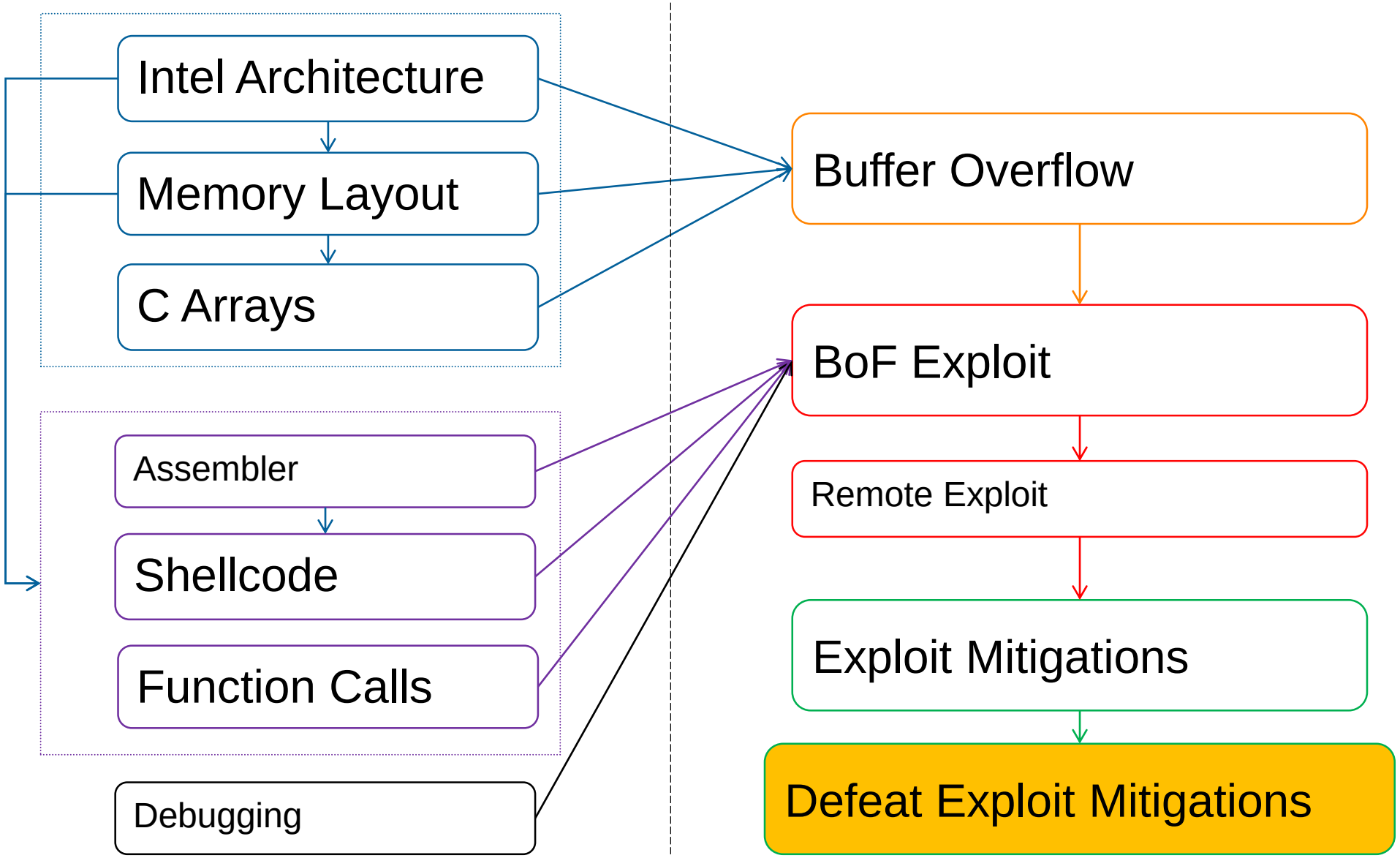
# Defeat Exploit Mitigations

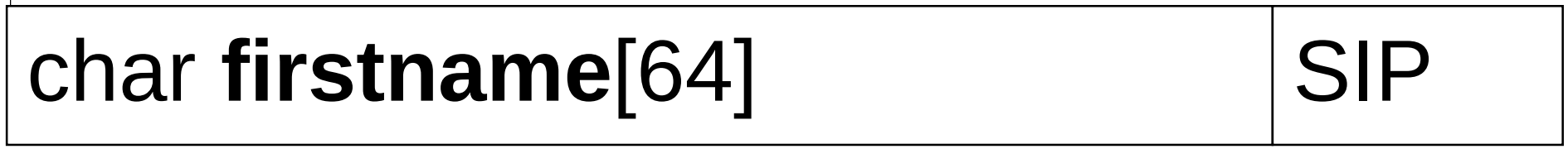
## Contemporary exploiting

# Content

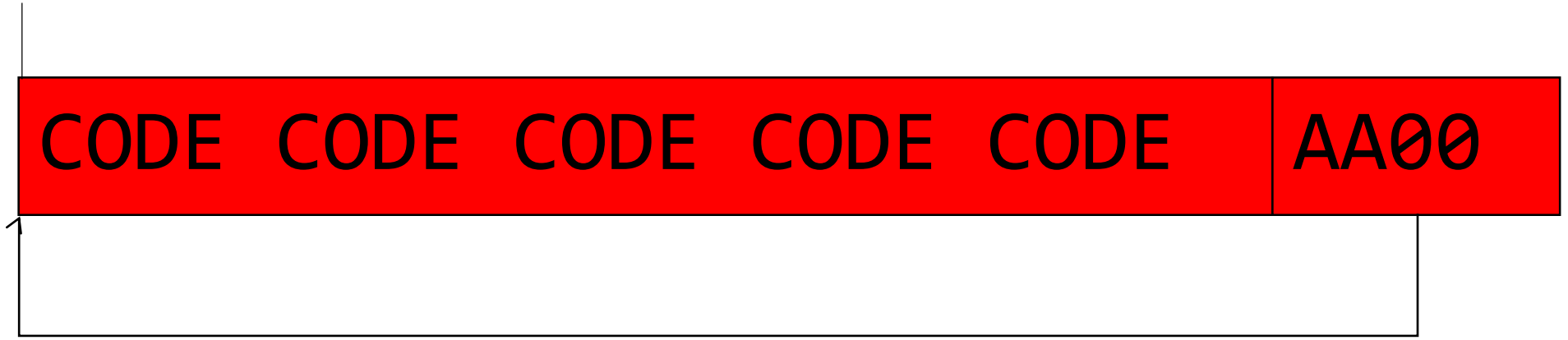

Intel Architecture
Memory Layout
C Arrays
Assembler
Shellcode
Function Calls
Debugging
Buffer Overflow
BoF Exploit
Remote Exploit
Exploit Mitigations
Defeat Exploit Mitigations

## Buffer Overflow Exploit

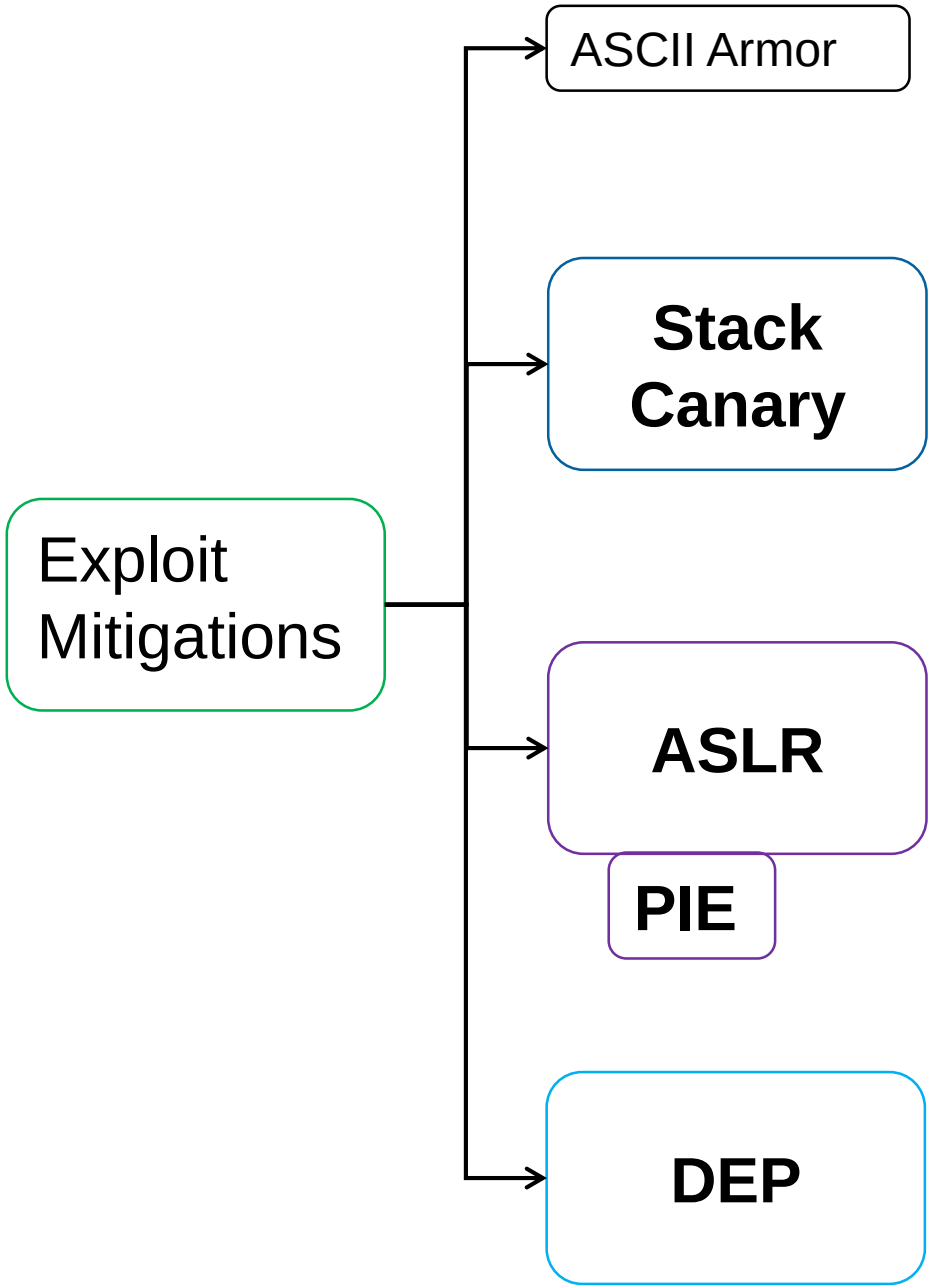
**0xAA00** (not the real address)

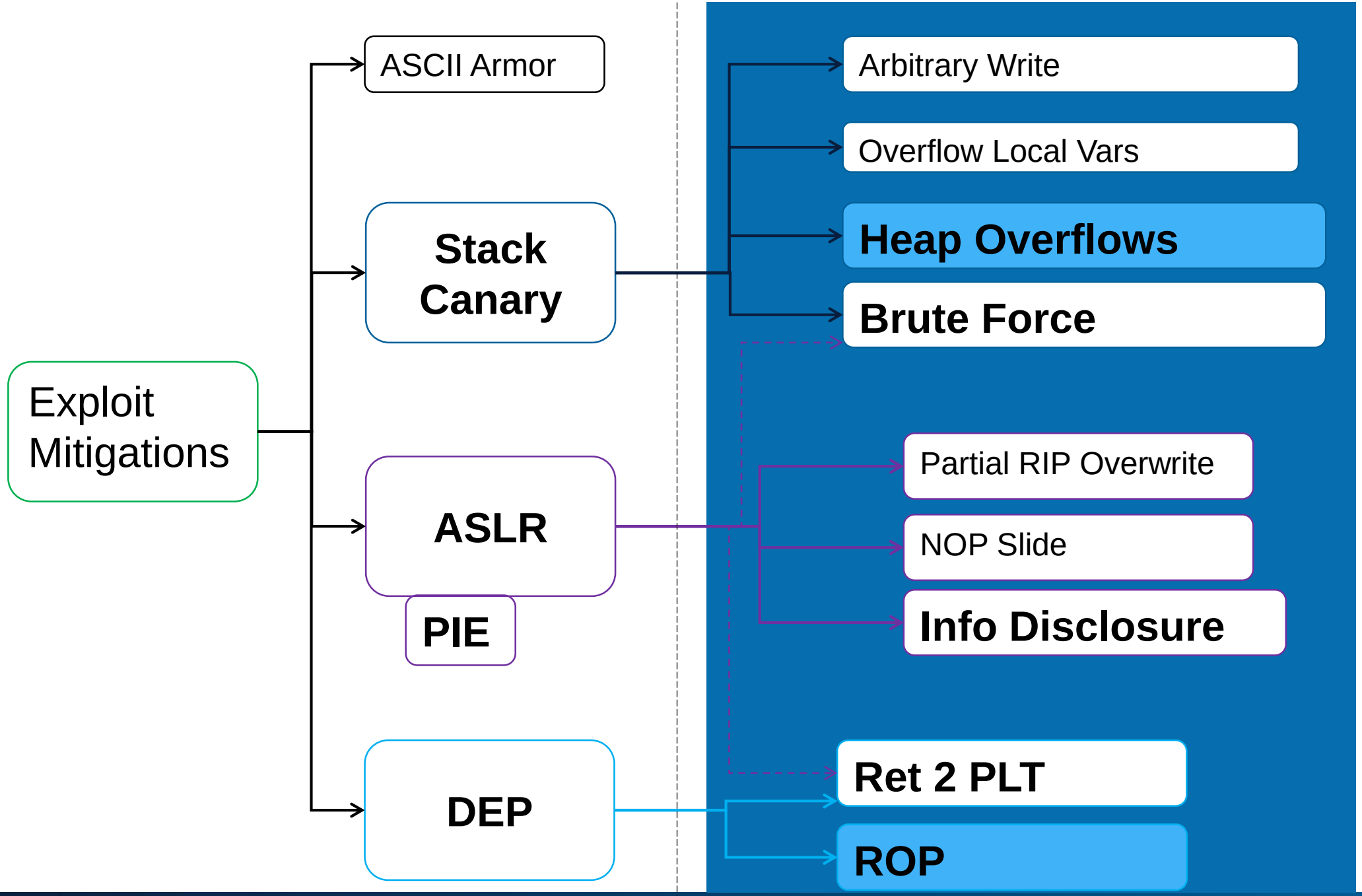



0xAA00




Jump to buffer with shellcode

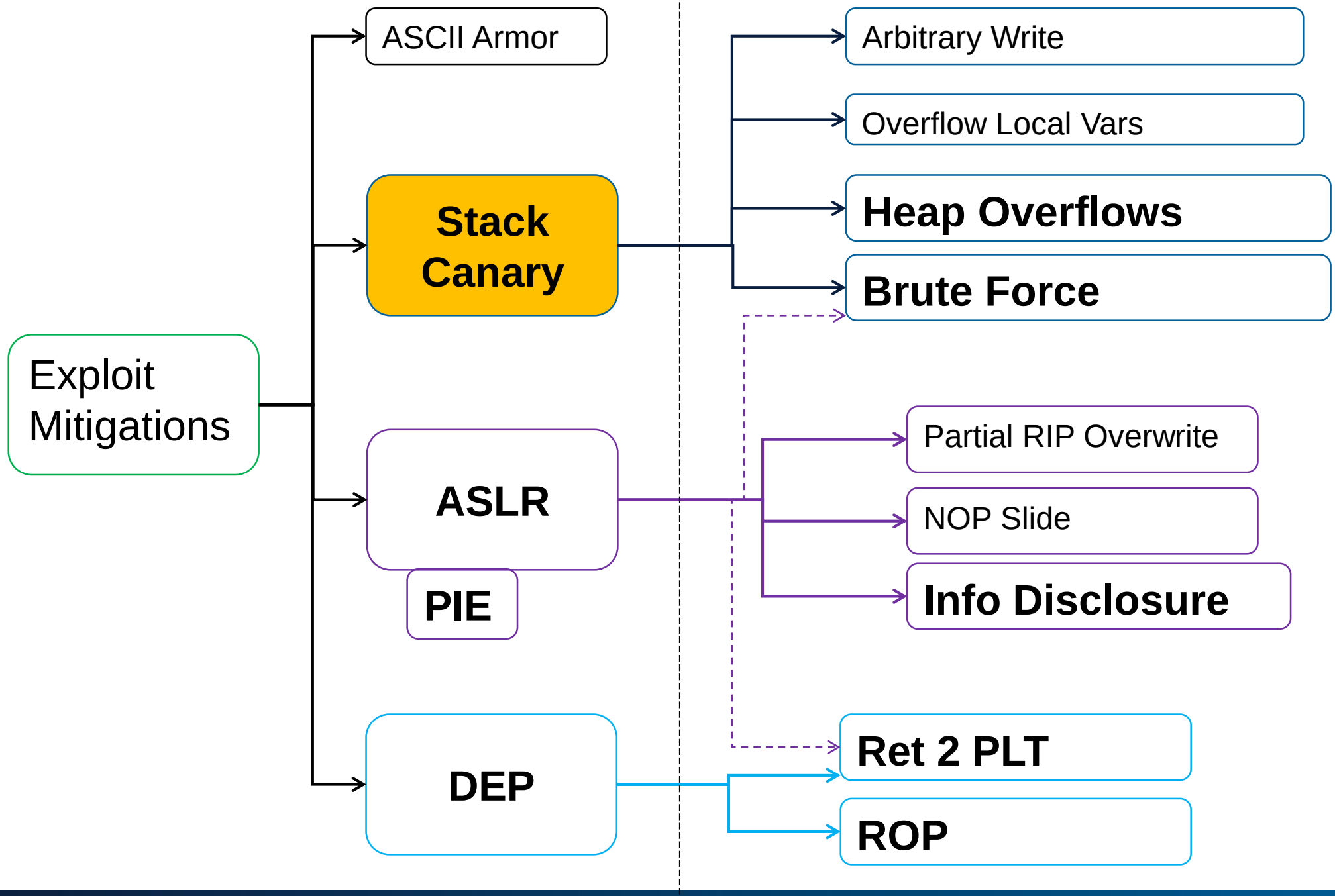

ASCII Armor
Stack Canary
Exploit Mitigations
ASLR
PIE
DEP


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Anti Exploit Mitigations

# Defeat Exploit Mitigations

Stack Canary


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating Stack Canary

Recap:

