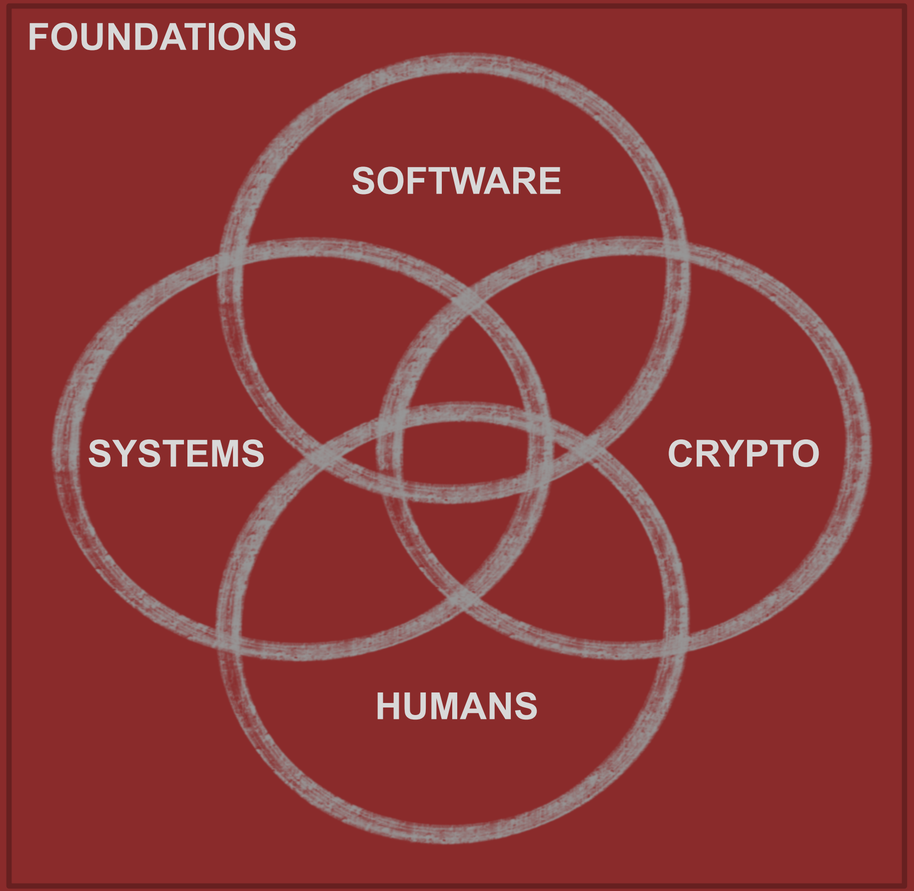
Διάλεξη #7-8 - Bypassing Defenses & Return-Oriented Programming (ROP)



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

Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #1 βγήκε - προθεσμία 24 Απριλίου 10:59πμ

- Πόσα vulnerabilities έχουμε;

NVD Dashboard

CVEs Received and Processed

Time Period	New CVEs Received by NVD	New CVEs Analyzed by NVD	Modified CVEs Received by NVD	Modified CVEs Re-analyzed by NVD
Today	76	0	0	1
This Week	353	26	0	2
This Month	1032	50	0	17
Last Month	3370	199	0	102
This Year	9760	4349	0	1225

CVE Status Count

Total	244898
Received	353
Awaiting Analysis	5799
Undergoing Analysis	191
Modified	93931
Rejected	14018

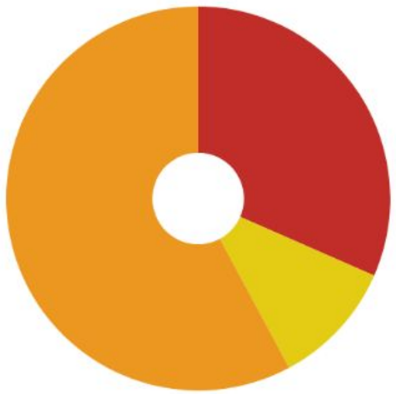
NVD Contains

CVE Vulnerabilities	244898
Checklists	784
US-CERT Alerts	249
US-CERT Vuln Notes	4486
OVAL Queries	10286
CPE Names	1263462

CVSS V3 Score Distribution

Severity	Number of Vulns
CRITICAL	23039
HIGH	60944
MEDIUM	59916
LOW	2529

CVSS V2 Score Distribution



Severity	Number of Vulns
HIGH	56837
MEDIUM	104170
LOW	19074

https://nvd.nist.gov/general/nvd-dashboard

Την Προηγούμενη Φορά

1. Adversary and Classifications
2. Mitigations
    - Canaries
    - DEP
    - ASLR

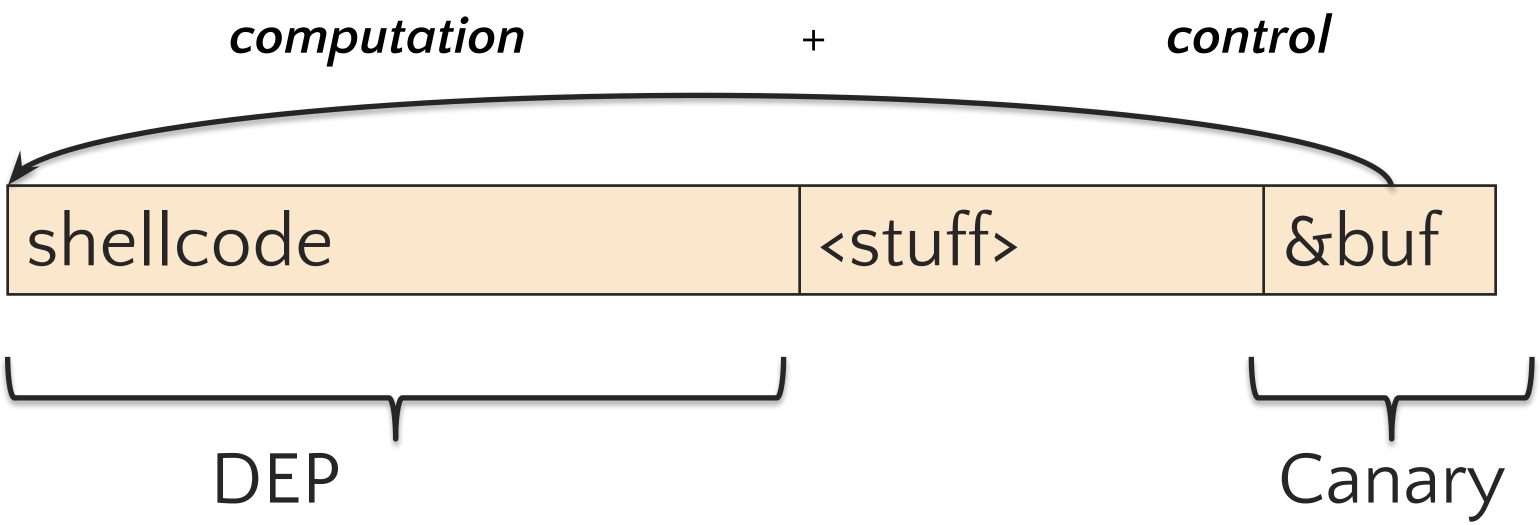
Σήμερα

- Bypassing Mitigations
- Return-Oriented Programming (ROP)

