

Exploit Mitigations

Exploit Mitigations: Recap

You know how to exploit a buffer overflow.

Like it's 1996.

Lets take you to 2016

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

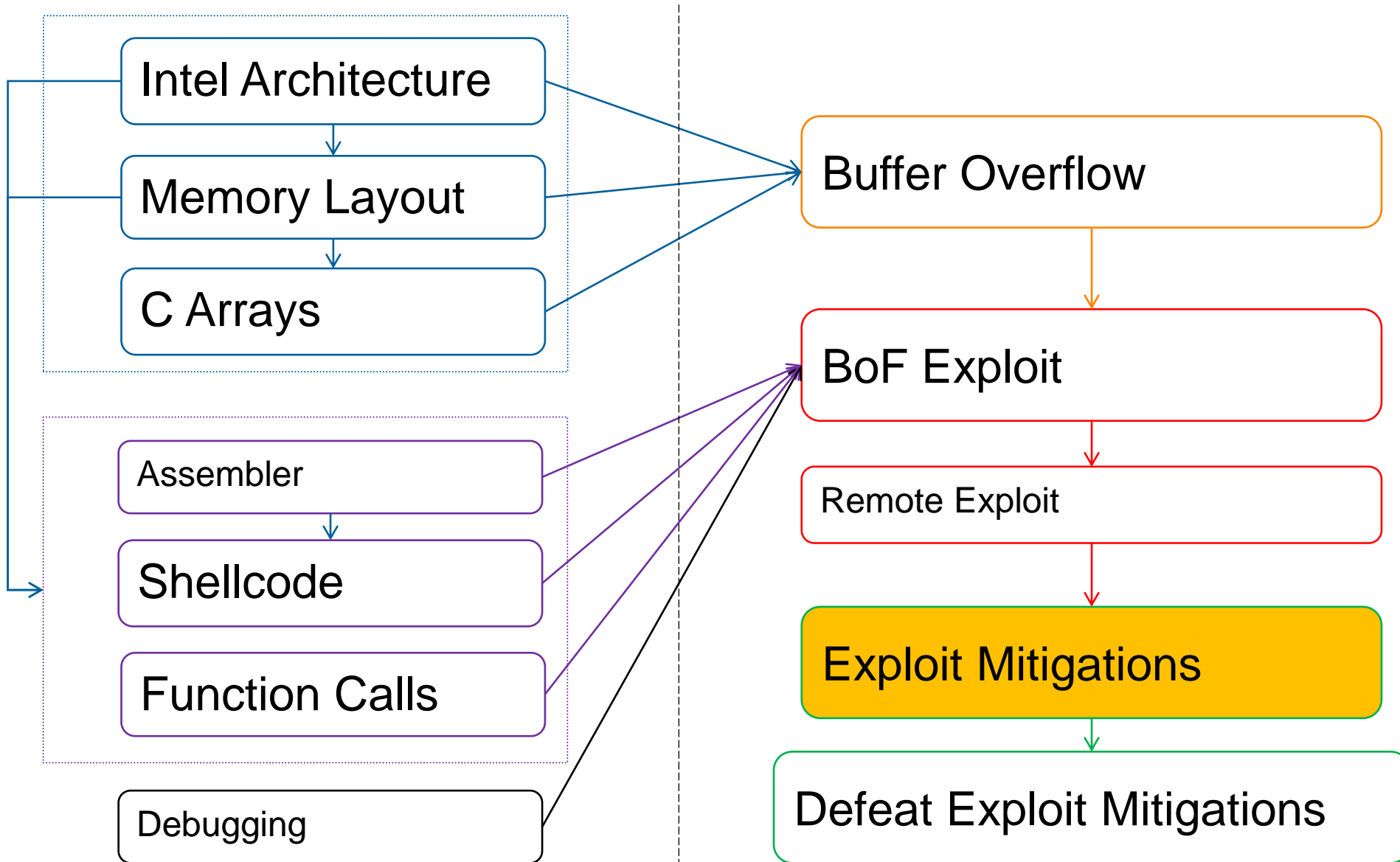
by Aleph One
aleph1@underground.org

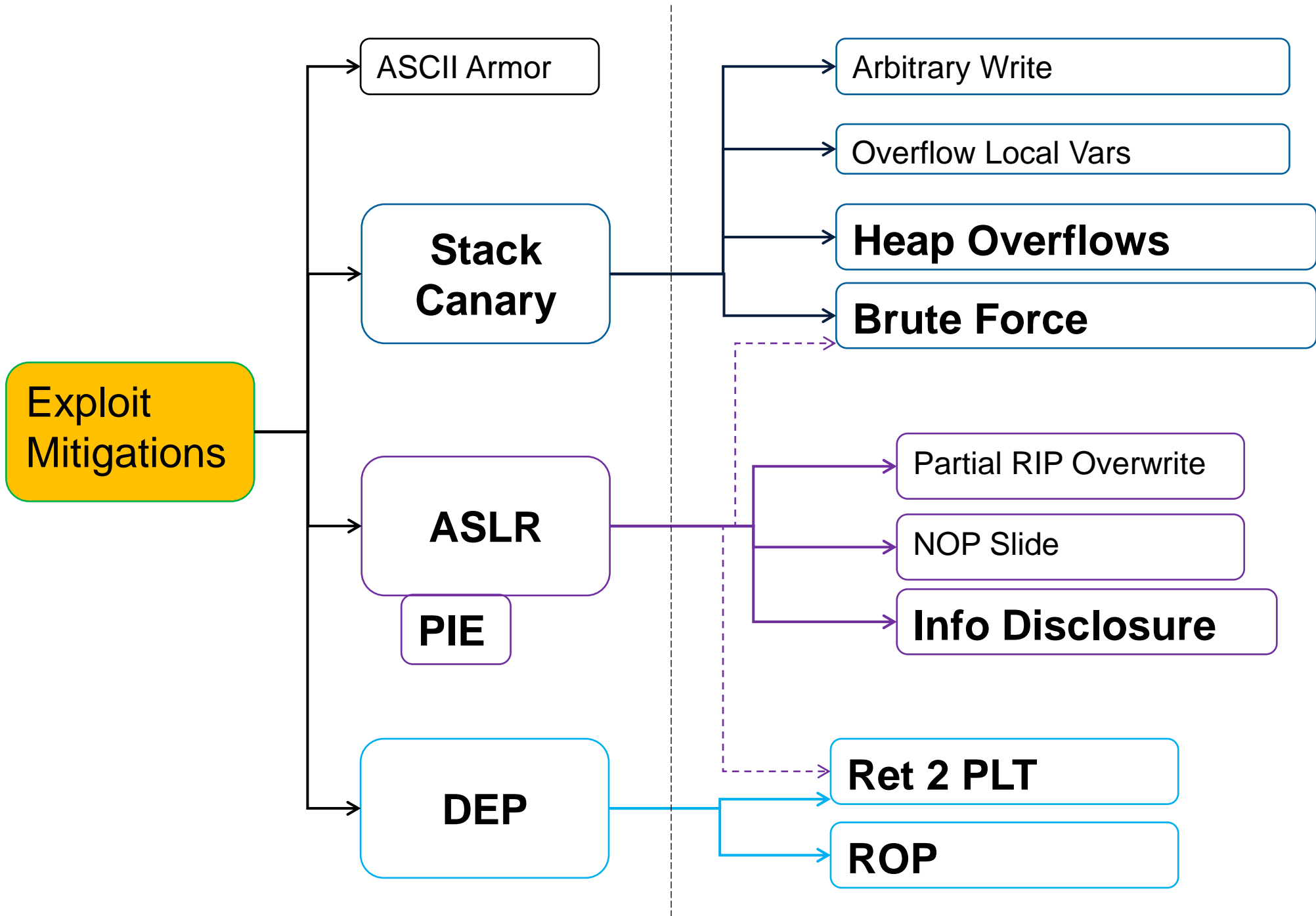
`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Exploit Mitigations: Recap



Content





Exploit Mitigations: Content

DEP (Data Execution Prevention)

Stack Canary

ASLR (Address Space Layout Randomization)

Exploit Mitigations: Security News

Subject: [anti-ROP mechanism in libc](#)
From: [Theo de Raadt <deraadt \(\) openbsd ! org>](#)
OpenBSD [2016-04-25 13:10:25](#)
[26067.1461589825 \(\) cvs ! openbsd ! org](#)
[\[Download message RAW\]](#)

This change randomizes the order of symbols in libc.so at boot time.

This is done by saving all the independent .so sub-files into an ar archive, and then relinking them into a new libc.so in random order, at each boot. The cost is less than a second on the systems I am using.

Grsecurity/PAX

RAP is here. Public demo in 4.5 test patch and commercially available today!

April 28, 2016

Today's release of grsecurity® for the Linux 4.5 kernel marks an important milestone in the project's history. It is the first kernel to contain RAP, a defense mechanism against code reuse attacks. RAP was announced to the

Linux Kernel 4.6

Currently on i386 and on X86_64 when emulating X86_32 in legacy mode, only the stack and the executable are randomized but not other mmaped files (libraries, vDSO, etc.). This patch enables randomization for the libraries, vDSO and mmap requests on i386 and in X86_32 in legacy mode.

Exploit Mitigations

Best Exploit Mitigation:

(Security relevant-) Bugs should not exist at all

Write secure code!

- Use **secure libraries**
- Perform **Static Analysis** of the source code
- Perform **Dynamic Analysis** of programs
- Perform **fuzzing** of input vectors
- Have a secure development lifecycle (**SDL**)
- Manual source code **reviews**
- ...

Developers, developers, developers

Not the focus of this lessons

Practical Exploit Mitigations

Our focus: “Sysadmin/user view”

What can WE do to improve security on our systems?

- Without fixing other people’s code

Two things:

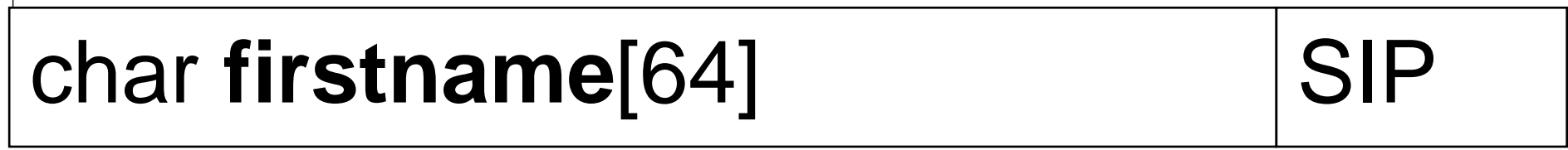
- Compile Time Protection
- Runtime Protection

Practical Exploit Mitigations

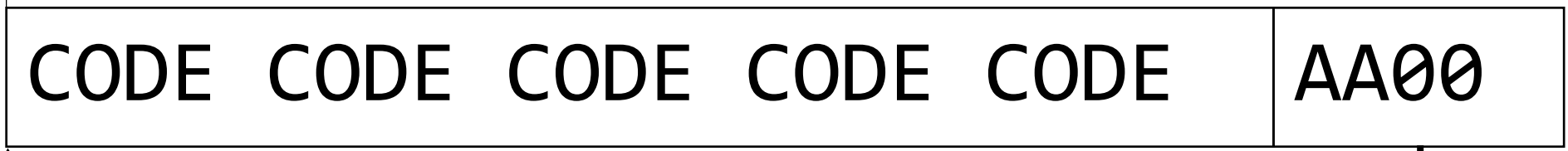


Buffer Overflow Exploit

0xAA00



0xAA00



Exploit Mitigations: Recap

What is required to create an exploit?

