CSED211 Lab 04.

Buffer overflow lab.

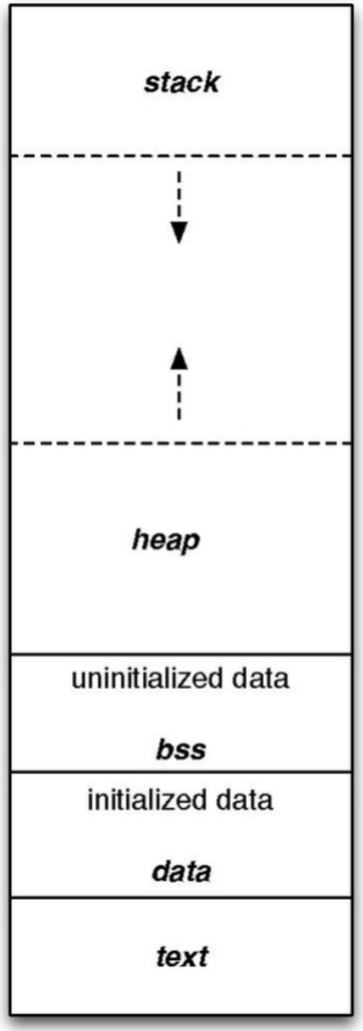
19. 10. 02.

Sangwoo Ji

Office hour

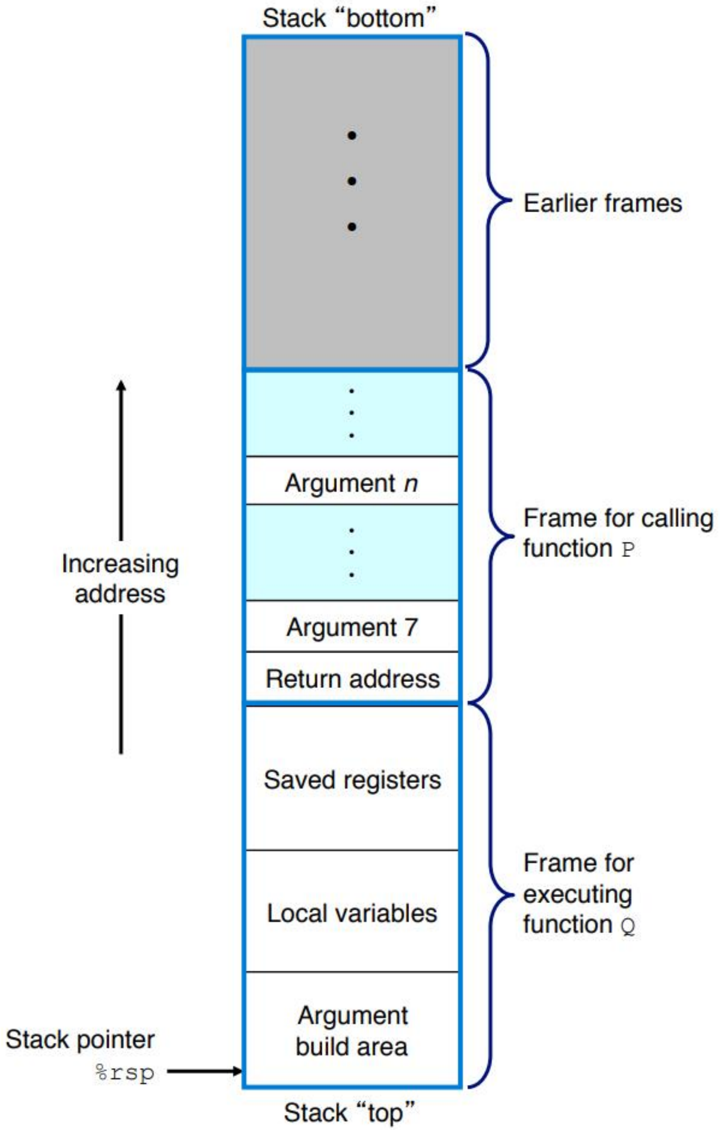
- 지상우
    - TUE 10:15-11:15 / PIRL 454
    - 부재중 -> PIRL 441 / sangwooji@postech.ac.kr / 010-9807-1279
- 나동빈
    - WED 13:30-14:30 / PIRL 454

Program memory layout



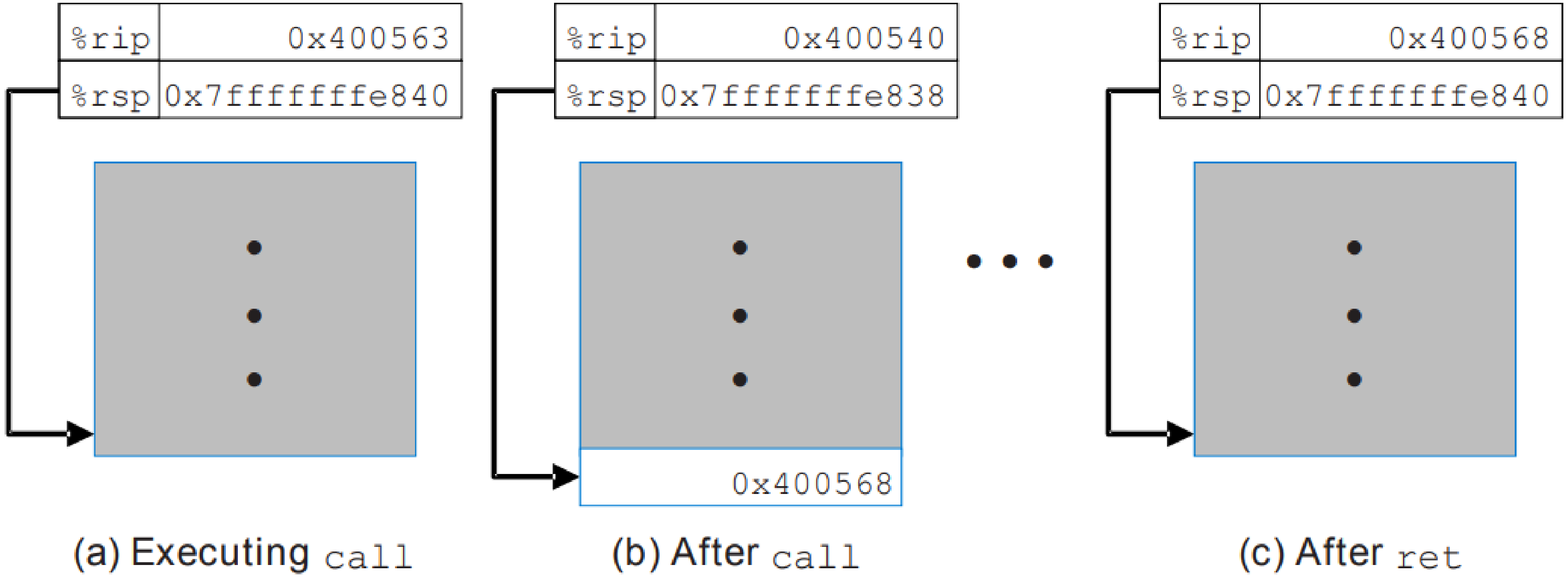
- Each program has its own address space
- Program memory layout consists of
  - Stack
  - Heap
  - Uninitialized data
  - Initialized data
  - Code

Stack discipline



- Stack grows downwards
- Stack stores
  - Local variable
  - Function arguments
  - Return address
  - Register value

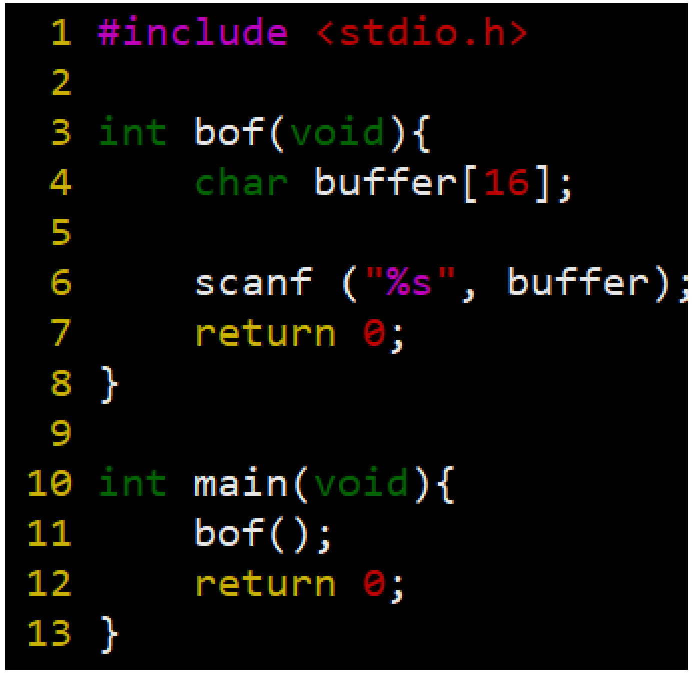
Call and Return



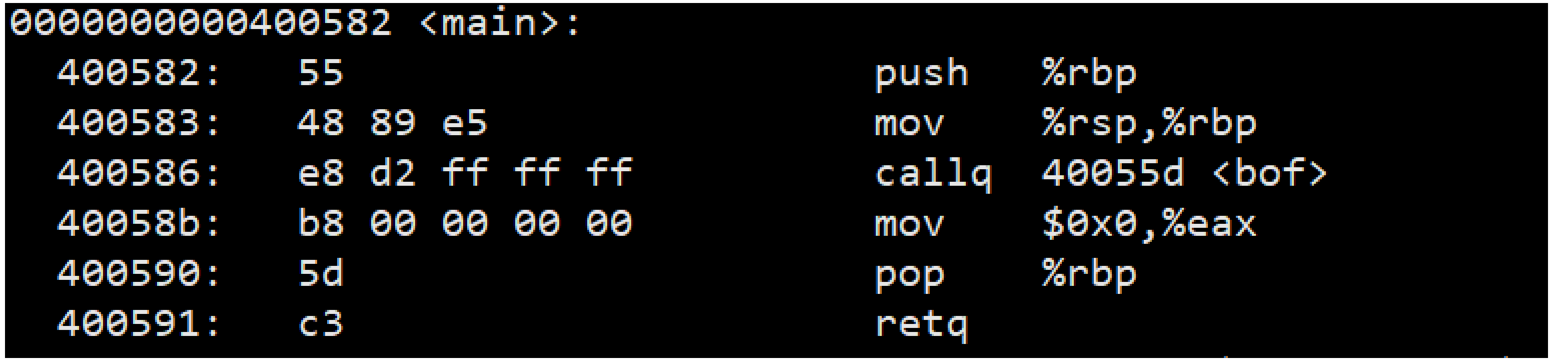
(a) Executing `call` (b) After `call` (c) After `ret`

- Procedure call saves the next instruction address of the current instruction, and return restores the saved instruction address

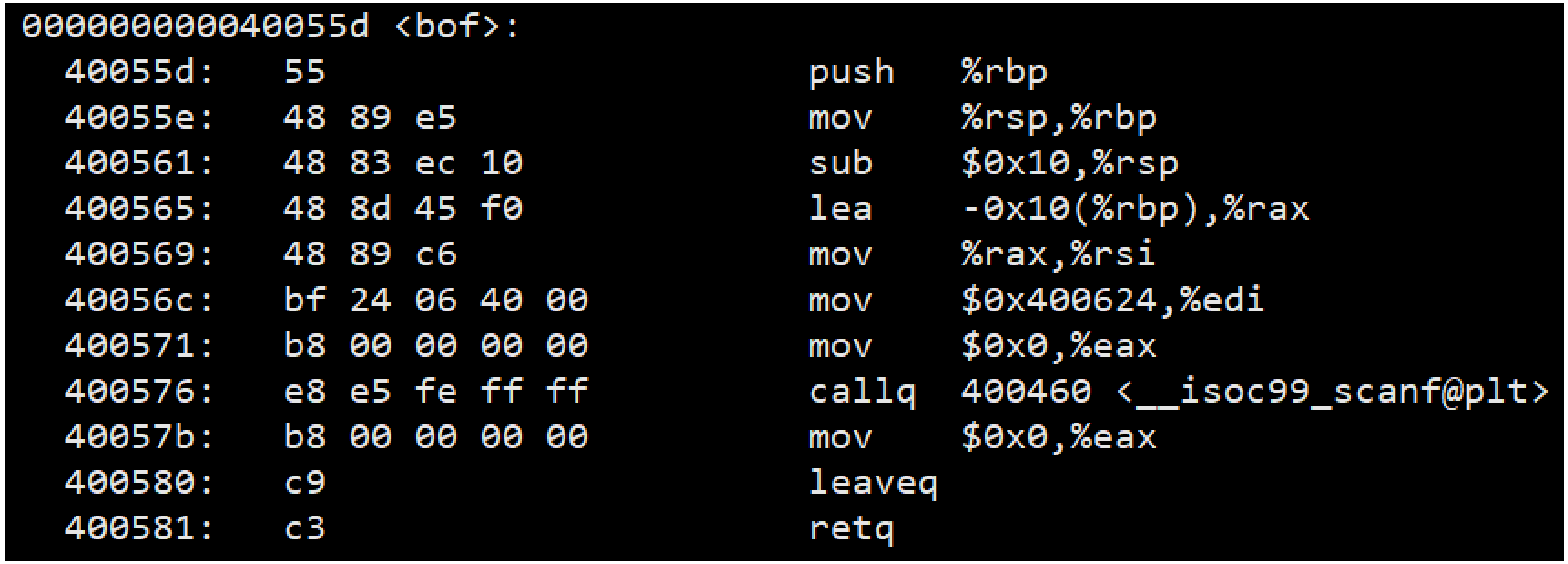
Example: simple scanf program



```c
#include <stdio.h>

int bof(void){
    char buffer[16];

    scanf ("%s", buffer);
    return 0;
}

int main(void){
    bof();
    return 0;
}
```



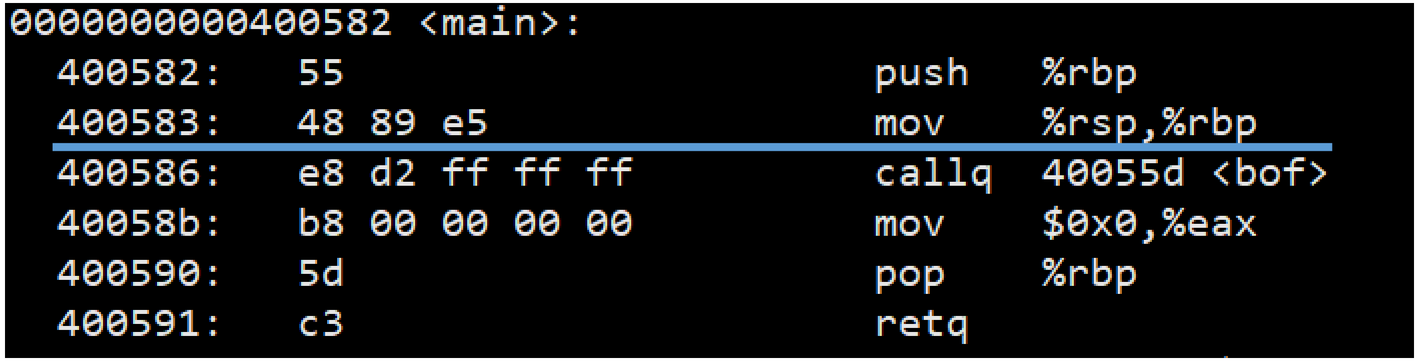
```
0000000000400582 <main>:
  400582:	55                   	push   %rbp
  400583:	48 89 e5             	mov    %rsp,%rbp
  400586:	e8 d2 ff ff ff       	callq  40055d <bof>
  40058b:	b8 00 00 00 00       	mov    $0x0,%eax
  400590:	5d                   	pop    %rbp
  400591:	c3                   	retq
```



