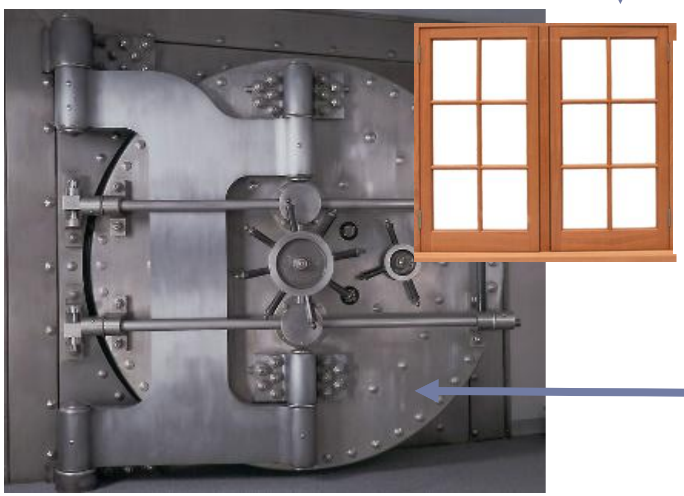
SC3010
Computer Security

Lecture 2: Software Security (I)

Basic Concepts in Software Security

Vulnerability: a *weakness* which allows an attacker to reduce a system's information assurance.



Software system

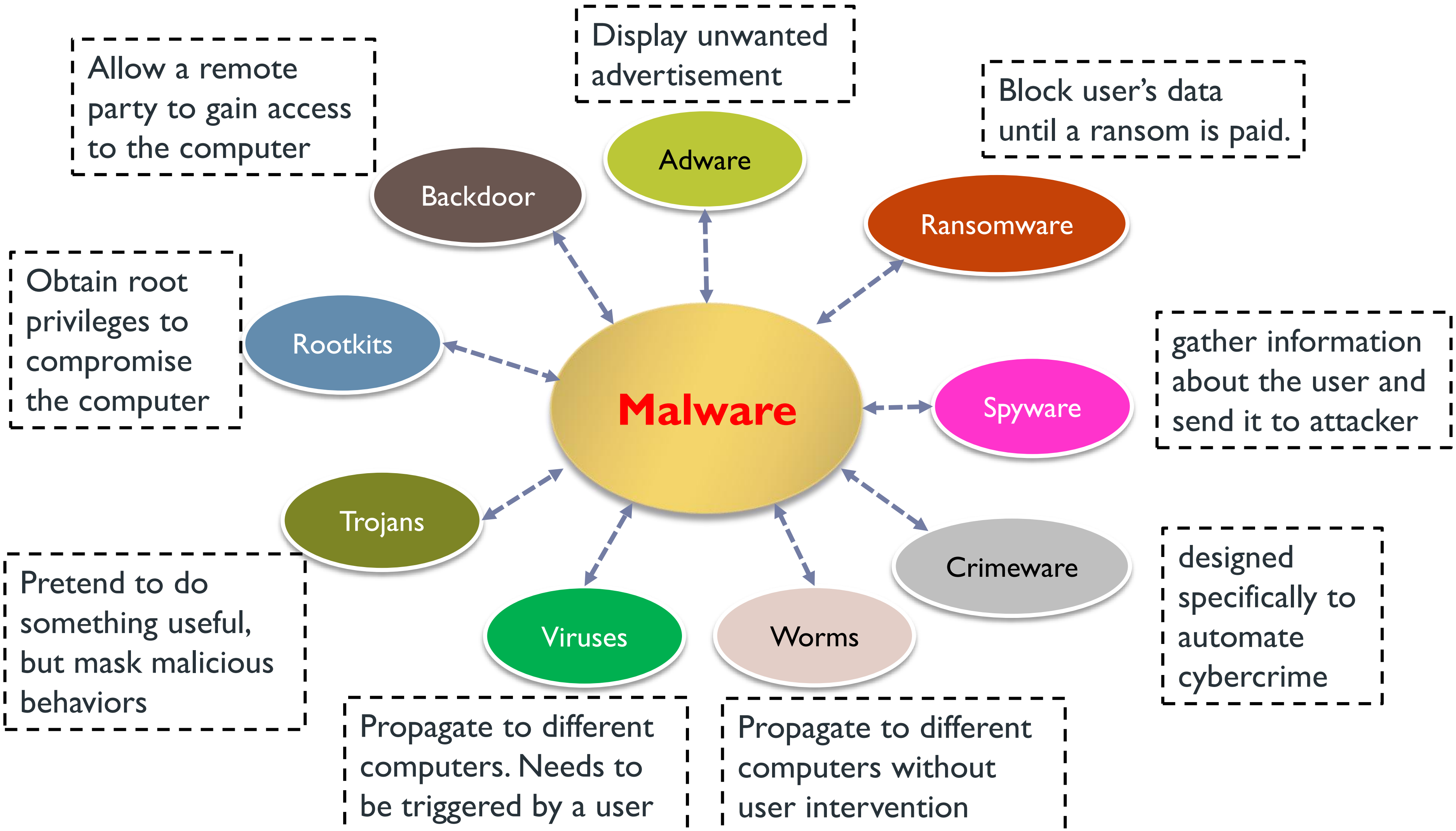
Exploit: a *technique* that takes advantage of a vulnerability, and used by the attacker to attack a system

Payload: a *custom code* that the attacker wants the system to execute

Different Kinds of Vulnerabilities

Memory safety violations	• Buffer overflows and over-reads • Dangling pointers
Input validation errors	• Format string attacks • SQL injection • Code injection • Cross-site scripting in web applications
Race conditions	• Time-of-check-to-time-of-use bugs • Symlink races
Privilege-confusion bugs	• Cross-site request forgery in web applications • Clickjacking • FTP bounce attack
Privilege escalation	• Privilege escalation
Side-channel attack	• Timing attack

Different Kinds of Malware



Why Does Software Have Vulnerabilities

Human factor

- Programs are developed by humans. Humans make mistakes
- Programmers are not security-aware
- Misconfigurations could lead to exploit of software vulnerabilities

Language factor

- Some programming languages are not designed well for security
  - Mainly due to more flexible handling of pointers/references.
  - Lack of strong typing.
  - Manual memory management. Easier for programmers to make mistakes.

