Return-Oriented Programming

Nathan Chong
22 March 2016

This talk

- Study a stack-smash exploit
- Defences (particularly XN)
- Attacks
  - Return-to-Libc
  - Return-Oriented Programming

```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

Source: http://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/

```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

Read 400 bytes from STDIN (fd 0) into buf

Bug found: Job done! End-of-Talk :-)

```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

Read 400 bytes from STDIN (fd 0) into buf

Buffer-overflow bug allows stack-smash exploit

i.e., subvert the program's control flow to a payload of the attacker's choosing

```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

```
gcc -masm=intel -S \
    -fno-stack-protector
```

revisit this later

```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

```
gcc -masm=intel -S \
    -fno-stack-protector
```

```asm
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

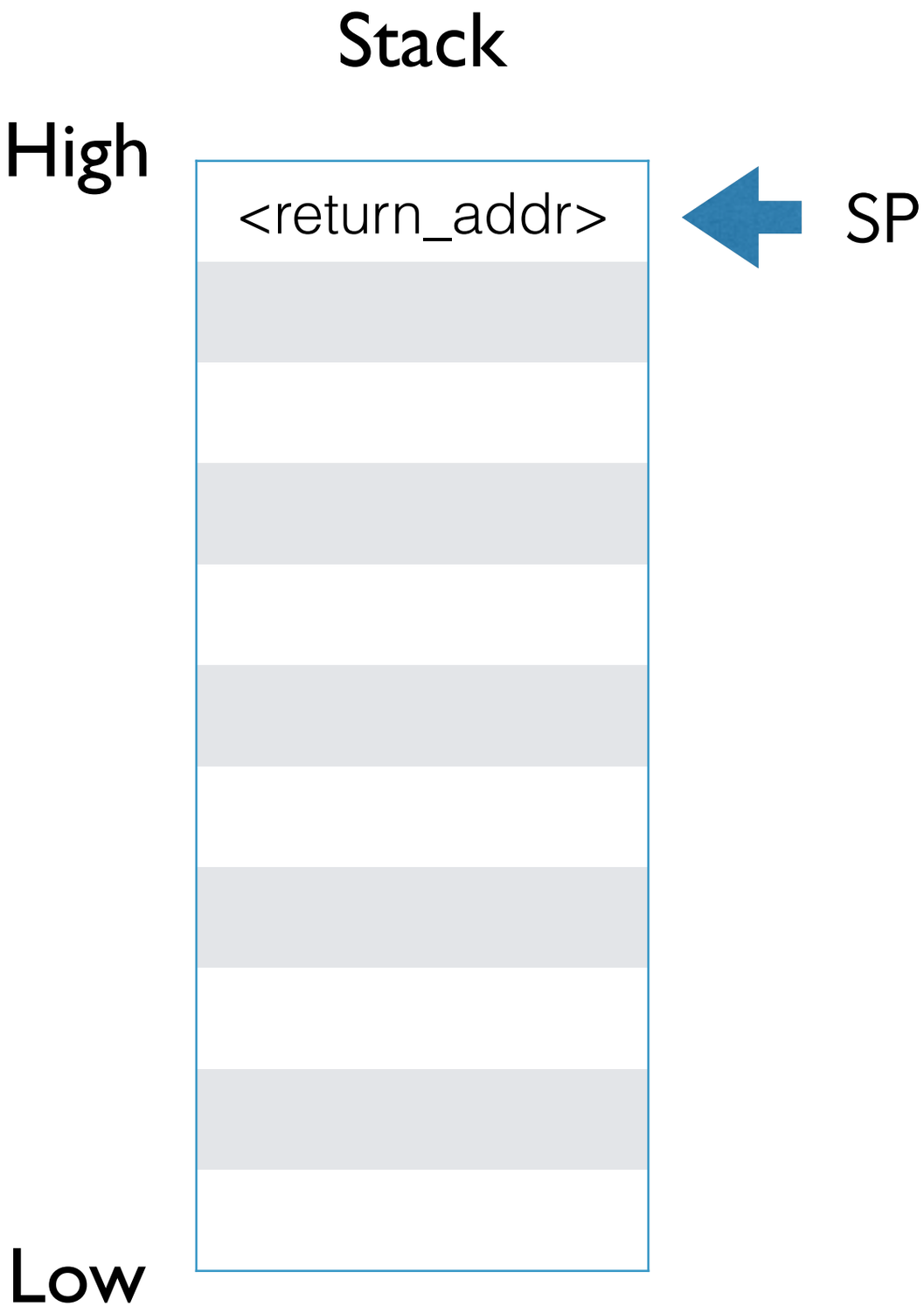
```c
int vuln() {
  char buf[80];
  int r;
  r = read(0, buf, 400);
  // ...
  return 0;
}
```

```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

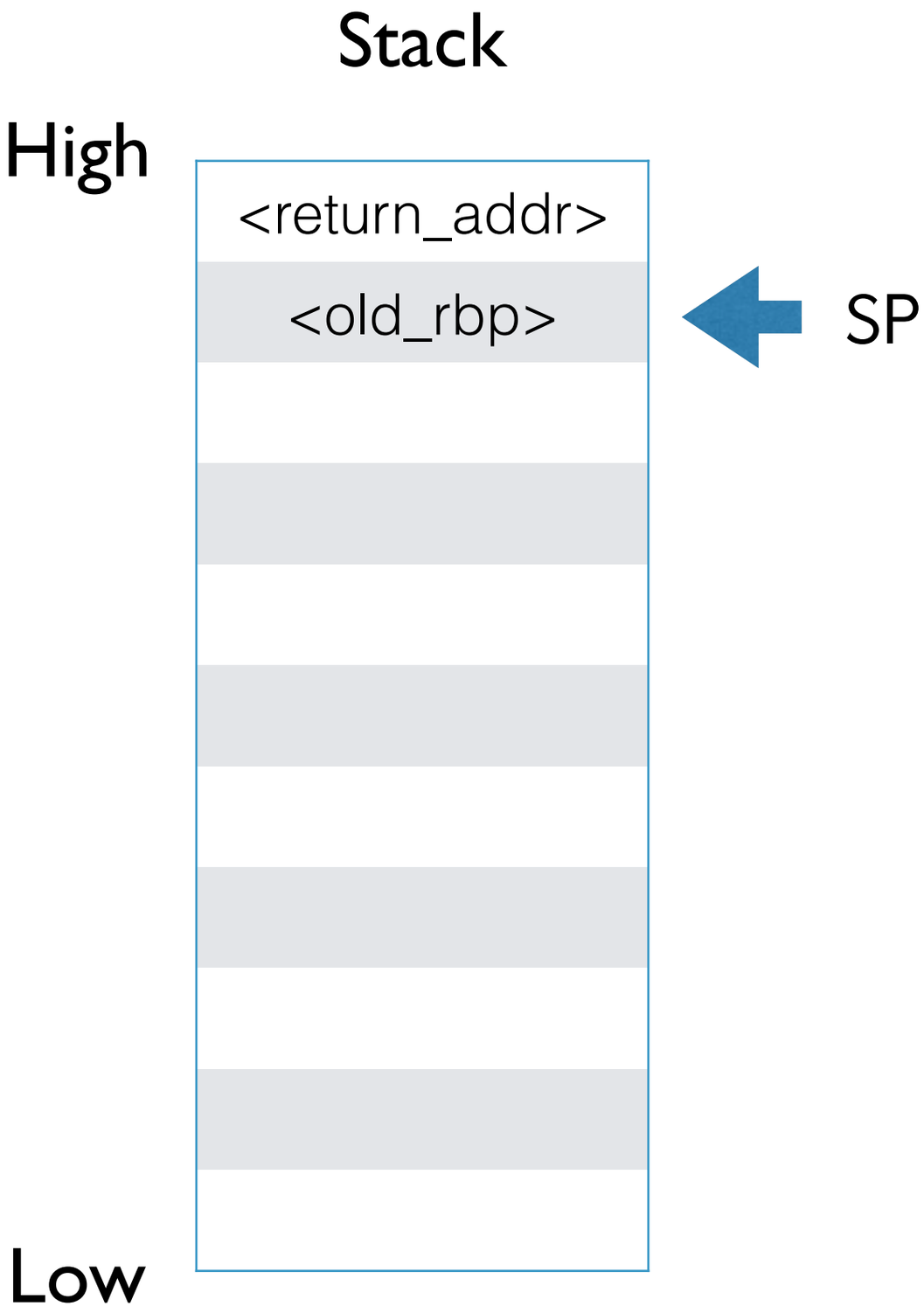
function entry

argument handling
and read() call

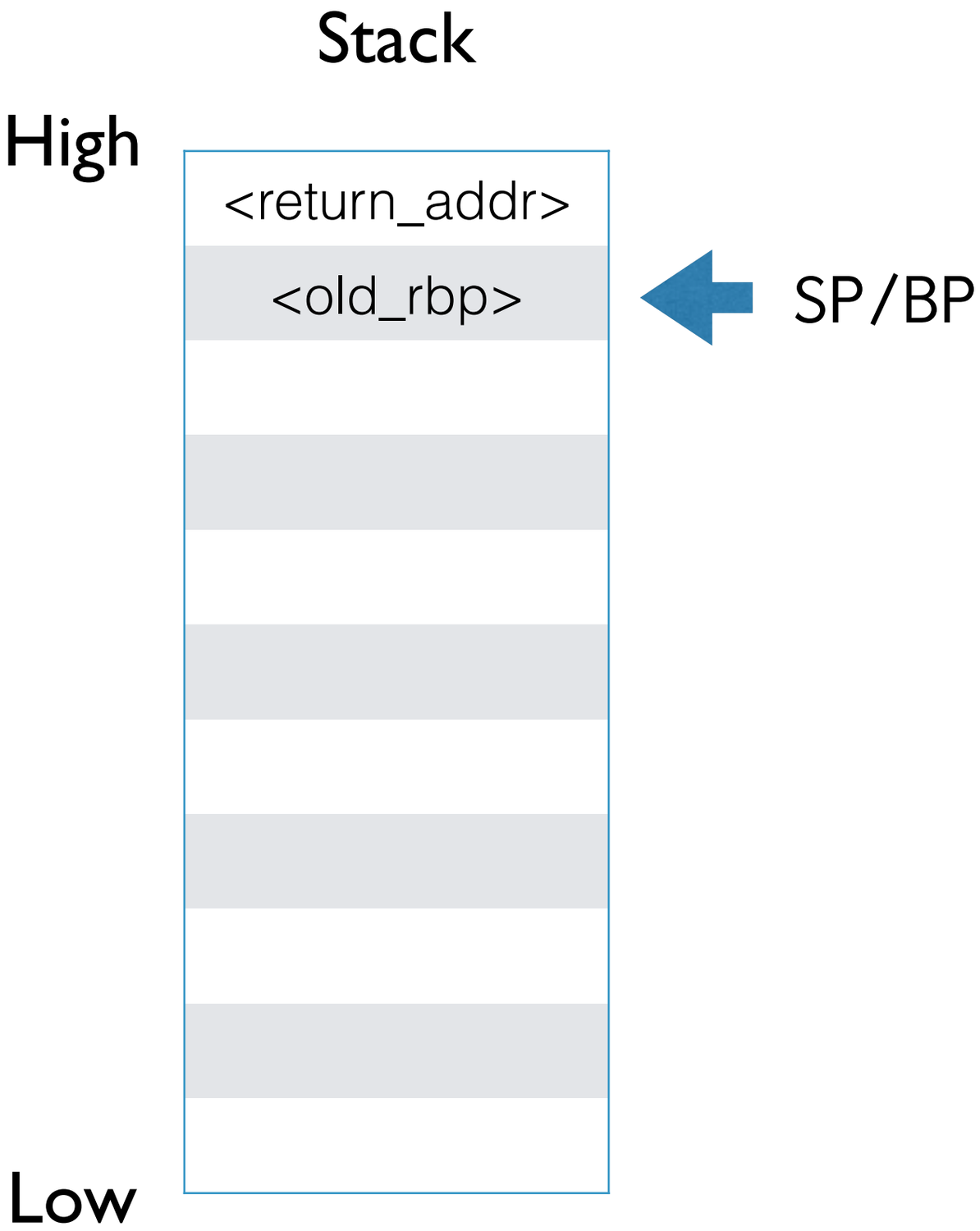
function return



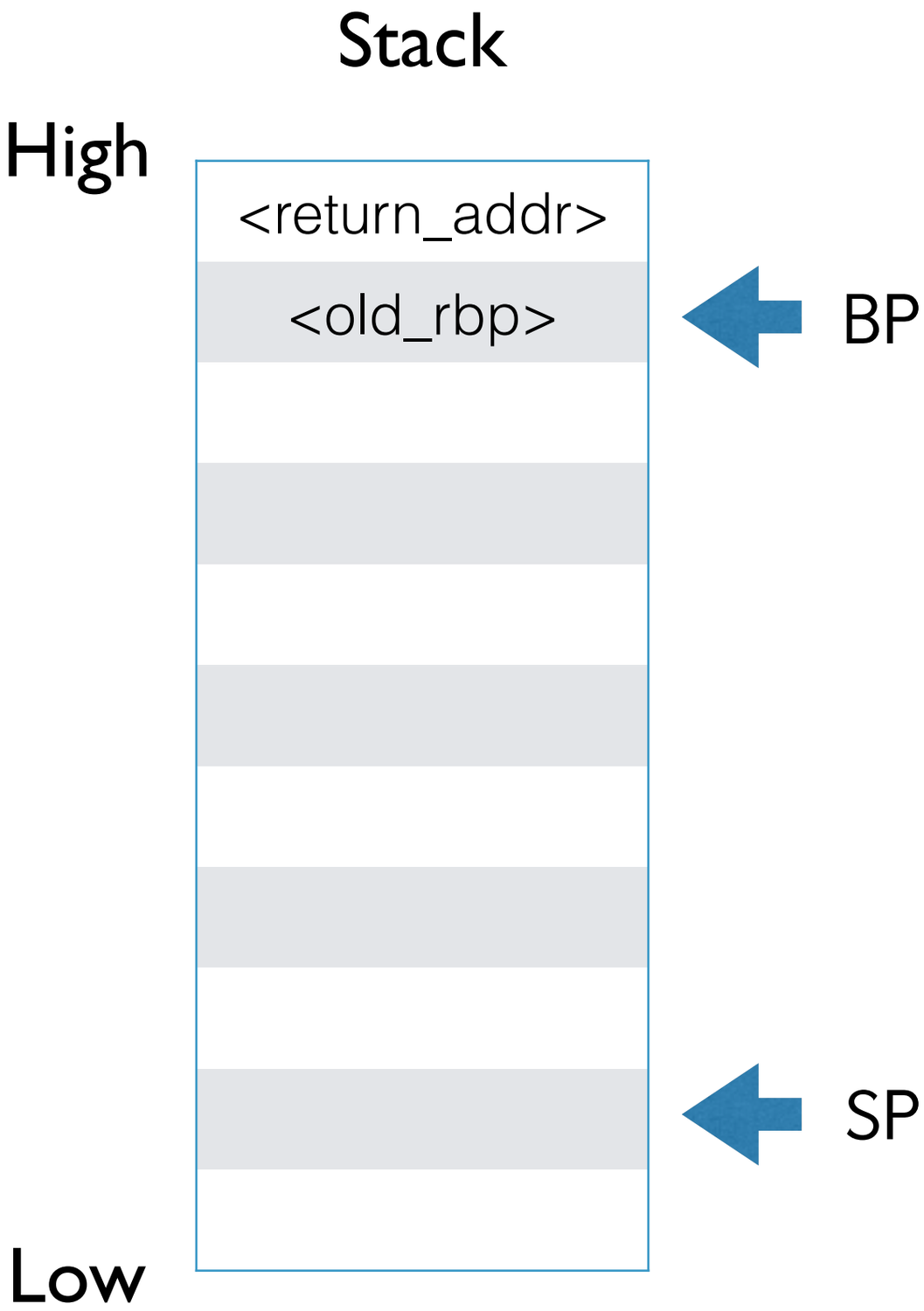