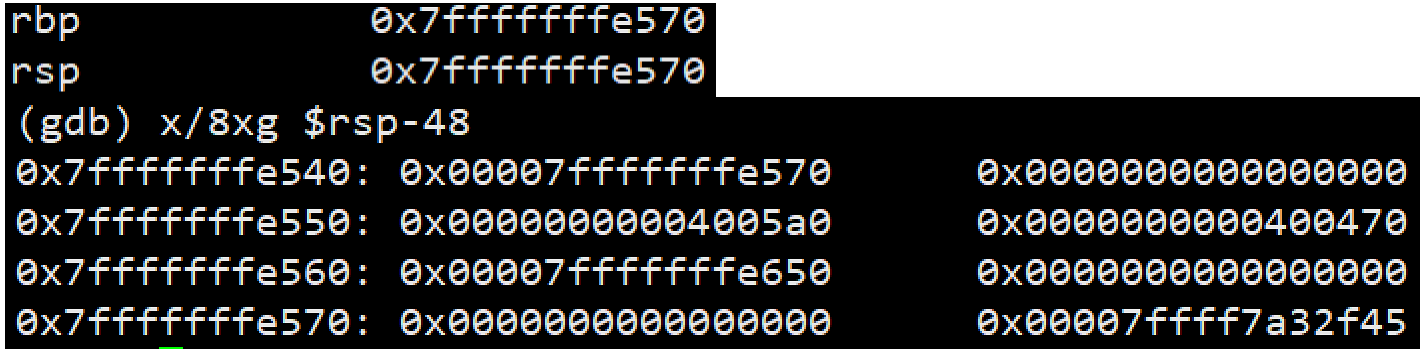
```
000000000040055d <bof>:
  40055d:	55                   	push   %rbp
  40055e:	48 89 e5             	mov    %rsp,%rbp
  400561:	48 83 ec 10          	sub    $0x10,%rsp
  400565:	48 8d 45 f0          	lea    -0x10(%rbp),%rax
  400569:	48 89 c6             	mov    %rax,%rsi
  40056c:	bf 24 06 40 00       	mov    $0x400624,%edi
  400571:	b8 00 00 00 00       	mov    $0x0,%eax
  400576:	e8 e5 fe ff ff       	callq  400460 <__isoc99_scanf@plt>
  40057b:	b8 00 00 00 00       	mov    $0x0,%eax
  400580:	c9                   	leaveq
  400581:	c3                   	retq
```

Memory layout example

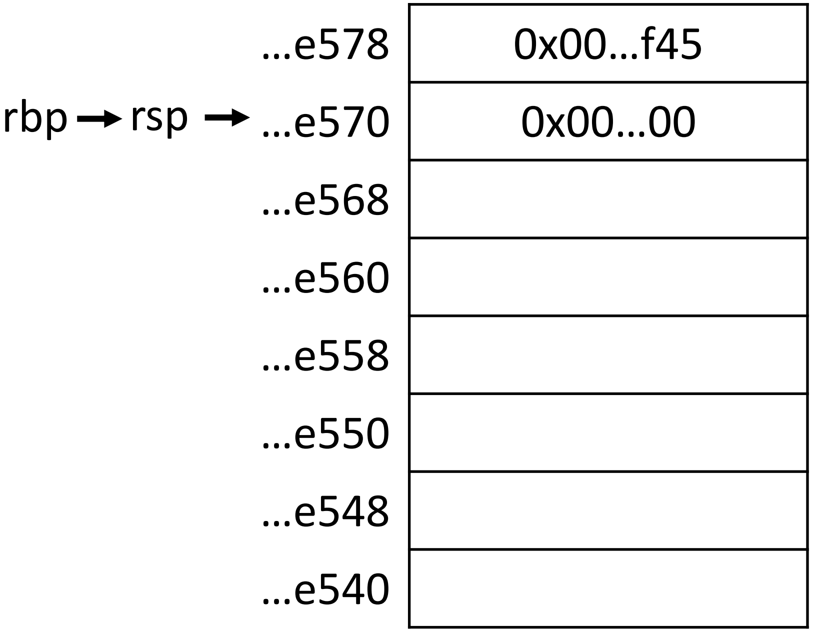
- Before bof call



```
0000000000400582 <main>:
  400582:   55                         push   %rbp
  400583:   48 89 e5                   mov    %rsp,%rbp
  400586:   e8 d2 ff ff ff             callq  40055d <bof>
  40058b:   b8 00 00 00 00             mov    $0x0,%eax
  400590:   5d                         pop    %rbp
  400591:   c3                         retq
```



```
rbp            0x7fffffffe570
rsp            0x7fffffffe570
(gdb) x/8xg $rsp-48
0x7fffffffe540: 0x00007fffffffe570      0x0000000000000000
0x7fffffffe550: 0x00000000004005a0      0x0000000000400470
0x7fffffffe560: 0x00007fffffffe650      0x0000000000000000
0x7fffffffe570: 0x0000000000000000      0x00007ffff7a32f45
```

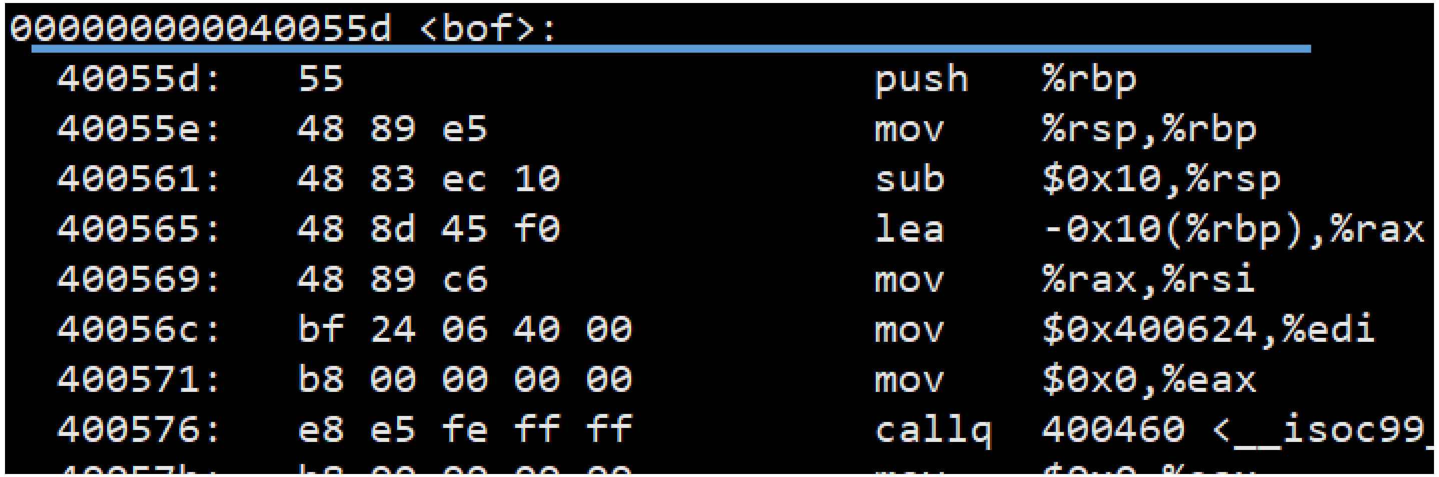


rbp → rsp → …e570

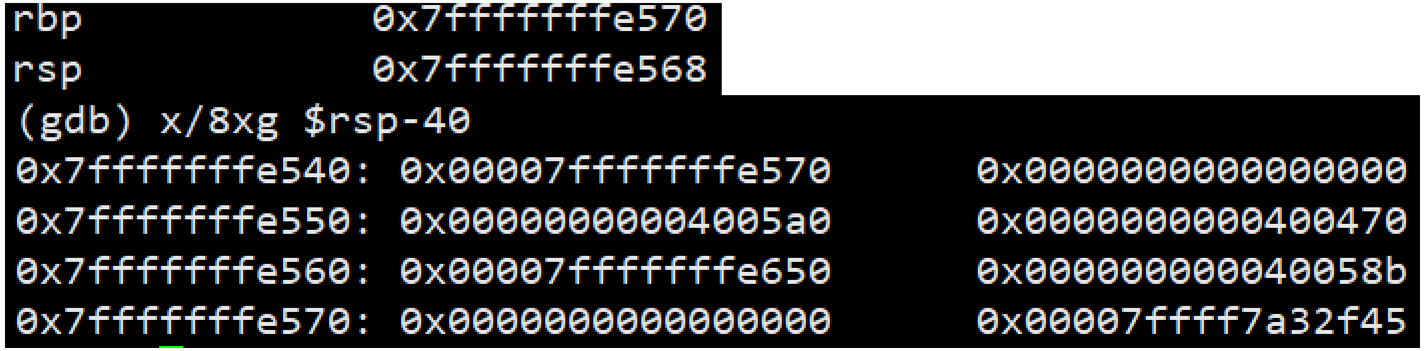
…e578	0x00…f45
…e570	0x00…00
…e568	
…e560	
…e558	
…e550	
…e548	
…e540	

Memory layout example (cont.)

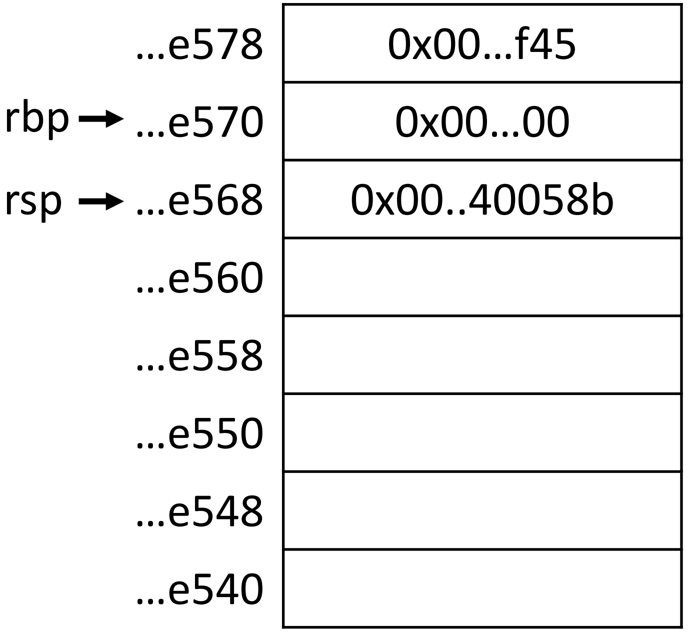
- After bof call



```
000000000040055d <bof>:
  40055d:	55                   	push   %rbp
  40055e:	48 89 e5             	mov    %rsp,%rbp
  400561:	48 83 ec 10          	sub    $0x10,%rsp
  400565:	48 8d 45 f0          	lea    -0x10(%rbp),%rax
  400569:	48 89 c6             	mov    %rax,%rsi
  40056c:	bf 24 06 40 00       	mov    $0x400624,%edi
  400571:	b8 00 00 00 00       	mov    $0x0,%eax
  400576:	e8 e5 fe ff ff       	callq  400460 <__isoc99_
  40057b:	b8 00 00 00 00       	mov    $0x0,%eax
```



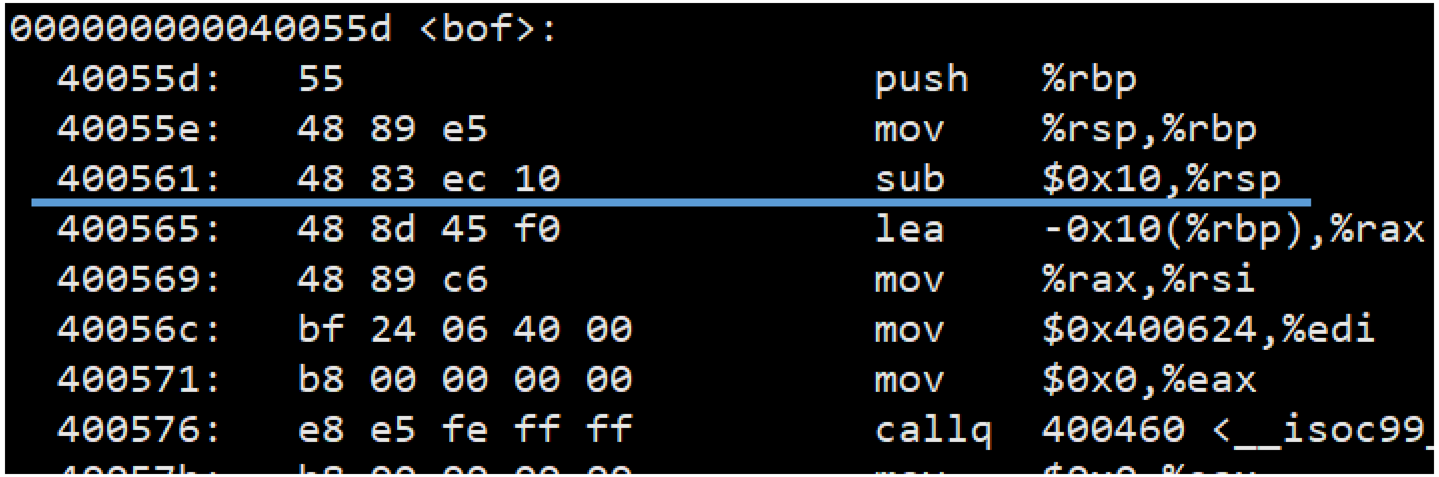
```
rbp            0x7fffffffe570
rsp            0x7fffffffe568
(gdb) x/8xg $rsp-40
0x7fffffffe540: 0x00007fffffffe570      0x0000000000000000
0x7fffffffe550: 0x00000000004005a0      0x0000000000400470
0x7fffffffe560: 0x00007fffffffe650      0x000000000040058b
0x7fffffffe570: 0x0000000000000000      0x00007ffff7a32f45
```



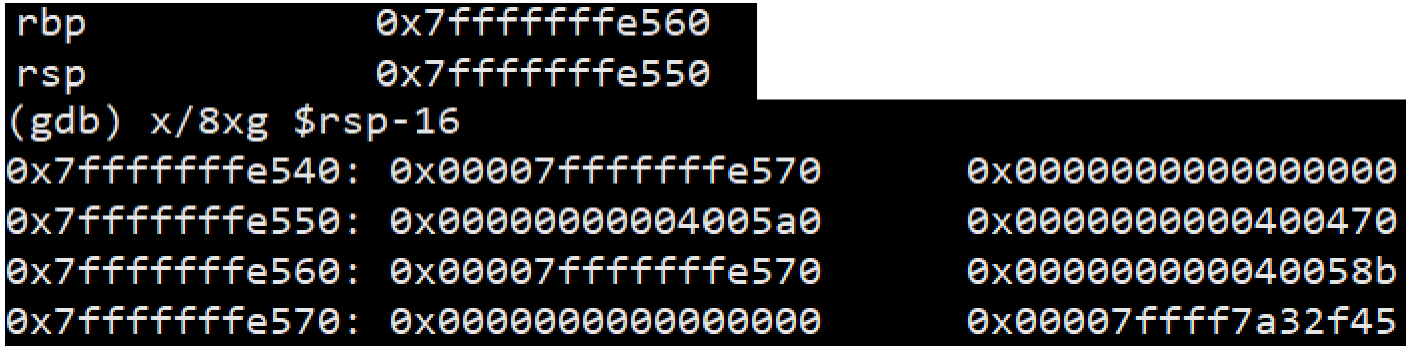
	…e578	0x00…f45
rbp →	…e570	0x00…00
rsp →	…e568	0x00..40058b
	…e560	
	…e558	
	…e550	
	…e548	
	…e540	

Memory layout example (cont.)