- Executable Shellcode
 - Aka “Hacker instructions”
- The distance from buffer to SIP
 - Offset for the overflow
- The Address of shellcode
 - in memory of the target process

Buffer Overflow Exploit

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

```
buf_size = 64
```

```
offset = ??
```

```
ret_addr = "\x??\x??\x??\x??"
```

```
# fill up to 64 bytes
```

```
exploit = "\x90" * (buf_size - len(shellcode))
```

```
exploit += shellcode
```

```
# garbage between buffer and RET
```

```
exploit += "A" * (offset - len(exploit))
```

```
# add ret
```

```
exploit += ret_addr
```

```
# print to stdout
```

```
sys.stdout.write(exploit)
```

Practical Exploit Mitigations

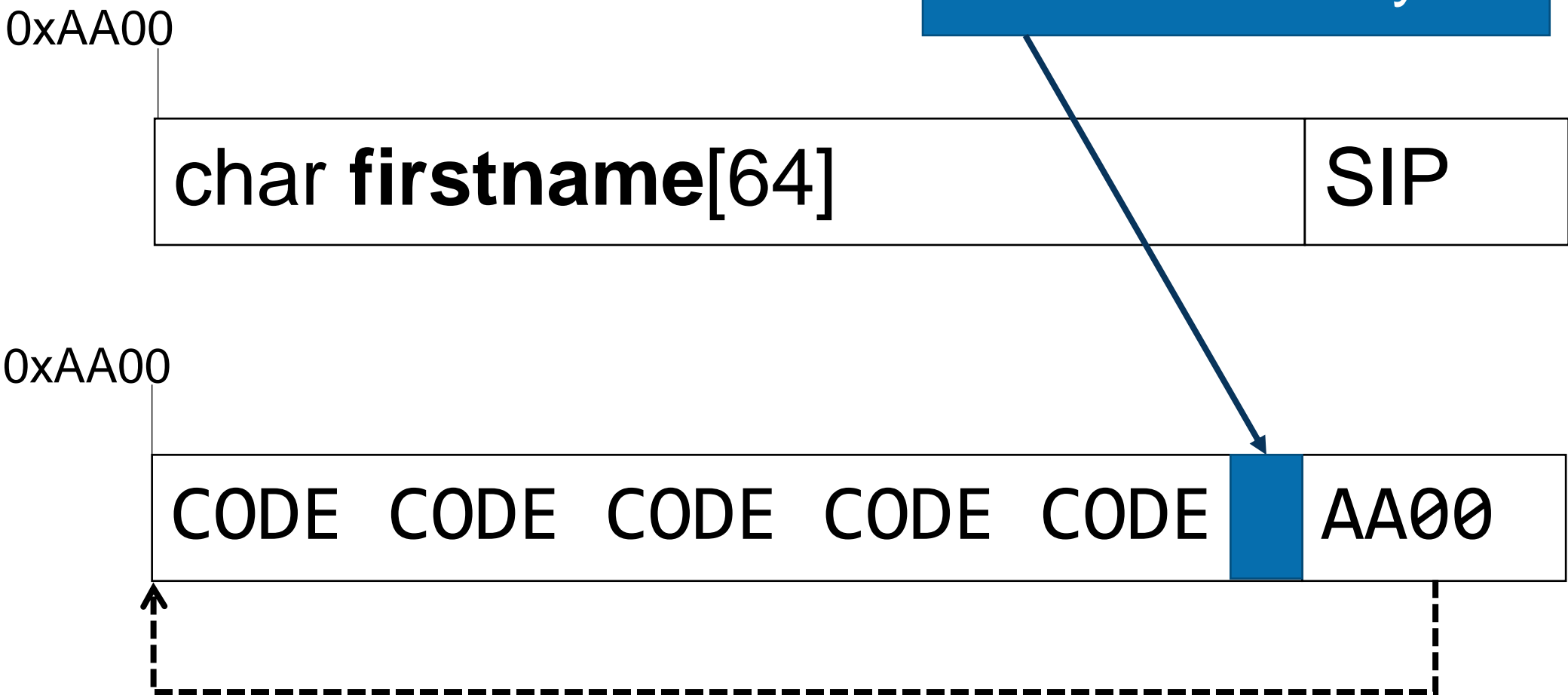
Compile Time:

- Stack canaries
- PIE

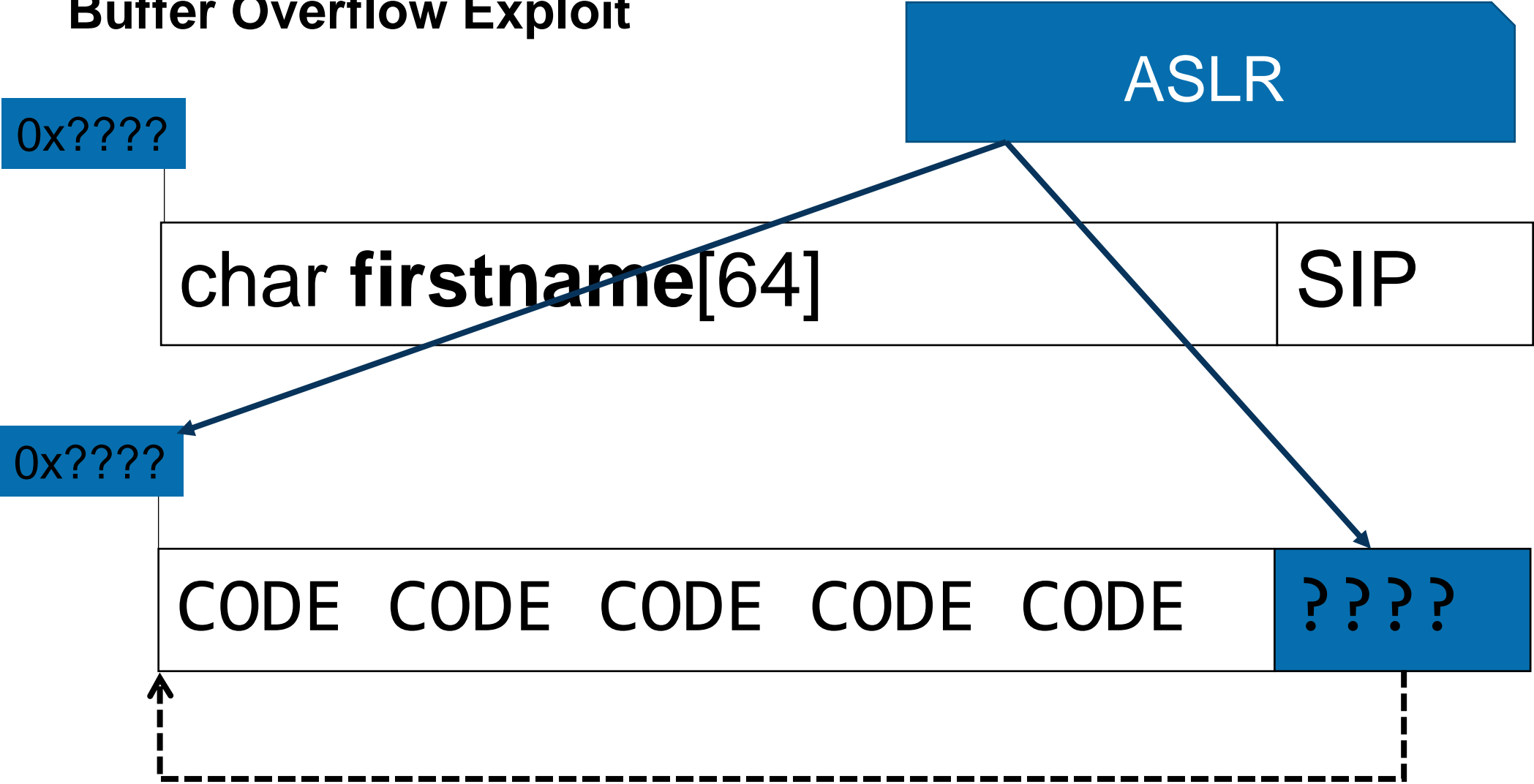
Runtime:

- ASLR
- DEP
- ASCII Armor

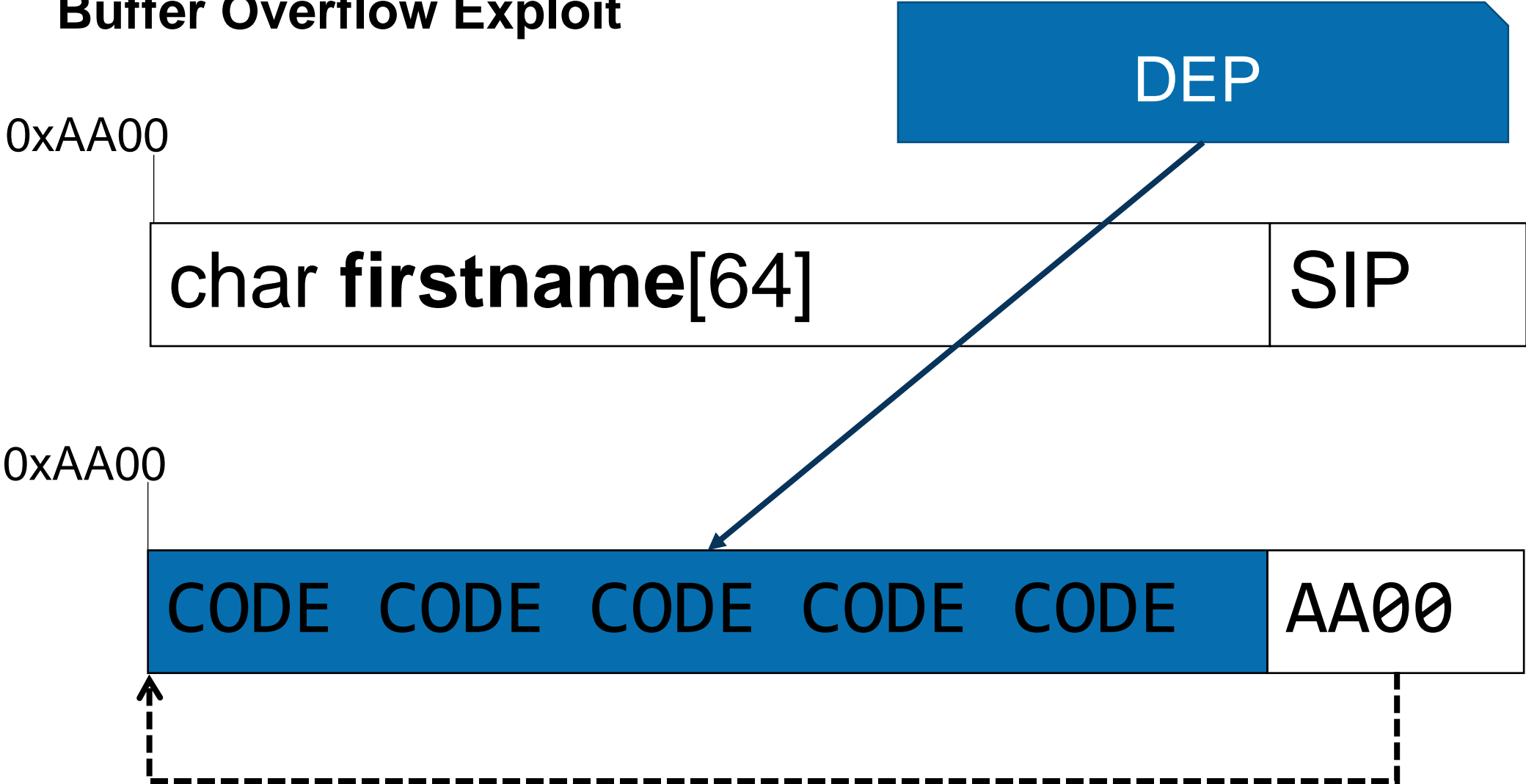
Buffer Overflow Exploit



Buffer Overflow Exploit



Buffer Overflow Exploit



Buffer Overflow Exploit

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\b\xcd\x80"

buf_size = 64
offset = ??

ret_addr = "\x??\x??\x??\x??"

