CMPSC 447
Buffer Overflow Vulnerabilities

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Buffer Overflow

- Early example of a method to exploit a "memory error" in a C program
- Discovered in the 1970s
- Leveraged by the Morris Worm in 1988 – first large-scale exploit
- Leveraged by subsequent attacks in the early 2000s that led to security rethink
- Still a problem today – Check out CVEs for "buffer overflow"

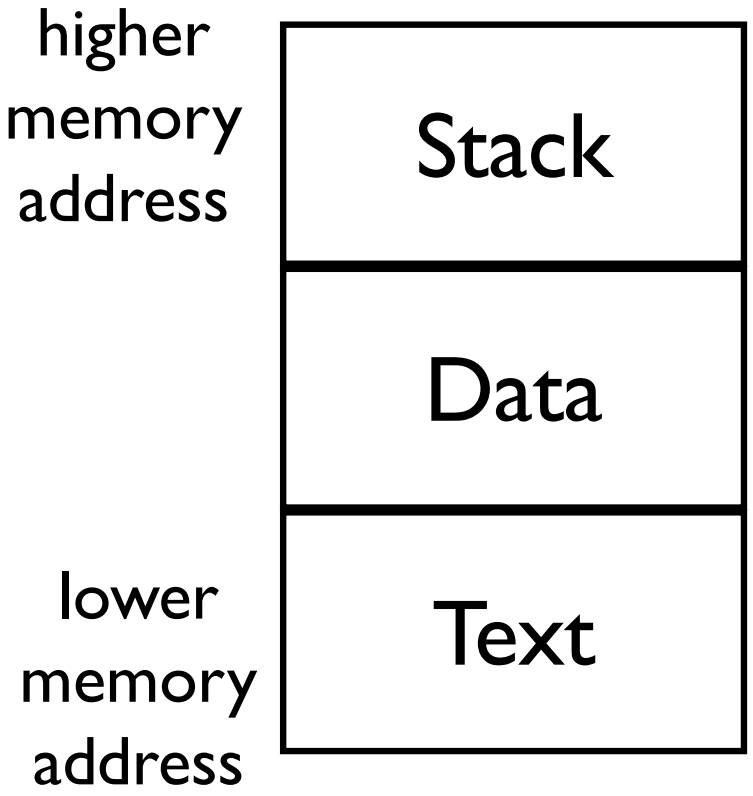
Memory Error

- A memory error allows a program statement to access memory outside of that allocated for the variables processed in the statement
- Common case: Buffer overflow
  - The C language allows writes to memory addresses specified by pointers
    - char buf[10] – *buf* can be used as a pointer
  - C functions enable writing based on the size of the input or a length value
    - strcpy and strncpy
  - However, does not ensure writes only within the buffer

Morris Worm

- Robert Morris, a 23-year old Cornell PhD student
  - Wrote a small (99 line) program
  - Launched on November 3, 1988
  - Simply disabled the Internet
- Used a buffer overflow in a program called *fingerd*
  - To get adversary-controlled code running
- Then spread to other hosts – cracked passwords and leveraged open LAN configurations
- Covered its tracks (set is own process name to *sh*, prevented accurate cores, re-forked itself)

Process Address Space



- Text: static code
- Data: also called heap
  - static variables
  - dynamically allocated data (malloc, new)
- Stack: program execution stacks

Program Stack

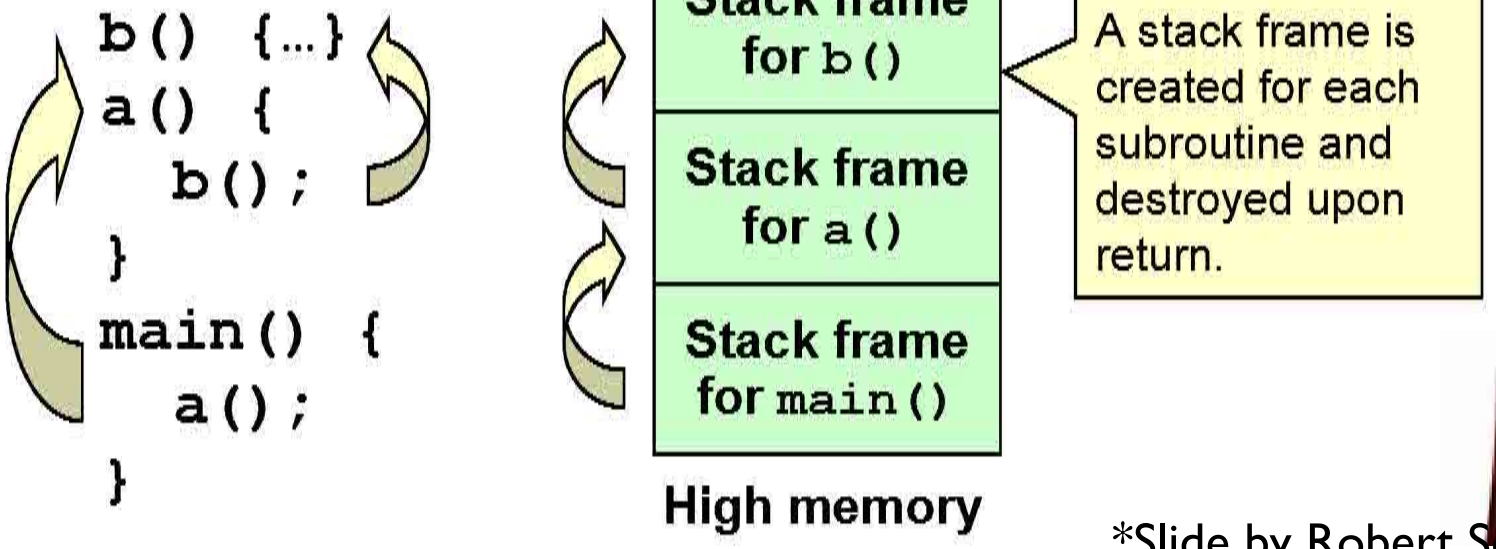
- For implementing procedure calls and returns
- Keep track of program execution and state by storing
  - local variables
  - arguments to the called procedure (callee)
  - return address of the calling procedure (caller)
  - ...