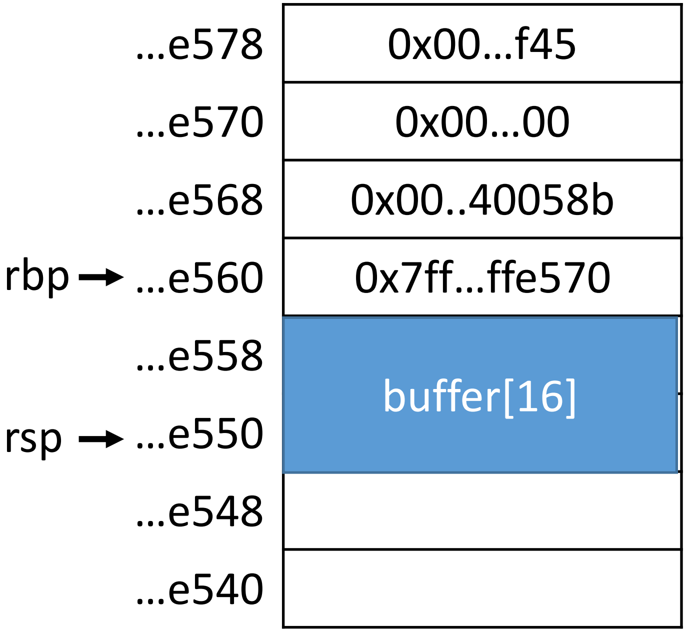
- Before scanf



```
000000000040055d <bof>:
  40055d:   55                      push   %rbp
  40055e:   48 89 e5                mov    %rsp,%rbp
  400561:   48 83 ec 10             sub    $0x10,%rsp
  400565:   48 8d 45 f0             lea    -0x10(%rbp),%rax
  400569:   48 89 c6                mov    %rax,%rsi
  40056c:   bf 24 06 40 00          mov    $0x400624,%edi
  400571:   b8 00 00 00 00          mov    $0x0,%eax
  400576:   e8 e5 fe ff ff          callq  400460 <__isoc99_
  40057b:   b8 00 00 00 00          mov    $0x0,%eax
```

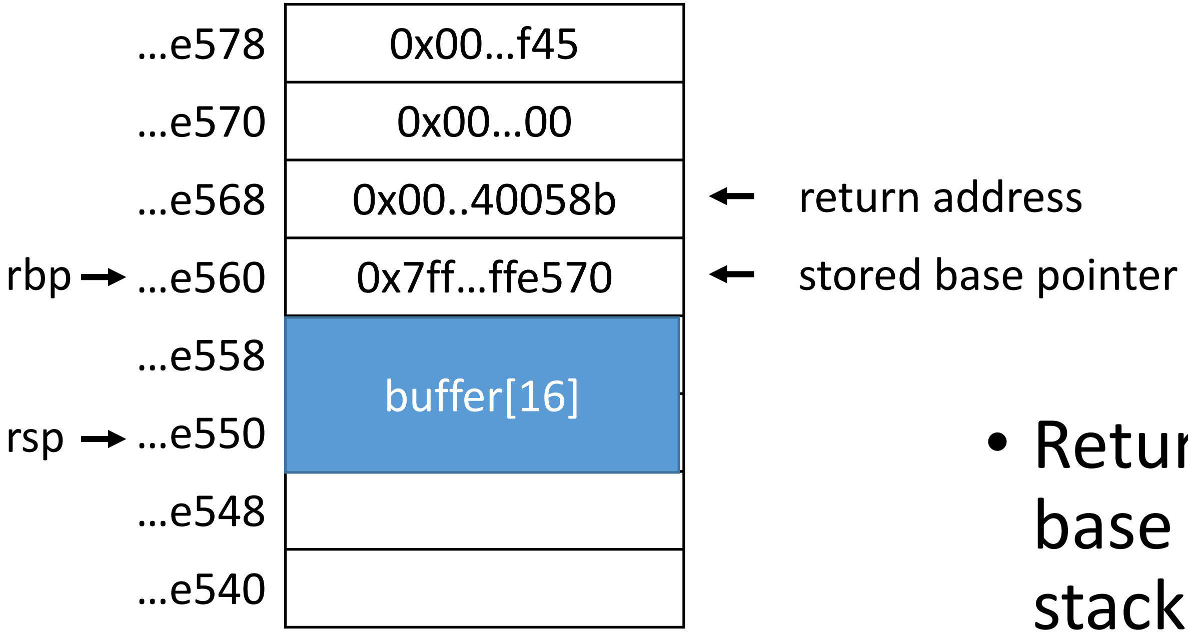


```
rbp            0x7fffffffe560
rsp            0x7fffffffe550
(gdb) x/8xg $rsp-16
0x7fffffffe540: 0x00007fffffffe570      0x0000000000000000
0x7fffffffe550: 0x00000000004005a0      0x0000000000400470
0x7fffffffe560: 0x00007fffffffe570      0x000000000040058b
0x7fffffffe570: 0x0000000000000000      0x00007ffff7a32f45
```



	…e578	0x00…f45
	…e570	0x00…00
	…e568	0x00..40058b
rbp →	…e560	0x7ff…ffe570
	…e558	buffer[16]
rsp →	…e550	
	…e548	
	…e540	

Memory layout example (cont.)

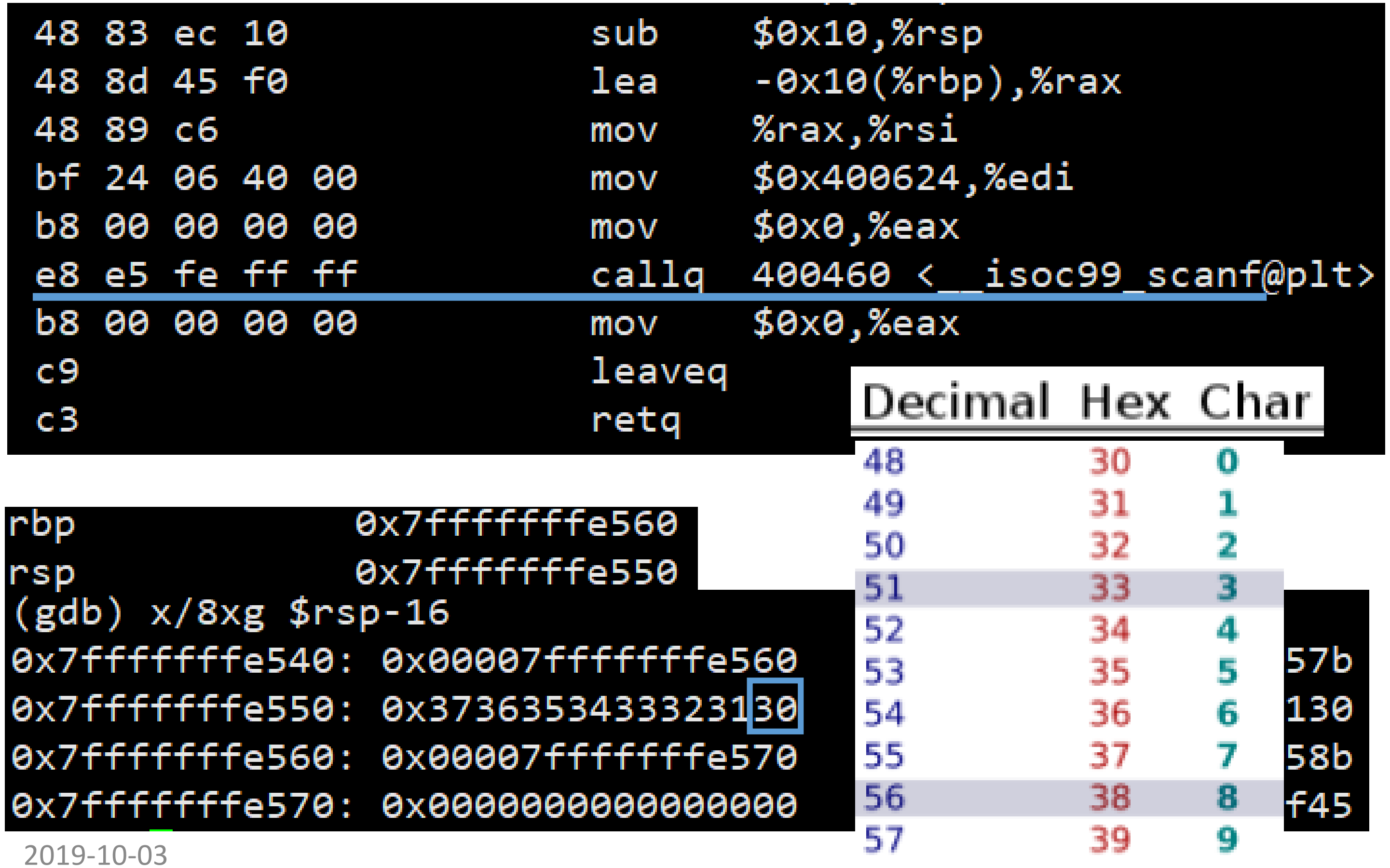


	...e578	0x00...f45	
	...e570	0x00...00	
	...e568	0x00..40058b	← return address
rbp →	...e560	0x7ff...ffe570	← stored base pointer
	...e558	buffer[16]	
rsp →	...e550		
	...e548		
	...e540		

- Return address and previous base pointer are stored in the stack

Memory layout example (cont.)

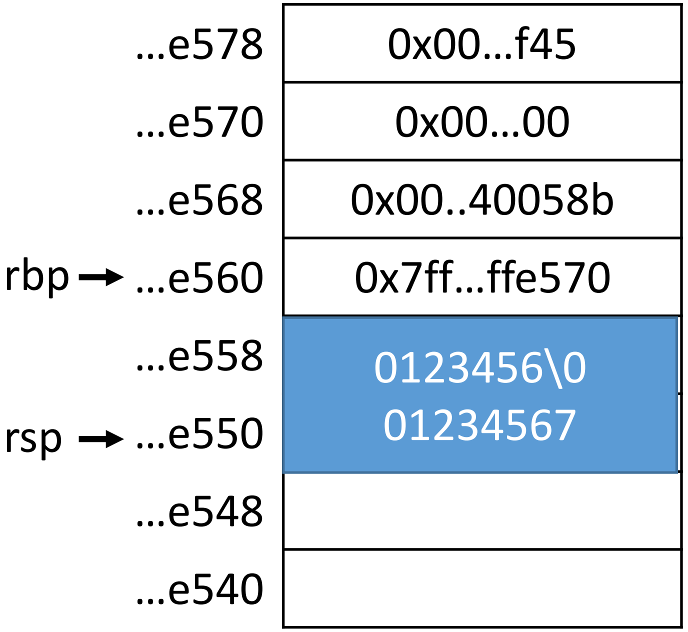
- After scanf (input: 012345670123456)



```
48 83 ec 10                 sub    $0x10,%rsp
48 8d 45 f0                 lea    -0x10(%rbp),%rax
48 89 c6                    mov    %rax,%rsi
bf 24 06 40 00              mov    $0x400624,%edi
b8 00 00 00 00              mov    $0x0,%eax
e8 e5 fe ff ff              callq  400460 <__isoc99_scanf@plt>
b8 00 00 00 00              mov    $0x0,%eax
c9                          leaveq
c3                          retq
```

```
rbp            0x7fffffffe560
rsp            0x7fffffffe550
(gdb) x/8xg $rsp-16
0x7fffffffe540: 0x00007fffffffe560                57b
0x7fffffffe550: 0x3736353433323130                130
0x7fffffffe560: 0x00007fffffffe570                58b
0x7fffffffe570: 0x0000000000000000                f45
```

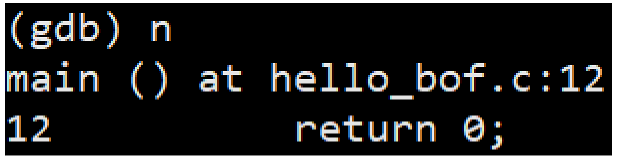
Decimal	Hex	Char
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9



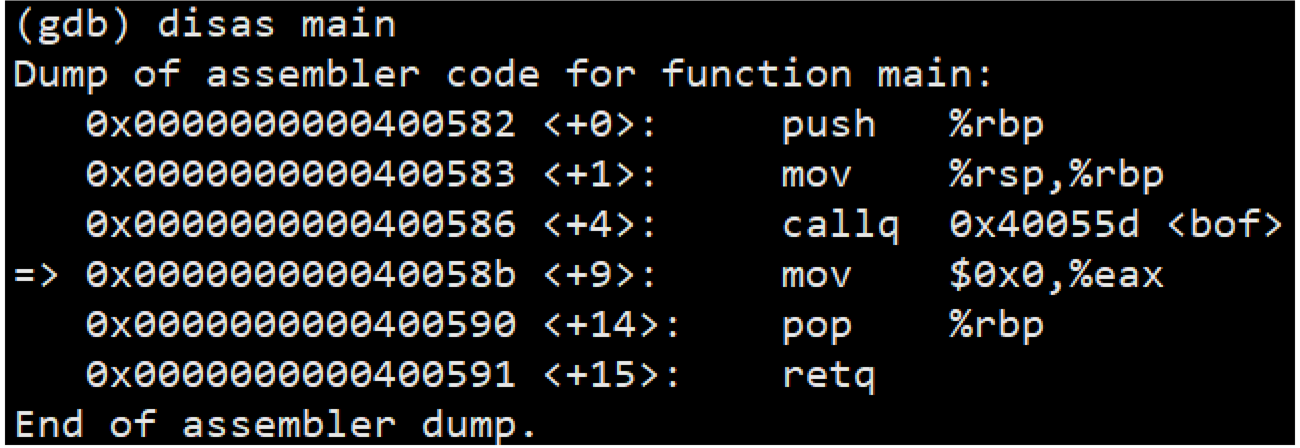
	…e578	0x00…f45
	…e570	0x00…00
	…e568	0x00..40058b
rbp →	…e560	0x7ff…ffe570
	…e558	0123456\0
rsp →	…e550	01234567
	…e548	
	…e540	

Memory layout example (cont.)