Outline

- **Review: Memory Layout and Function Call Convention**
- **Buffer Overflow Vulnerability**

Outline

- **Review: Memory Layout and Function Call Convention**
- Buffer Overflow Vulnerability

Memory Layout of a Program (x86)

Code

- The program code: fixed size and read only

Static data

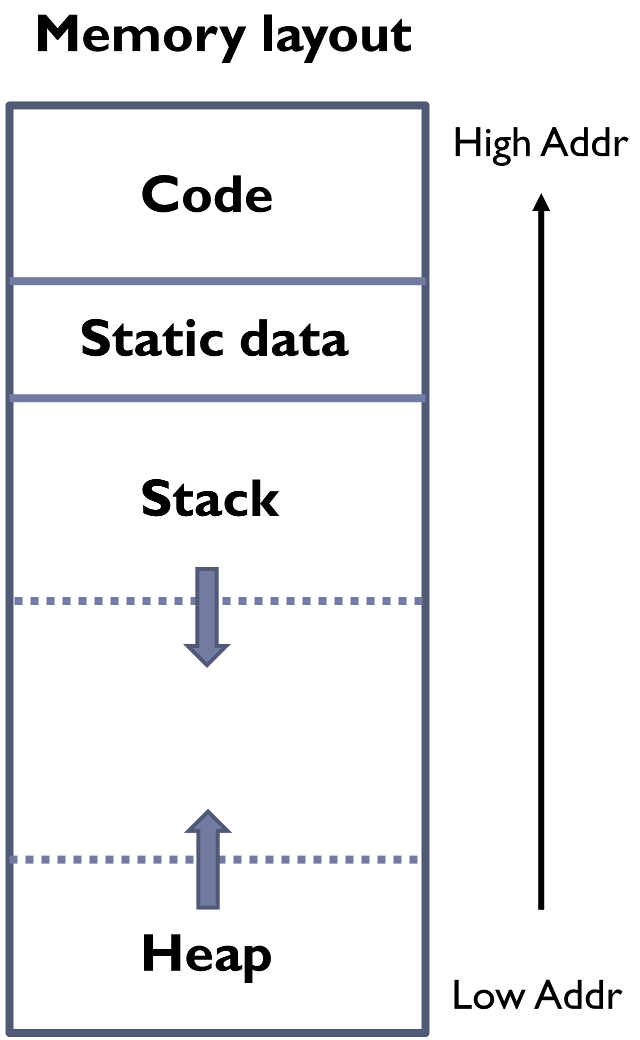
- Statically allocated data, e.g., variables, constants

Stack

- Parameters and local variables of methods as they are invoked.
- Each invocation of a method creates one frame which is pushed onto the stack
- Grows to lower addresses

Heap

- Dynamically allocated data, e.g., class instances, data array
- Grows towards higher addresses



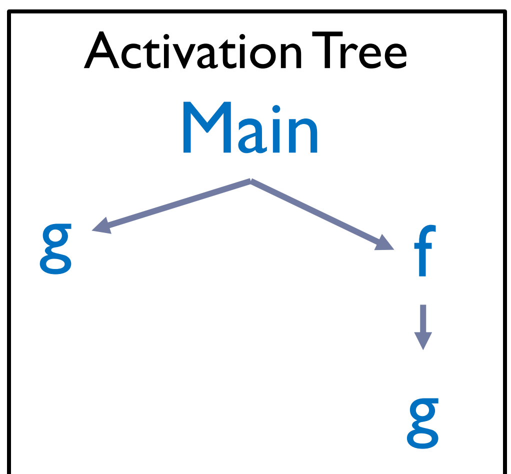
Stack

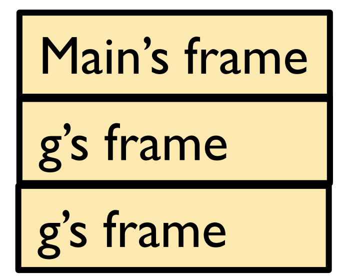
Store local variables (including method parameters) and intermediate computation results

A stack is subdivided into multiple **frames**:

- <u>A method is invoked</u>: a new frame is pushed onto the stack to store local variables and intermediate results for this method;
- <u>A method exits</u>: its frame is popped off, exposing the frame of its caller beneath it

```
Main( ) {
  g( );
  f( );
}
f( ) {
  return g( );
}
g( ) {
  return 1;
}
```





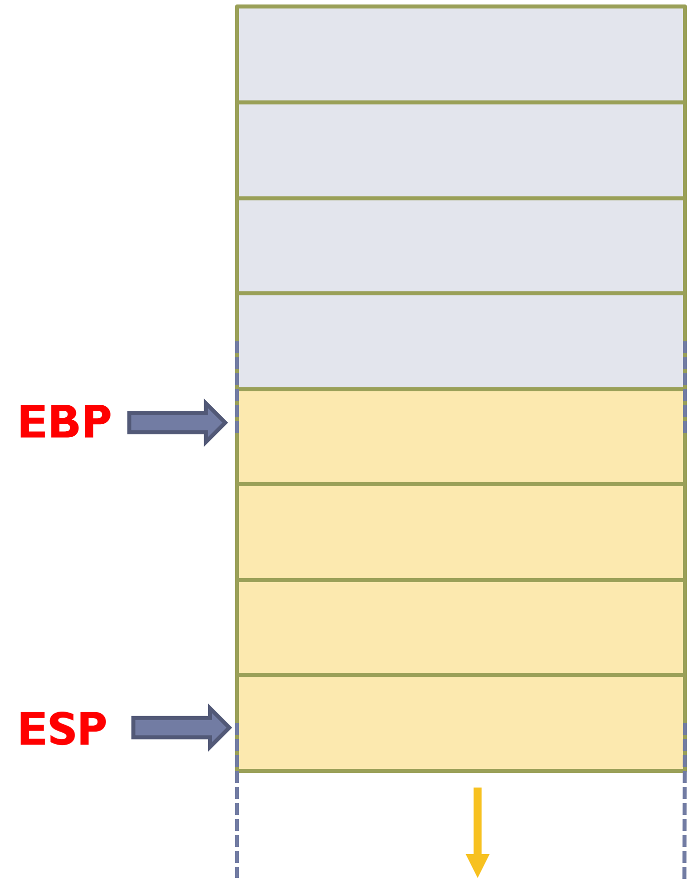
Inside a Frame for One Function

Two pointers:

- **EBP**: base pointer. Fixed at the frame base
- **ESP**: stack pointer. Current pointer in frame (current lowest value on the stack)

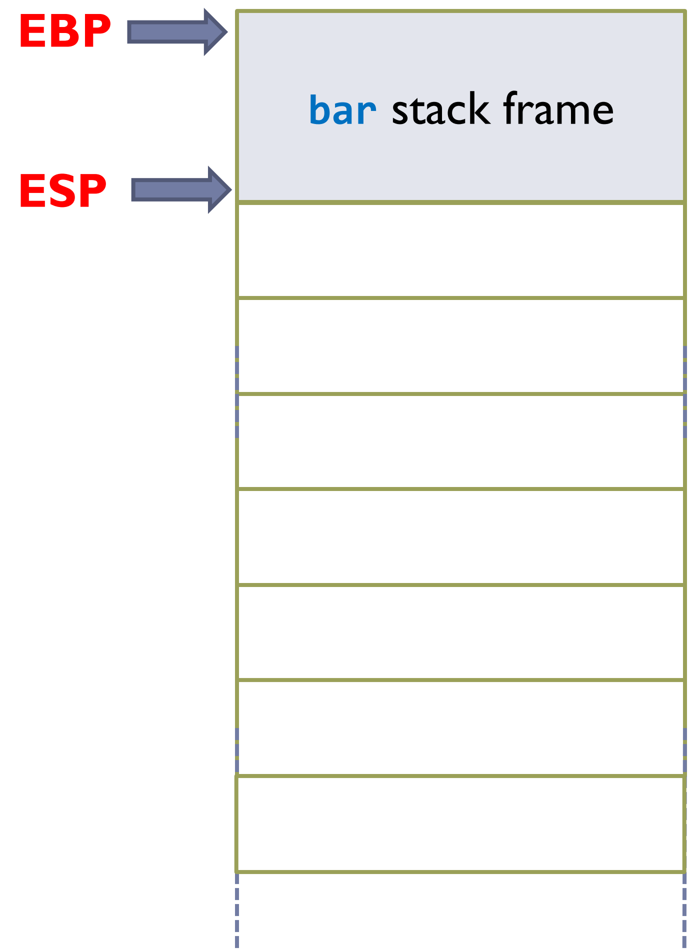
A frame consists of the following parts:

- Function parameters
- Return address of the caller function
  - When the function is finished, execution continues at this return address
- Base pointer of the caller function
- Local variables
- Intermediate operands



Function Call Convention

Initially: EBP and ESP point to the top and bottom of the bar stack frame.

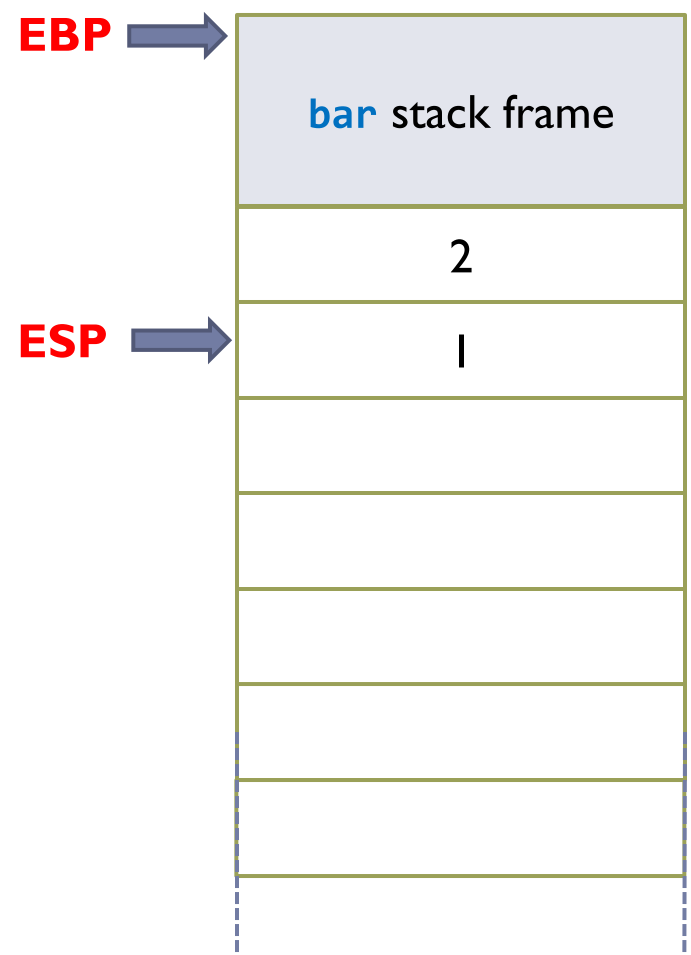



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 1: Push function parameters to the stack.

- Function parameters are stored in reverse order.
- ESP is updated to denote the lowest stack location due to the push operation.