```
vuln:
IP  push rbp
    mov rbp, rsp
    sub rsp, 96
    lea rax, [rbp-96]
    mov edx, 400
    mov rsi, rax
    mov edi, 0
    call read

    // ...

    leave
    ret
```



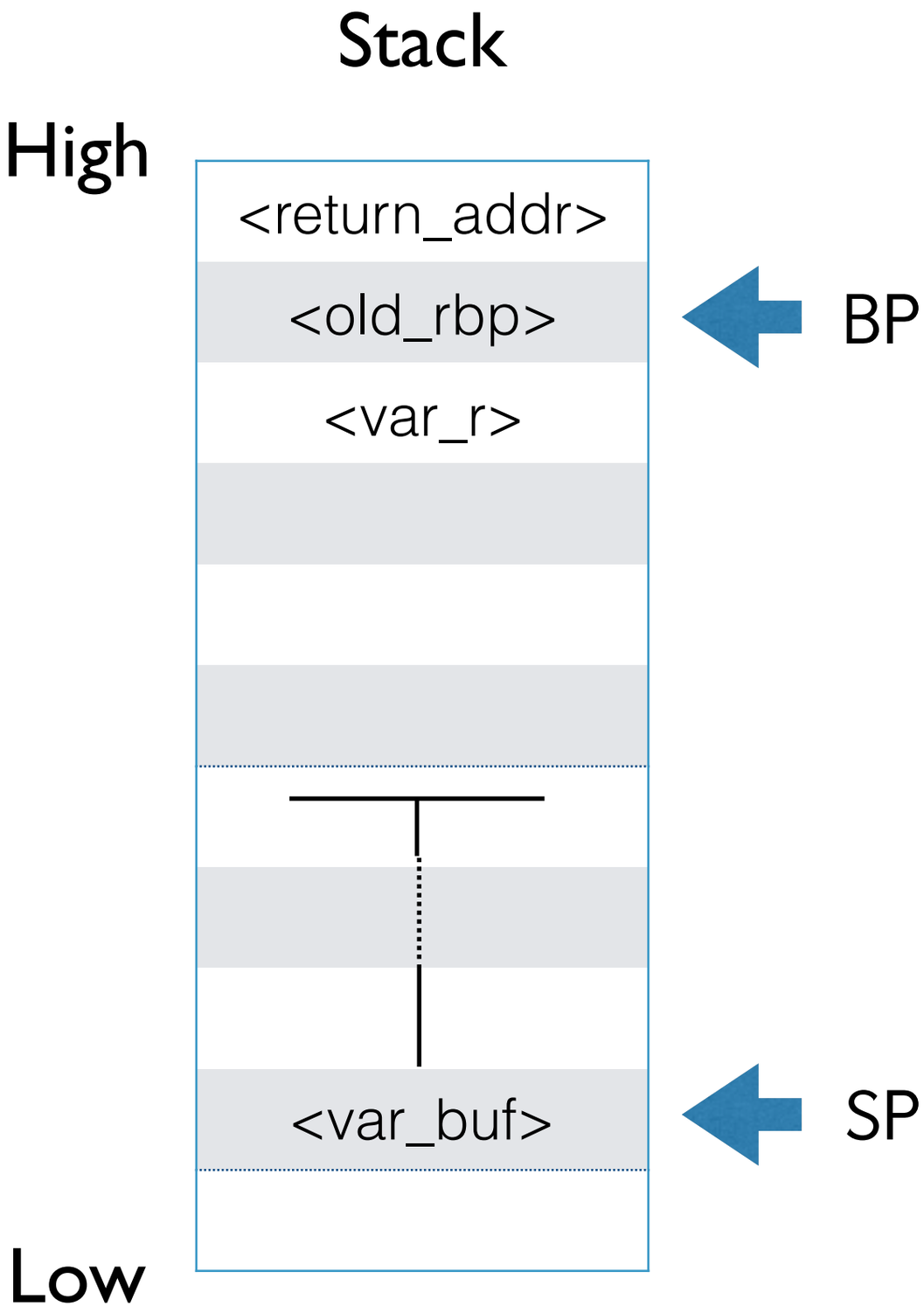

```
vuln:
  push rbp
IP mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```



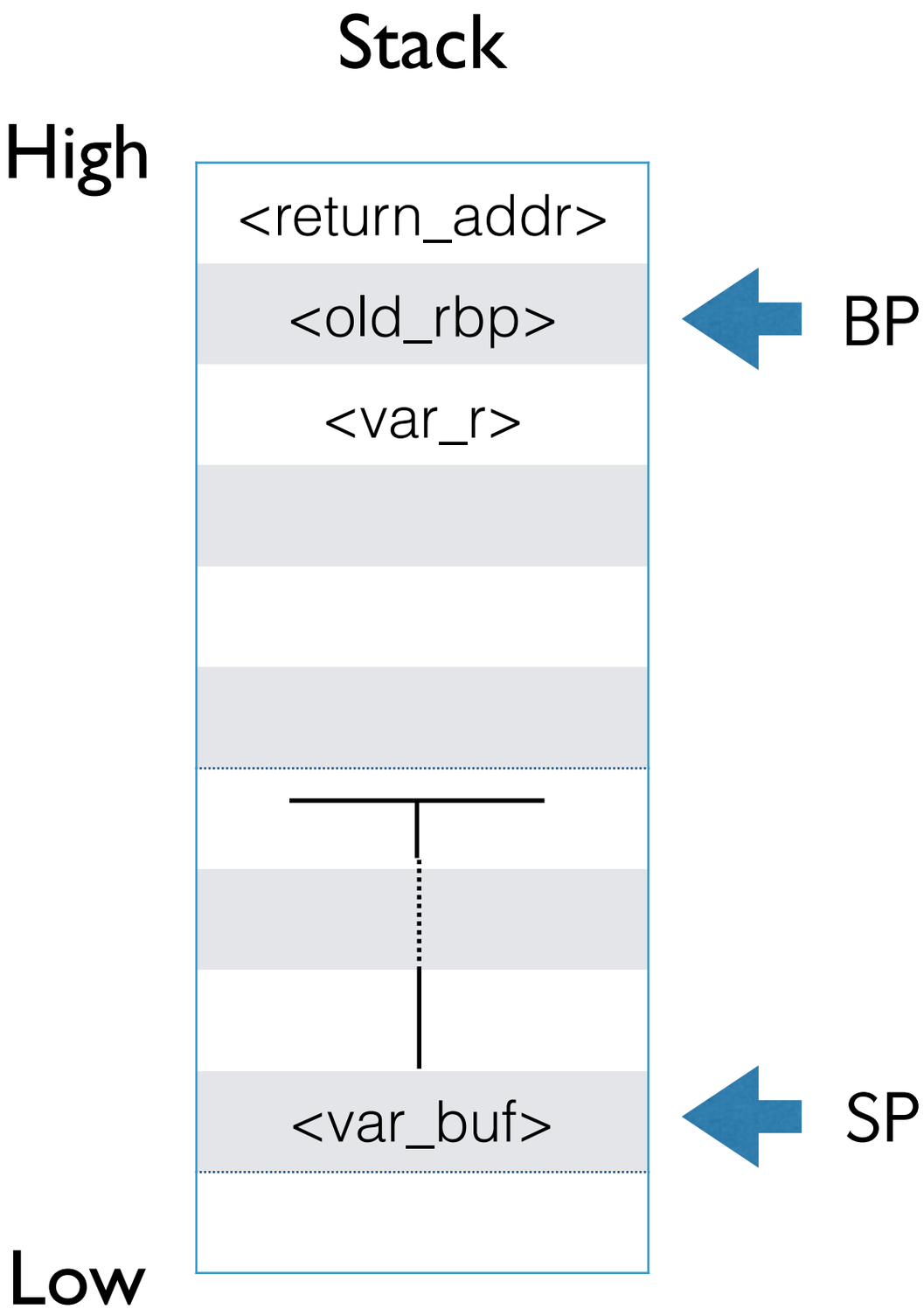

```
vuln:
  push rbp
  mov rbp, rsp
IP sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
IP lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

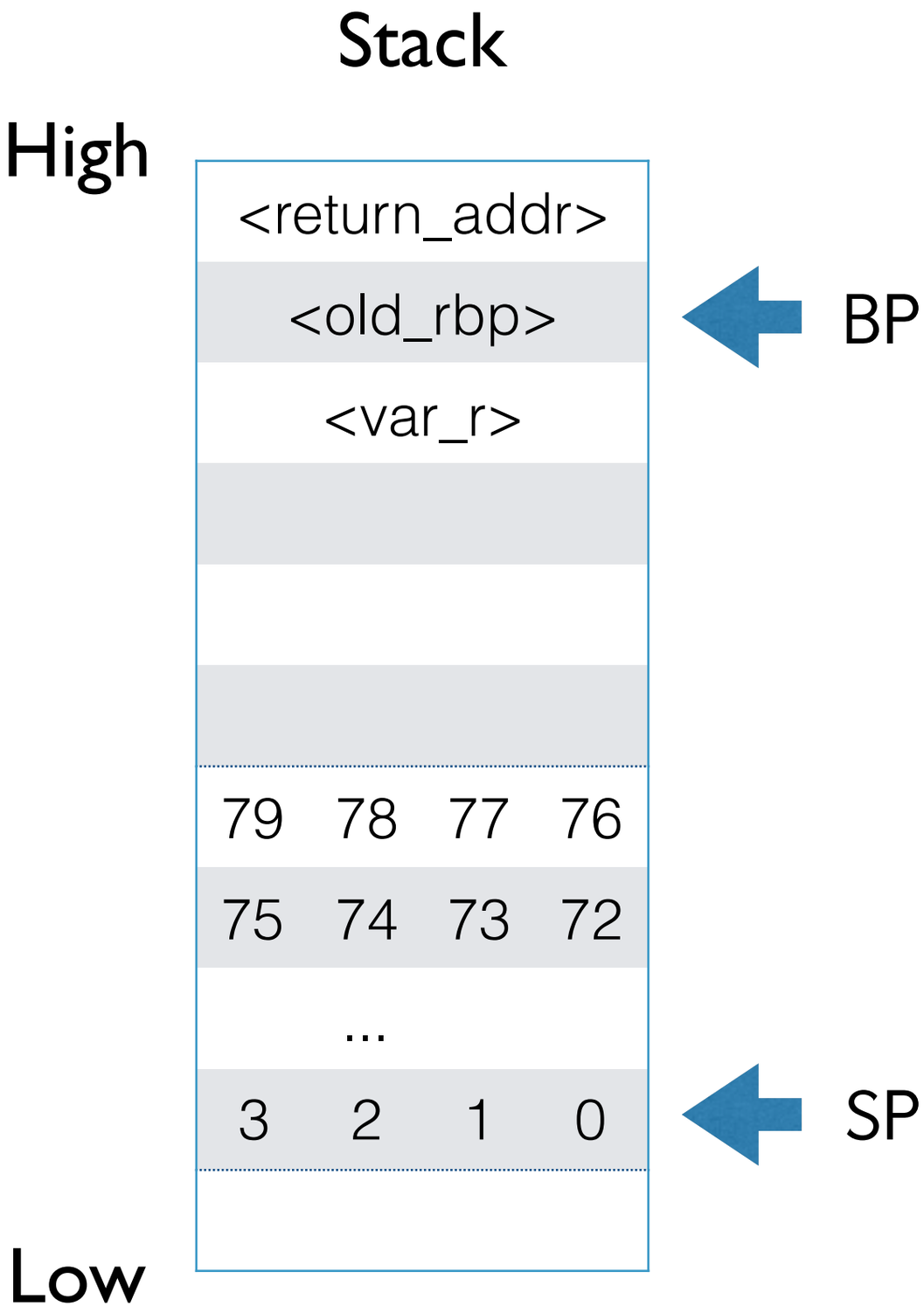



```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
IP lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