# fill up to 64 bytes
exploit = "\x90" * (buf_size - len(shellcode))
exploit += shellcode

# garbage between buffer and RET
exploit += "A" * (offset - len(exploit))

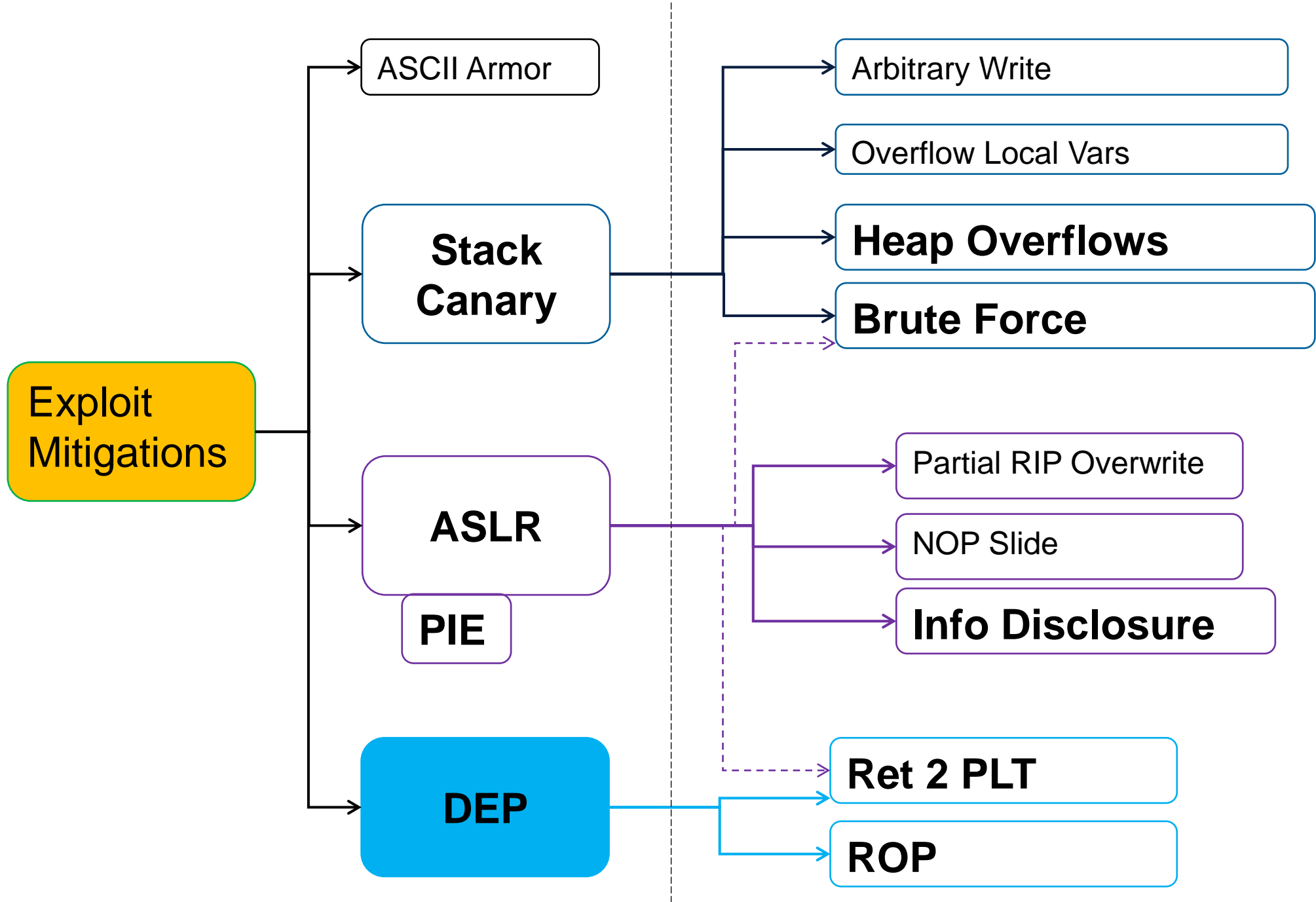
# add ret
exploit += ret_addr

# print to stdout
sys.stdout.write(exploit)
```

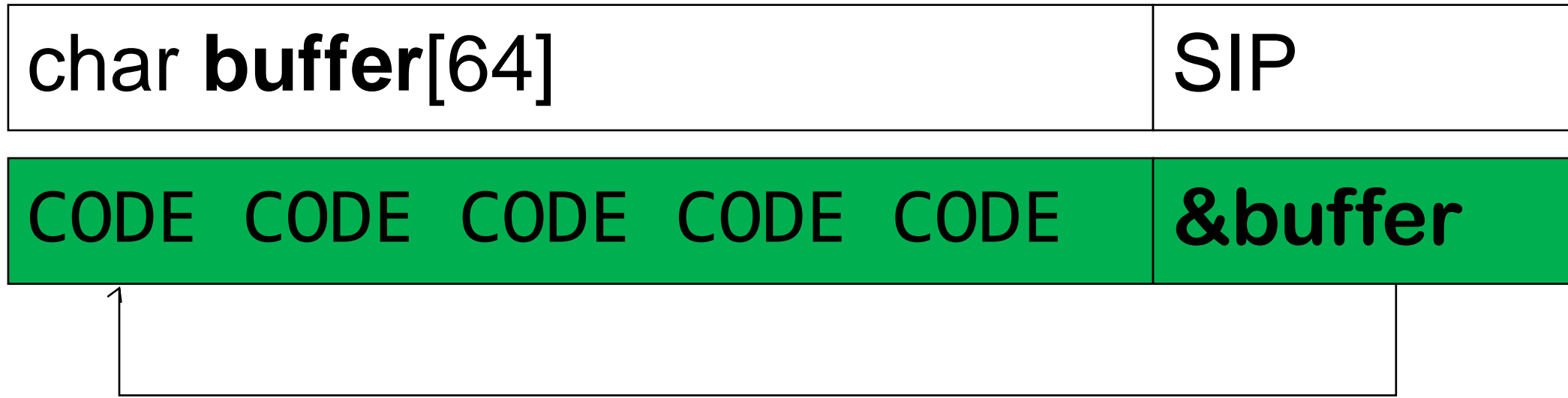
DEP

ASLR

Exploit Mitigation: DEP



Exploit Mitigations: Recap



DEP: Make stack not executable

DEP

DEP – **D**ata **E**xecution **P**revention

- Aka: No-Exec Stack
- Aka: W^X (Write XOR eXecute)(OpenBSD)
- Aka: NX (Non-Execute) Bit

32 bit (x86)

- Since 386
- “saved” Xecute bit (Read / Write are available)

AMD64 (x86-64)

- introduced NX bit in HW
- Or kernel patches like PaX
- For 32 bit, need PAE (Physical Address Extension, 32->36bit)

Linux

- Support in 2004, Kernel 2.6.8, default active

DEP

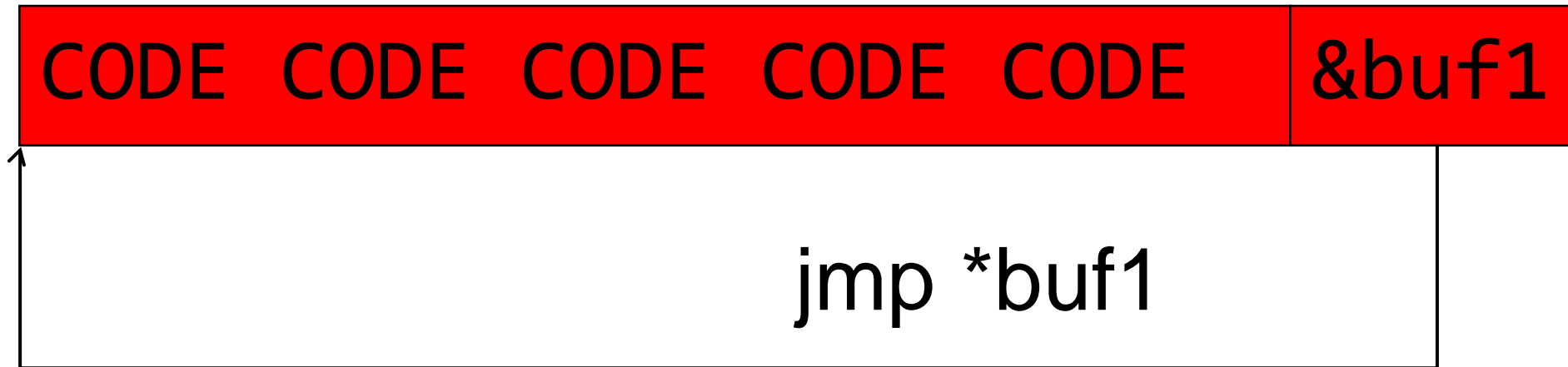
Memory regions

- Are mapped with permissions
- Like files
 - R Read
 - W Write
 - X eXecute
- DEP removes X bit from memory which do not contain code
 - Stack
 - Heap
 - (Possibly others)

Anti-Exploitation: No-Exec Stack

Without DEP:

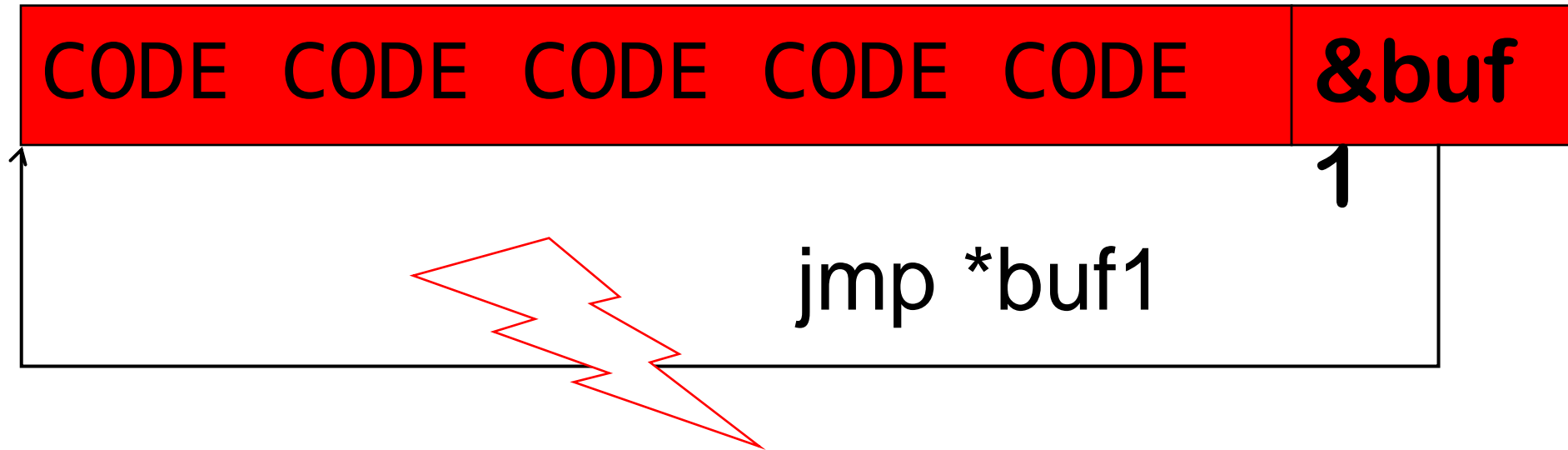
Permissions: rwX



Anti-Exploitation: No-Exec Stack

With DEP:

Permissions: rw-



“Segmentation Fault”

DEP Example

