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

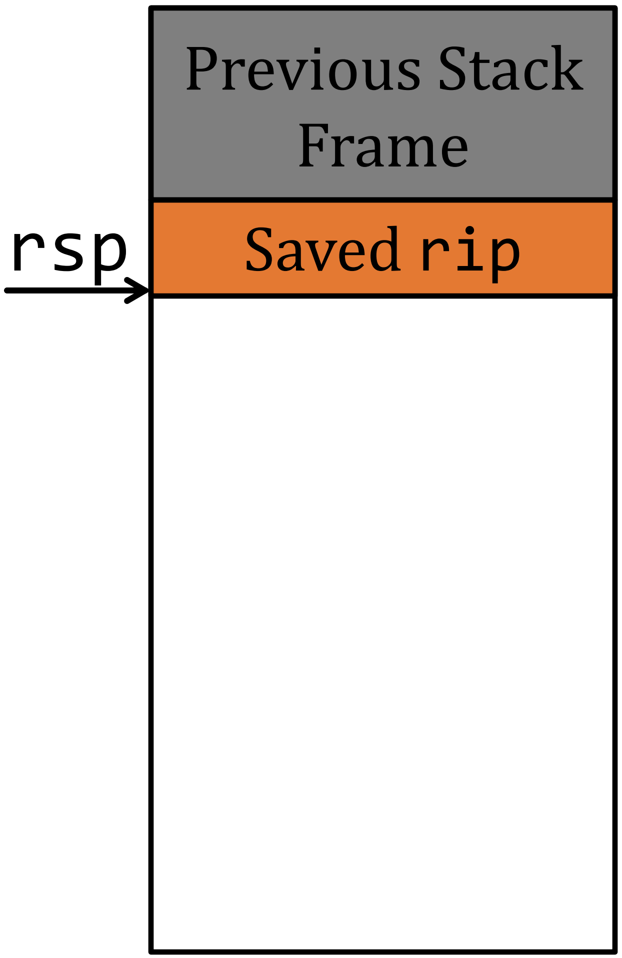
```
> ./vuln `perl -e ‘print “a”x10;’`
aaaaaaaaaa
> ./vuln `perl -e ‘print “a”x300;’`
aaaaaaaaaa…
Segmentation Fault
```

- Let’s look at the mechanics of an exploit

Vulnerable Program

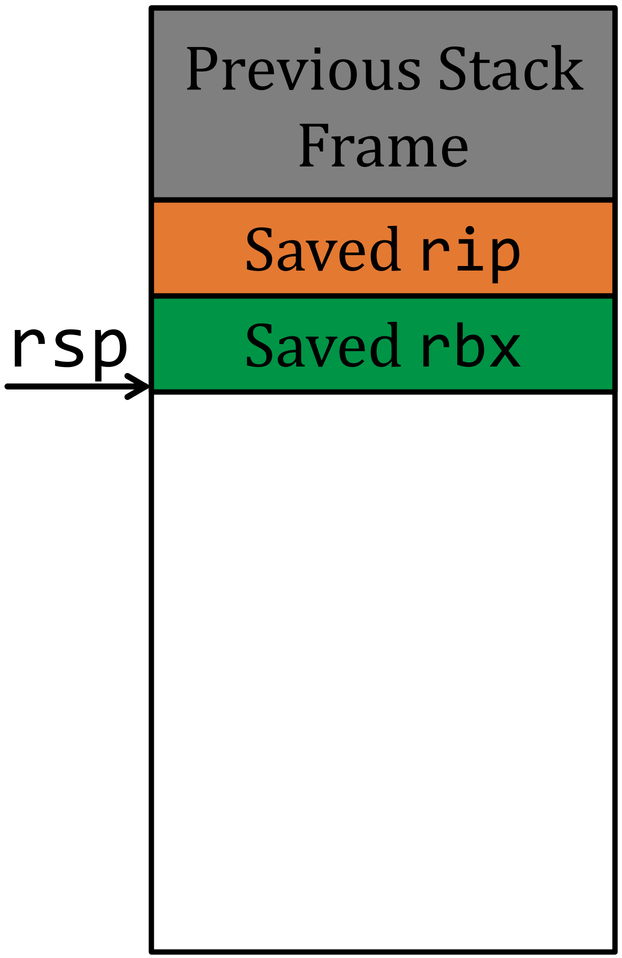
```
main:
    push   rbx                        ; callee saves rbx
    sub    rsp,0x100                  ; allocate buf
    mov    rsi,QWORD PTR [rsi+0x8]    ; move argv[1] to rsi
    lea    rbx,[rsp]                  ; move buf to rbx