Where we left off

Data Execution Prevention

Mark stack as non-executable using NX bit

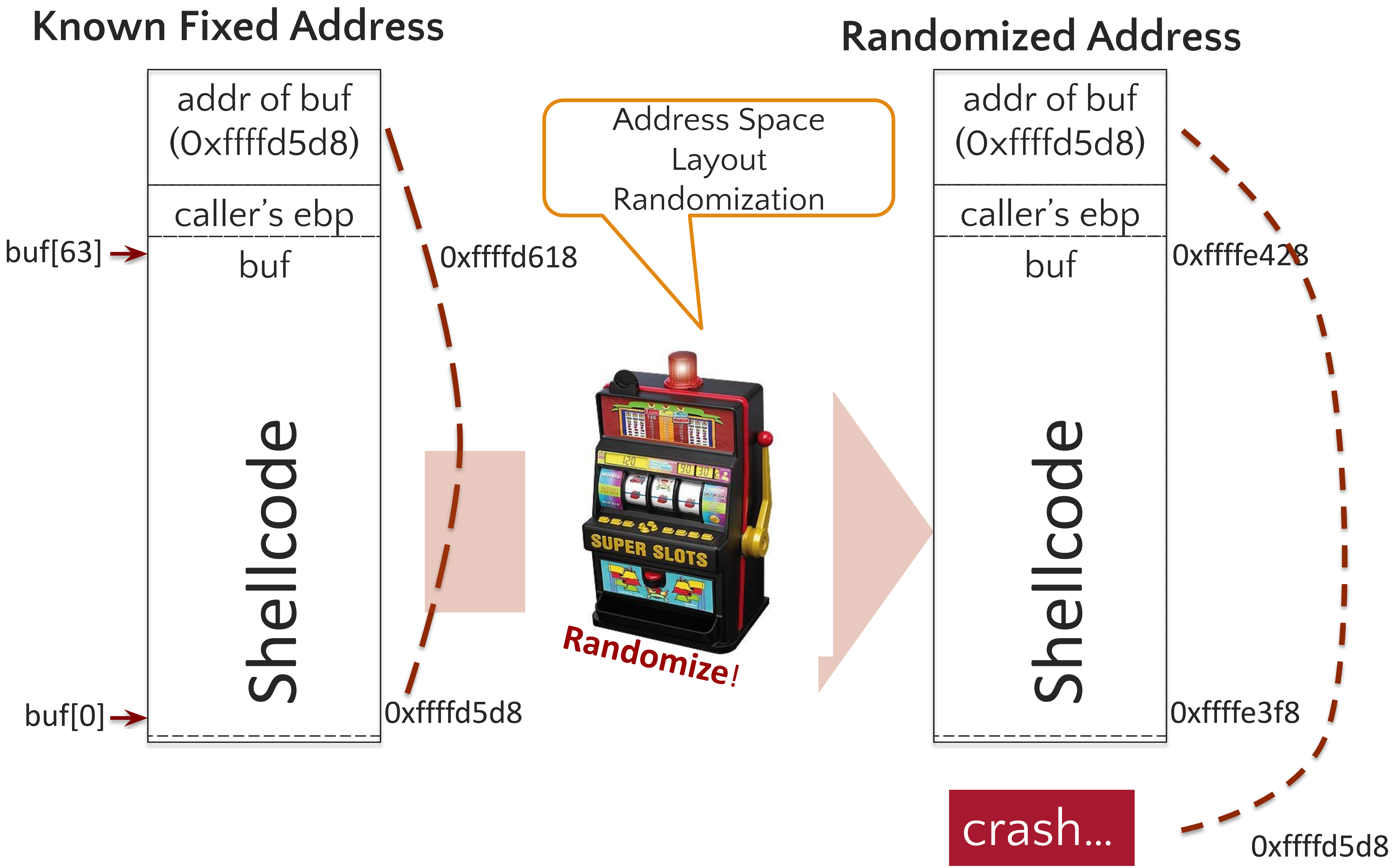


CRASH

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…*



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

Memory

Base address a

Program
- Code
- Uninitialized data
- Initialized data

Base address b

Mapped
- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Base address c

Stack
- Main stack

ASLR Randomization

a + **16 bit rand r_1** | b + **16 bit rand r_2** | c + **24 bit rand r_3**

Program
- Code
- Uninitialized data
- Initialized data

Mapped
- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack
- Main stack

* ≈ 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…*

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/

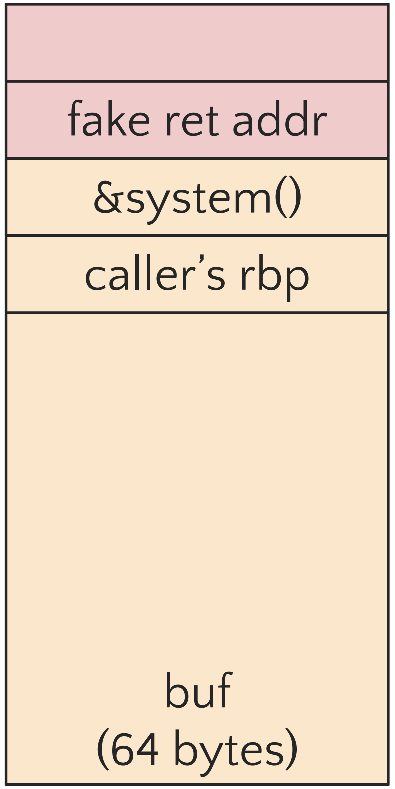
return-oriented programming

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

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

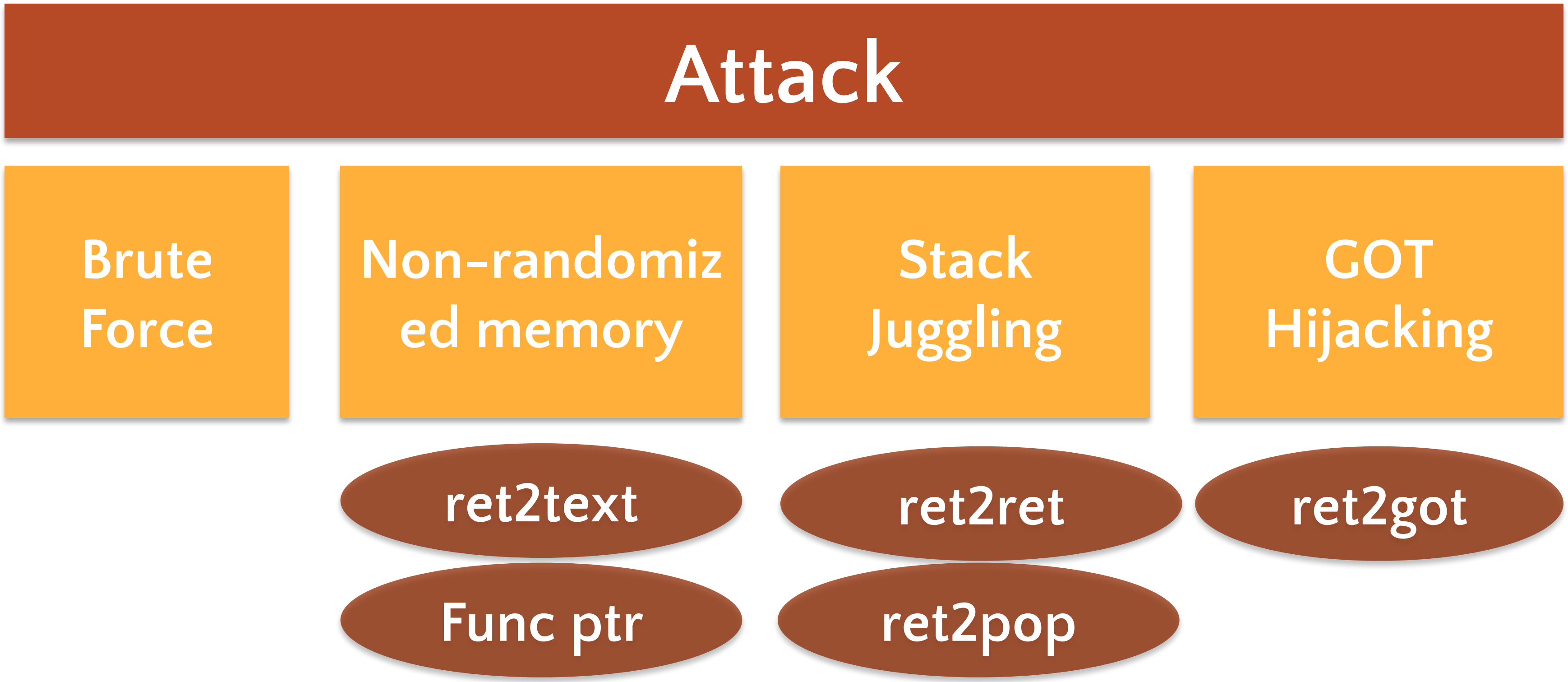
- setup fake return address
- put arguments (e.g. "/bin/sh") in correct registers
- `ret` will "call" libc function

No injected code!