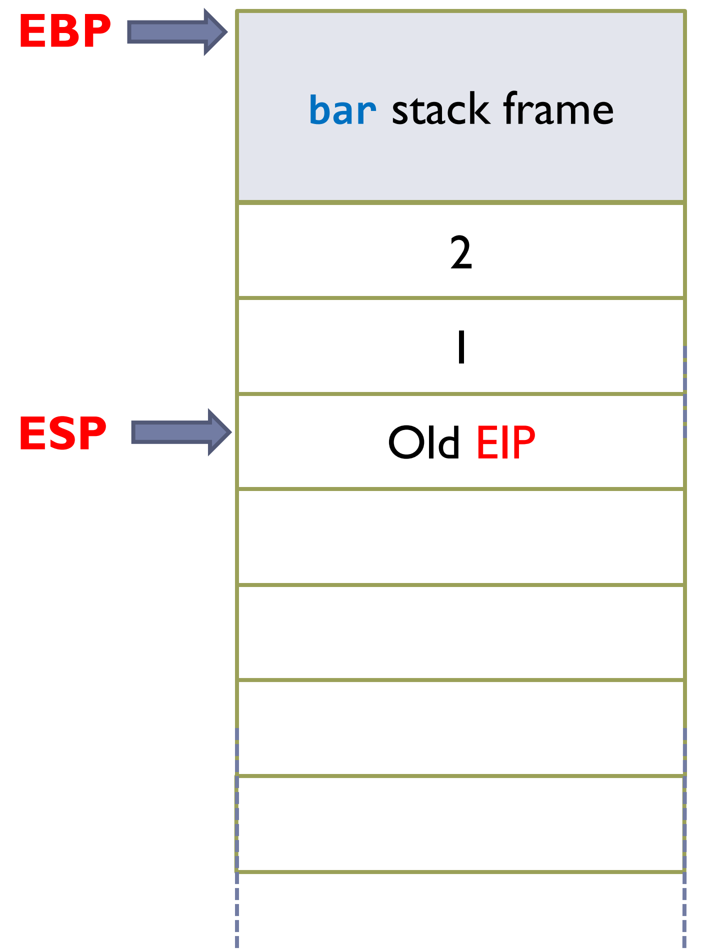



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 2: Push the current instruction pointer (EIP) to the stack.

- This is the return address in function **bar** after we finish function **foo**.
- ESP is updated to denote the lowest stack location due to the push operation.

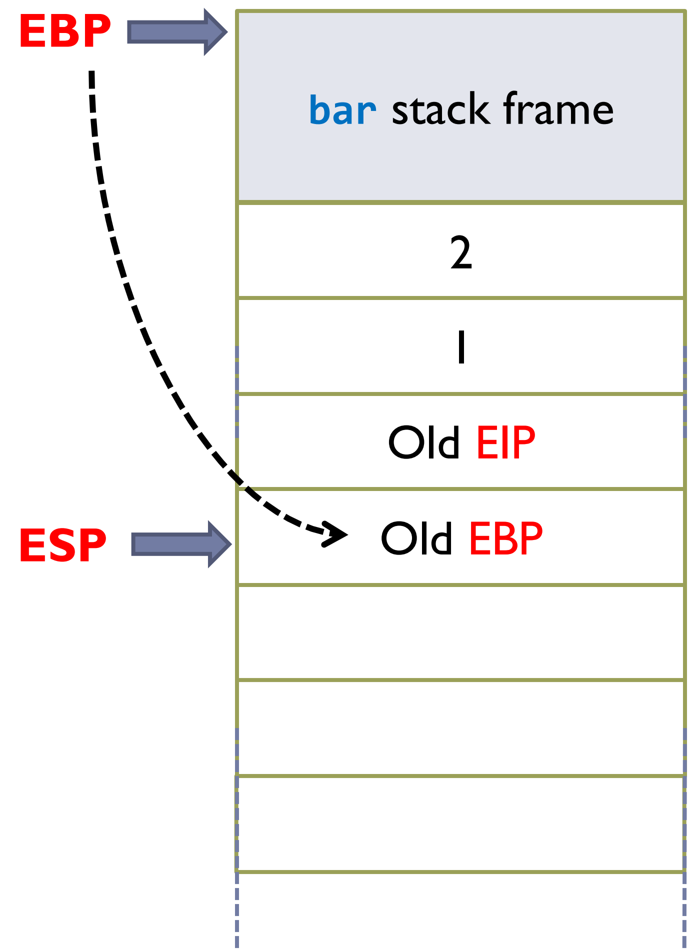



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 3: Push the EBP of function bar to the stack.

- This can help restore the top of function **bar** stack frame when we finish function **foo**.
- ESP is updated to denote the lowest stack location due to the push operation.

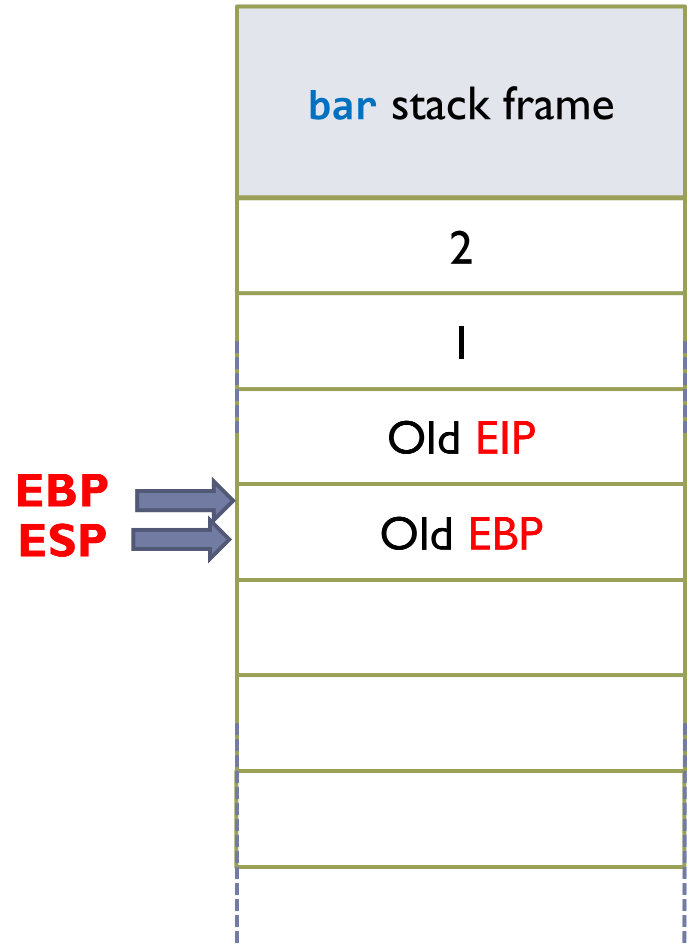



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 4: Adjust EBP for function foo stack frame.