Stack Canary is a secret in front of SBP/SIP

Gets checked upon return()

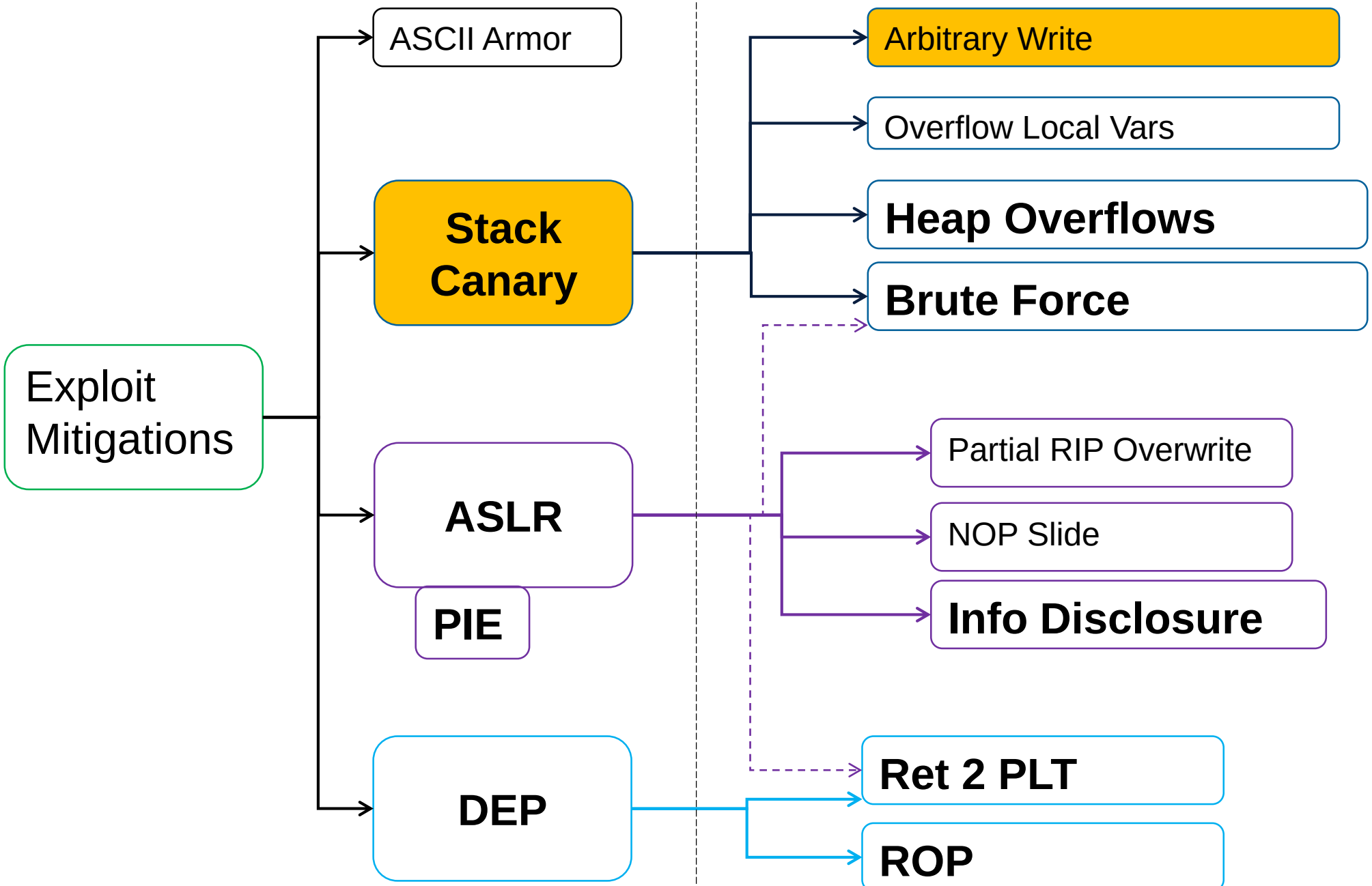
Prohibits overflows into SIP

# Defeating Stack Canary

Stack canary protects only stack **overflows into SIP**

**e.g:**

```
strcpy(a, b);
memcpy(a, b, len);
for(int n=0; n<len; n++) a[n] = b[n]
```


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating Stack Canary: Arb. Write

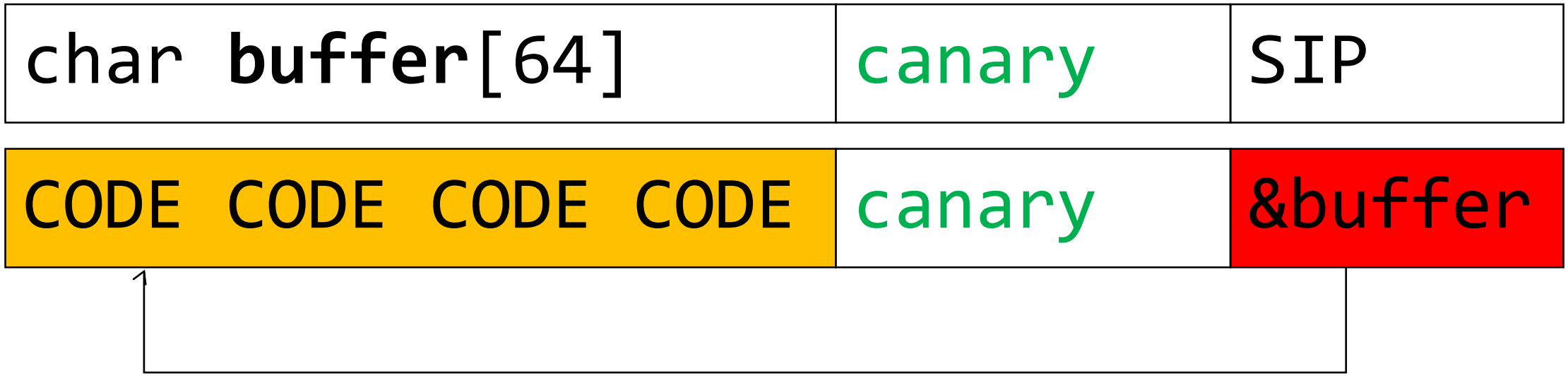
Arbitrary write:

```
char array[16];
array[userIndex] = userData;
char *a = &array;
a += 100;
*a = 0xdeadbeef;
```

- No overflow
- But: write “behind” stack canary

# Defeating Stack Canary: Arb. Write

Overwrite SIP without touching the canary:



# Defeating Stack Canary: Arb. Write

Example: Formatstring attacks

```
userData = "AAAA%204x%n";
```

Skip 204 bytes

# Defeating Stack Canary: Arb. Write

Wrong:

```
printf(userData);
```

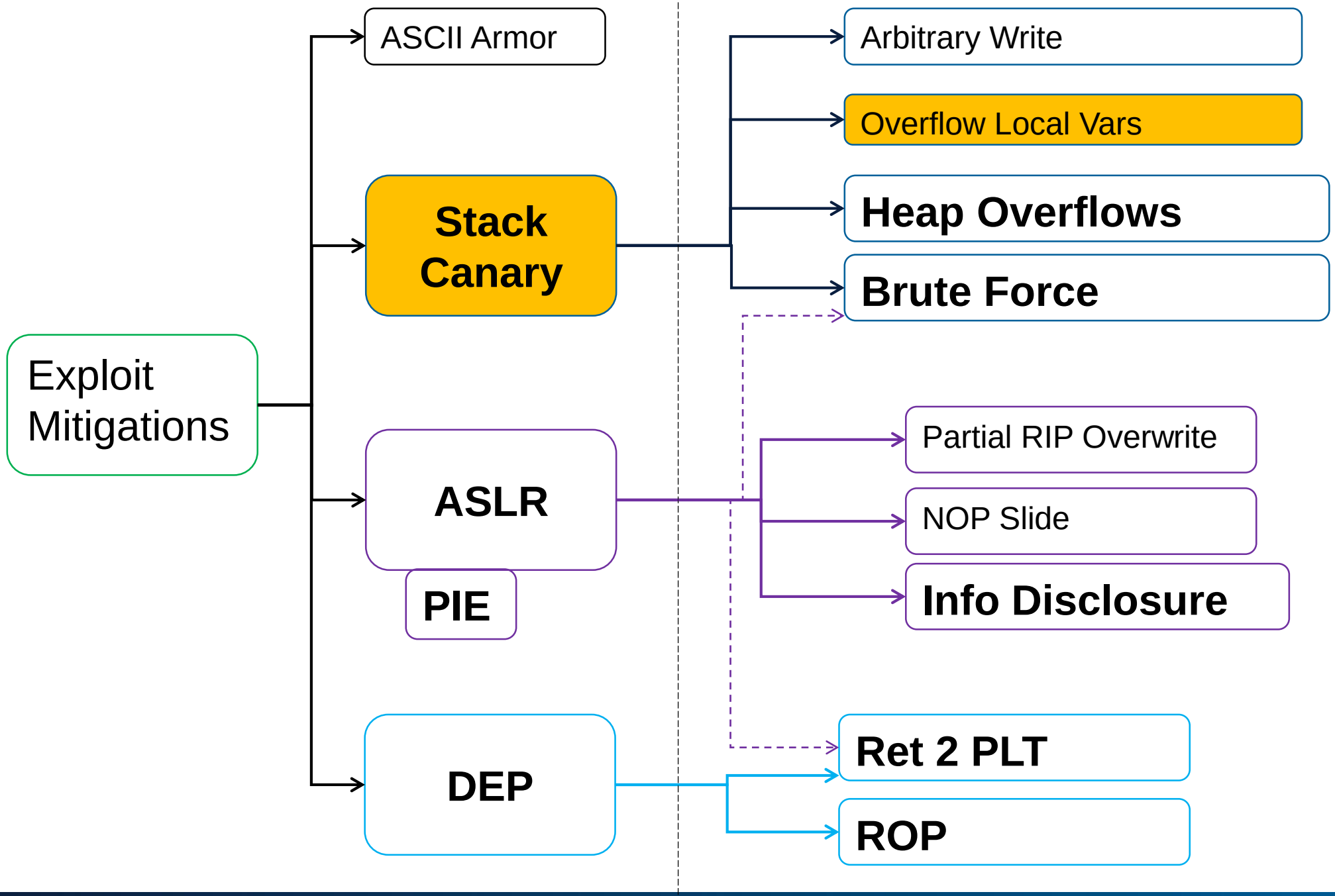
Correct:

```
printf("%s", userData);
```

# Defeating Stack Canary: Arb. Write

Example: Formatstring attacks

- Problem:
  - Did not specify format in source
  - Problem: %n writes data
- Nowadays:
  - Easy to detect on compile time (static analysis)
  - Easy to completely fix (remove %n)
  - Nowadays: Not a problem anymore, solved


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating Stack Canary: local vars

Stack canary protects metadata of the stack (SBP, SIP, …)

Not protected: **Local variables**

# Defeating Stack Canary: local vars

Overwrite local vars:

```
{
 void (*ptr)(char *) = &handleData;
 char buf[16];

 strcpy(buf, input);    // overflow
 (*ptr)(buf);           // exec ptr
}
```

# Defeating Stack Canary: local vars

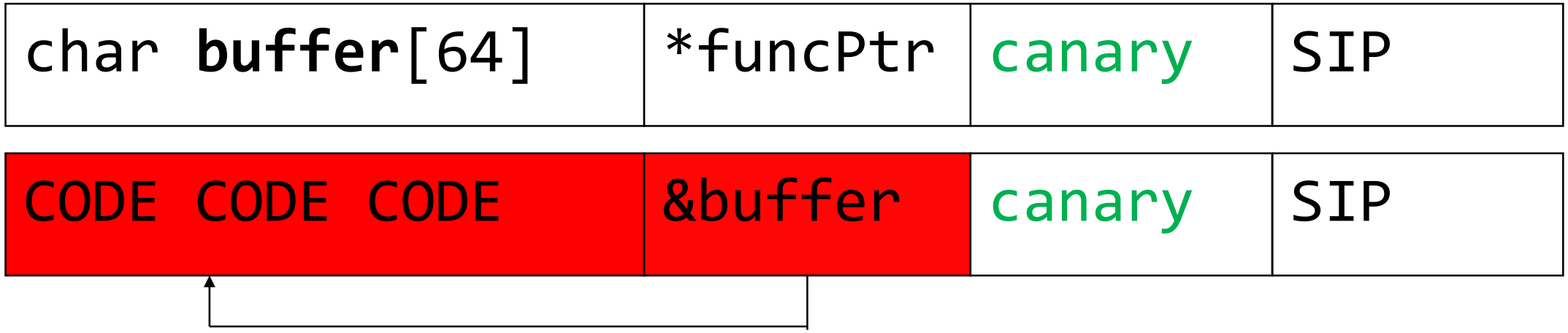
Overwrite local vars:

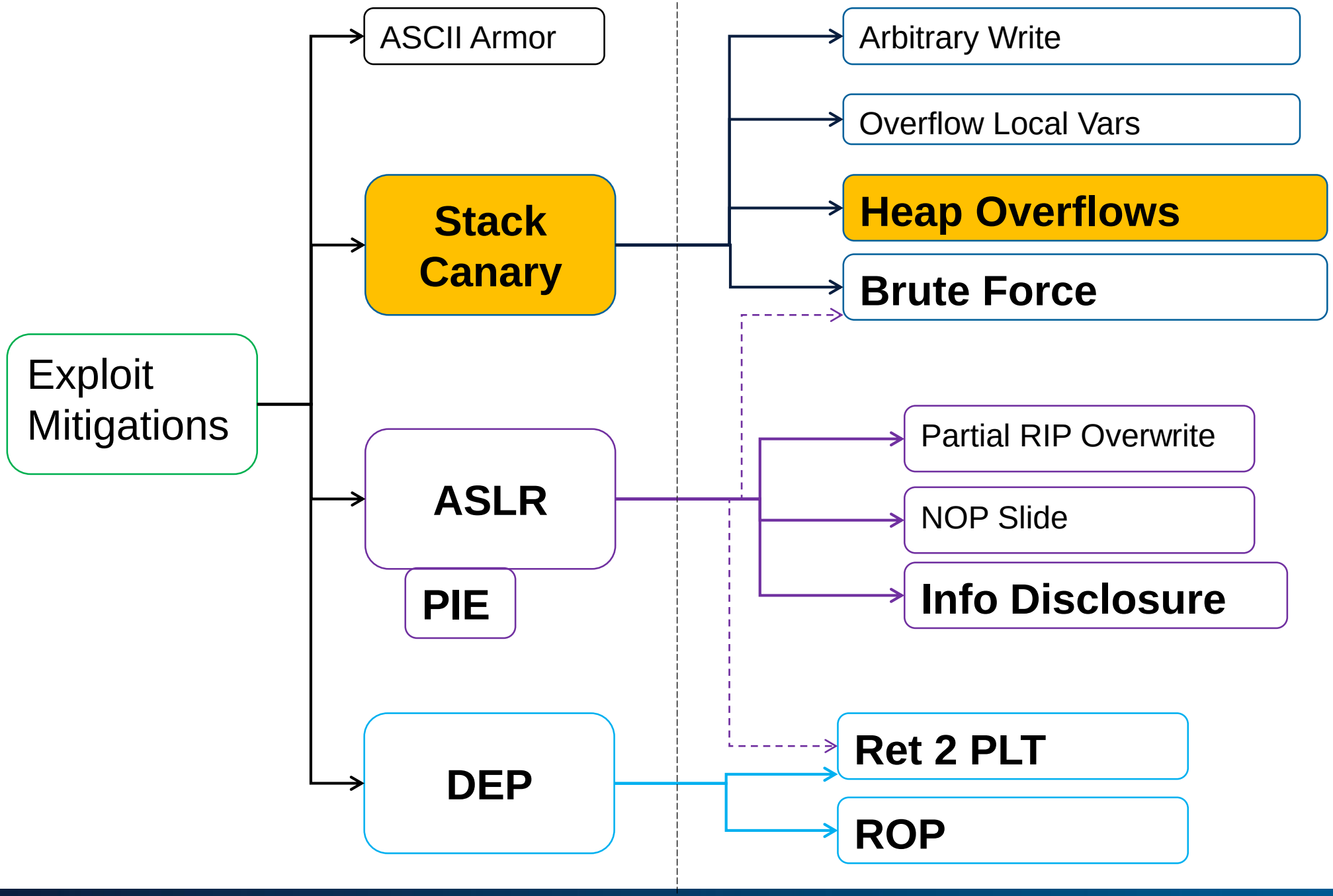
```
{
 void (*ptr)(char *) = &handleData;
 char buf[16];

 strcpy(buf, input);    // overflow
 (*ptr)(buf);           // exec ptr
}
```

Here: Possible to overwrite function pointers

# Defeating Stack Canary: Arb. Write

Overwrite a local function pointer:

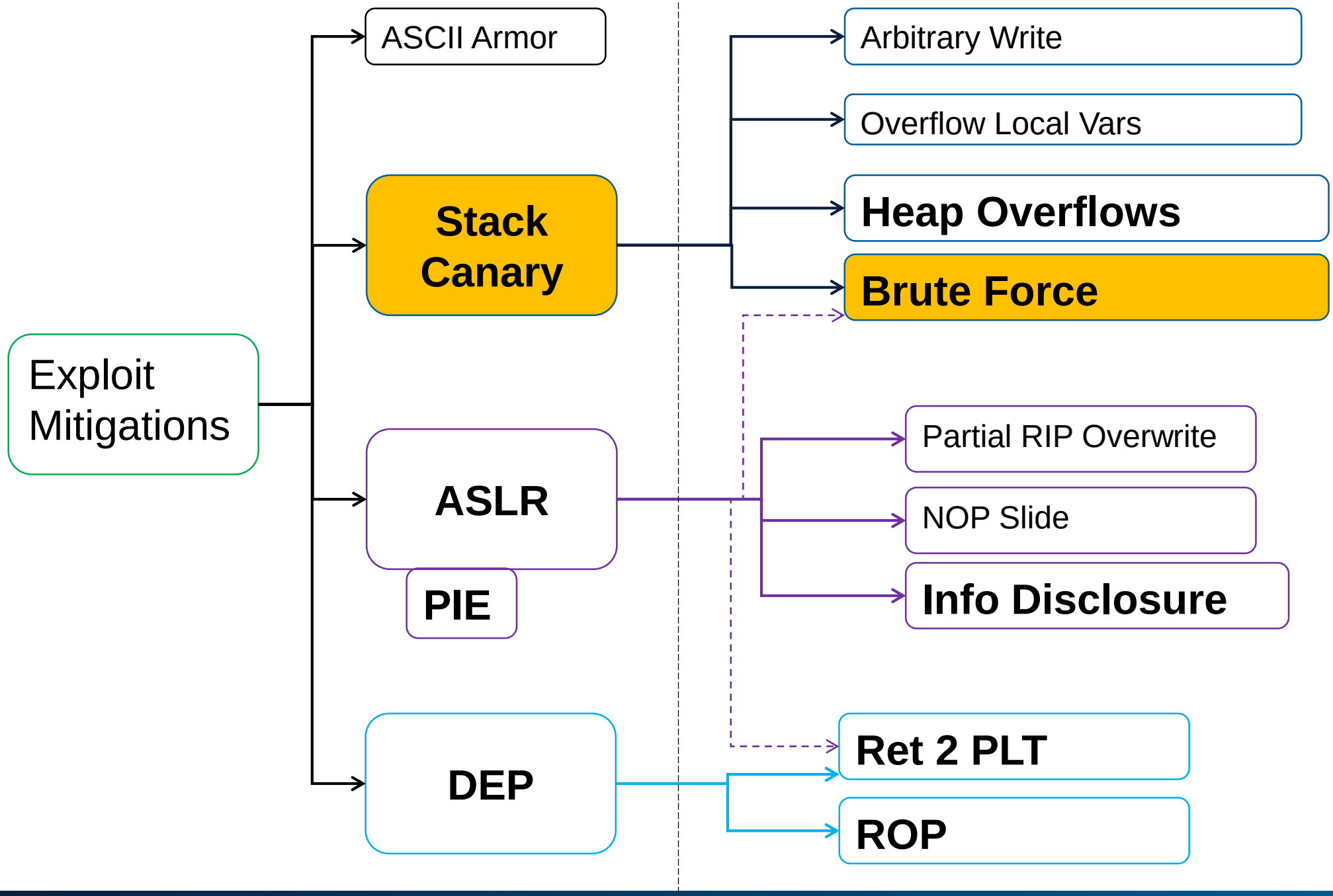



Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating Stack Canary: heap

Heap is not protected

- Heap bug classes:
  - Inter-chunck heap overflow/corruption
  - Use after free
  - Intra-chunk heap overflow / relative write
  - Type confusion
- -> Have an own, dedicated chapter


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating Stack Canary: Brute force

A network server fork()'s on connect()

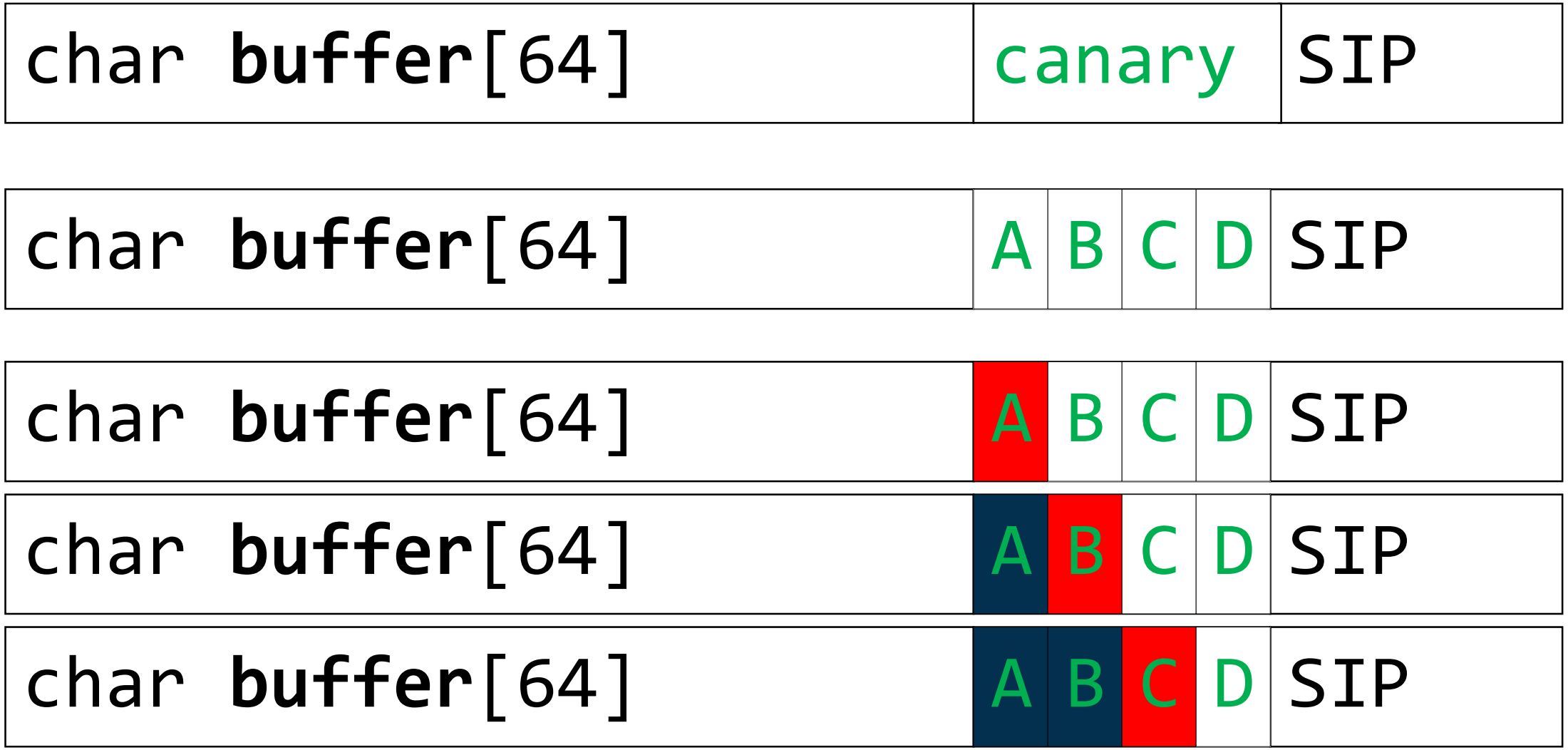
- If child crashes, next connection gets an "identical" child

But stack canary stay's the same

We can brute force it!

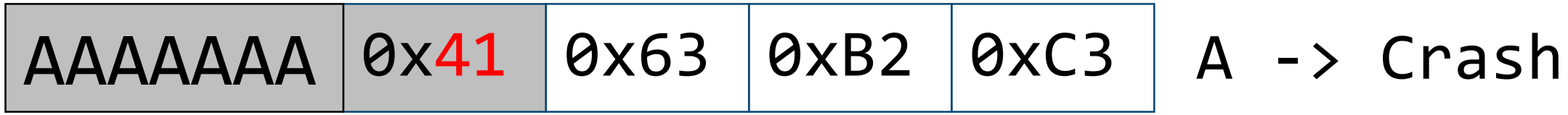
- 32 bit value, so 2^32 =~ 4 billion possibilities?

# Defeating Stack Canary: Brute force



# Defeating Stack Canary: Brute force

Example stack canary: **0xc3b26342**



# Defeating Stack Canary: Brute force

Example stack canary: **0xc3b26342**

| | | | | | |
|---|---|---|---|---|---|
| AAAAAAAA | 0x41 | 0x63 | 0xB2 | 0xC3 | A -> Crash |
| AAAAAAAA | 0x42 | 0x63 | 0xB2 | 0xC3 | B -> No crash |

# Defeating Stack Canary: Brute force

Example stack canary: **0xc3b26342**

| | | | | | |
|---|---|---|---|---|---|
| AAAAAAAA | 0x41 | 0x63 | 0xB2 | 0xC3 | A -> Crash |
| AAAAAAAA | 0x42 | 0x63 | 0xB2 | 0xC3 | B -> No crash |
| AAAAAAAA | 0x42 | 0x61 | 0xB2 | 0xC3 | Ba -> Crash |

# Defeating Stack Canary: Brute force

Example stack canary: **0xc3b26342**

| | | | | | |
|---|---|---|---|---|---|
| AAAAAAAA | 0x41 | 0x63 | 0xB2 | 0xC3 | A -> Crash |
| AAAAAAAA | 0x42 | 0x63 | 0xB2 | 0xC3 | B -> No crash |
| AAAAAAAA | 0x42 | 0x61 | 0xB2 | 0xC3 | Ba -> Crash |
| AAAAAAAA | 0x42 | 0x62 | 0xB2 | 0xC3 | Bb -> Crash |

# Defeating Stack Canary: Brute force

Example stack canary: **0xc3b26342**

| | | | | | |
|---|---|---|---|---|---|
| AAAAAAAA | 0x41 | 0x63 | 0xB2 | 0xC3 | A -> Crash |
| AAAAAAAA | 0x42 | 0x63 | 0xB2 | 0xC3 | B -> No crash |
| AAAAAAAA | 0x42 | 0x61 | 0xB2 | 0xC3 | Ba -> Crash |
| AAAAAAAA | 0x42 | 0x62 | 0xB2 | 0xC3 | Bb -> Crash |
| AAAAAAAA | 0x42 | 0x63 | 0xB2 | 0xC3 | Bc -> No Crash |