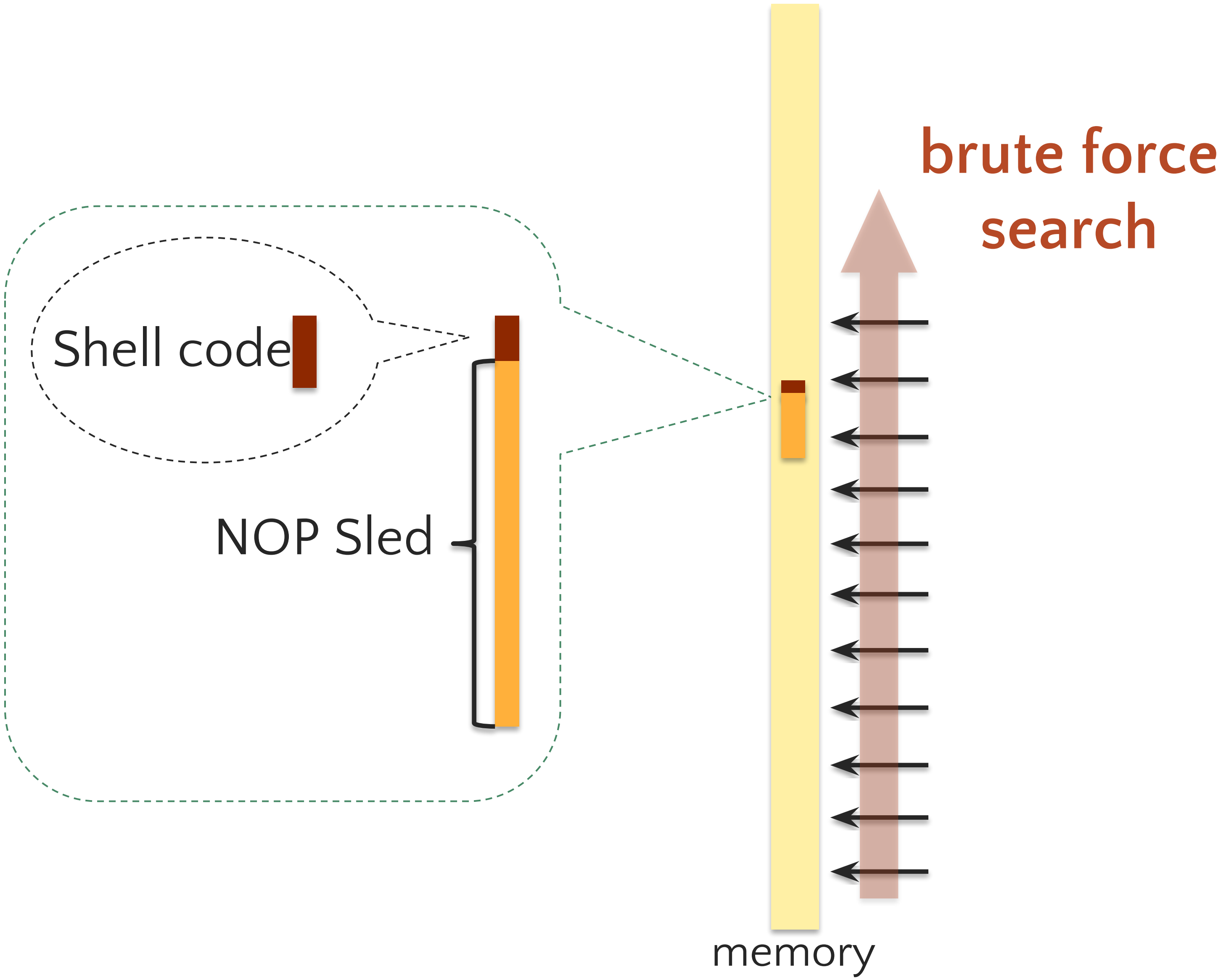


Example ret2libc

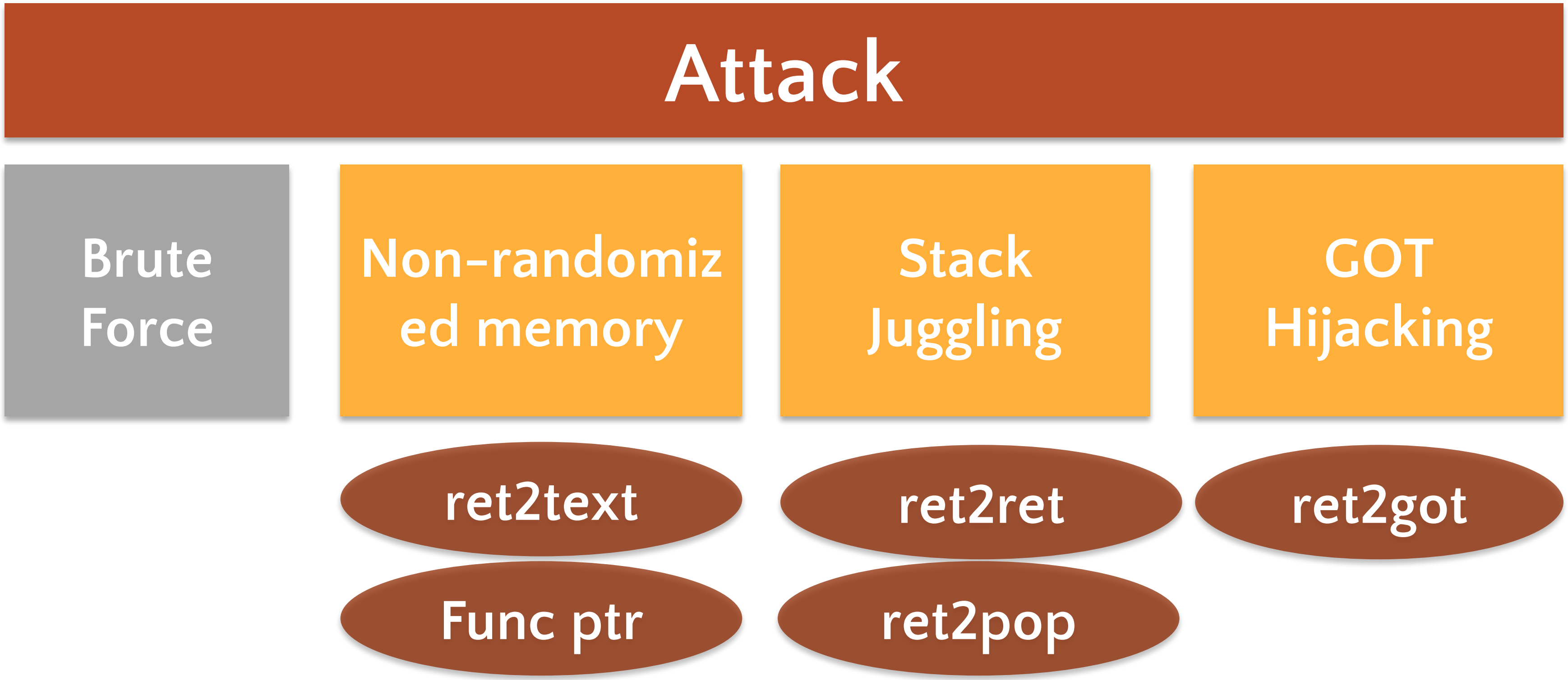
How to Attack ASLR?



Brute Force



How to Attack ASLR?