```
(gdb) r
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0xbffff4ec in ?? ()
```

```
(gdb) info proc mappings
```

```
Mapped address spaces:
```

```
[...]
```

```
0xbffdf000 0xc0000000 0x21000 0x0 [stack]
```

```
(gdb) i r eip
```

```
eip 0xbffff4ec 0xbffff4ec
```

Anti-Exploitation: No-Exec Stack

```
$ gcc system.c -o system && readelf -l system
```

Program Headers:

Type	Offset	VirtAddr	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x00013	R	0x1
LOAD	0x000000	0x08048000	0x005d0	R E	0x1000
LOAD	0x000f14	0x08049f14	0x00108	RW	0x1000
DYNAMIC	0x000f28	0x08049f28	0x000c8	RW	0x4
NOTE	0x000168	0x08048168	0x00044	R	0x4
GNU_EH_FRAME	0x0004d8	0x080484d8	0x00034	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000	RW	0x4
GNU_RELRO	0x000f14	0x08049f14	0x000ec	R	0x1

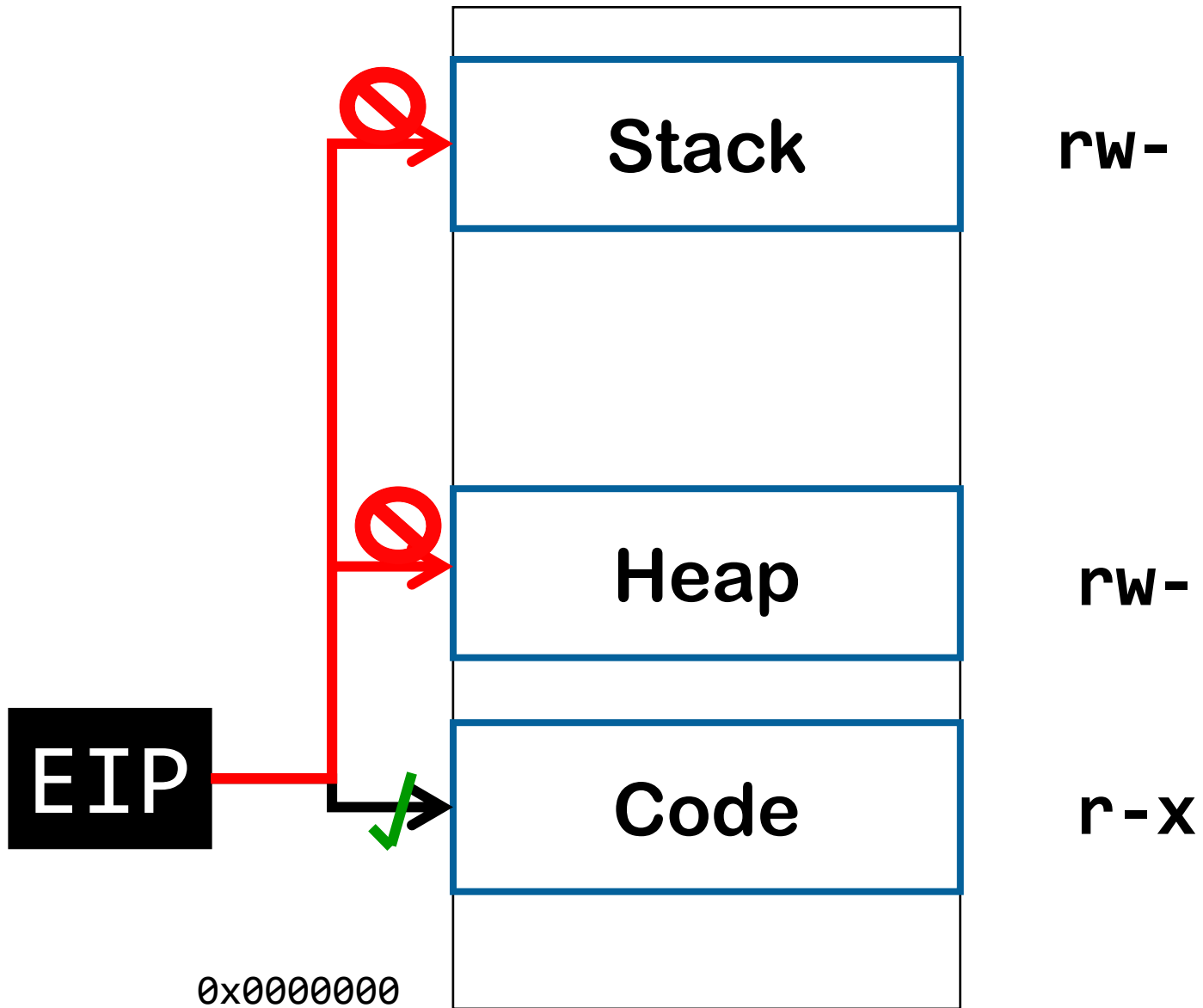
Anti-Exploitation: No-Exec Stack

```
$ gcc system.c -z execstack -o system
$ readelf -l system
```

Program Headers:

Type	Offset	VirtAddr	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x00013	R	0x1
LOAD	0x000000	0x08048000	0x005d0	R E	0x1000
LOAD	0x000f14	0x08049f14	0x00108	RW	0x1000
DYNAMIC	0x000f28	0x08049f28	0x000c8	RW	0x4
NOTE	0x000168	0x08048168	0x00044	R	0x4
GNU_EH_FRAME	0x0004d8	0x080484d8	0x00034	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000	RWE	0x4
GNU_RELRO	0x000f14	0x08049f14	0x000ec	R	0x1

Anti-Exploitation: No-Exec Stack

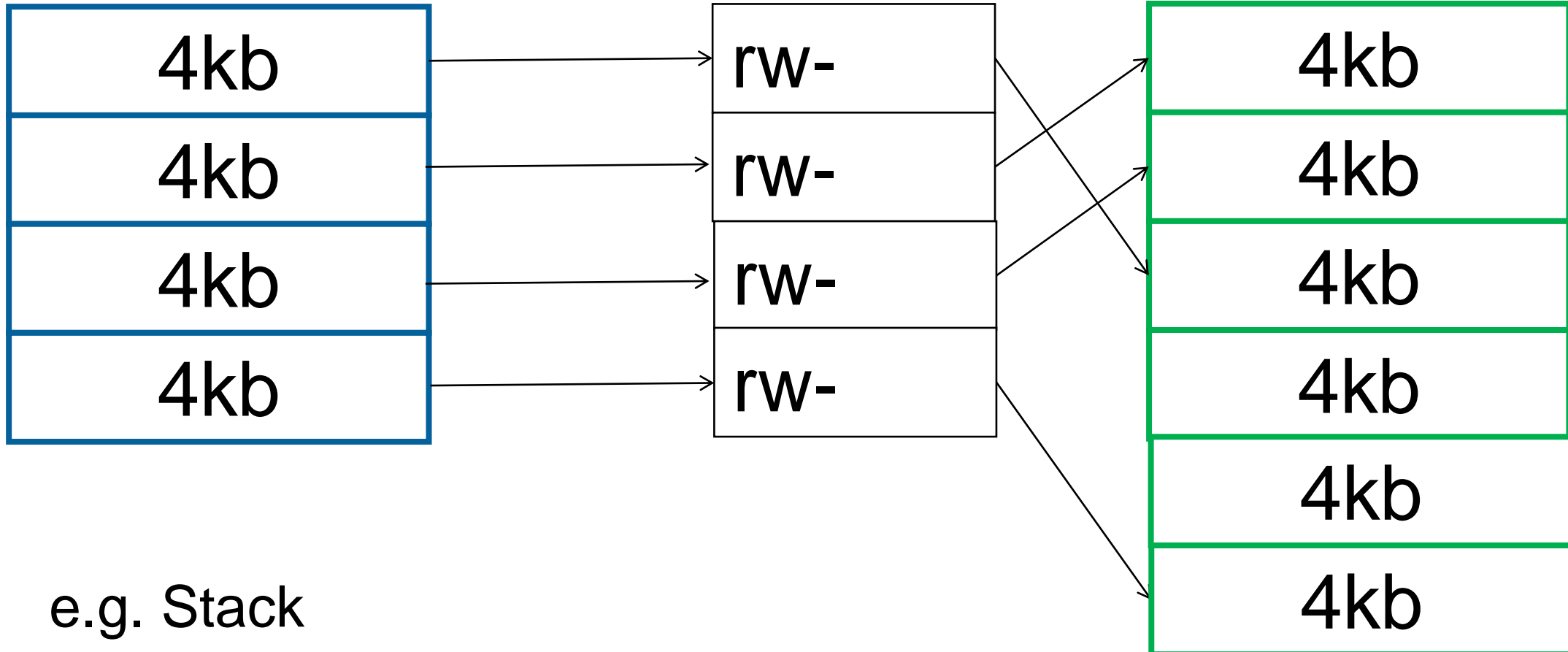


MMU

Memory Segment:

MMU

RAM:



Anti-Exploitation: No-Exec Stack

GCC compiles automatically with no-exec stack



Recap! DEP

Exploit Mitigation – DEP

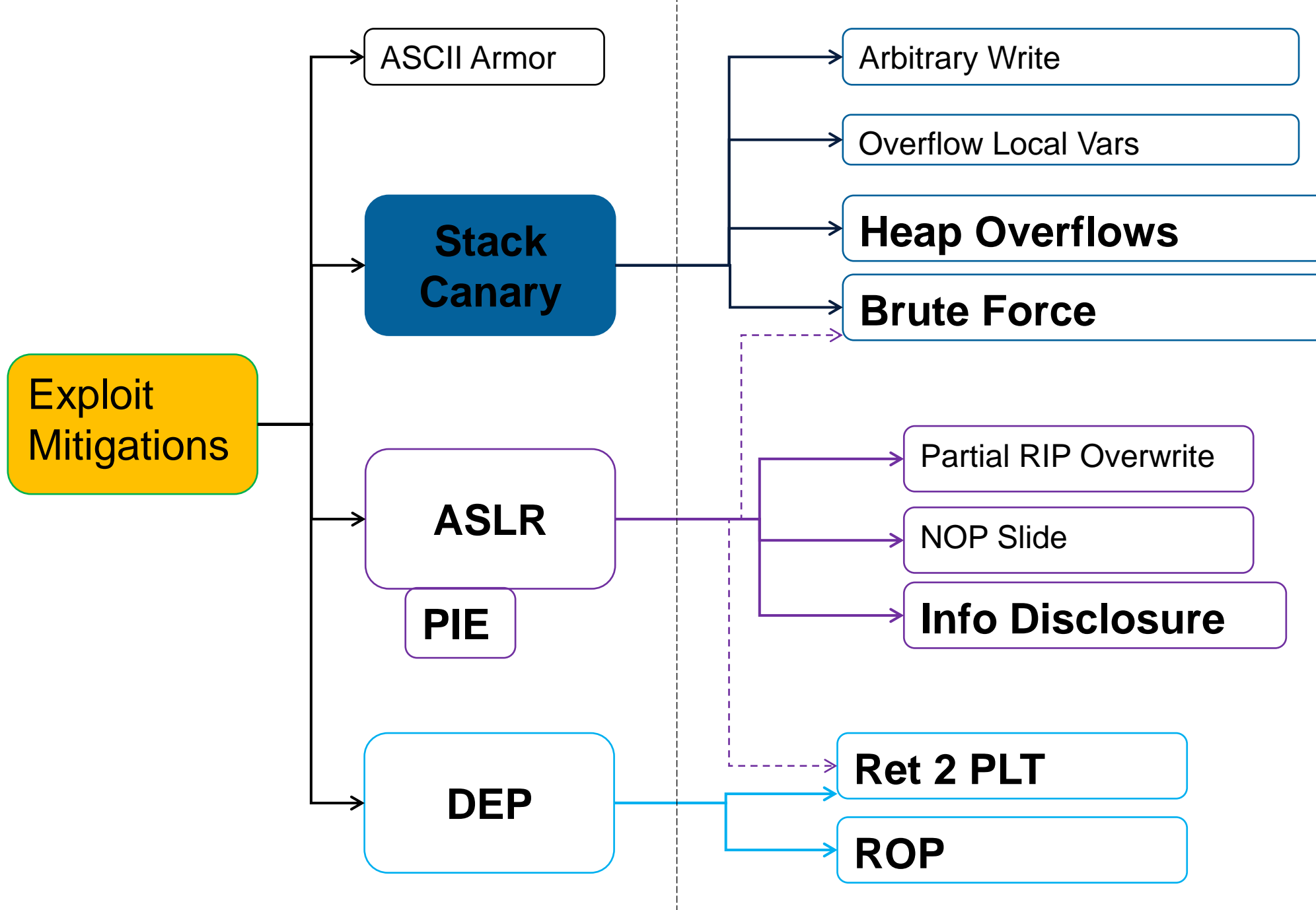
- Makes it impossible for an attacker to execute his own shellcode
- Code segment: eXecute (no write)
- Heap, Stack: Write (no execute)

Recap! DEP

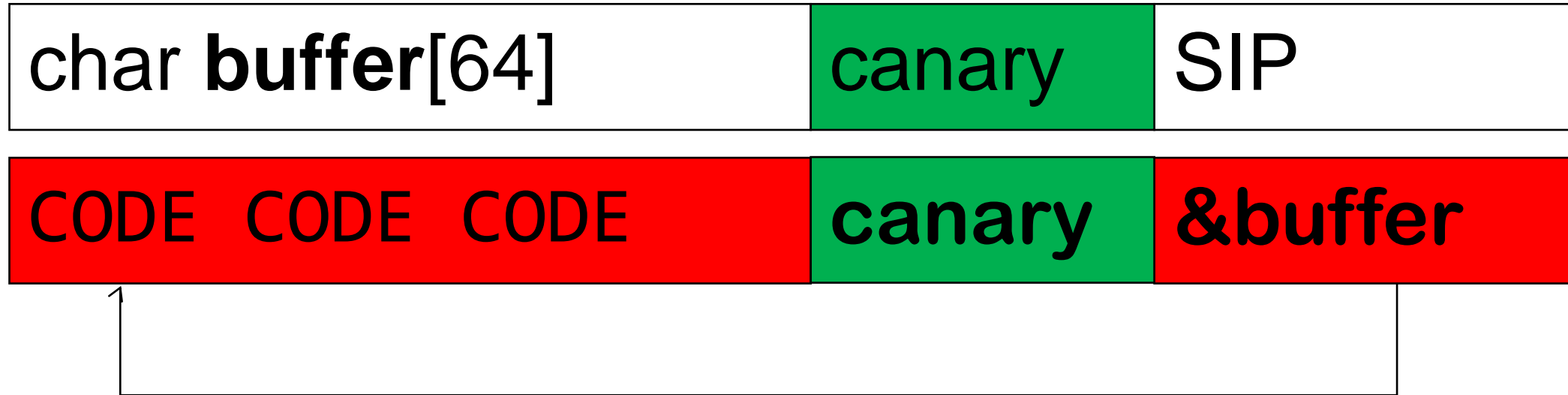
Exploit Mitigation – DEP

- No-no: Write AND Execute
- Sometimes necessary
- Interpreted Languages
 - E.g. Java
 - Or JavaScript
 - Ähem *Browser* ähem

Exploit Mitigation – Stack Protector



Exploit Mitigations: Recap



Exploit Mitigation – Stack Protector

Aka:

- SSP: Stack Smashing Protector
- Stack Cookie
- Stack Canary

Exploit Mitigation – Stack Protector

Secret value in front of control data

A value unknown to the attacker

Checked before performing a “ret”

- When returning from a function; “return;”
- Before using SIP