Program Stack

Stack Segment

The stack supports nested invocation calls

Information pushed on the stack as a result of a function call is called a frame



*Slide by Robert Seacord

Stack Frames

PENNSTATE 1855

- Stack grows from high mem to low mem addresses
- The stack pointer points to the current "top of the stack" – last thing pushed on the stack (that matters)
  - ESP in Intel architectures
- The frame pointer points to the start of the current frame
  - also called the base pointer
    - EBP in Intel architectures
- The stack is modified during
  - function calls, function prologue, function epilogue and operations on stack variables (locals and args)

A Running Example

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

Run "gcc –S –o example.s example.c" to see its assembly code

Function Calls

function (1,2)

```
pushl $2
pushl $1
call function
```

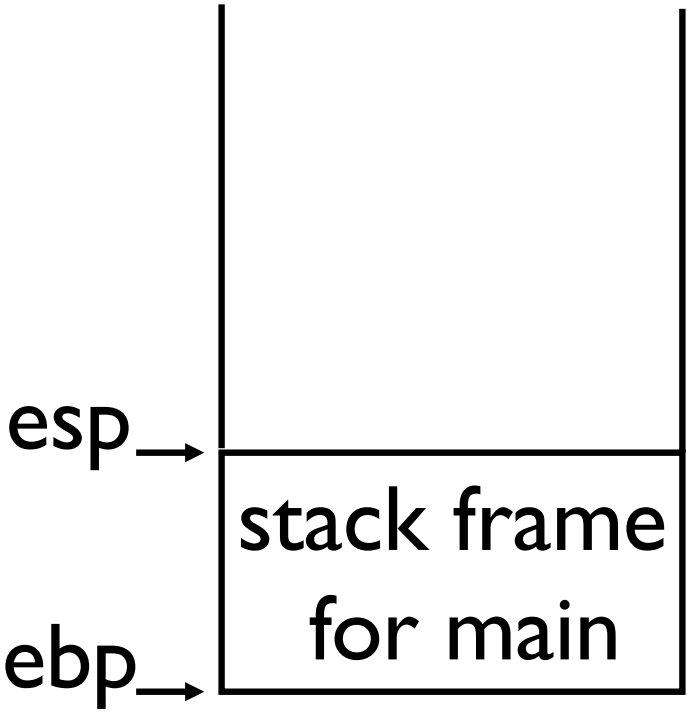
push the 2nd arg to stack

push the 1st arg to stack

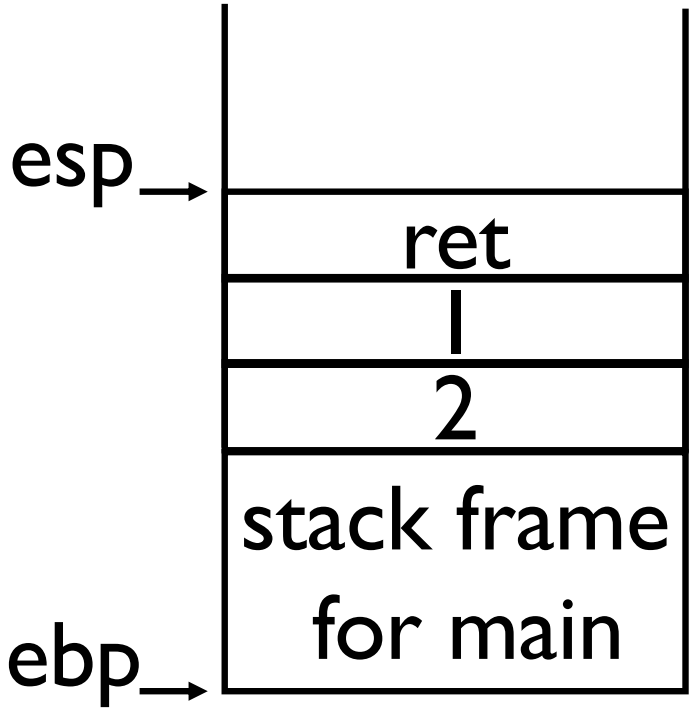
push the ret addr onto the stack, and jumps to the function

