Assembly Lab, Part 2

RECAP

Stacks and Return Addresses

echo:

24 bytes (decimal)

```
0000000000040069c <echo>:
 40069c:   48 83 ec 18          sub    $0x18,%rsp
 4006a0:   48 89 e7             mov    %rsp,%rdi
 4006a3:   e8 a5 ff ff ff       callq  40064d <gets>
 4006a8:   48 89 e7             mov    %rsp,%rdi
 4006ab:   e8 50 fe ff ff       callq  400500 <puts@plt>
 4006b0:   48 83 c4 18          add    $0x18,%rsp
 4006b4:   c3                   retq
```

After echo() finishes execution, the return address 4006c3 is popped from the stack

call_echo:

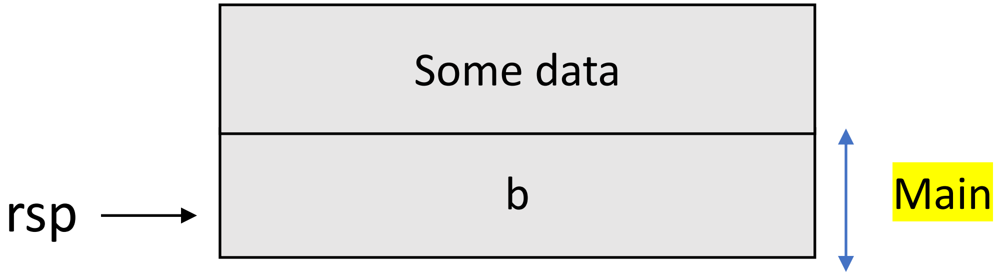
return address

```
 4006b5:   48 83 ec 08          sub    $0x8,%rsp
 4006b9:   b8 00 00 00 00       mov    $0x0,%eax
 4006be:   e8 d9 ff ff ff       callq  40069c <echo>
 4006c3:   48 83 c4 08          add    $0x8,%rsp
 4006c7:   c3                   retq
```