```
if (secret_on_stack == global_secret) {  
    return;  
} else {  
    crash();  
}
```

Exploit Mitigation – Stack Protector

char buf1[16]	EIP
---------------	-----

Exploit Mitigation – Stack Protector

char buf1[16]	EIP
---------------	-----

char buf1[16]	secret	EIP
---------------	--------	-----

Exploit Mitigation – Stack Protector

char buf1[16]	secret	EIP
---------------	--------	-----

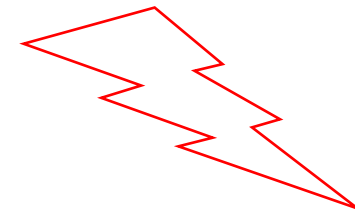
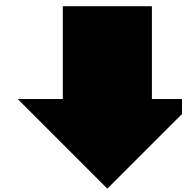
char buf1[16]	55667	FF12
---------------	-------	------

CODE CODE CODE CODE	BBBB	AA00
---------------------	------	------

Exploit Mitigation – Stack Protector

char buf1[16]	55667	FF12
---------------	-------	------

CODE CODE CODE CODE	BBBB	AA00
---------------------	------	------



“Segmentation Fault”

BBBB != 55667

Exploit Mitigation – Stack Protector

Stack Protector

- GCC patch
 - First: StackGuard in 1997
 - Then: ProPolice in 2001, by IBM
- Finally: Re-implement ProPolice in 2005 by RedHat
 - introduced in GCC 4.1
 - -fstack-protector
- Update: Better implementation by Google in 2012
 - -fstack-protector-strong
- Enabled since like forever by default
 - most distributions
 - most packages

Exploit Mitigation – Stack Protector

When does the stack protector change?

- On `execve()`
 - (replace current process with a ELF file from disk)
- NOT on `fork()`
 - (copy current process)

Exploit Mitigation – Stack Protector

Stack canary properties:

- Not predictable
- Be located in a non-accessible location
- Cannot be brute-forced
- Should contain at least one termination character

Exploit Mitigation – Stack Protector

```
gdb-peda$ disas handleData
```

```
Dump of assembler code for function handleData:
```

```
[...]
```

```
0x080488b5 <+136>:    call    0x8048650 <puts@plt>
0x080488ba <+141>:    add     esp,0x10
0x080488bd <+144>:    nop
0x080488be <+145>:    mov     eax,DWORD PTR [ebp-0xc]
0x080488c1 <+148>:    xor     eax,DWORD PTR gs:0x14
0x080488c8 <+155>:    je      0x80488cf <handleData+162>
0x080488ca <+157>:    call    0x8048600 <__stack_chk_fail@plt>
0x080488cf <+162>:    leave
0x080488d0 <+163>:    ret
```

Exploit Mitigation – Stack Protector Check

Stack protector in ASM, static analysis:

```
// get stack canary from current frame
```

```
mov      -0xc(%ebp),%eax
```

```
// compare hat with reference value
```

```
xor      %gs:0x14,%eax
```

```
// skip next instruction if ok
```

```
je      0x804846e <bla+58>
```

```
// was not ok - crash/exit program
```

```
call    0x8048340 <__stack_chk_fail@plt>
```

Exploit Mitigation – Stack Protector Check

Stack protector in ASM, dynamic analysis:

```
=> 0x08048458 <+36>:      call    0x8048350 <strcpy@plt>
    0x0804845d <+41>:      mov     -0xc(%ebp),%eax
    0x08048460 <+44>:      xor     %gs:0x14,%eax
    0x08048467 <+51>:      je      0x804846e <bla+58>
    0x08048469 <+53>:      call    0x8048340 <__stack_chk_fail@plt>
```

```
(gdb) x/1x $ebp-0xc
```

```
0xbffff5cc:      0x2f140600
```

```
(gdb) info auxv
```

```
25   AT_RANDOM          Address of 16 random bytes      0xbffff7bb
```

```
(gdb) x/1x 0xbffff7bb
```

```
0xbffff7bb:      0x2f1406ae
```


Exploit Mitigation – Stack Protector Setup

```
0x0804888c <+0>:    push    ebp                // Prologue
0x0804888d <+1>:    mov     ebp,esp            // Prologue
0x0804888f <+3>:    sub     esp,0x38           // Make some space
0x08048892 <+6>:    mov     eax,DWORD PTR [ebp+0x8] // Parameter 1
0x08048895 <+9>:    mov     DWORD PTR [ebp-0x2c],eax
0x08048898 <+12>:   mov     eax,DWORD PTR [ebp+0xc] // Parameter 2
0x0804889b <+15>:   mov     DWORD PTR [ebp-0x30],eax
0x0804889e <+18>:   mov     eax,gs:0x14         // Stack cookie
0x080488a4 <+24>:   mov     DWORD PTR [ebp-0xc],eax
0x080488a7 <+27>:   xor     eax,eax
```

Stack Smashing Example

```
$ ./strcpy AAAAAAAAAAAAAA
*** stack smashing detected ***: ./strcpy terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45) [0xb76ff095]
/lib/i386-linux-gnu/libc.so.6(+0x10404a) [0xb76ff04a]
./strcpy[0x804846e]
./strcpy[0x8048489]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0xb7614533]
./strcpy[0x80483a1]
===== Memory map: =====
```

Stack Smashing Example

```
(gdb) disas overflow
```

```
Dump of assembler code for function overflow:
```

```
0x08048434 <+0>:      push    %ebp
0x08048435 <+1>:      mov     %esp,%ebp
0x08048437 <+3>:      sub     $0x38,%esp
```