ret2text attack

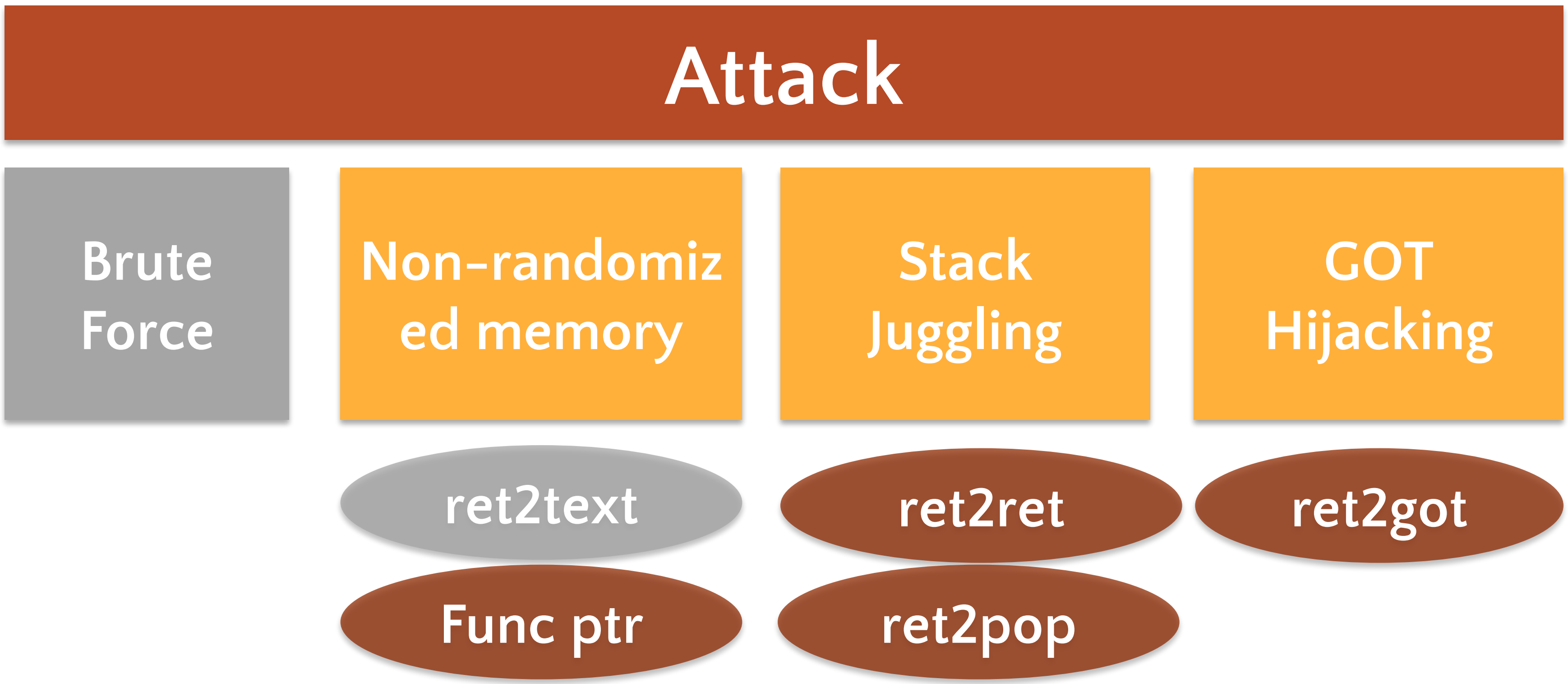
Use this if .text section is ***not*** randomized

(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?