- Move EBP to ESP of bar stack frame



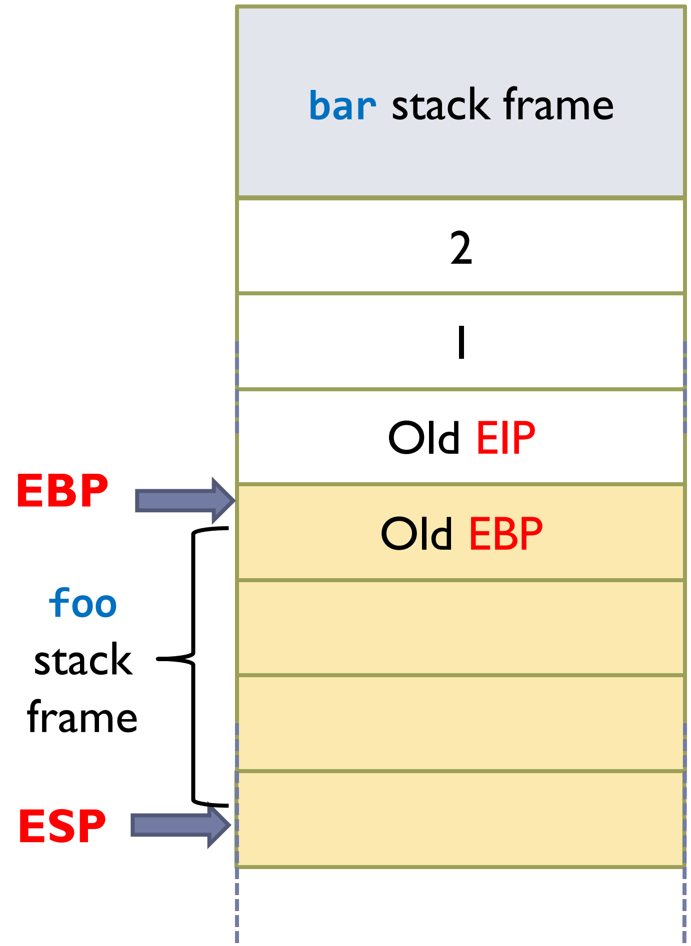

```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 5: Adjust ESP for function foo stack frame.

- Move ESP to some location below to create a new stack frame for function foo
- The stack space for function foo is pre-calculated based on the source code. It is used for storing the local variables and intermediate results.

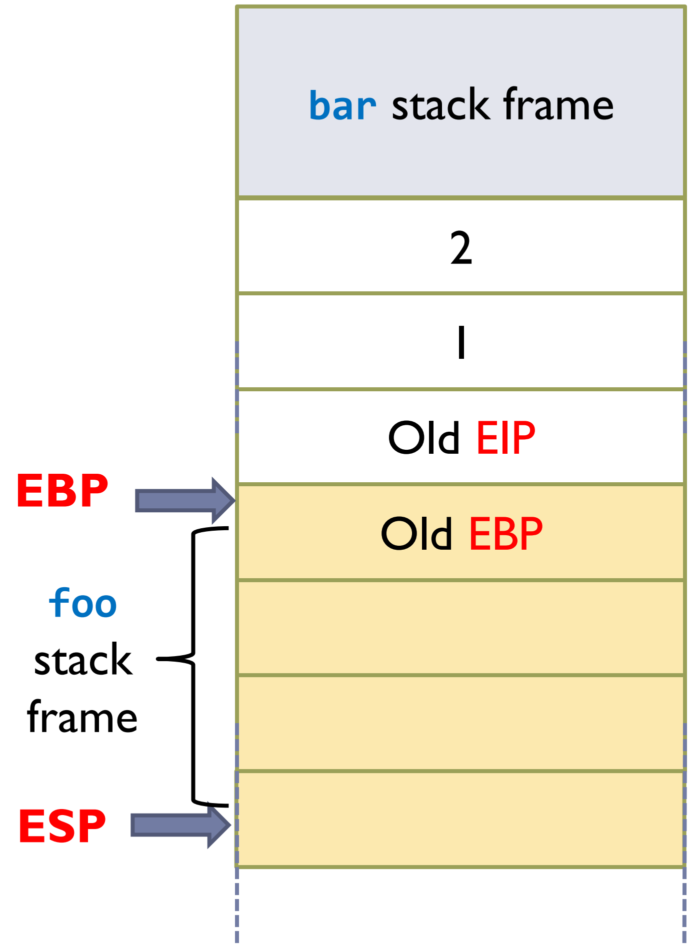
```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```



Function Call Convention

Step 6: Execute function foo within its stack frame.

- The returned result will be stored in the register EAX.

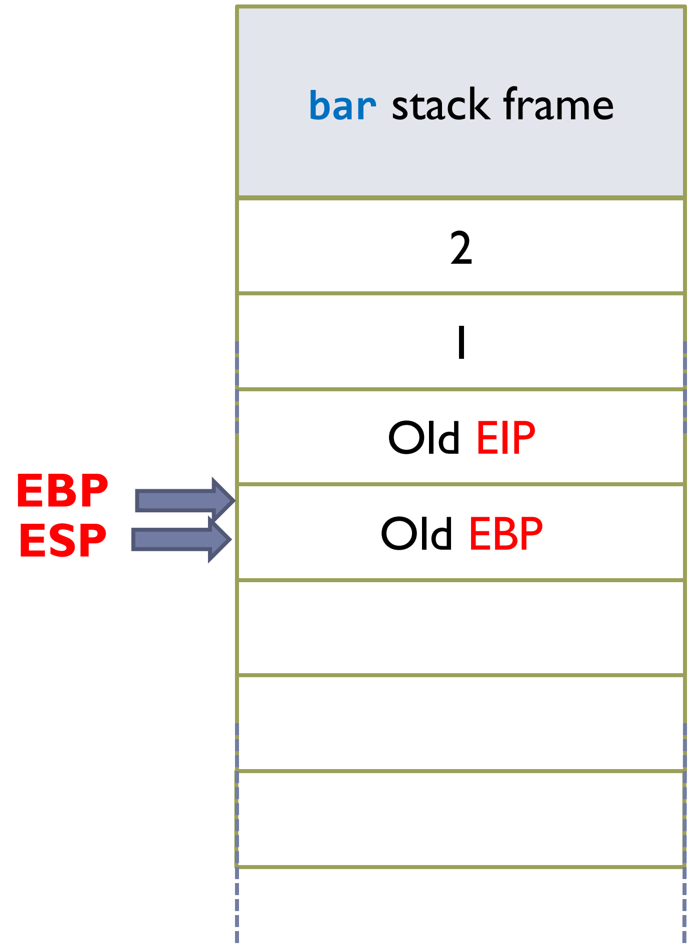



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 7: Adjust ESP.