```
[...]
```

```
0x08048458 <+36>:     call    0x8048350 <strcpy@plt>
0x0804845d <+41>:     mov     -0xc(%ebp),%eax
0x08048460 <+44>:     xor     %gs:0x14,%eax
0x08048467 <+51>:     je      0x804846e <overflow+58>
0x08048469 <+53>:     call    0x8048340
                    <__stack_chk_fail@plt>
0x0804846e <+58>:     leave
0x0804846f <+59>:     ret
```

Exploit Mitigation – Stack Protector

Stack Canary

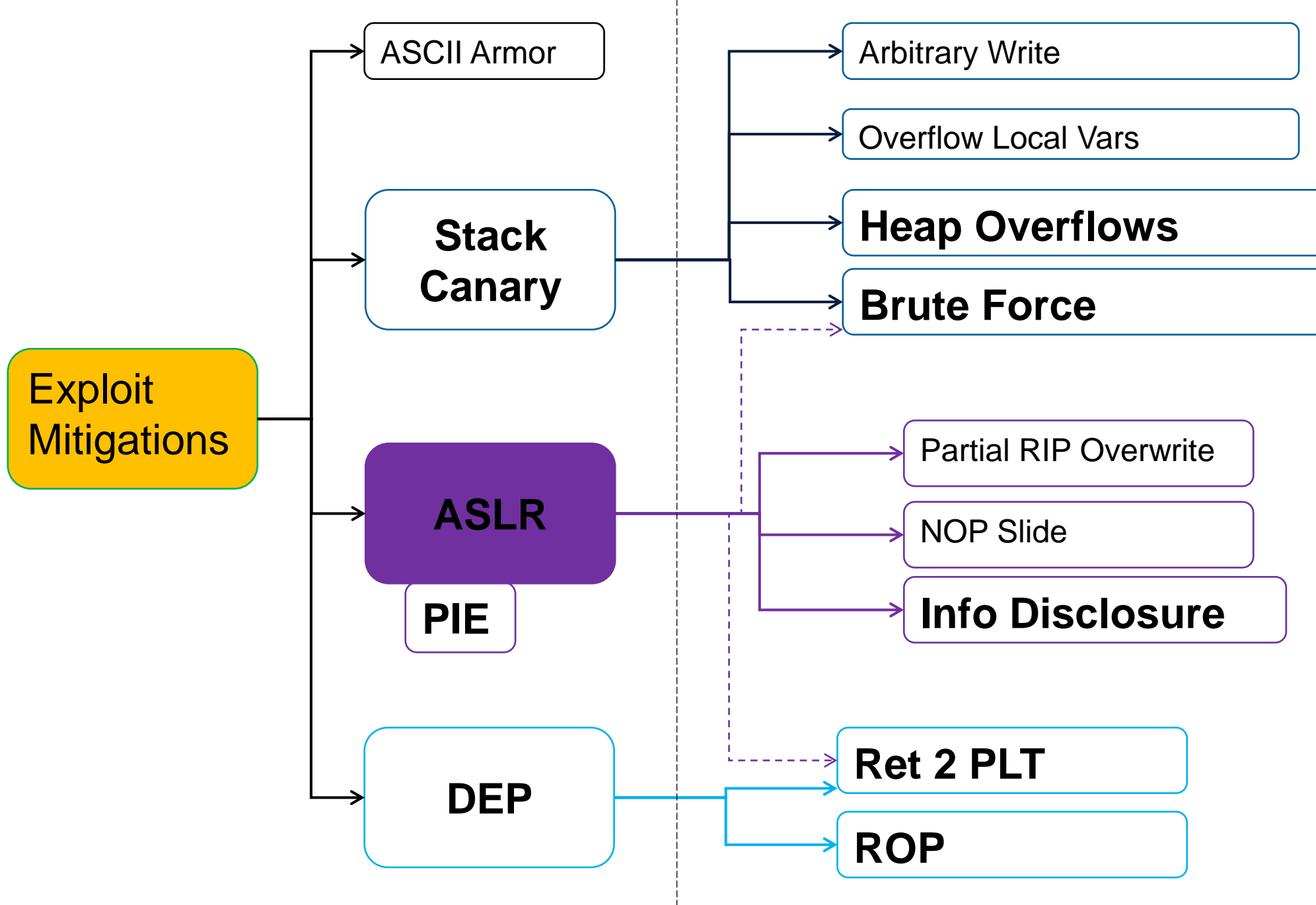


Exploit Mitigation – Stack Protector

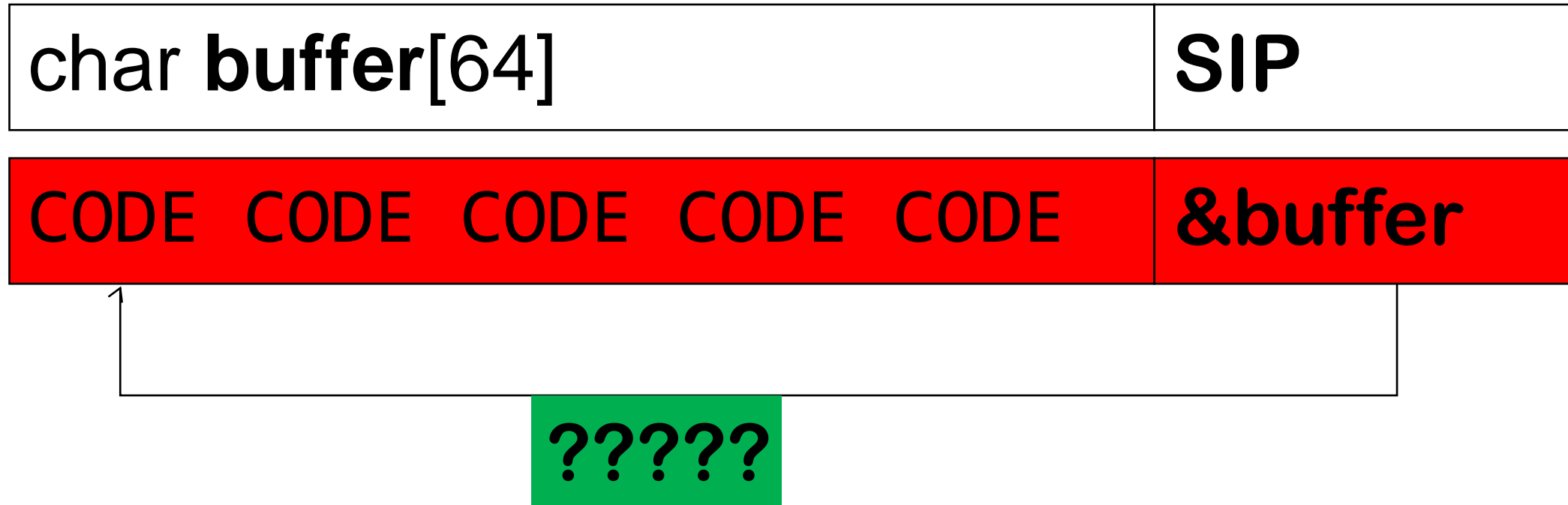
Arrival: Canary



Exploit Mitigation: ASLR



Exploit Mitigations: ASLR

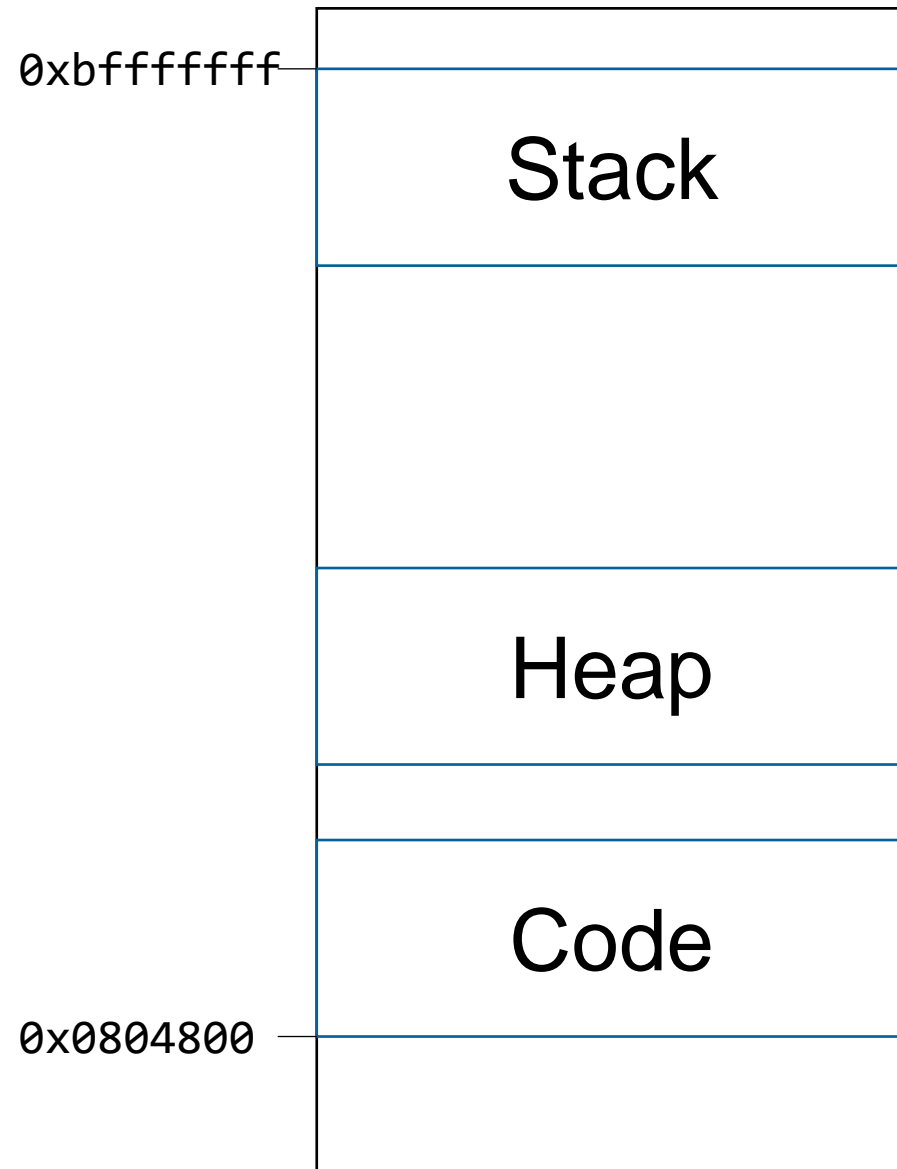


Exploit Mitigation - ASLR

Code execution is surprisingly deterministic

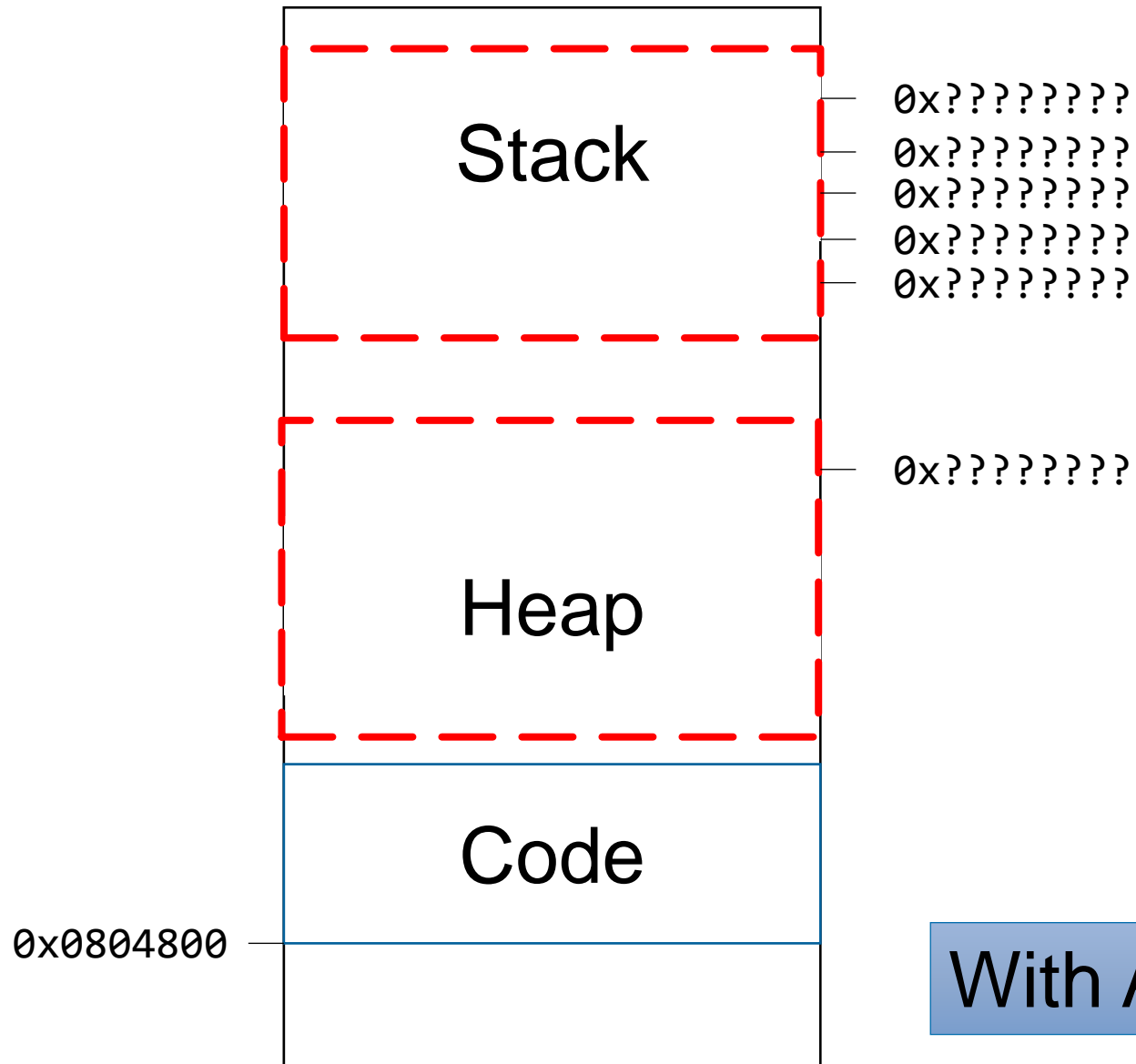
- E.g. Network service:
 - fork()
 - Parse incoming data
 - Buffer Overflow is happening at module X line Y
- On every exploit attempt, memory layout looks the same!
 - Same stack/heap/code layout
 - Same address of the buffer(s)
- ASLR: Address Space Layout Randomization
 - Introduces randomness in memory regions

Memory Layout

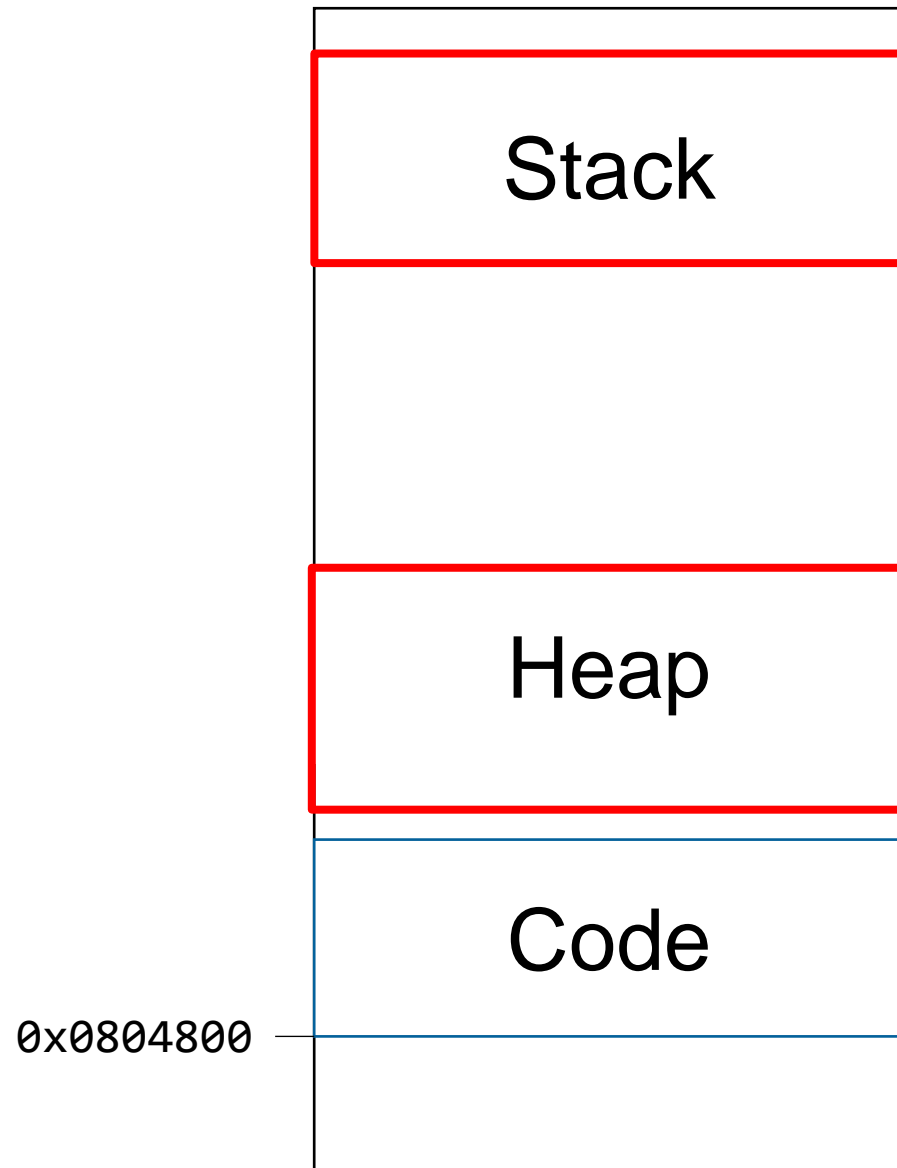


Without ASLR

Memory Layout

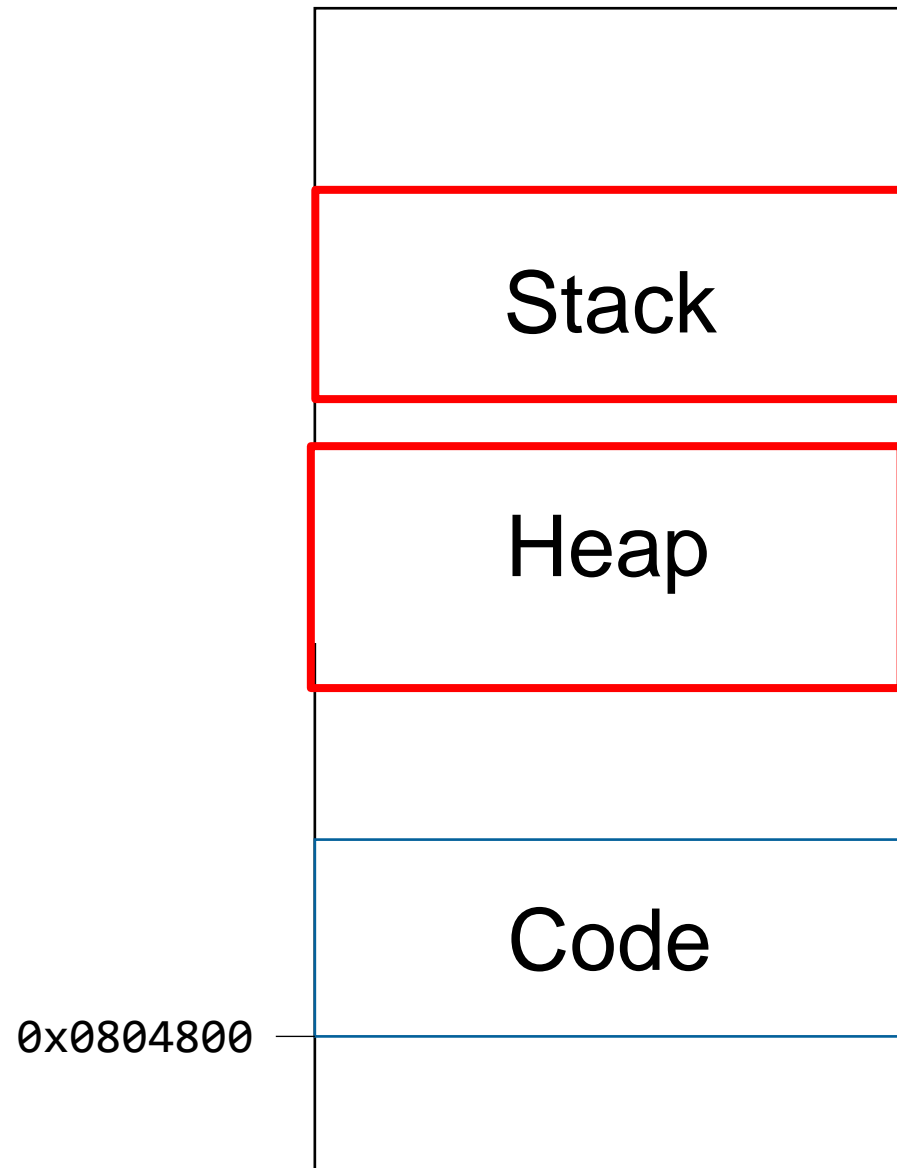