Function Calls: Stacks

Before



After



Function Initialization

PENNSTATE 1855

```
void function(int a, int b) {
```

```
pushl %ebp
movl %esp, %ebp
subl $12, %esp
```

saves the prior frame pointer

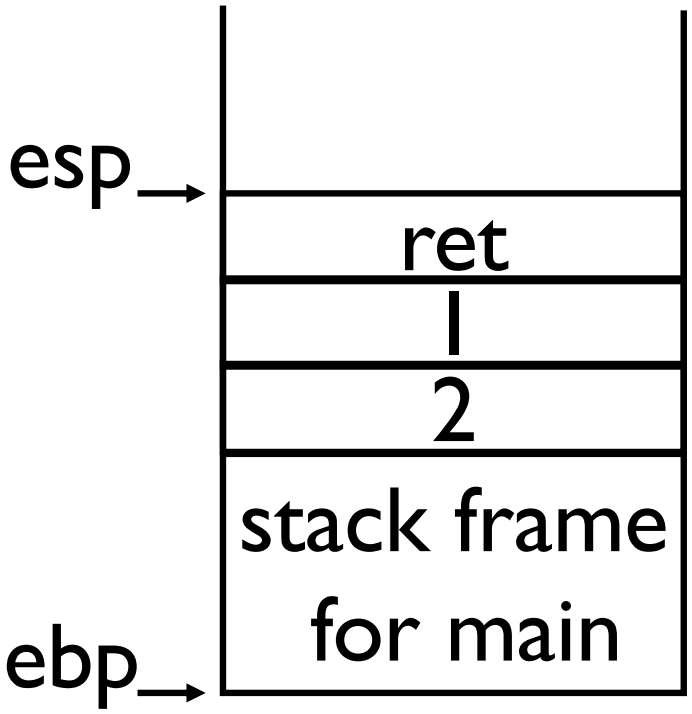
sets the new frame pointer

allocate space for local variables

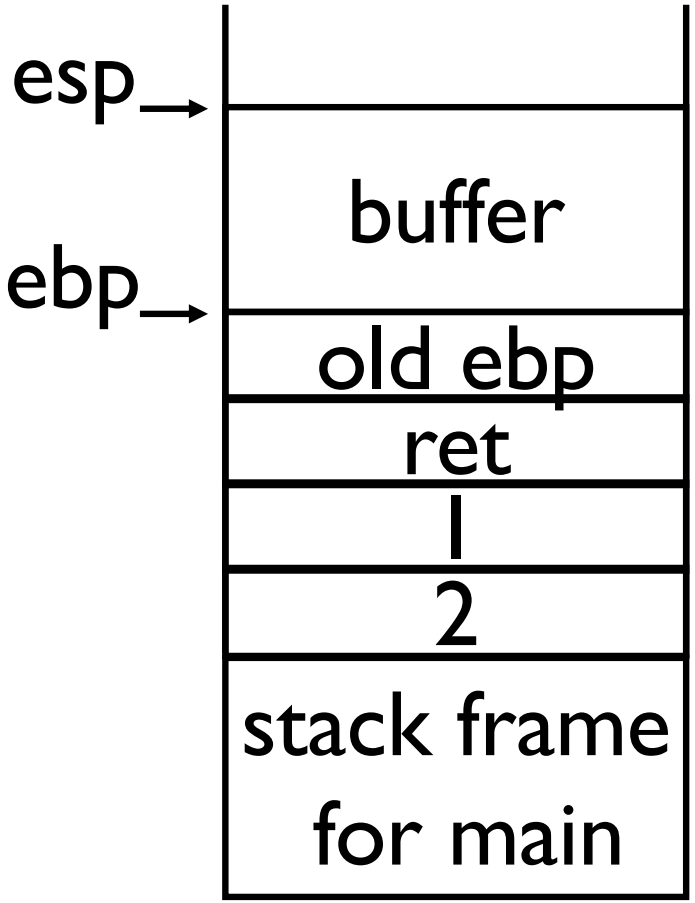
Function prologue

Function Initialization: Stacks

