Disassembly with objdump

```
> objdump -d -M intel exit
exit:     file format elf64-x86-64


...

Disassembly of section .text:

0000000000400230 <_start>:
  400230:  48 c7 c7 00 00 00 00  mov    rdi,0x0
  400237:  e8 e4 ff ff ff        call   400220 <_exit@plt>
  40023c:  cc                    int3
```

Dumping ELF Objects

```
> objdump -x exit
exit:     file format elf64-x86-64
[ … ]
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400230

Program Header:
[ … ]
Dynamic Section:
  NEEDED               libc.so.6
[ … ]
Sections:
Idx Name Size VMA LMA File off  Algn
[ … ]
SYMBOL TABLE:
[ ... ]
0000000000000000       F *UND* 0000000000000000              _exit@@GLIBC_2.2.5
[ … ]
```

Invoking Syscalls

- **We can, of course, bypass libc and directly ask the kernel for services**
  - Need to know the syscall number (index into syscall table in the kernel)
- **On Linux/x86_64, we use the *`syscall`* instruction to transfer control to the kernel**
  - On linux/x86, traditionally INT 0x80 (interrupt 128)
- **Let's write a program to print a message and exit cleanly**

Invoking Syscalls

```
bits 64                         ; as before
section .text

global _start

_start:
    mov rdx, msg_len            ; len(msg) to rdx
    mov rsi, msg                ; msg to rsi
    mov rdi, 1                  ; fd 1 (stdout) to rdi
    mov rax, 1                  ; write is syscall 1
    syscall                     ; call write(rdi, rsi, rdx)

    mov rdi, 0                  ; status code 0 to rdi
    mov rax, 60                 ; exit is syscall 60
    syscall                     ; call exit(rdi)

section .data                   ; the program's .data section
msg:        db 'aha',0x0a       ; the message as a byte array
msg_len:    equ $-msg           ; len is current addr - msg
```

Invoking Syscalls

```
> yasm -f elf64 -o hello.o hello.asm
> ld -o hello hello.o
> ./hello
aha
> echo $?
0
```

- You should see something similar to the above
  - Notice we didn't link against libc – no need
  - And, we do not need to invoke the runtime loader

Debugging Programs

Sometimes your program (or exploit) is buggy

– Well, mine are

An interactive debugger is an invaluable tool in these cases

– Start, stop execution
– Set breakpoints, watchpoints
– Directly inspect memory and CPU state
– Modify program state

Debugging binary programs? gdb!

GDB Quick Reference Card

GDB QUICK REFERENCE GDB Version 4

Essential Commands

gdb *program* [*core*]	debug *program* [using coredump *core*]
b [*file*:]*function*	set breakpoint at *function* [in *file*]
run [*arglist*]	start your program [with *arglist*]
bt	backtrace: display program stack
p *expr*	display the value of an expression
c	continue running your program
n	next line, stepping over function calls
s	next line, stepping into function calls

Starting GDB

gdb	start GDB, with no debugging files
gdb *program*	begin debugging *program*
gdb *program core*	debug coredump *core* produced by *program*
gdb --help	describe command line options

Stopping GDB

quit	exit GDB; also q or EOF (eg C-d)
INTERRUPT	(eg C-c) terminate current command, or

Break

break
b
break
break +
break -
break *
break
break ..
cond *n*
tbreak .
rbreak
watch
catch
info bre
info wat
clear
clear

Get it! from the web!

Initializing GDB

```
> cat .gdbinit
set disassembly-flavor intel
disp/i $rip
```

- **Initial setup**
  - Set default disassembly syntax
  - Display current instruction at each prompt
- **Also useful for**
  - Setting breakpoints
  - Scripting execution of program to known bad state
  - etc.
- **Let’s debug our hello world program from before**

Starting GDB

```
> gdb hello
GNU gdb (Debian 7.7.1+dfsg-3) 7.7.1
[ ... ]
(gdb) b _start
Breakpoint 1 at 0x4000b0
```

- **We load the program in gdb and set a breakpoint at the entry point (_start)**
- **Breakpoints insert (by default) a software interrupt (`int3`) at the given address**
- **When `int3` executes, control transfers to gdb**
  - Replaces the original instruction
  - After original instruction is executed, the `int3` is restored

Running the Program

```
(gdb) r
Starting program: [...]/hello

Breakpoint 1, 0x00000000004000b0 in _start ()
2: x/i $rip
=> 0x4000b0 <_start>:   movabs rdx,0x4
(gdb)
```

