CS165 – Computer Security

Control Flow Defenses

Oct 14, 2021

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows ✓
- Format String Vulnerability ✓

What's new since 2000

NEXT

What's new since 2000?

Assigned Reading:

Smashing the stack in 2011
by Paul Makowski

http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/

A lot has happened…

- Heap-based buffer overflows also common
- [not mentioned] fortified source by static analysis (e.g., gcc can sometimes replace strcpy by strcpy_chk)

Additional materials:

- Canary (e.g. ProPolice in gcc)
- Data Execution Protection/No eXecute
- Address Space Layout Randomization

A lot has happened…

- Heap-based buffer overflows also common
- [not mentioned] fortified source by static analysis (e.g., gcc can sometimes replace strcpy by strcpy_chk)

Additional materials:

- Canary (e.g. ProPolice in gcc)
- Data Execution Protection/No eXecute
- Address Space Layout Randomization

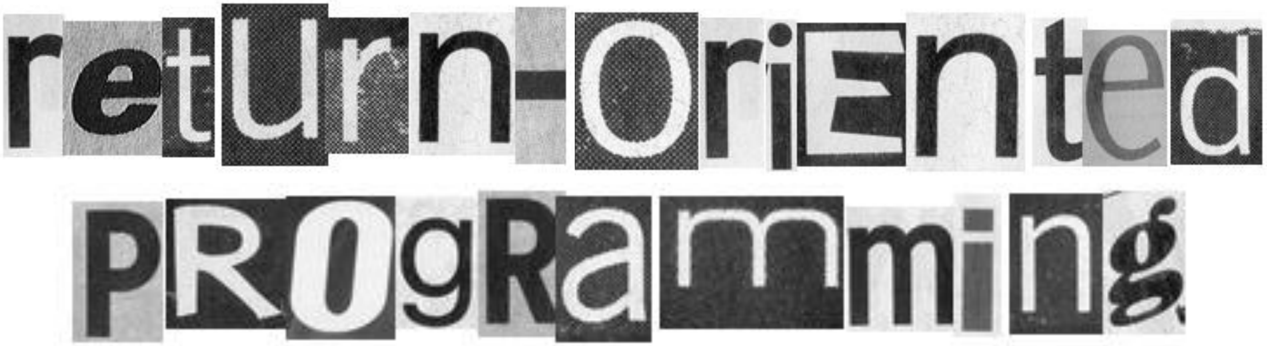
```
alias gcc732='gcc -m32 -g3 -O1 -fverbose-asm -fno-omit-frame-pointer
-mpreferred-stack-boundary=2 -fno-stack-protector -fno-pie -fno-PIC
-D_FORTIFY_SOURCE=0'
```

But little has changed…

Method to gain entry remains the same

- buffer overflows
- format strings

What's different is shellcode:



Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows ✓
- Format String Vulnerability ✓

What’s new since 2000 ✓

Reading list

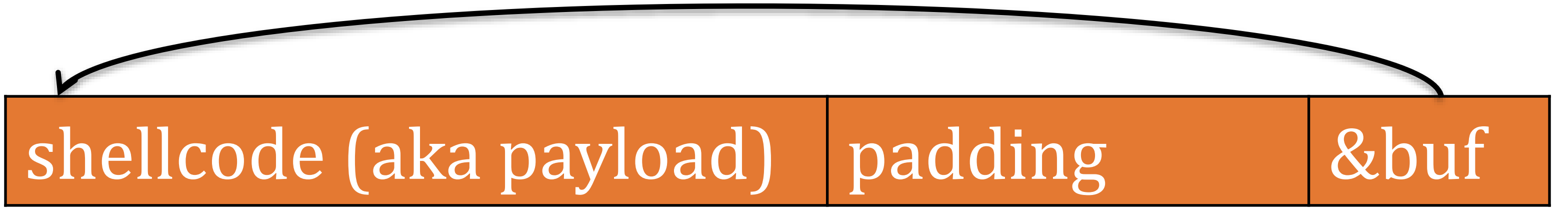
- “Smashing The Stack For Fun And Profit”
  - http://www.phrack.org/issues.html?issue=49&id=14#article

- “Exploit the format string vulnerabilities”
  - http://www.utdallas.edu/~zhiqiang.lin/file/formatstring.pdf

- Smashing the Stack in 2011
  - http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/

Control flow hijack defenses

Control Flow Hijack: Always control + computation



computation + ***control***

- code injection
- return-to-libc
- heap metadata overwrite
- return-oriented programming
- ...

Same principle, different mechanism

Control Flow Hijacks

… happen when an attacker gains control of

Control Flow Hijacks

… happen when an attacker gains control of ***the instruction pointer****.*

Two common hijack methods:

- buffer overflows
- format string attacks

Control Flow Hijack Defenses

Control Flow Hijack Defenses

Bugs are the root cause of hijacks (hard/costly)!

- Find bugs with analysis tools
- Prove program correctness

Control Flow Hijack Defenses

Bugs are the root cause of hijacks (hard/costly)!

- Find bugs with analysis tools
- Prove program correctness

Mitigation Techniques (simple/cheap):

- Canaries
- Data Execution Prevention/No eXecute
- Address Space Layout Randomization

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 to easy-to-bypass

* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

Agenda

Canary / Stack Cookies

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

Address Space Layout Randomization (ASLR)

Agenda

Canary / Stack Cookies

← NEXT

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

Address Space Layout Randomization (ASLR)

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.”

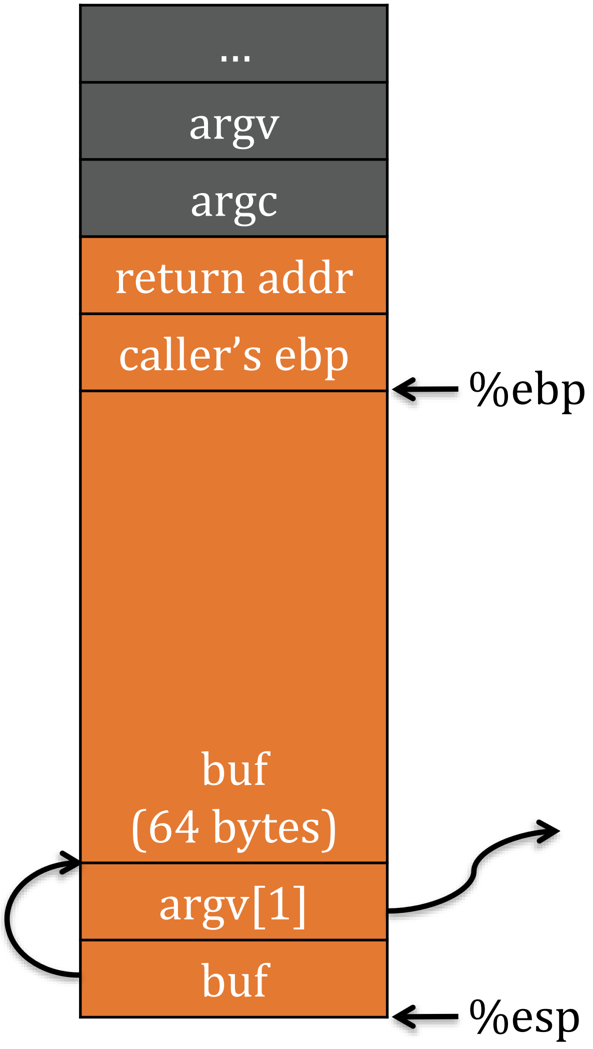
Canary / Stack Cookies

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

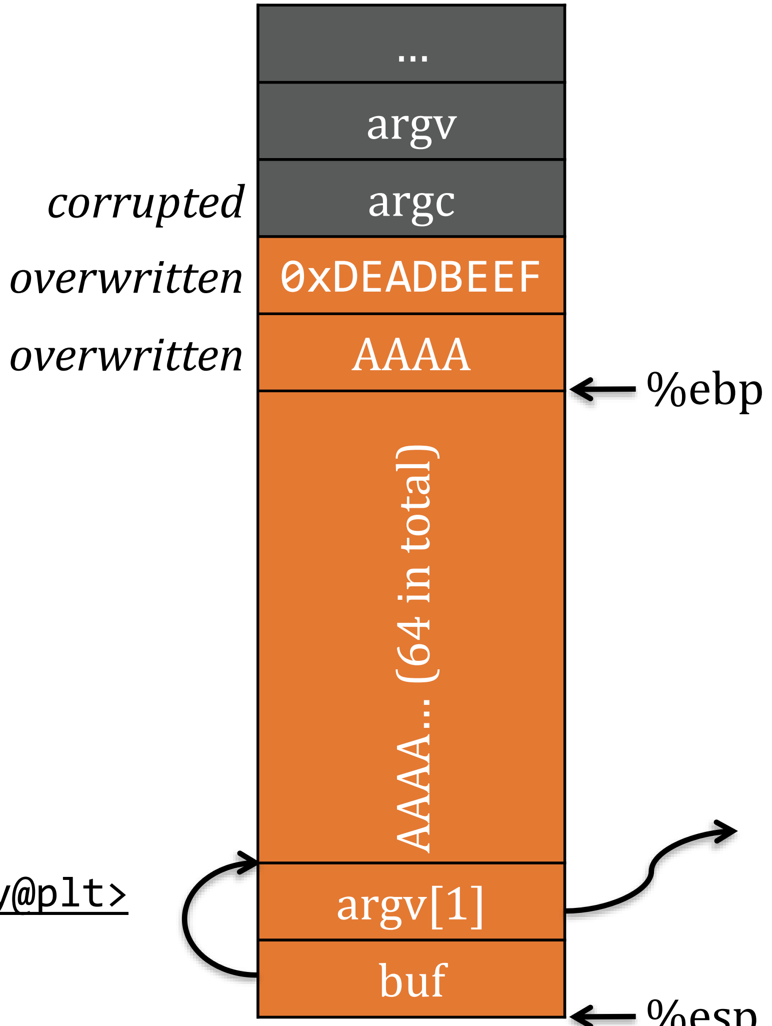
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

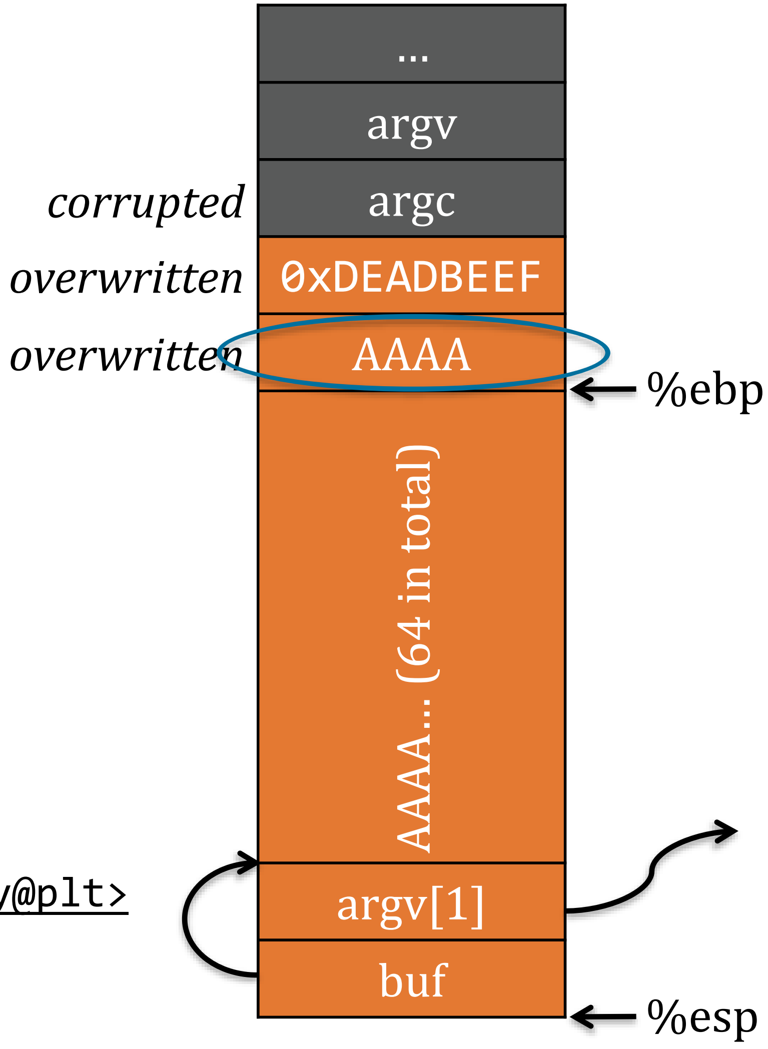
```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

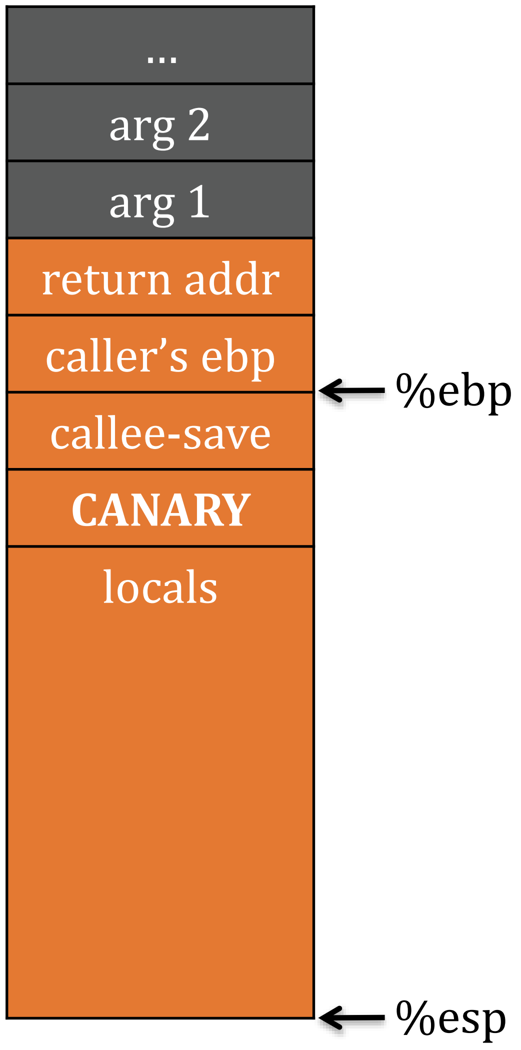
```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



StackGuard [Cowen etal. 1998]

Idea:

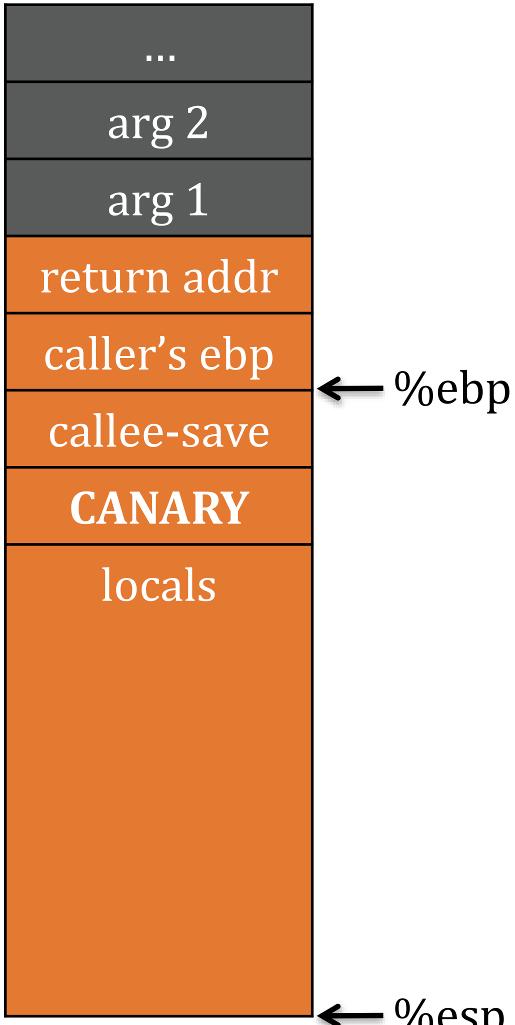
- prologue introduces a ***canary word*** between return addr and locals