    mov    rdi,rbx                    ; move buf to rdi
    call   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



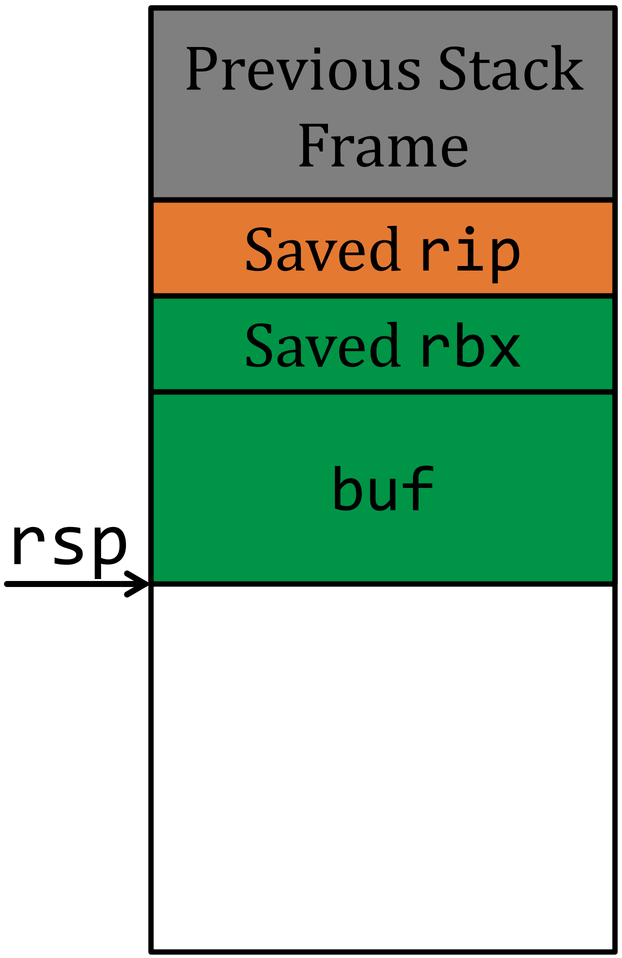

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



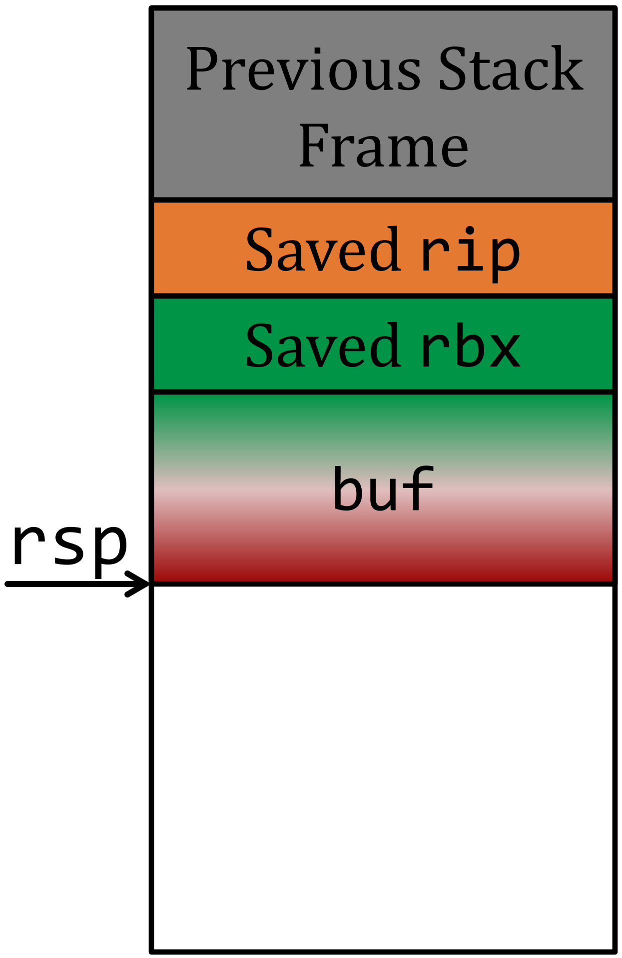

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



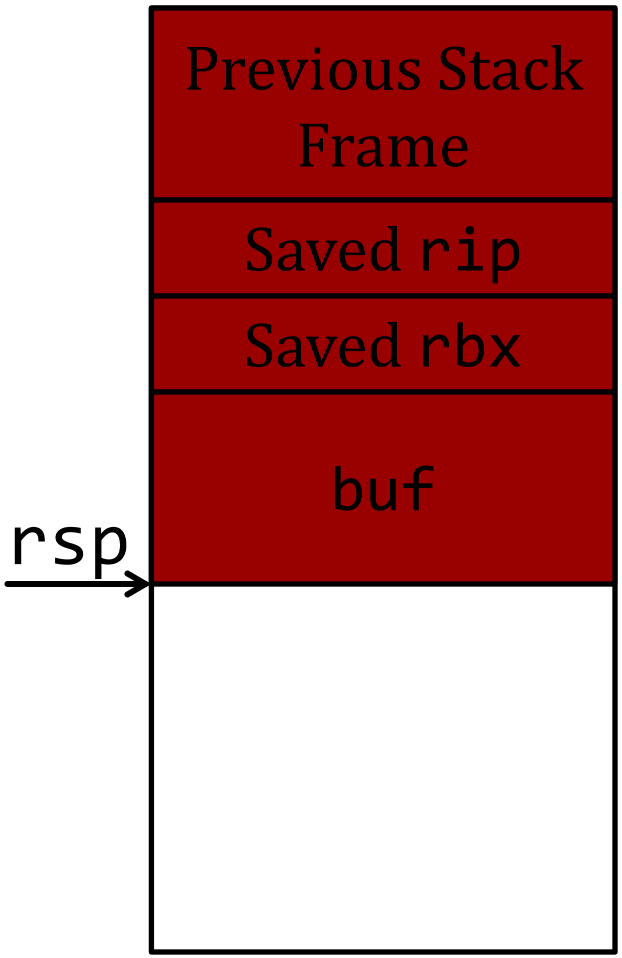

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



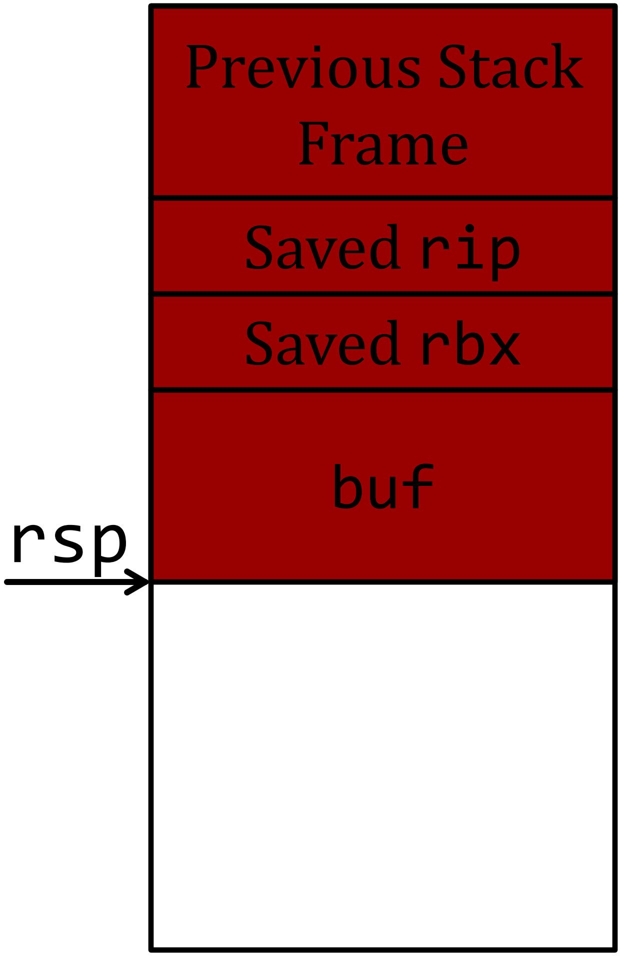

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



