Intro to Binary Exploitation

Nathan Huckleberry

University of Texas at Austin

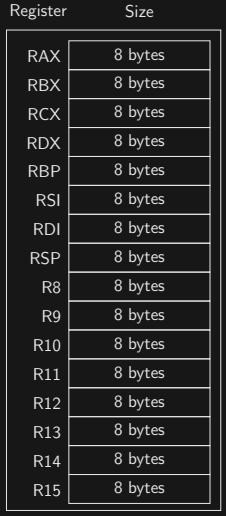
April 14, 2021

Table of Contents

Table of Contents

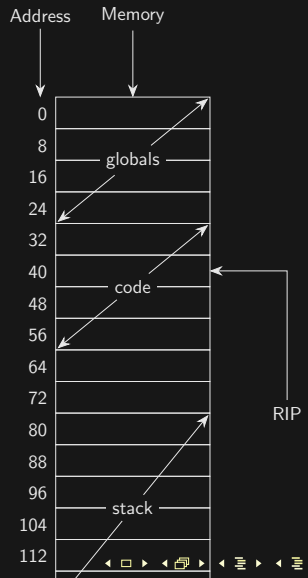
Registers

- Stores integer data.
- Only 16 Registers.
- Many have special purposes.
- Not much room for program variables, data structures etc.



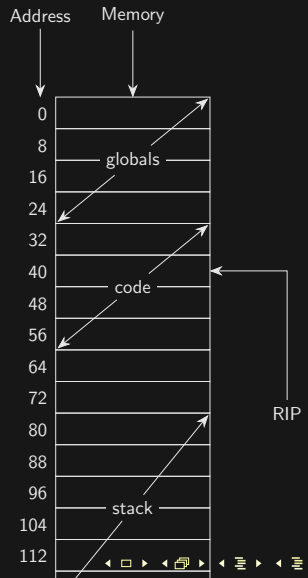
Memory

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



Memory

- Memory stores 99% of variables.
- The actual machine code is stored in memory.
- The register RIP is a pointer into memory
- RIP keeps track of what instruction should be executed next.



Memory

- The registers RBP and RSP are pointers into memory
- RBP and RSP keep track of function local variables

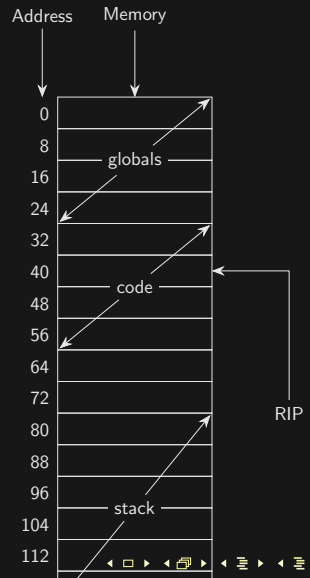


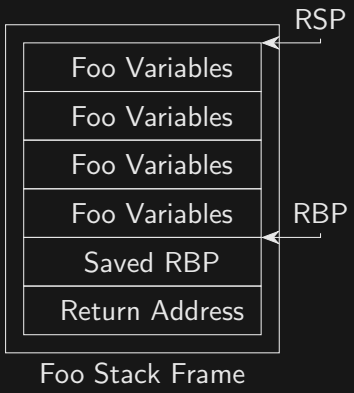
Table of Contents

Function Calls

- Each function call must keep track of its own local variables.
- These local variables are stored in a memory region called the program stack.
- Each function call is given a piece of the program stack called a stack frame.

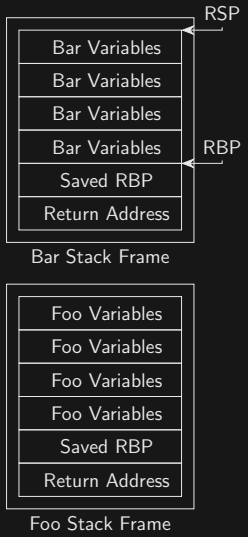
Stack Frames

- Each function call is allocated some memory.
- The function call stores its local variables here.
- RBP and RSP keep track of the beginning and end of this region.



Stack

- The program stack is just a stack of stack frames
- RSP and RBP keep track of the top-most stack frame
- The bottom of the stack is at a very high address and it grows towards smaller addresses



Stack Metadata

- Metadata about the calling function is stored between stack frames.
- Tells us where the previous stack frame base was.
- Tells us where to return after this function is finished.

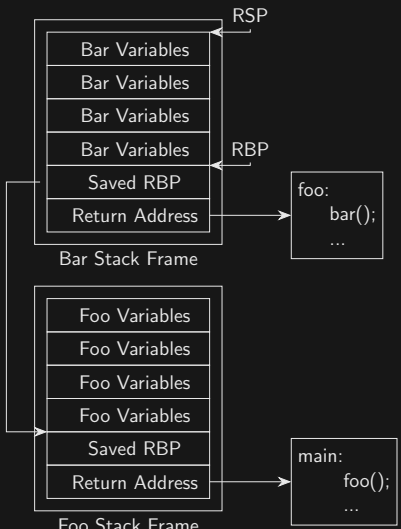


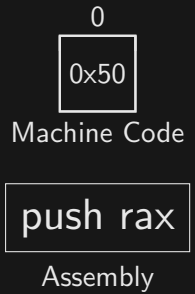
Table of Contents

Assembly

- For the purposes of binary exploitation we can ignore most assembly instructions
- We only really care about push, pop, call and ret
- These are the instructions that interact with the program stack

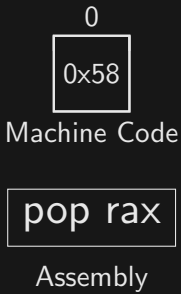
Push Instruction

- Pushes a register value onto the top of the stack.
- Decrements rsp by 8.
- Stores the value in rax into memory at rsp.