StackGuard [Cowen etal. 1998]

Idea:

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

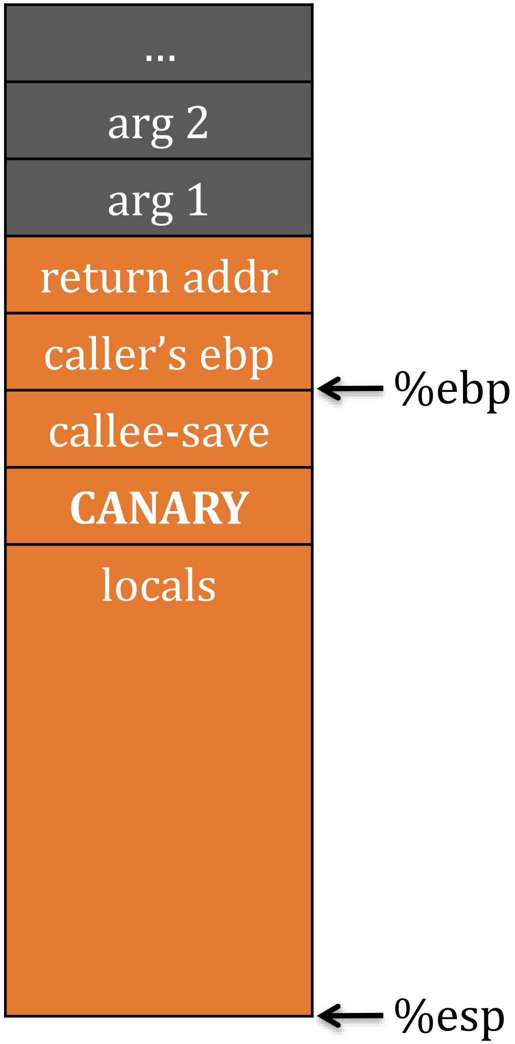


StackGuard [Cowen etal. 1998]

Idea:

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

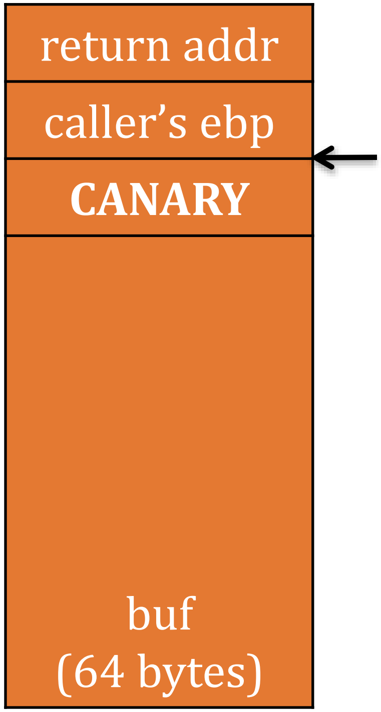
Wrong Canary => Overflow



gcc Stack-Smashing Protector (ProPolice)

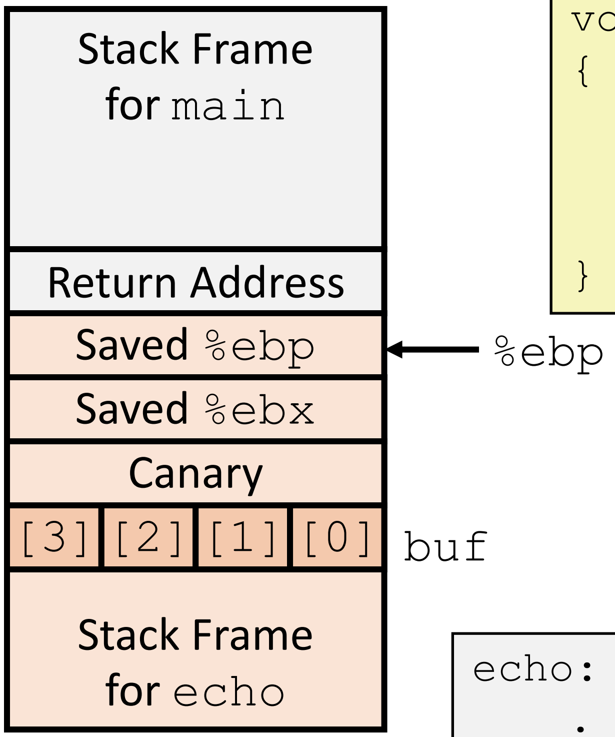
Compiled with v4.6.1:
`gcc -fstack-protector -O1 …`

```
Dump of assembler code for function main:
   0x08048440 <+0>:  push   %ebp
   0x08048441 <+1>:  mov    %esp,%ebp
   0x08048443 <+3>:  sub    $76,%esp
   0x08048446 <+6>:  mov    %gs:20,%eax
   0x0804844c <+12>: mov    %eax,-4(%ebp)
   0x0804844f <+15>: xor    %eax,%eax
   0x08048451 <+17>: mov    12(%ebp),%eax
   0x08048454 <+20>: mov    4(%eax),%eax
   0x08048457 <+23>: mov    %eax,4(%esp)
   0x0804845b <+27>: lea    -68(%ebp),%eax
   0x0804845e <+30>: mov    %eax,(%esp)
   0x08048461 <+33>: call   0x8048350 <strcpy@plt>
   0x08048466 <+38>: mov    -4(%ebp),%edx
   0x08048469 <+41>: xor    %gs:20,%edx
   0x08048470 <+48>: je     0x8048477 <main+55>
   0x08048472 <+50>: call   0x8048340 <__stack_chk_fail@plt>
   0x08048477 <+55>: leave
   0x08048478 <+56>: ret
```



Setting Up Canary

Before call to gets

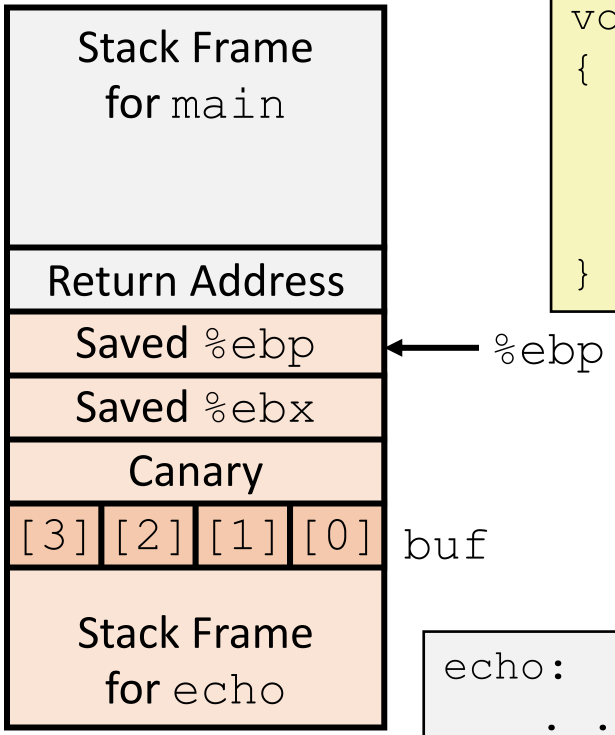


```
/* Echo Line */
void echo()
{
     char buf[4];  /* Way too small! */
     gets(buf);
     puts(buf);
}
```

```
echo:
    . . .
    movl     %gs:20, %eax      # Get canary
    movl     %eax, -8(%ebp)    # Put on stack
    xorl     %eax, %eax        # Erase canary
    . . .
```

Checking Canary

Before call to gets

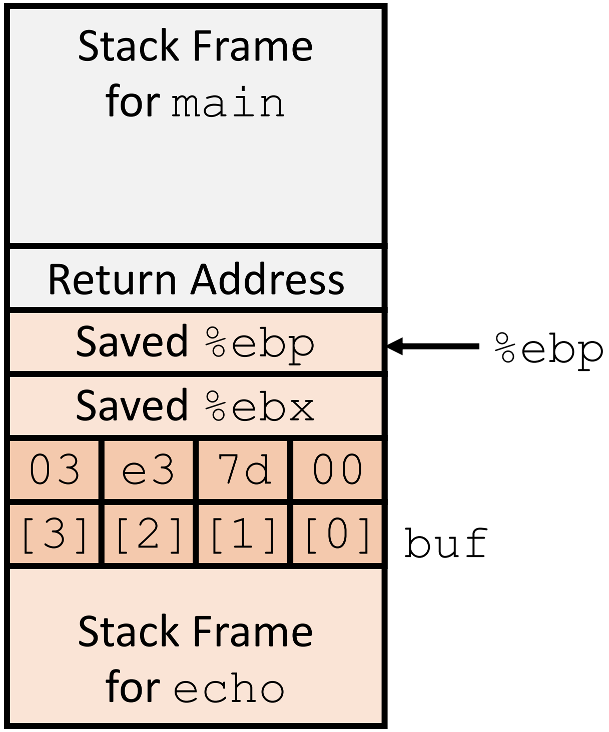


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movl     -8(%ebp), %eax    # Retrieve from stack
    xorl     %gs:20, %eax      # Compare with Canary
    je       .L24              # Same: skip ahead
    call     __stack_chk_fail  # ERROR
.L24:
    . . .
```

Canary Example

Before call to gets



Stack Frame for main

Return Address

Saved %ebp

Saved %ebx

03 e3 7d 00

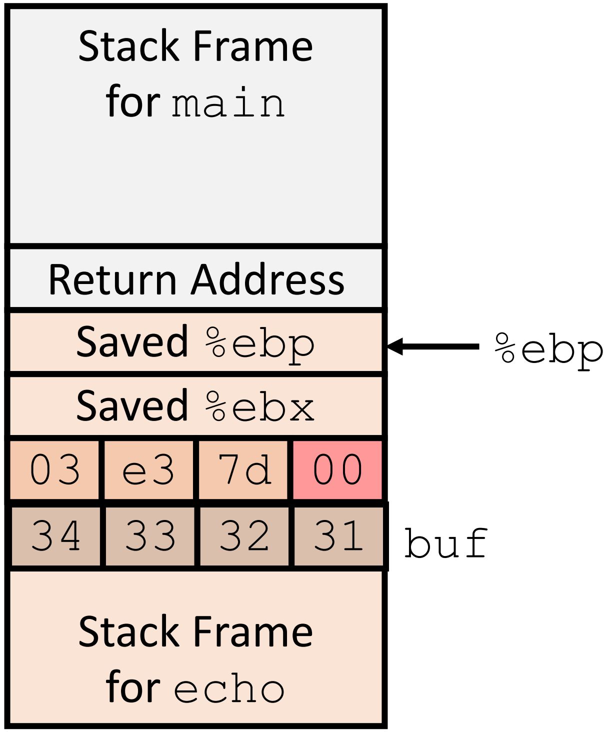
[3] [2] [1] [0]

%ebp

buf

Stack Frame for echo

Input 1234



Stack Frame for main

Return Address

Saved %ebp

Saved %ebx

03 e3 7d 00

34 33 32 31

%ebp

buf

Stack Frame for echo

```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption!
(allows programmers to make silent off-by-one errors)

Canary should be **HARD** to Forge

- Random Canary
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness

Canary Scorecard

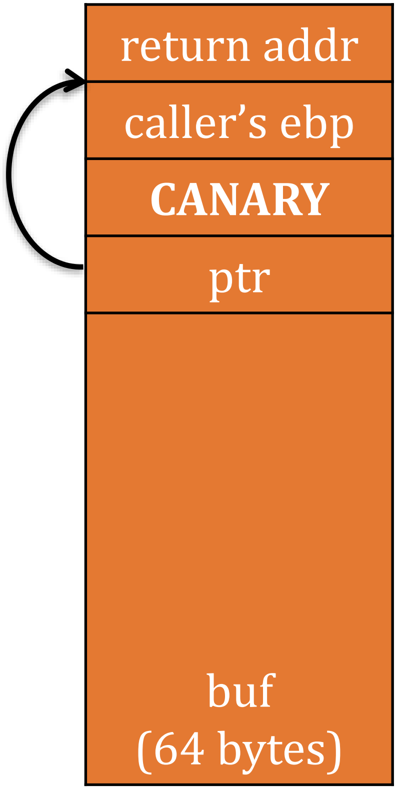
Aspect	**Canary**
Performance	• several instructions per function • time: a few percent on average • size: can optimize away in safe functions (but see MS08-067 *)
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *not really…*

* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*…

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



Canary Weakness

Check does ***not*** happen until epilogue…

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

Code Examples:

http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx

VS 2003: /GS

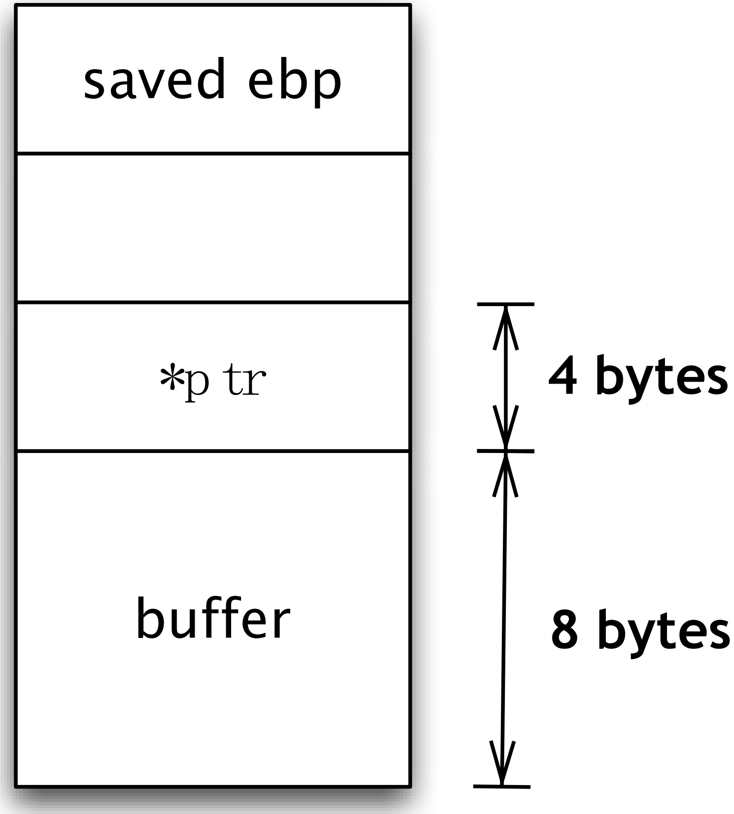
Function Pointer Subterfuge

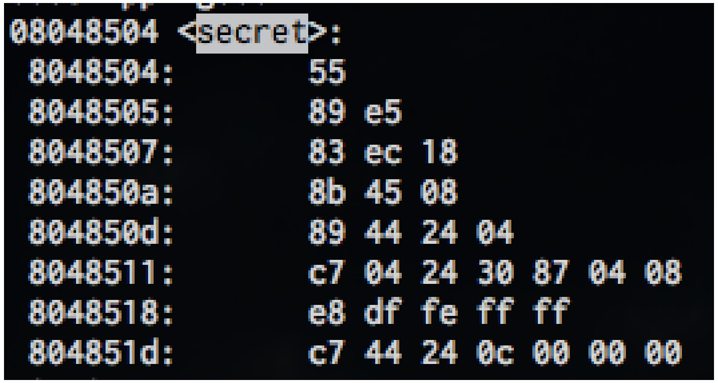
Overwrite a function pointer to point to:

- program function (similar to ret2text)
- Other non-randomized functions

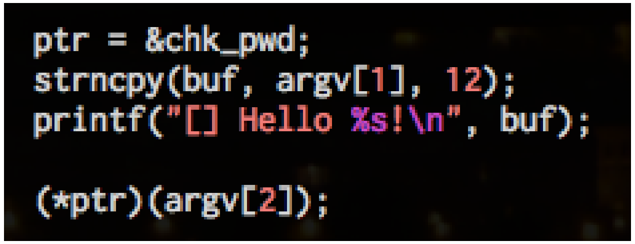
```
/*please call me!*/
int secret(char *input) { … }

int chk_pwd(char *input) { … }

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]);
}
```

Function Pointers



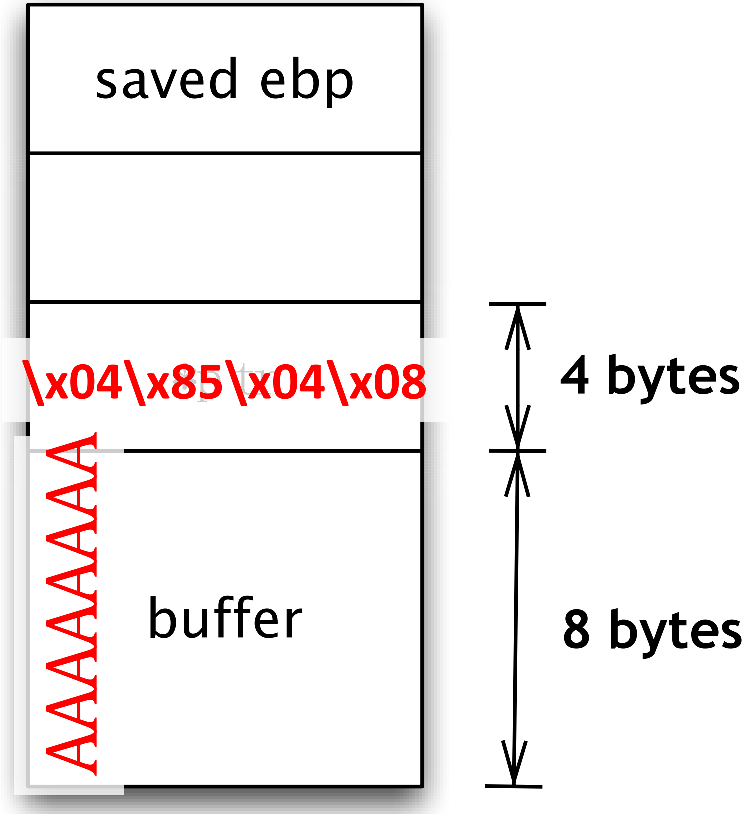


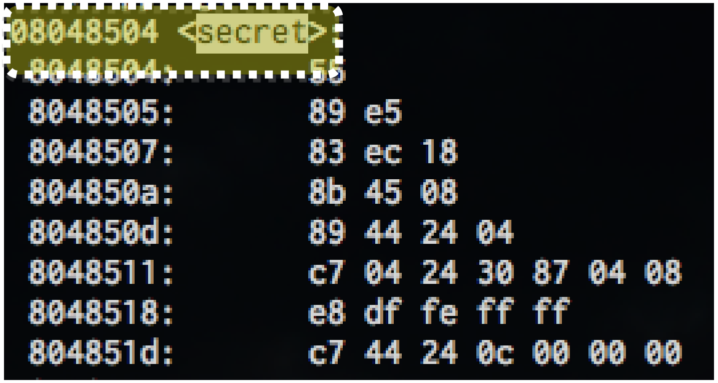
```
08048504 <secret>:
 8048504:       55
 8048505:       89 e5
 8048507:       83 ec 18
 804850a:       8b 45 08
 804850d:       89 44 24 04
 8048511:       c7 04 24 30 87 04 08
 8048518:       e8 df fe ff ff
 804851d:       c7 44 24 0c 00 00 00
```



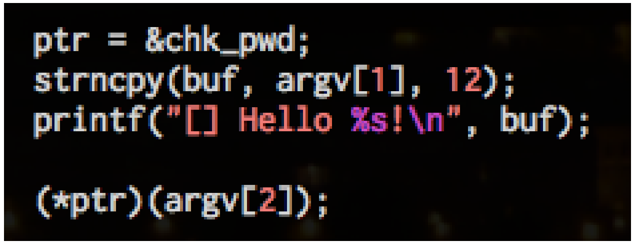
```
ptr = &chk_pwd;
strncpy(buf, argv[1], 12);
printf("[] Hello %s!\n", buf);

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

Function Pointers





