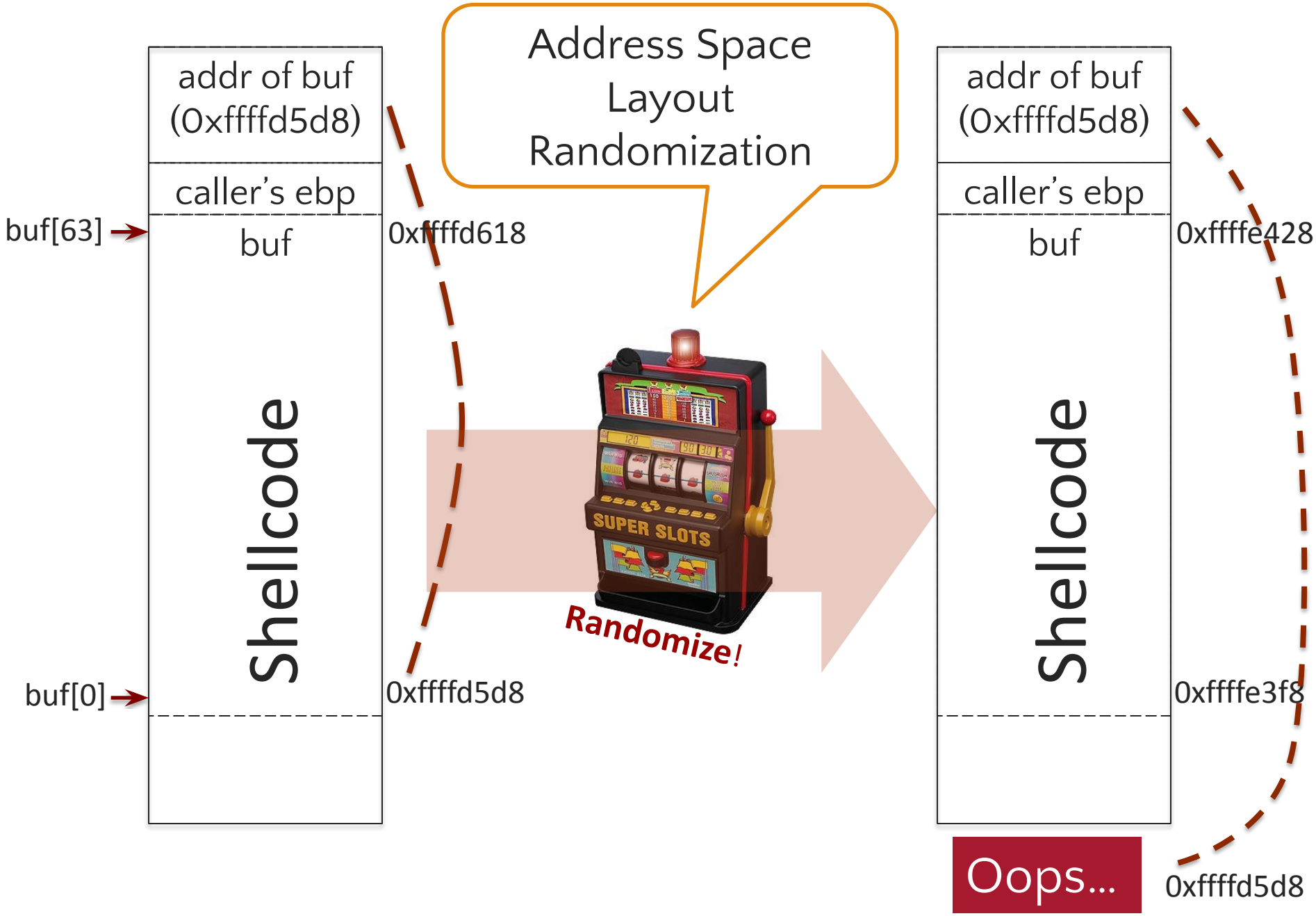
Return-to-libc Attack

Overwrite return address with the address of a libc function

- setup fake return address
- put arguments (e.g. “/bin/sh”) in correct registers / memory
- `ret` will “call” libc function

No injected code!




Address Space Layout Randomization
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
buf[63]
buf[0]
0xffffd618
0xffffd5d8
SUPER SLOTS
Randomize!
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
0xffffe428
0xffffe3f8
Oops...
0xffffd5d8

ASLR

Traditional exploits need precise addresses

- *stack-based overflows:* location of shell code
- *return-to-libc:* library addresses

- **Problem:** program's memory layout is fixed
  - stack, heap, libraries etc.

- **Solution:** randomize addresses of each region!

The /proc filesystem (1/2)

proc is a virtual filesystem in Linux that provides a way to access system and process information.

- No actual files on disk everything generated dynamically in memory.

Key Features:

- Mounted at /proc
- Provides real-time system and process details
- Used for debugging, monitoring, and system configuration

Not sure how it works? Guess how we find out!