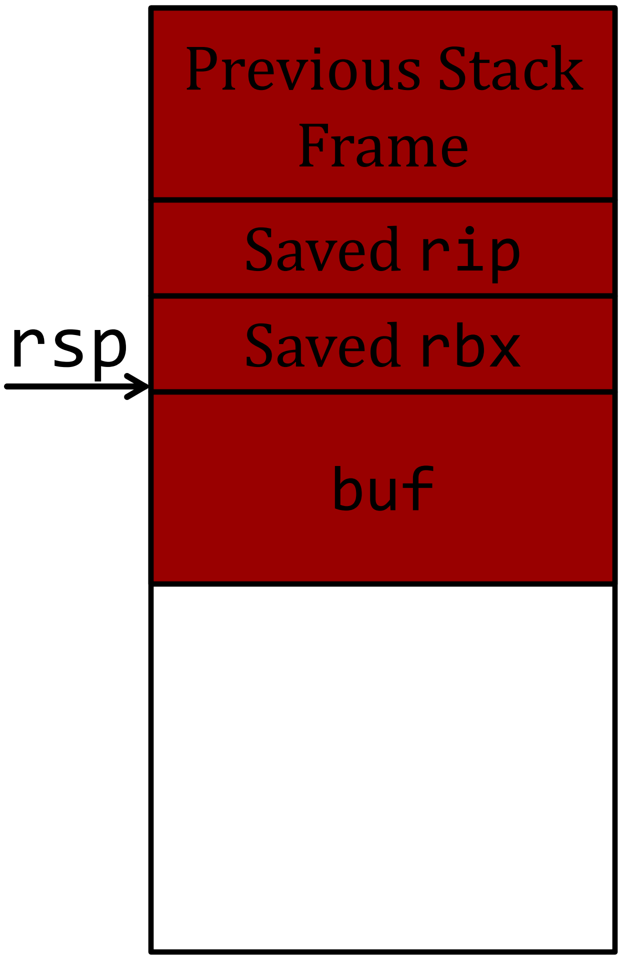

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

Smashing the Stack



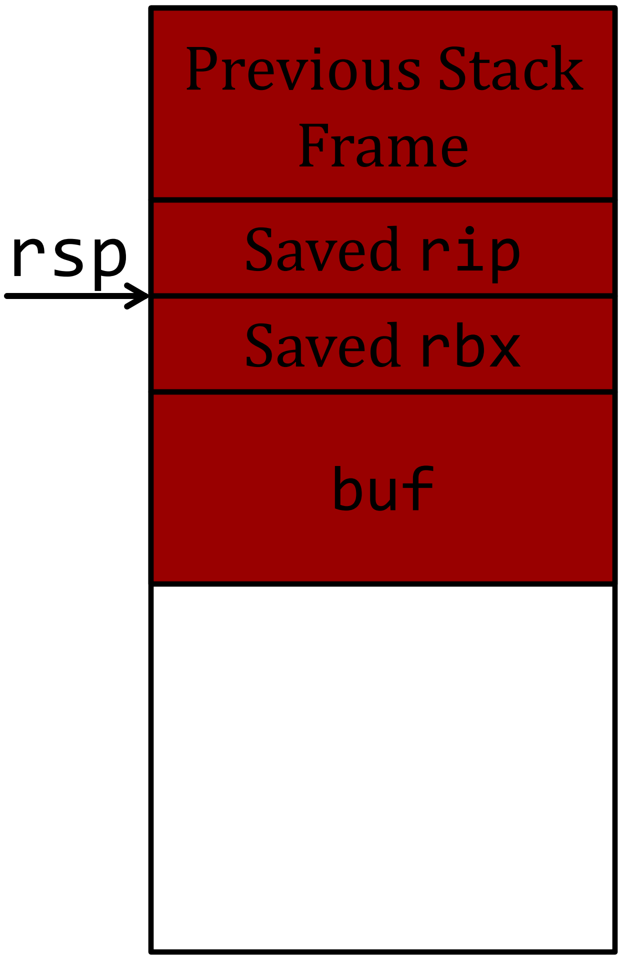

```
main:
    push    rbx
    sub     rsp,0x100
    mov     rsi,QWORD PTR [rsi+0x8]
    lea     rbx,[rsp]
    mov     rdi,rbx
    call    400440 <strcpy@plt>
    mov     rdi,rbx
    call    400450 <puts@plt>
    xor     eax,eax
    add     rsp,0x100
    pop     rbx
    ret
```

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 $rip \leftarrow Mem(rsp)$

$rsp \leftarrow rsp + 8$

Smashing the Stack

- In the previous example, we overwrote the saved instruction pointer
  - When `main` returns, control transfers to a location of the attacker's choosing
- Potential overwrite targets aren't limited to the IP
  - Procedure arguments
  - Frame pointer
  - User data

User Data Overwrites

```
void suid_cmd(const char * user_input) {
    uid_t uid;
    char buf[64];
    uid = get_nobody();
    strcpy(buf, user_input);
    setuid(uid);
    system(buf);
}
```

- What's the problem here?