# Defeating Stack Canary: Brute force

So: not 2^32 = 4 billion possibilities

But:

```
4 * 2^8 =
4 * 256 =
1024 possibilities
```

512 tries (crashes) on average

# Defeating Stack Canary: Brute force

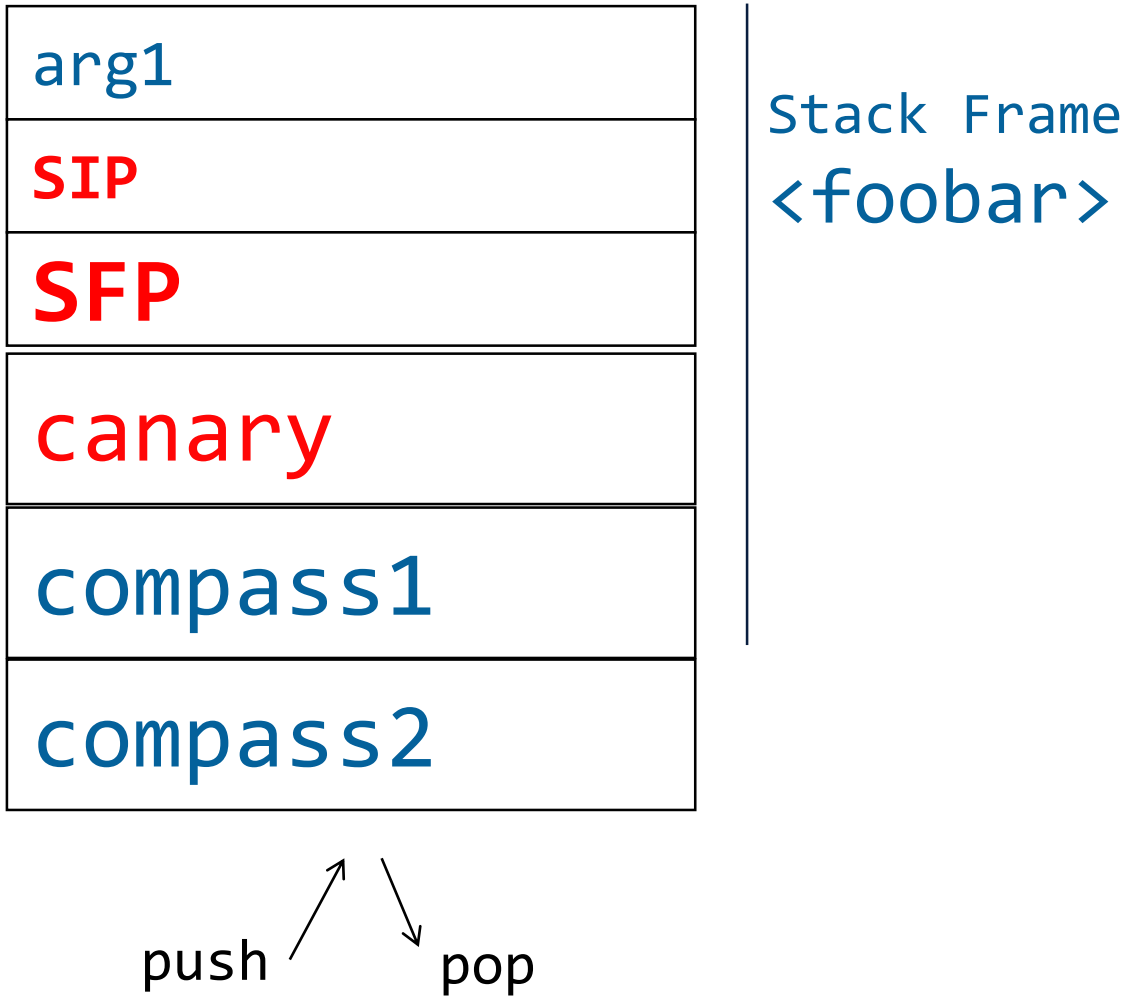
**I forgot… SFP**

Argument for <foobar>

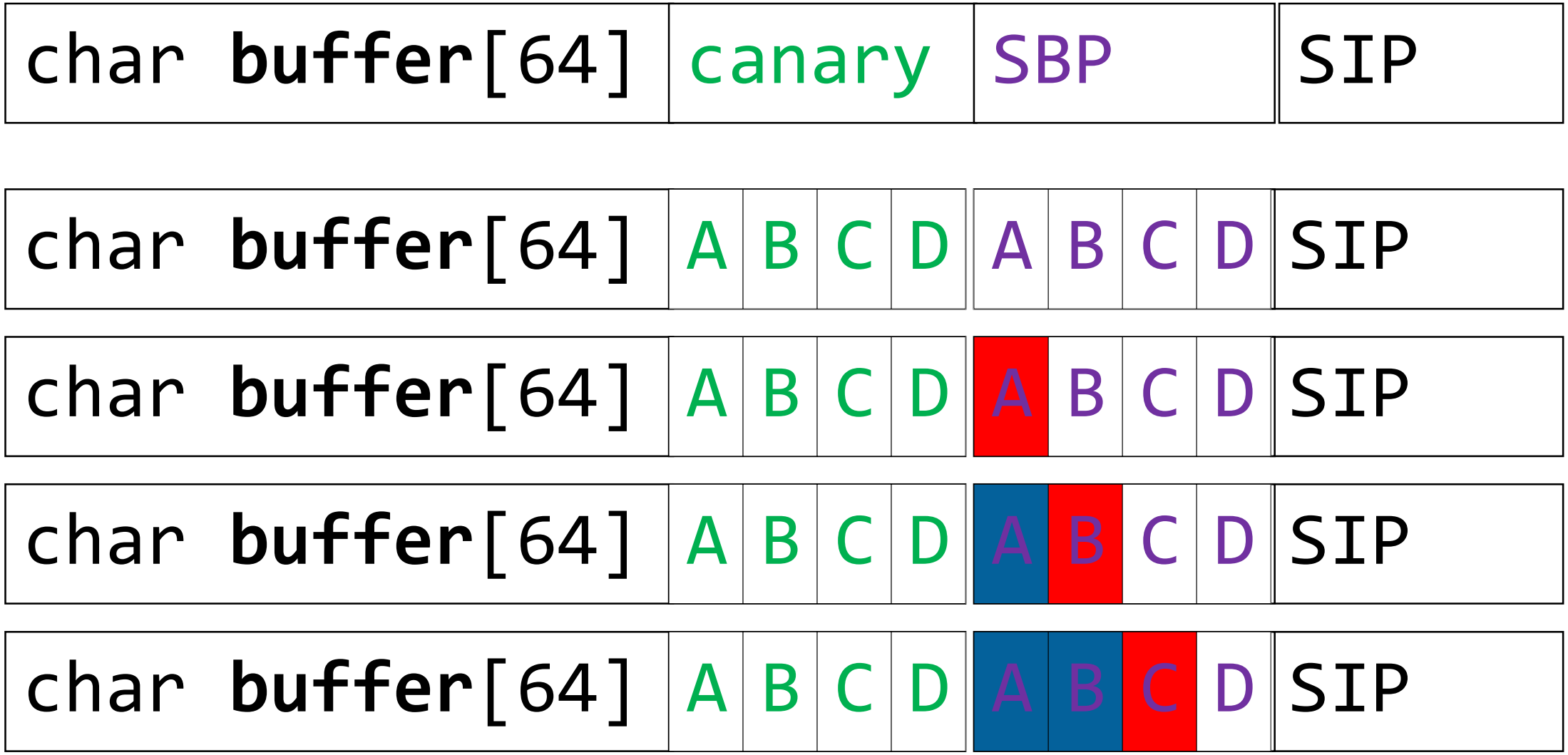
**Saved IP (&main)**

**Saved Frame Pointer**

Local Variables <func>



# Defeating Stack Canary: Brute force



# Defeating Stack Canary: Brute force

Need to break SBP first…

Defeat ASLR for free, because brute force SBP ☺

- (SBP points into stack segment)

# Recap: Defeating Stack Canary

Conclusion: Stack Canary:

Can be just circumvented

- With the right vulnerability

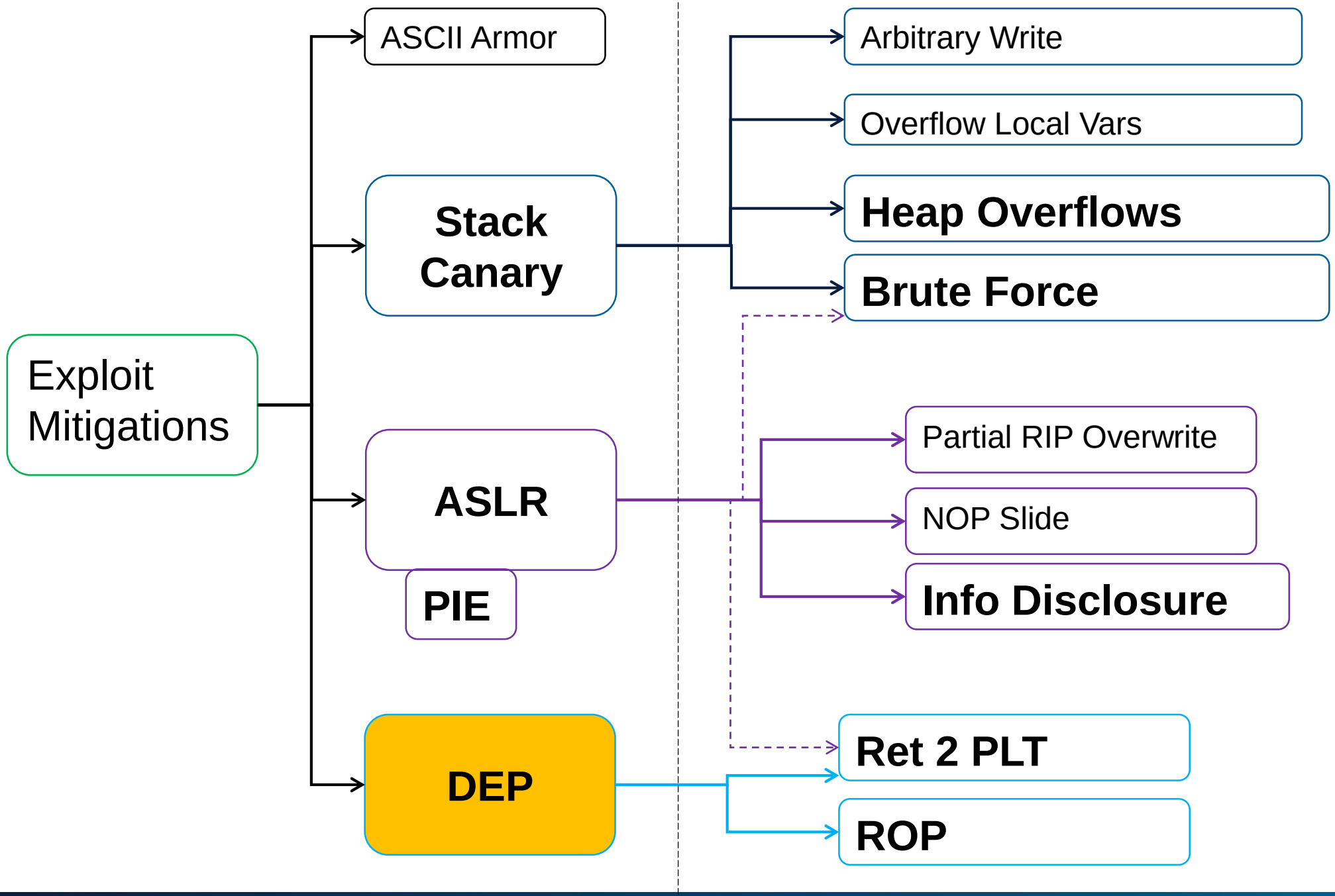
Or brute-forced

- If the vulnerable program is a network server

# Recap: Defeating Stack Canary

# Defeat Exploit Mitigations

Defeating: DEP

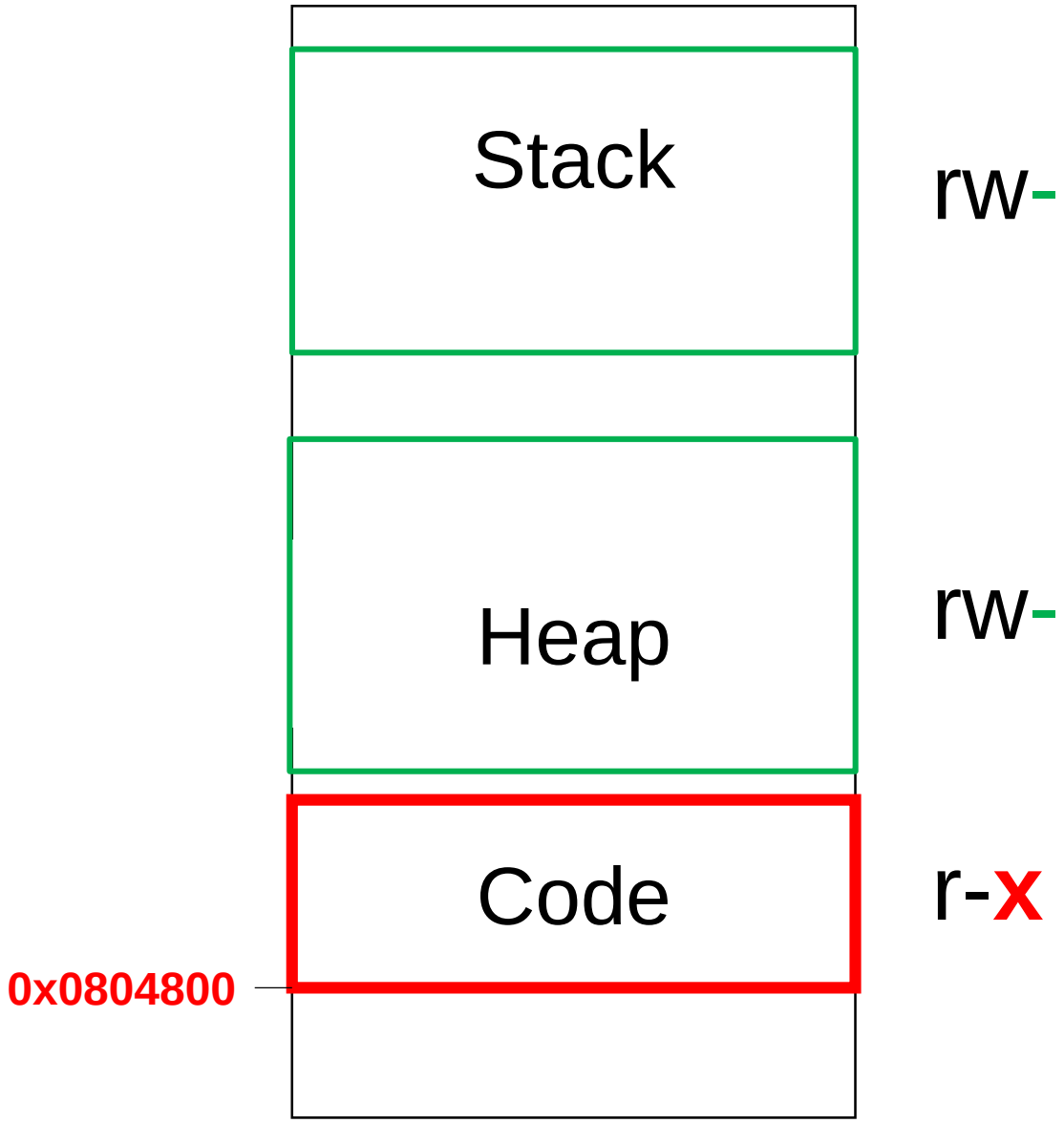

Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

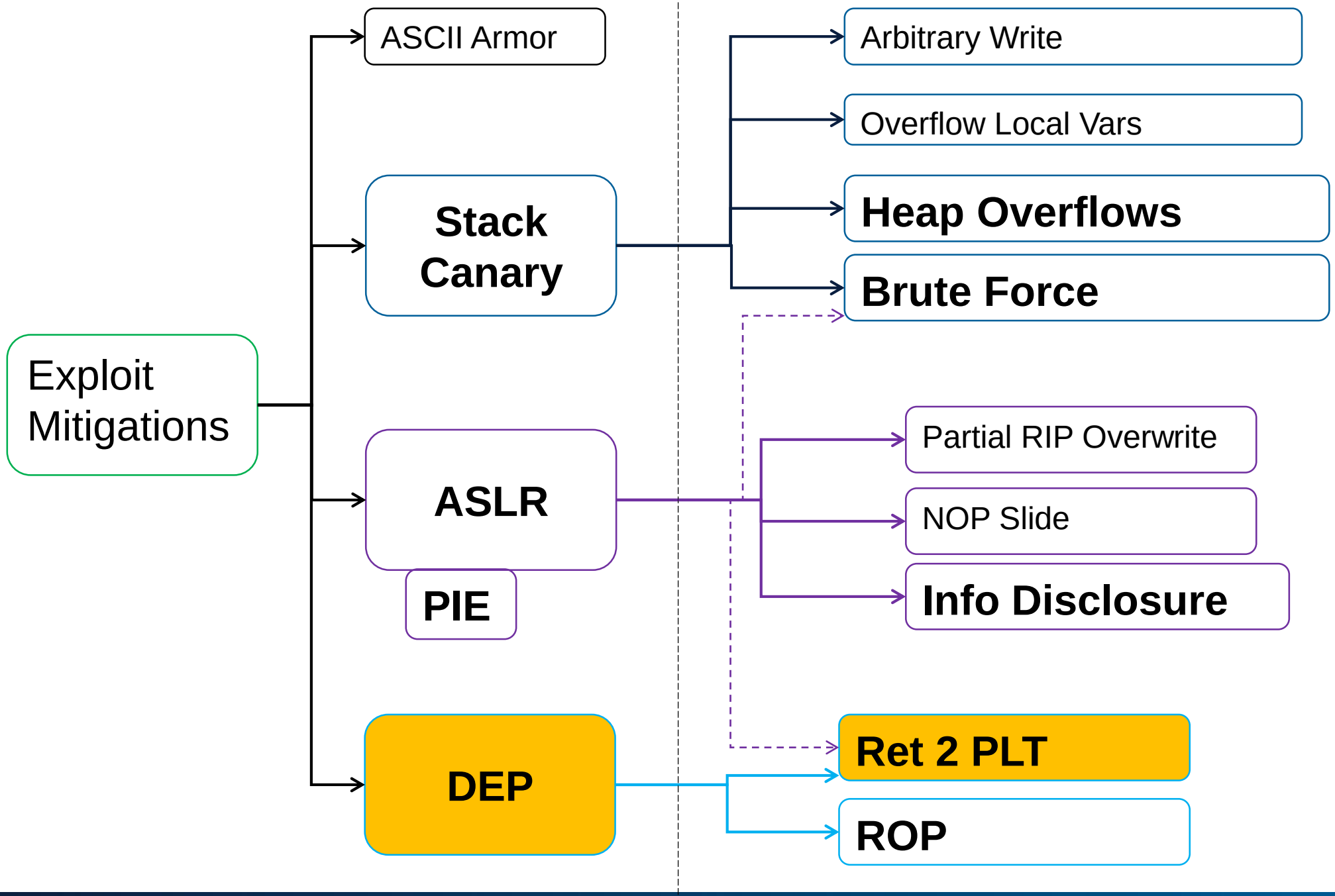
# Defeating DEP

Recap:
DEP makes Stack and Heap non-executable

- Shellcode cannot be executed anymore

# Defeating DEP - Intro




Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating DEP - Intro

DEP does not allow execution of uploaded code

But what about existing code?

- Existing LIBC Functions (ret2plt)
- Existing Code (ROP)

# Defeating DEP – Ret2plt

Solution:

- ret2libc / ret2got / ret2plt

# Defeating DEP – Ret2plt

Introducing shared libraries!