- We run the program and immediately hit our breakpoint
- Our display command prints the current instruction

Single-Stepping

```
(gdb) si
0x00000000004000ba in _start ()
2: x/i $rip
=> 0x4000ba <_start+10>:    mov    rsi,0x6000e4
(gdb)
0x00000000004000c1 in _start ()
2: x/i $rip
=> 0x4000c1 <_start+17>:    mov    rdi,0x1
(gdb)
0x00000000004000c8 in _start ()
2: x/i $rip
=> 0x4000c8 <_start+24>:    mov    rax,0x1
(gdb)
0x00000000004000cf in _start ()
2: x/i $rip
=> 0x4000cf <_start+31>:    syscall
```

- We can single-step the program by issuing `si`

Inspecting State

```
(gdb) p $rax
$1 = 1
(gdb) p $rsi
$2 = 6291684
(gdb) p/x $rsi
$3 = 0x6000e4
(gdb) x/s $rsi
0x6000e4:   "aha\n"
```

- p (print) prints register contents
- x (examine) dereferences addresses
- Formatting suffixes control how values are interpreted
  - /x for hex
  - /s for null-terminated strings

More Inspection

```
(gdb) x/8xb $rsp
0x7fffffffe790: 0x01    0x00    0x00    0x00    0x00    0x00
0x00    0x00
(gdb) x/8xg $rsp
0x7fffffffe790: 0x0000000000000001  0x00007fffffffea5f
0x7fffffffe7a0: 0x0000000000000000  0x00007fffffffea9f
0x7fffffffe7b0: 0x00007fffffffeaaa  0x00007fffffffeace
0x7fffffffe7c0: 0x00007fffffffeadf  0x00007fffffffeaf2
```

- We can print the stack by referencing $rsp
- Repetition suffix (above, 8)
- Additional width suffix
  - b for bytes
  - w for dwords
  - g for qwords

Dumping CPU State

```
(gdb) info registers
rax            0x1   1
rbx            0x0   0
rcx            0x0   0
rdx            0x4   4
rsi            0x6000e4 6291684
rdi            0x1   1
rbp            0x0   0x0
rsp            0x7fffffffe790   0x7fffffffe790
r8             0x0   0
[ … ]
r15            0x0   0
rip            0x4000cf 0x4000cf <_start+31>
eflags         0x202    [ IF ]
cs             0x33 51
ss             0x2b 43
[ … ]
```

- We can dump the entire (visible) CPU state

Mutating State

```
(gdb) set $r10=0x31337
(gdb) p/x $r10
$1 = 0x31337
(gdb) x/s $rsi
0x6000e4: "aha\n"
(gdb) set *0x6000e4=0x0a485542
(gdb) x/s $rsi
0x6000e4: "BUH\n"
```

- We can set values both in registers and in memory

Continuing Execution

```
(gdb) c
Continuing.
BUH
[Inferior 1 (process 7228) exited normally]
```

- And we can continue execution
  - Here, we exit since we don't hit our breakpoint again
  - And, we print out our modified string

Summary

You should now be able to

1. Follow and understand simple assembly programs
2. Create your own programs
3. Debug them using gdb

We are only scratching the surface, but it’s enough to get started

Outline

Assembly Review

Vulnerabilities I

Vulnerabilities II

Defenses & Evasion of Defenses

Malware Analysis

Stack-based Overflows

- Stack-based buffer overflows are the quintessential memory corruption vulnerability
  - Problem has been known since the 1970s
  - First known exploitation by the Morris worm in 1989
  - Rediscovered in a 1995 Bugtraq post
- *"Smashing the stack for Fun and Profit"*
  - Published by Aleph One in Phrack in 1996
- People realized they were everywhere, and that they were a serious security problem

Stack-based Overflows

- Vulnerability stems from several factors
  - Low-level languages like C/C++ are not *memory-safe*
    - Programmers can directly manipulate pointers
    - Memory accesses are not bounds-checked for validity
  - Control information (saved return address) is stored inline with user data on the stack
- Overflowing (writing past the end of) a stack buffer could allow users to control the value of a saved return address
  - When the program returns using a corrupted stack frame, the user then controls execution
  - Then, it's a simple matter of redirecting control flow to injected code