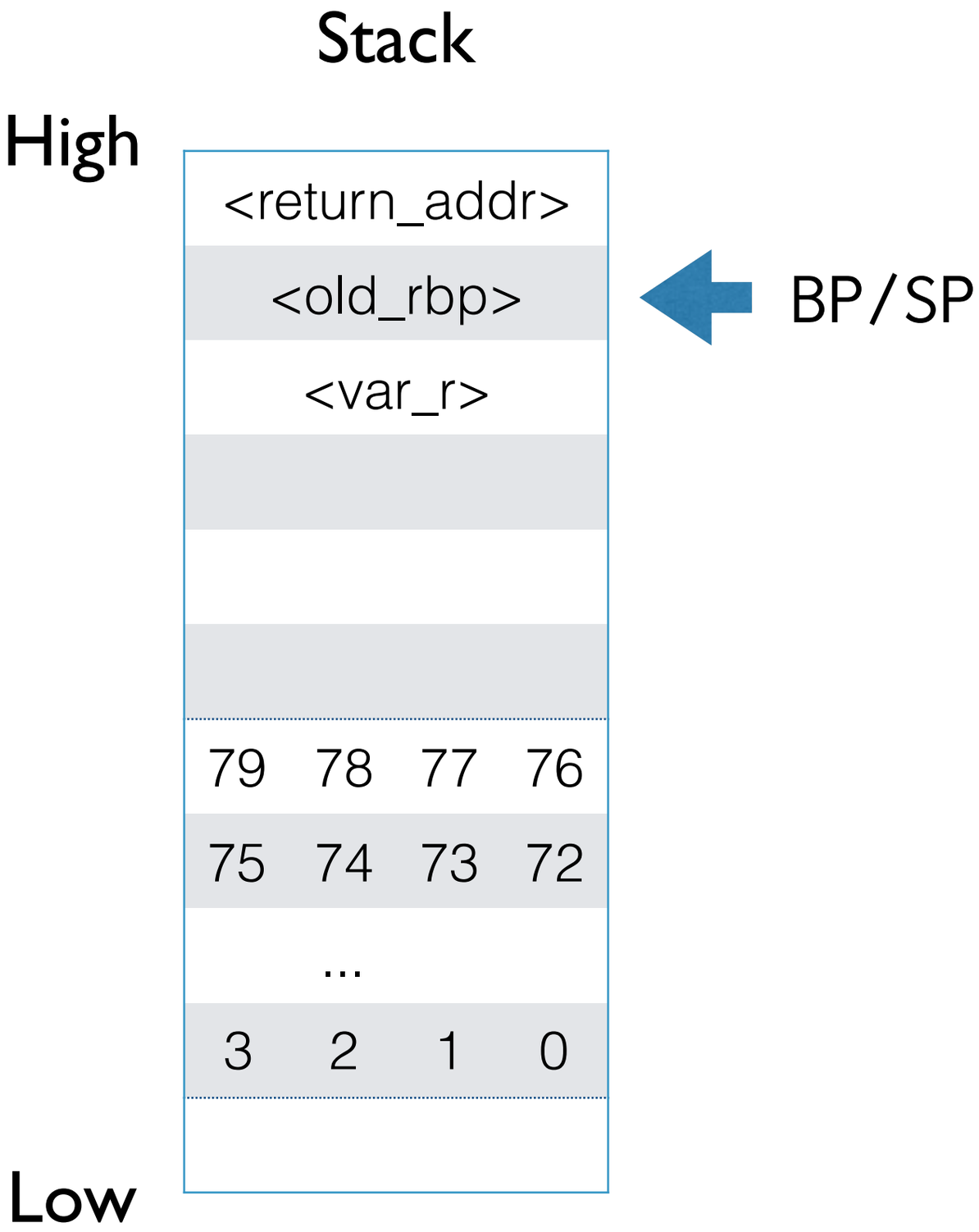



```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
IP call read

  // ...

  leave
  ret
```

edi	rsi	edx
0	&buf	400

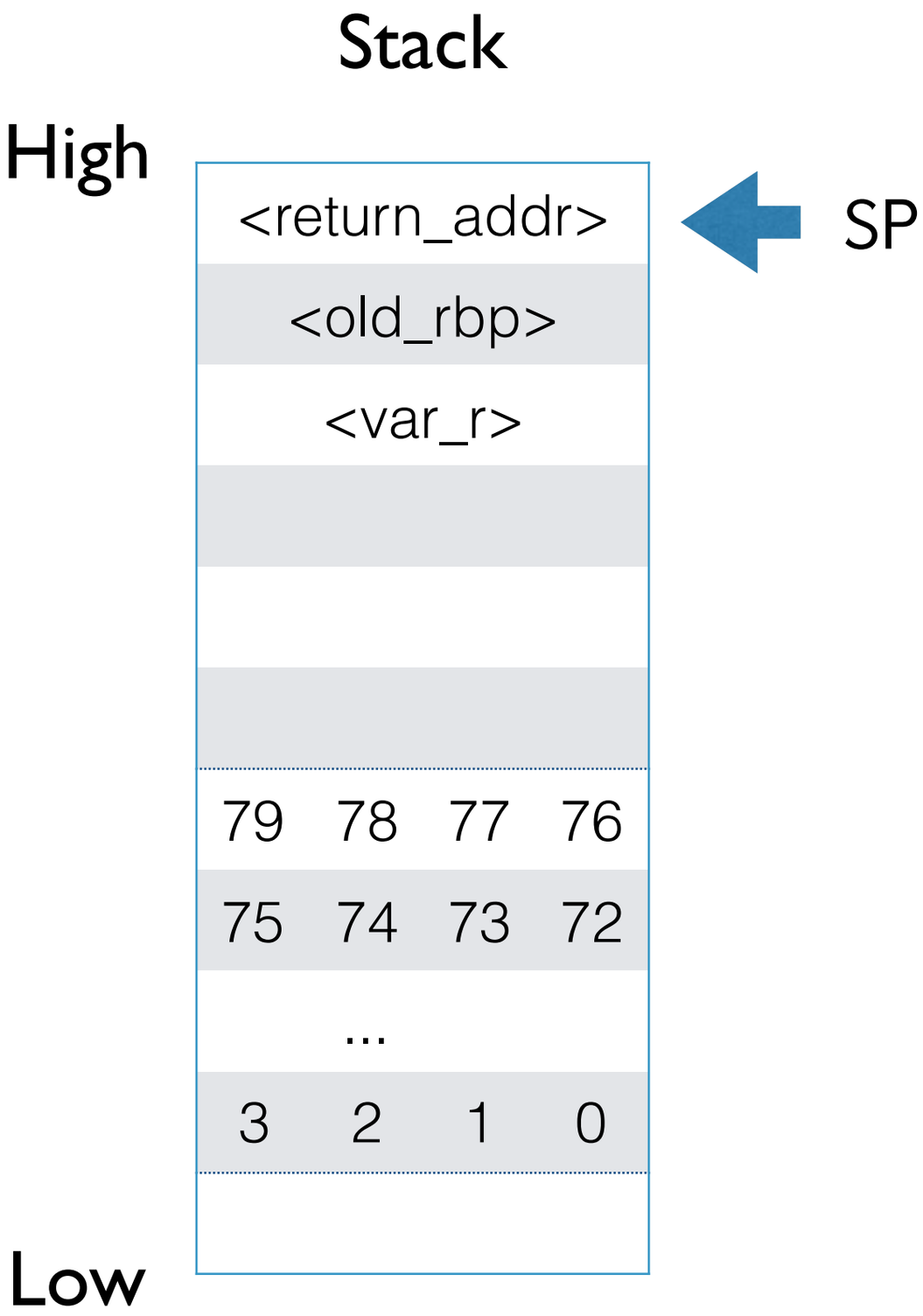



```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read
IP
  // ...
  leave
  ret
```