- Like windows DLL's
- Located in /lib and other directories
- Often end in ".so"

- Provide shared functionality
- E.g. libc, openssl, and much more

- Use "ldd" to check shared libraries

# Defeating DEP – Ret2plt

```
$ ldd /bin/bash
        linux-gate.so.1 =>  (0xb7724000)
        libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb76f9000)
        libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb76f4000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb754a000)
        /lib/ld-linux.so.2 (0xb7725000)
```

# Defeating DEP – Ret2plt

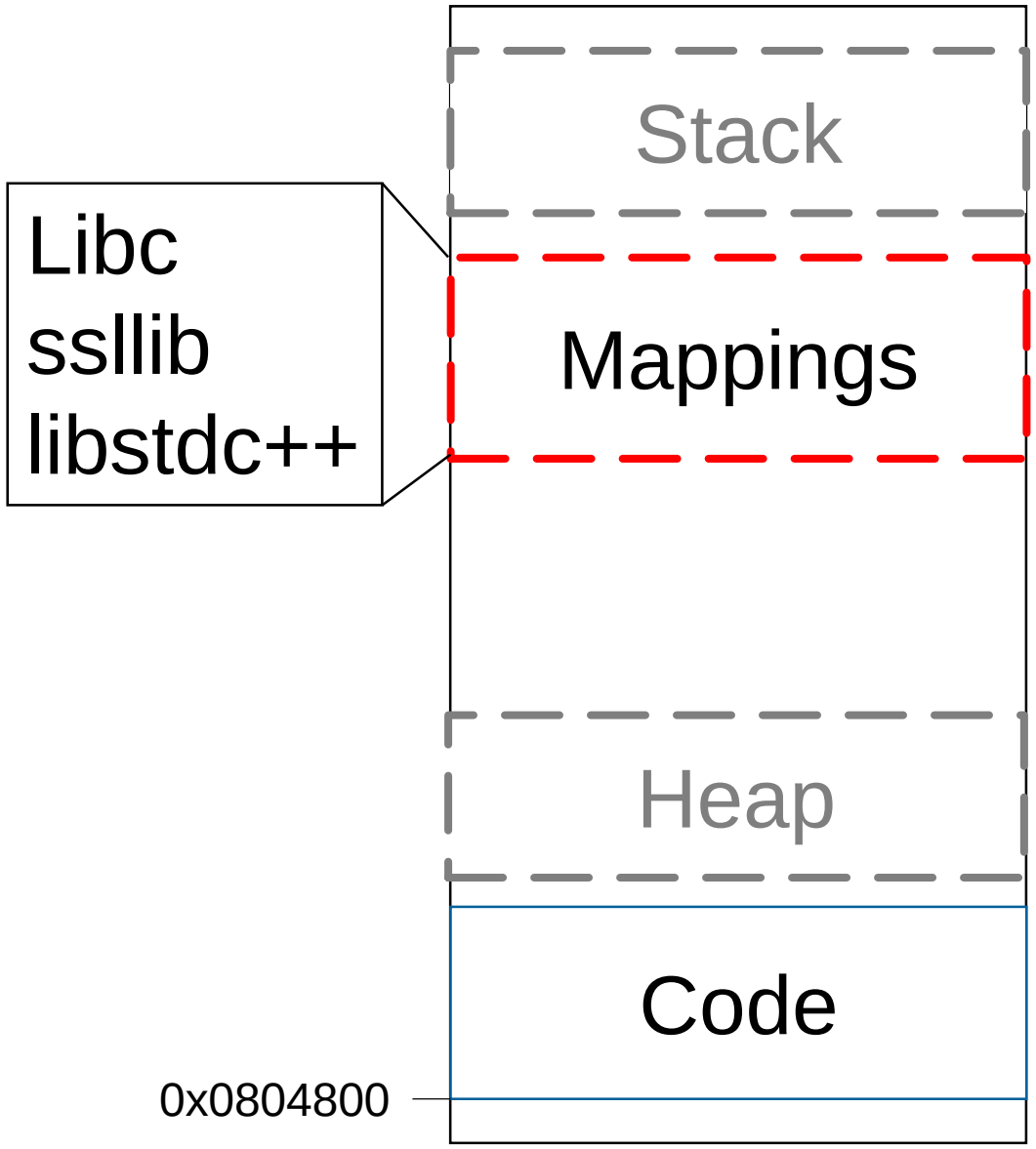
```
$ ldd `which nmap`
        linux-gate.so.1 =>  (0xb777f000)
        libpcap.so.0.8 => /usr/lib/i386-linux-gnu/libpcap.so.0.8
        libssl.so.1.0.0 => /lib/i386-linux-gnu/libssl.so.1.0.0
        libcrypto.so.1.0.0 => /lib/i386-linux-gnu/libcrypto.so.1.0.0
        libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7532000)
        libstdc++.so.6 => /usr/lib/i386-linux-gnu/libstdc++.so.6
        libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xb7421000)
        libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xb7403000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7259000)
        libz.so.1 => /lib/i386-linux-gnu/libz.so.1 (0xb7243000)
        /lib/ld-linux.so.2 (0xb7780000)
```

# Defeating DEP – Ret2plt

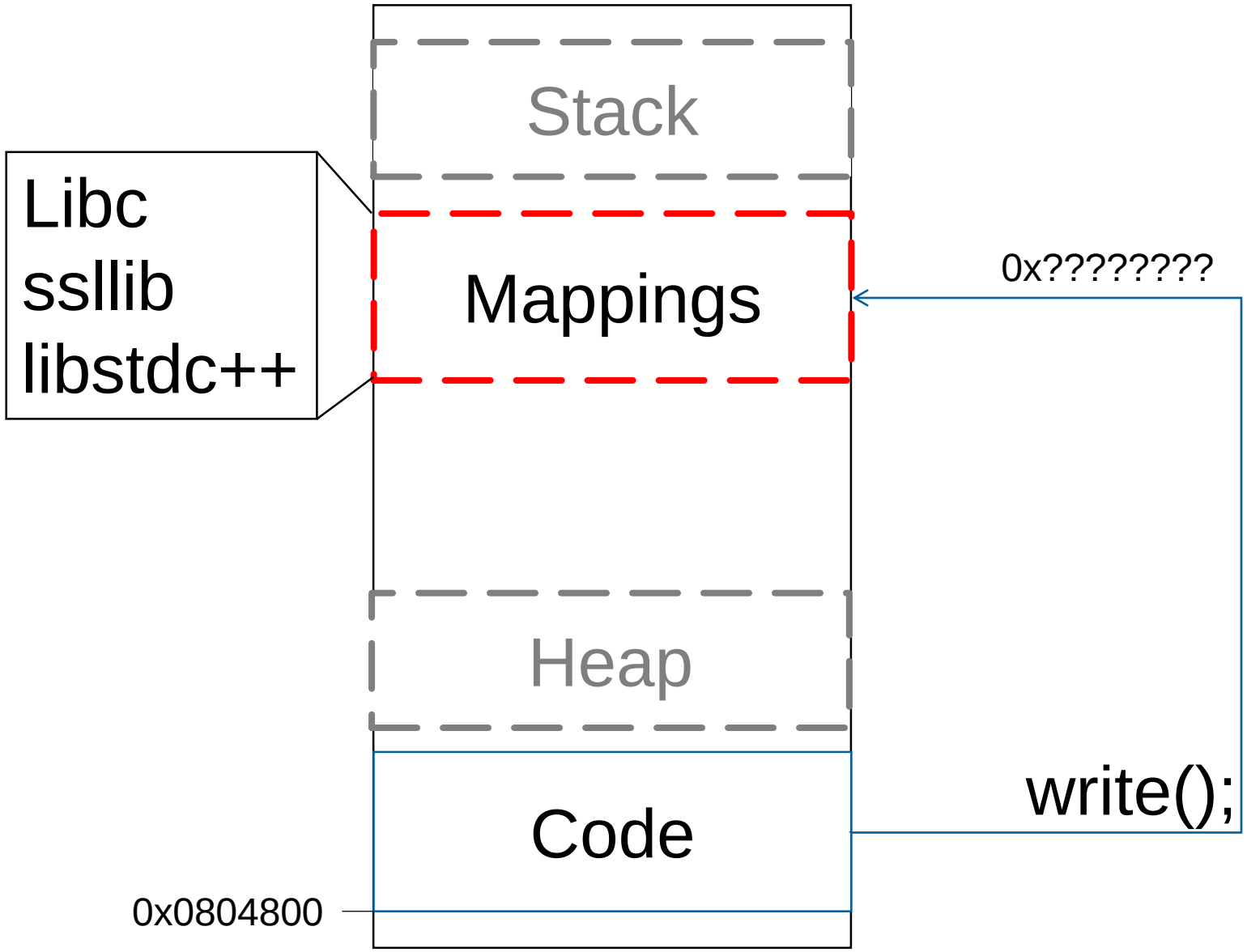
Shared Library Properties

- Shared libraries reference a certain version of a library
- Shared libraries can:
  - Be updated (grow in size)
  - Load in arbitrary order
- Therefore: Unknown exact location of shared library in memory space!

# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt

Call's in ASM are ALWAYS to absolute addresses

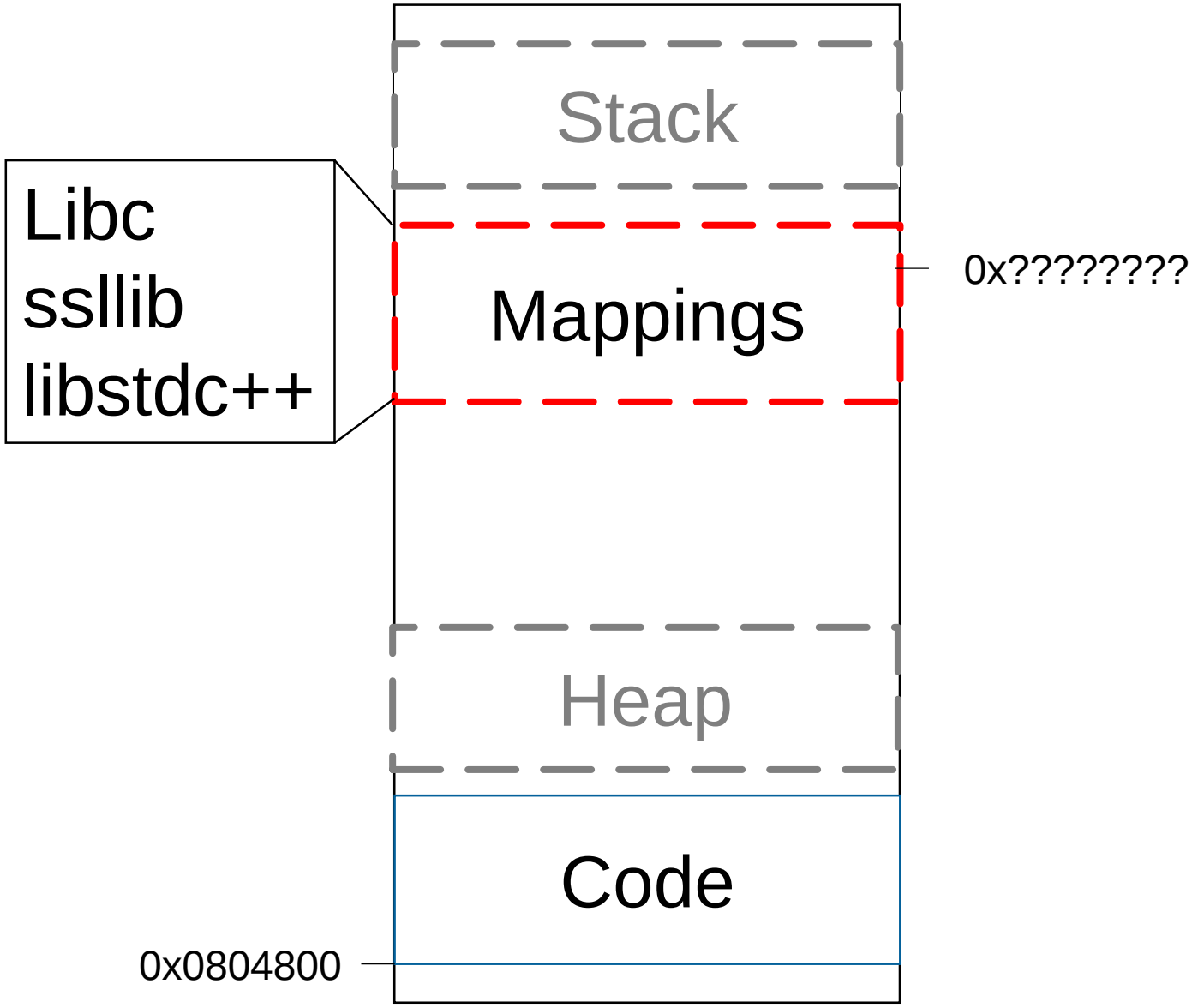
```
e8 d5 38 fd ff       call   805e4c0 <strlen@plt>
```

How does it work with dynamic addresses for shared libraries?

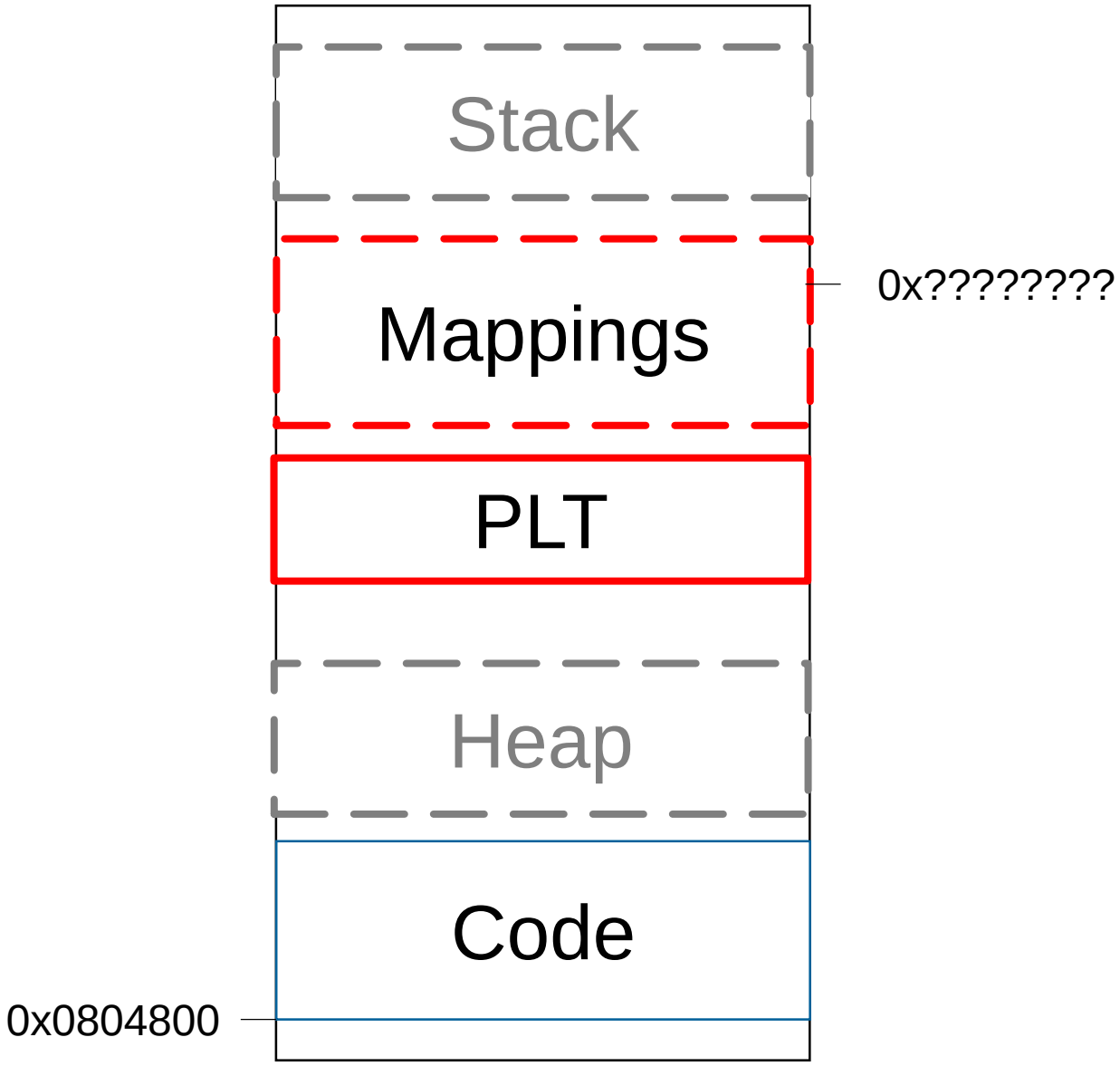
Solution:

- A "helper" at a static location
- In Linux: PLT+GOT (they work together in tandem)

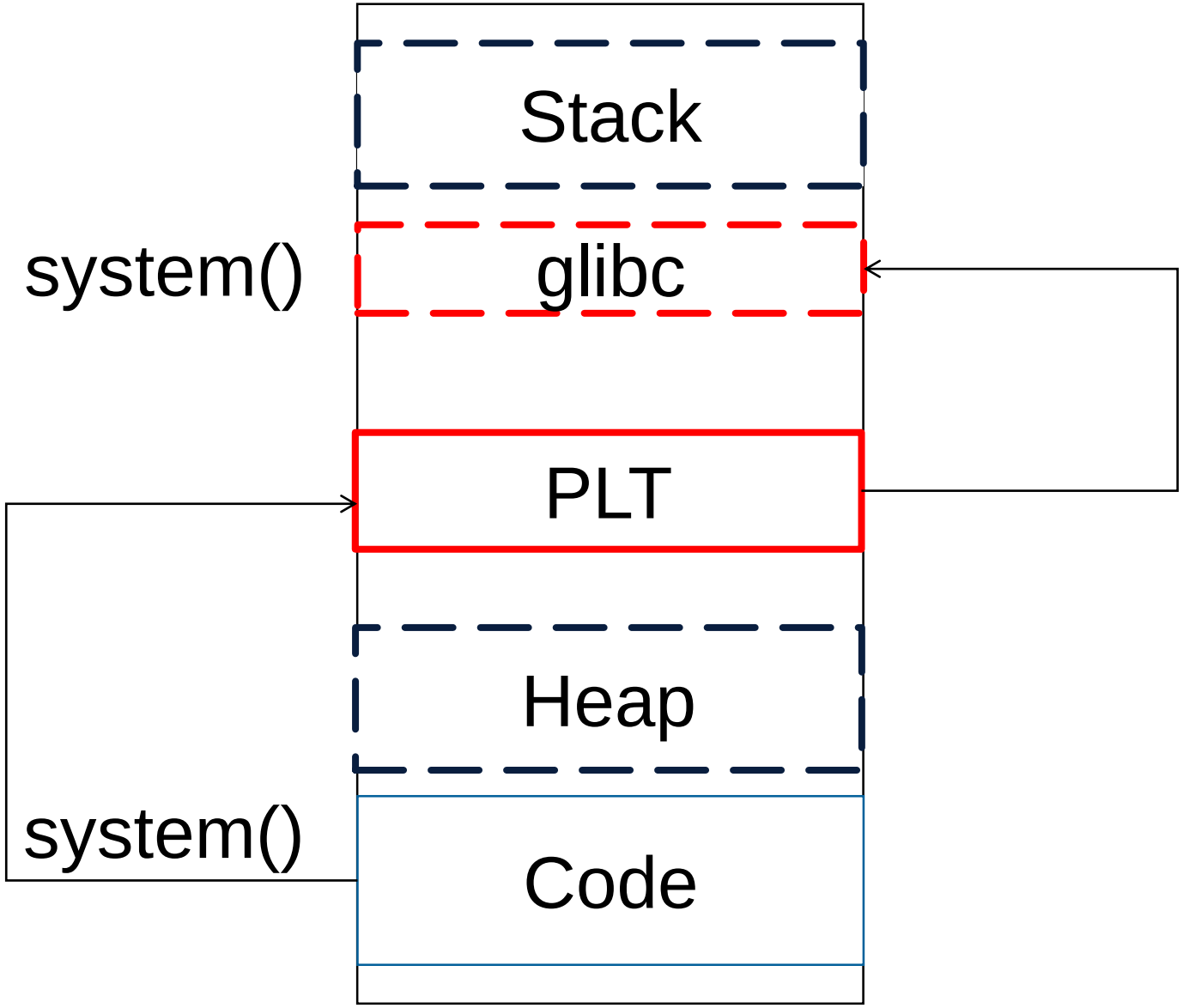
# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt

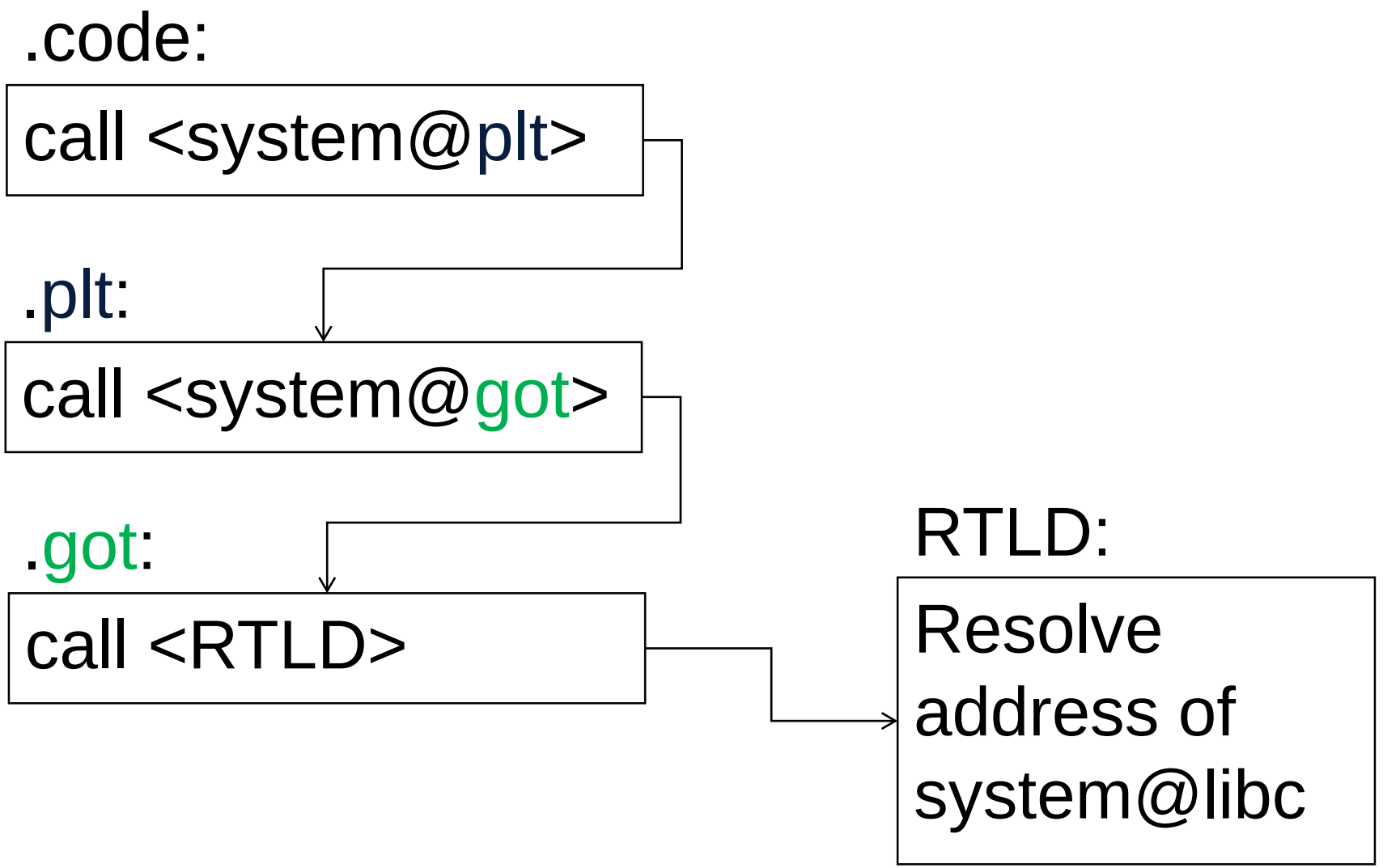


# Defeating DEP – Ret2plt

How does it work?

- “call system” is actually “call system@plt”
- The PLT resolves system@libc at runtime
- The PLT stores system@libc in system@got

# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt

.code:

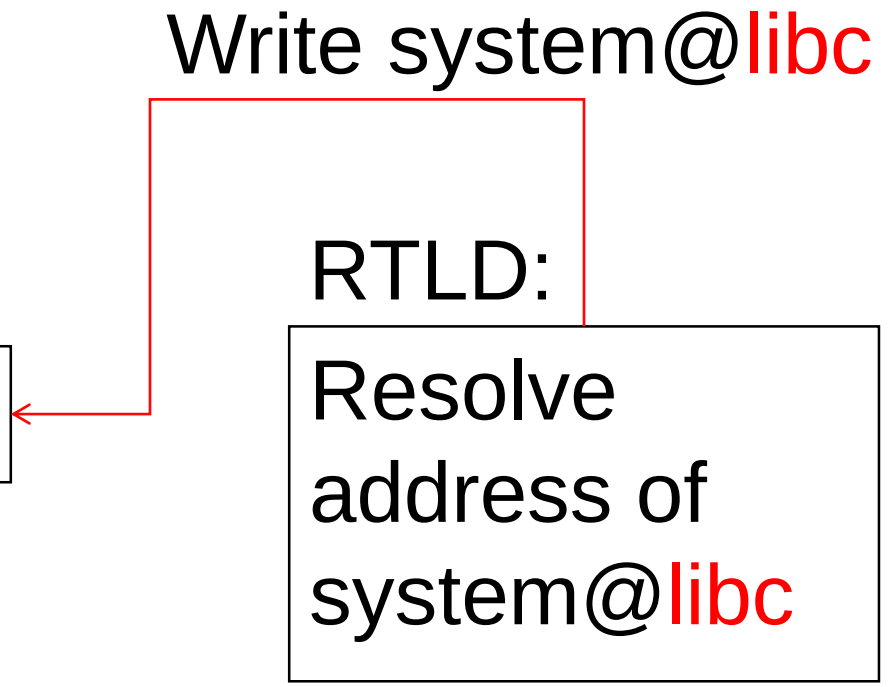
call <system@plt>

.plt:

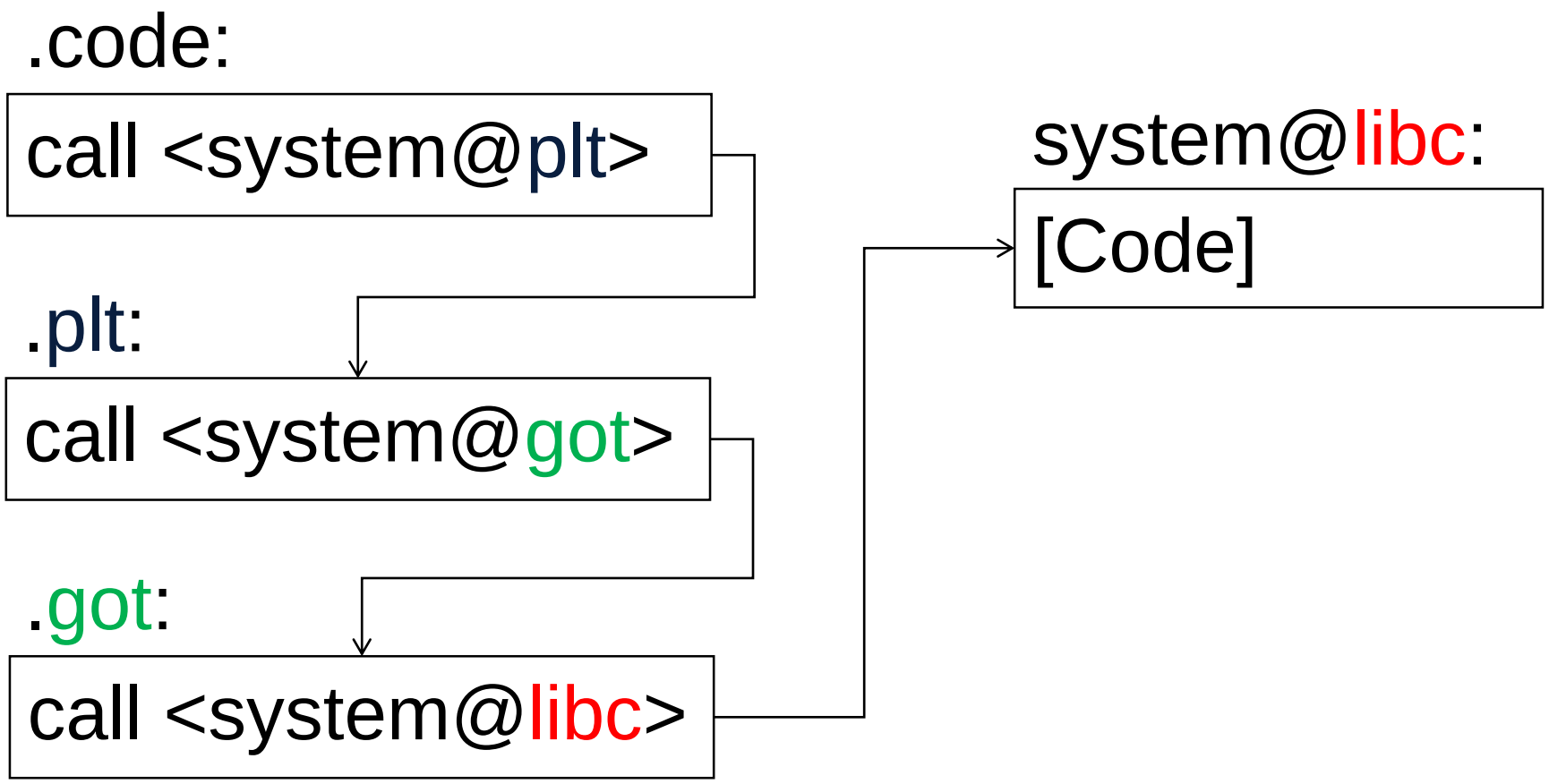
call <system@got>

.got:

call <system@libc>



# Defeating DEP – Ret2plt



# Defeating DEP – Ret2plt

Before executing system():

```
gdb-peda$ print &system
$1 = 0x8048300 <system@plt>
```

After executing system():

```
gdb-peda$ print &system
$2 = 0xb7e67060 <system>    @libc
```

# Defeating DEP – Ret2plt

Before executing system():

```
gdb-peda$ print &system
$1 = 0x8048300 <system@plt>
```

After executing system():

```
gdb-peda$ print &system
$2 = 0xb7e67060 <system>   @libc

Program Headers:
  Type           Offset   VirtAddr    Flg Align
  PHDR           0x000034 0x08048034  R E 0x4
  INTERP         0x000154 0x08048154  R   0x1
  LOAD           0x000000 0x08048000  R E 0x1000
  LOAD           0x000f14 0x08049f14  RW  0x1000

02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
.eh_frame_hdr .eh_frame
```

# Defeating DEP – Ret2plt

Before executing system():

```
gdb-peda$ print &system
$1 = 0x8048300 <system@plt>
```

After executing system():

```
gdb-peda$ print &system
$2 = 0xb7e67060 <system>   @libc
```

```
$ cat /proc/31261/maps
…
b7e27000-b7e28000 rw-p 00000000 00:00 0
b7e28000-b7fcb000 r-xp 00000000 08:02 672446     /lib/i386-linux-gnu/libc-2.15.so
b7fcb000-b7fcd000 r--p 001a3000 08:02 672446     /lib/i386-linux-gnu/libc-2.15.so
…
```

# Defeating DEP – Ret2plt

Conclusion:

- LIBC interface is stored at a static location
- Can jump to system() at known location to execute arbitrary code
- No need for shellcode on stack or heap