The /proc filesystem (2/2)

Typical useful examples:

Command	Description
/proc/cpuinfo	CPU details (cores, speed, vendor)
/proc/meminfo	Memory usage info
/proc/[PID]/cmdline	Command-line arguments of a process
/proc/[PID]/fd/	Open file descriptors of a process
/proc/[PID]/maps	Process memory layout
/proc/sys/	Kernel parameters (modifiable via sysctl)

Want to reference your own PID's resources? Use "self"! E.g., cat /proc/self/cmdline

Running cat Twice

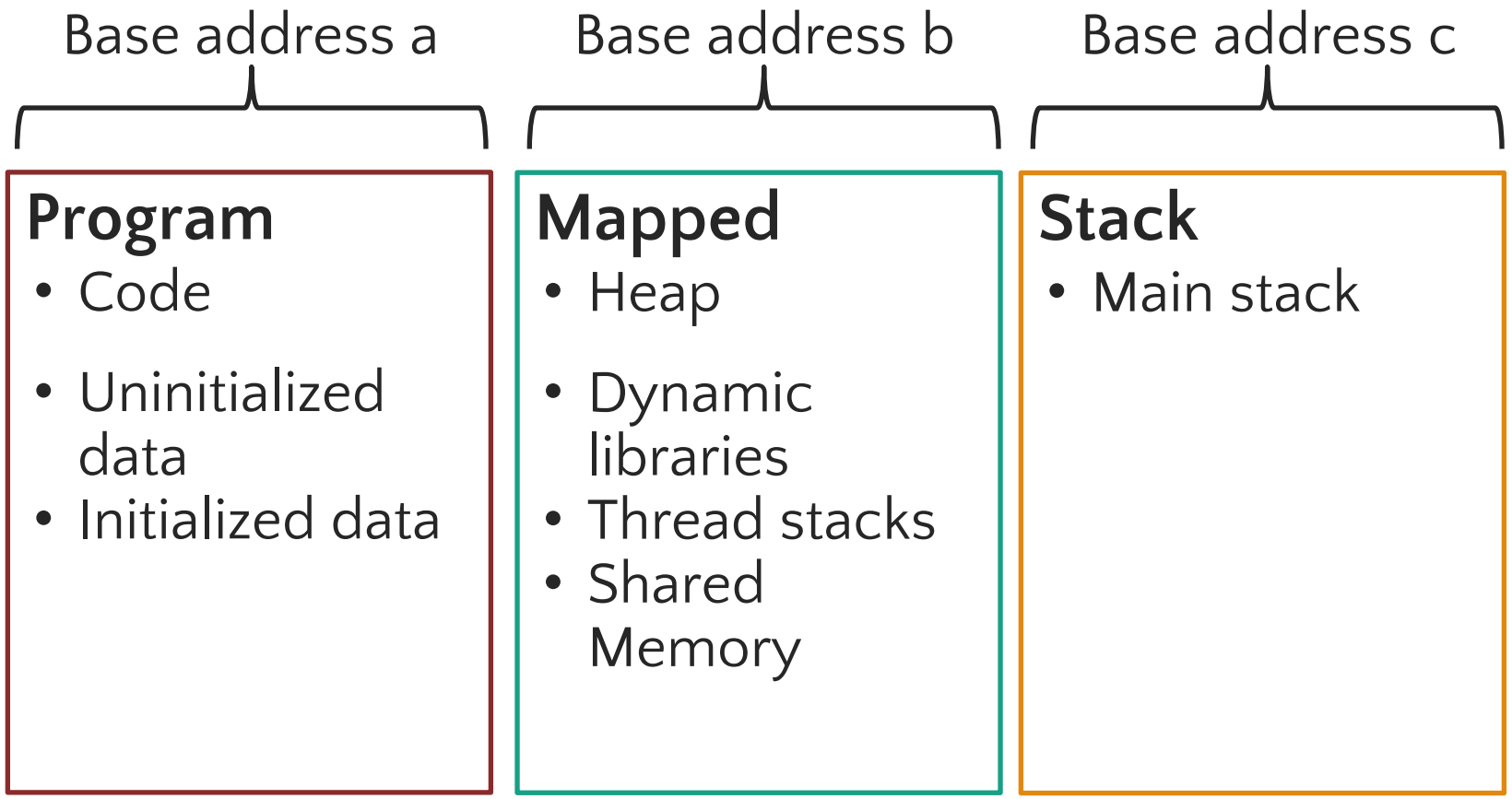
- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