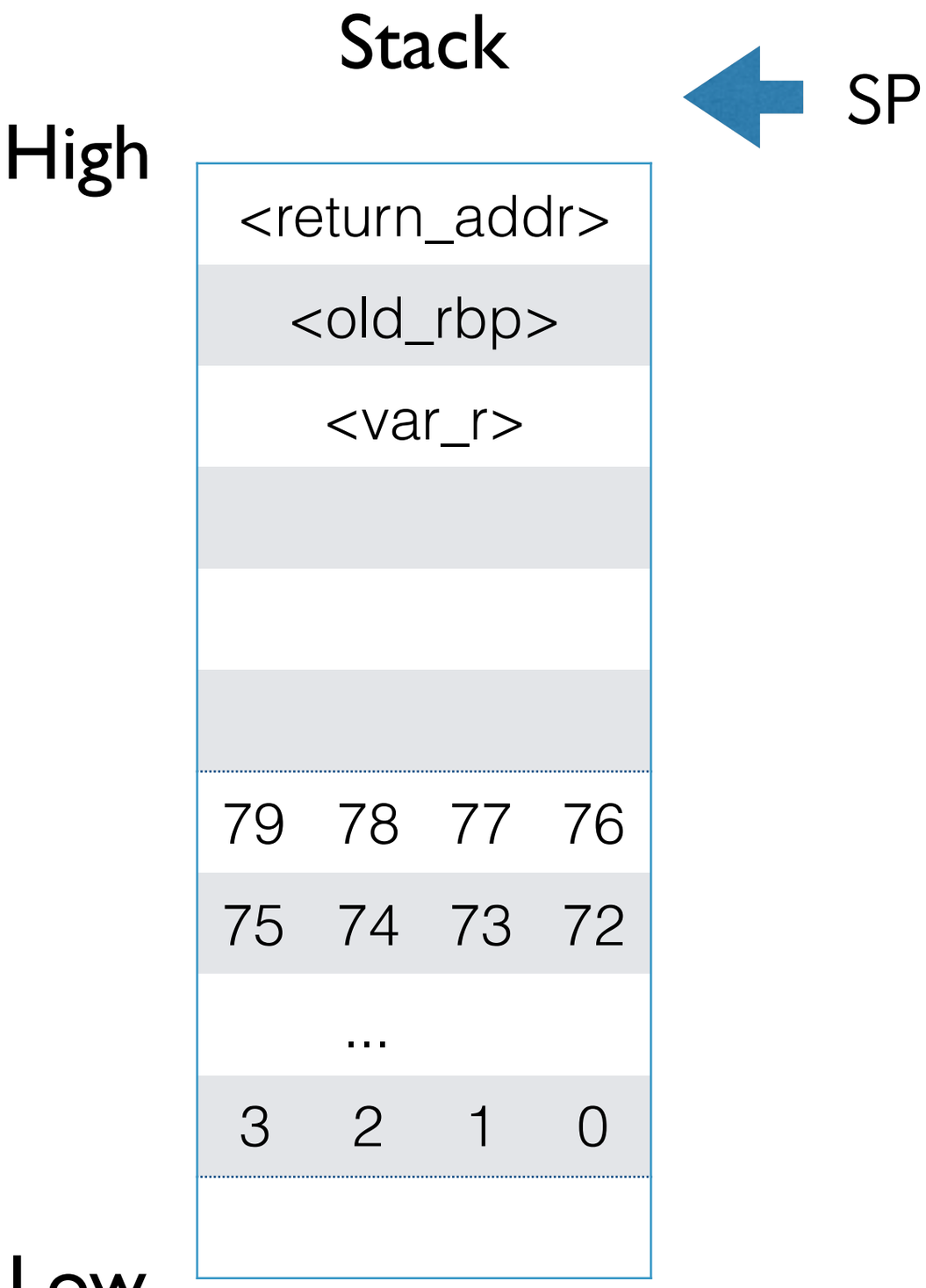

```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

IP




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

IP




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

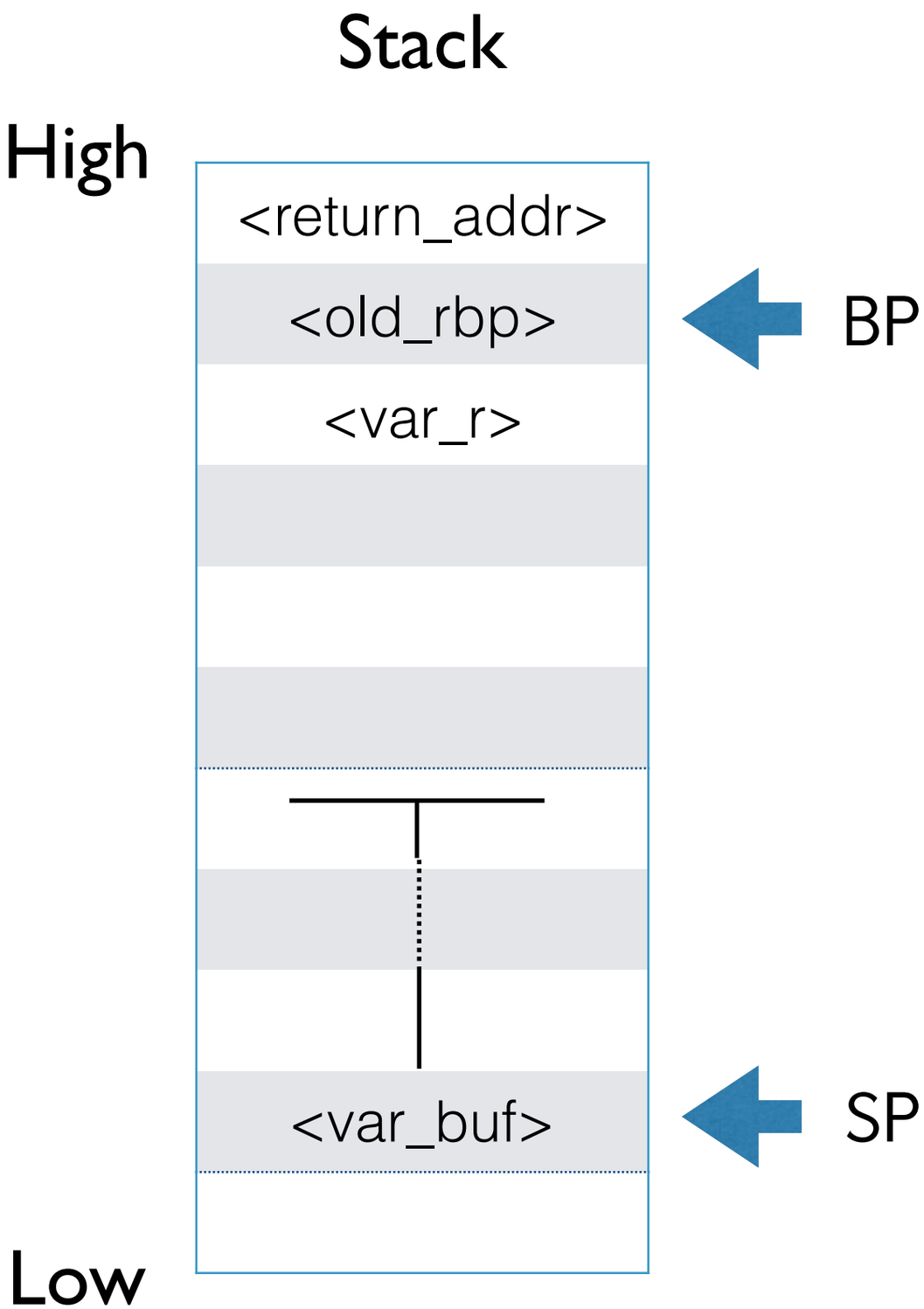
After function return

`IP` set to <return_addr>

Smashing the stack

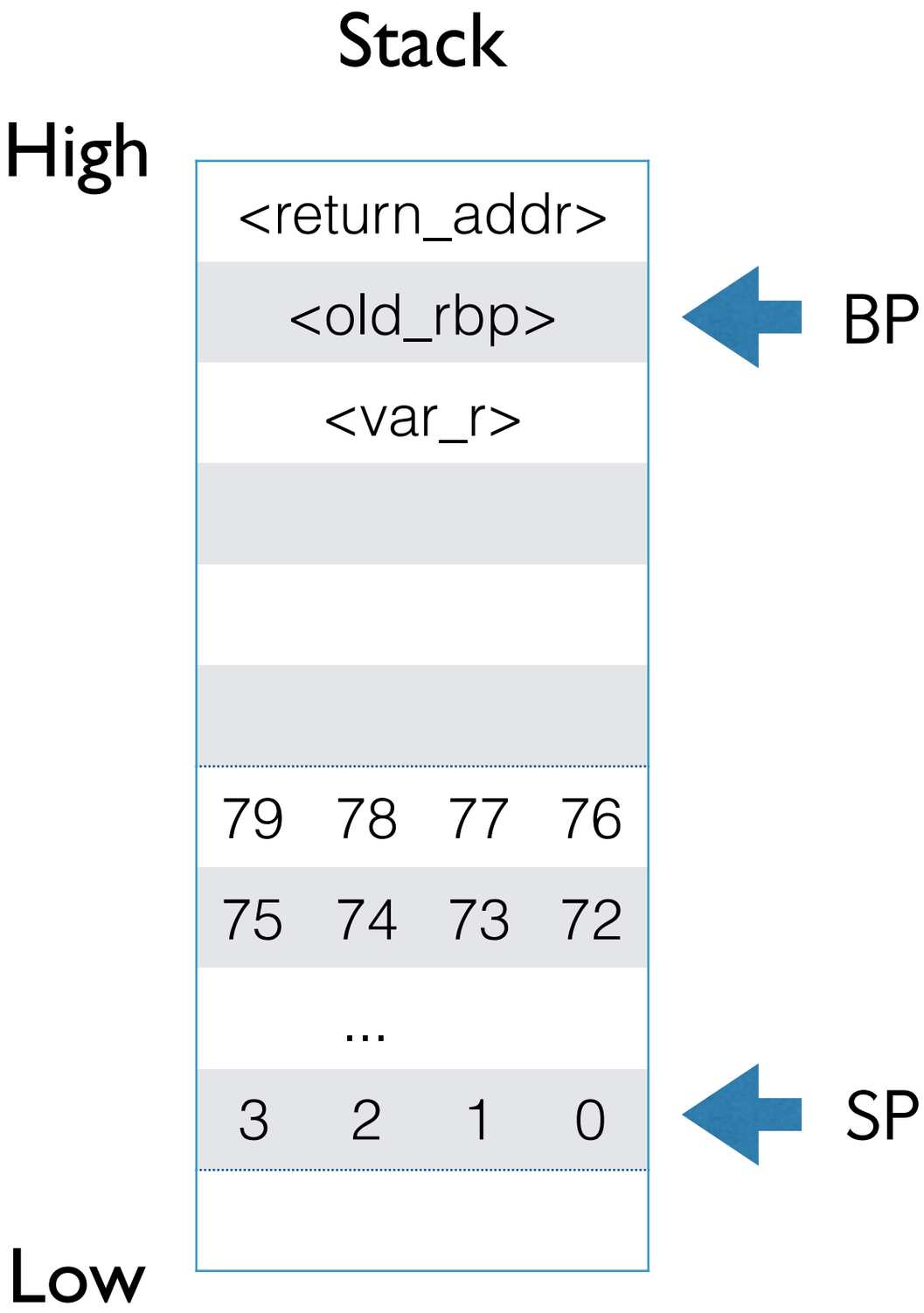
/

Subverting control flow

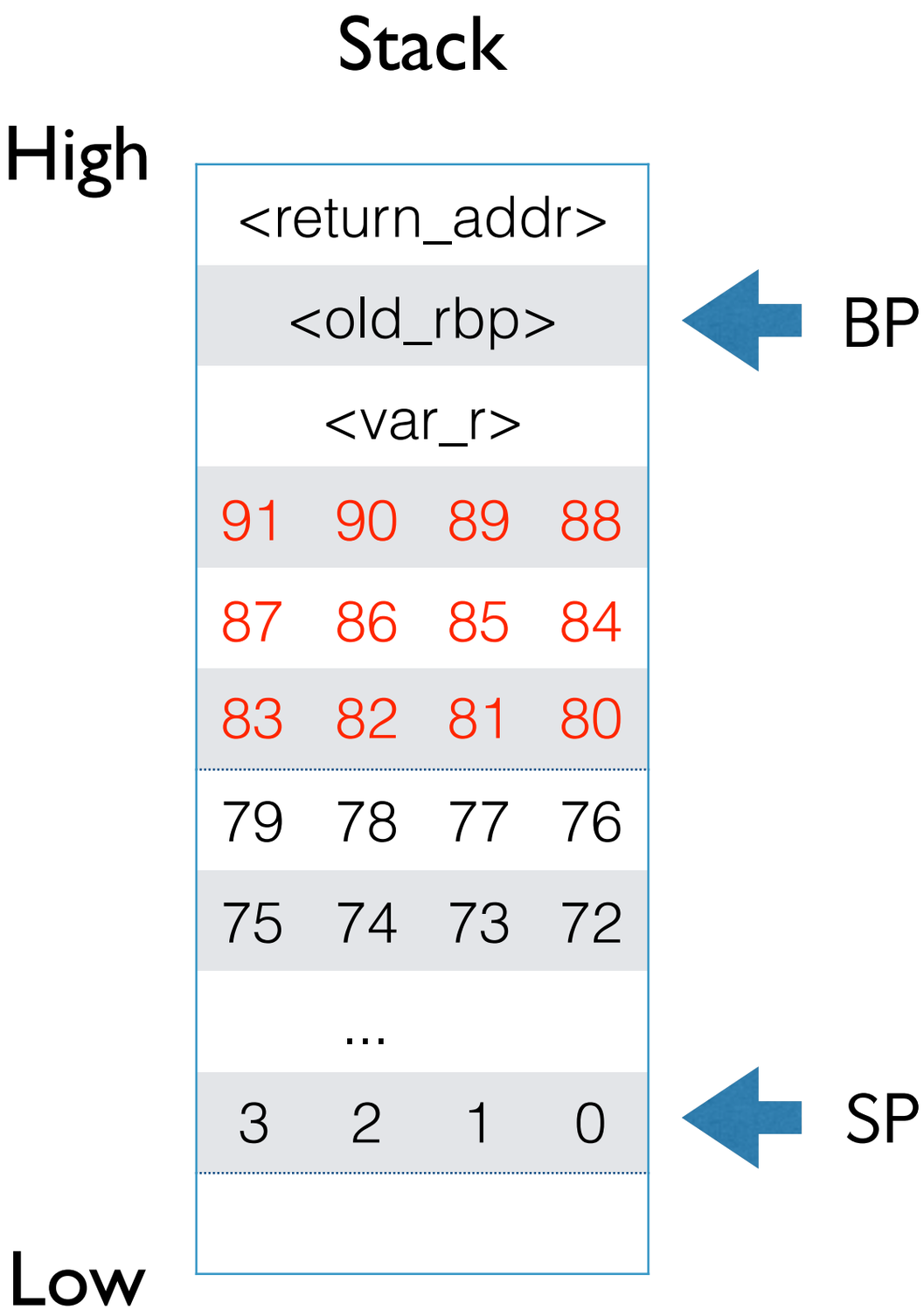



```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
IP call read

  // ...

  leave
  ret
```

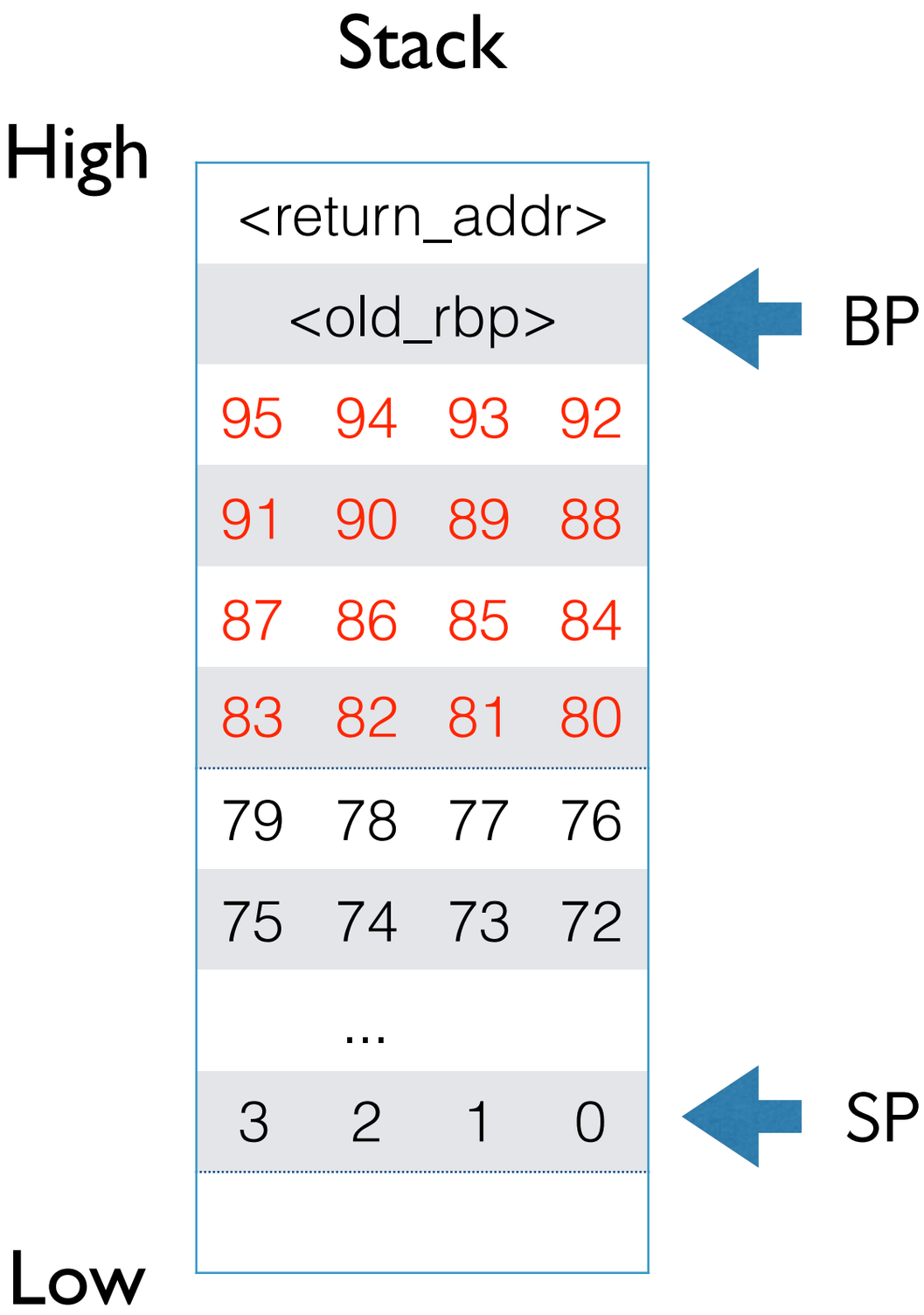
edi	rsi	edx
0	&buf	400



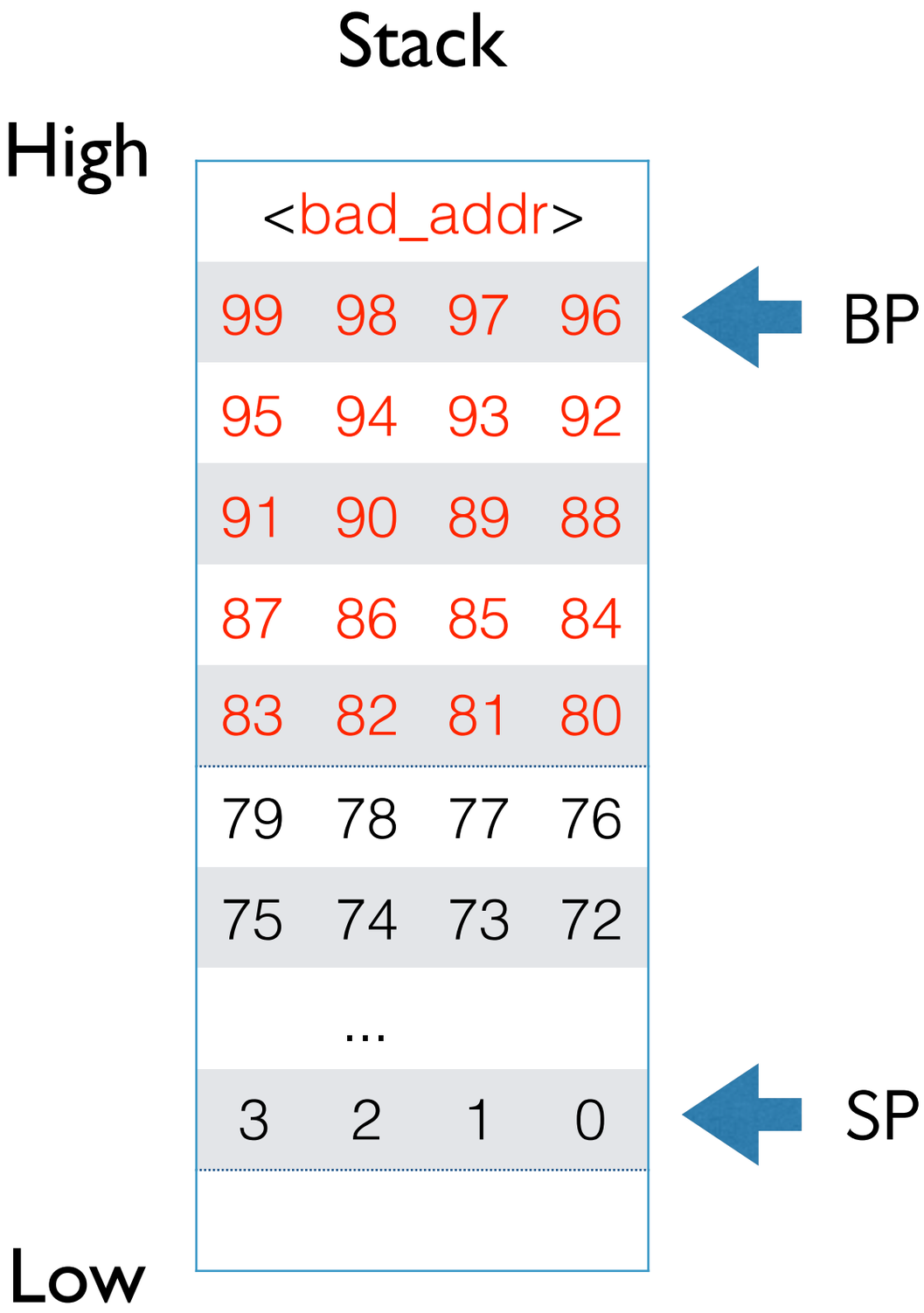