- Move ESP to EBP
- This deletes the stack space allocated for function foo.

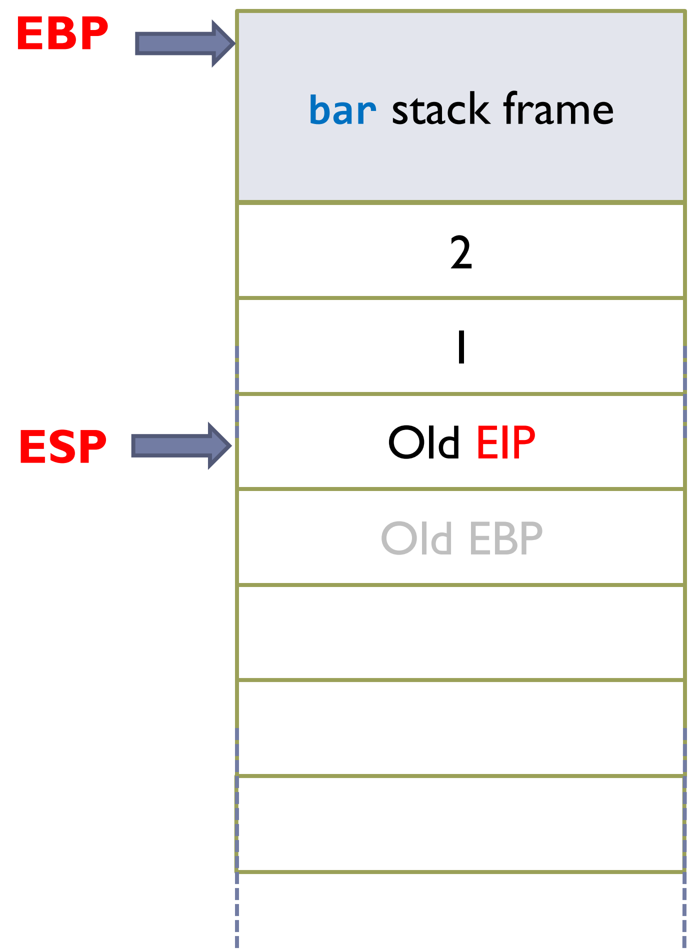



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 8: Restore EBP.

- Pop a value from the stack (old EBP), and assign it to EBP.
- ESP is also updated (old EIP) due to the pop operation.
- (old EBP) is deleted from the stack.

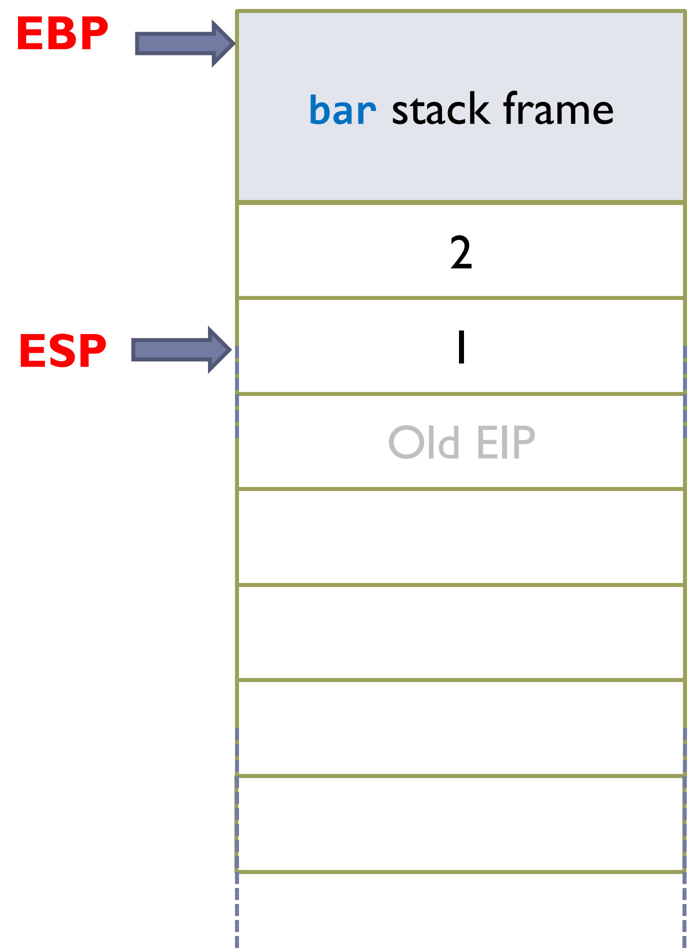



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 9: Restore EIP.

- Pop a value from the stack (old EIP), and assign it to EIP.
- ESP is also updated (1) due to the pop operation.
- (old EIP) is deleted from the stack.