call to function echo() pushes the return address 4006c3 to the stack

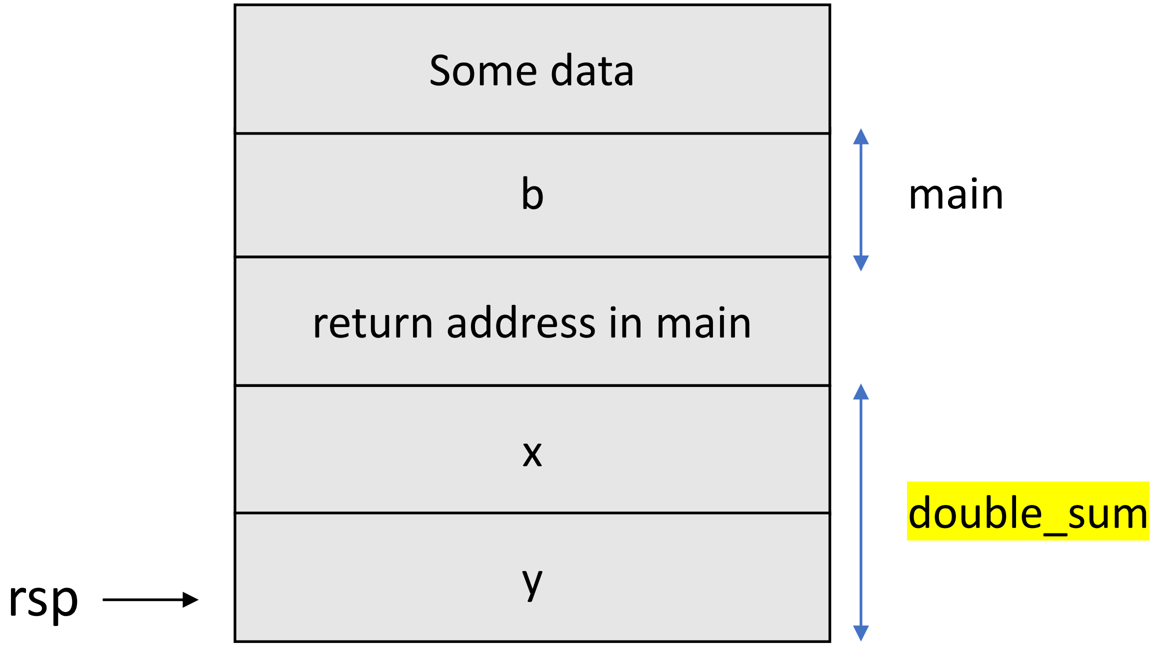
Functions and Memory Stack

```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```



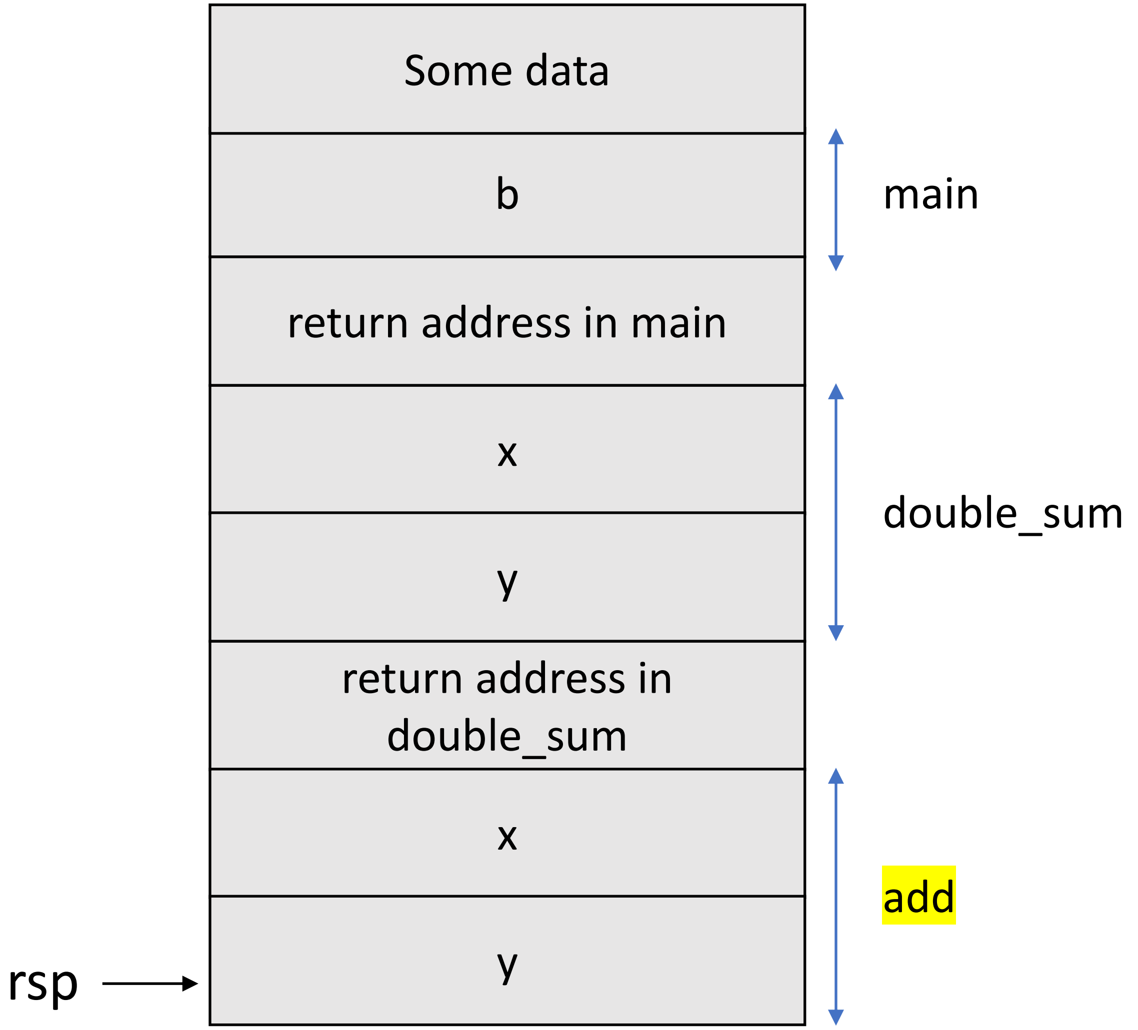
Functions and Memory Stack