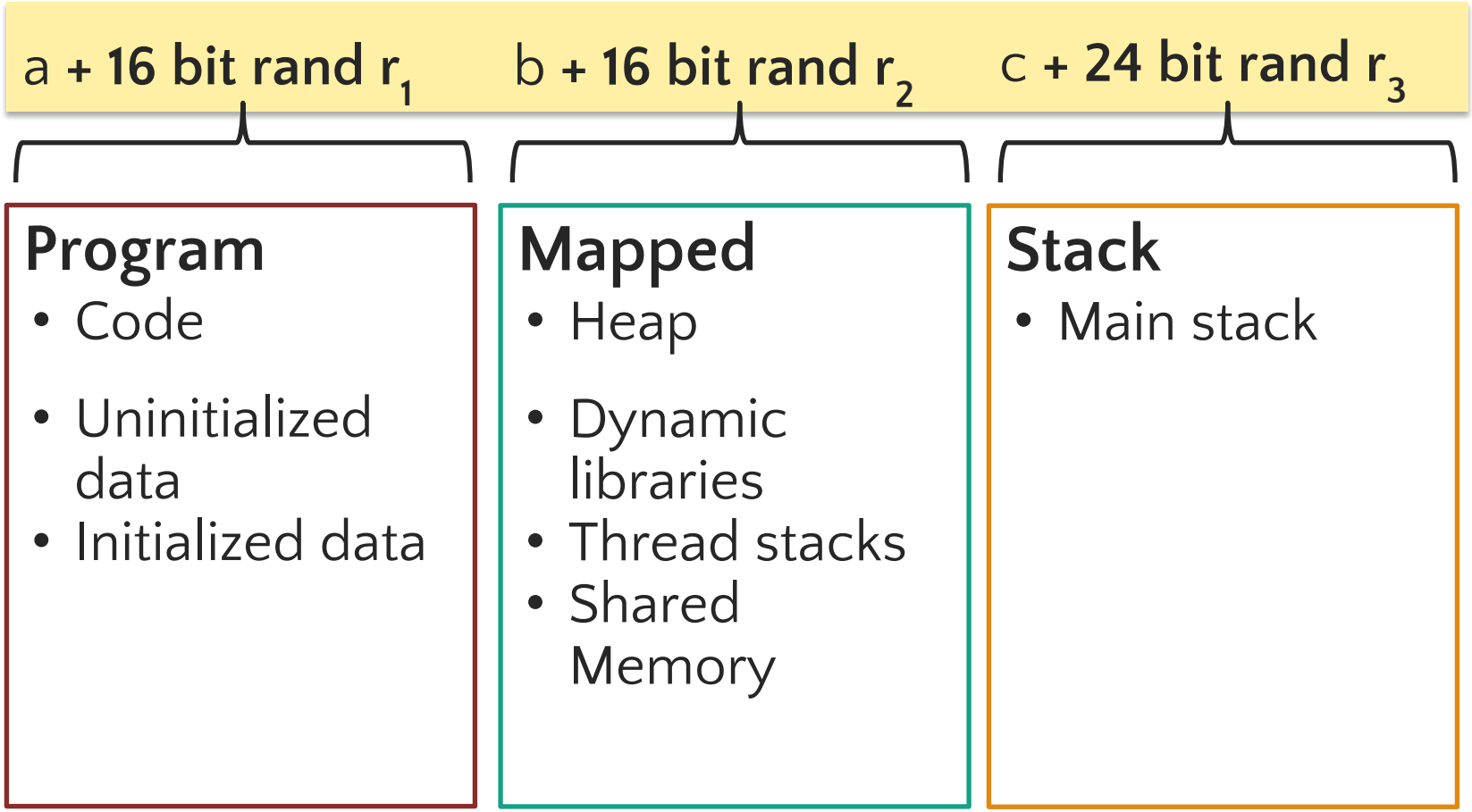
- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

Memory


Base address a
Base address b
Base address c
Program
• Code
• Uninitialized data
• Initialized data
Mapped
• Heap
• Dynamic libraries
• Thread stacks
• Shared Memory
Stack
• Main stack

ASLR Randomization



* ≈ 16 bit random number of 32-bit system. More on 64-bit systems.