Before



After



Function Return

return;

```
movl %ebp, %esp
popl %ebp
ret
```

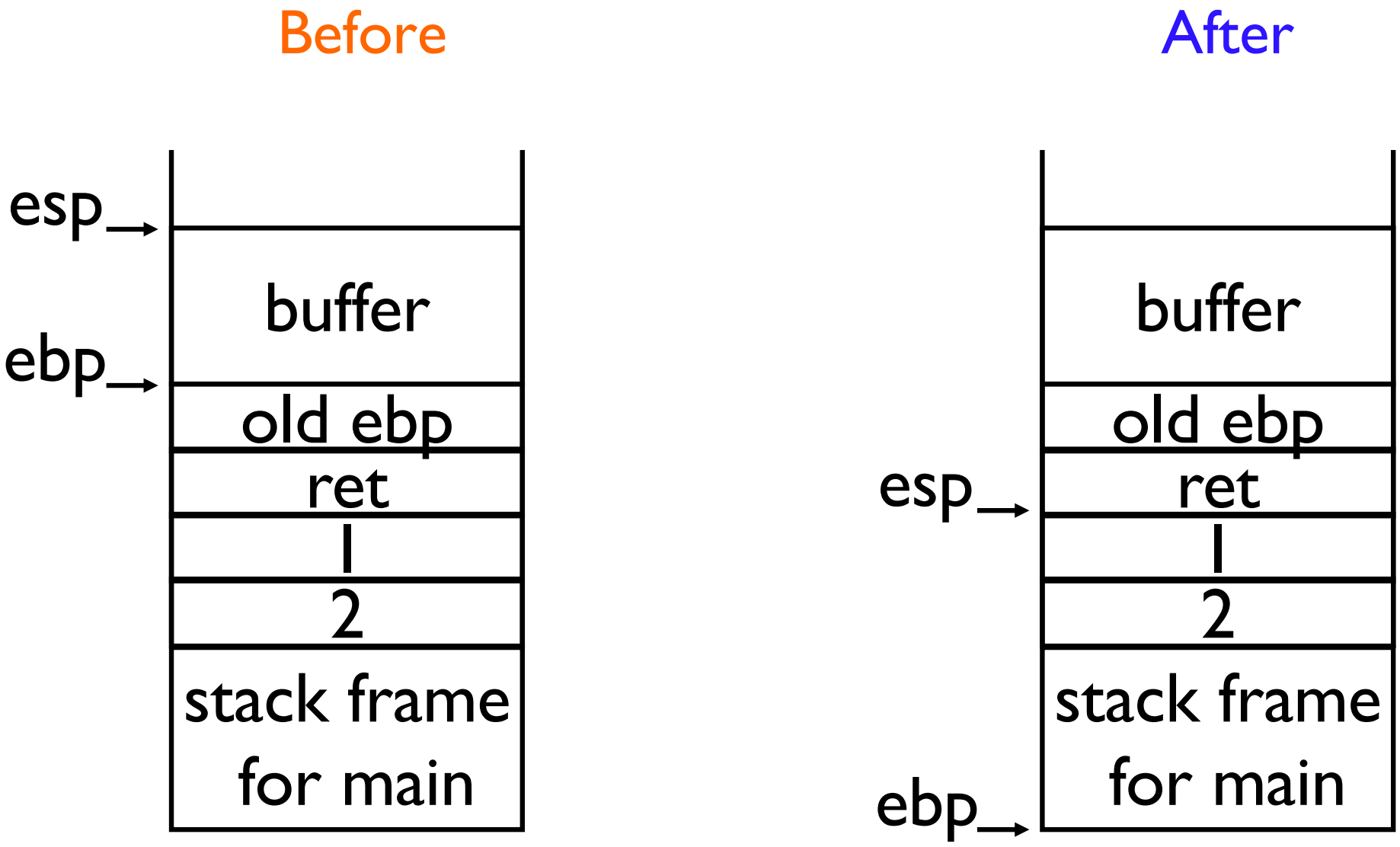
restores the old stack pointer

restores the prior frame pointer

gets the return address at current stack pointer, and jumps to it

Function epilogue

Function Return: Stacks



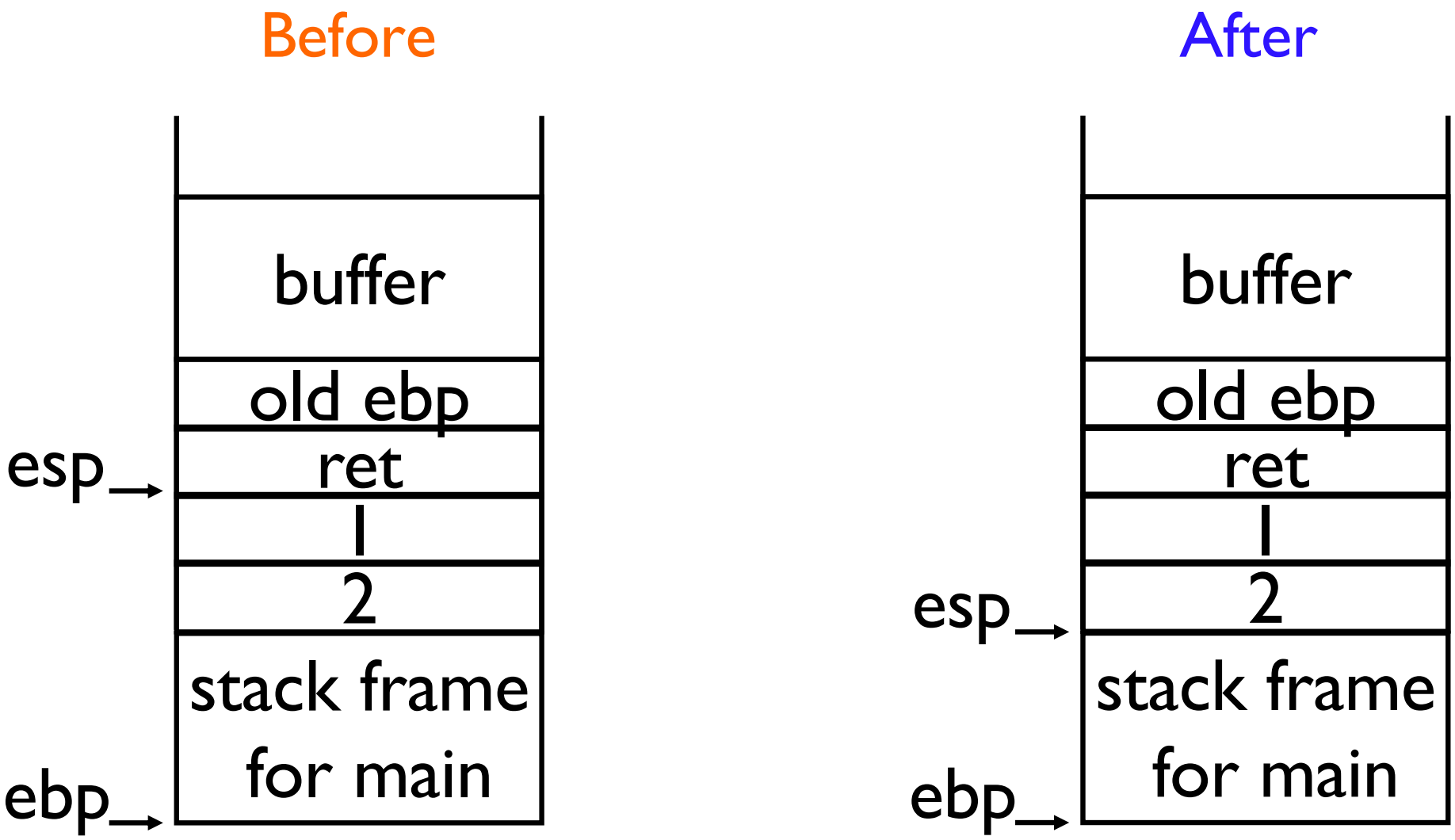
Return to Calling Function