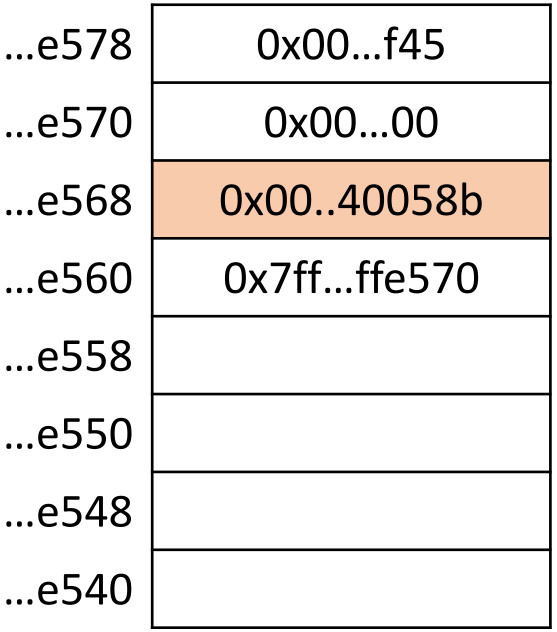
- After return instruction



```
(gdb) n
main () at hello_bof.c:12
12          return 0;
```



```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400582 <+0>:     push   %rbp
   0x0000000000400583 <+1>:     mov    %rsp,%rbp
   0x0000000000400586 <+4>:     callq  0x40055d <bof>
=> 0x000000000040058b <+9>:     mov    $0x0,%eax
   0x0000000000400590 <+14>:    pop    %rbp
   0x0000000000400591 <+15>:    retq
End of assembler dump.
```



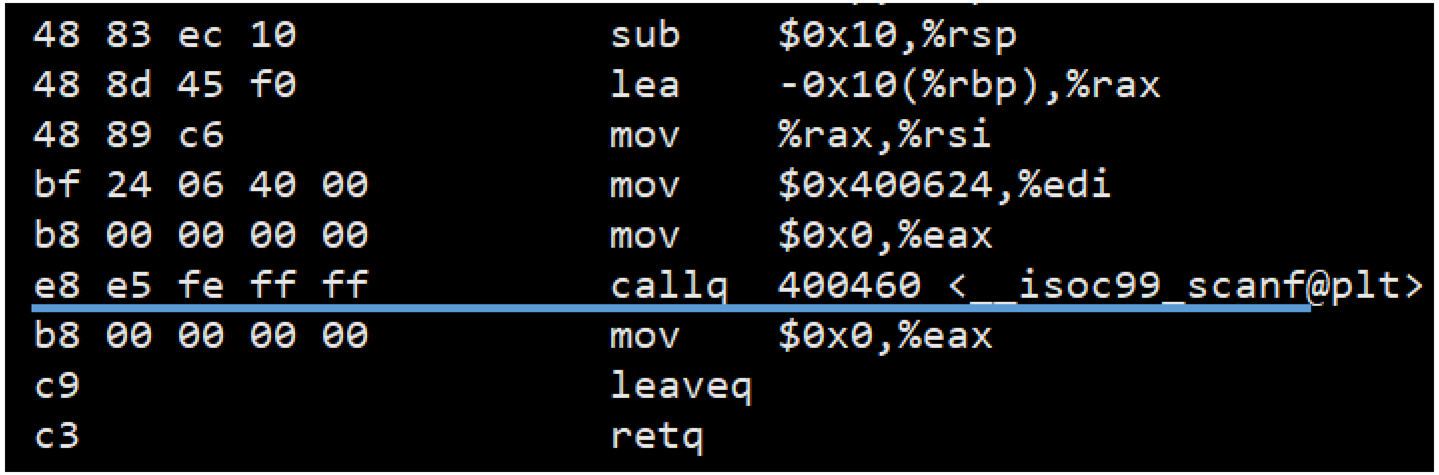
…e578	0x00…f45
…e570	0x00…00
…e568	0x00..40058b
…e560	0x7ff…ffe570
…e558	
…e550	
…e548	
…e540	

What happens
when input exceeds buffer size?

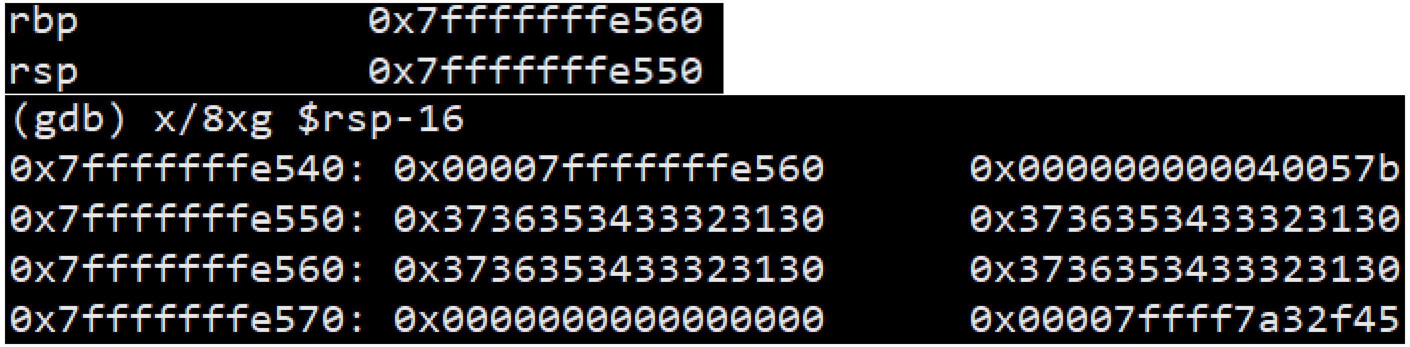
- e.g., $\underbrace{\text{012345670123456701234567}}_{\text{24 bytes! (+ 1 byte for null)}}$\0

Memory layout example with long input

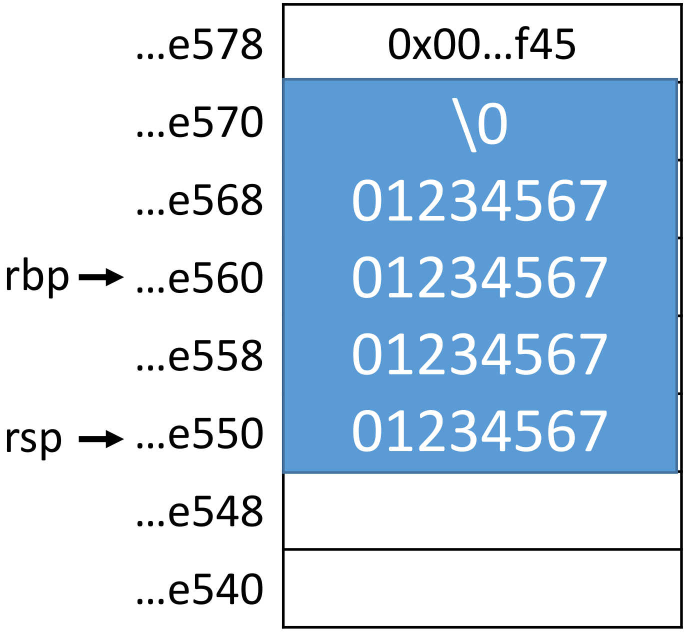
- After scanf (input: 01234567012345670123456701234567)



```
48 83 ec 10                 sub    $0x10,%rsp
48 8d 45 f0                 lea    -0x10(%rbp),%rax
48 89 c6                    mov    %rax,%rsi
bf 24 06 40 00              mov    $0x400624,%edi
b8 00 00 00 00              mov    $0x0,%eax
e8 e5 fe ff ff              callq  400460 <__isoc99_scanf@plt>
b8 00 00 00 00              mov    $0x0,%eax
c9                          leaveq
c3                          retq
```

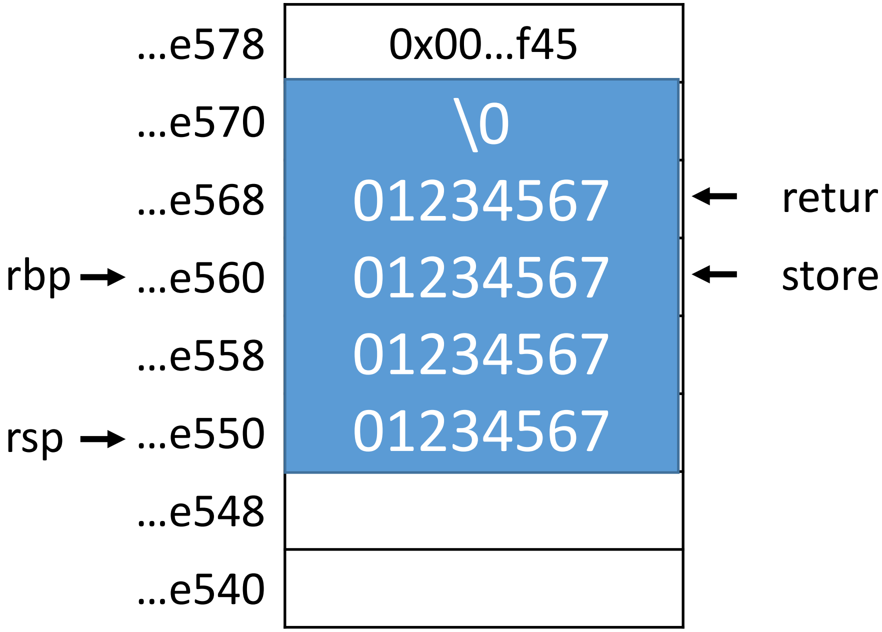


```
rbp            0x7fffffffe560
rsp            0x7fffffffe550
(gdb) x/8xg $rsp-16
0x7fffffffe540: 0x00007fffffffe560      0x000000000040057b
0x7fffffffe550: 0x3736353433323130      0x3736353433323130
0x7fffffffe560: 0x3736353433323130      0x3736353433323130
0x7fffffffe570: 0x0000000000000000      0x00007ffff7a32f45
```



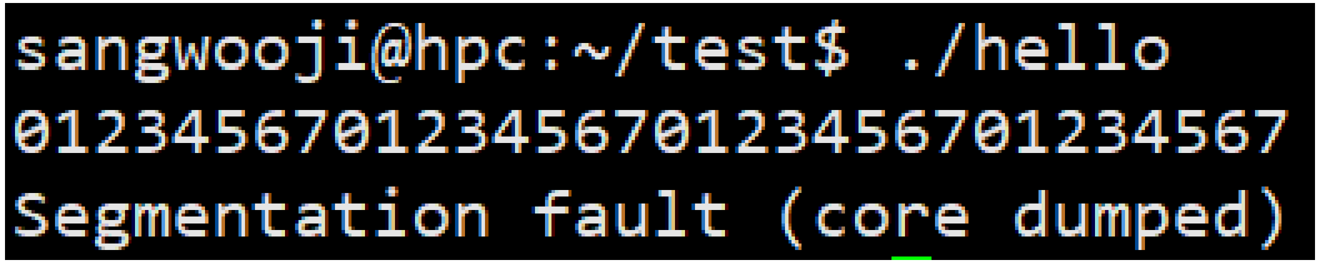
...e578 0x00...f45
...e570 \0
...e568 01234567
rbp → ...e560 01234567
...e558 01234567
rsp → ...e550 01234567
...e548
...e540

Return address is corrupted



	...e578	0x00...f45	
	...e570	\0	
	...e568	01234567	← return address
rbp →	...e560	01234567	← stored base pointer
	...e558	01234567	
rsp →	...e550	01234567	
	...e548		
	...e540		

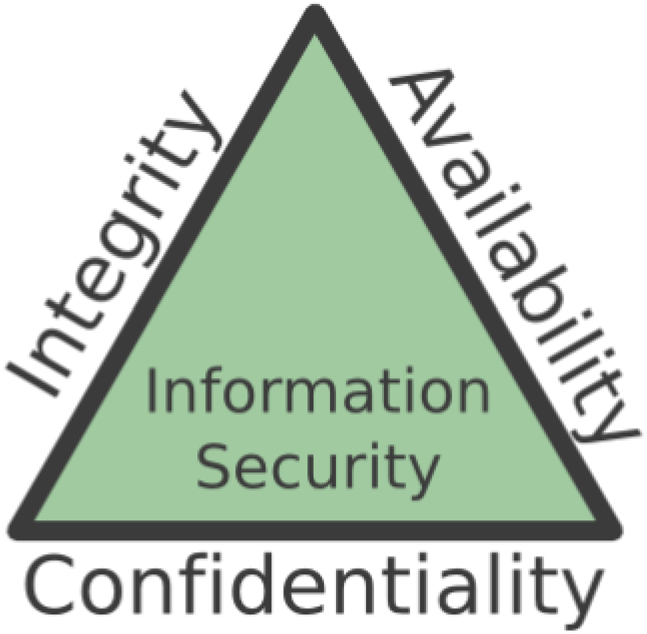
- return instruction will jump to 0x3736353433323130!



```
sangwooji@hpc:~/test$ ./hello
01234567012345670123456701234567
Segmentation fault (core dumped)
```