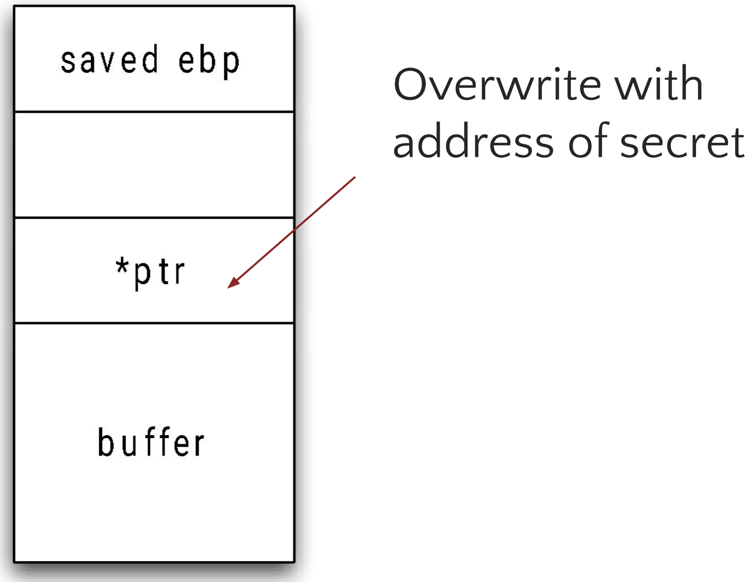


Function Pointer Subterfuge

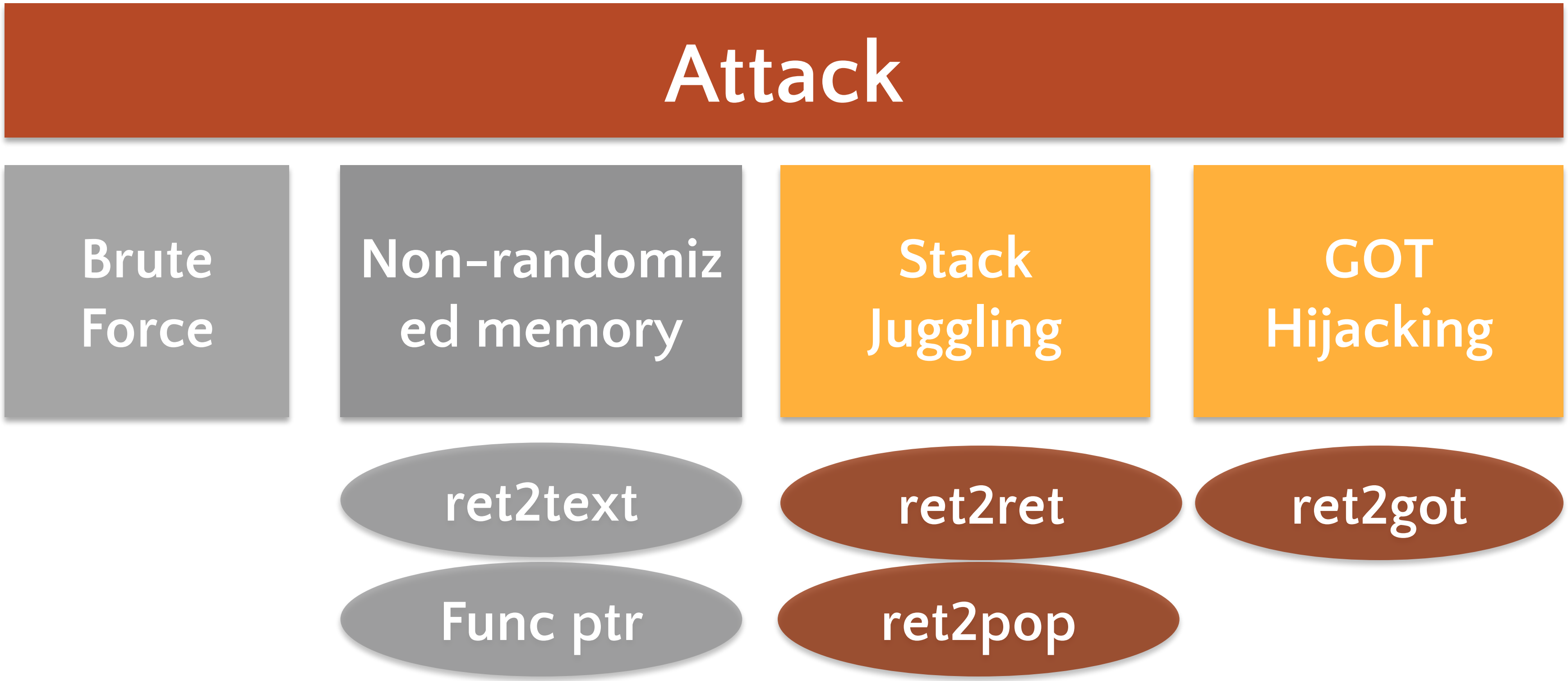
Overwrite a function pointer to point to:

- 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

Image by Dino Dai Zovi

Idea:

We forge shell code out of existing application logic gadgets

Requirements:

vulnerability + gadgets + some *unrandomized* code

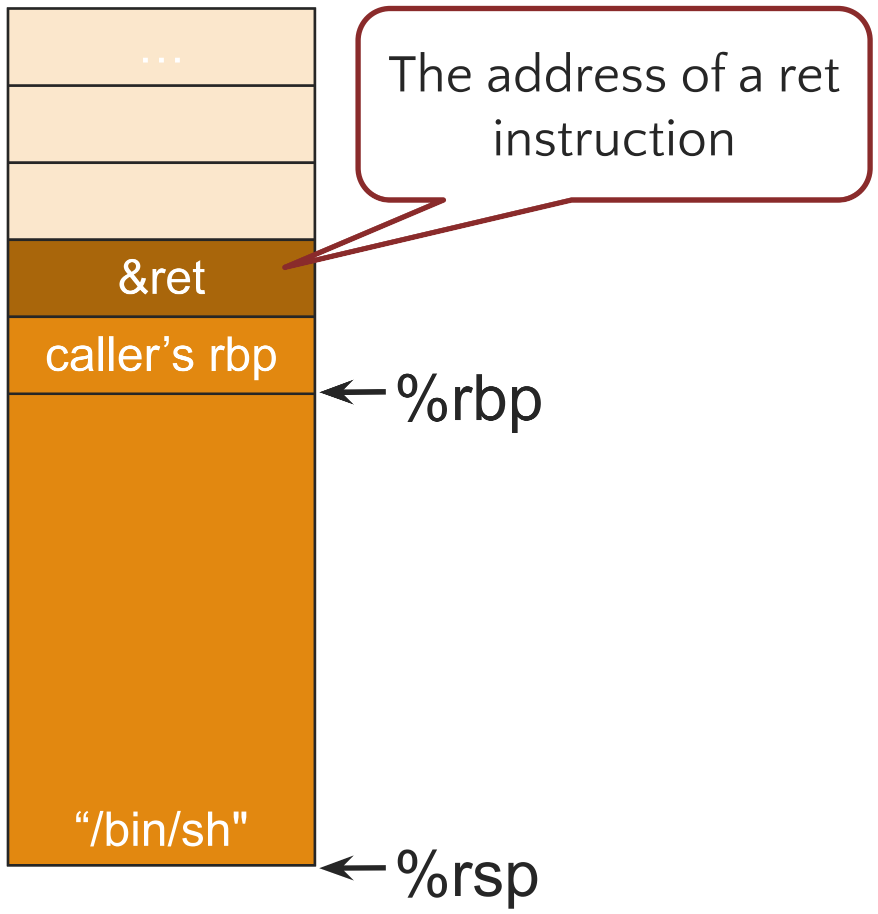
Where do we get unrandomized code?