```
08048504 <secret>:
 8048504:       55
 8048505:       89 e5
 8048507:       83 ec 18
 804850a:       8b 45 08
 804850d:       89 44 24 04
 8048511:       c7 04 24 30 87 04 08
 8048518:       e8 df fe ff ff
 804851d:       c7 44 24 0c 00 00 00
```



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

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

Canary Weakness

Check does ***not*** happen until epilogue…

- func ptr subterfuge } PointGuard
- C++ vtable hijack
- exception handler hijack } SafeSEH SEHOP
- …

} ProPolice puts arrays above others *when possible*

Code Examples:
http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx

VS 2003: /GS

Canary Weakness

Check does ***not*** happen until epilogue…

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

PointGuard

ProPolice puts arrays above others *when possible*

SafeSEH
SEHOP

`struct` is fixed;
& what about heap?

Code Examples:
http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx

VS 2003: /GS

Agenda

Canary / Stack Cookies ✓

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

Address Space Layout Randomization (ASLR)

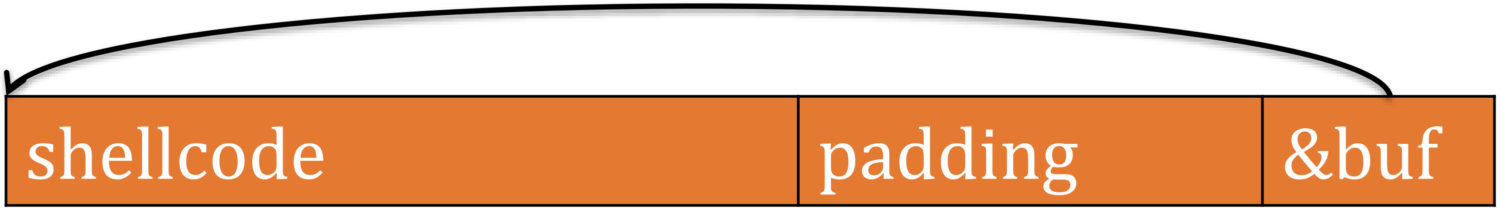
Agenda

Canary / Stack Cookies ✓

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

Address Space Layout Randomization (ASLR)

How to defeat exploits?



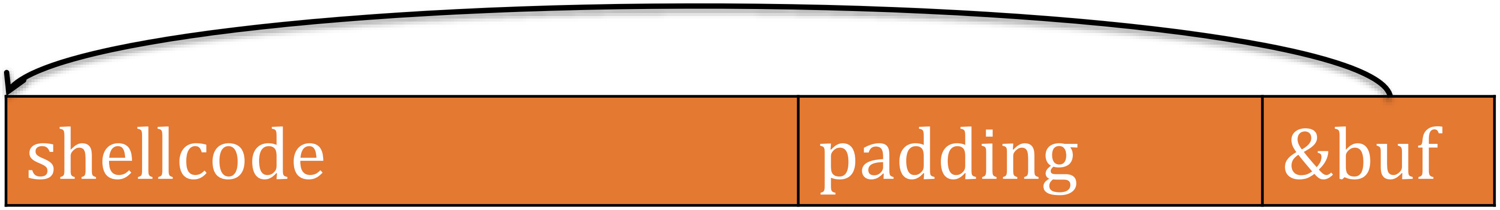
shellcode

padding

&buf

computation + ***control***

How to defeat exploits?



shellcode

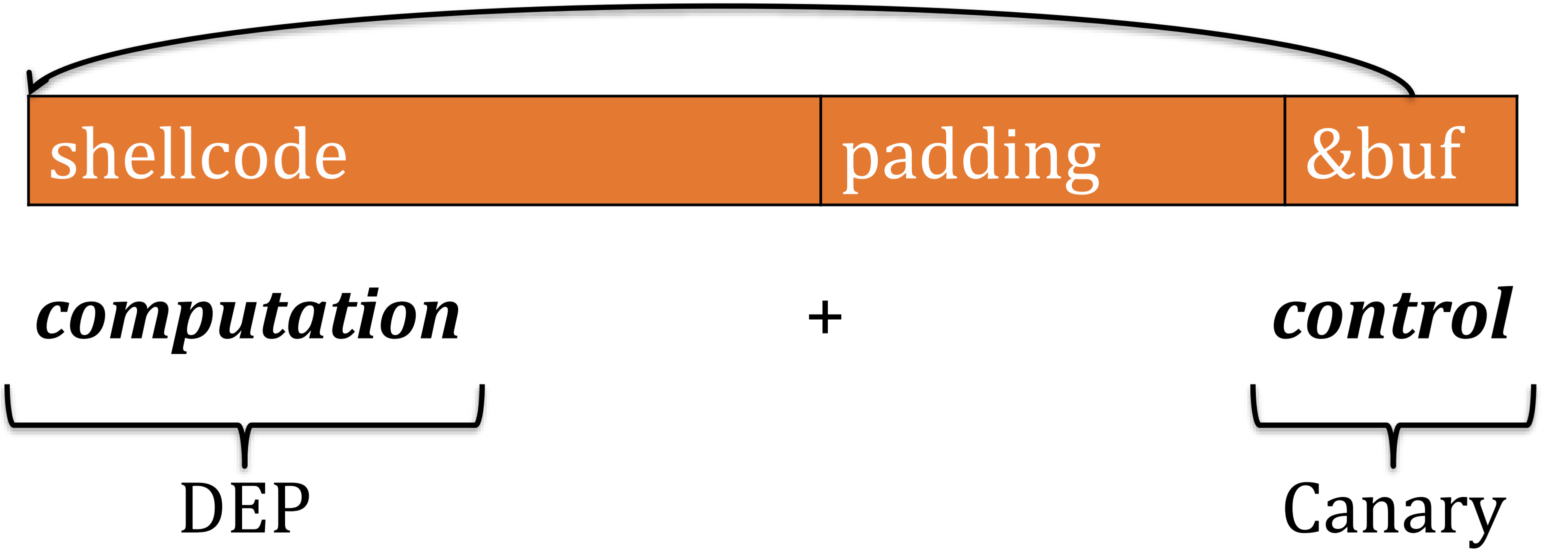
padding

&buf

computation + ***control***

Canary

How to defeat exploits?

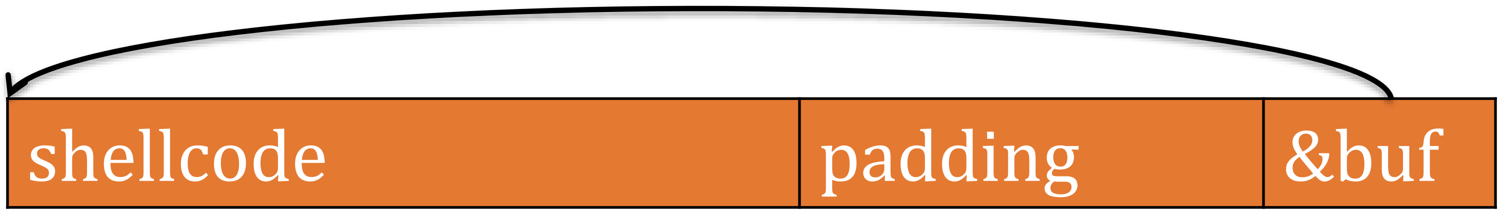


shellcode | padding | &buf

computation + ***control***

DEP Canary

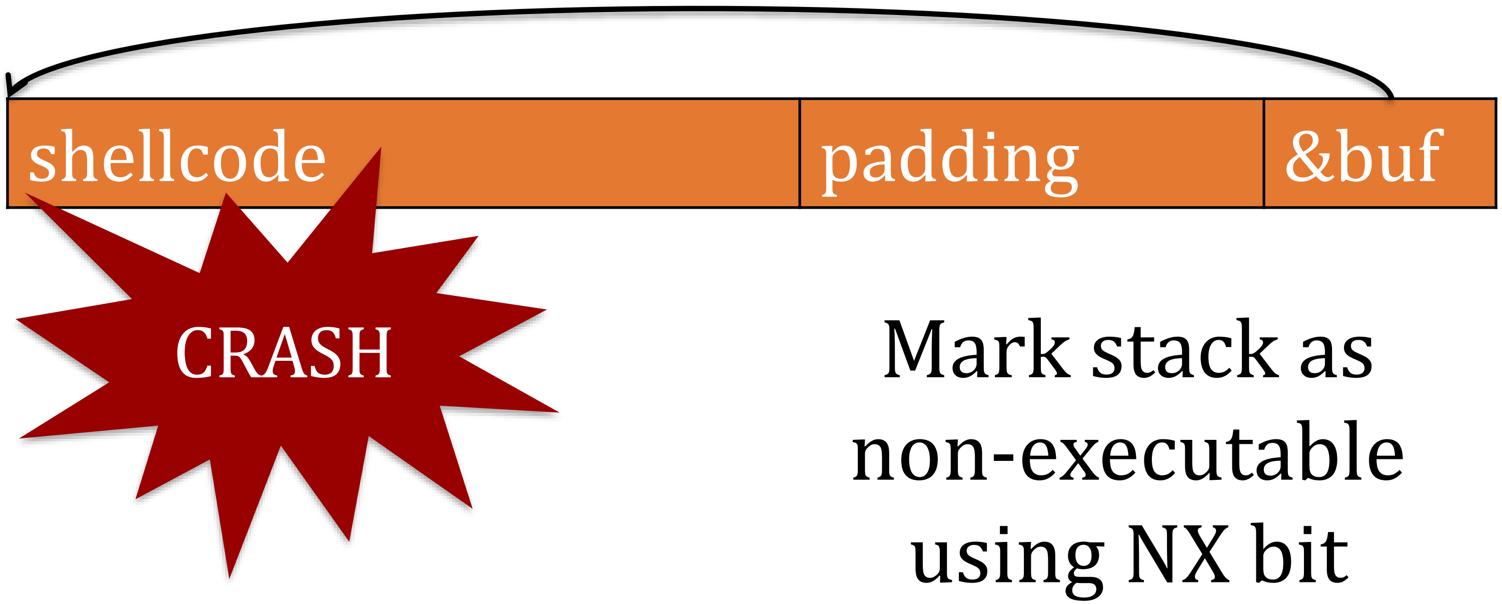
Data Execution Prevention



shellcode padding &buf

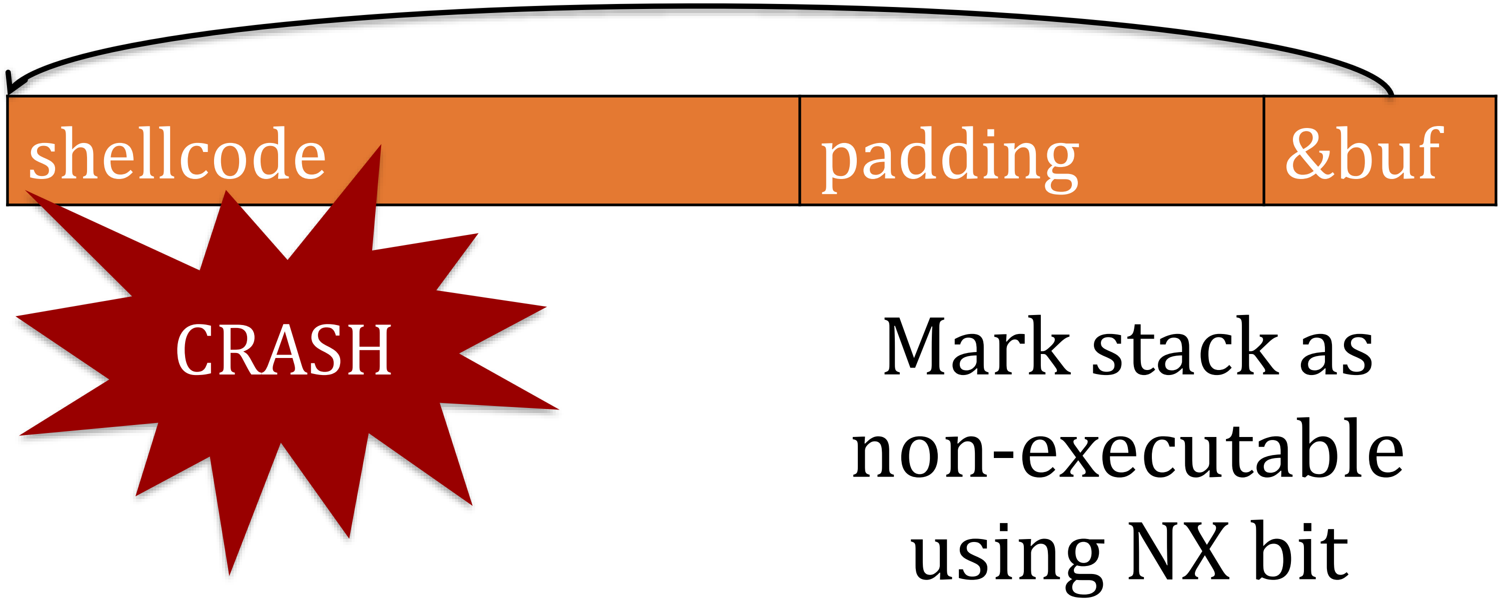
Mark stack as
non-executable
using NX bit

Data Execution Prevention



Mark stack as
non-executable
using NX bit

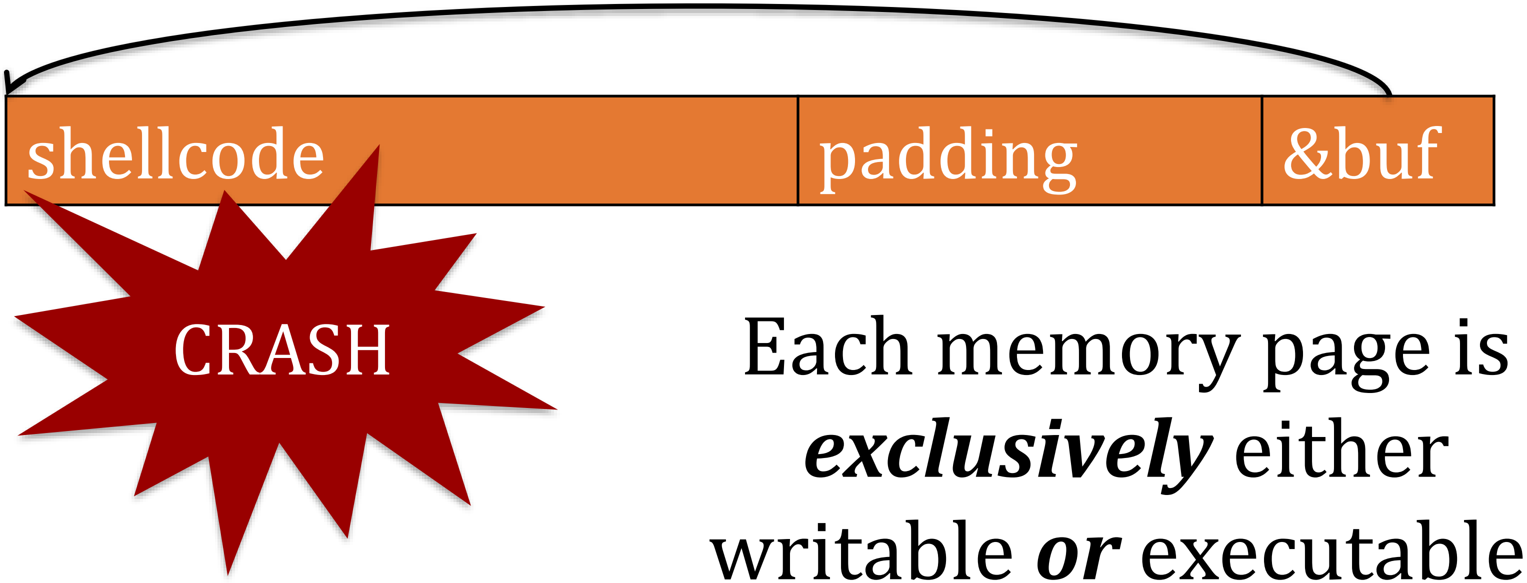
Data Execution Prevention



Mark stack as non-executable using NX bit

(still a Denial-of-Service attack!)

W ^ X



shellcode

padding

&buf

CRASH

Each memory page is ***exclusively*** either writable ***or*** executable.

(still a Denial-of-Service attack!)

DEP Scorecard

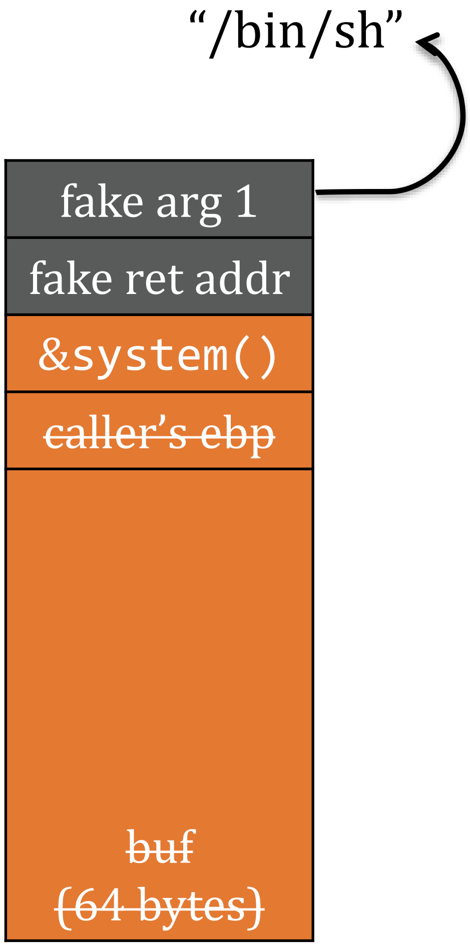
Aspect	Data Execution Prevention
Performance	• with hardware support: no impact • otherwise: reported to be <1% in PaX
Deployment	• kernel support (common on all platforms) • modules opt-in (now enabled by default)
Compatibility	• can break legitimate programs - Just-In-Time compilers - unpackers
Safety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

Return-to-libc Attack

Overwrite return address by address of a libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!



Reading:
The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), CCS 2007

More to come later

Agenda

Canary / Stack Cookies ✓

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

Address Space Layout Randomization (ASLR)

Agenda

Canary / Stack Cookies ✓

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

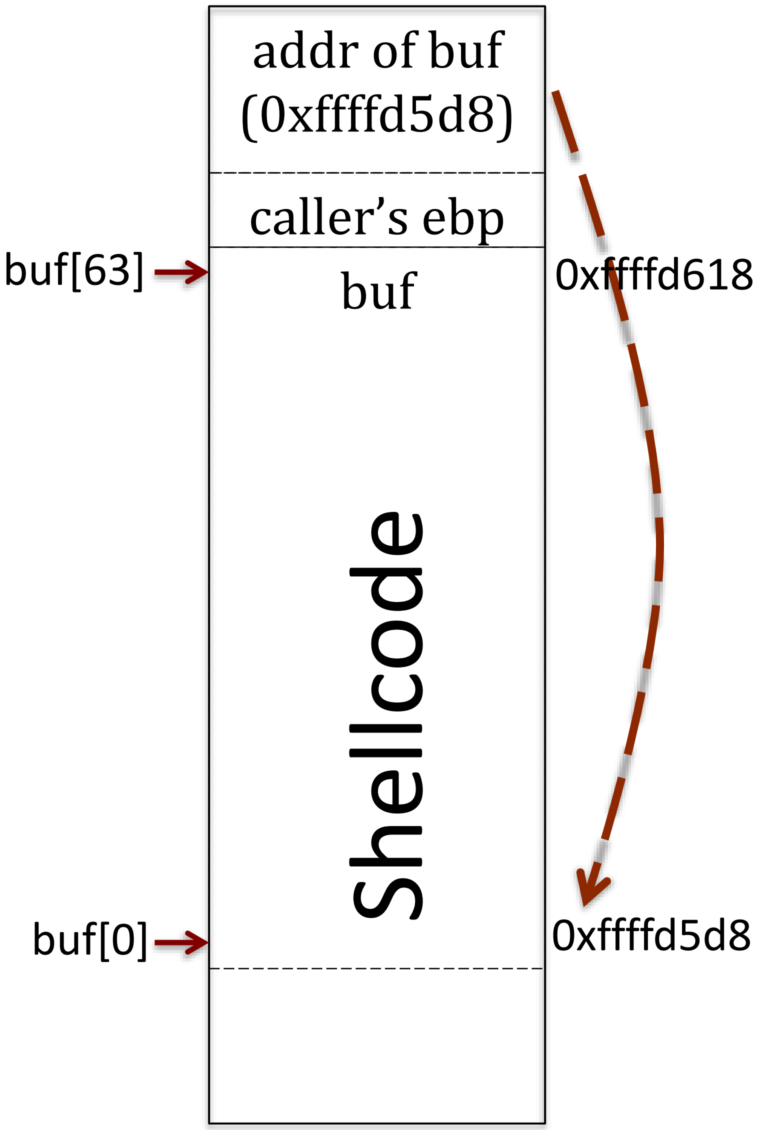
Address Space Layout Randomization (ASLR) ← NEXT

Address Space Layout Randomization (ASLR)

Assigned Reading:

ASLR Smack and Laugh Reference
by Tilo Muller

http://www.cs.ucr.edu/~zhiyunq/teaching/cs165/resources/paper/aslr_smack.pdf



addr of buf
(0xffffd5d8)

caller's ebp

buf[63]

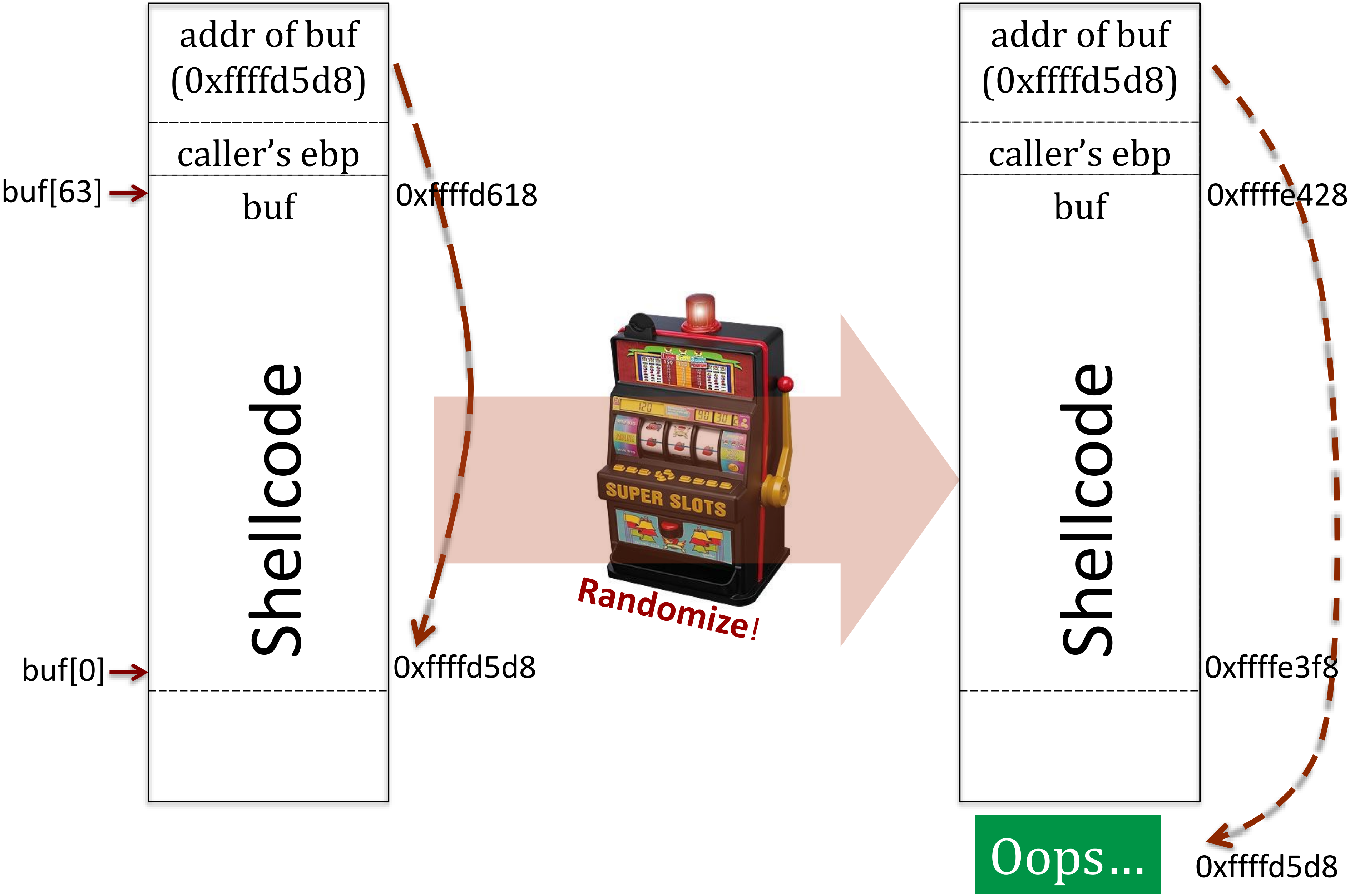
buf

0xffffd618

Shellcode

buf[0]

0xffffd5d8



addr of buf (0xffffd5d8)

caller's ebp

buf

Shellcode

buf[63]

buf[0]

0xffffd618

0xffffd5d8

Randomize!

addr of buf (0xffffd5d8)

caller's ebp

buf

Shellcode

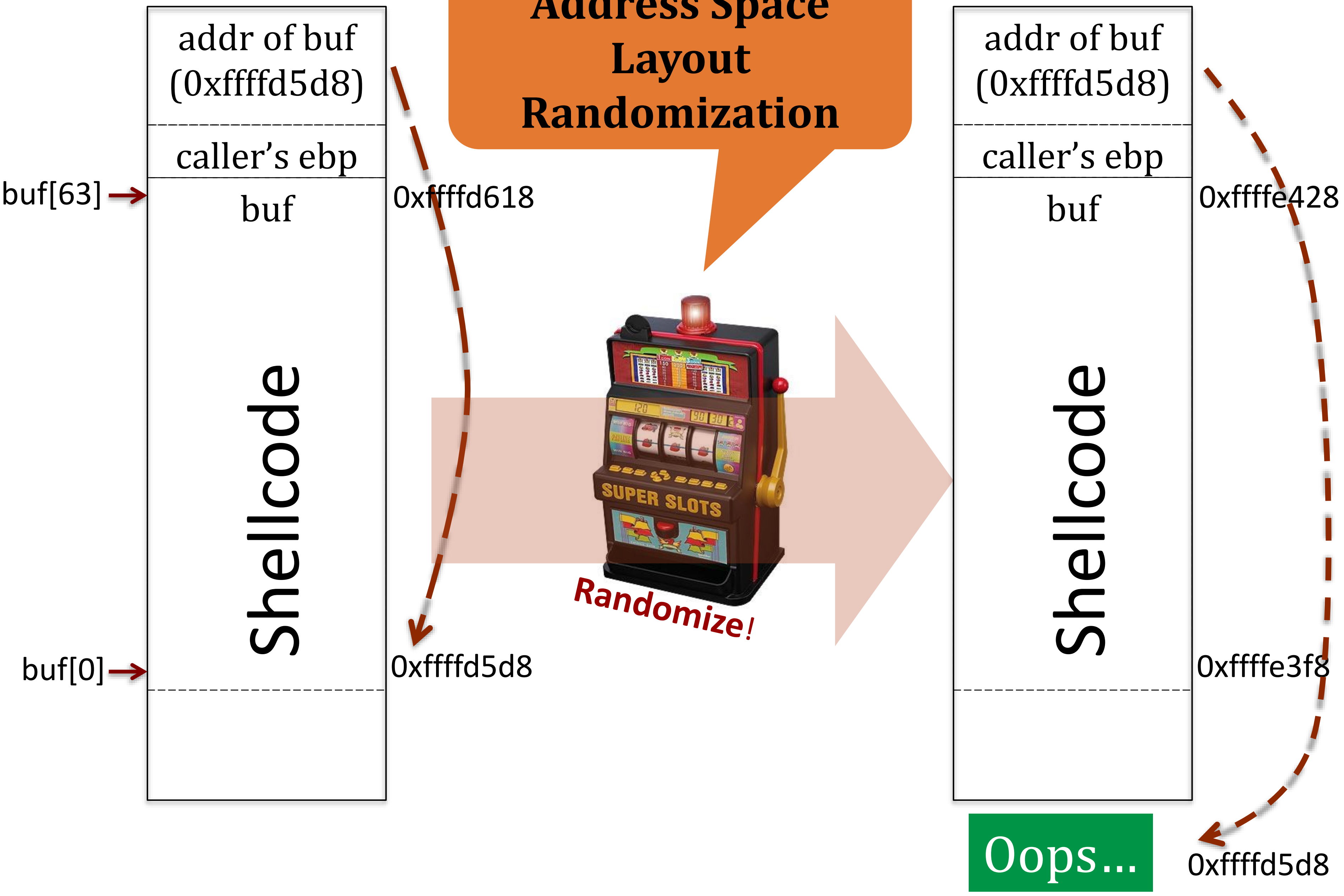
0xffffe428

0xffffe3f8

Oops…

0xffffd5d8

Address Space Layout Randomization



addr of buf (0xffffd5d8)

caller's ebp

buf

Shellcode

buf[63]

buf[0]

0xffffd618

0xffffd5d8

Randomize!

addr of buf (0xffffd5d8)

caller's ebp

buf

Shellcode

0xffffe428

0xffffe3f8

Oops…

0xffffd5d8

ASLR

Traditional exploits need precise addresses

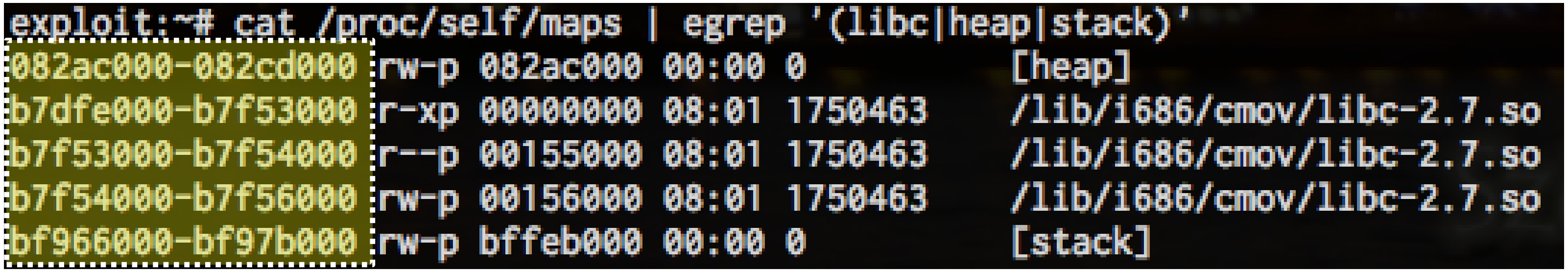
- *stack-based overflows:* location of shell code
- *return-to-libc:* library addresses

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

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

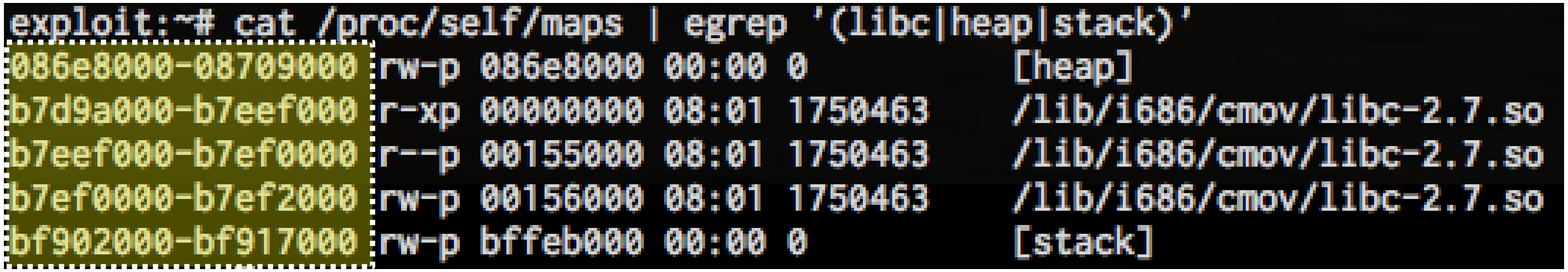
Running cat Twice

- Run 1



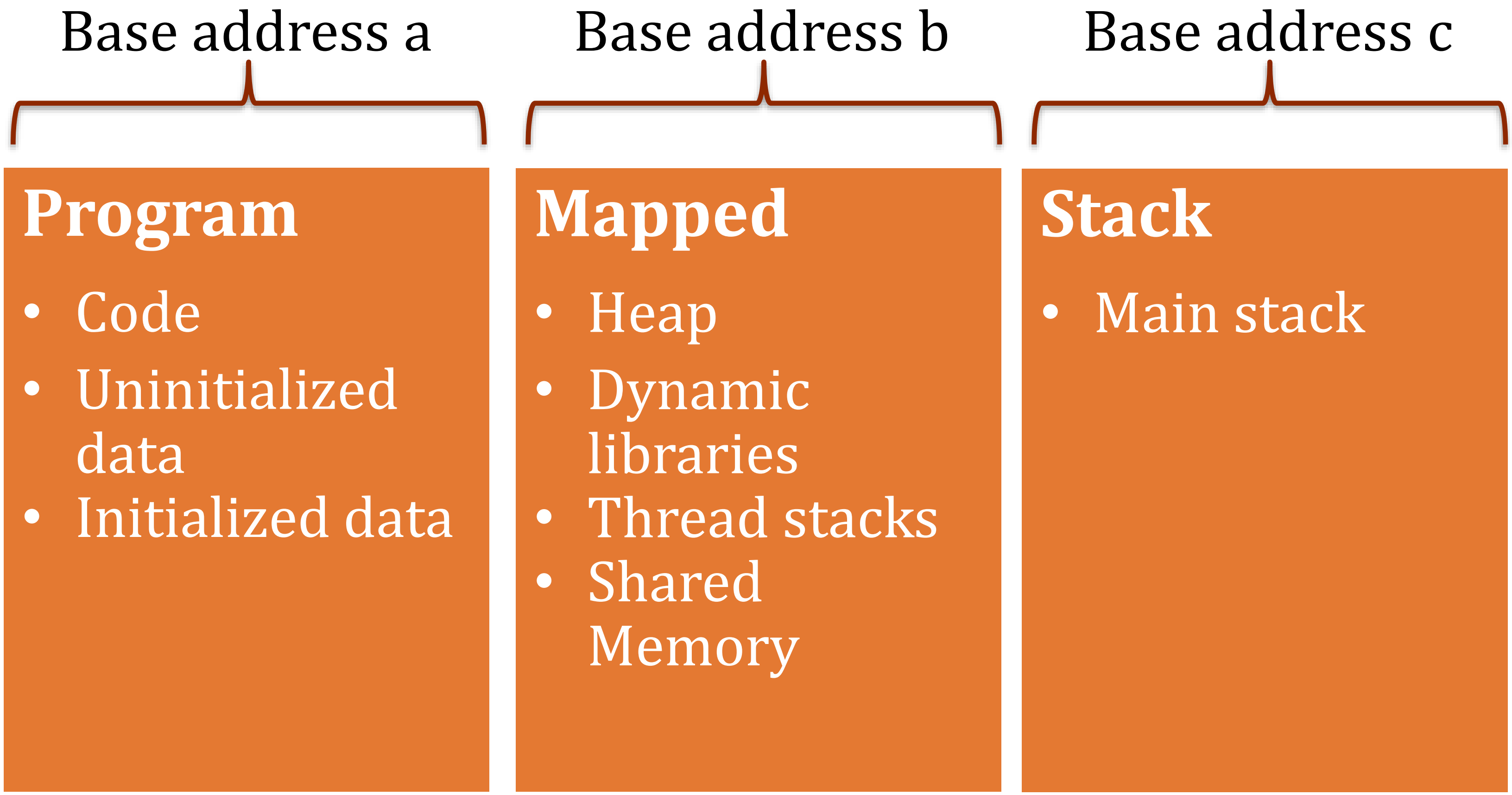
```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000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