Vulnerable Program

```
int main(int argc, char ** argv) {
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

- This program is clearly vulnerable
  - `strcpy()` performs no bounds-checking, relying instead on finding a terminating null character in the source string
  - Length of `argv[1]` can be longer than the size of `buf`
  - What happens if it does?

Vulnerable Program

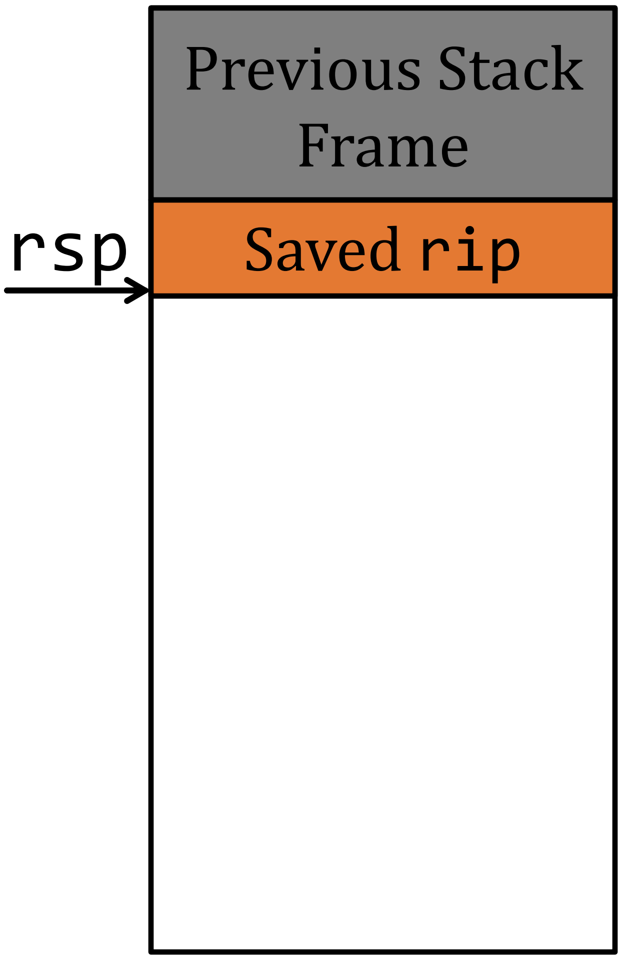
```
> ./vuln `perl -e ‘print “a”x10;’`
aaaaaaaaaa
> ./vuln `perl -e ‘print “a”x300;’`
aaaaaaaaaa…
Segmentation Fault
```

- Let’s look at the mechanics of an exploit

Vulnerable Program

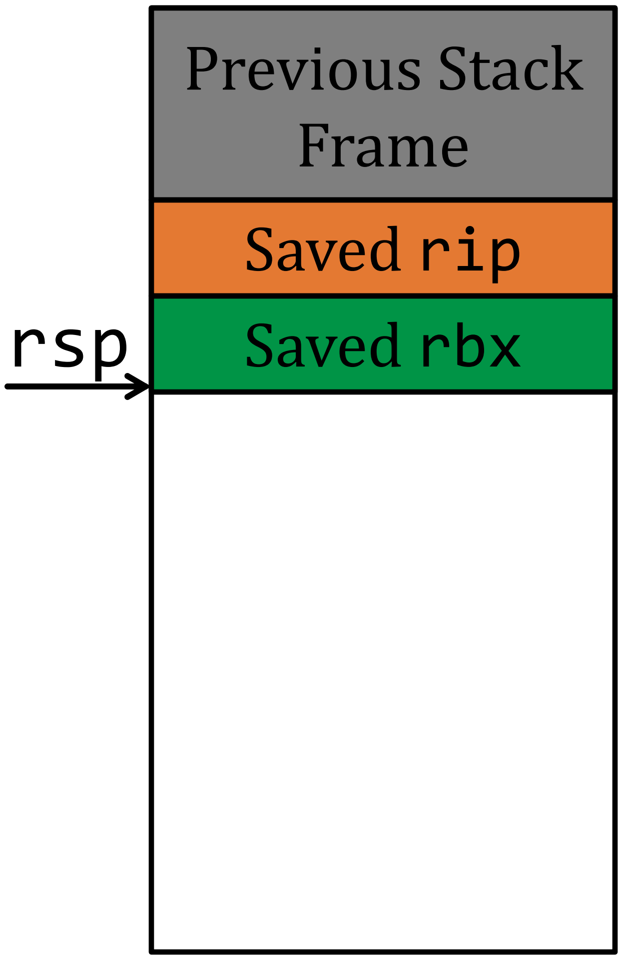
```
main:
    push    rbx                        ; callee saves rbx
    sub     rsp,0x100                  ; allocate buf