ASLR Scorecard

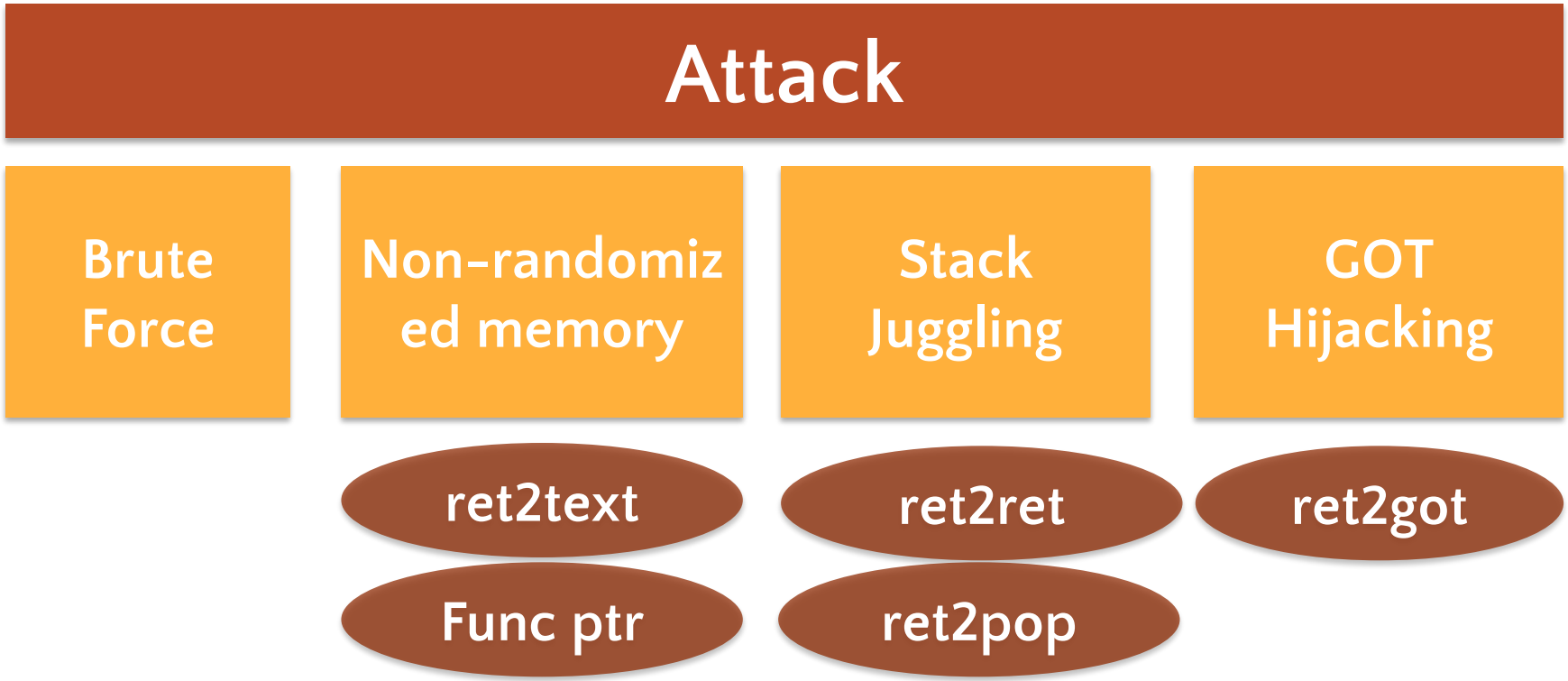
Aspect	Address Space Layout Randomization
Performance	• excellent—randomize once at load time
Deployment	• turn on kernel support (Windows: opt-in per module, but system override exists) • no recompilation necessary
Compatibility	• transparent to safe apps (position independent)
Safety Guarantee	• not good on x32, much better on x64 • *code injection may not be necessary…*

Ubuntu - ASLR

- ASLR is **ON** by default [Ubuntu-Security]
  - cat /proc/sys/kernel/randomize_va_space
    - In older systems: **1** *(stack/mmap ASLR)*
    - In later releases: **2** *(stack/mmap/brk ASLR)*
  - stack/mmap/brk/exec ASLR: available since 2008 - still systems around without it
    - Position Independent Executable (PIE) with "-fPIE –pie"

Remember: you probably want this enabled

How to Attack ASLR?



Checking which defenses are on

- Can be done by inspecting the binary
- Or using tools made for this – e.g., checksec (apt install)

```
$ checksec --file=/bin/ls
RELRO           STACK CANARY     NX         PIE             RPATH      RUNPATH    Symbols     FORTIFY    Fortified    Fortifiable    FILE
Full RELRO      Canary found     NX enabled PIE enabled     No RPATH   No RUNPATH   No Symbols      Yes    6       18         /bin/ls
```

http://slimm609.github.io/checksec.sh/

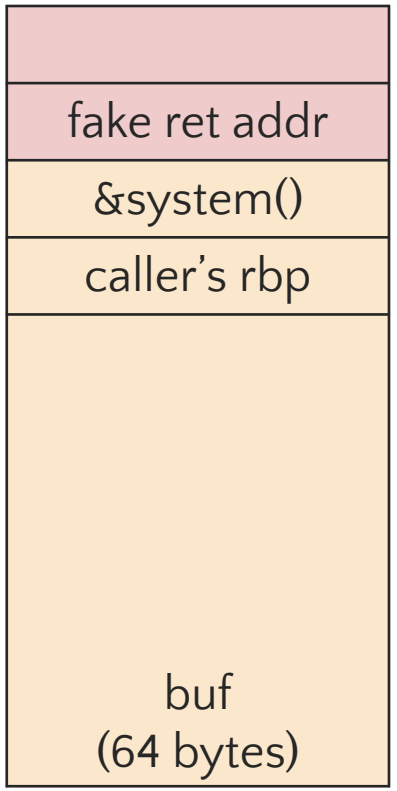

return-Oriented
PROgRamming

Bypass with return-to-libc Attack (beat DEP)