Program

- Code
- Uninitialized data
- Initialized data

Base address b

Mapped

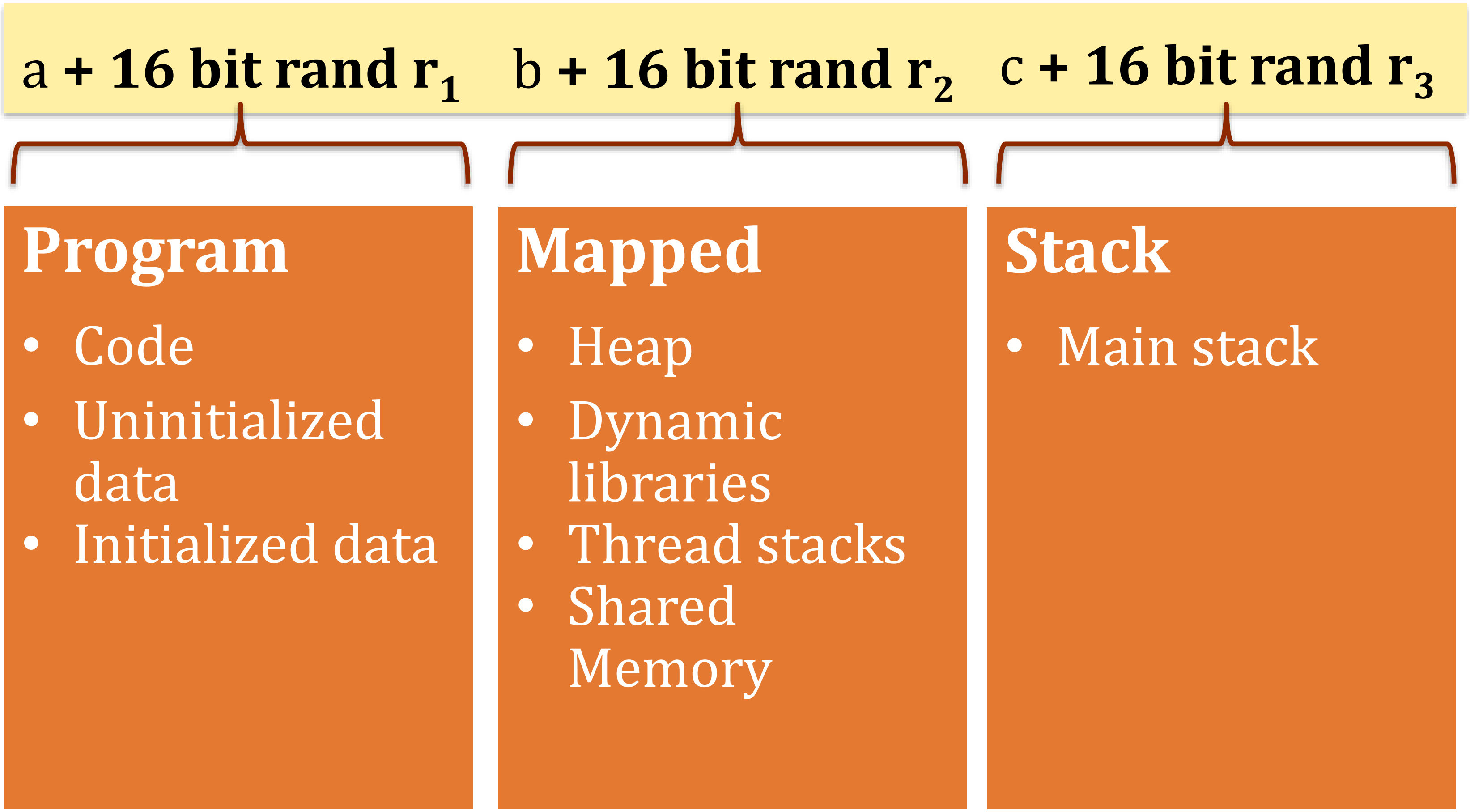
- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Base address c

Stack

- Main stack

ASLR Randomization



a **+ 16 bit rand r_1** b **+ 16 bit rand r_2** c **+ 16 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 (up to 32) on 64-bit systems.

ASLR Scorecard

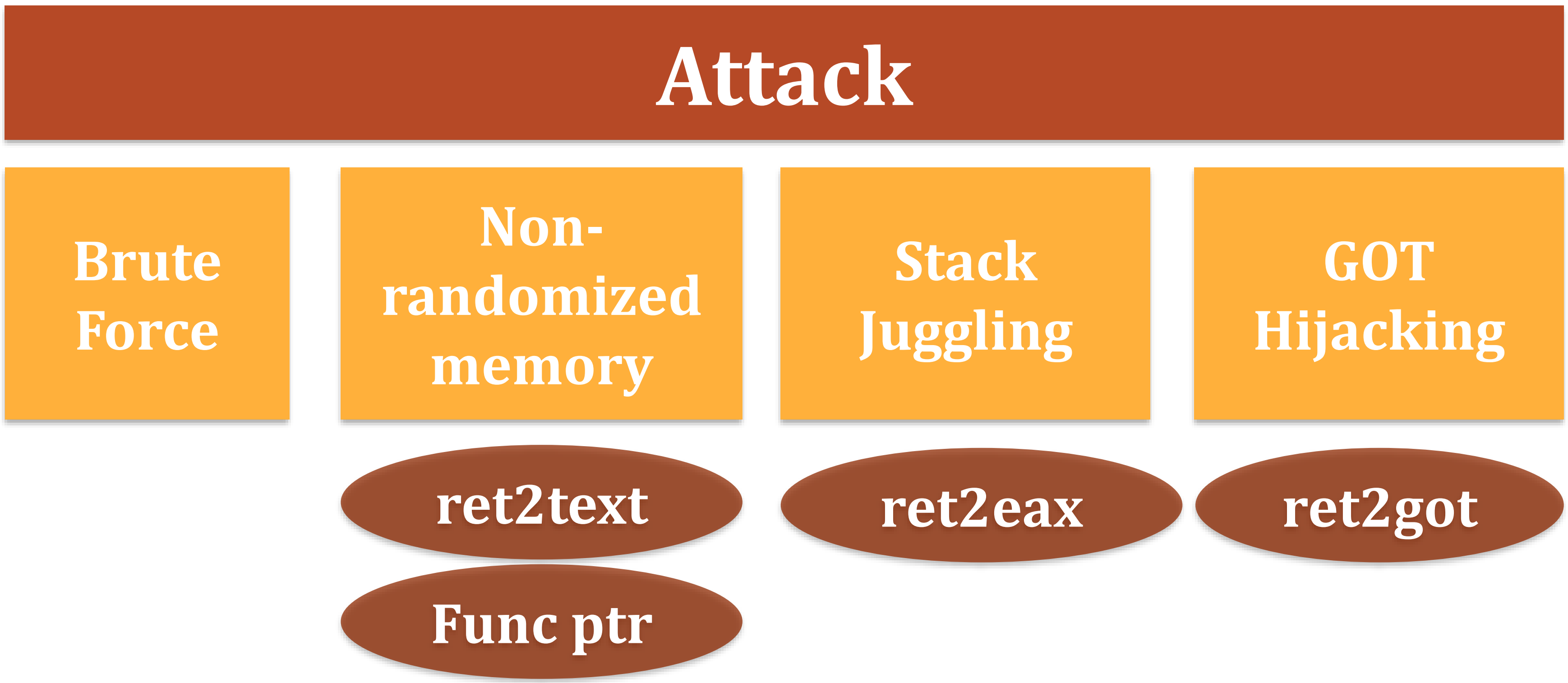
Aspect	Address Space Layout Randomization
Performance	• excellent—randomize once at load time
Deployment	• turn on kernel support (Windows: opt-in per module, but system override exists) • no recompilation necessary
Compatibility	• transparent to safe apps (position independent)
Safety Guarantee	• not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

Ubuntu - ASLR

- ASLR is **ON** by default [Ubuntu-Security]
  - cat /proc/sys/kernel/randomize_va_space
    - Prior to Ubuntu 8.10: **1** *(stack/mmap ASLR)*
    - In later releases: **2** *(stack/mmap/brk ASLR)*

  - stack/mmap ASLR: since kernel 2.6.15 (Ubuntu 6.06)
  - brk ASLR: since kernel 2.6.26 (Ubuntu 8.10)
  - exec ASLR: since kernel 2.6.25
    - Position Independent Executable (PIE) with "-fPIE –pie"

How to attack with ASLR?