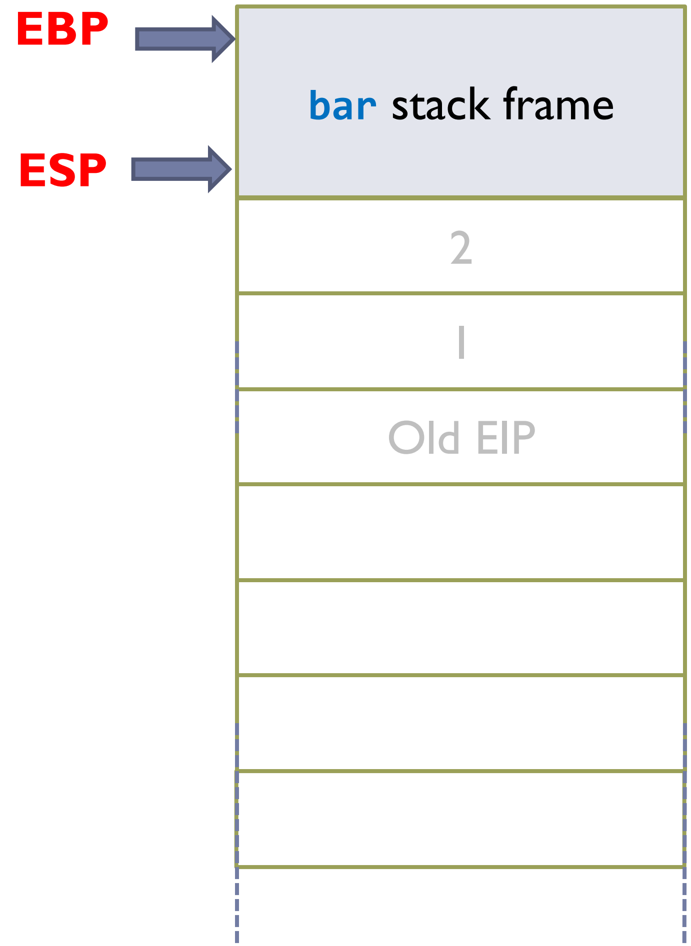



```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Function Call Convention

Step 10: Delete function parameters.

- Pop values from the stack (1, 2).
- ESP is also updated (old ESP) due to the pop operation.
- Function parameters (1, 2) are deleted from the stack.
- Continue the execution in function **bar**.




```
void bar( ) {
  foo(1, 2);
}
int foo(int x, int y){
  int z = x + y;
  return z;
}
```

Outline

- Review: Memory Layout and Function Call Convention
- **Buffer Overflow Vulnerability**

A Common Vulnerability in C Language

String

- An array of characters (1 Byte).
- Must end with NULL (or `\0’). A string of length n can hold only $n-1$ characters, while the last character is reserved for NULL.

`char* strcpy (char* dest, char* src)`

- Copy string *src* to *dest*
- No checks on the length of the destination string.

What if the source string is larger than destination string?

```
char str[6] = “Hello”;
```

	X
	X
[5]	\0
[4]	o
[3]	l
[2]	l
[1]	e
[0]	H

```
char* strcpy (char* dest, const char* src) {
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];
  dest[i] = '\0';
  return dest;
}
```

General Idea

More data into a memory buffer than the capacity allocated.

Overwriting other information adjacent to that memory buffer.

Key reason: C does not check boundaries when copying data to the memory.

High coverage

Any system implemented using C or C++ can be vulnerable.

- Program receiving input data from untrusted network

  sendmail, web browser, wireless network driver, ...
- Program receiving input data from untrusted users or multi-user systems

  services running with high privileges (root in Unix/Linux, SYSTEM in Windows)
- Program processing untrusted files

  downloaded files or email attachment.
- Embedded software

  mobile phones with Bluetooth, wireless smartcards, airplane navigation systems, …



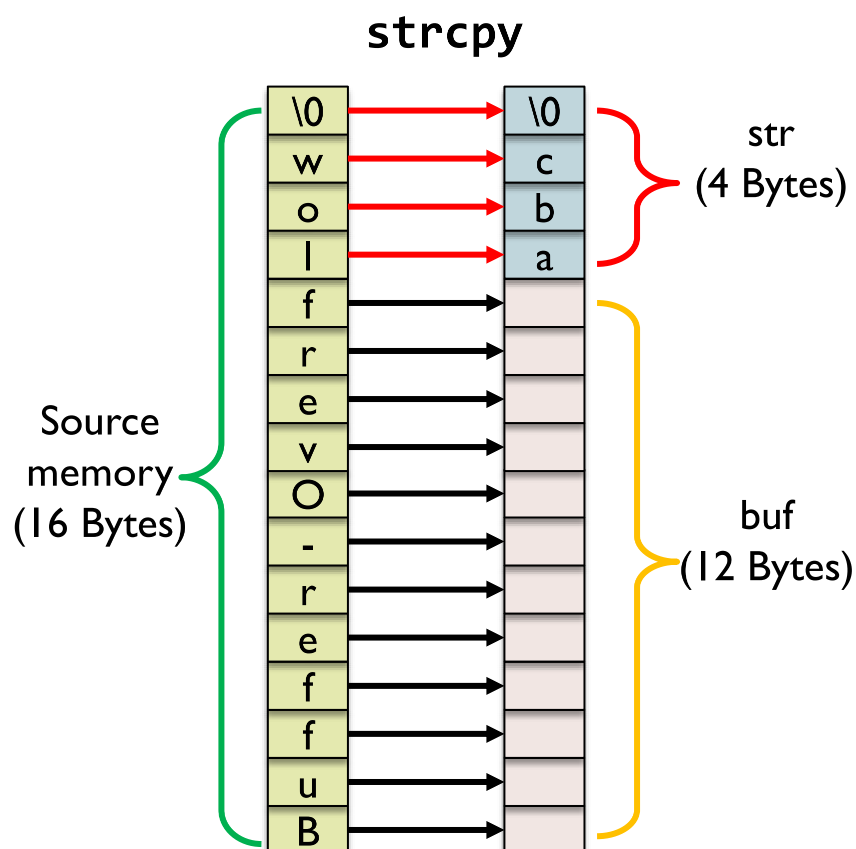





Example of Buffer Overflow

Corruption of program data

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

int main(int argc, char* argv[]) {
    char str[4] = "abc";
    char buf[12];
    strcpy(buf, "Buffer-Overflow");
    printf("str is %s\n",str);
    return 0;
}
```



Potential Consequences

```
int Privilege-Level = 3;
char buf[12];
strcpy(buf, "......");
```

Privilege escalation

```
int Authenticated = 0;
char buf[12];
strcpy(buf, "......");
```

Bypass authentication

```
char command[] = "/usr/bin/ls";
char buf[12];
strcpy(buf, "......");
execv(command, ...);
```

Execute arbitrary command

```
int (*foo)(void);
char buf[12];
strcpy(buf, "......");
foo();
```