- 3rd party library not randomized
- Compiler did not randomize
- Information disclosure vuln leaks the randomization (e.g., base address)
  - Info disclosure exploit *that chains into*
  - Control flow hijack exploit

```
ret = pop rip; jmp rip;
```

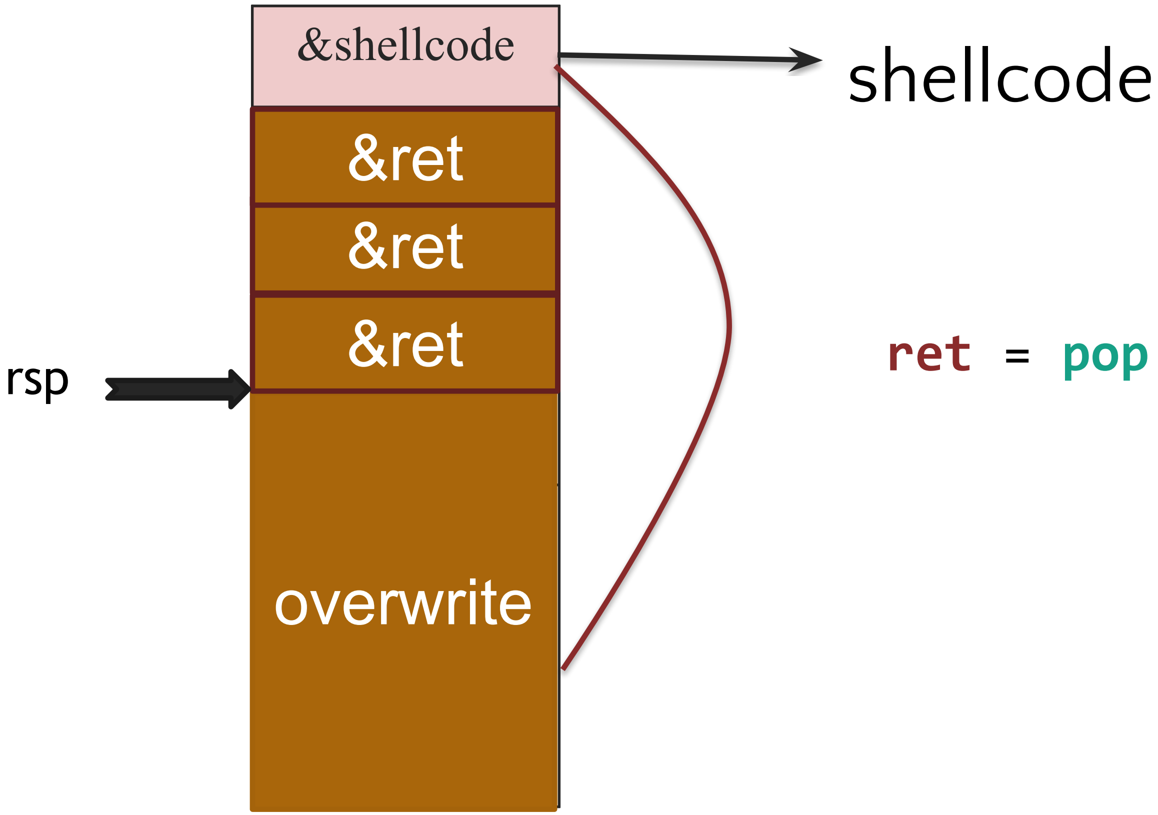
`ret` is an indirect jump to whatever is on the stack.

ROP is like programming a stack-based machine.



ret2ret

If there is a valuable (***potential shellcode***) **pointer** on a stack, you might consider this technique.

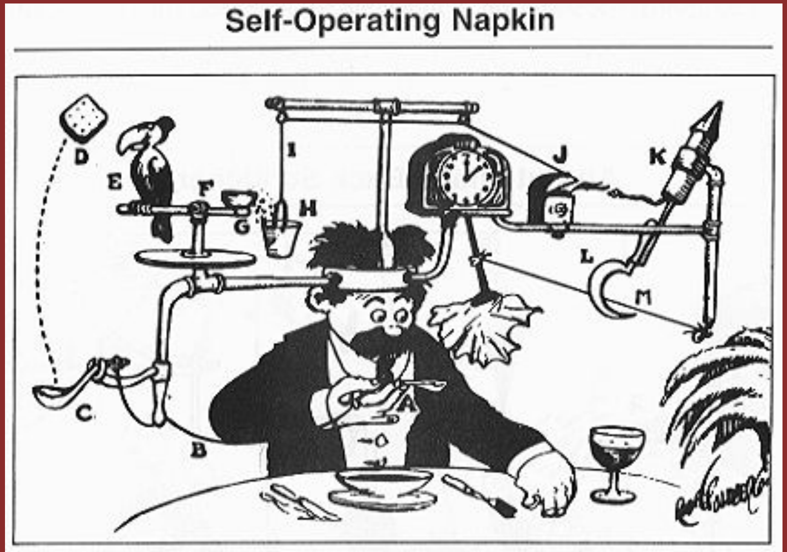



```
ret = pop rip; jmp rip;
```

Shellcode isn’t restricted to us manually encoding instructions.

We can write shellcode “programs” using “gadgets” from existing instructions

Gadgets



Ανακοινώσεις / Διευκρινίσεις

- Η Εργασία #1 είναι διαθέσιμη και στο https://hackintro.di.uoa.gr
  - χθες είχαμε ένα outage - apologies
- Τι είναι το "p" στα perms του /proc/maps; - man proc!
- Που τοποθετείται το core αρχείο όταν ένα πρόγραμμα κρασάρει;

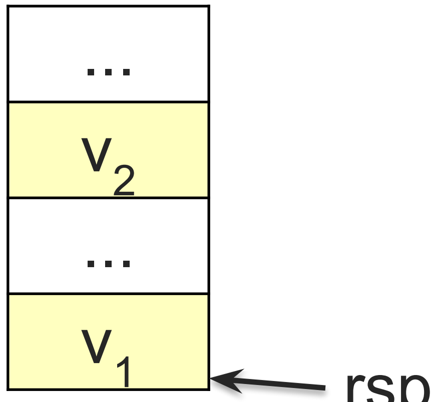
Χθες και Σήμερα

- Bypassing Mitigations
- Return-Oriented Programming (ROP)

An Example Operation

`Mem[v2] := v1`

Desired Logic



Stack

a_1: mov rax, [rsp] ; rax has v1

a_2: mov rbx, [rsp+16] ; rbx has v2

a_3: mov [rbx], rax ; Mem[v2] := rax

Implementation 1

Intel syntax

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic

Suppose a_5 and a_3 on stack

a_5
v_2
a_3
v_1 ← rsp

Stack

rax	v_1
rbx	
rip	a_1

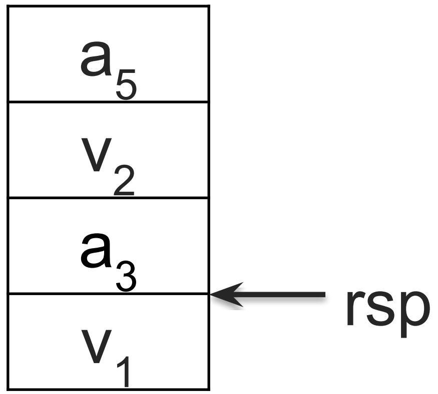
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

```
Mem[v2] := v1
```