```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```



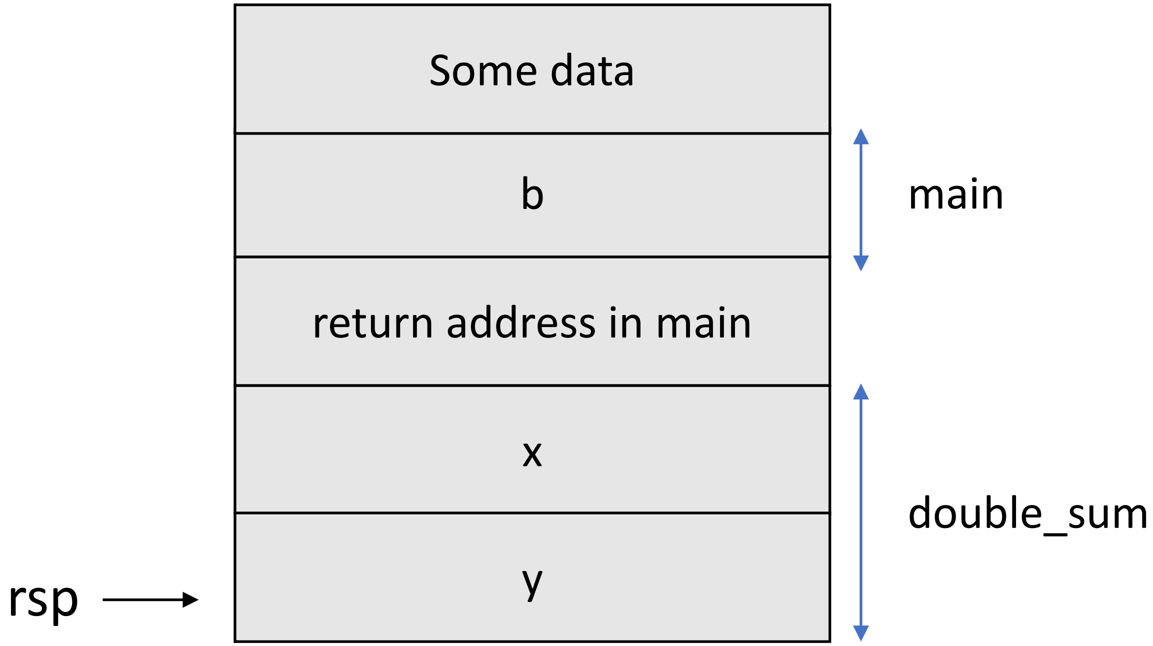
Functions and Memory Stack

```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```



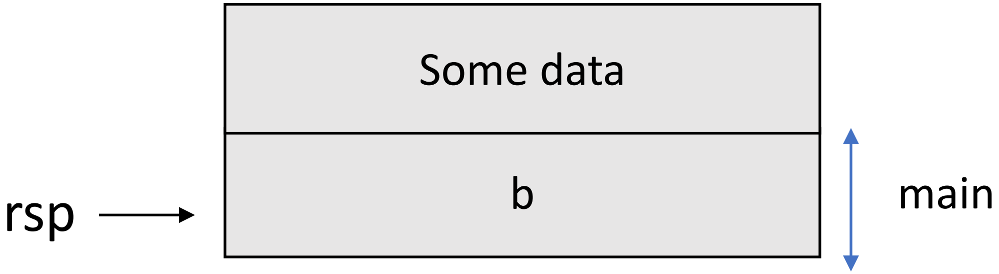
Functions and Memory Stack