Rely on existing code (e.g., `system()`)
rather than injecting new code

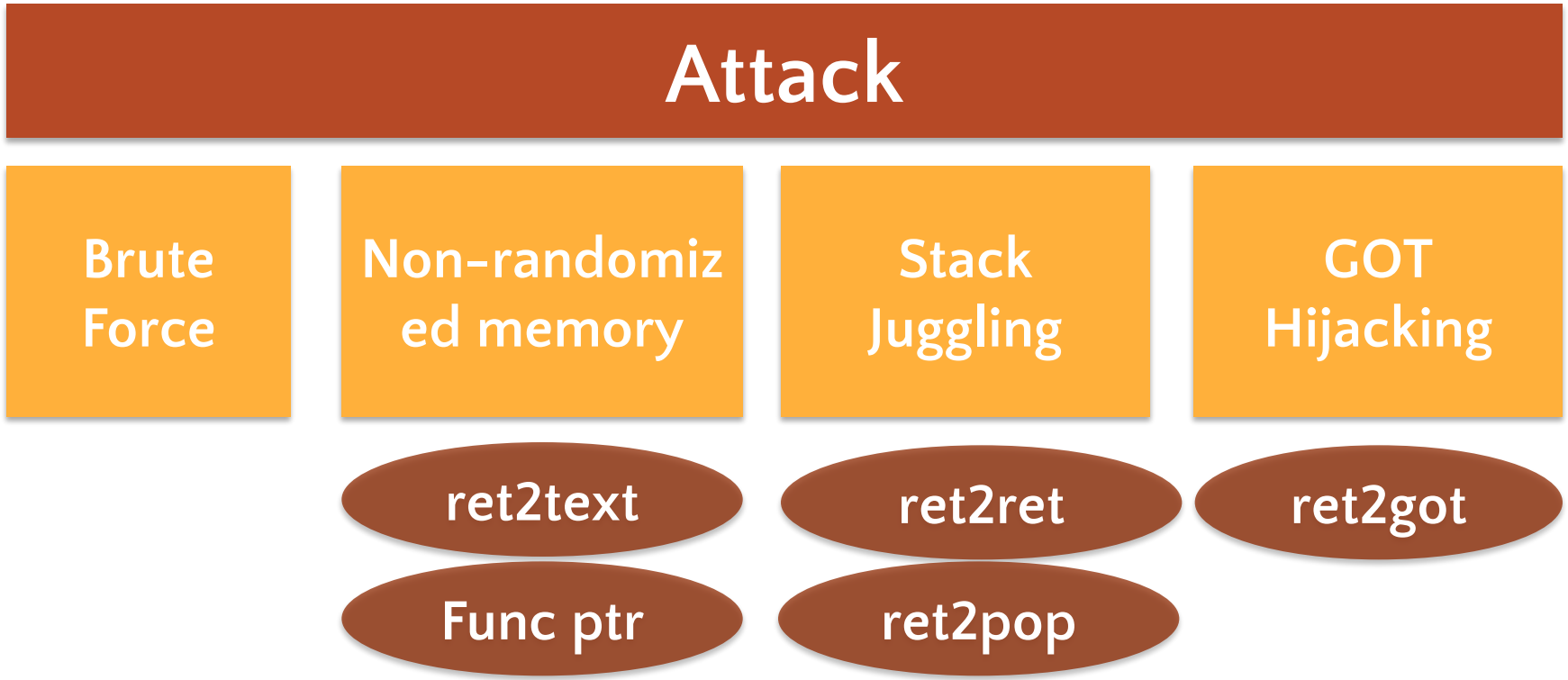
- setup fake return address
- put arguments (e.g. "/bin/sh") in correct registers
- `ret` will "call" libc function

No injected code!