    mov     rsi,QWORD PTR [rsi+0x8]    ; move argv[1] to rsi
    lea     rbx,[rsp]                  ; move buf to rbx
    mov     rdi,rbx                    ; move buf to rdi
    call    400440 <strcpy@plt>        ; strcpy(rdi, rsi)
    mov     rdi,rbx                    ; mov buf to rdi
    call    400450 <puts@plt>          ; puts(rdi)
    xor     eax,eax                    ; set eax to 0
    add     rsp,0x100                  ; deallocate buf
    pop     rbx                        ; restore rbx
    ret                                ; return to where?
```

Smashing the Stack



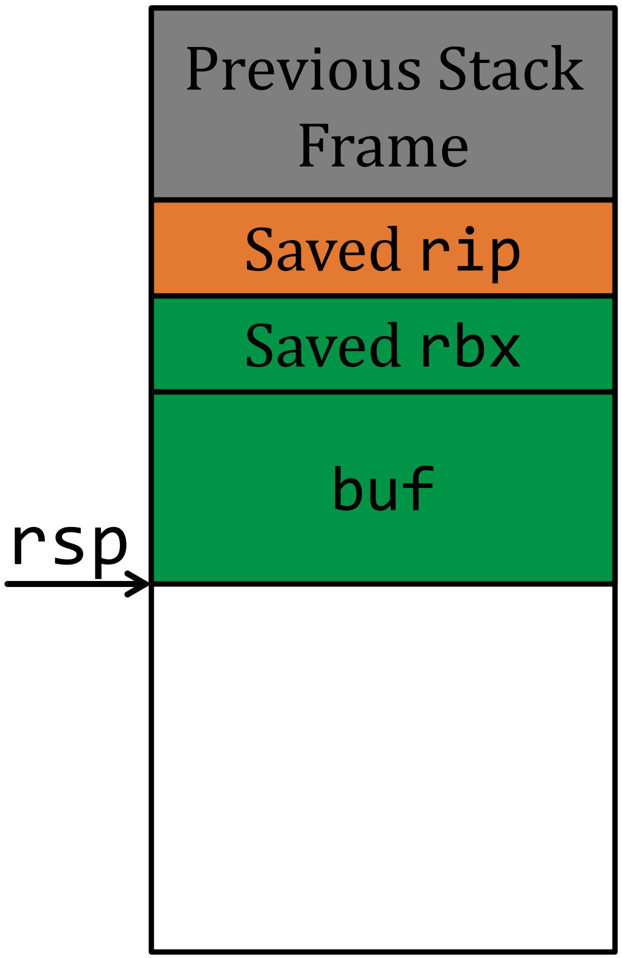

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



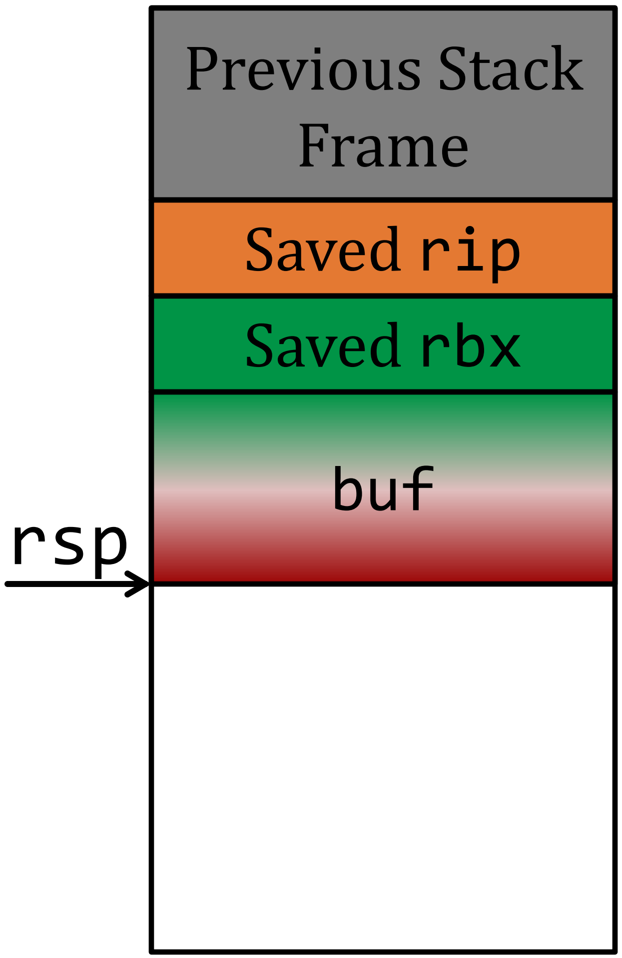