```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```



Functions and Memory Stack

```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```



Functions and Memory Stack


```
int add(int x, int y)
{  return x+y;
}
int double_sum(int x, int y){
return 2*add(x,y);
}
int main() {
int b = double_sum(10, 20);
return 0;
}
```

Buffer Overflow

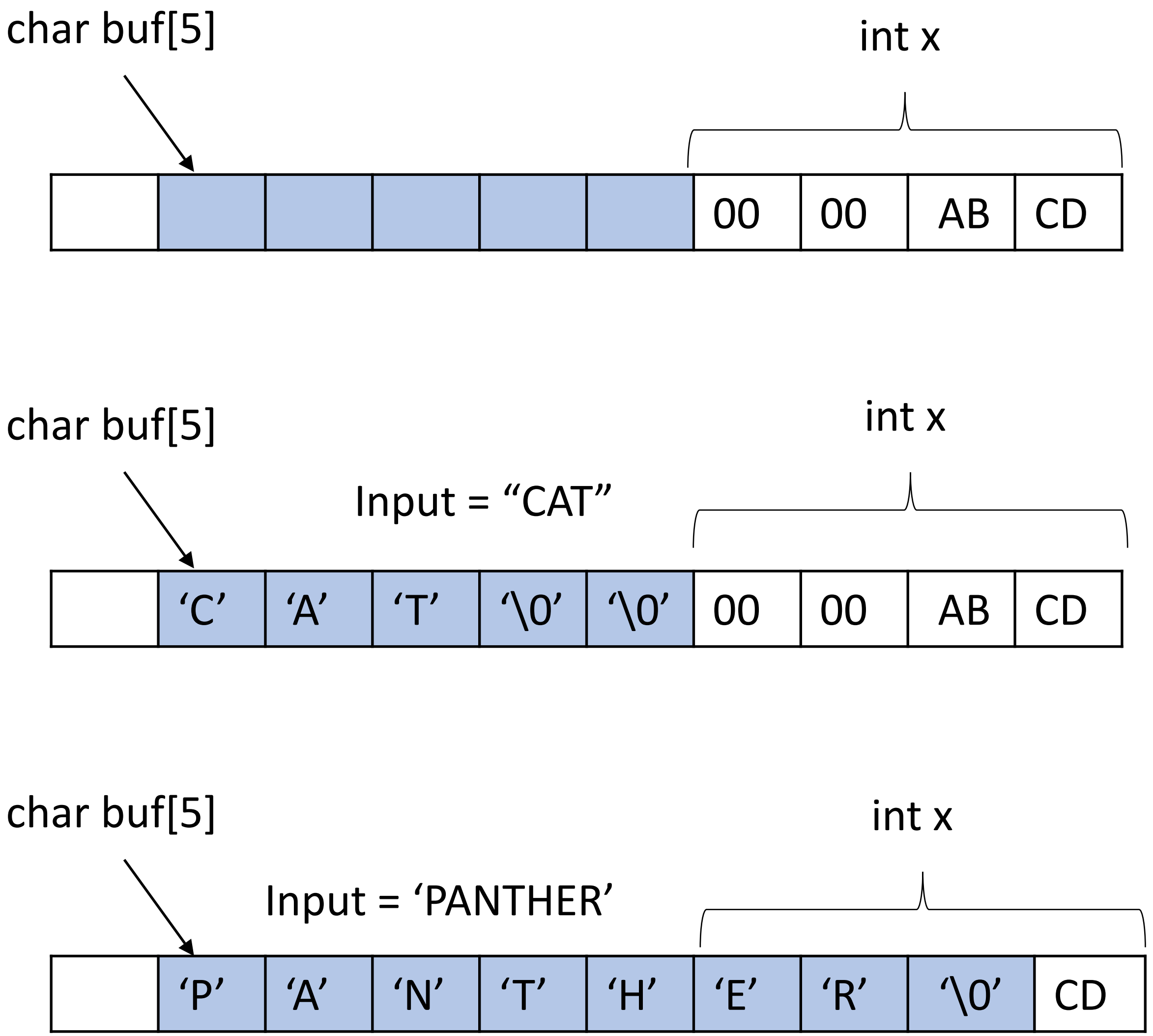
Problem with gets(buf)