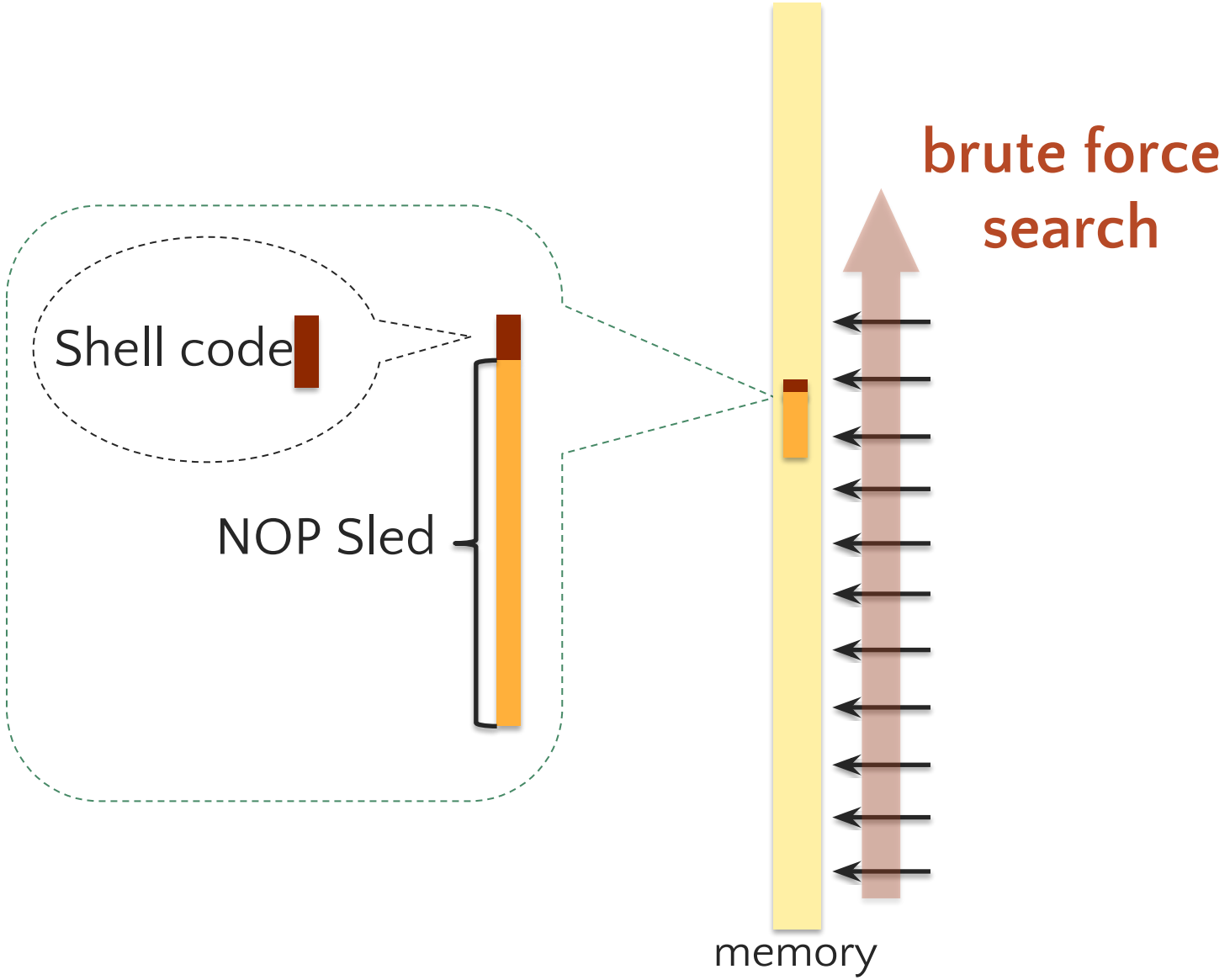


Example ret2libc

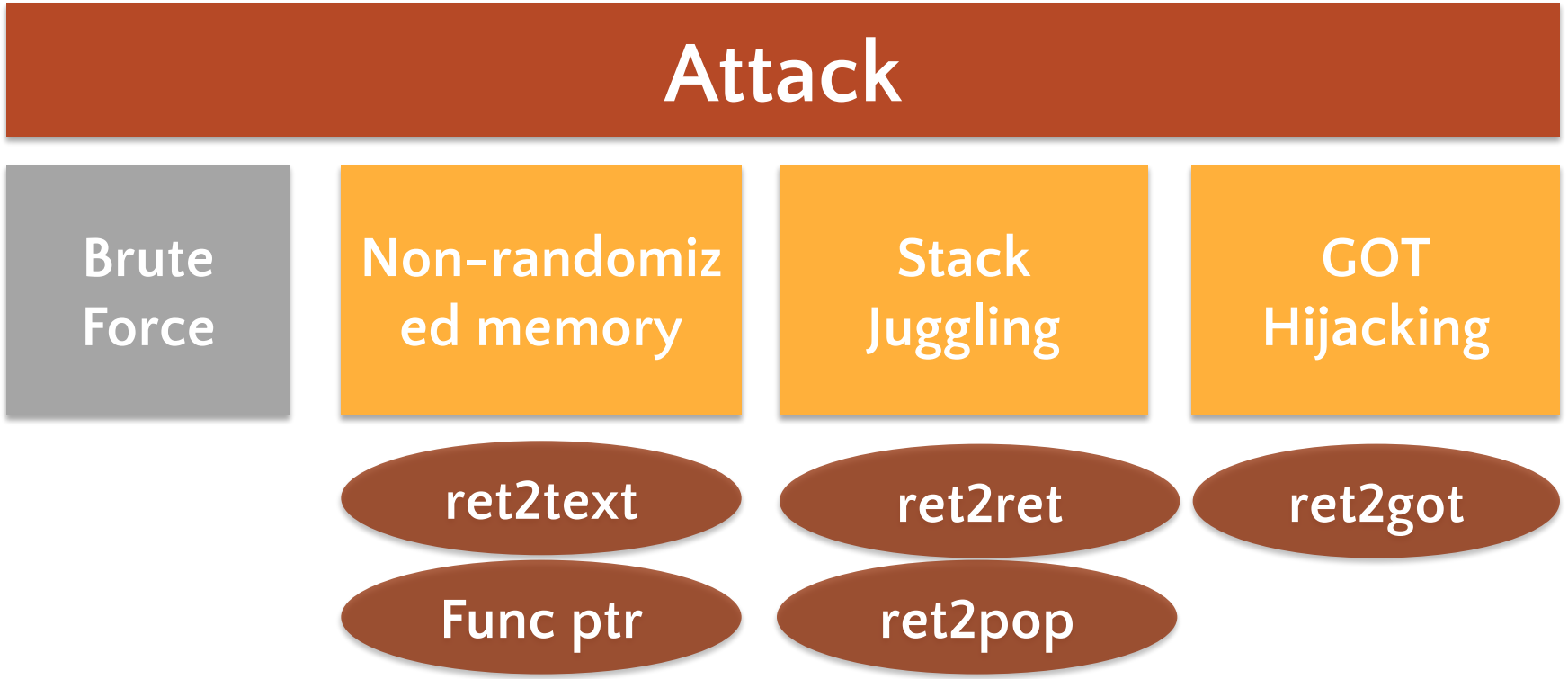
How to Attack ASLR?