Pop Instruction

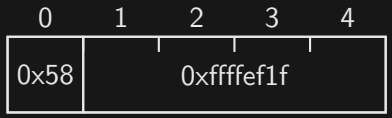
- Pops the value at the top of the stack into a register.
- Load the value in memory at rsp into rax.
- Increment rsp by 8.



Call Instruction

- Calls a function and sets up stack frame metadata.

- Push rip onto stack
- Jump to call address




```
call <function>
```

Assembly

Ret Instruction

- Use the stack frame metadata to return from a function call.
- Pop into rip
- Continue execution at rip

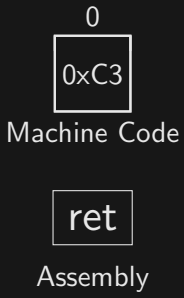


Table of Contents

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 1: x86-64 go.o

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

Vulnerable functions

Listing 2: Man Page For gets

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

Overwriting Sensitive Data

Listing 3: 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
```

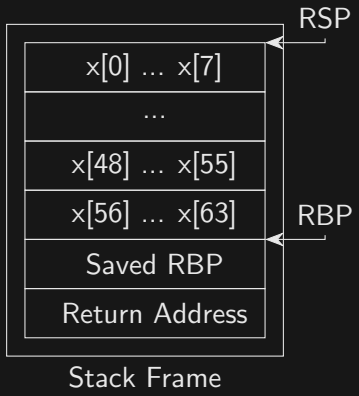
Listing 4: x86-64 go.o

```
1  void user_input() {
2      char x[64];
3      gets(x);
4  }
```

Overwriting Sensitive Data

Listing 5: x86-64 go.o

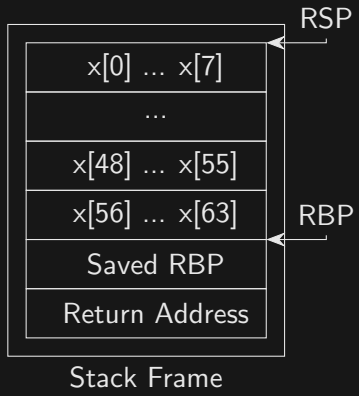
```
1  user_input:
2      push rbp
3      mov rbp, rsp
4      sub rsp, 64
5      lea rax, [rbp-64]
6      mov rdi, rax
7      mov eax, 0
8      call gets
9      nop
10     leave
11     ret
```



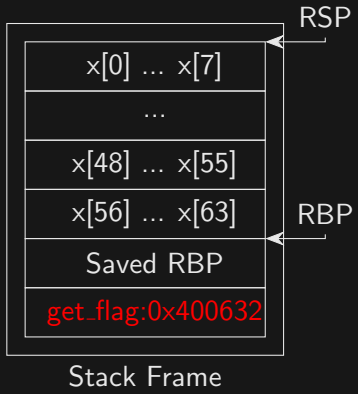
Overwriting Sensitive Data

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

Stack Frame


RSP
x[0] ... x[7]
...
x[48] ... x[55]
x[56] ... x[63]
RBP
Saved RBP
Return Address
Stack Frame

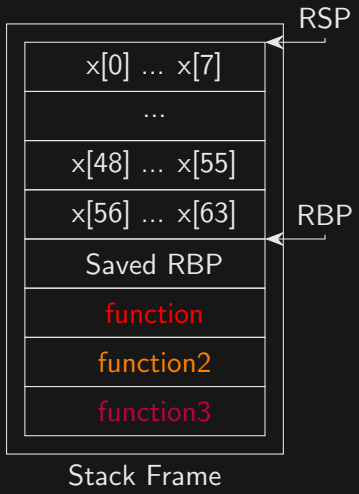
Stack Frame



[1]When overflowing the return address, we must put the bytes in backwards to account for endianness (... x[63] Saved RBP 0x32 0x06 0x40 0x00 ...)

Stack Frame

- Functions can also be chained together
- This is known as a ROP-chain
- function1 will be called, then function2, then function3



What to overwrite RIP to

- Easy problems have a "get flag" function
- Libc built-in functions like system() are useful
- ROP gadgets
- Jump straight to libc system() using an address leak
- Jump to onegadget in libc using an address leak

ROP Gadgets

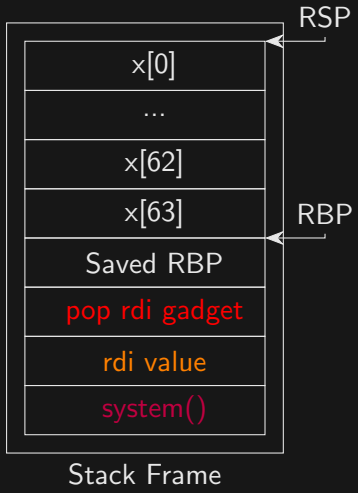
- A short instruction sequence followed by a ret instruction
- Can be put into a ROP-chain the same way as a function
- Pop gadgets are useful for controlling registers

Listing 6: pop rdi gadget

```
1 pop rdi
2 ret
```

ROP Gadgets

- A short instruction sequence followed by a ret instruction
- Can be put into a ROP-chain the same way as a function
- This will call system() with an attacker controlled rdi



ROP Gagets

- Rop gadgets can be found using a tool called ROPGadget

Demo

Commands to Remember

Listing 7: Commands

```
objdump -d prog.o
gdb prog.o
(python3 -c "print('a'*20)"; cat -) | nc ctf.isss.io 9002
ROPGadget --binary pwnable
```