Desired Logic



Stack

rax	v_1
rbx	
rip	a_3

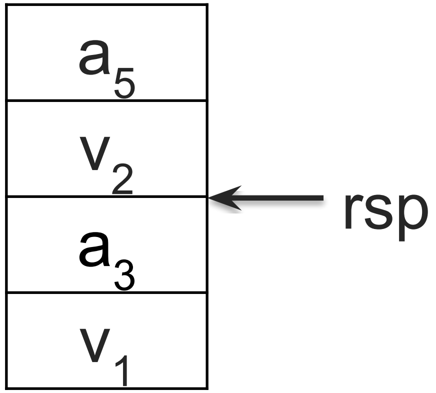
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

`Mem[v2] := v1`

Desired Logic



Stack

rax	v_1
rbx	v_2
rip	a_3

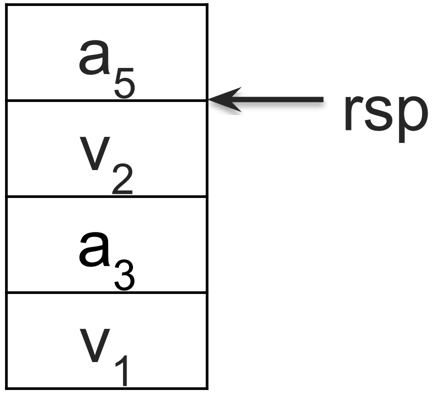
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

`Mem[v2] := v1`

Desired Logic



Stack

rax	v_1
rbx	v_2
rip	a_5

a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Implementing with Gadgets

`Mem[v2] := v1`

Desired Logic

Stack	
a_5	← rsp
v_2	
a_3	
v_1	

Stack

rax	v_1
rbx	v_2
rip	a_5

a_1: pop rax;
a_2: ret
} **Gadget 1**

a_3: pop rbx;
a_4: ret
} **Gadget 2**

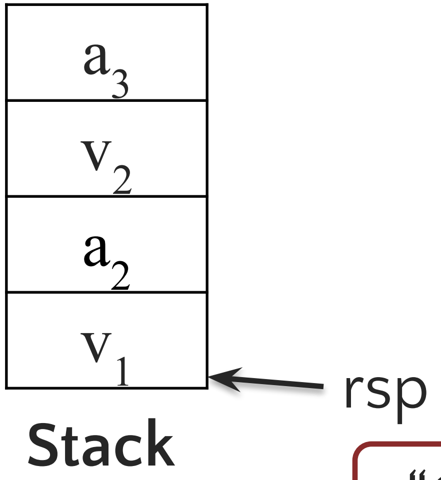
a_5: mov [rbx], rax

Implementation 2

Equivalence

```
Mem[v2] := v1
```

Desired Logic



semantically equivalent

"Gadgets"

Implementation 1		Implementation 2
a_1: `mov rax, [rsp]`	⟷	a_1: `pop rax; ret`
a_2: `mov rbx, [rsp+16]`	⟷	a_2: `pop rbx; ret`
a_3: `mov [rbx], rax`	⟷	a_3: `mov [rbx], rax`

Return-Oriented Programming (ROP)

`Mem[v2] := v1`

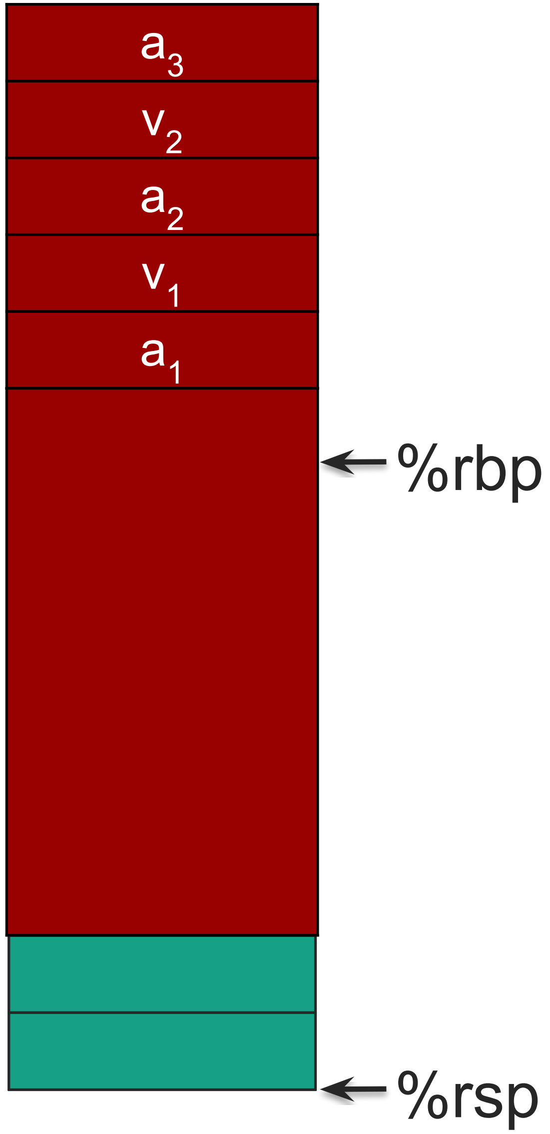
Desired *Shellcode*

a_1: `pop rax; ret`

a_2: `pop rbx; ret`

a_3: `mov [rbx], rax`

Desired store executed!



Gadgets

- A gadget is a set of instructions for carrying out a semantic action
  - mov, add, etc.

- Gadgets typically have a number of instructions
  - One instruction = native instruction set
  - More instructions = synthesize <– ROP

- Gadgets in ROP generally (but not always) end in return

ROP Intuition/Analogy

- In regular x64, RIP is instruction pointer
- In ROP, RSP is the effective instruction pointer
- In regular x64, assembly, instruction is "atomic" unit of execution
- In ROP, "gadget" is the atomic unit
- Think of ROP as a "weird" program written in an alternative "assembly language"

ROP Programming

1. Disassemble code
2. Identify ***useful*** code sequences as gadgets
3. Assemble gadgets into desired shellcode

Disassemble code

Compiler-created gadget: A sequence of instructions inserted by the compiler ending in `ret`.

Unintended gadget: A sequence of instructions not created by the compiler, e.g., by starting disassembly at an unaligned start.