Brute Force



How to Attack ASLR?



ret2text attack

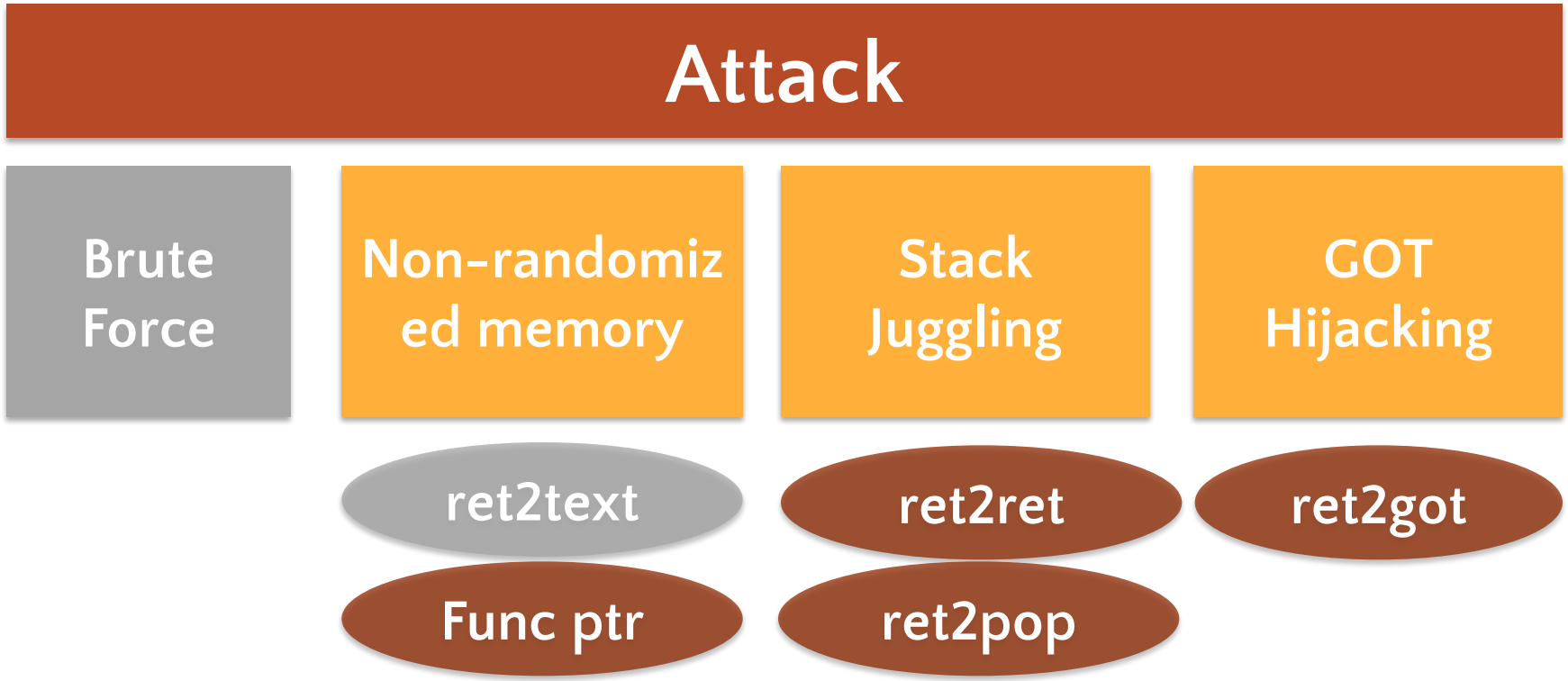
Use this if .text section is ***not*** randomized

(Older gcc did not randomize text without –PIE flag.)

```
# Old GCC (<2017) did not randomize text
$ gcc main.c –o main          # Default does not create PIE
$ gcc main.c -o main –fPIE    # Flag required to enable PIE

# Modern GCC (~2017)
$gcc main.c –o main –no-pie   # Specifically disable PIE
$ gcc main.c –o main          # PIE by default!
```

Reference: https://leimao.github.io/blog/PIC-PIE/

How to Attack ASLR?