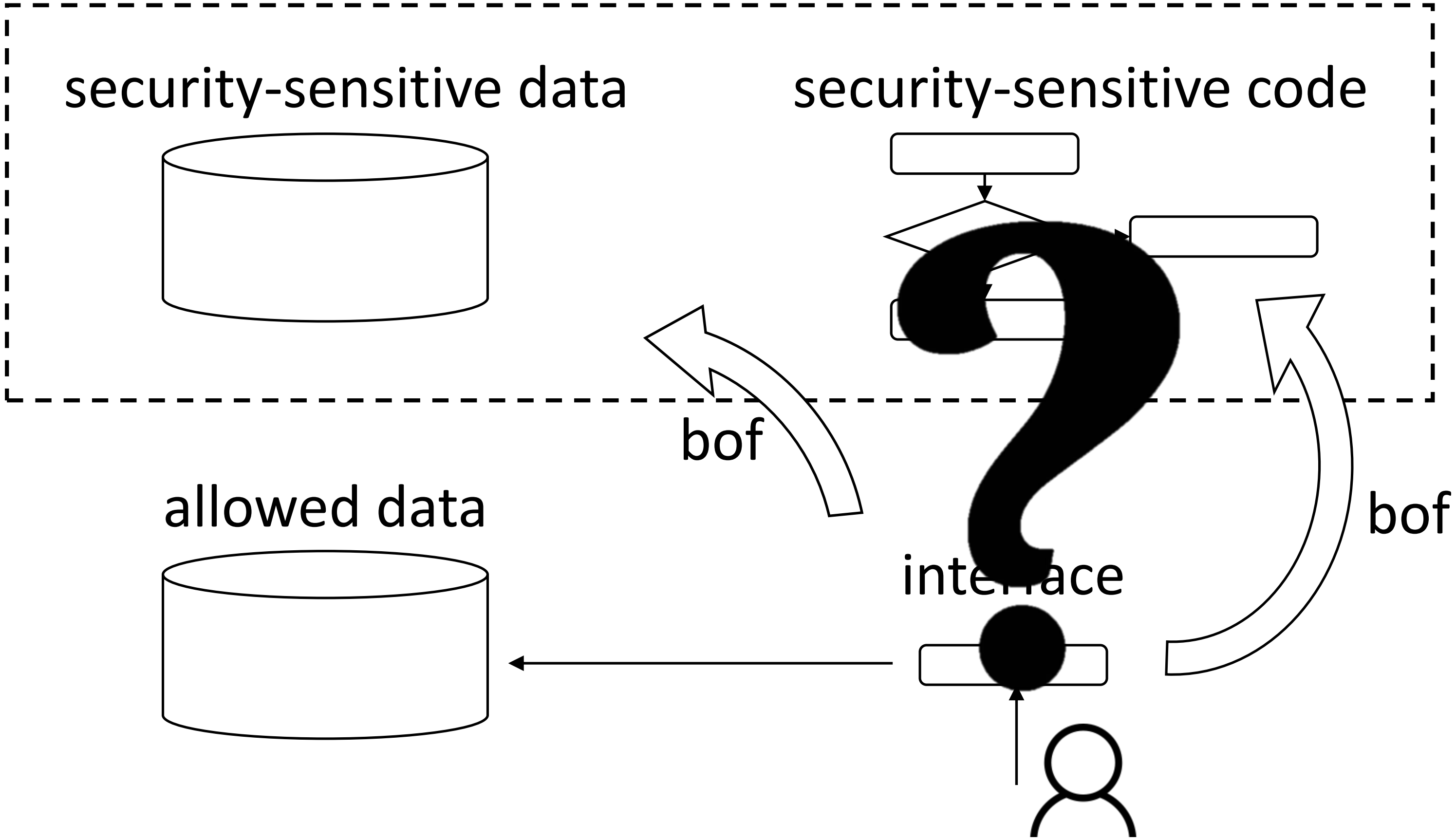
Why this becomes severe problem?

- User can tweak control flow (execution path) of a program.



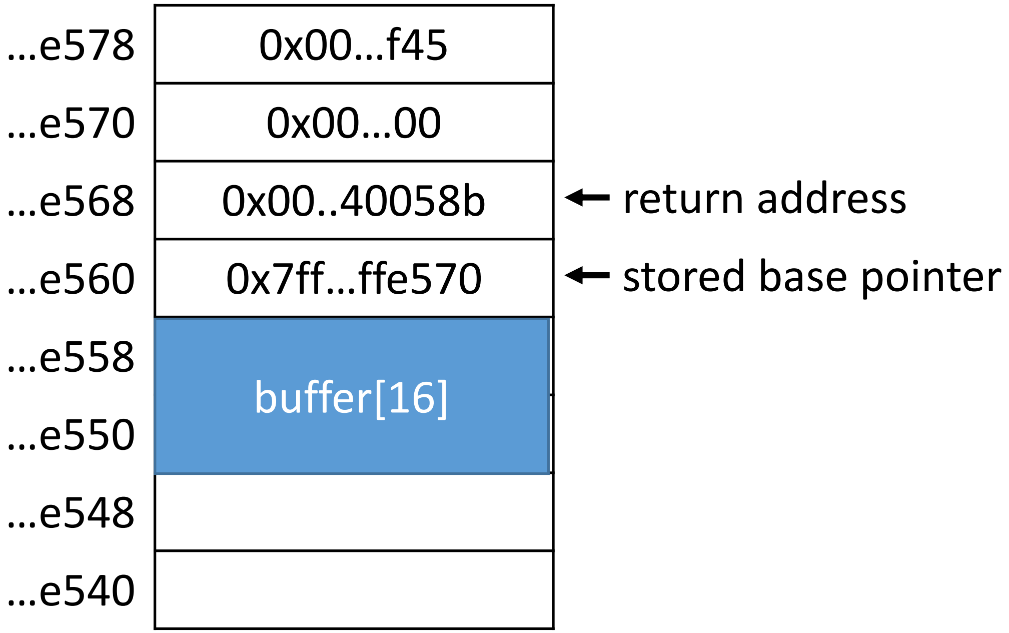
Interface as a security mechanism

- Interface does not allow a user to access security-sensitive part

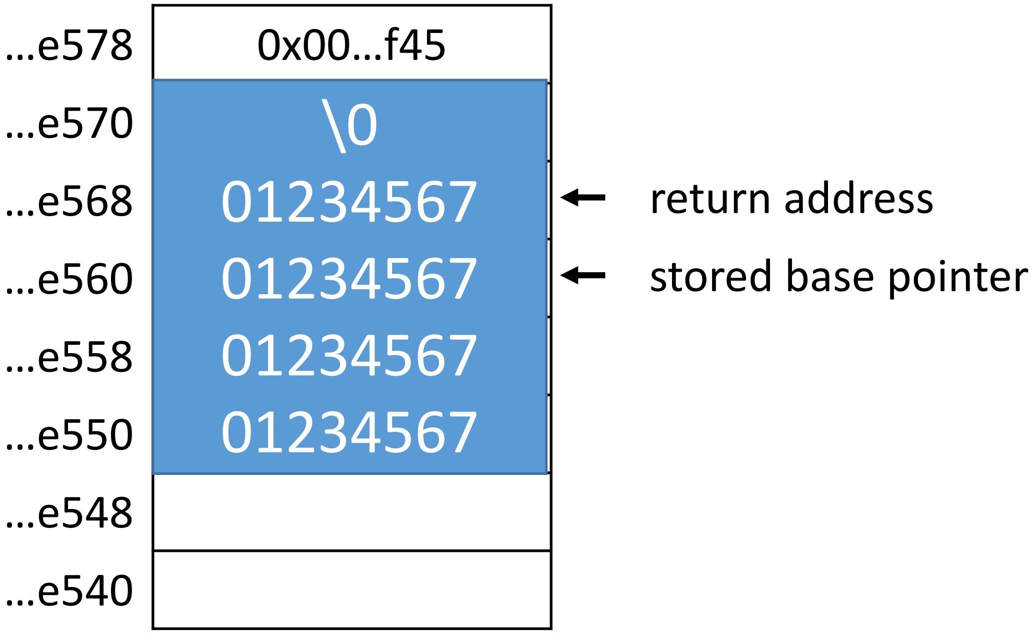


What we learn last week

- Stacks can be corrupted by a user input



…e578	0x00…f45	
…e570	0x00…00	
…e568	0x00..40058b	← return address
…e560	0x7ff…ffe570	← stored base pointer
…e558	buffer[16]	
…e550		
…e548		
…e540		

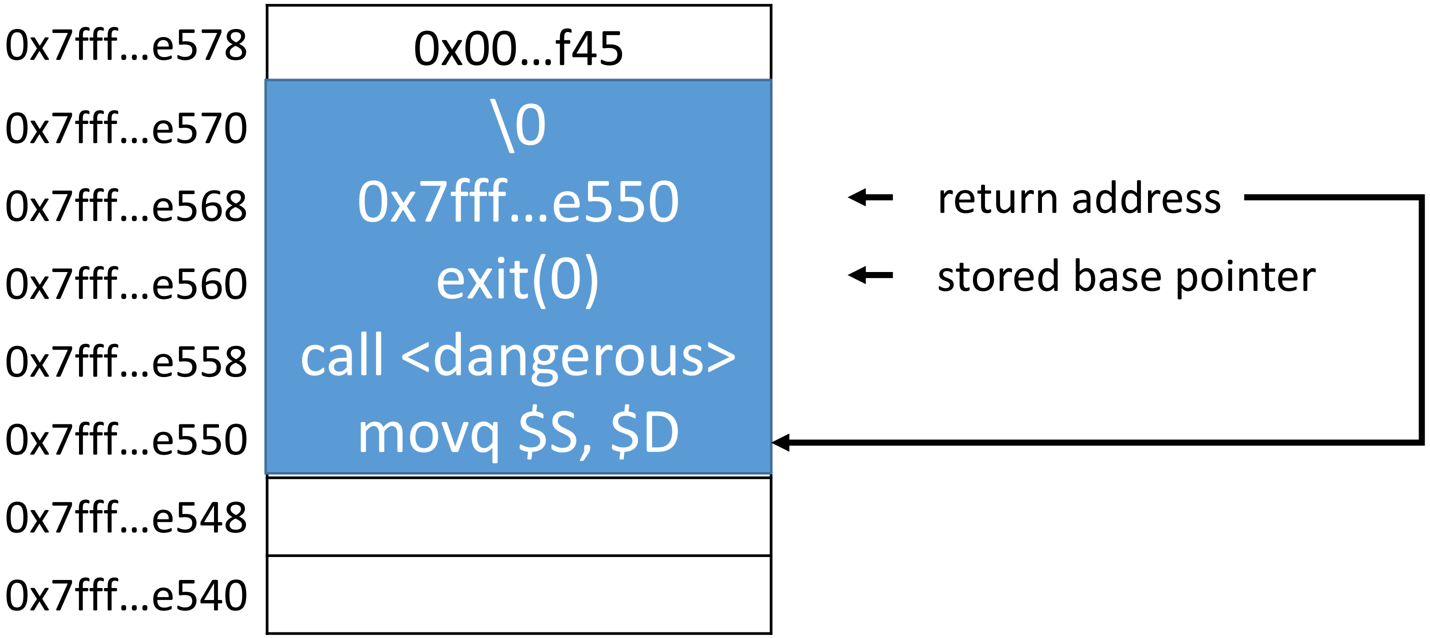


…e578	0x00…f45	
…e570	\0	
…e568	01234567	← return address
…e560	01234567	← stored base pointer
…e558	01234567	
…e550	01234567	
…e548		
…e540		

Code injection and defenses

Code injection attack

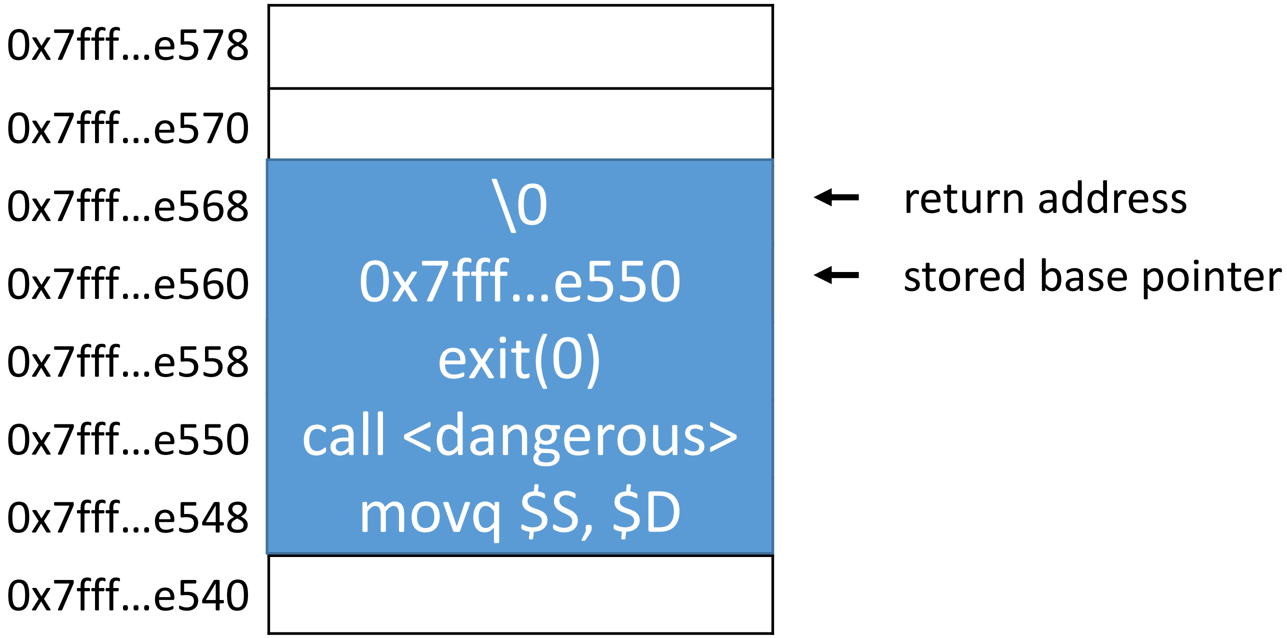
- Insert an arbitrary instruction sequence into the stack and return to the injected code.