```
main:
    push   rbx
    sub    rsp,0x100
    mov    rsi,QWORD PTR [rsi+0x8]
    lea    rbx,[rsp]
    mov    rdi,rbx
    call   400440 <strcpy@plt>
    mov    rdi,rbx
    call   400450 <puts@plt>
    xor    eax,eax
    add    rsp,0x100
    pop    rbx
    ret
```

Smashing the Stack



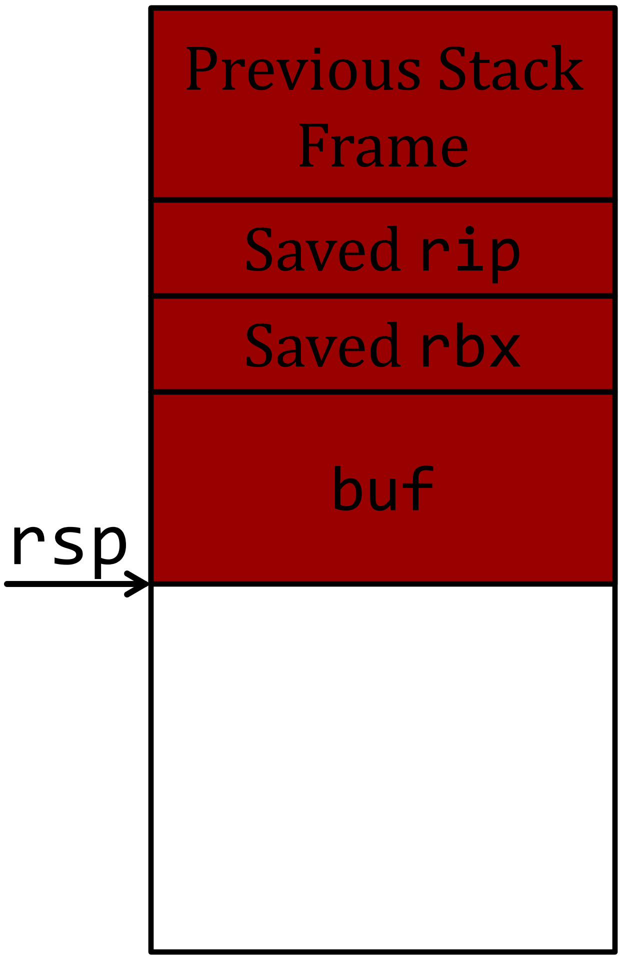

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



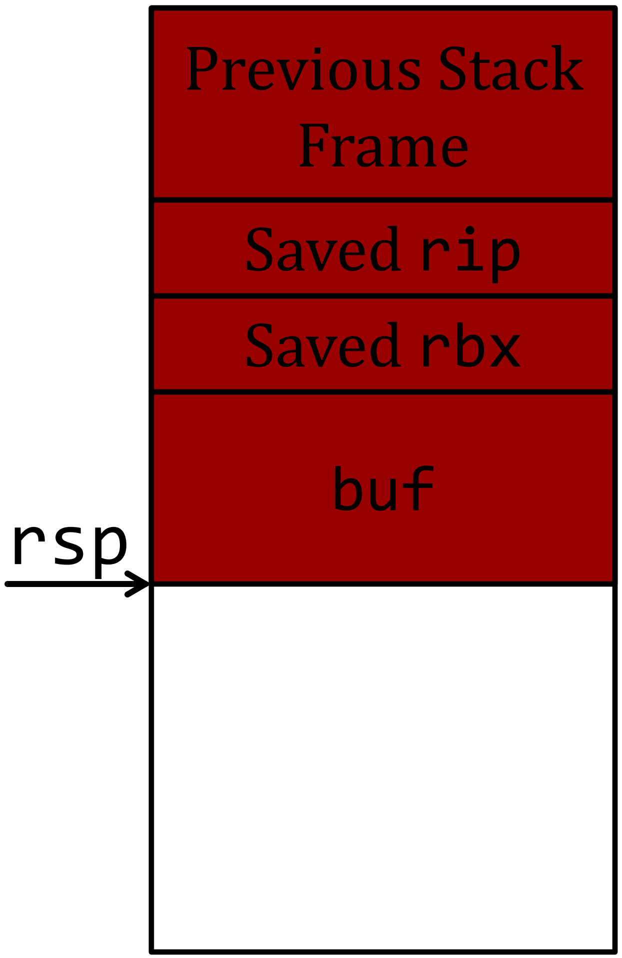