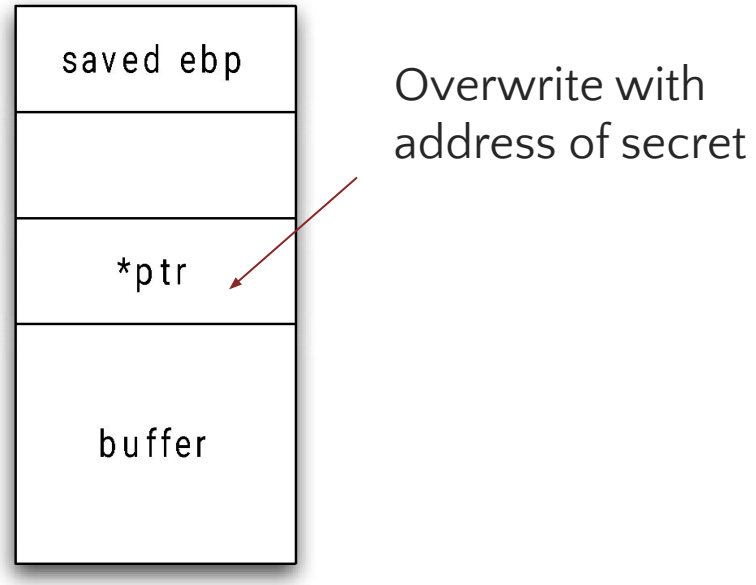


Function Pointer Subterfuge

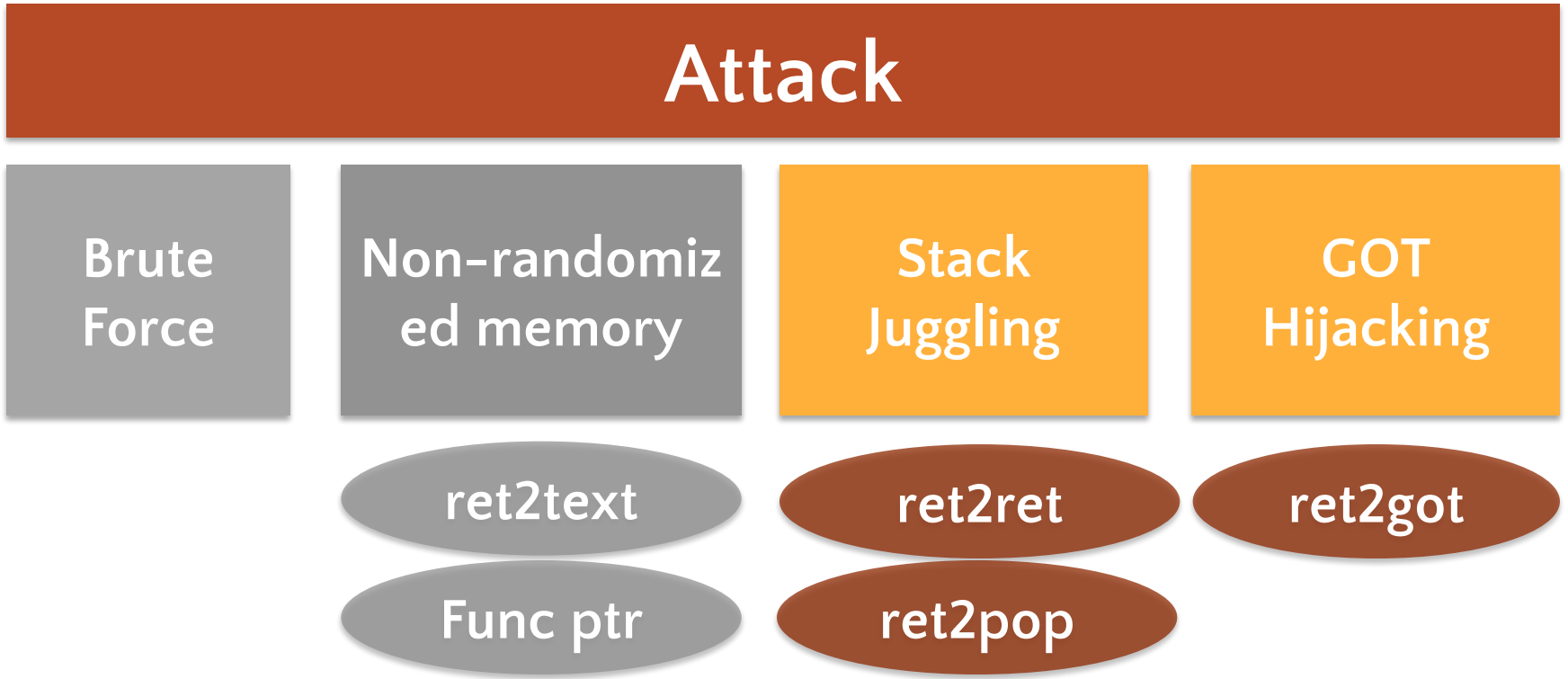
Overwrite a function pointer to point to:

- program function (similar to ret2text)
- another lib function in Procedure Linkage Table

```
/*please call me!*/
int secret(char *input) { … }

int chk_pwd(char *intput) { … }

int main(int argc, char *argv[]) {
    int (*ptr)(char *input);
    char buf[8];

    ptr = &chk_pwd;
    strncpy(buf, argv[1], 12);
    printf("[] Hello %s!\n", buf);

    (*ptr)(argv[2]);
}
```



How to Attack ASLR?



Quiz Question

Which of the following can undermine ASLR?

A. A static .text section

B. A memory disclosure vulnerability that leaks the location of libc functions

C. Function pointers at a known address

D. All of the above

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!