0x7fff…e578	0x00…f45
0x7fff…e570	\0
0x7fff…e568	0x7fff…e550
0x7fff…e560	exit(0)
0x7fff…e558	call <dangerous>
0x7fff…e550	movq $S, $D
0x7fff…e548	
0x7fff…e540	

Defenses: stack canary

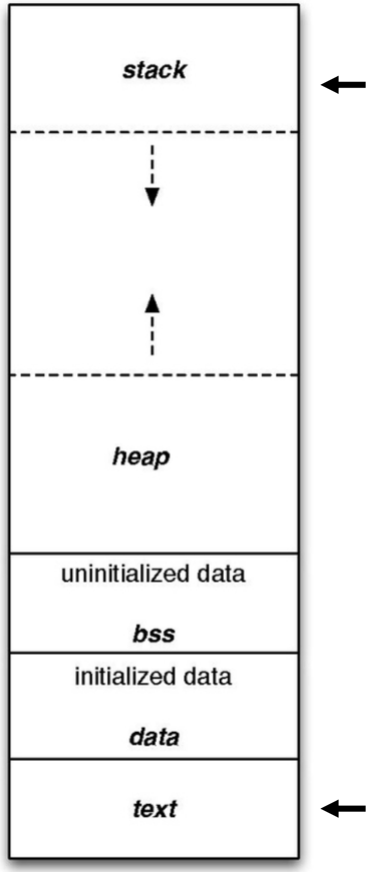
- Detect buffer overflow attack by observing value of the stack canary



0x7fff…e578		
0x7fff…e570		
0x7fff…e568	\0	← return address
0x7fff…e560	0x7fff…e550	← stored base pointer
0x7fff…e558	exit(0)	
0x7fff…e550	call <dangerous>	
0x7fff…e548	movq $S, $D	
0x7fff…e540		

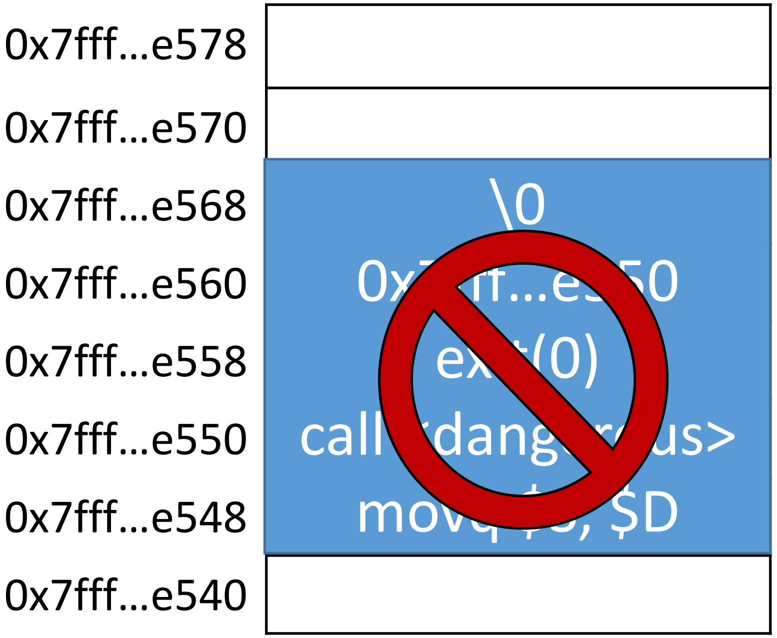
Defenses: write XOR execute

- Injected instructions are not executable anymore



writable
not executable

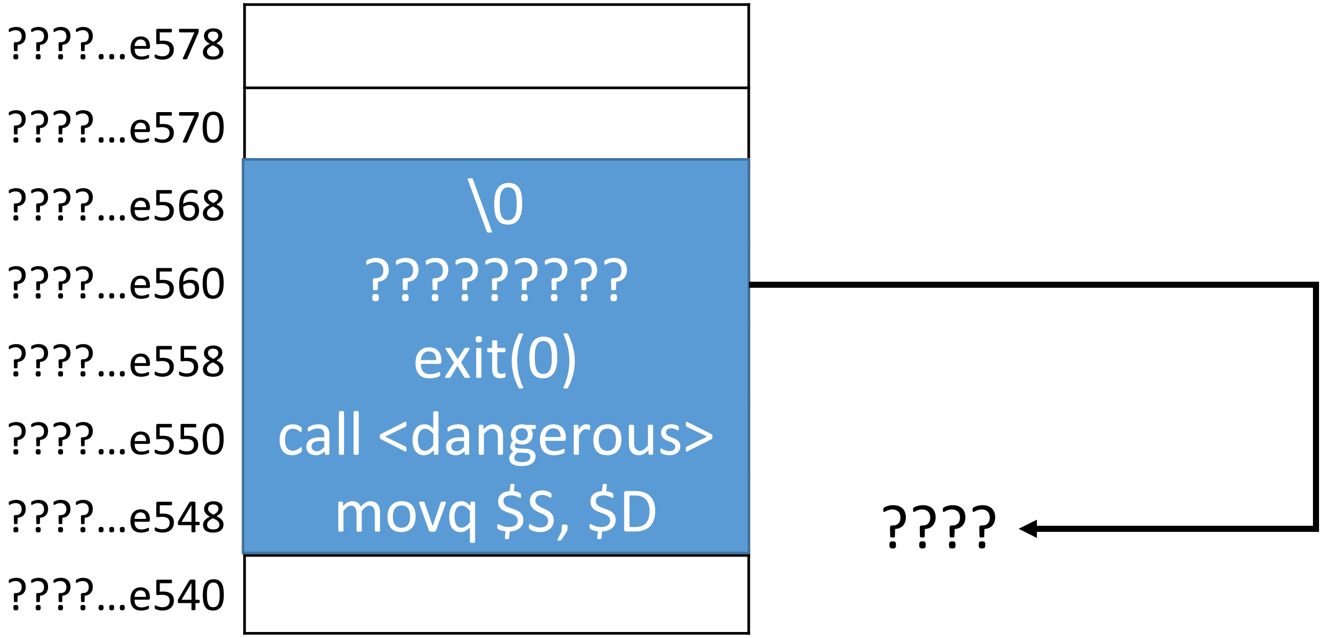
executable
not writable



0x7fff…e578	
0x7fff…e570	
0x7fff…e568	\0
0x7fff…e560	0x7fff…e550
0x7fff…e558	exit(0)
0x7fff…e550	call <dangerous>
0x7fff…e548	movq $3, $D
0x7fff…e540	

Defenses: address space layout randomization

- Randomize address of the stack
- Cannot jump to the address where we inject the instructions



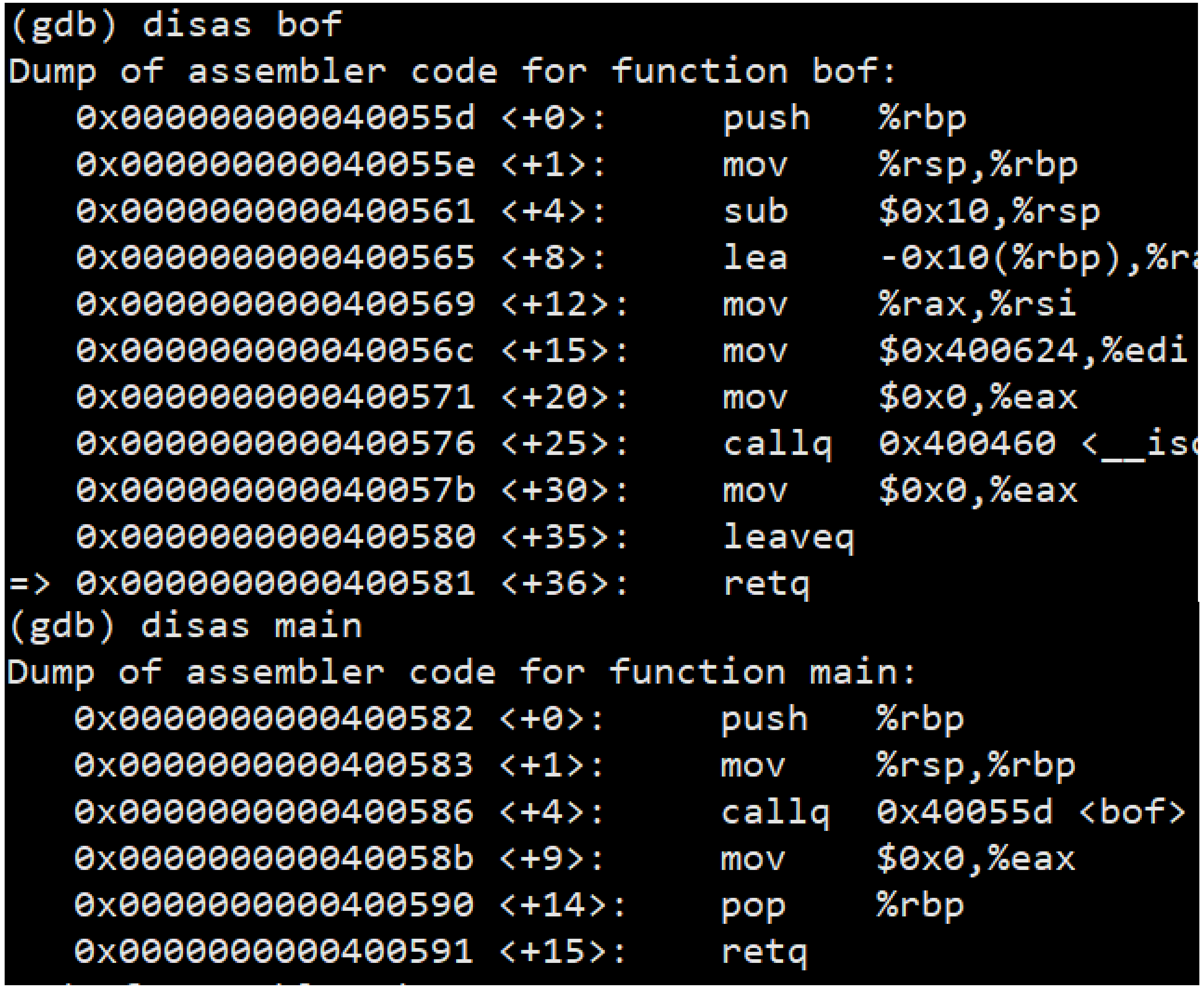
Return oriented programming (ROP)

Recall the RET instruction

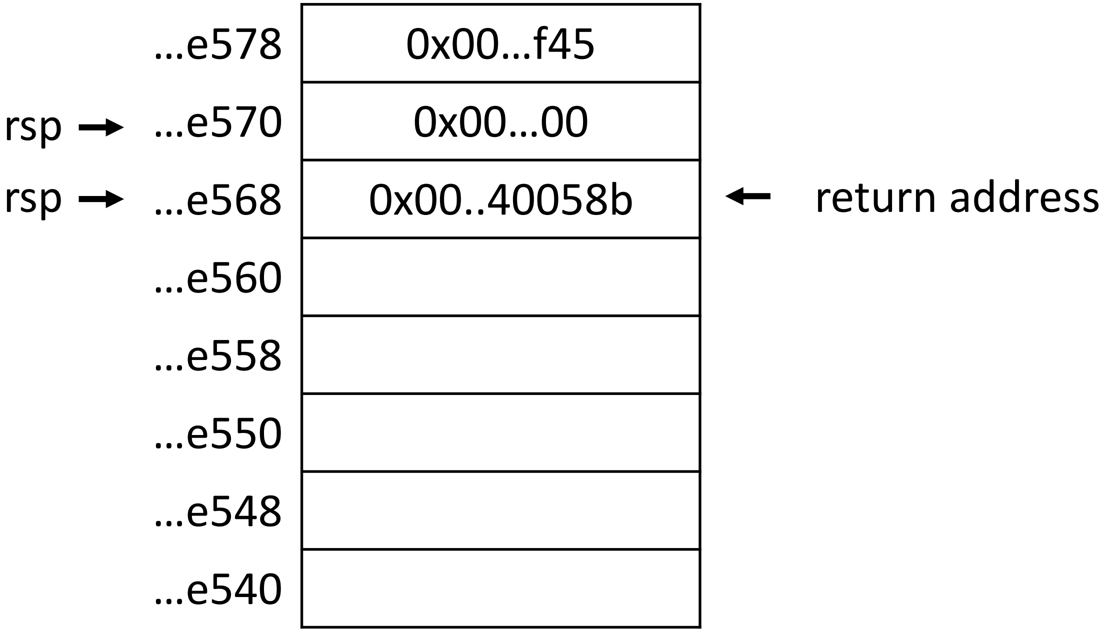
- When executing a near return, **the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register** and begins program execution at the new instruction pointer. The CS register is unchanged.
- When executing a far return, **the processor pops the return instruction pointer from the top of the stack into the EIP register**, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

https://www.felixcloutier.com/x86/ret

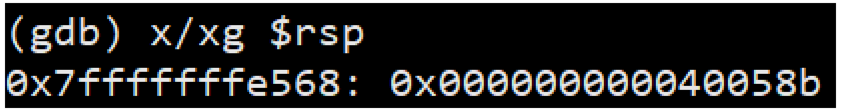
Recall the RET instruction



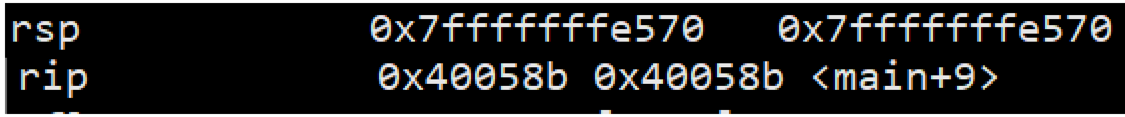
```
(gdb) disas bof
Dump of assembler code for function bof:
   0x000000000040055d <+0>:     push   %rbp
   0x000000000040055e <+1>:     mov    %rsp,%rbp
   0x0000000000400561 <+4>:     sub    $0x10,%rsp
   0x0000000000400565 <+8>:     lea    -0x10(%rbp),%ra
   0x0000000000400569 <+12>:    mov    %rax,%rsi
   0x000000000040056c <+15>:    mov    $0x400624,%edi
   0x0000000000400571 <+20>:    mov    $0x0,%eax
   0x0000000000400576 <+25>:    callq  0x400460 <__is
   0x000000000040057b <+30>:    mov    $0x0,%eax
   0x0000000000400580 <+35>:    leaveq
=> 0x0000000000400581 <+36>:    retq
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400582 <+0>:     push   %rbp
   0x0000000000400583 <+1>:     mov    %rsp,%rbp
   0x0000000000400586 <+4>:     callq  0x40055d <bof>
   0x000000000040058b <+9>:     mov    $0x0,%eax
   0x0000000000400590 <+14>:    pop    %rbp
   0x0000000000400591 <+15>:    retq
```



	…e578	0x00…f45	
rsp →	…e570	0x00…00	
rsp →	…e568	0x00..40058b	← return address
	…e560		
	…e558		
	…e550		
	…e548		
	…e540		



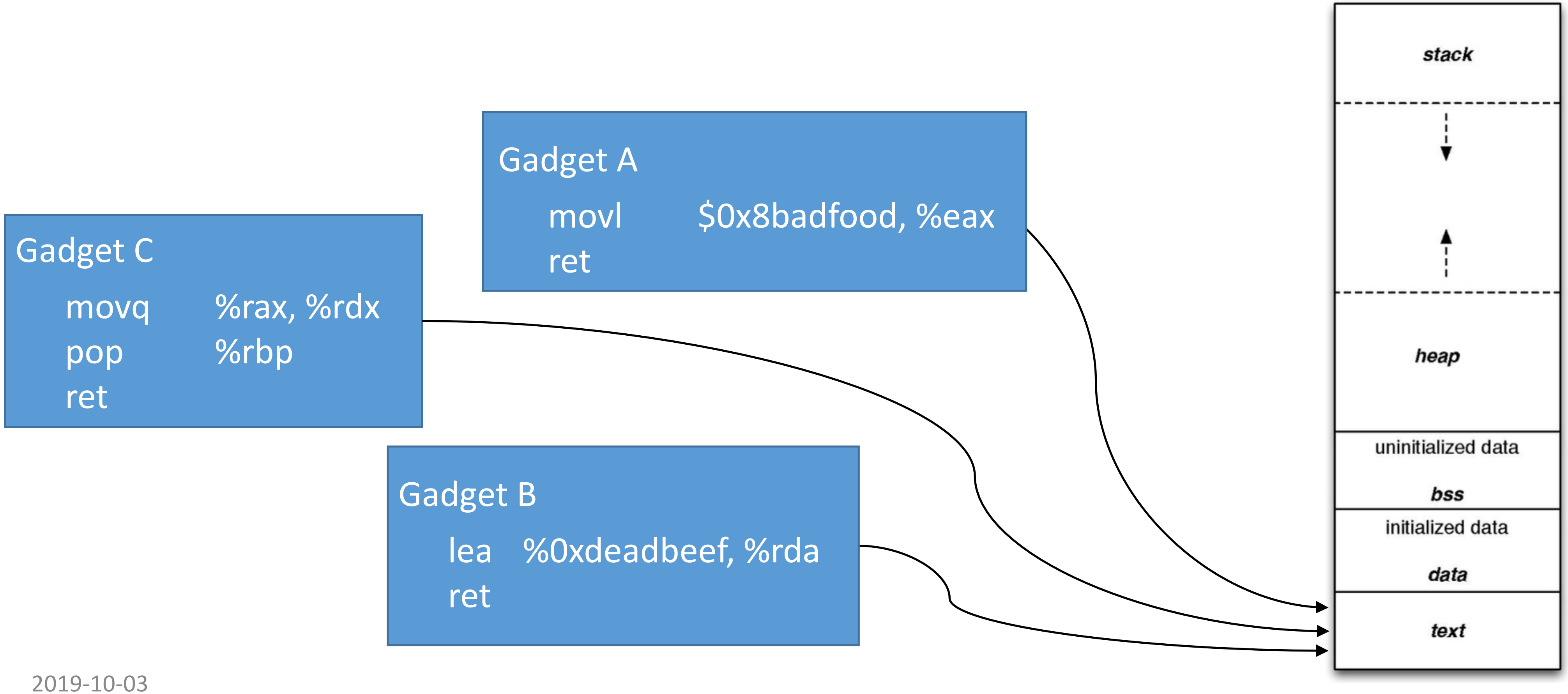
```
(gdb) x/xg $rsp
0x7fffffffe568: 0x000000000040058b
```



```
rsp            0x7fffffffe570   0x7fffffffe570
rip            0x40058b 0x40058b <main+9>
```