Attack

Brute Force

Non-randomized memory

Stack Juggling

GOT Hijacking

ret2text

Func ptr

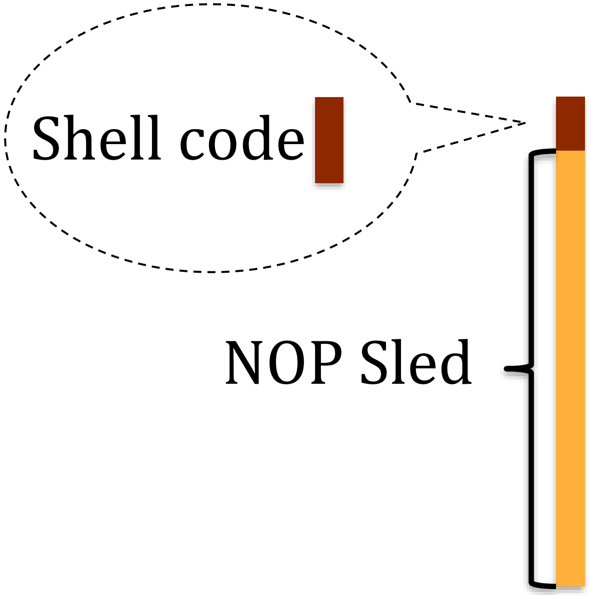
ret2eax

ret2got

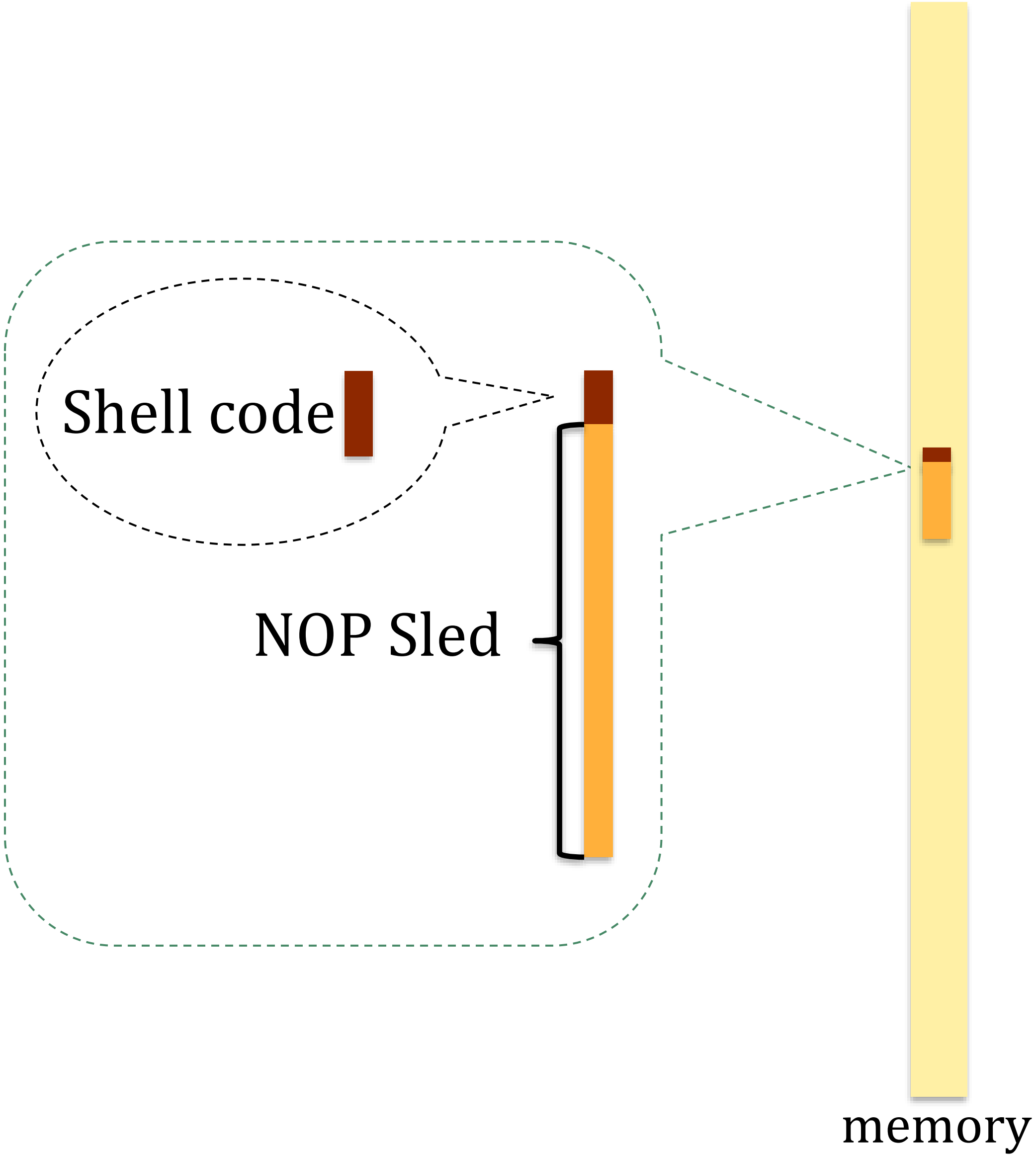
Brute Force

Shell code

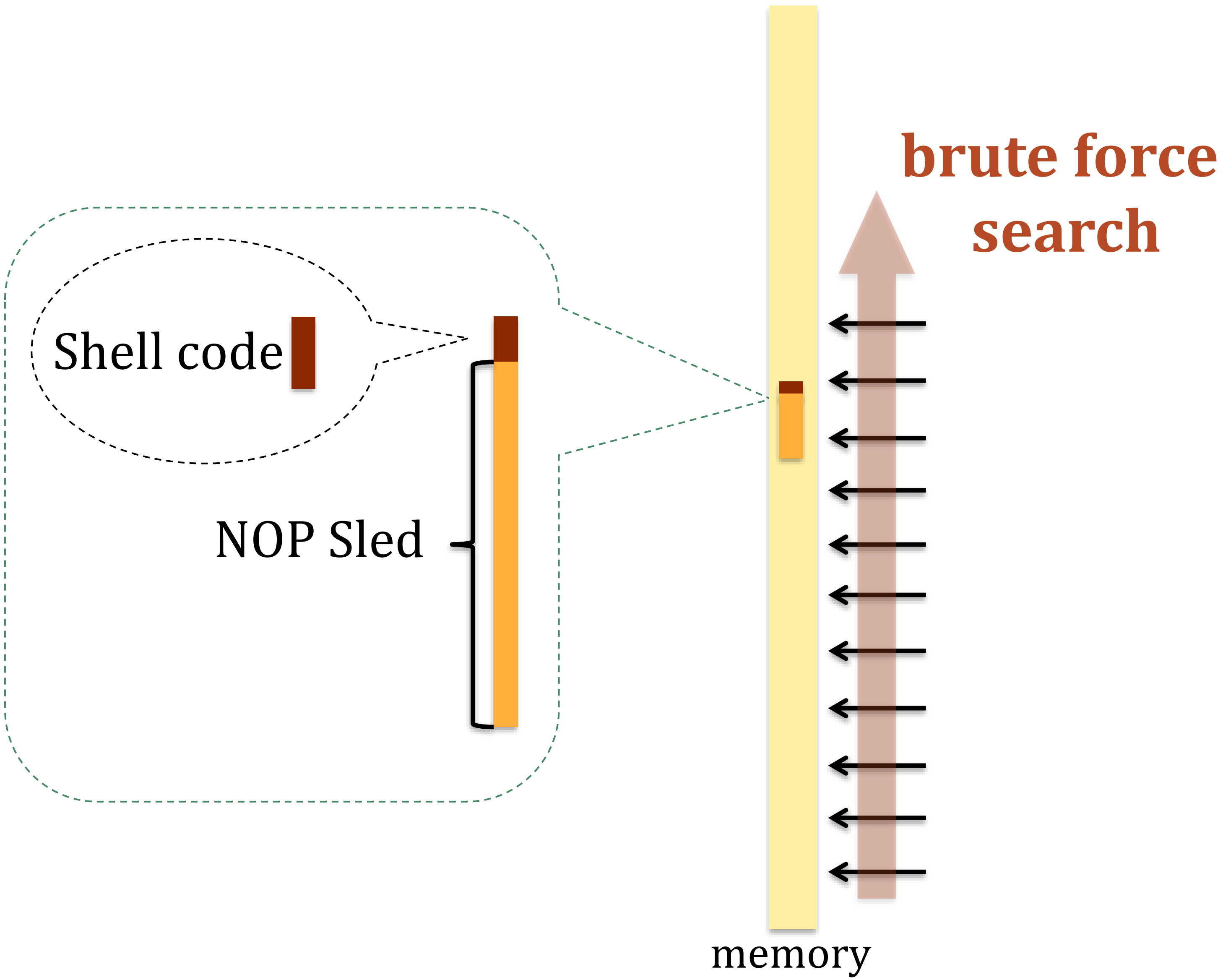
Brute Force



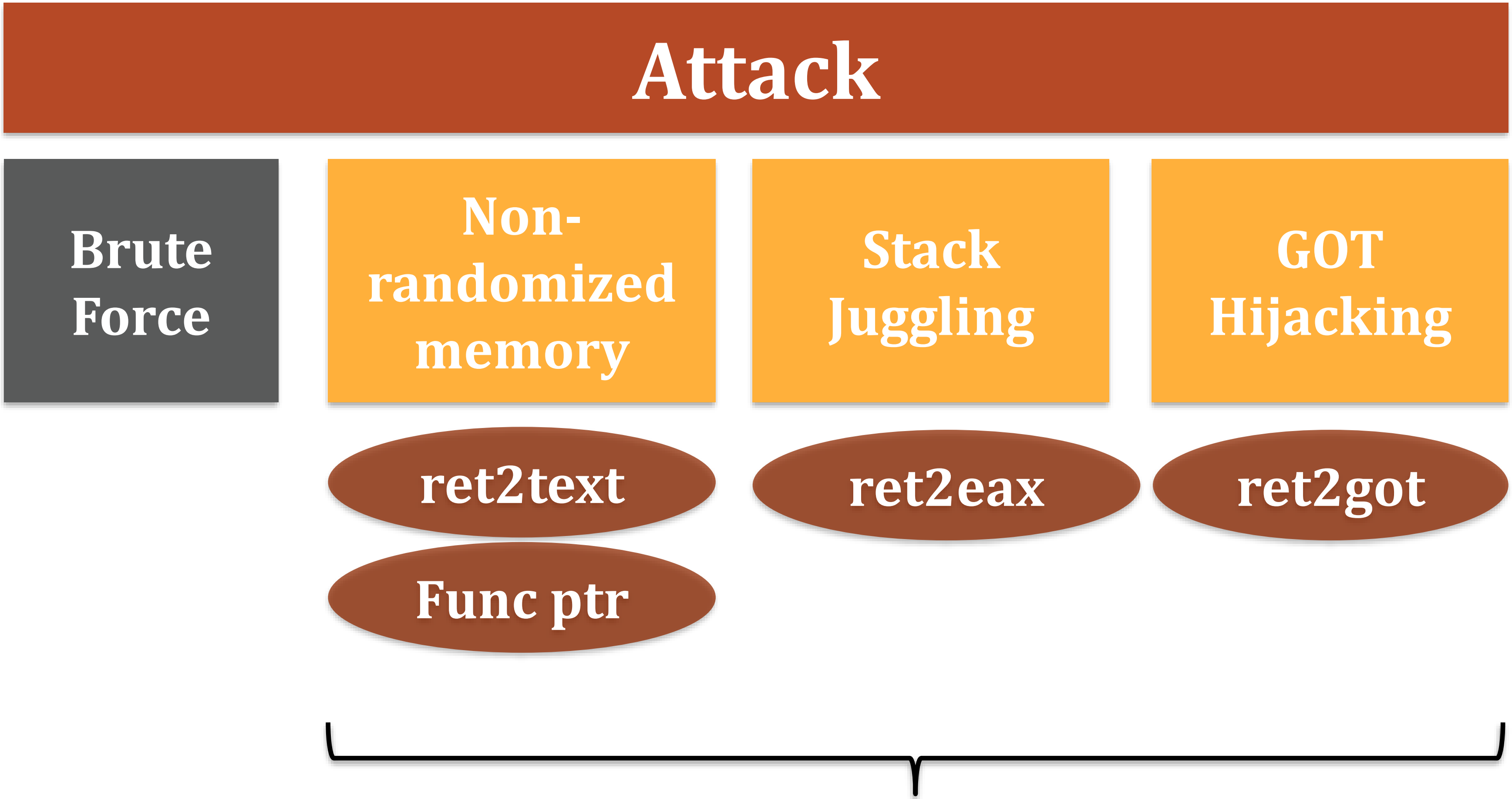
Brute Force



Brute Force



How to attack with ASLR?