```
int getBuffer()
{
	char buf[SIZE];
	int x = 16;
	gets(buf);
}
```

- Gets(buf) reads in user input till there is *'\n'* or *EOF*
- No constraint on the size of input
- Gets() returns the input to the given address even if size is longer than allocated space of memory

Problem with Gets(buf)

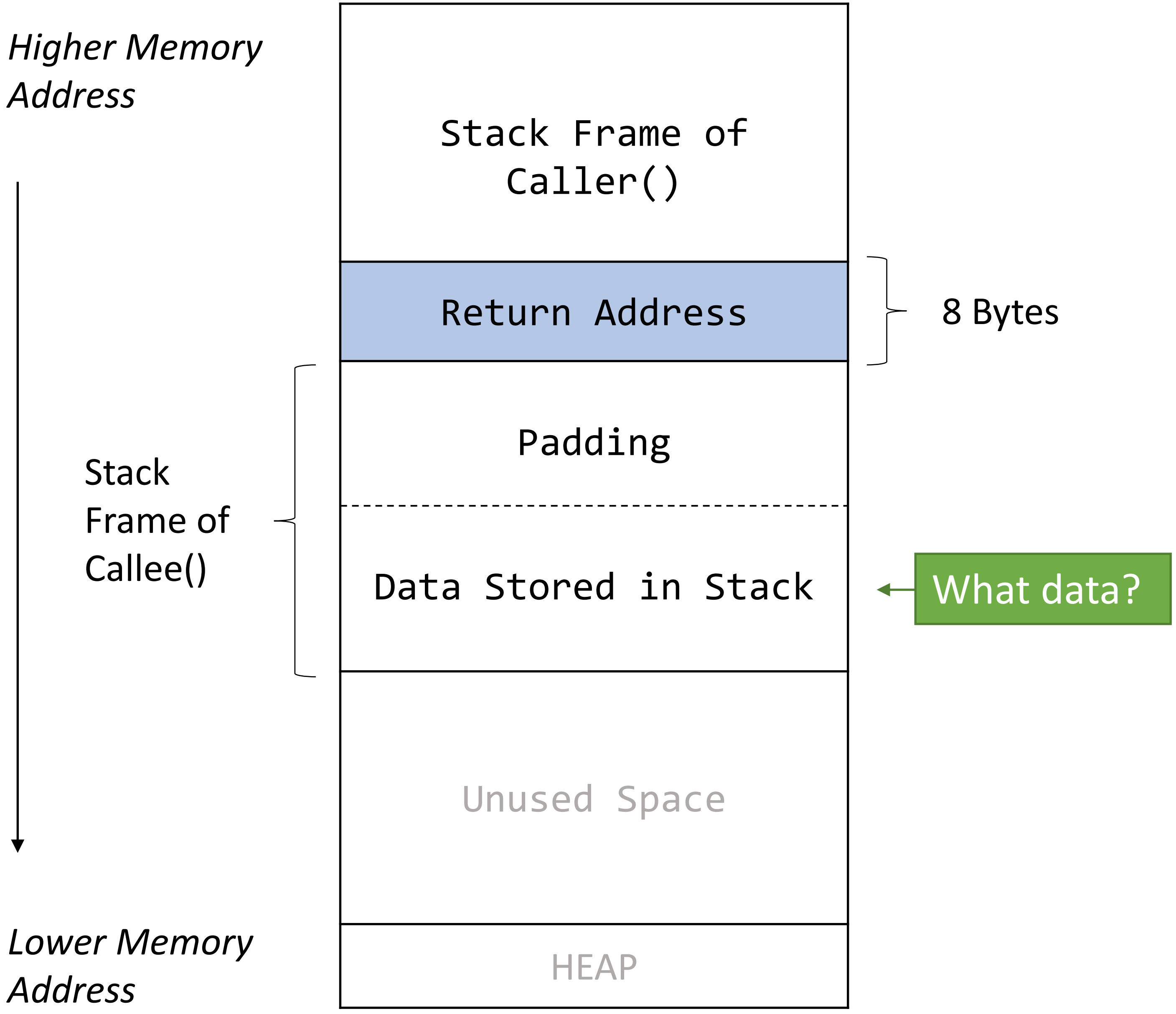
```
int getBuffer()
{
        char buf[SIZE];
        int x = 43981;
        gets(buf);
}
```



What if it overwrites the return address?

Stack

```
void Caller(){
        …
        Callee();
        …
}

void Callee(){
        …
        local variables
}
```



Attack 1

- Change the value of a constant integer variable in the stack by overflowing the buffer

User Input

USER INPUT:

exploit1.txt

```
void read_my_number() {
  // stack allocate some variables
  const int my_number = 72;  // my number is a constant so it can't change
  char buf[BUFSIZE];