ROP attack: gadget

- A small code (code snippet) which ends with RET instruction



Gadget A

```
movl      $0x8badfood, %eax
ret
```

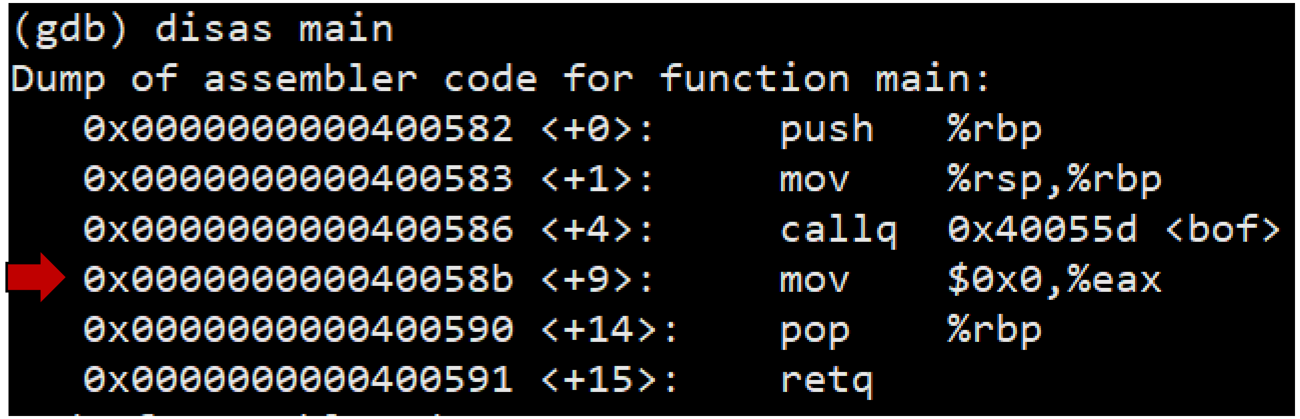
Gadget C

```
movq      %rax, %rdx
pop       %rbp
ret
```

Gadget B

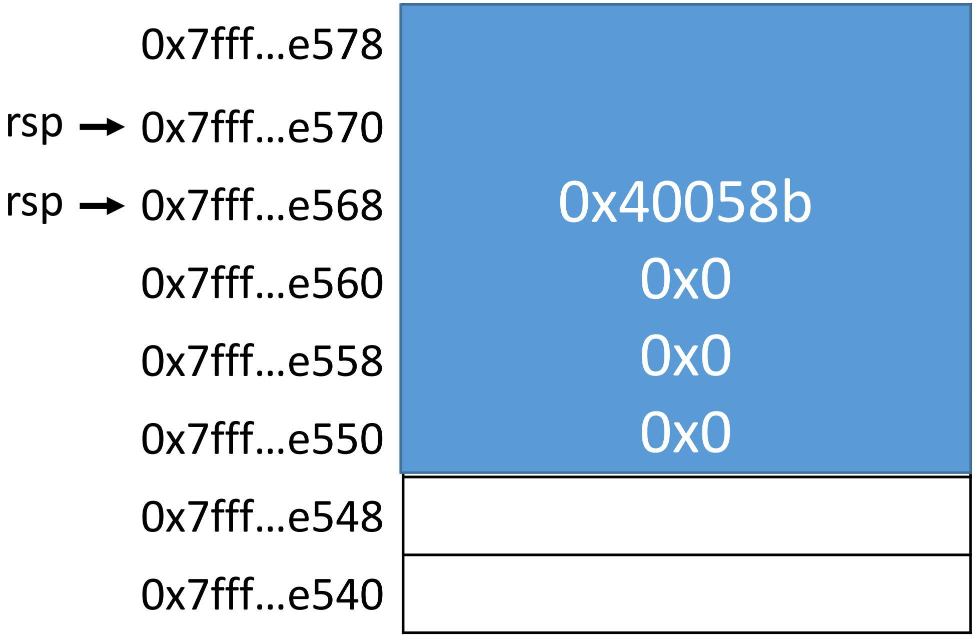
```
lea   %0xdeadbeef, %rda
ret
```

ROP attack: ordinary return

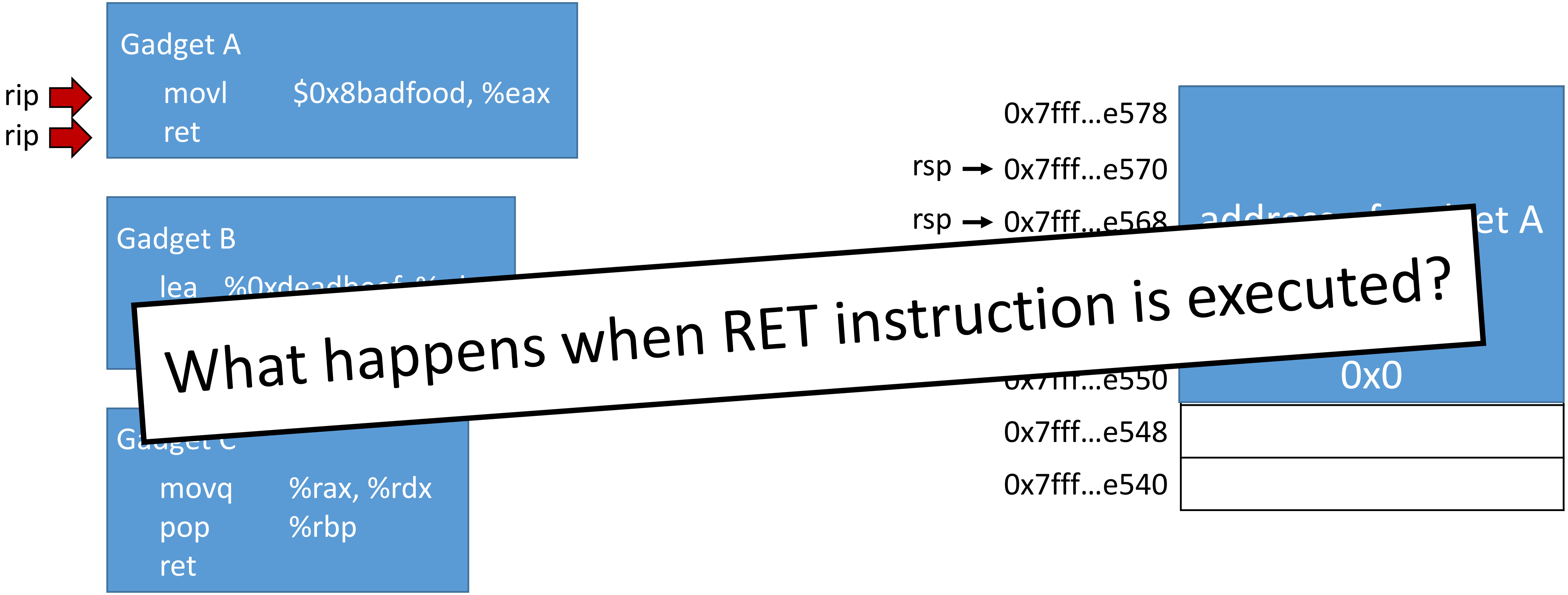


```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400582 <+0>:     push   %rbp
   0x0000000000400583 <+1>:     mov    %rsp,%rbp
   0x0000000000400586 <+4>:     callq  0x40055d <bof>
   0x000000000040058b <+9>:     mov    $0x0,%eax
   0x0000000000400590 <+14>:    pop    %rbp
   0x0000000000400591 <+15>:    retq
```

rip

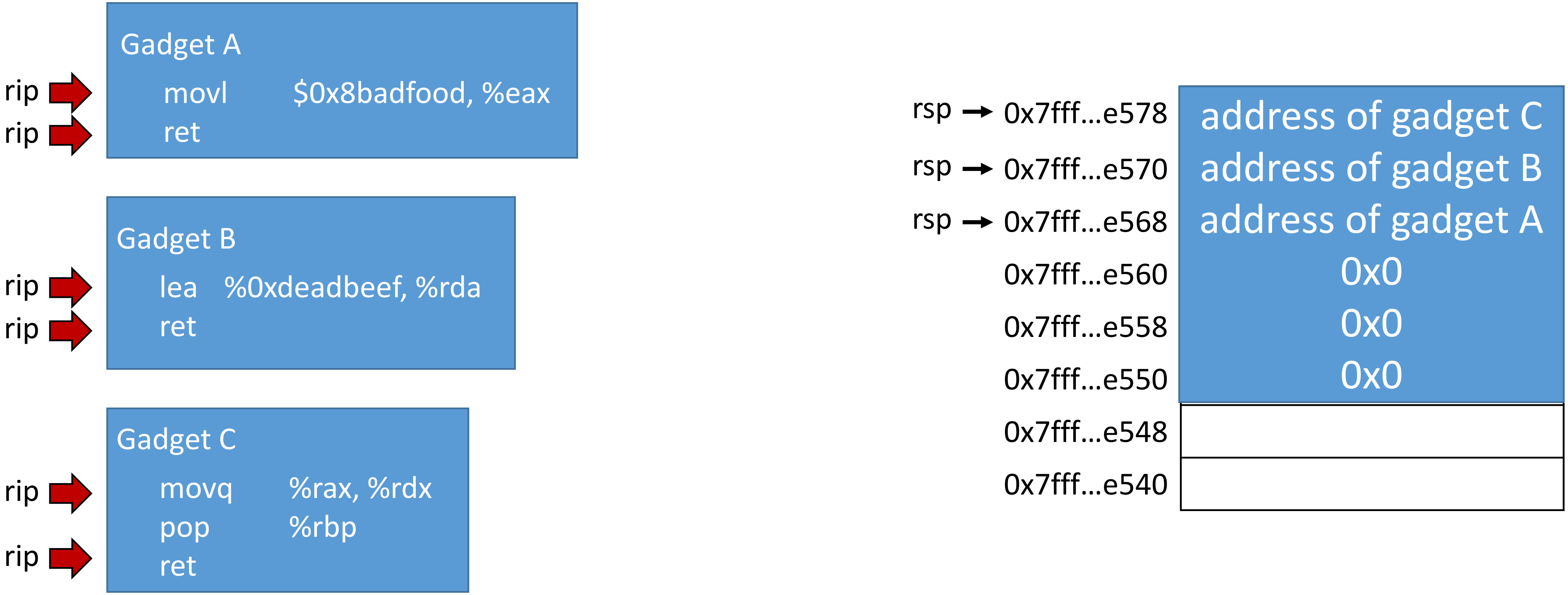


ROP attack: return to a gadget



What happens when RET instruction is executed?

ROP attack: gadget chaining



```
Gadget A
    movl     $0x8badfood, %eax
    ret
```

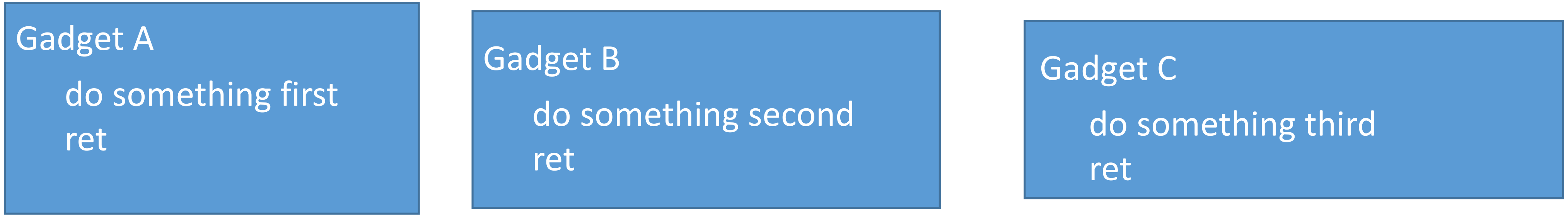
```
Gadget B
    lea   %0xdeadbeef, %rda
    ret
```

```
Gadget C
    movq     %rax, %rdx
    pop      %rbp
    ret
```

rsp →	0x7fff…e578	address of gadget C
rsp →	0x7fff…e570	address of gadget B
rsp →	0x7fff…e568	address of gadget A
	0x7fff…e560	0x0
	0x7fff…e558	0x0
	0x7fff…e550	0x0
	0x7fff…e548	
	0x7fff…e540	

ROP attack: gadget chaining (cont.)