```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read
IP
  // ...
  leave
  ret
```




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read
IP
  // ...
  leave
  ret
```

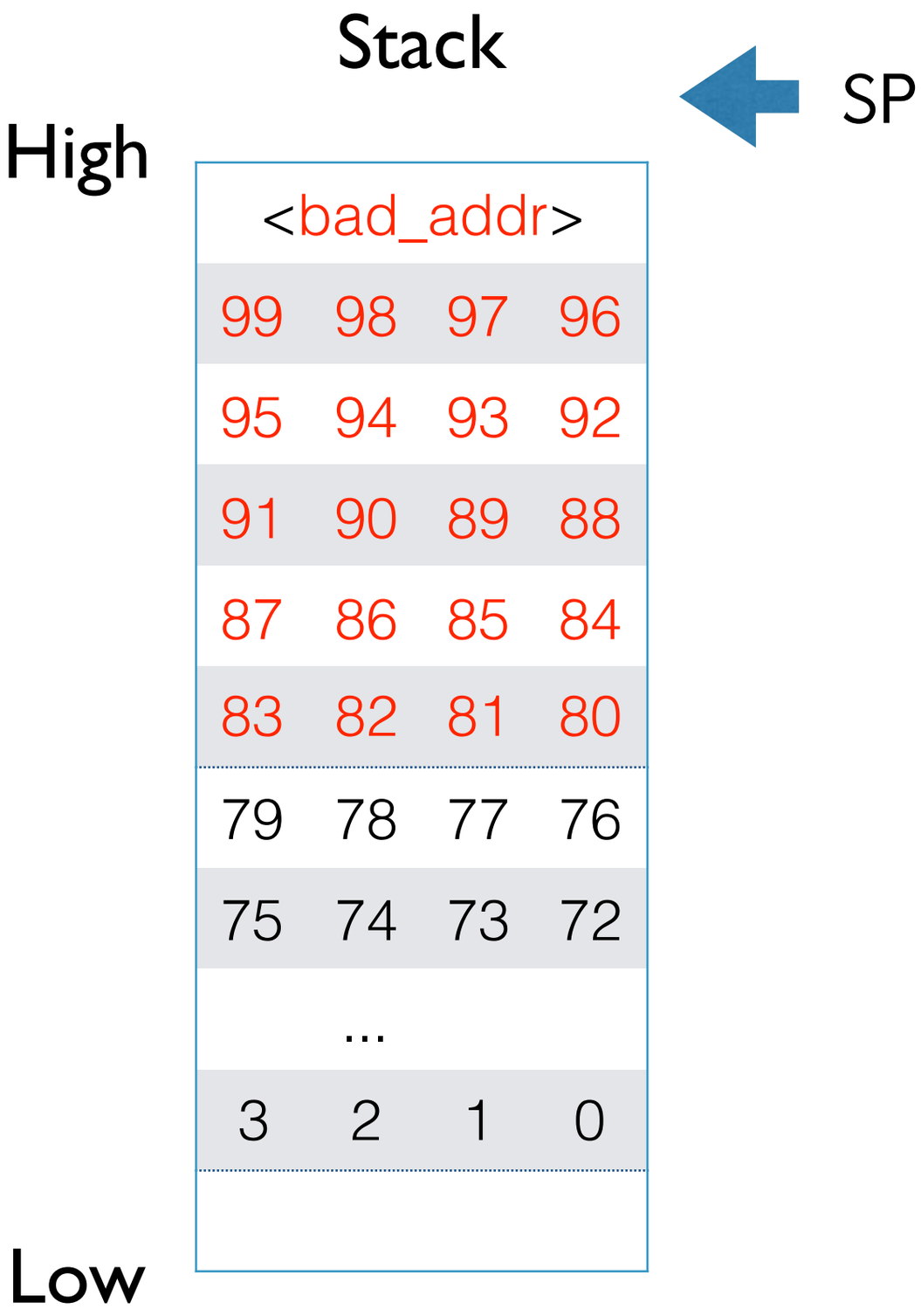



```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read
IP
  // ...
  leave
  ret
```



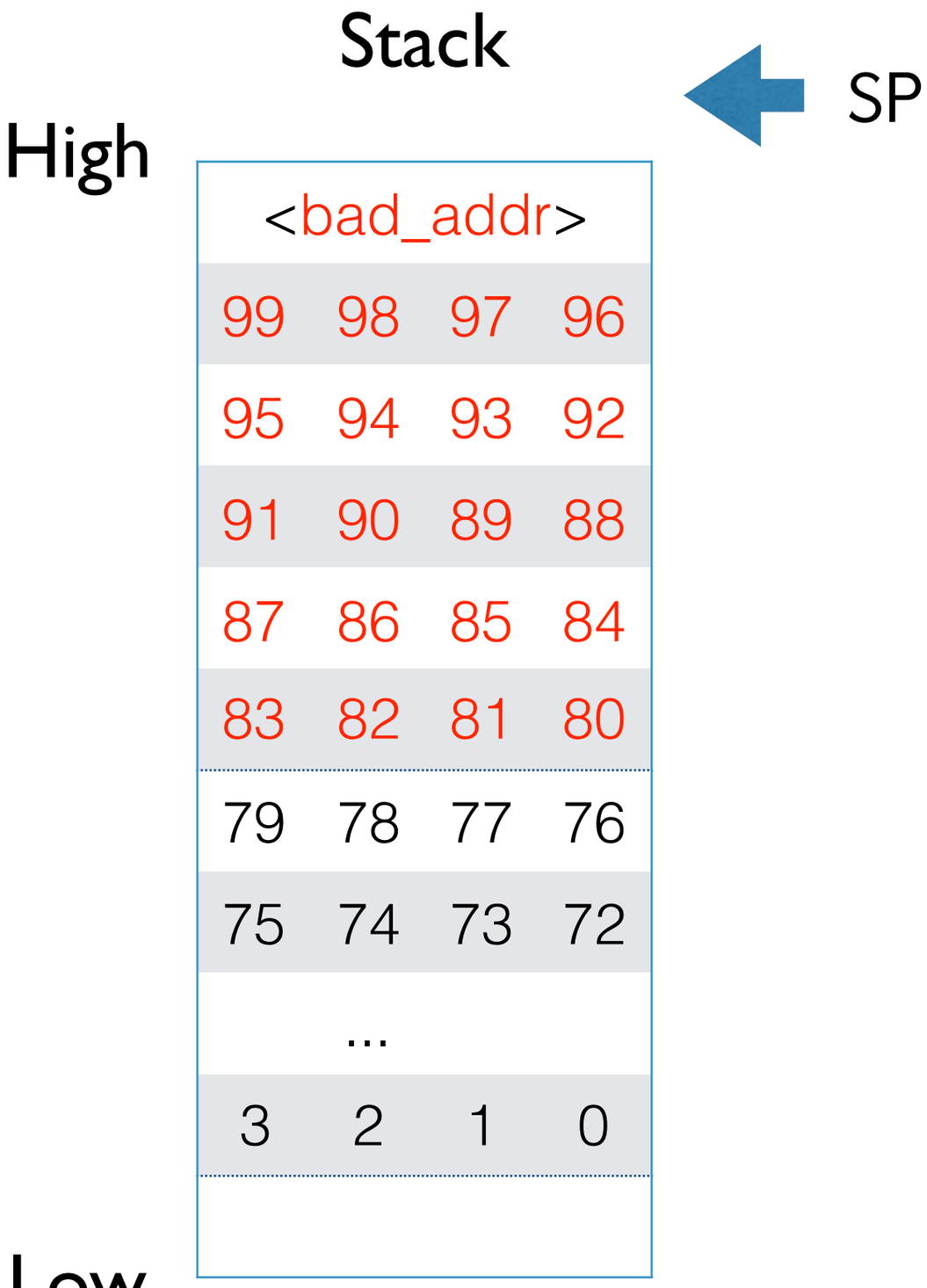

```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

IP




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

IP




```
vuln:
  push rbp
  mov rbp, rsp
  sub rsp, 96
  lea rax, [rbp-96]
  mov edx, 400
  mov rsi, rax
  mov edi, 0
  call read

  // ...

  leave
  ret
```

After function return

`IP` set to <bad_addr>

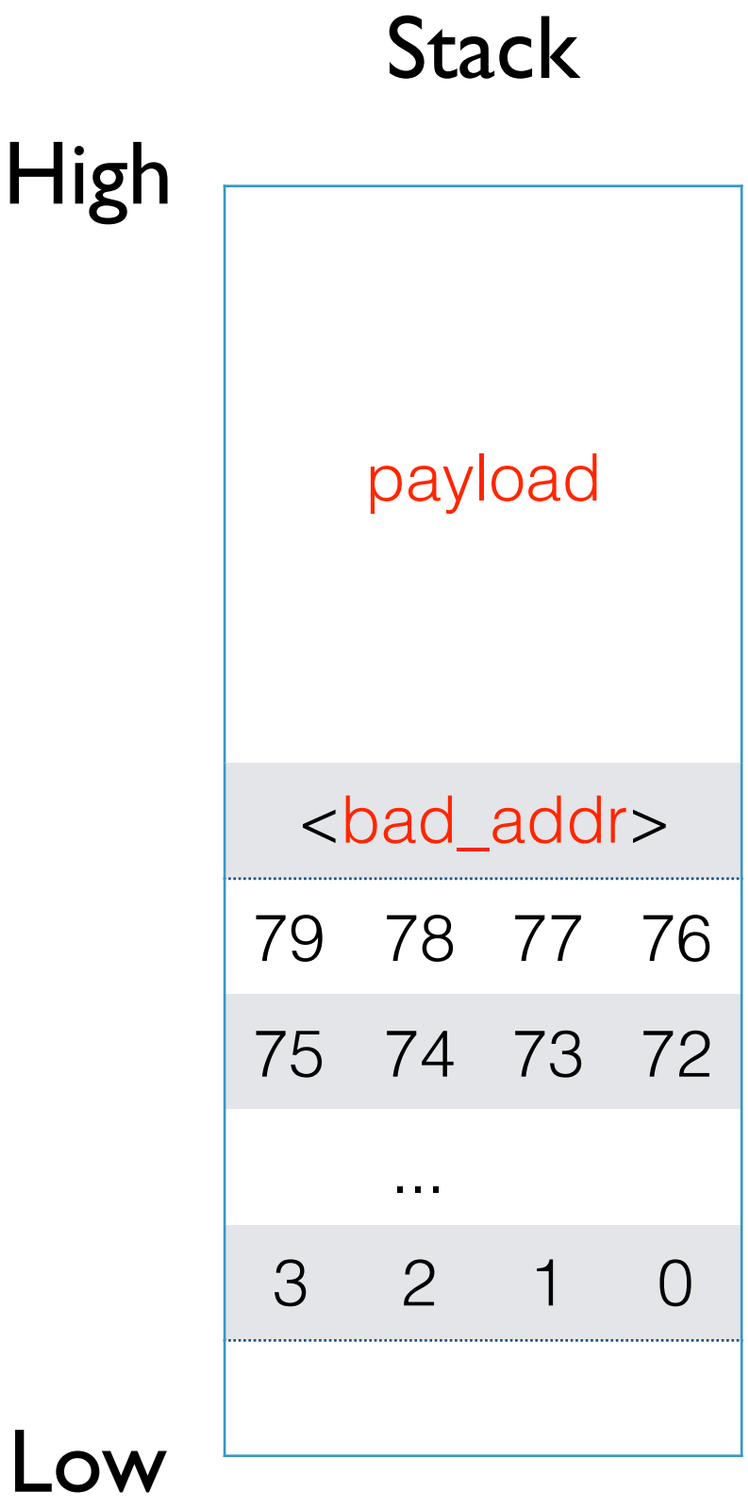
Where do we jump to?

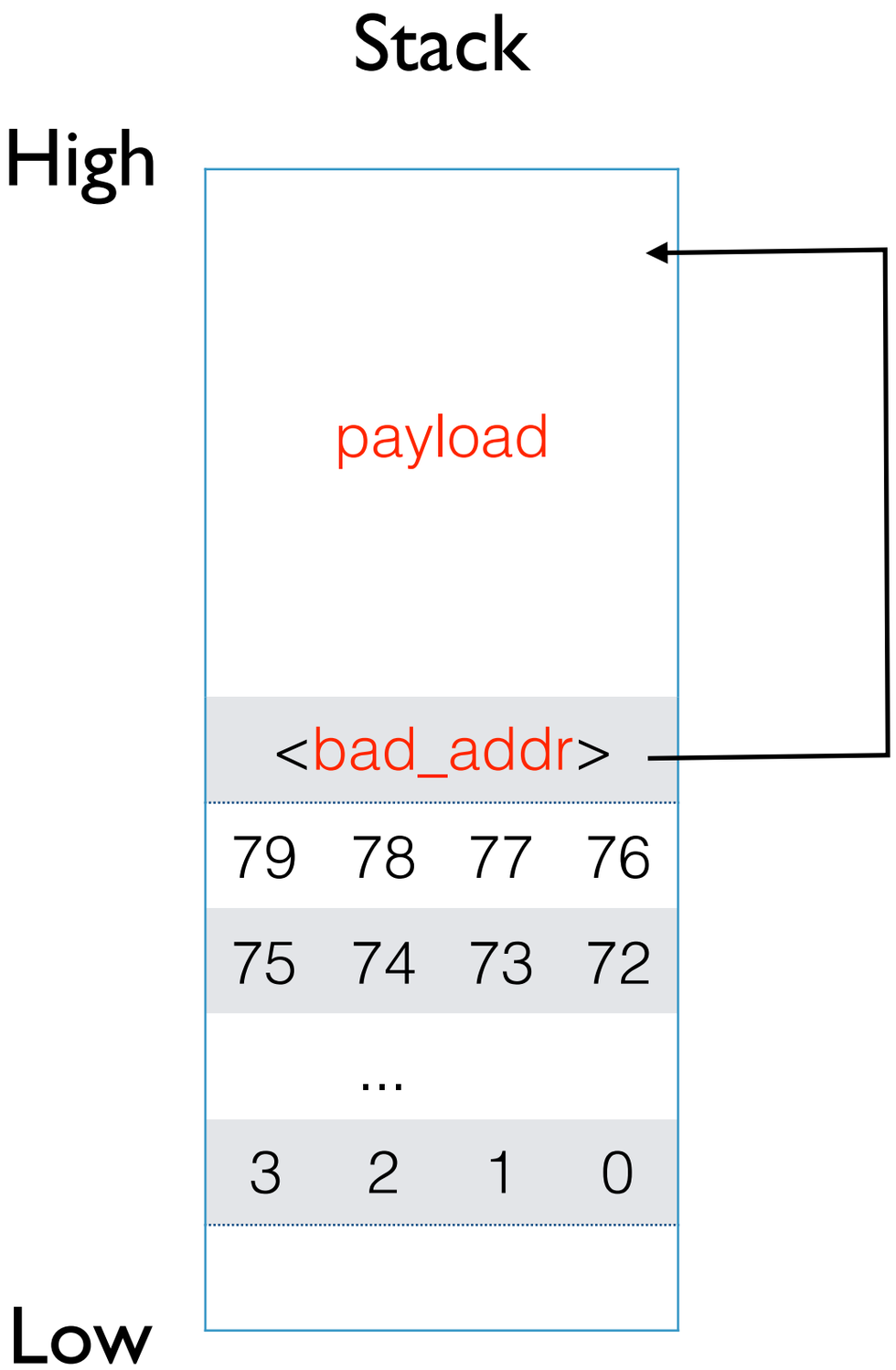
Stack

High

<bad_addr>			
79	78	77	76
75	74	73	72
	...		
3	2	1	0

Low


Stack
High
payload
<bad_addr>
79 78 77 76
75 74 73 72
...
3 2 1 0
Low



Shellcode, e.g.,