  // print favorite number
  printf("My number is %d and nothing can change that\n", my_number);

  // get a string from stdin
  gets(buf);

  // print favorite number again bc its a great number
  printf("My number is %d and nothing can change that\n", my_number);
}
```

How to prepare exploit1.txt?

1. Overflow *buf*

```
void read_my_number() {
  // stack allocate some variables
  const int my_number = 72;  // my number is a constant so it can't change
  char buf[BUFSIZE];

  // print favorite number
  printf("My number is %d and nothing can change that\n", my_number);

  // get a string from stdin
  gets(buf);

  // print favorite number again bc its a great number
  printf("My number is %d and nothing can change that\n", my_number);
}
```

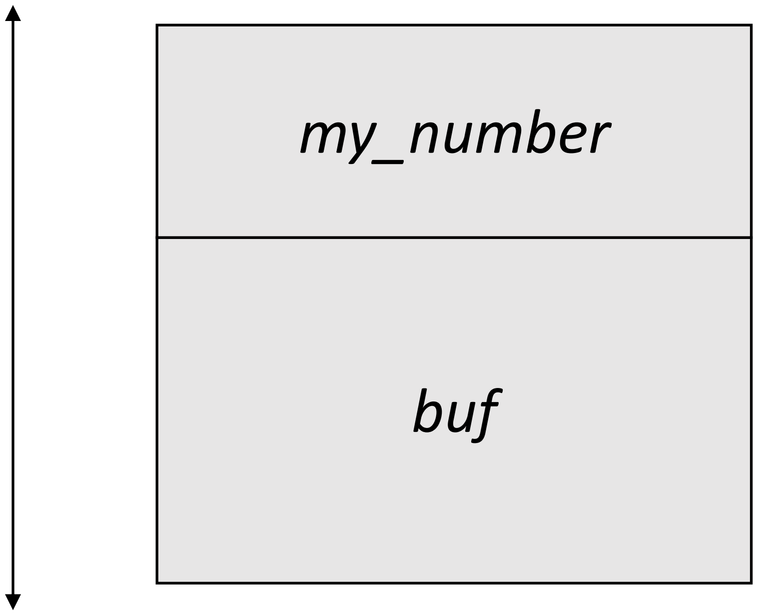
How to prepare exploit1.txt?

```
void read_my_number() {
  // stack allocate some variables
  const int my_number = 72;  // my number is a constant so it can't change
  char buf[BUFSIZE];

  // print favorite number
  printf("My number is %d and nothing can change that\n", my_number);

  // get a string from stdin
  gets(buf);

  // print favorite number again bc its a great number
  printf("My number is %d and nothing can change that\n", my_number);
}
```

2. Overwrite *my_number*

Stack Content

What is the total size??