Identify Useful Gadgets

Definition:

*A sequence of instructions is **useful***

– if it is a sequence of valid instructions ending in a ret instruction

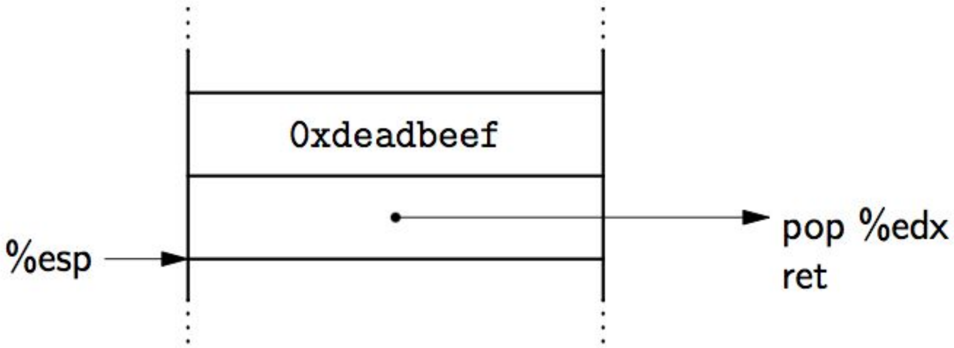
– none of the instructions causes the processor to transfer execution away without reaching the ret

Note: can be intended or unintended (alignment)

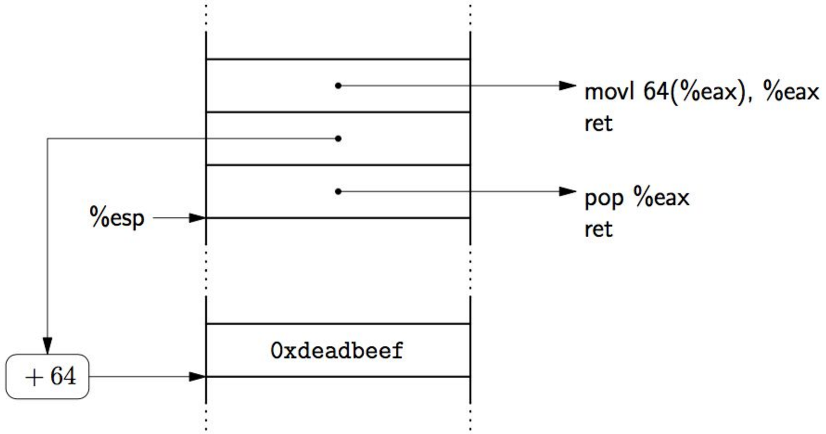
Useful ROP Gadgets

- Load/Store
- Arithmetic/Logic operations
- Control Flow
- System calls
- Function calls

Turing complete!



Gadget that loads a constant



Gadget that loads from memory

ROP Programming

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode

Finding Gadgets

- Active community has developed several tools for automatically identifying such gadgets

  https://github.com/JonathanSalwan/ROPgadget
  https://github.com/Ben-Lichtman/ropr
  https://scoding.de/ropper/

and many more!

ROP Probability of Success

Can call libc functions in 80% of programs greater than /bin/true (20KB)

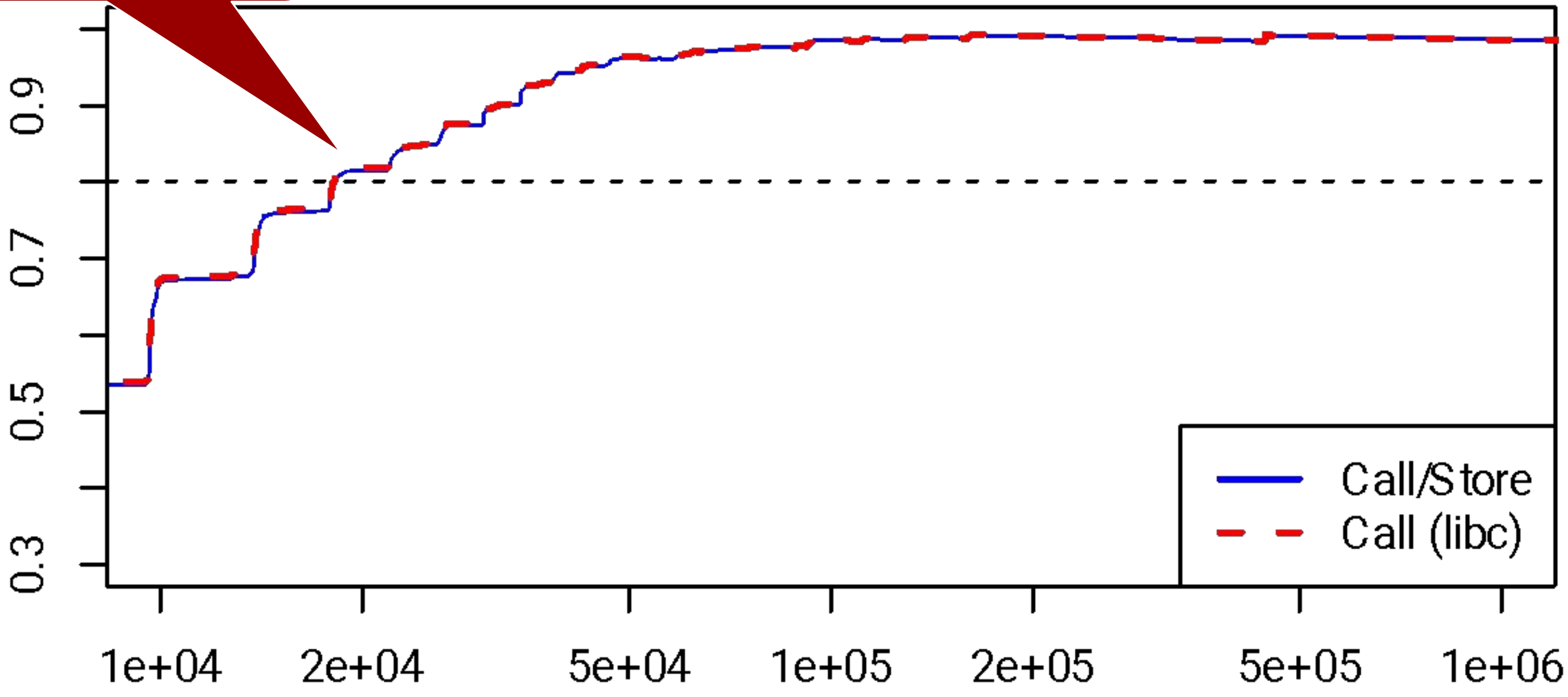


Figure taken from Q: Exploit Hardening Made Easy (a Compiler for ROP programs)

Quiz Question

Which of the following defenses complicates ROP attacks the ***MOST***?

A. Stack canaries
B. Data execution prevention
C. Fully applied ASLR (including .text)
D. Removing unneeded `system`-like functions from libc

Making our lives easier

- Reverse engineering tools
  - https://github.com/wtsxDev/reverse-engineering
- Exploitation libraries
  - https://github.com/Gallopsled/pwntools
- Mixed
  - https://github.com/pwndbg/pwndbg

Takeaways

- Control Flow Hijack:
  Control + Computation
- Buffer overflows overwrite return address
- Format string vulnerabilities
  - Read/write arbitrary memory
- Defenses
  - Canary, DEP, ASLR
  - Beatable using various clever tricks

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!