```
execve("/bin/sh")
```

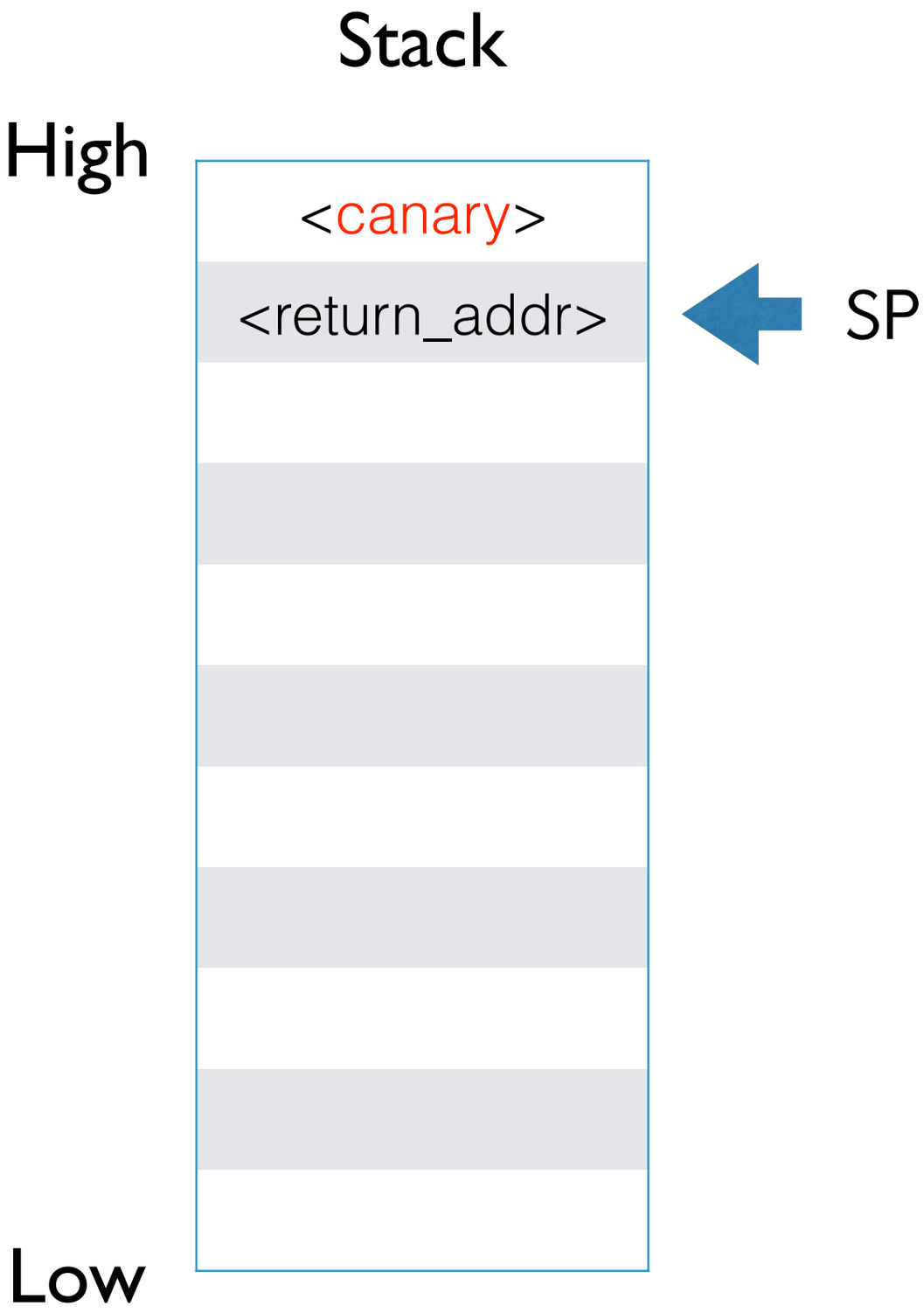
as a string of bytes

```
\x31\xc0\x48 ...
```

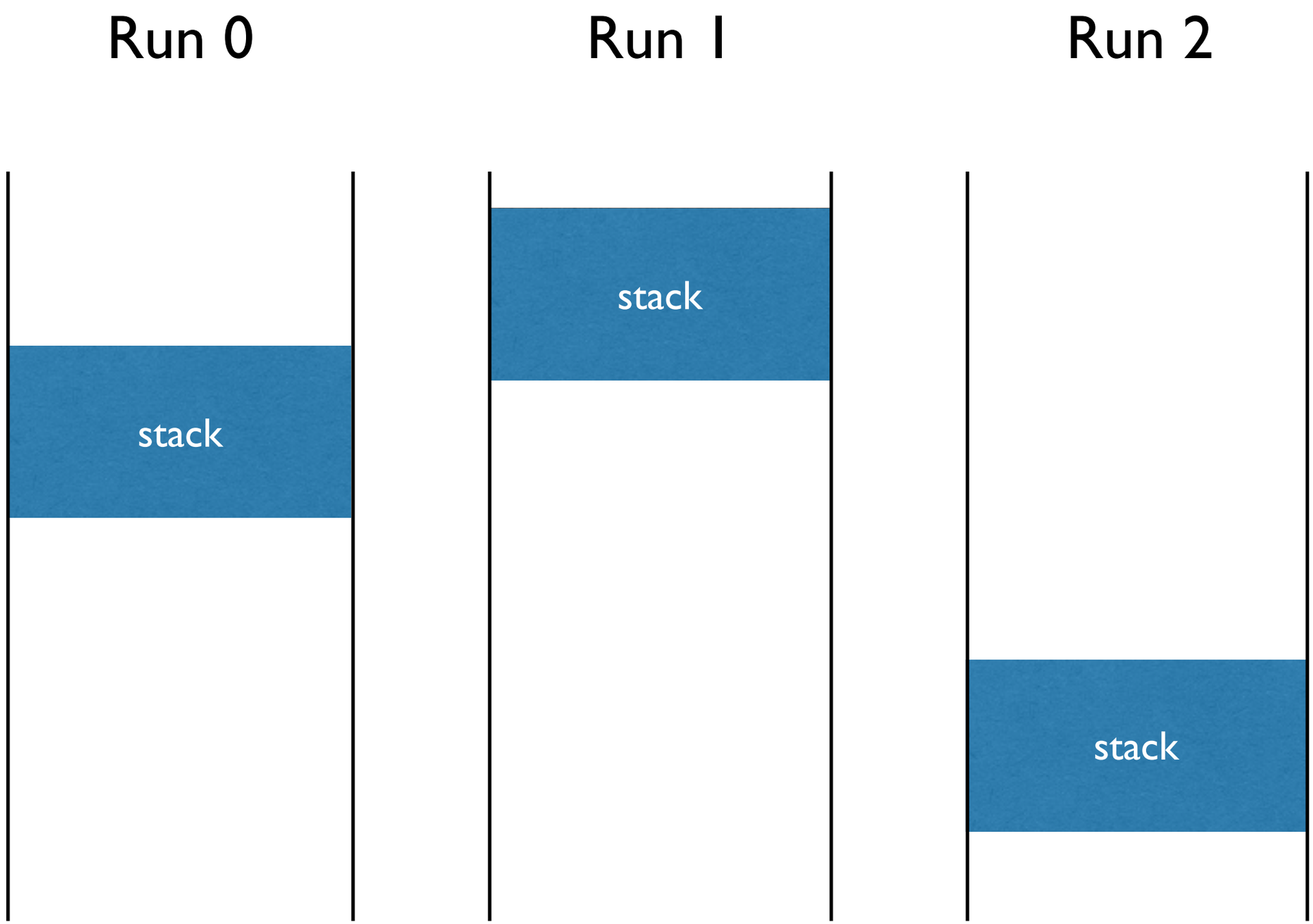
Vulnerability (bug) +
Subvert Control Flow +
Payload (shellcode)

Defences

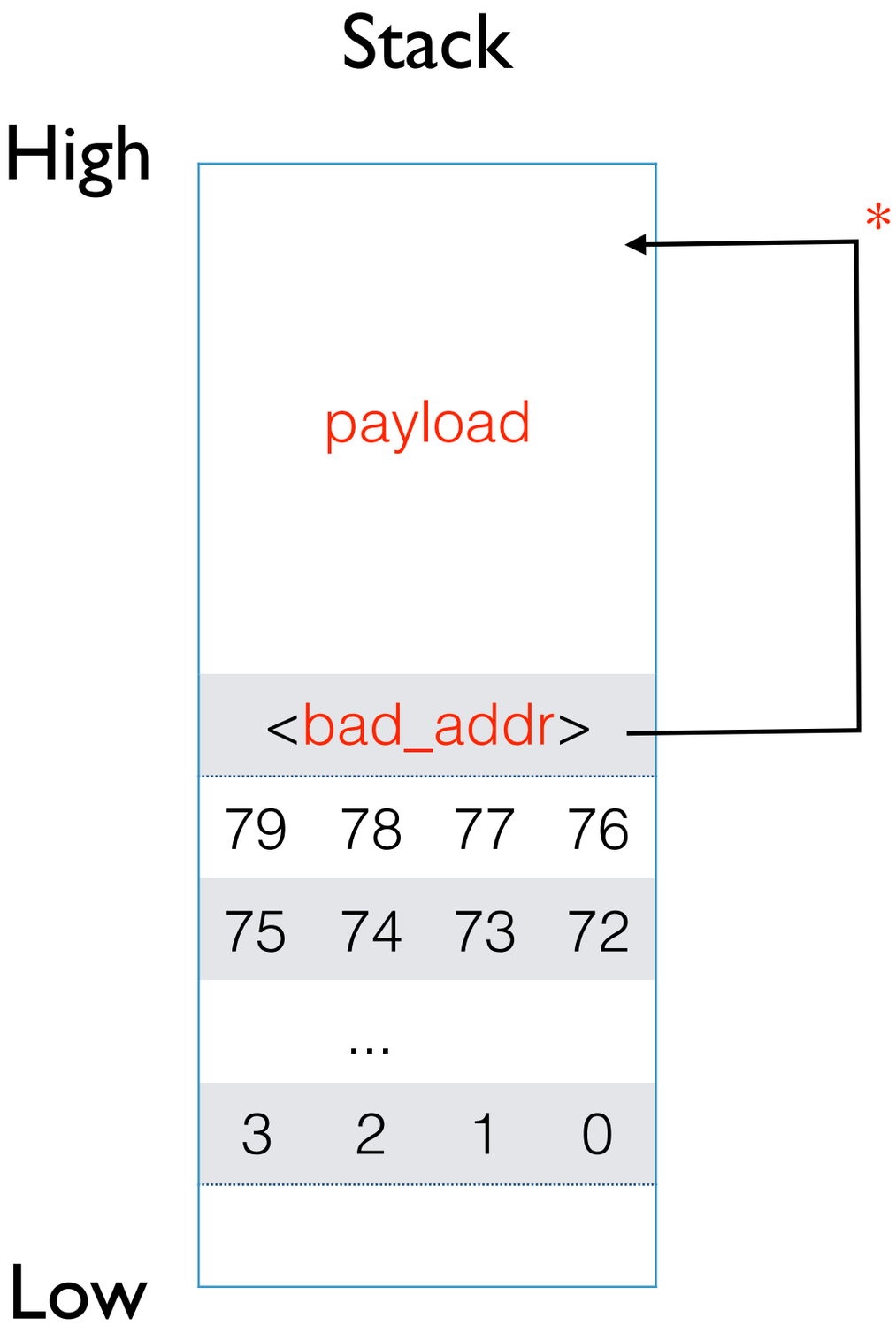
- Stack Canaries (remember `-fno-stack-protector`)
- Address Space Level Randomisation (ASLR)
- *Sanitise User Input*
- Execute Never (XN/W^X/DEP)