In main again – following return…

```
pushl $2
pushl $1
call function
addl $8, %esp
```

restores the stack pointer for caller

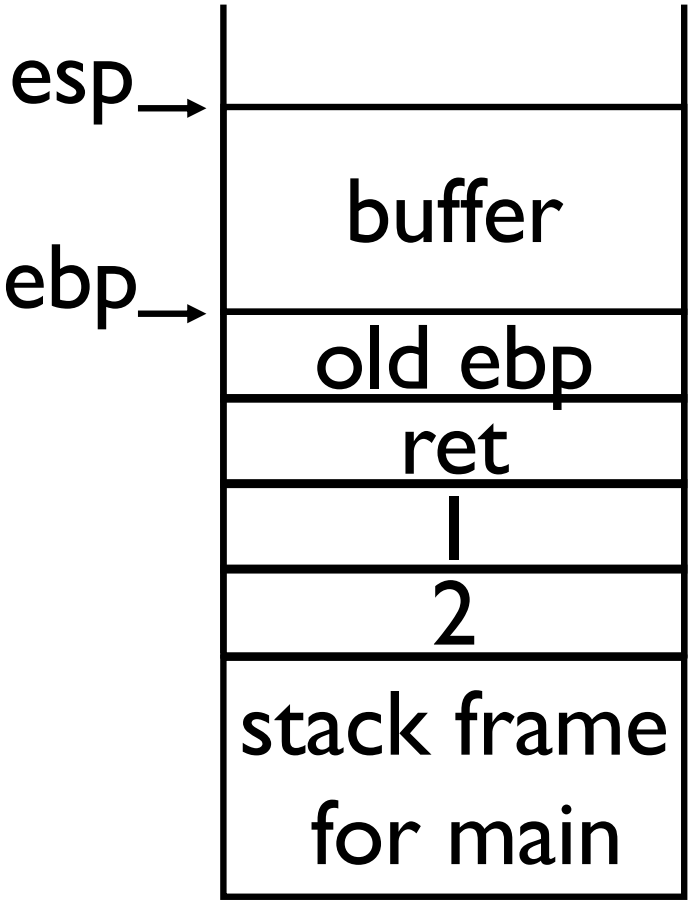
Return to Calling Function: Stacks



A Running Example

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

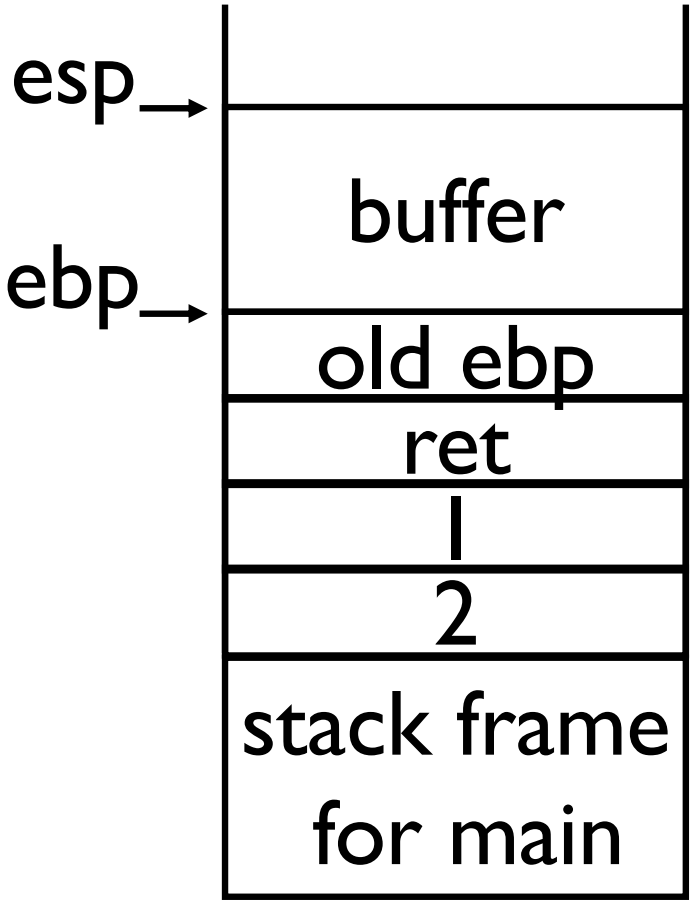


Overwriting the Return Address

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);