# Exploiting: DEP – Ret2plt

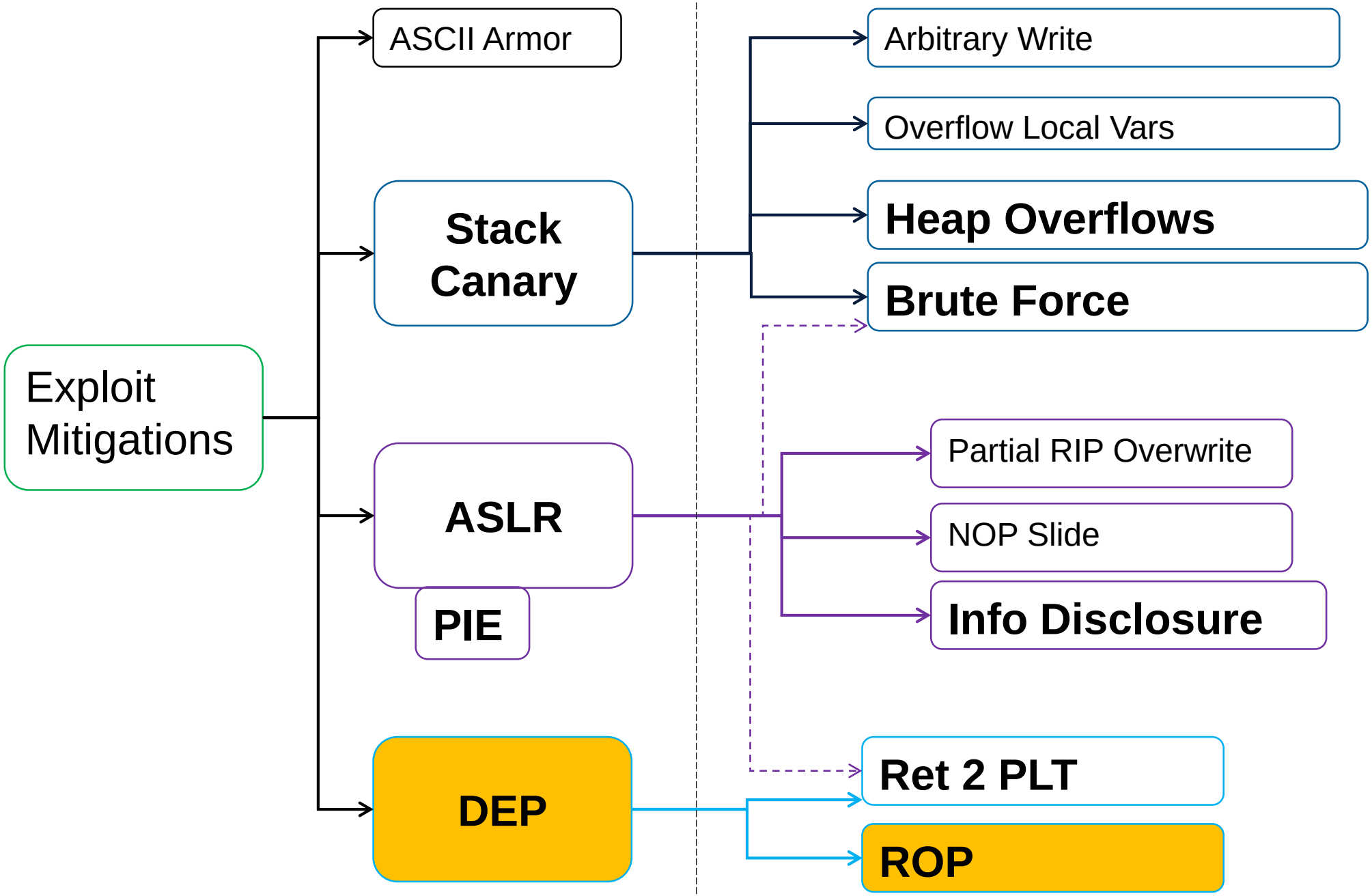
ret2plt

- Defeats DEP

EIP = &system@plt

arg = &meterpreter_bash_shellcode

system(“/bin/bash nc –l –p 31337”)


Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# ROP

ROP

- Extension of “return to libc”
- “Borrowed Code Junks”
- Code from binary, followed by a RET
- Called “gadgets”
- Return Oriented Programming (ROP)

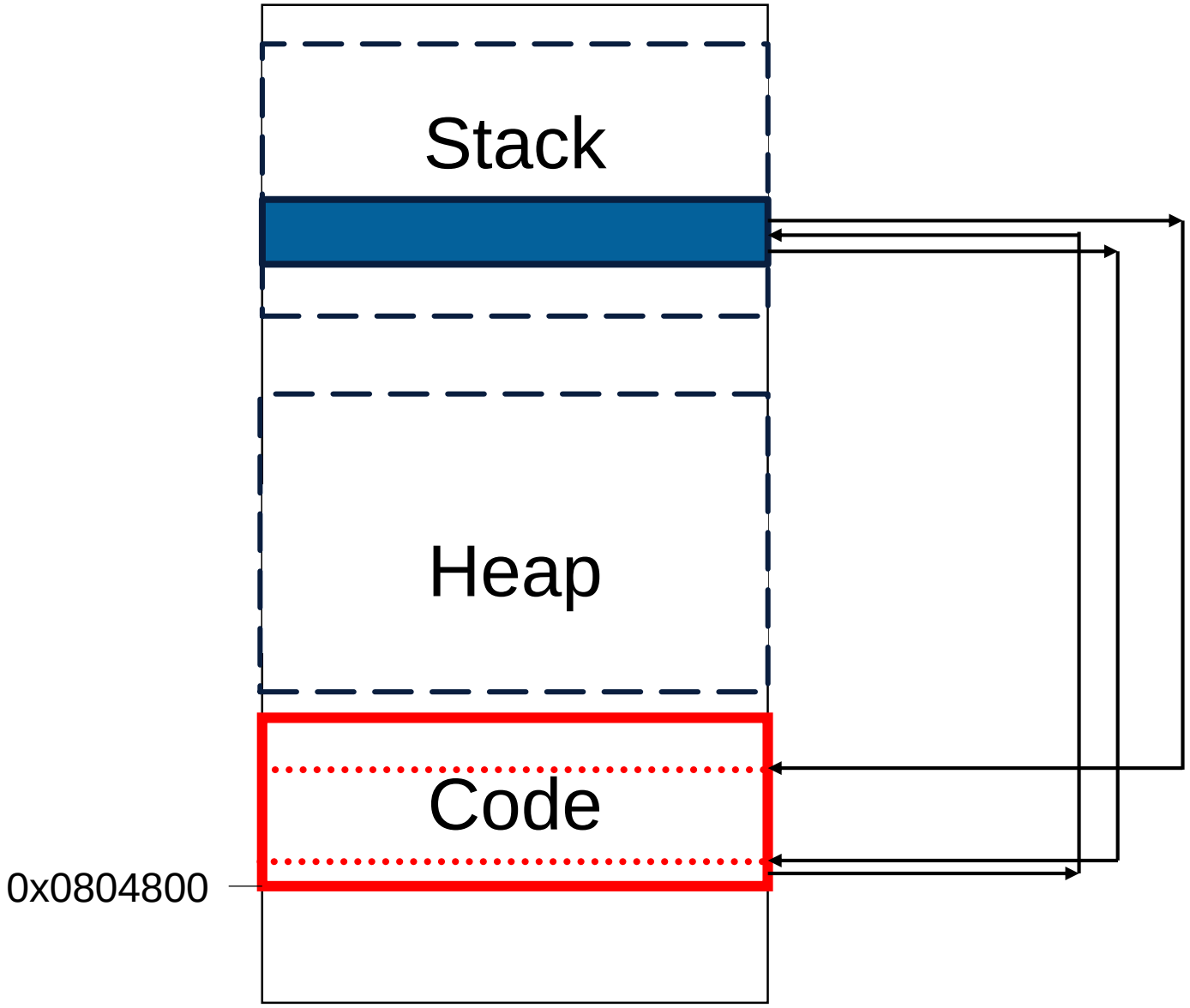
# Defeating DEP - ROP

So, what is ROP?

- Code sequence followed by a “ret”

```
pop r15 ; ret
add byte ptr [rcx], al ; ret
dec ecx ; ret
```

# Defeating DEP - ROP



# Defeating DEP - ROP

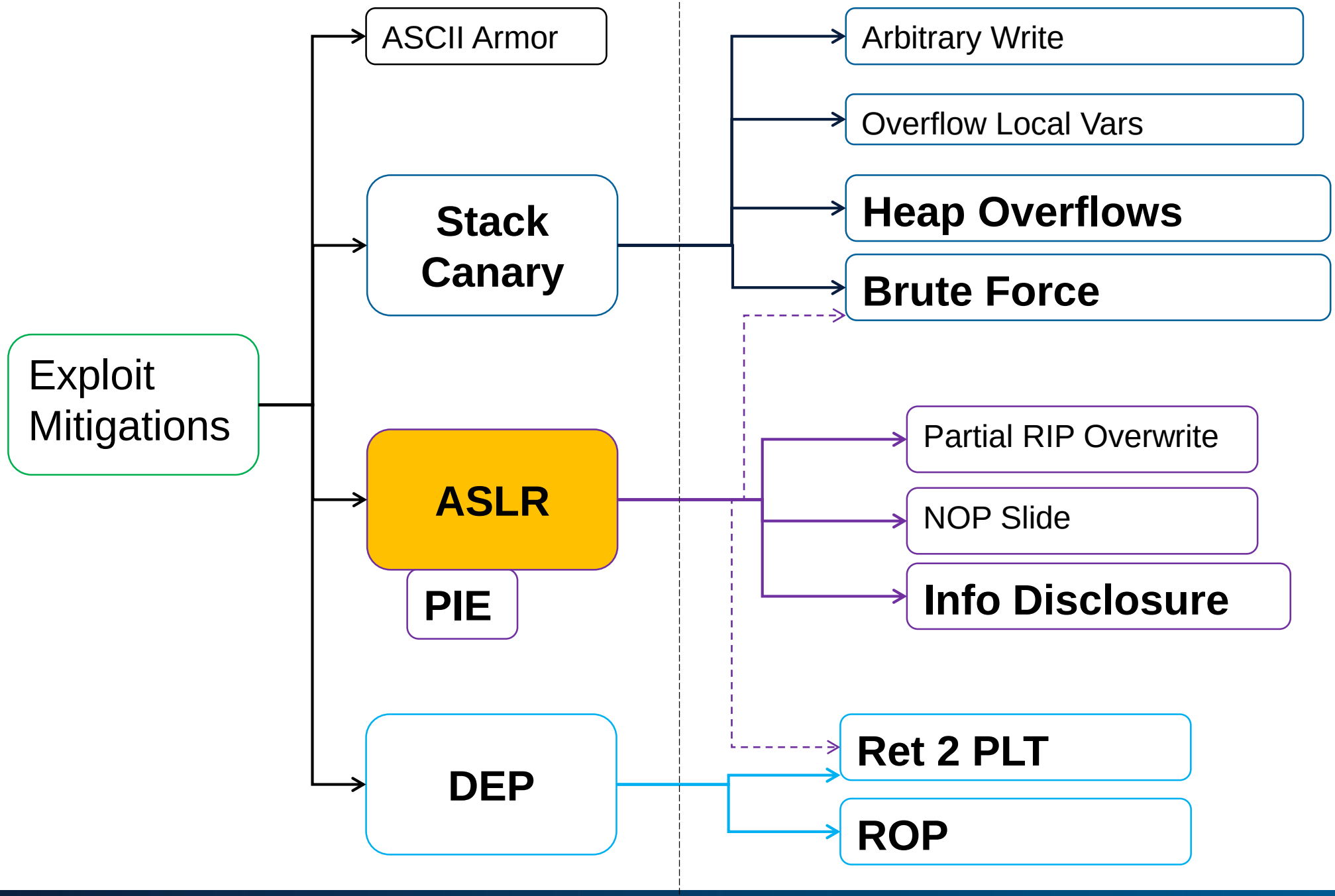
Conclusion:

Code section is not randomized

Just smartly re-use existing code

We’ll have a look at it later

# Defeat Exploit Mitigations: ASLR

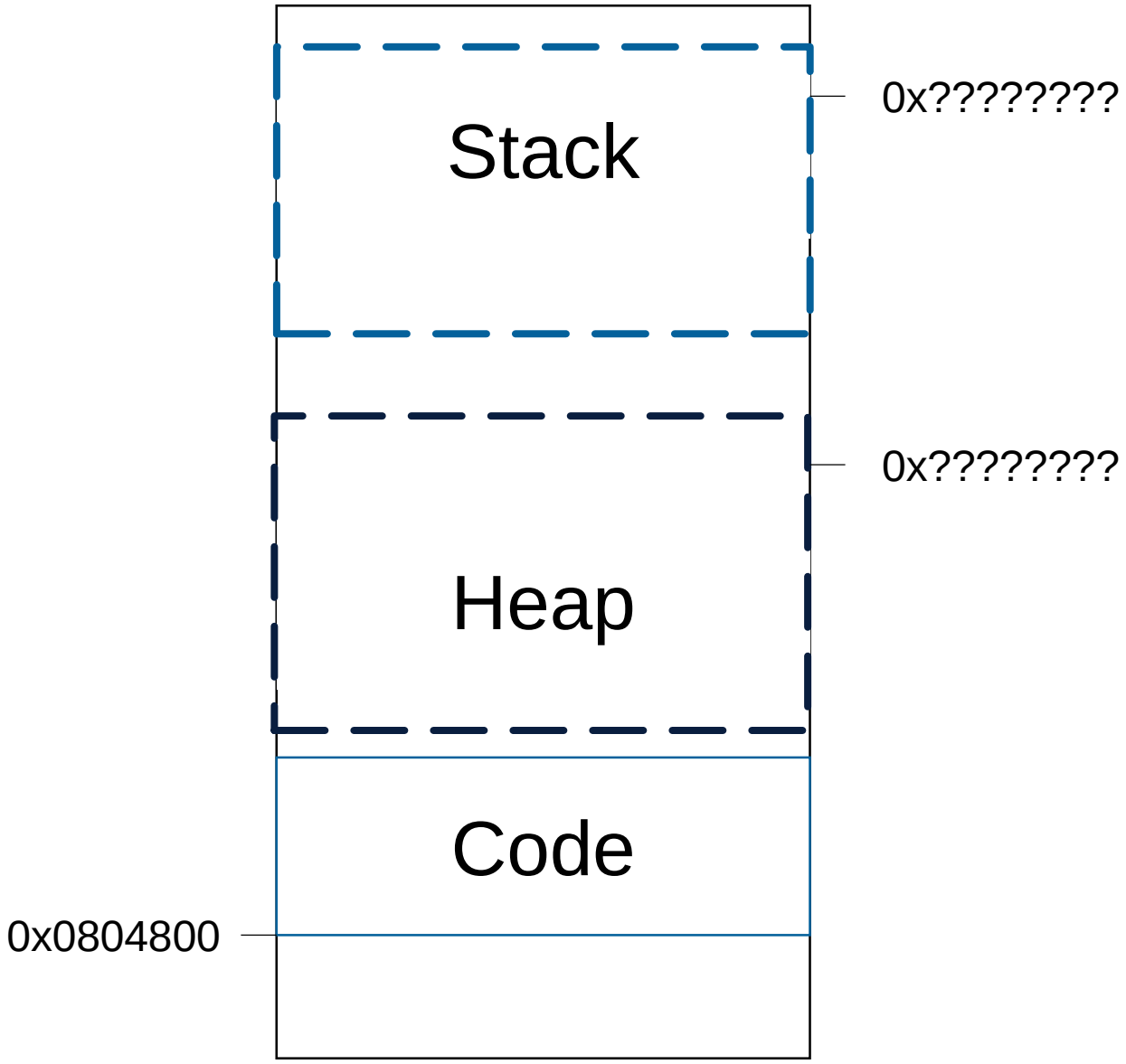

Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

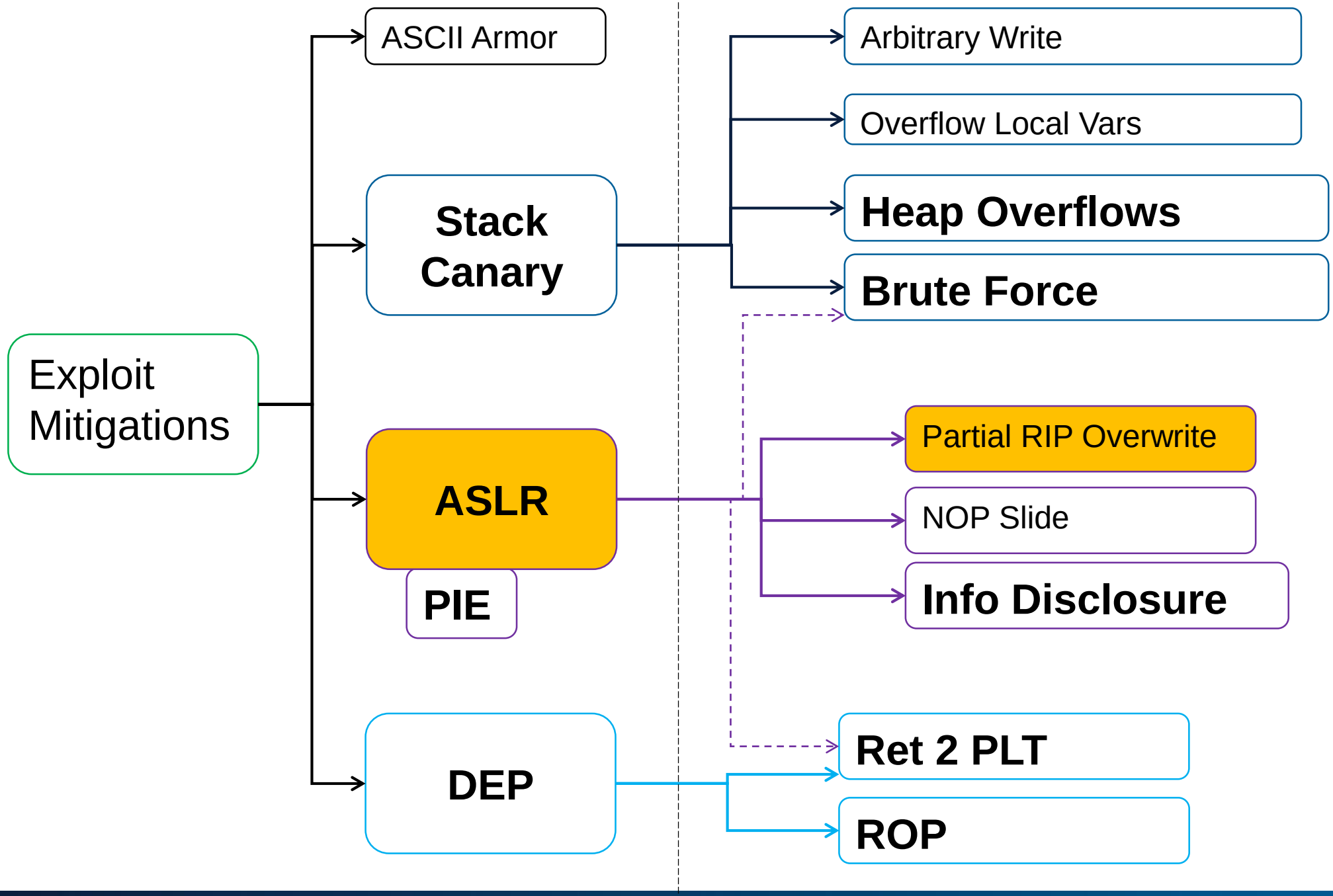
# Defeating ASLR

Recap:

ASLR map’s Stack & Heap at random locations

# Defeating ASLR - Intro

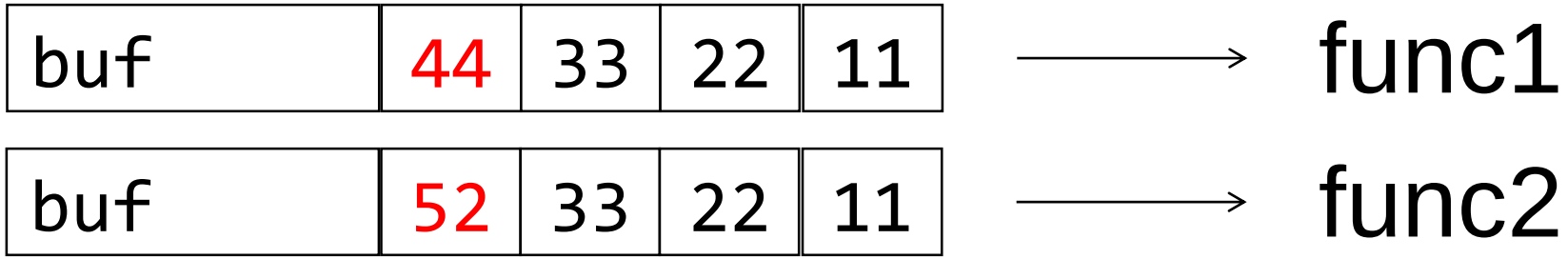



Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

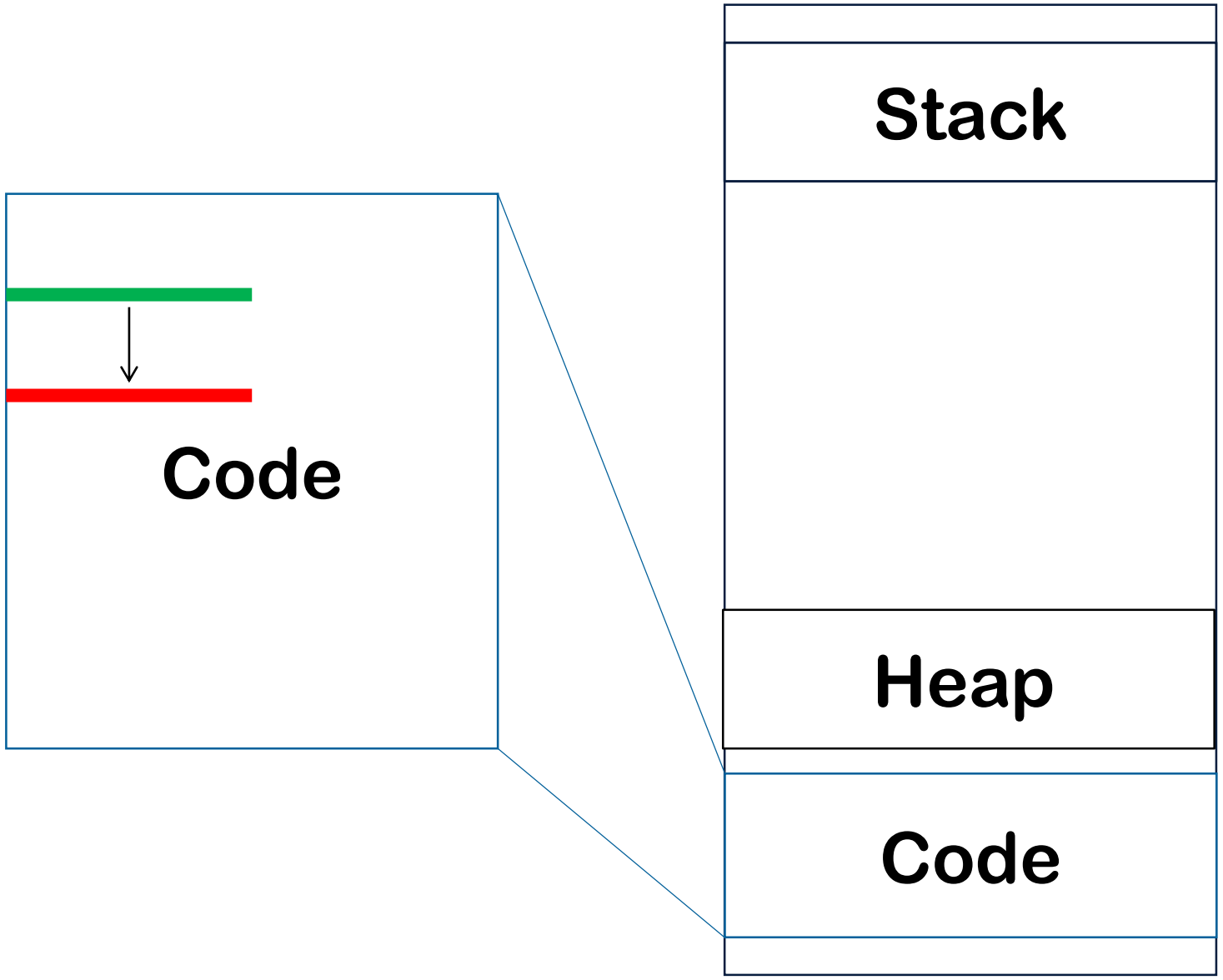
# Defeating ASLR – Partial overwrite

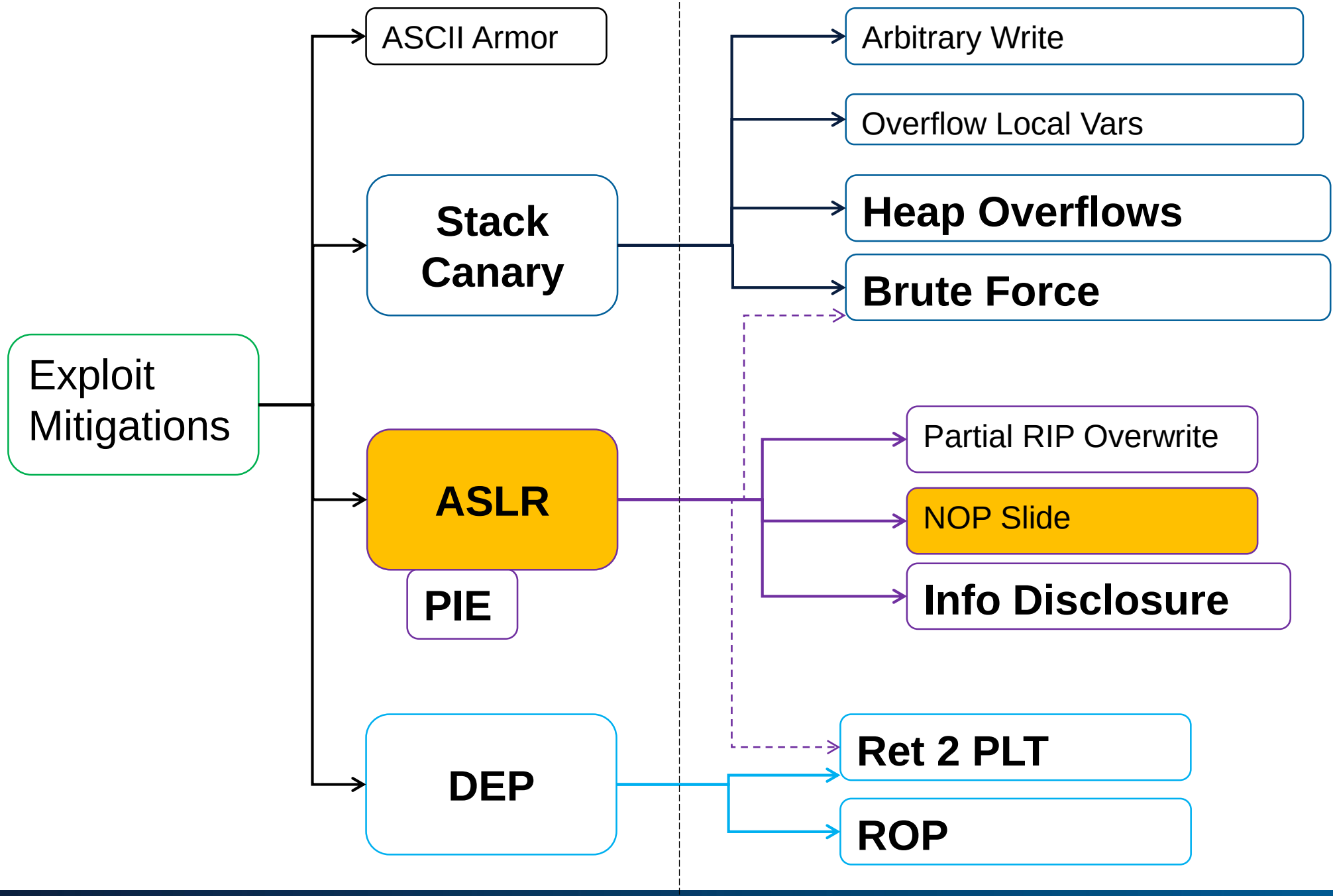
Partial function pointer overwrite

- little endianness: 0x11223344



# Defeating ASLR – Partial overwrite

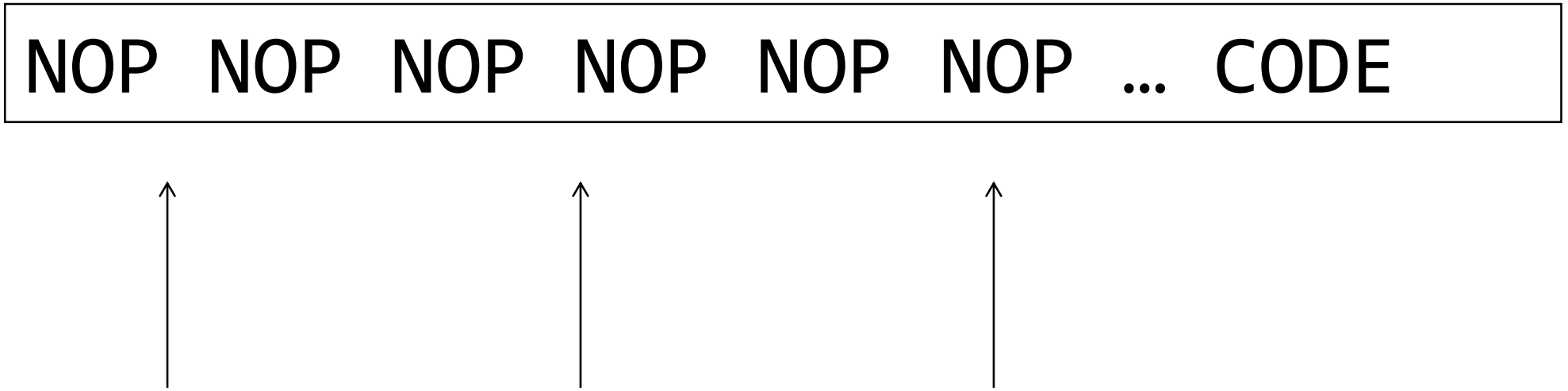



Exploit Mitigations
ASCII Armor
Stack Canary
ASLR
PIE
DEP
Arbitrary Write
Overflow Local Vars
Heap Overflows
Brute Force
Partial RIP Overwrite
NOP Slide
Info Disclosure
Ret 2 PLT
ROP

# Defeating ASLR – NOP sleds

NOP sleds

- As often used with JavaScript
- Heap spray a few megabytes…



# Defeating ASLR – NOP sleds

NOP sleds

- As often used with JavaScript
- Heap spray a few megabytes…

| NOP NOP NOP NOP NOP NOP … CODE |
| --- |
| NOP NOP NOP NOP NOP NOP … CODE |
| NOP NOP NOP NOP NOP NOP … CODE |
| NOP NOP NOP NOP NOP NOP … CODE |

# Heap Spray with NOP Sleds

Old, old **string** based NOP sled:

```
var nop = unescape("%u9090%u9090");

// Create a 1MB string of NOP instructions followed by shellcode:
//
// malloc header   string length   NOP slide   shellcode   NULL terminator
// 32 bytes        4 bytes         x bytes     y bytes     2 bytes

while (nop.length <= 0x100000/2) nop += nop;

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2);

var x = new Array();

// Fill 200MB of memory with copies of the NOP slide and shellcode
for (var i = 0; i < 200; i++) {
    x[i] = nop + shellcode;
}
```

https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf

# Heap Spray with ASM.JS

ASM.JS:

```
VAL = (VAL + 0xA8909090)|0;
VAL = (VAL + 0xA8909090)|0;
```

Firefox ASM.JS JIT generates:

```
00: 05909090A8 ADD EAX, 0xA8909090
05: 05909090A8 ADD EAX, 0xA8909090
```

Jump offset 1:

```
01: 90 NOP
02: 90 NOP
03: 90 NOP
04: A805 TEST AL, 05
06: 90 NOP
07: 90 NOP
08: 90 NOP
```

# Recap: Anti ASLR

Anti-ASLR:

- Find static locations (like PLT)
- Mis-use existing pointers
- Spray & Pray

# Conclusion

# Defeat Exploit Mitigations - Conclusion

Three default Exploit Mitigations:

- Stack Canary (crash on overflow)
- ASLR (make memory locations unpredictable)
- DEP (make writeable memory non-executable)

There are several techniques which circumvent these Exploit Mitigations

# Advanced Exploitation Techniques

## Stack-Protector?

- Arbitrary write
- Byte-wise stack-protector brute-force
- Heap Overflow

## No-Exec Stack?

- Return to LIBC
- Return to PLT (my favorite ;-) )
- ROP

## ASLR/PIE?

- Brute Force
  - 12 bit entropy for 32 bit
  - byte-wise brute force
- ROP
- Information Disclosure
- Pointer re-use
- Spray NOP sled

# Advanced Techniques

RET 2 PLT:

- jump to static address which executes system(), with bash-shell shellcode
- Circumvent DEP
- Fix: PIE

ROP:

- Return Oriented Programming
- Take gadgets from binary
- Gadget are little code sequences, followed with a RET
- Fix: PIE
- Super fix: CFI + Shadowstack

# Advanced Exploits

## Information Disclosure

- The death of anti-exploiting techniques
- Get content past a buffer -> get SIP (Saved Instruction Pointer) or stack pointer
- Relocation happens en-block, so just calculate base address and offset for ret2plt or ROP

## Partial Overwrite

- Because of Little-Endianness, can overwrite LSB of function pointers to point to other stuff (not affected by ASLR because in same segment)

## Heap attacks

- Use after free
- Double Free
- And lots more