```
vuln:
IP  push rbp
    mov rbp, rsp
    sub rsp, 96
    lea rax, [rbp-96]
    mov edx, 400
    mov rsi, rax
    mov edi, 0
    call read

    // ...

    call check_canary
    leave
    ret
```



run-to-run variation in layout of program



* mark stack (and other data areas) as non-executable so this jump causes a memory protection fault

Stack Canaries

Vulnerability (bug) +
Subvert Control Flow +
Payload (shellcode)

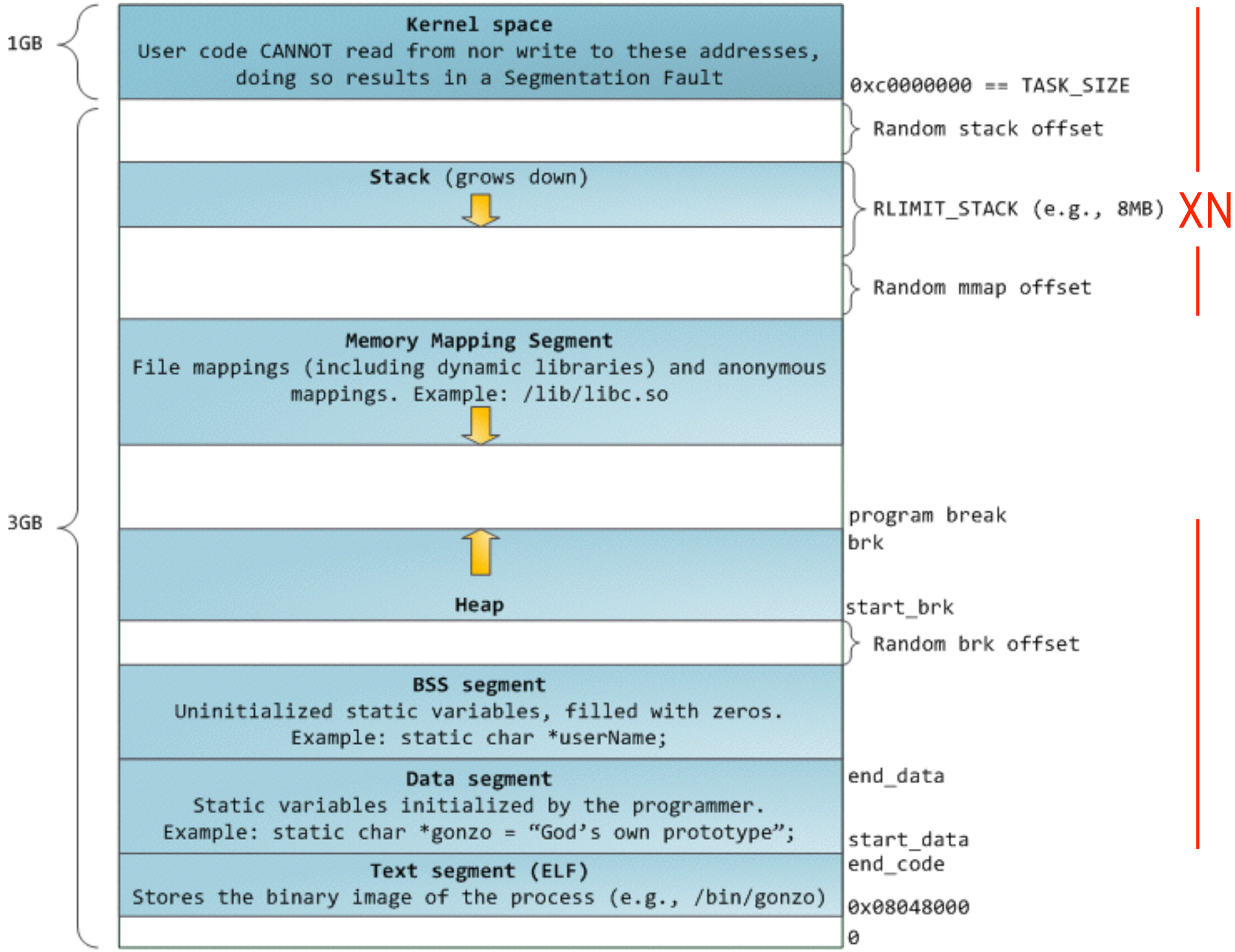
ASLR

Sanitise input, XN

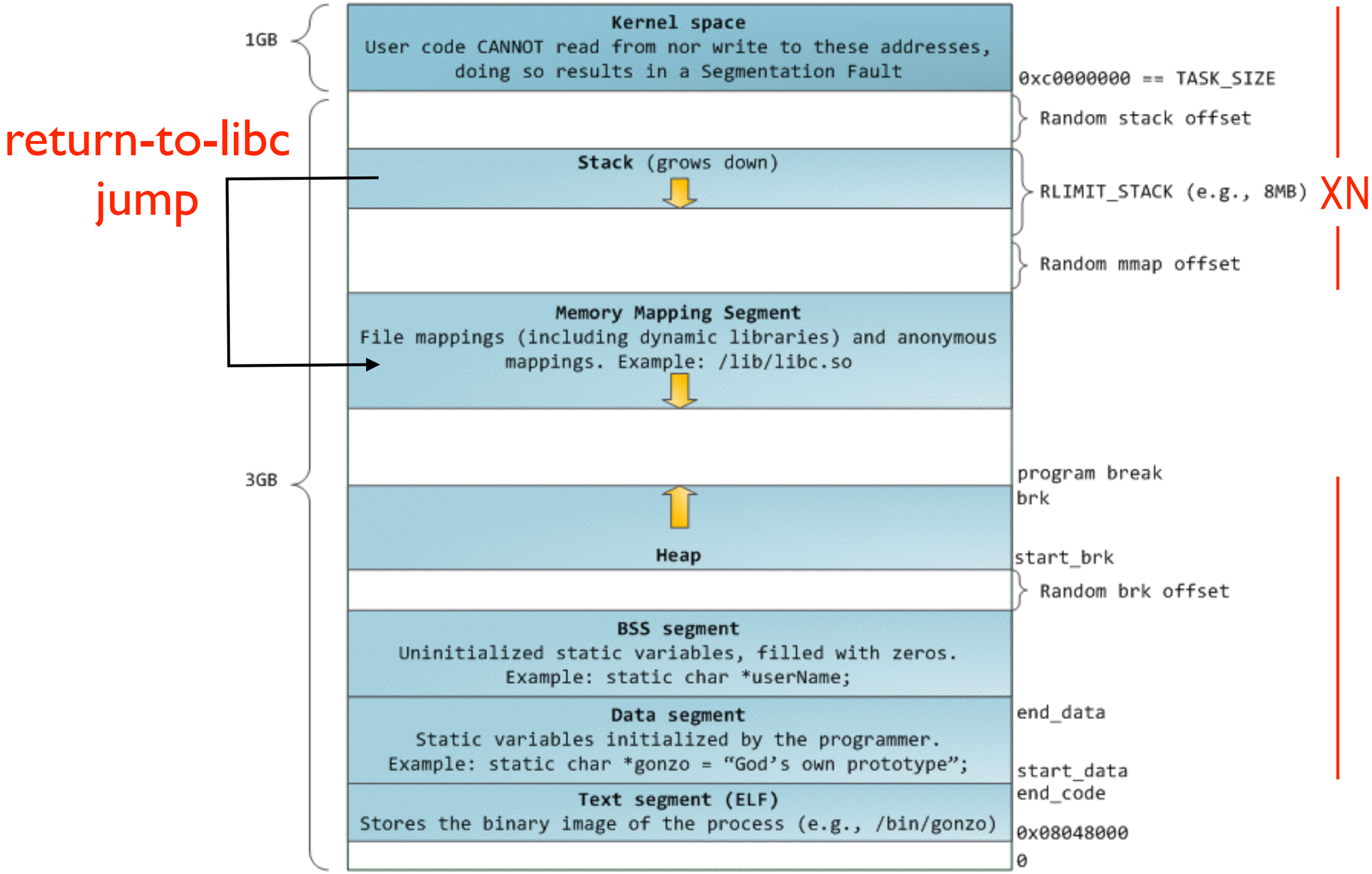
When?

- Programming (**test and verify**, input sanitisation)
- Compile-time / Dynamic Instrument (stack canaries)
- Runtime (ASLR, XN)

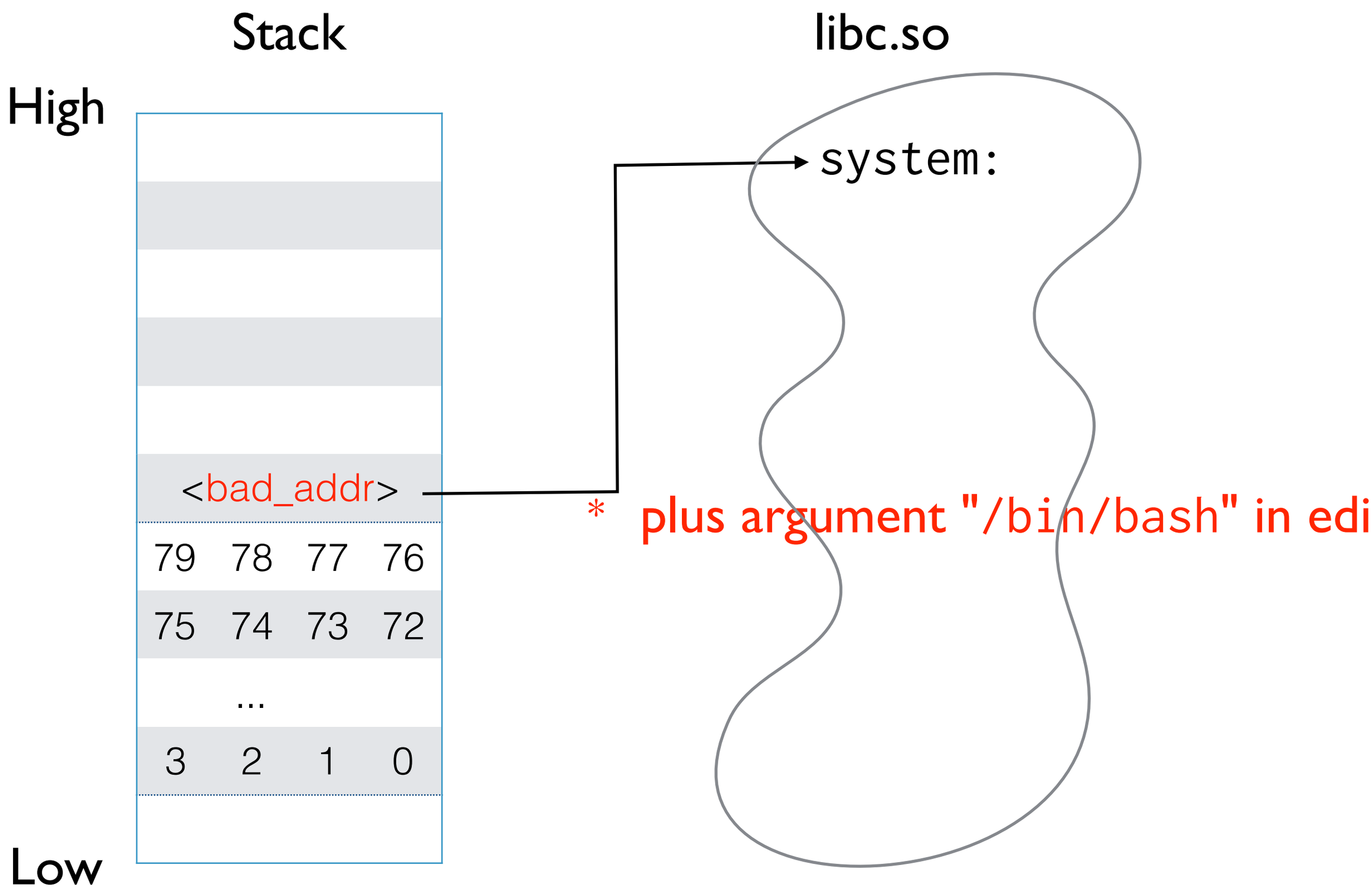
Return-to-Libc



http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png



http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png


Stack
libc.so
High
system:
<bad_addr>
* plus argument "/bin/bash" in edi
79 78 77 76
75 74 73 72
...
3 2 1 0
Low

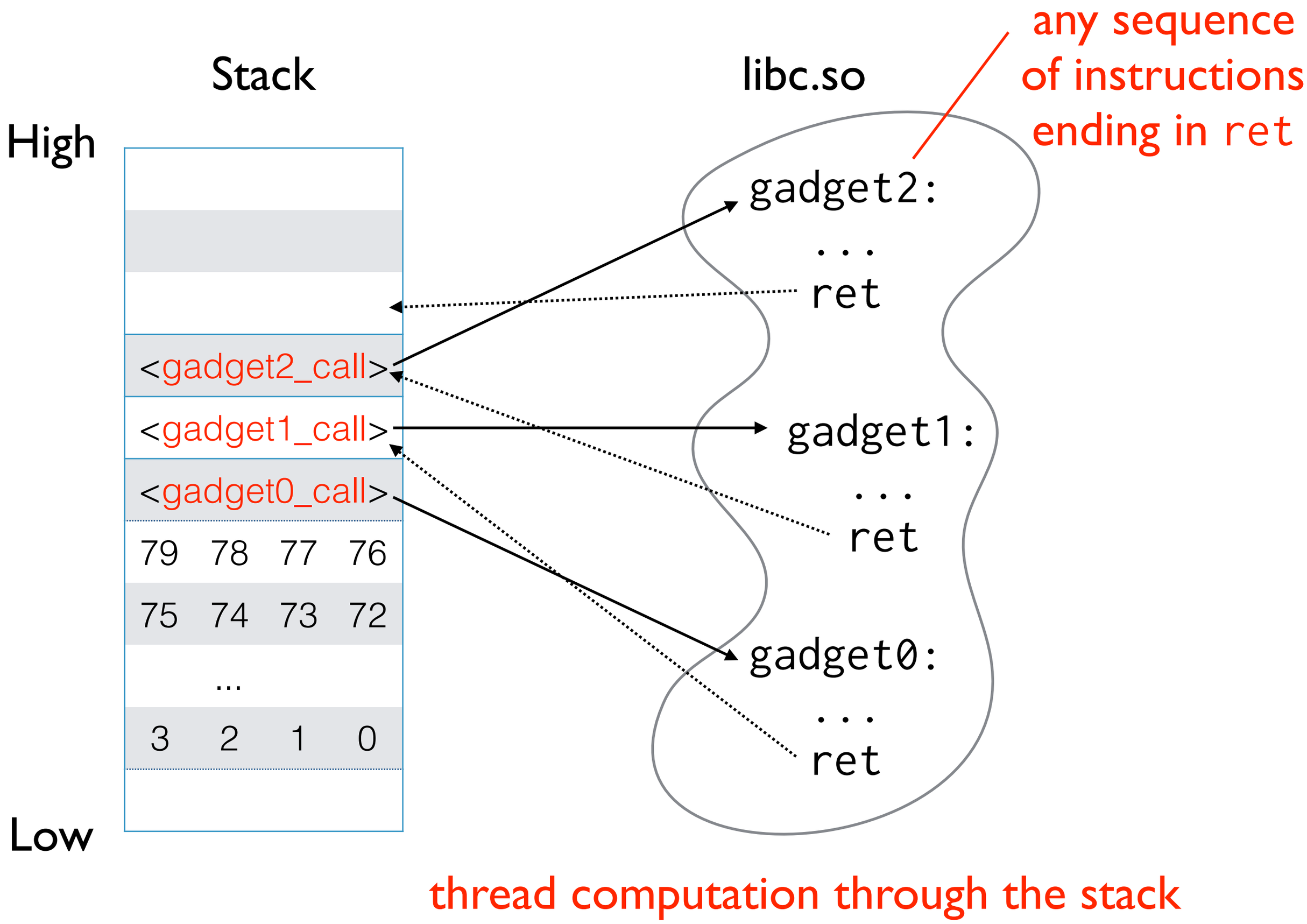
Assumed Limitations

- Straight-line calls into functions (no branching)
- Restricted to code in loaded libraries / text segment

Full version of an extended abstract published in Proceedings of ACM CCS 2007, ACM Press, 2007.

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

Hovav Shacham*
hovav@cs.ucsd.edu


Stack
libc.so
any sequence of instructions ending in ret
High
Low
<gadget2_call>
<gadget1_call>
<gadget0_call>
79 78 77 76
75 74 73 72
...
3 2 1 0
gadget2:
...
ret
gadget1:
...
ret
gadget0:
...
ret
thread computation through the stack

Key Observation