Attack

Brute Force

Non-randomized memory

Stack Juggling

GOT Hijacking

ret2text

Func ptr

ret2eax

ret2got

All assume some memory non-randomized

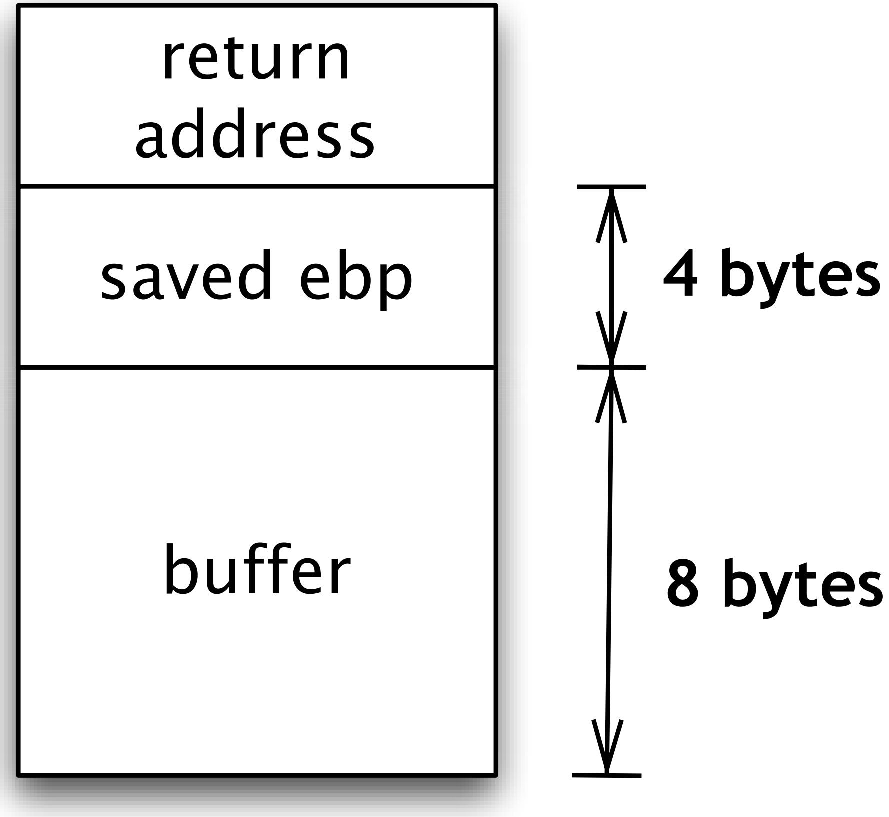
ret2text

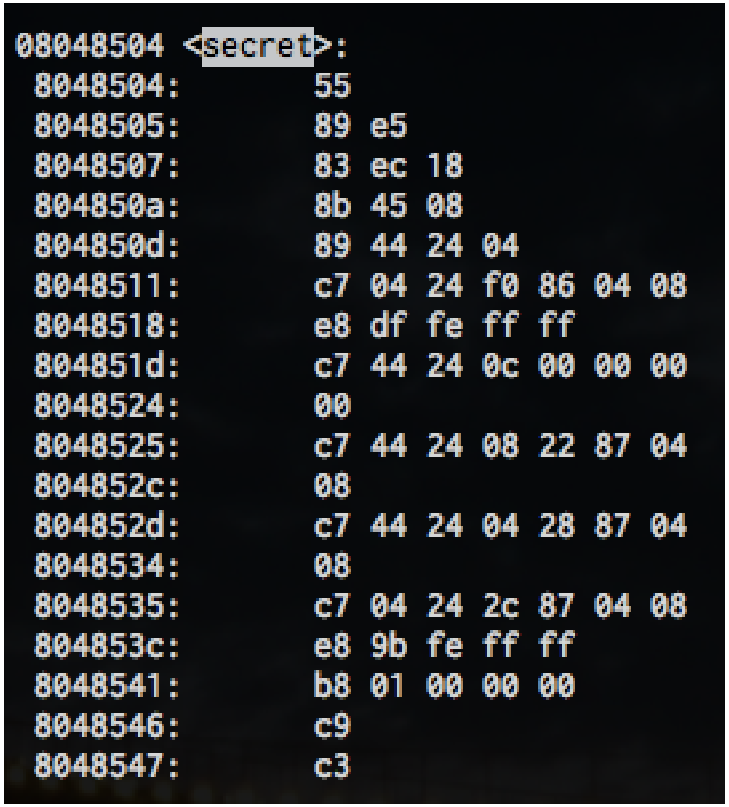
- `text` section has executable program code
  - but not typically randomized by ASLR except PIE

- can hijack control flow to unintended (but existing) program function

ret2text

.text not randomized

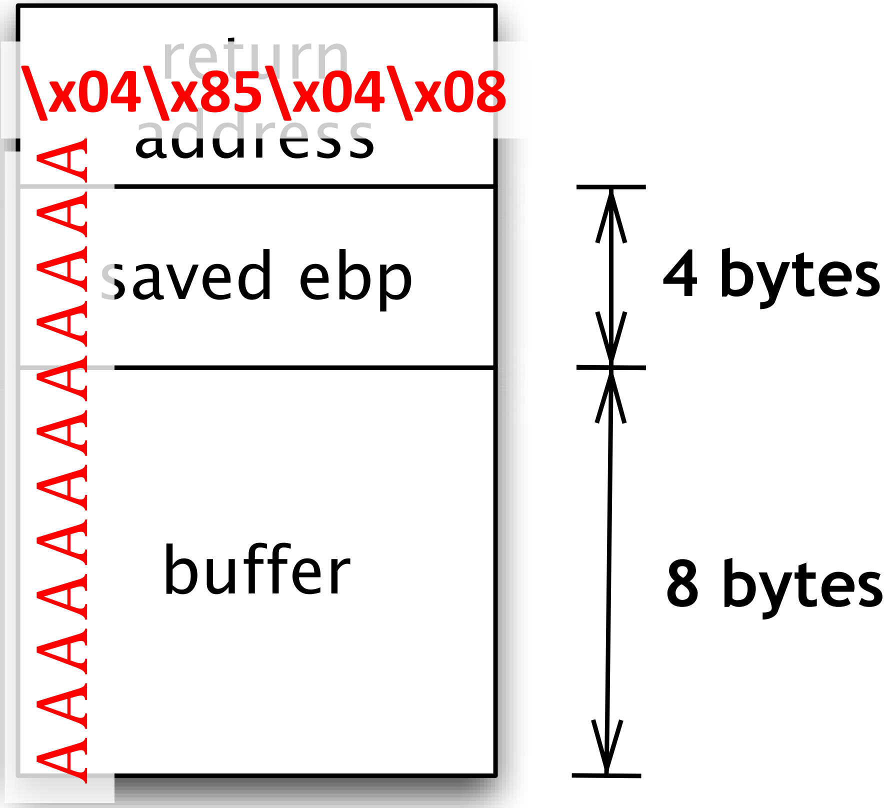


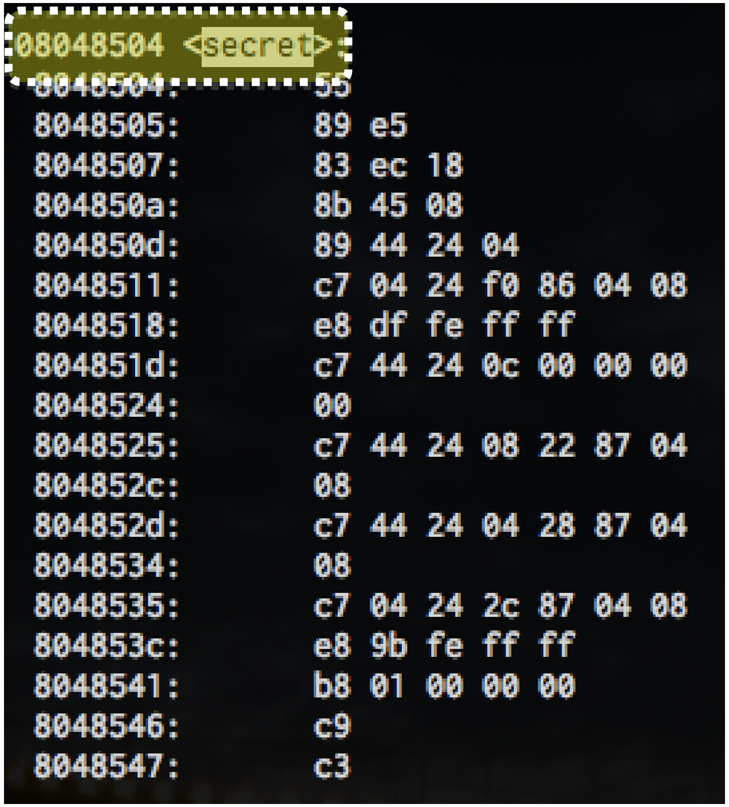


```
08048504 <secret>:
 8048504:       55
 8048505:       89 e5
 8048507:       83 ec 18
 804850a:       8b 45 08
 804850d:       89 44 24 04
 8048511:       c7 04 24 f0 86 04 08
 8048518:       e8 df fe ff ff
 804851d:       c7 44 24 0c 00 00 00
 8048524:       00
 8048525:       c7 44 24 08 22 87 04
 804852c:       08
 804852d:       c7 44 24 04 28 87 04
 8048534:       08
 8048535:       c7 04 24 2c 87 04 08
 804853c:       e8 9b fe ff ff
 8048541:       b8 01 00 00 00
 8048546:       c9
 8048547:       c3
```

ret2text

.text not randomized

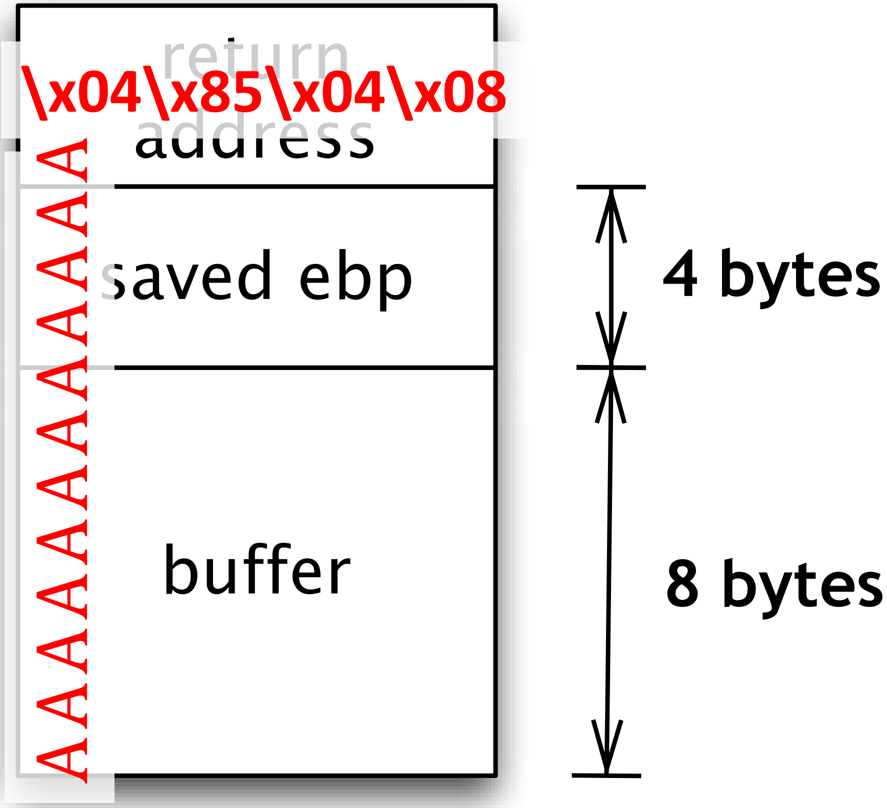


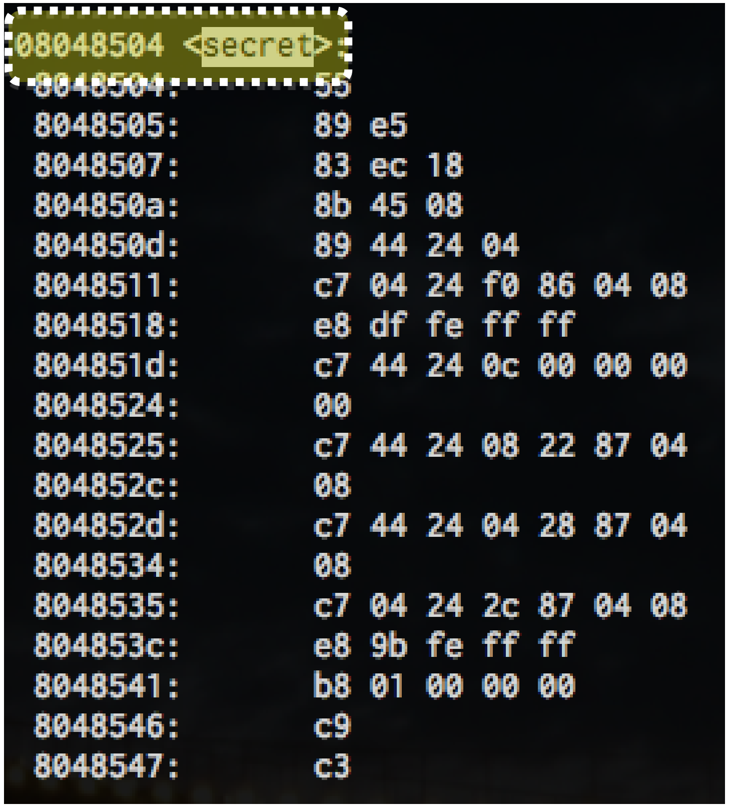


```
08048504 <secret>:
 8048504:               55
 8048505:               89 e5
 8048507:               83 ec 18
 804850a:               8b 45 08
 804850d:               89 44 24 04
 8048511:               c7 04 24 f0 86 04 08
 8048518:               e8 df fe ff ff
 804851d:               c7 44 24 0c 00 00 00
 8048524:               00
 8048525:               c7 44 24 08 22 87 04
 804852c:               08
 804852d:               c7 44 24 04 28 87 04
 8048534:               08
 8048535:               c7 04 24 2c 87 04 08
 804853c:               e8 9b fe ff ff
 8048541:               b8 01 00 00 00
 8048546:               c9
 8048547:               c3
```

ret2text

.text not randomized

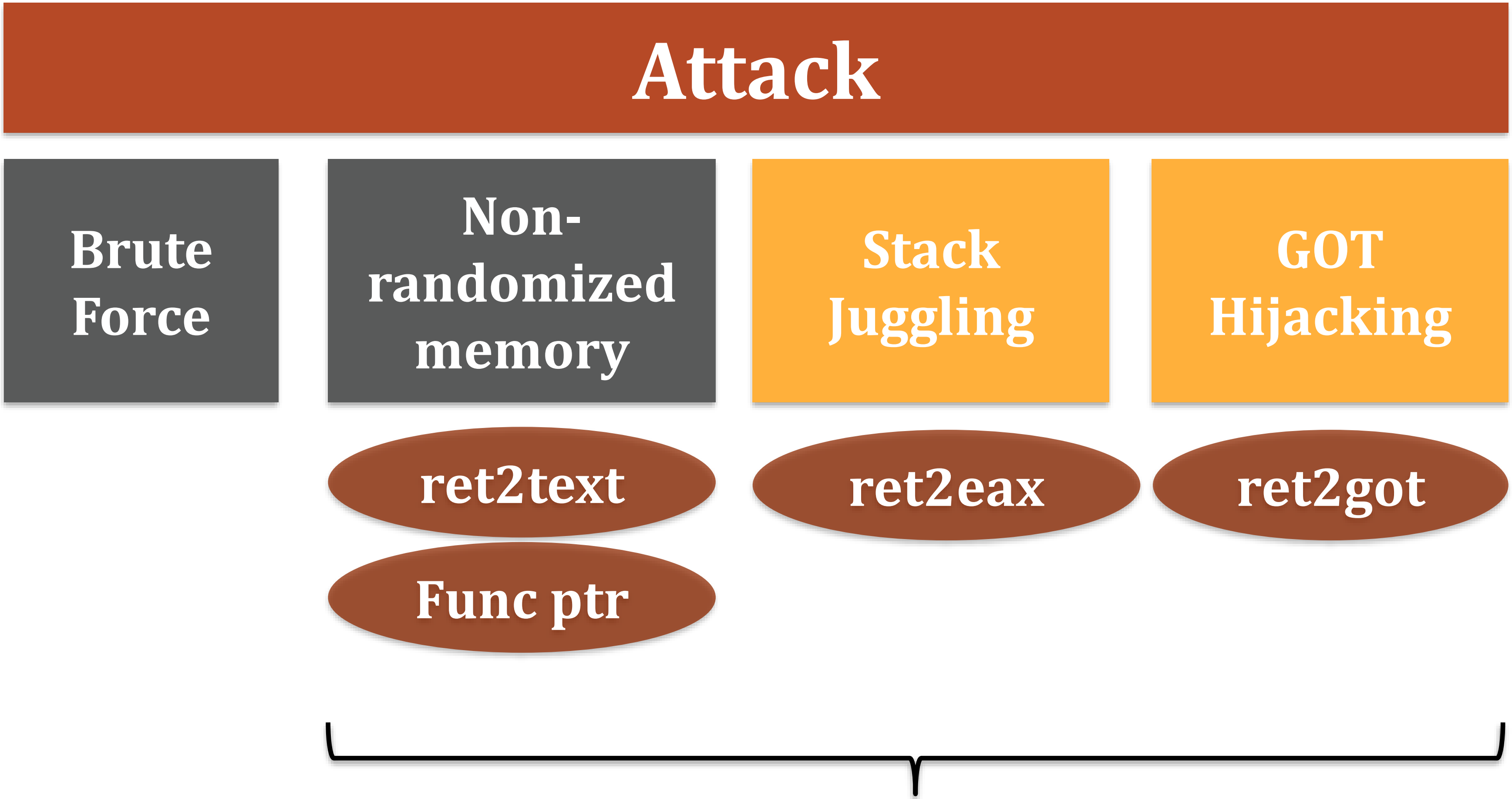




```
08048504 <secret>:
 8048504:       55
 8048505:       89 e5
 8048507:       83 ec 18
 804850a:       8b 45 08
 804850d:       89 44 24 04
 8048511:       c7 04 24 f0 86 04 08
 8048518:       e8 df fe ff ff
 804851d:       c7 44 24 0c 00 00 00
 8048524:       00
 8048525:       c7 44 24 08 22 87 04
 804852c:       08
 804852d:       c7 44 24 04 28 87 04
 8048534:       08
 8048535:       c7 04 24 2c 87 04 08
 804853c:       e8 9b fe ff ff
 8048541:       b8 01 00 00 00
 8048546:       c9
 8048547:       c3
```

Same as running a “secret” function in project 3

How to attack with ASLR?