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



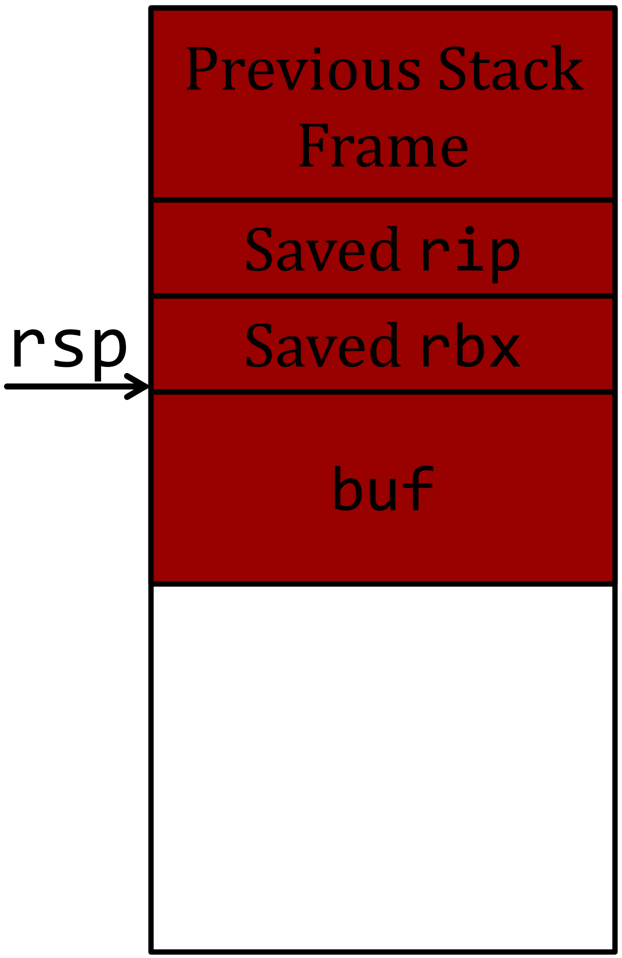

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



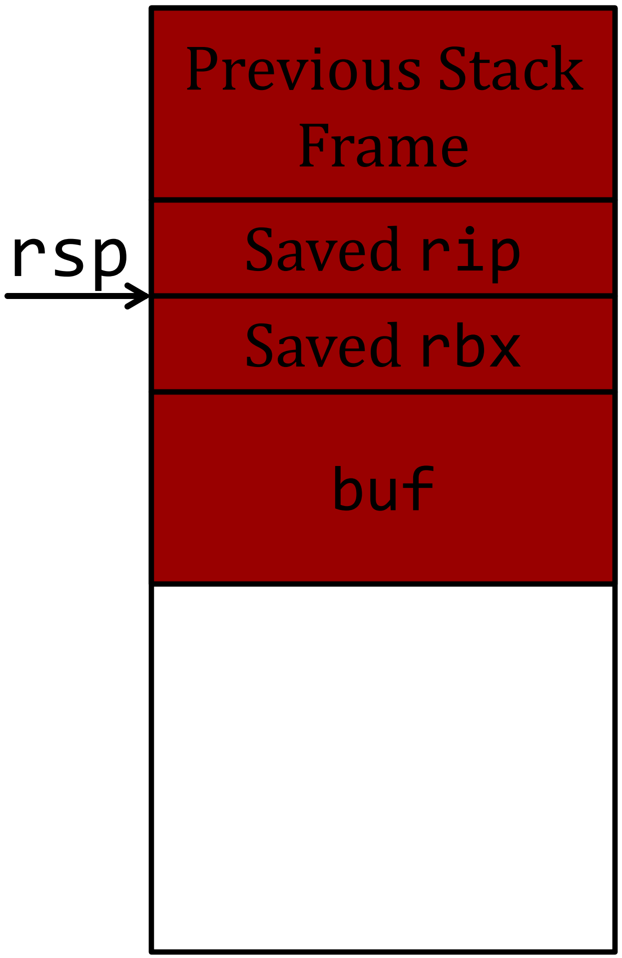

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack




```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack




```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

ret $rip \leftarrow Mem(rsp)$

$rsp \leftarrow rsp + 8$