```
f7 c7 07 00 00 00          test $0x0000007, %edi
0f 95 45 c3                setnzb -61(%ebp)

c7 07 00 00 00 0f          movl $0x0f000000, (%edi)
95                         xchg %ebp, %eac
45                         inc %ebp
c3                         ret
```

Our thesis: In any sufficiently large body of **x86** executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

Gadgets

- NOP
- Load Immediate and Load / Store Memory
- Arithmetic: Add, Sub, Xor, Shifts, Rotates
- Control Flow: Jumps, System Calls, Function Calls

- Found in unintended sequences of libc (thwart XN)
- Allowing arbitrary computation (thwart limitations of ret2libc)

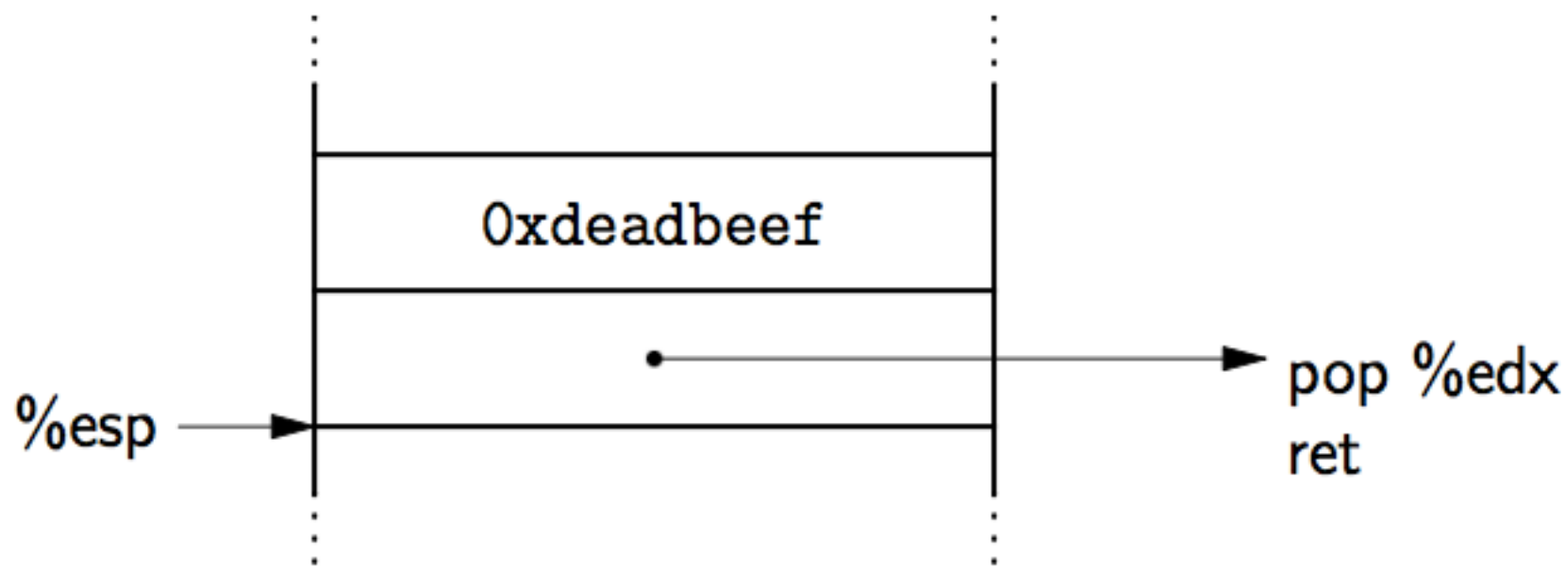


Figure 2: Load the constant `0xdeadbeef` into %edx.

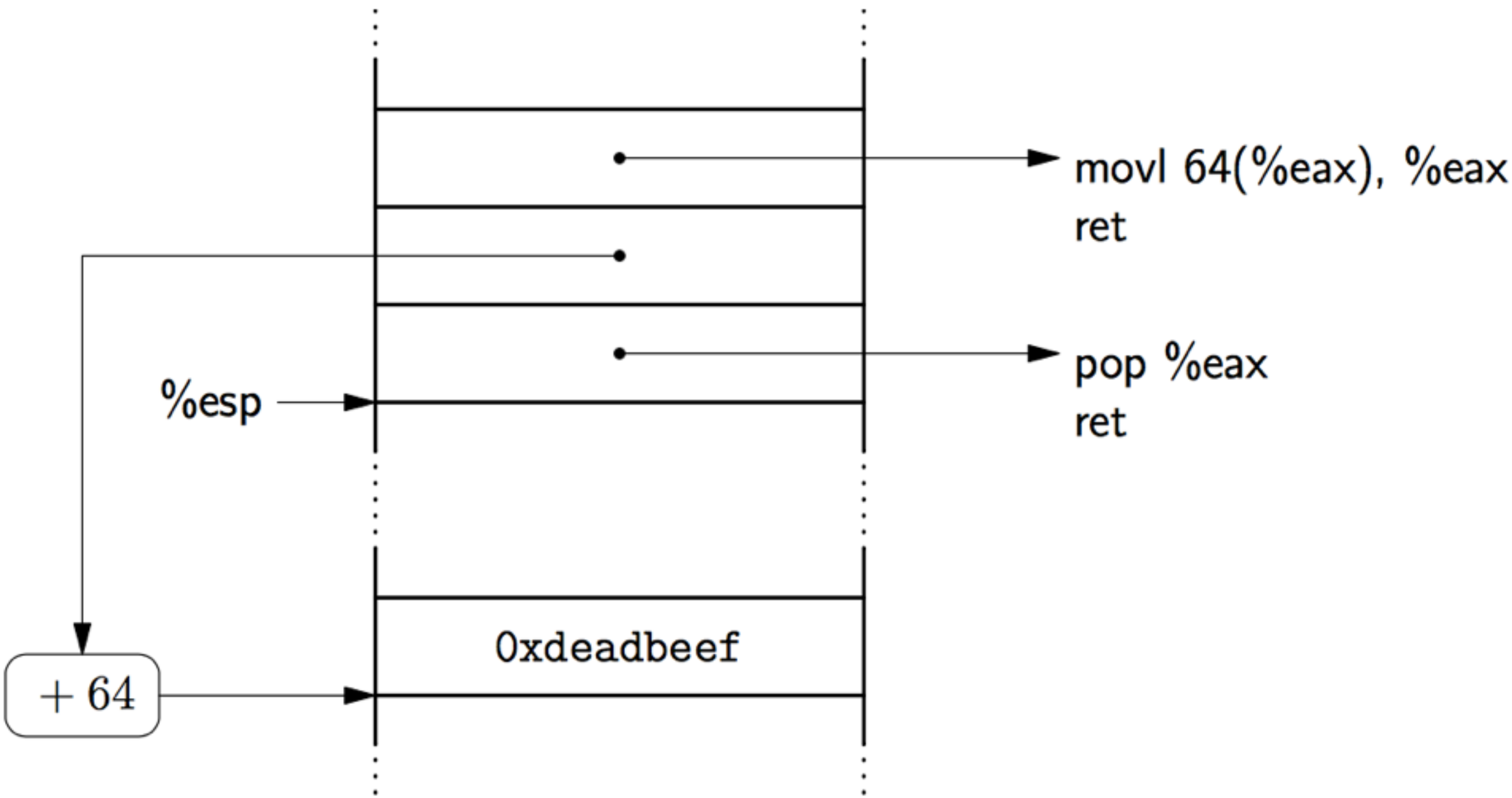


Figure 3: Load a word in memory into %eax.

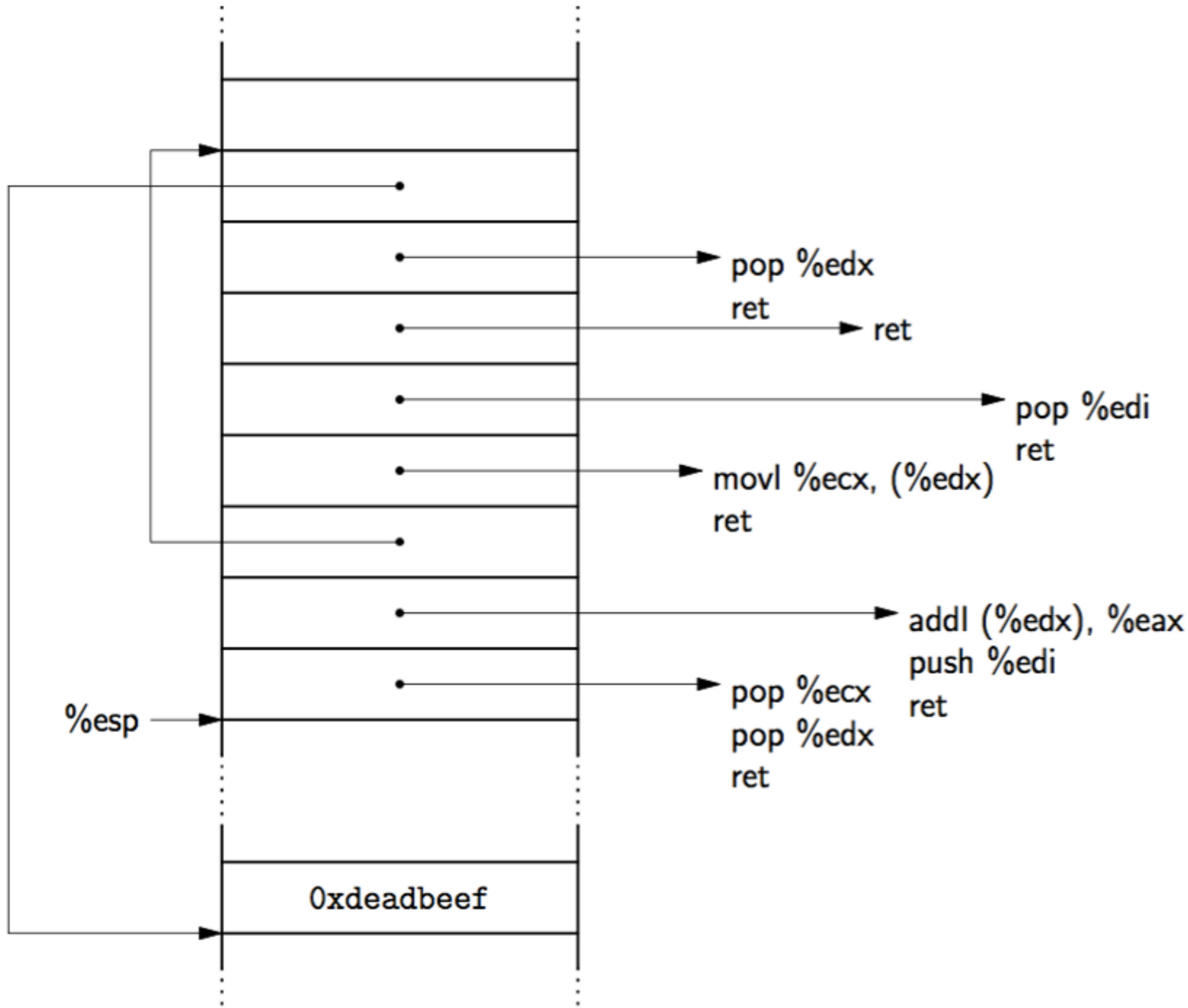


Figure 7: Repeatable add into %eax.

More precisely

- Search a given libc version for "interesting" sequences
- Construct gadgets for necessary operations
- Program a shellcode (e.g., execv('/bin/bash')) using this gadget programming language

Summary

- Interesting interplay between software testing/ verification and security

- Arms race between attackers and defenders

- New architectural extensions to combat return-oriented programming