Attack

Brute Force

Non-randomized memory

Stack Juggling

GOT Hijacking

ret2text

Func ptr

ret2eax

ret2got

All assume some memory non-randomized

ret2eax

```
void msglog(char *input) {
    char buf[64];
    strcpy(buf, input);
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("exploitme <msg>\n");
        return -1;
    }

    msglog(argv[1]);

    return 0;
}
```

returns pointer to buf in eax
eax = buf

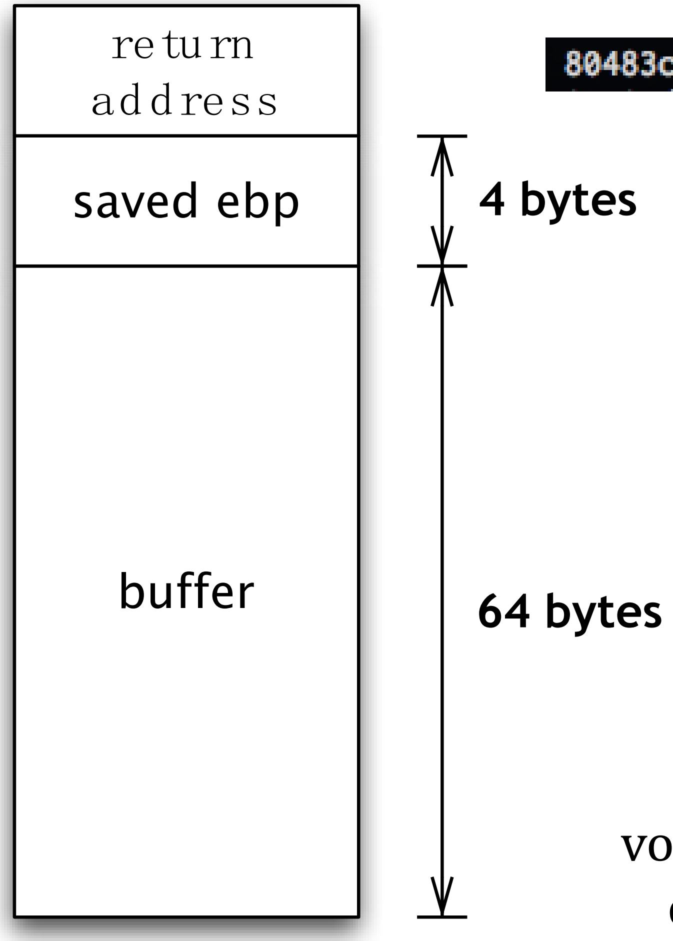
ret2eax

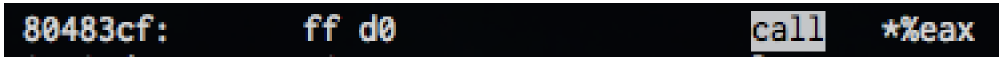
```
void msglog(char *input) {
   char buf[64];
   strcpy(buf, input);
}

int main(int argc, char *argv[]) {
   if(argc != 2) {
      printf("exploitme <msg>\n");
      return -1;
   }

   msglog(argv[1]);

   return 0;
}
```

returns pointer to buf in eax
eax = buf

A subsequent call *eax would redirect control to buf

ret2eax



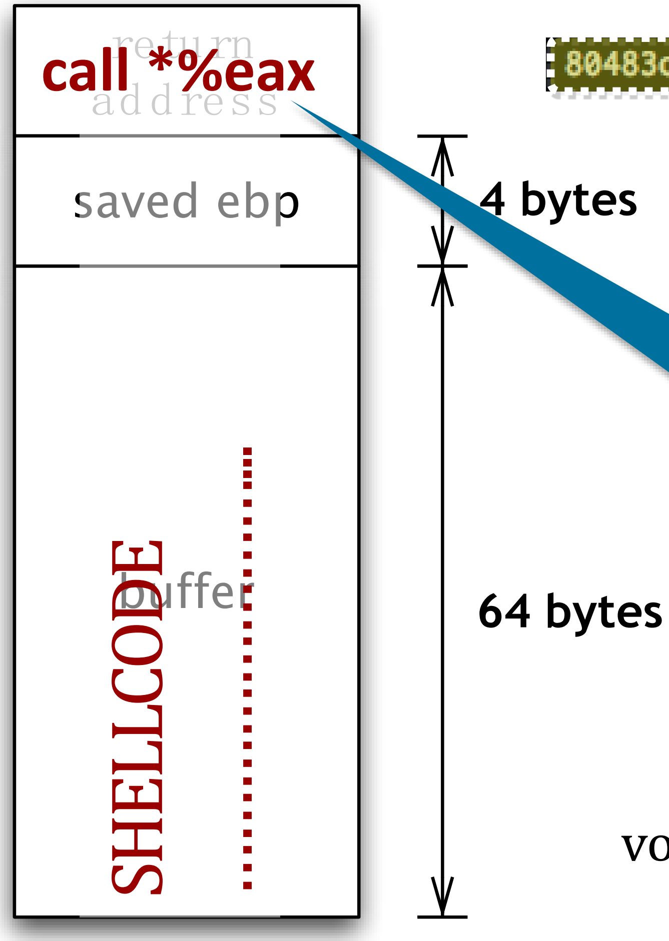


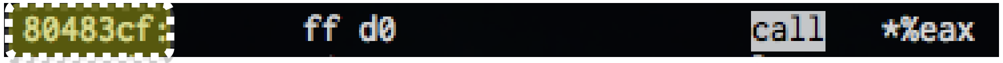
```
80483cf:       ff d0                    call   *%eax
```

Disassemble to find call *eax

```
void msglog(char *input) {
   char buf[64];
   strcpy(buf, input);
}
```

ret2eax





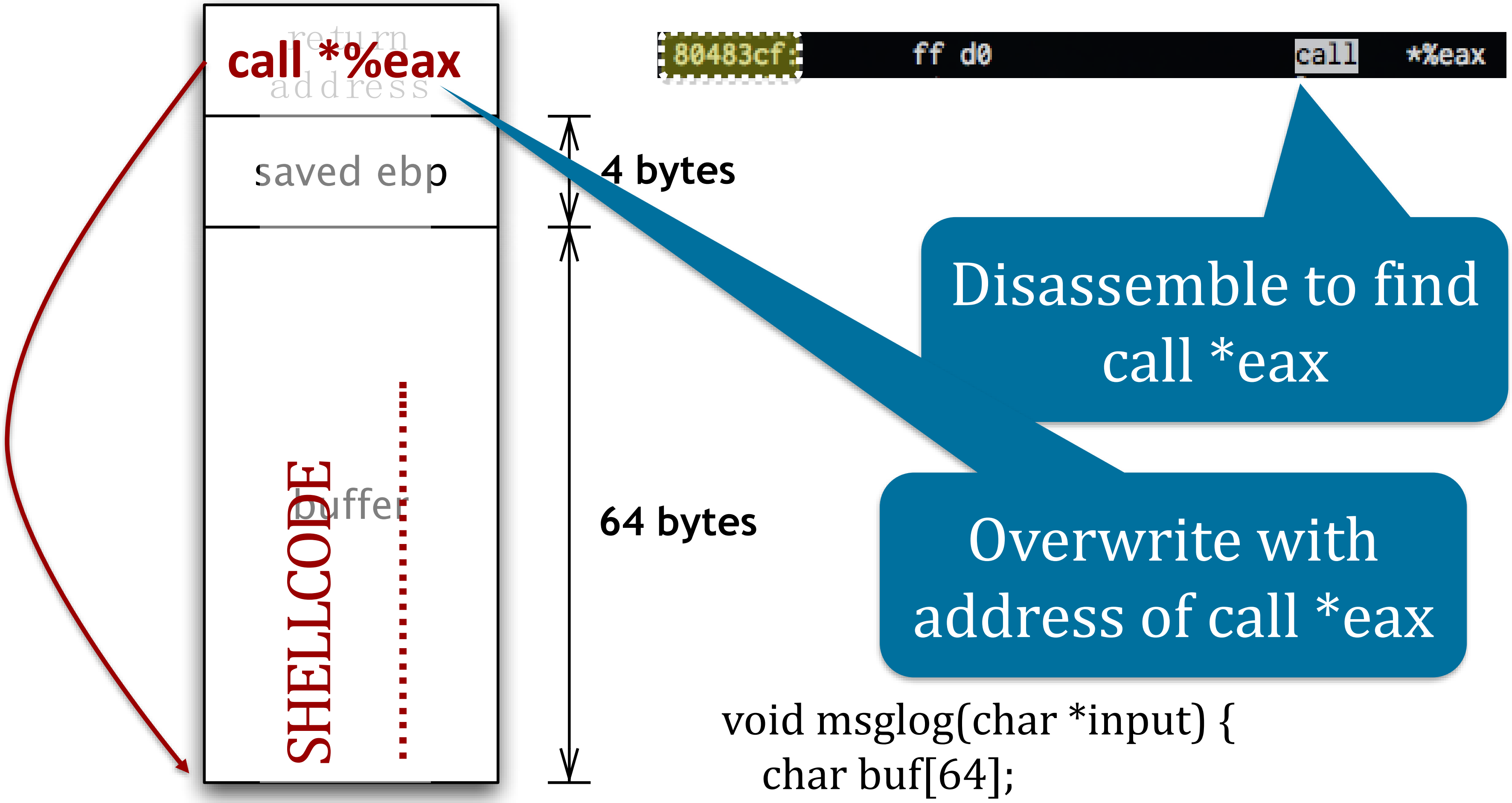
```
80483cf:       ff d0                   call   *%eax
```

Disassemble to find call *eax

Overwrite with address of call *eax

```
void msglog(char *input) {
   char buf[64];
   strcpy(buf, input);
}
```

ret2eax



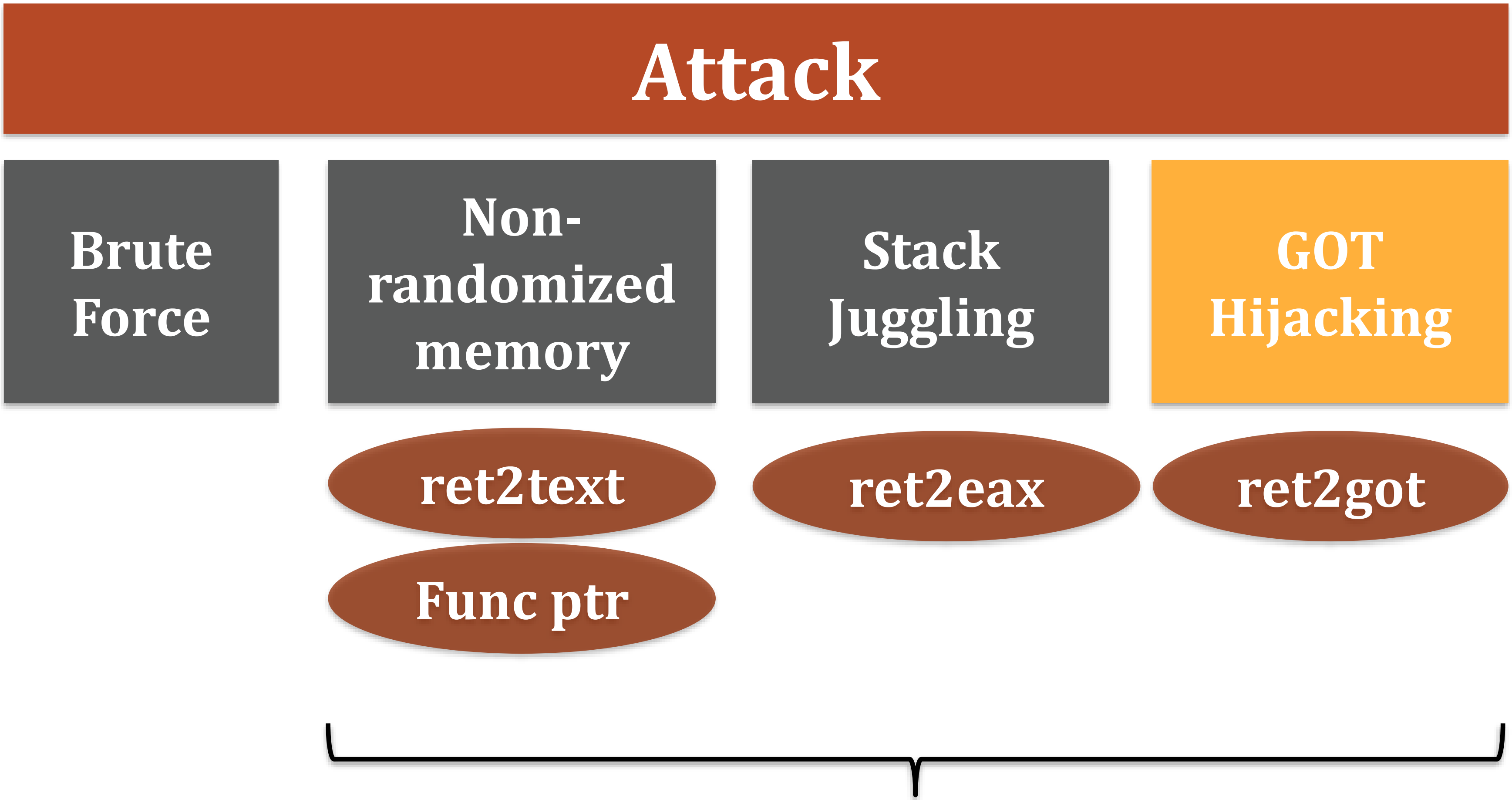
```
80483cf:       ff d0                    call   *%eax
```

Disassemble to find call *eax

Overwrite with address of call *eax

```
void msglog(char *input) {
   char buf[64];
   strcpy(buf, input);
}
```

How to attack with ASLR?



Attack

Brute Force

Non-randomized memory

Stack Juggling

GOT Hijacking

ret2text

Func ptr

ret2eax

ret2got

All assume some memory non-randomized

Questions