    int* ret = (int *)buffer+?;
    *ret = ?;

    return;
}
```



Overwriting the Return Address

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);

    int* ret = (int *) buffer+16;
    *ret = *ret + 1;   // assuming one-byte store

    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

The output will be 0

the original return address

the new return address

Previous Attack

- Not very realistic
  - Attackers are usually not allowed to modify code
  - Threat model: the only thing they can affect is the input
  - Can they still carry out similar attacks?
    - YES, because of possible buffer overflows

Buffer Overflows

PENNSTATE 1855

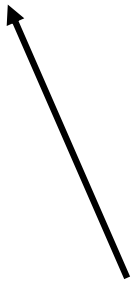
- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure (buffer)
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify memory after buffer
  - Stack: return address, local variables, function pointers, etc.
  - Heap: data structures and metadata (next time)
- Also, a buffer underflow to modify memory prior

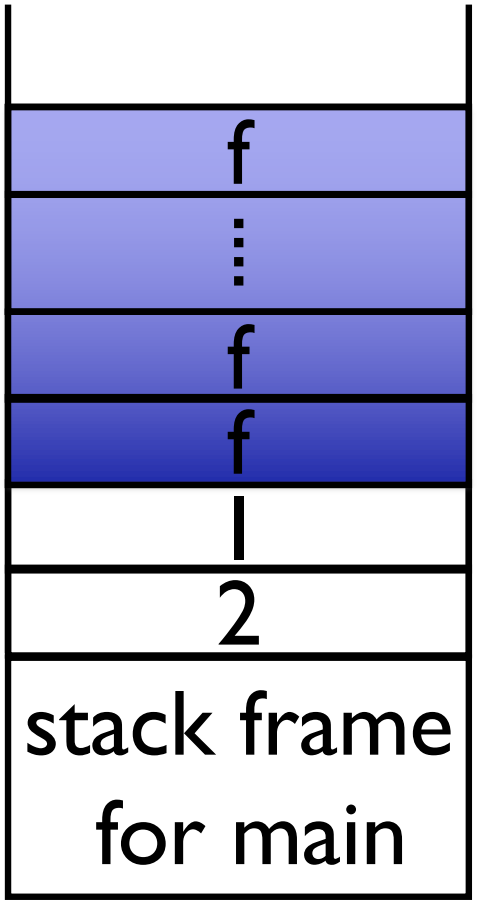
Smashing the Stack

- Occurs when a buffer overflow overwrites other data in the program stack
- Successful exploits can overwrite the return address on the stack enabling the execution of arbitrary code on the targeted machine
- What happens if we input a large string?
- ./example
  - ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
- Segmentation fault – why is that?

What Happened?

```
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}
```



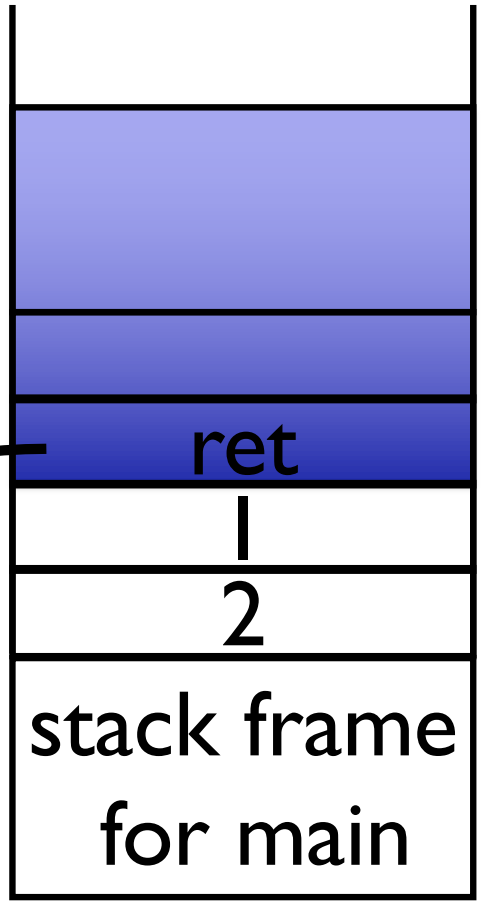


If the input is large, then gets(buffer) will write outside the bound of buffer, and the return address is overwritten – with "ffff" (in ASCII), which likely is not a legal code address – seg fault

Figure Out A Nasty Input