Memory Layout



With ASLR, #1

Memory Layout



With ASLR, #2

Exploit Mitigation - ASLR

0xAA00



0xAA00

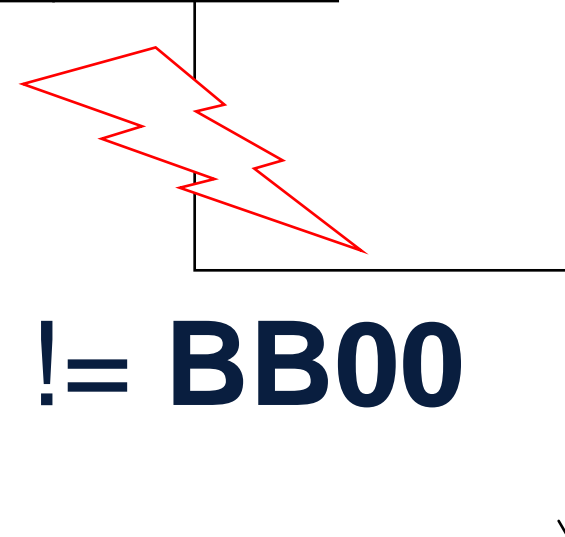


Exploit Mitigation - ASLR

0xBB00



0xBB00



“Segmentation Fault”

AA00 != BB00

Exploit Mitigation - ASLR

Randomness is measured in entropy

- Several restrictions
 - Pages have to be page aligned: 4096 bytes = 12 bit
- Very restricted address space in x32 architecture
 - ~8 bit for stack (256 possibilities)
- Much more space for x64
 - ~22 bit for stack

Exploit Mitigation - ASLR

Default ASLR:

- Stack
- Heap
- Libraries (new!)

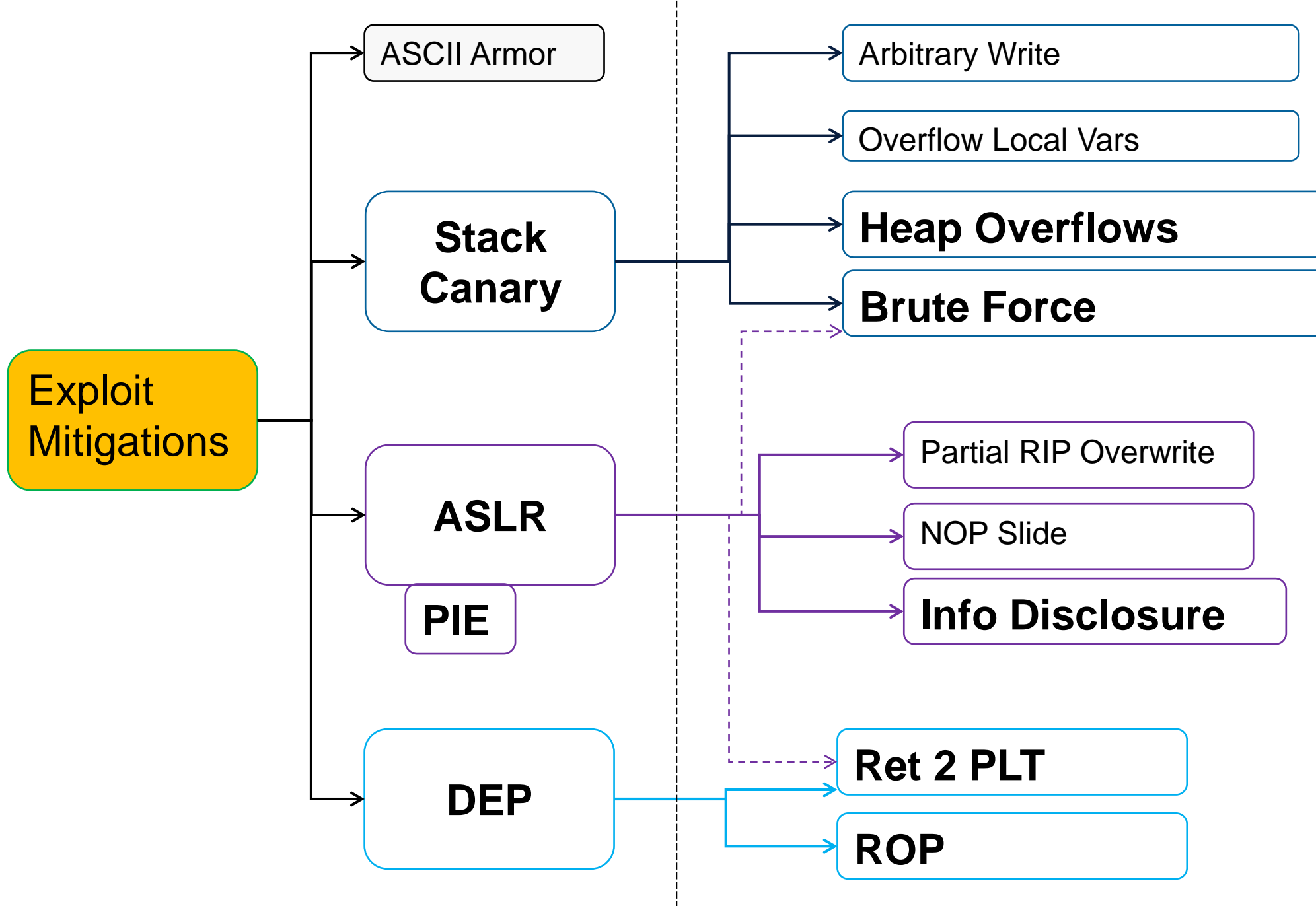
Re-randomization

- ASLR only applied on `exec()` [`exec` = execute new program]
- Not on `fork()` [`fork` = copy]

Recap! ASLR

Randomize Memory Layout

Attacker can't call/reference what he cant find



Exploit Mitigation: ASCII Armor

ASCII Armor:

- Maps Library addresses to memory addresses with null bytes

Exploit Mitigation: ASCII Armor

ASCII Armor:

- Maps Library addresses to memory addresses with null bytes

Why null bytes?

- In C, Null bytes are string determinator
- strcpy, strcat, strncpy, sprintf, ...

`strlen(AAAA\00BBBBB\00) = 4`

Exploit Mitigation: ASCII Armor