```
Let’s check the
x86 code!!
```

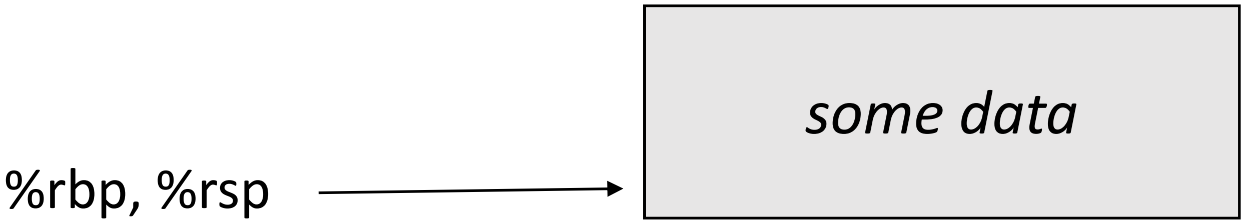
Code

Space allocated in the stack = 0x30 Bytes

- Where is the my_number??
  - at address *%rbp - 4*

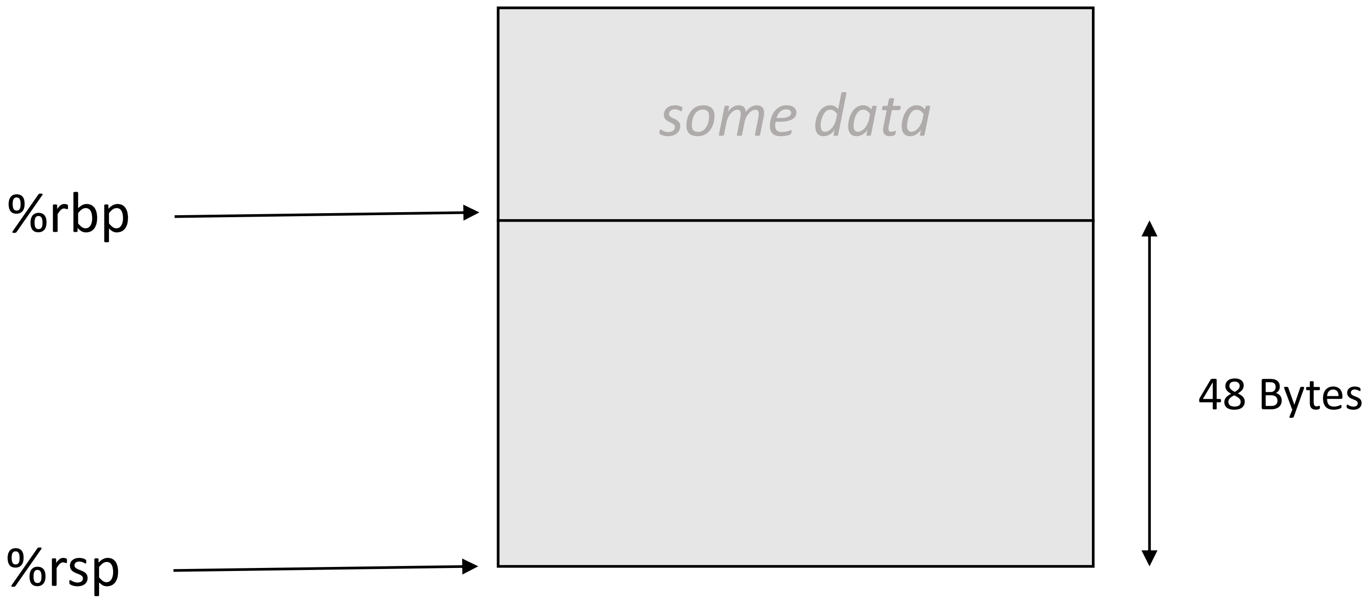
```
000000000040124c <read_my_number>:
  40124c:       f3 0f 1e fa             endbr64
  401250:       55                      push   %rbp
  401251:       48 89 e5                mov    %rsp,%rbp
  401254:       48 83 ec 30             sub    $0x30,%rsp
  401258:       c7 45 fc 48 00 00 00    movl   $0x48,-0x4(%rbp)
  40125f:       8b 45 fc                mov    -0x4(%rbp),%eax
  401262:       89 c6                   mov    %eax,%esi
  401264:       bf 88 20 40 00          mov    $0x402088,%edi
  401269:       b8 00 00 00 00          mov    $0x0,%eax
  40126e:       e8 4d fe ff ff          callq  4010c0 <printf@plt>
  401273:       48 8d 45 d0             lea    -0x30(%rbp),%rax
  401277:       48 89 c7                mov    %rax,%rdi
  40127a:       b8 00 00 00 00          mov    $0x0,%eax
  40127f:       e8 5c fe ff ff          callq  4010e0 <gets@plt>
  401284:       8b 45 fc                mov    -0x4(%rbp),%eax
  401287:       89 c6                   mov    %eax,%esi
  401289:       bf 88 20 40 00          mov    $0x402088,%edi
  40128e:       b8 00 00 00 00          mov    $0x0,%eax
  401293:       e8 28 fe ff ff          callq  4010c0 <printf@plt>
  401298:       90                      nop
  401299:       c9                      leaveq
  40129a:       c3                      retq
```

Stack Content



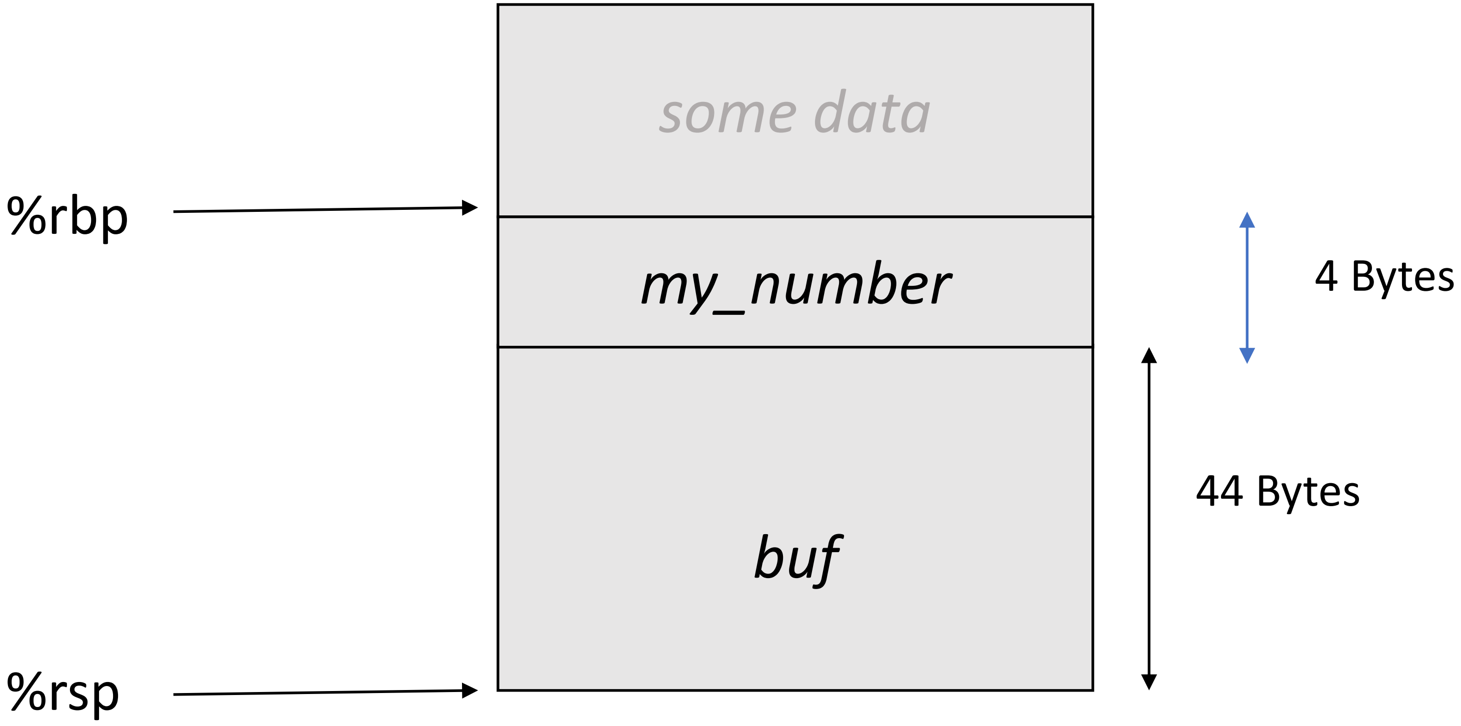
Stack Content