```
void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```



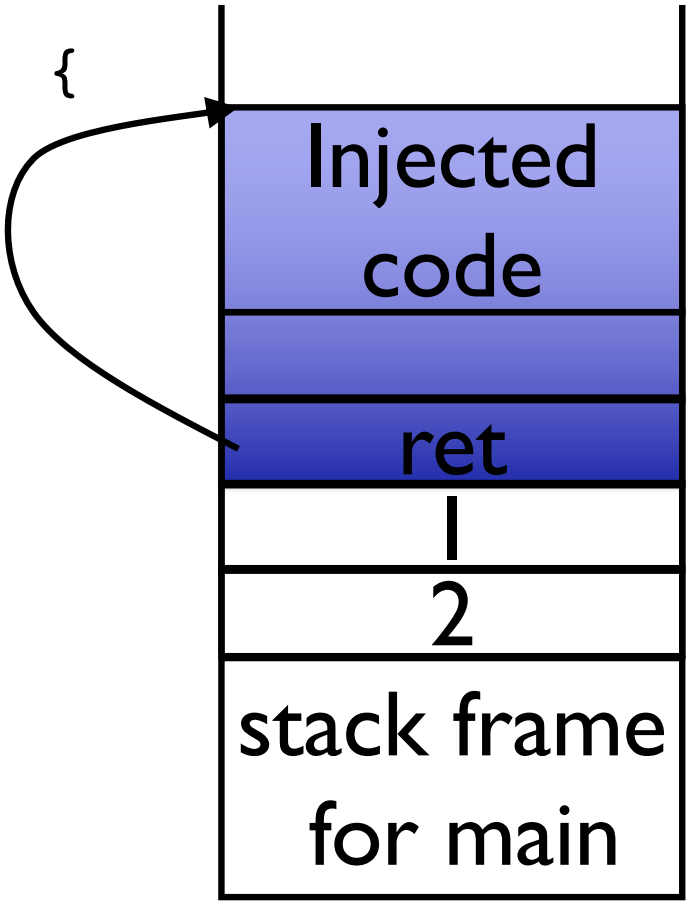
A nasty input puts the return address after x=1.

"Arc" injection – new control flow

Injecting Code

```
void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

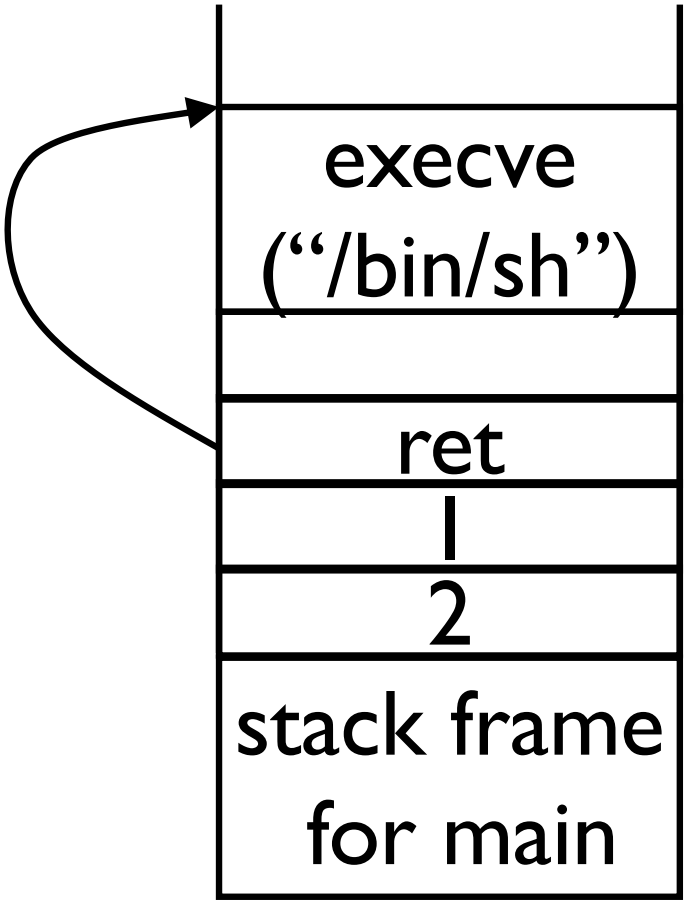


The injected code can do anything.
E.g., download and install a worm

Code Injection

- Attacker creates a malicious argument—a specially crafted string that contains a pointer to malicious code provided by the attacker
- When the function returns, control is transferred to the malicious code
  - Injected code runs with the permission of the vulnerable program when the function returns.
  - Programs running as root or other elevated privileges are normally targeted
    - Programs with the setuid bit on

Injecting Shell Code



- This brings up a shell (logical view – real later)
- Adversary can execute any command in the shell
- The shell has the same privilege as the process
- Often, a process with the root privilege is attacked

Injecting Shell Code

- How do you invoke "execve" using injected code?

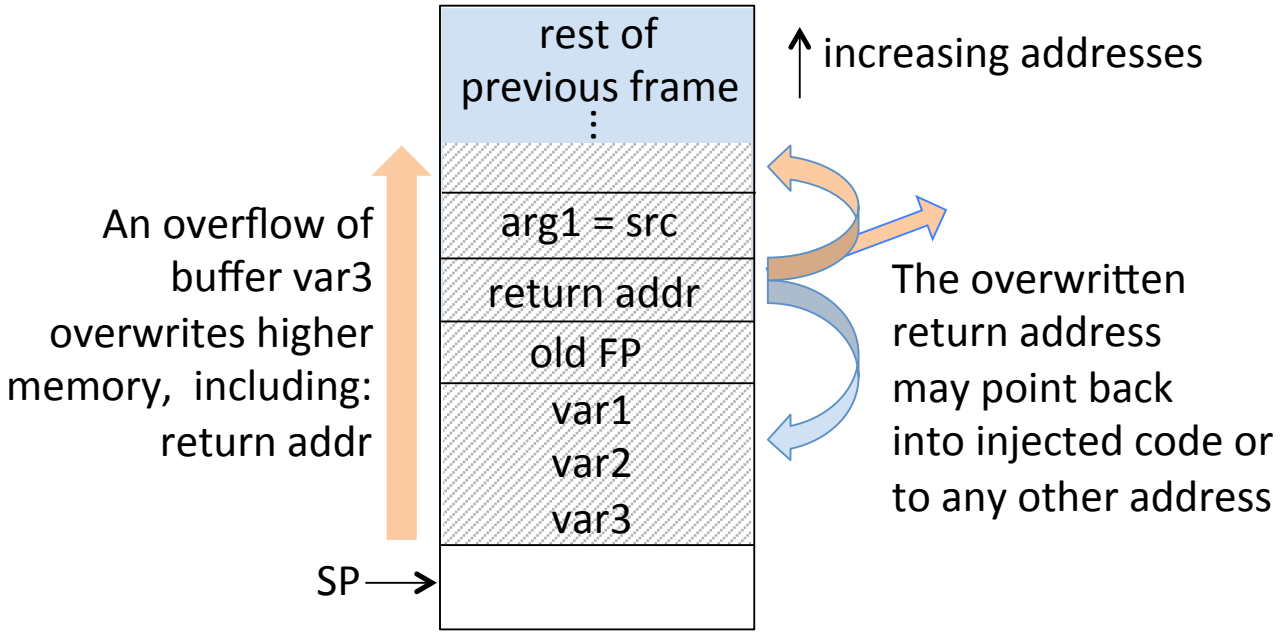


Figure 6.5: Buffer overflow of stack-based local variable.

Injecting Shell Code

- Inject the address of the "execve" function at the return address or elsewhere in stack reference by the return address
  - "execve" is a function in *libc* that is dynamically linked into the process address space
- To invoke a function in a library it must be able to find that address itself as well
- How is that done? Your program calls "execve" thru a stub (procedure linkage table), which retrieves the address set at link time (in the global offset table)

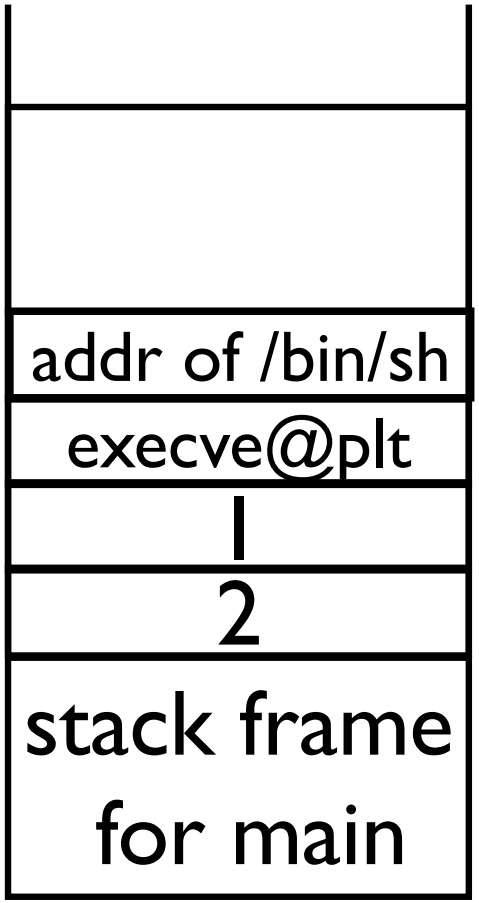
Injecting Shell Code

- Example of PLT code (from objdump -dl)