```
(gdb) info file
```

```
0x0000000000400980 - 0x0000000000400d92 is .text
```

```
0x0000000000400830 - 0x0000000000400980 is .plt
```

```
0x0000000000400980 - 0x0000000000400d92 is .text
```

```
0x00000000006011f8 - 0x0000000000601200 is .got
```

```
0x0000000000601200 - 0x00000000006012b8 is .got.plt
```

```
0x00007ffffff7b9ed80 - 0x00007ffffff7b9eff8 is .got in  
/lib/x86_64-linux-gnu/libc.so.6
```

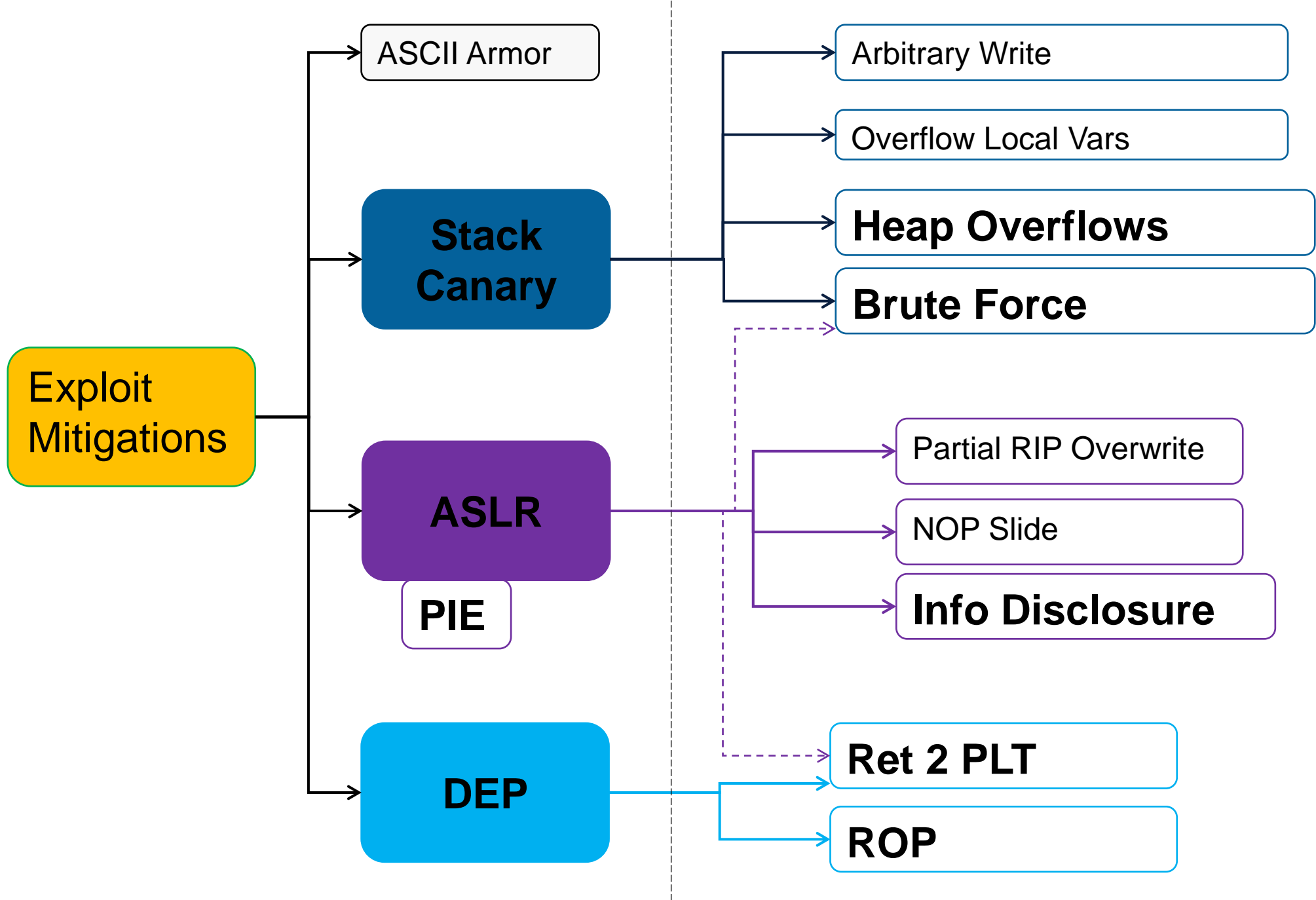
```
0x00007ffffff7b9f000 - 0x00007ffffff7b9f078 is .got.plt in  
/lib/x86_64-linux-gnu/libc.so.6
```

Exploit Mitigation: ASCII Armor

Recap:

- Putting important stuff at addresses with 0 bytes breaks strcpy etc.

Exploit Mitigation - Conclusion



Recap! All Exploit Mitigations

Stack canary: **detects/blocks** overflows

DEP: makes it impossible to **execute** uploaded code

ASLR: makes it impossible to **locate** data

ASCII Armor: makes it impossible to **insert** certain data

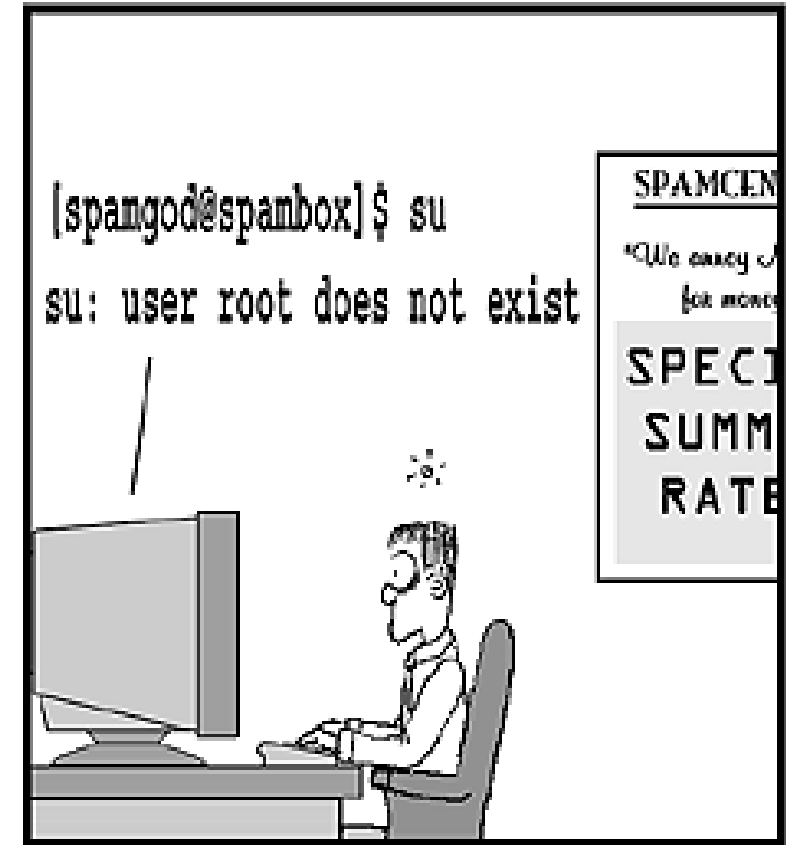
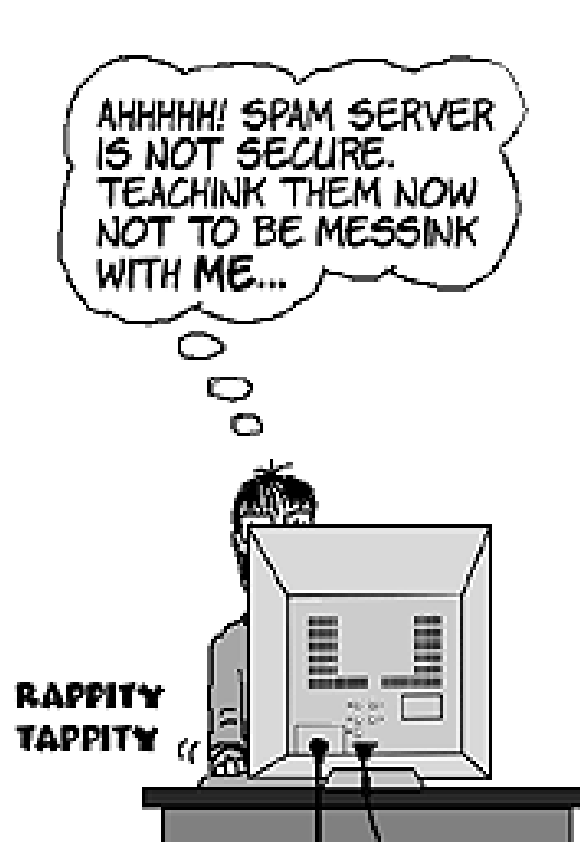
Recap! All Exploit Mitigations

USER FRIENDLY by Illiad



Copyright (c) 2000 Illiad

Copyright (c) 2000 Illiad



Anti Exploiting in Linux

How is the state of Exploit Mitigations in Linux?

Easy: Everything active by default!

ASLR: System-level

DEP: System level

Stack Canary: Per-program (3rd party programs?)

References

<https://www.elttam.com.au/blog/playing-with-canaries/>

- Playing with canaries
- Looking at SSP over several architectures.