- We can construct arbitrary instruction sequences with gadgets



Gadget A

```
    do something first
    ret
```

Gadget B

```
    do something second
    ret
```

Gadget C

```
    do something third
    ret
```

ROP can bypass two defenses

- Write XOR Execute
  - Code region (where gadgets exist) is still executable
- Address space layout randomization (ASLR)
  - Not require jump into the stack

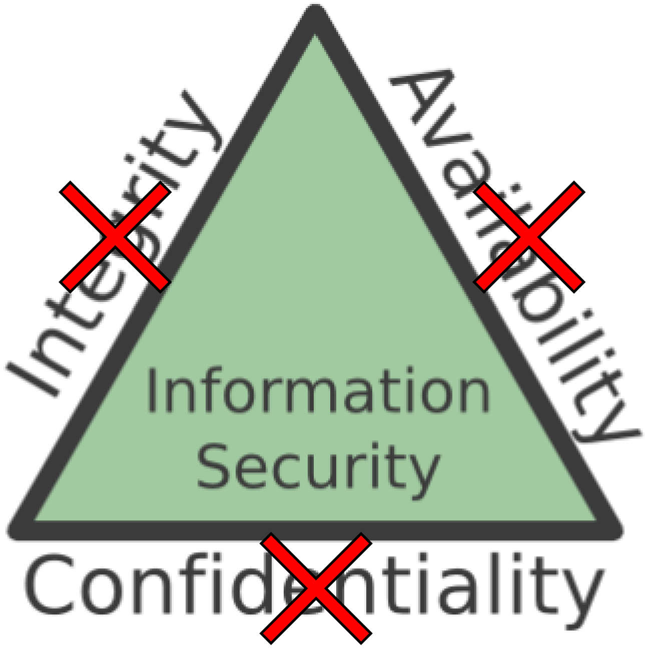
Fundamental defense

- Use safe functions

```
scanf (“%15s”, buf);
fgets (buf, 15, stream);
```

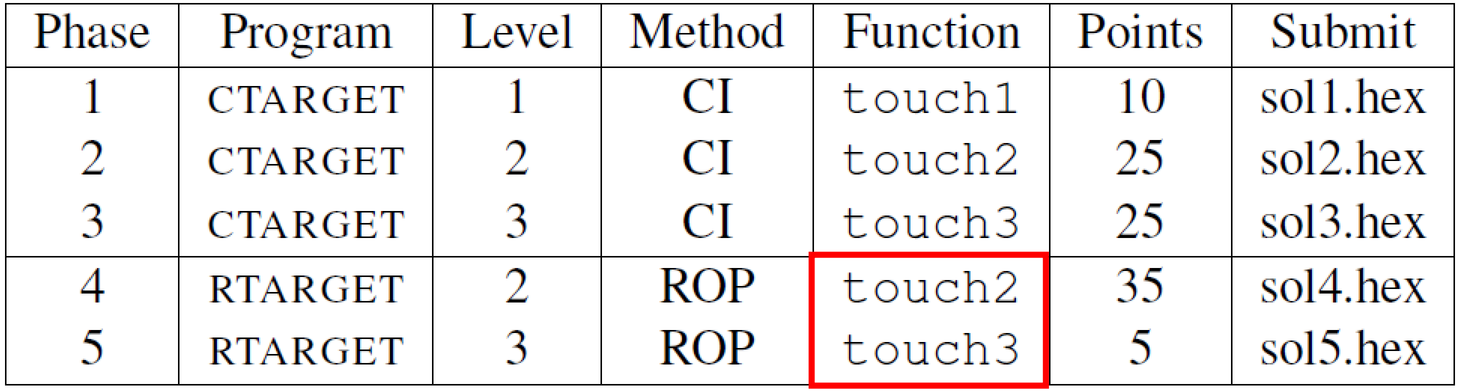
Today’s buffer overflow

- Execute a desired instruction sequence



Assignment: code injection and ROP

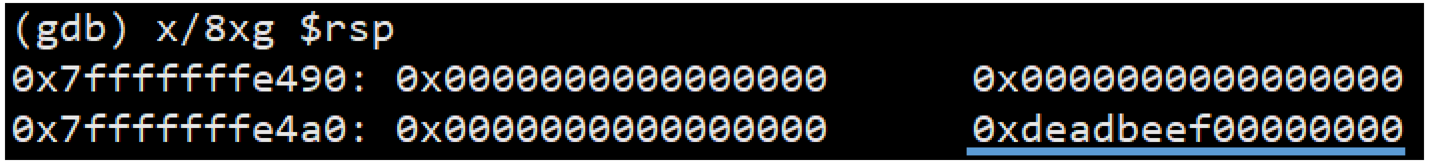
- Three code injection problems
- Two ROP problems
- Due ~10/16 23:59
- **PLEASE READ THE DOCUMENT BEFORE YOU START THE ASSIGNMENT**



Phase	Program	Level	Method	Function	Points	Submit
1	CTARGET	1	CI	`touch1`	10	sol1.hex
2	CTARGET	2	CI	`touch2`	25	sol2.hex
3	CTARGET	3	CI	`touch3`	25	sol3.hex
4	RTARGET	2	ROP	`touch2`	35	sol4.hex
5	RTARGET	3	ROP	`touch3`	5	sol5.hex

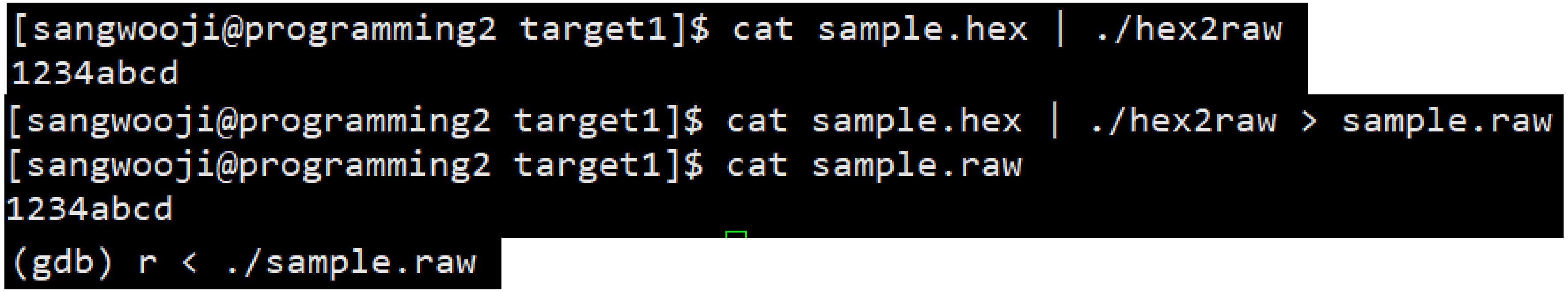
Practice

- Download hello_bof.tar and tar –xvf hello_bof.tar
- Determine the buffer size
- Jump to 0xdeadbeef00000000 using hex2raw



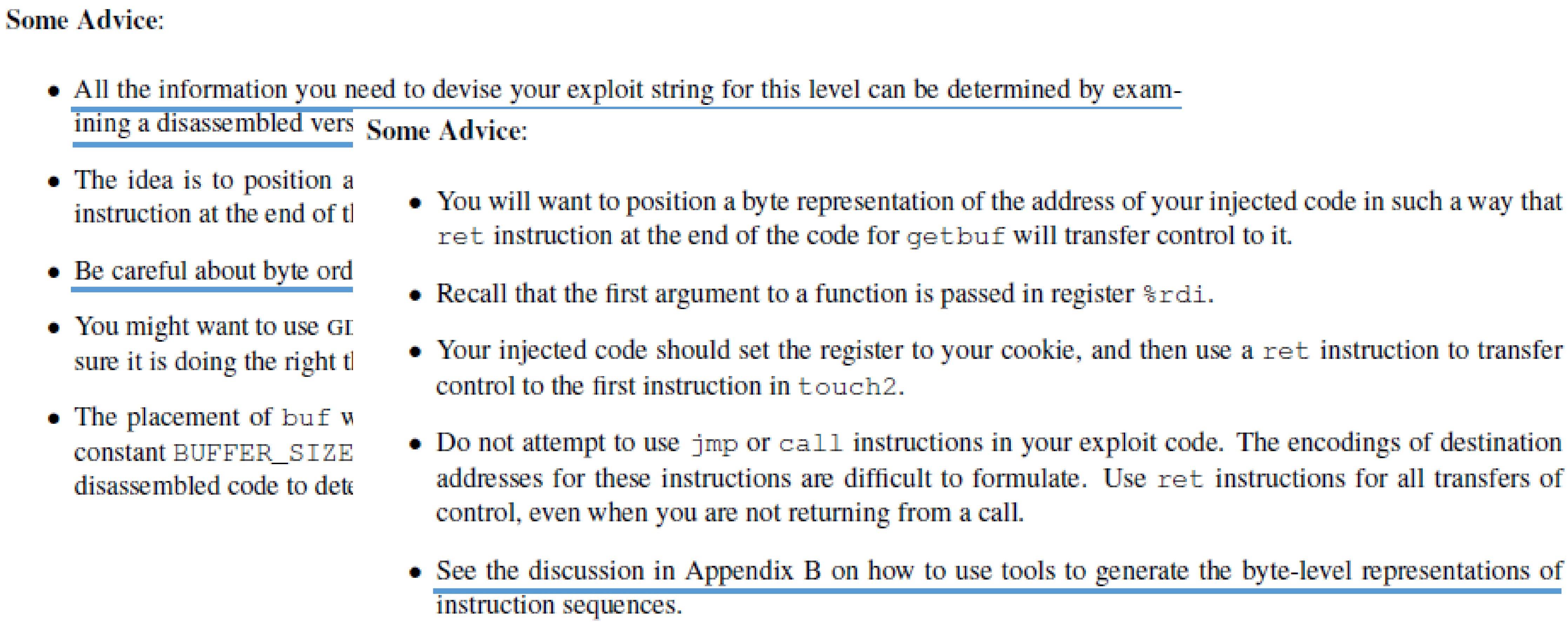
```
(gdb) x/8xg $rsp
0x7fffffffe490: 0x0000000000000000      0x0000000000000000
0x7fffffffe4a0: 0x0000000000000000      0xdeadbeef00000000
```

- Useful commands



```
[sangwooji@programming2 target1]$ cat sample.hex | ./hex2raw
1234abcd
[sangwooji@programming2 target1]$ cat sample.hex | ./hex2raw > sample.raw
[sangwooji@programming2 target1]$ cat sample.raw
1234abcd
(gdb) r < ./sample.raw
```

All you need is in README.pdf



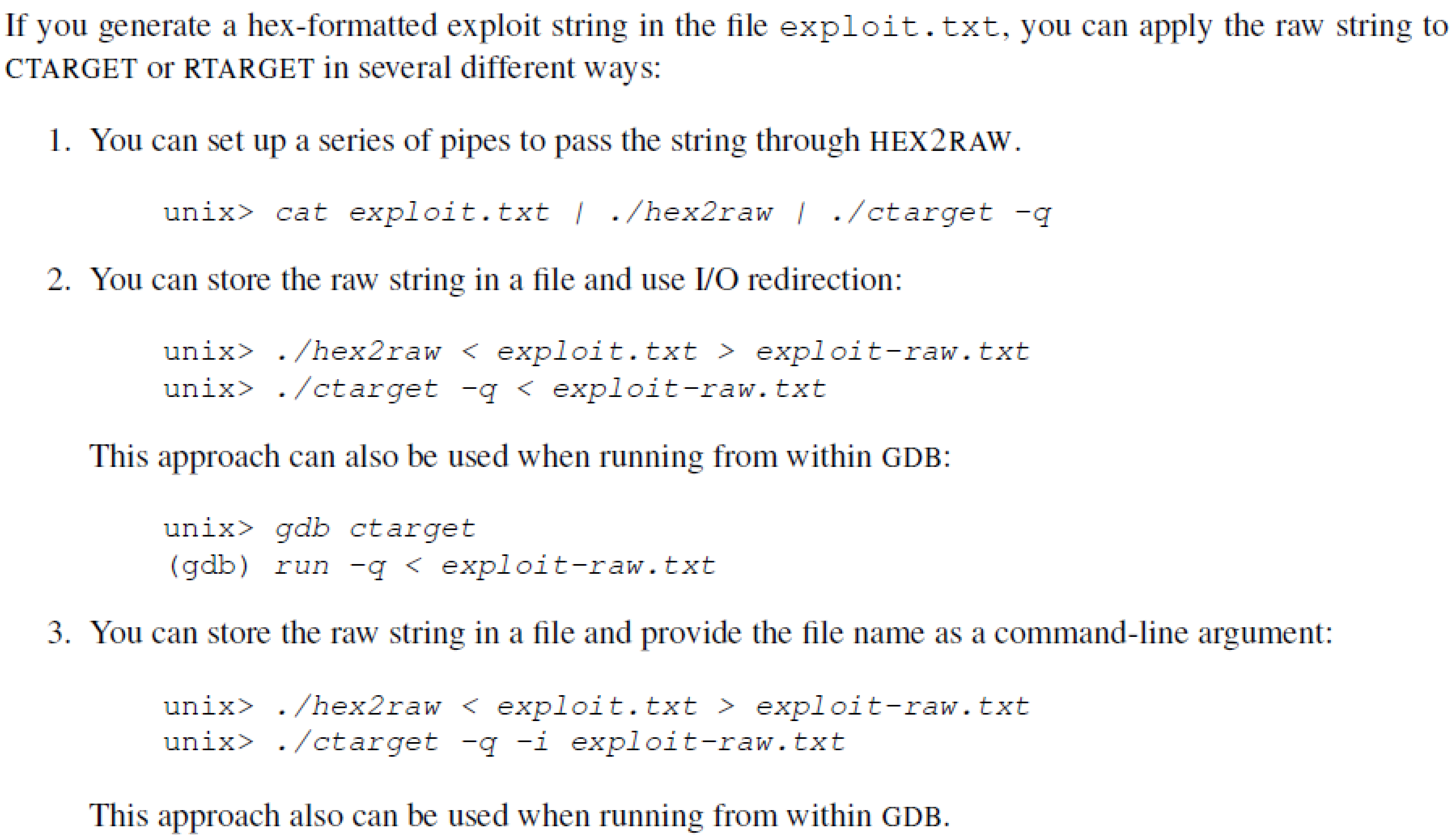
Some Advice:

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled vers
- The idea is to position a instruction at the end of th
- Be careful about byte ord
- You might want to use GD sure it is doing the right th
- The placement of buf w constant BUFFER_SIZE disassembled code to dete

Some Advice:

- You will want to position a byte representation of the address of your injected code in such a way that `ret` instruction at the end of the code for `getbuf` will transfer control to it.
- Recall that the first argument to a function is passed in register `%rdi`.
- Your injected code should set the register to your cookie, and then use a `ret` instruction to transfer control to the first instruction in `touch2`.
- Do not attempt to use `jmp` or `call` instructions in your exploit code. The encodings of destination addresses for these instructions are difficult to formulate. Use `ret` instructions for all transfers of control, even when you are not returning from a call.
- See the discussion in Appendix B on how to use tools to generate the byte-level representations of instruction sequences.

All you need is in README.pdf (cont.)



If you generate a hex-formatted exploit string in the file `exploit.txt`, you can apply the raw string to CTARGET or RTARGET in several different ways:

1. You can set up a series of pipes to pass the string through HEX2RAW.

```
unix> cat exploit.txt | ./hex2raw | ./ctarget -q
```

2. You can store the raw string in a file and use I/O redirection:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -q < exploit-raw.txt
```

   This approach can also be used when running from within GDB:

```
unix> gdb ctarget
(gdb) run -q < exploit-raw.txt
```

3. You can store the raw string in a file and provide the file name as a command-line argument:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -q -i exploit-raw.txt
```

   This approach also can be used when running from within GDB.

Q&A

- Do not perform CI, ROP, or just buffer overflow in other situation.
- Attacks are only allowed to *ctarget* and *rtarget*.