```
0x08048730 <execve@plt>:
 8048730:    ff 25 1c d1 04 08          jmp    *0x804d11c
 8048736:    68 28 00 00 00          push   $0x28
 804873b:          e9 90 ff ff ff      jmp    80486d0

0x08048740 <strncpy@plt>:
 8048740:    ff 25 20 d1 04 08          jmp    *0x804d120
 8048746:    68 30 00 00 00          push   $0x30
 804874b:          e9 80 ff ff ff      jmp    80486d0
```

Injecting Shell Code



- Overwrite return address with address of code to run next (e.g., execve@plt)
  - What address?
- Provide argument(s) above – pointer to "/bin/sh" command
  - Where to put it?
- And then "null" for last arg (env)

Any C(++) code acting on untrusted input is at risk

PENNSTATE 1855

- Code taking input over untrusted network
  - E.g., sendmail, web browser, wireless network driver,...
- Code taking input from untrusted user on multi-user system,
  - esp. services running with high privileges (as ROOT on Unix/Linux, as SYSTEM on Windows)
- Code processing untrusted files
  - that have been downloaded or emailed
- Also embedded software, e.g., in devices with (wireless) network connection such as mobile phones with Bluetooth, wireless smartcards in new passport or OV card, airplane navigation systems, ...

Take Away

- Memory errors enable processes to write to memory outside the expectation range
- The classic example is the buffer overflow, which is still a common attack vector today
- A buffer overflow vulnerability allows an adversary to overwrite the memory beyond the buffer on the stack
  - But runtime state is also on the stack – return address
- We discussed methods to inject and reuse code
- Available defenses are not complete