```
sub 0x30, %rsp
```



Stack Content

```
sub 0x30, %rsp
movl   $0x48,-0x4(%rbp)
```



How to prepare exploit string?

- Task 1:
  - Overflow buf
  - How many bytes of data?
  - Can you use any character?

- Task 2:
  - Change value of *my_num* to 449 or 0x01C1

Little Endian

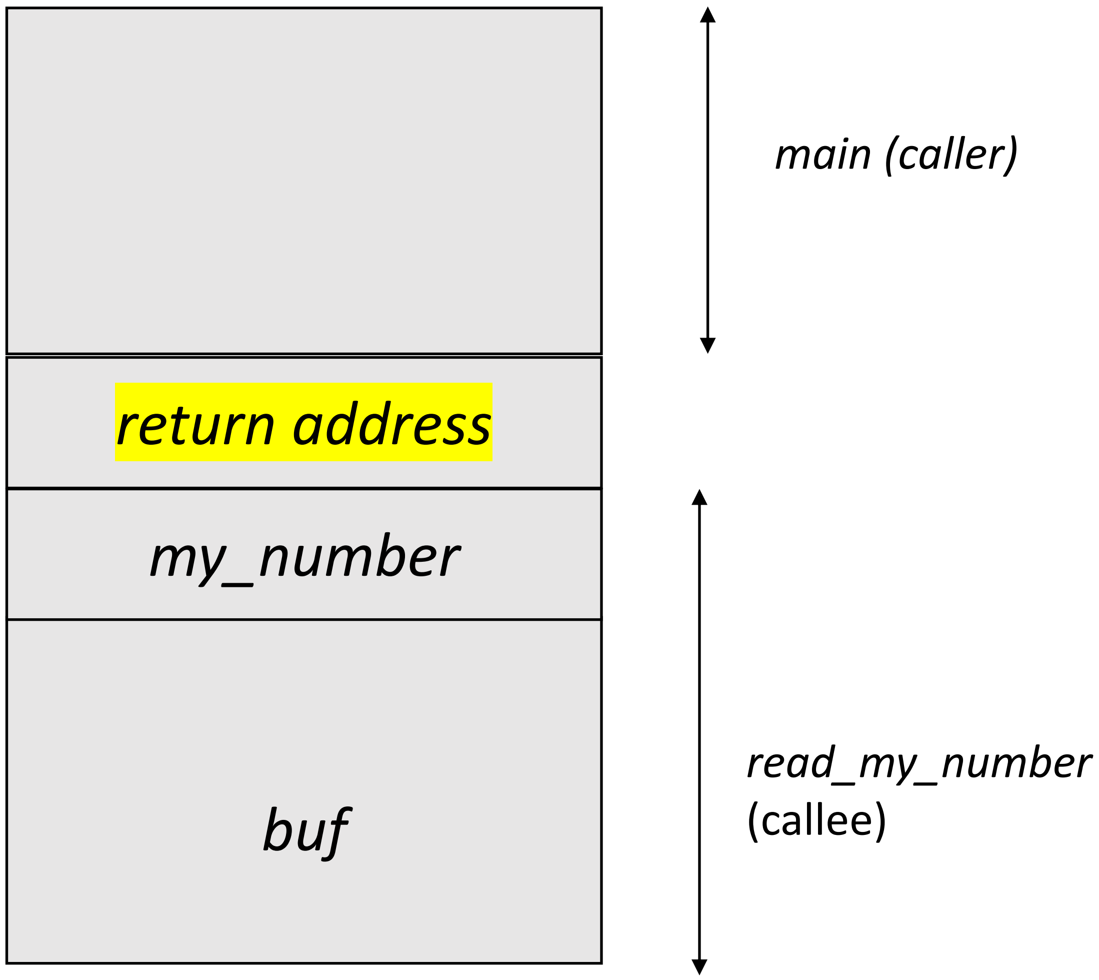
- Write the exploit string for the number 0xABCD10EF
    - Tip: LSB goes to the lowest address

- Exploit string: EF 10 CD AB

- *Demo*

Attack 2

- Overflow the buffer to overwrite the return address



Important Points

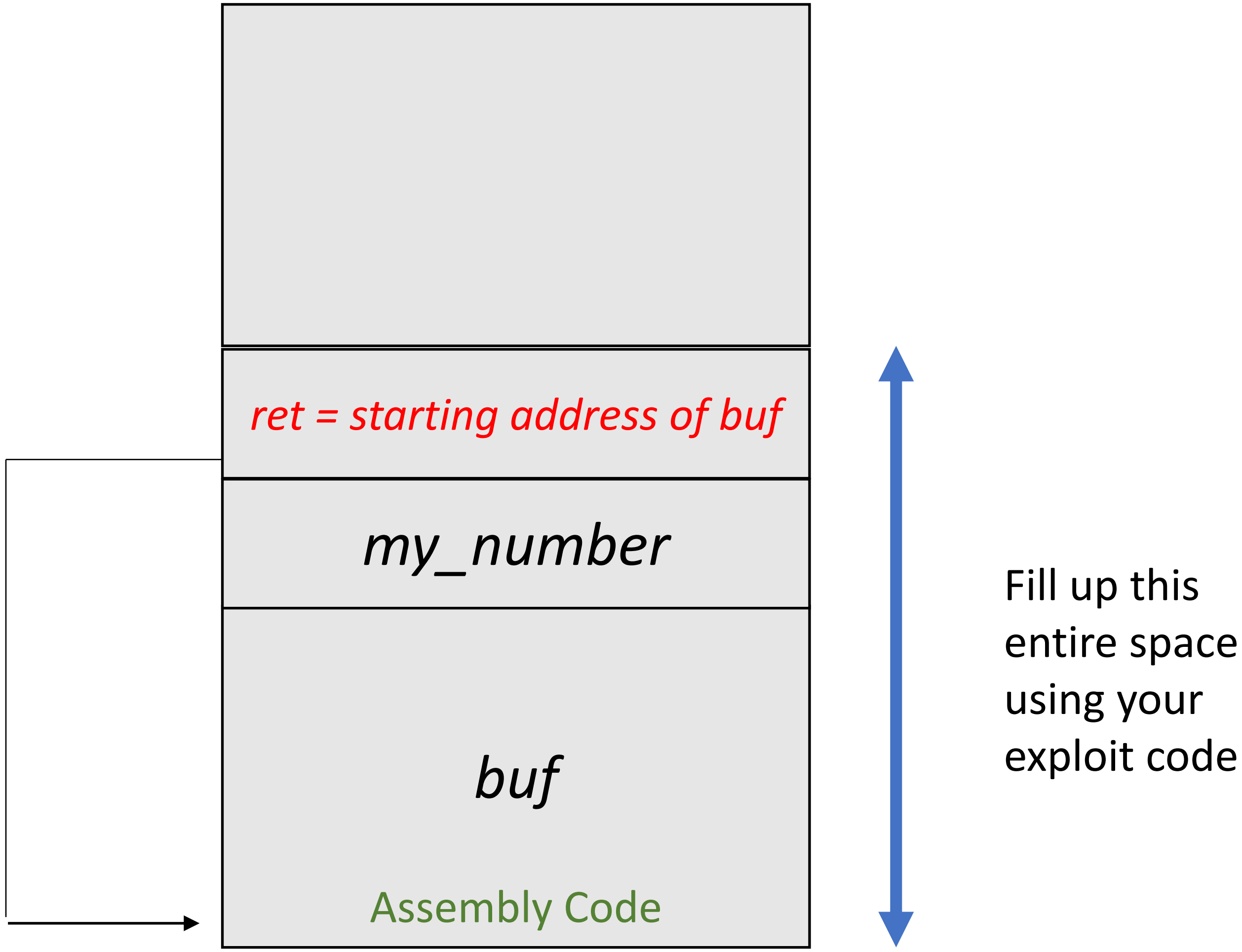
- What do we overwrite the return address with??
  - Address of function hack()
    - How to get this??
- How big should our exploit string be?
  - Is it 48 + 8 Bytes??
  - You need to find out using gdb.
    - Use ***x /10gx $rsp*** inside gdb to look at the contents of the stack

Attack 3

Code Injection

1. Replace the return address with address of buf

2. Insert assembly code in the buf



1. Address of buf

- How to get the address of buf?
- Approach 1:
  - What is the value of %rbp?
  - How many bytes was allocated in the stack?

Function hack()

```
void hack(unsigned val) {
  printf("You've been HACKED!\n");
  if (val == COOKIE) {
    printf("Yes! Called passing the right argument (%d)\n", val);

   } else {
    printf("Misfire! Called it passing the wrong argument (%d)\n", val);
   }
   exit(0);
 }
```

2. Insert Assembly Code

- What should your assembly code do?
  - Call hack() by passing the argument COOKIE

  - What is COOKIE?
    - Read the x86 code of hack – Demo

  - How to write the assembly for calling hack??
    - DO NOT use **call** or **jmp** instruction
    - Instead use **ret**

2. Insert Assembly Code

- What should your assembly code do?
  - Push the return address of hack to the stack
  - Put the value COOKIE in the correct register
    - Remember which registers are used for passing arguments??
  - Finally, return to the address of hack

3. Converting from x86 to machine code

- You cannot insert x86 code directly to the stack
  - Convert it to machine code

- Demo