Hijack the program control

……

More Vulnerability Functions

```
char* strcat (char* dest, char* src)
```

- Append the string *src* to the end of the string *dest.*

```
char* gets (char* str)
```

- Read data from the standard input stream (stdin) and store it into *str*.

```
int* scanf (const char* format, …)
```

- Read formatted input from standard input stream.

```
int sprintf (char* str, const char* format, …)
```

- Create strings with specified formats, and store the resulting string in *str*.

and more…

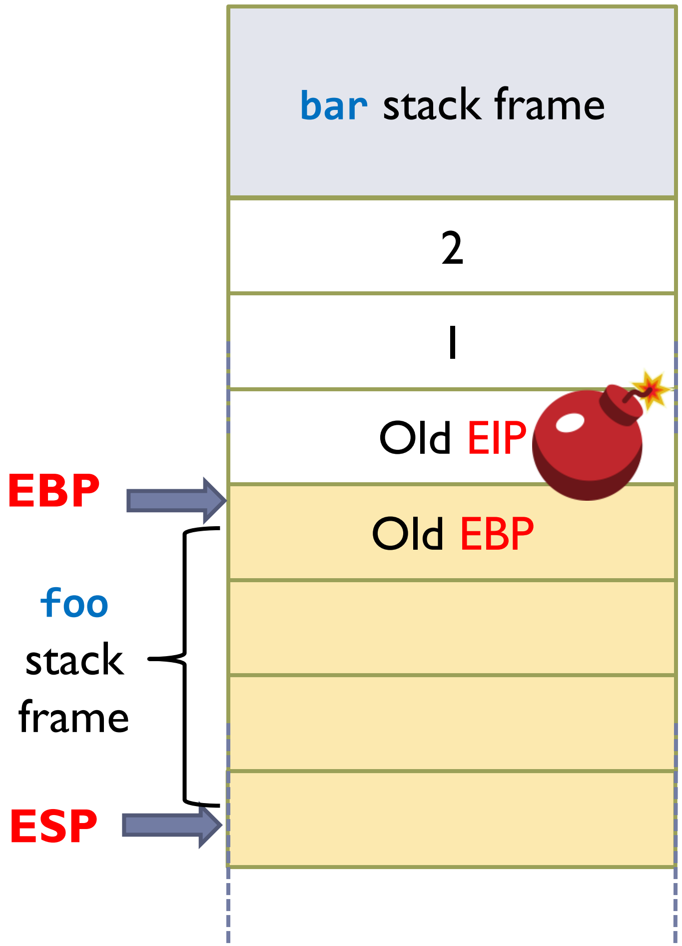
Stack Smashing

Function call convention:

- Step 2: Push the current instruction pointer (EIP) to the stack.
- Step 6: Execute the callee function within its stack frame.
- Step 9: Restore EIP from the stack.

Overwrite EIP on the stack during the execution of the callee function (step 6).

After callee function is completed (step 9), it returns to a different (malicious) function instead of the caller function!

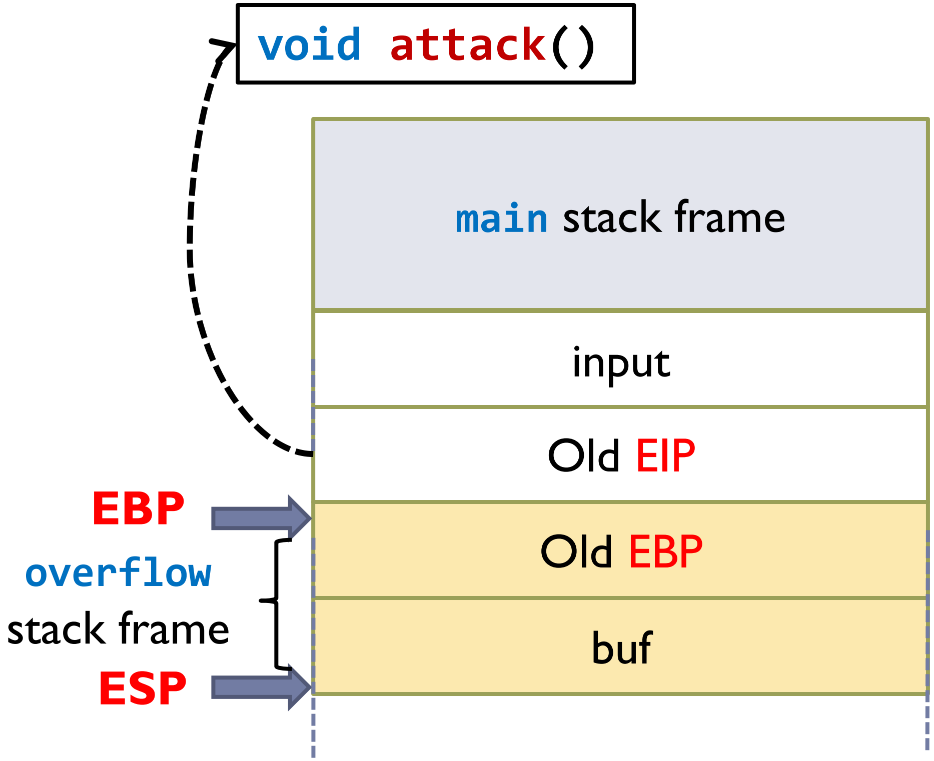


Example of Stack Smashing

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

void overflow(char* input){
    char buf[8];
    strcpy(buf,input);
}

void attack(){
    printf(“Attack succeed!\n");
}

int main(int argc, char **argv){
    char input[] =
"AAAAAAAAAAAAAAAA\xaf\x51\x55\x55\x55\x55";
    overflow(input);
    return 0;
}
```

Attack function address is:
\x55\x55\x55\x55\x51\xaf

Addresses are little-endian



input

\xaf	\x51	\x55	\x55	\x55	\x55	\x0	
A	A	A	A	A	A	A	A
A	A	A	A	A	A	A	A

Injecting Shellcode

Shellcode: a small piece of code the attacker injects into the memory as the payload to exploit a vulnerability

- Normally the code starts a command shell so the attacker can run any command to compromise the machine.

```c
#include <stdio.h>
int main( ) {
  char* name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

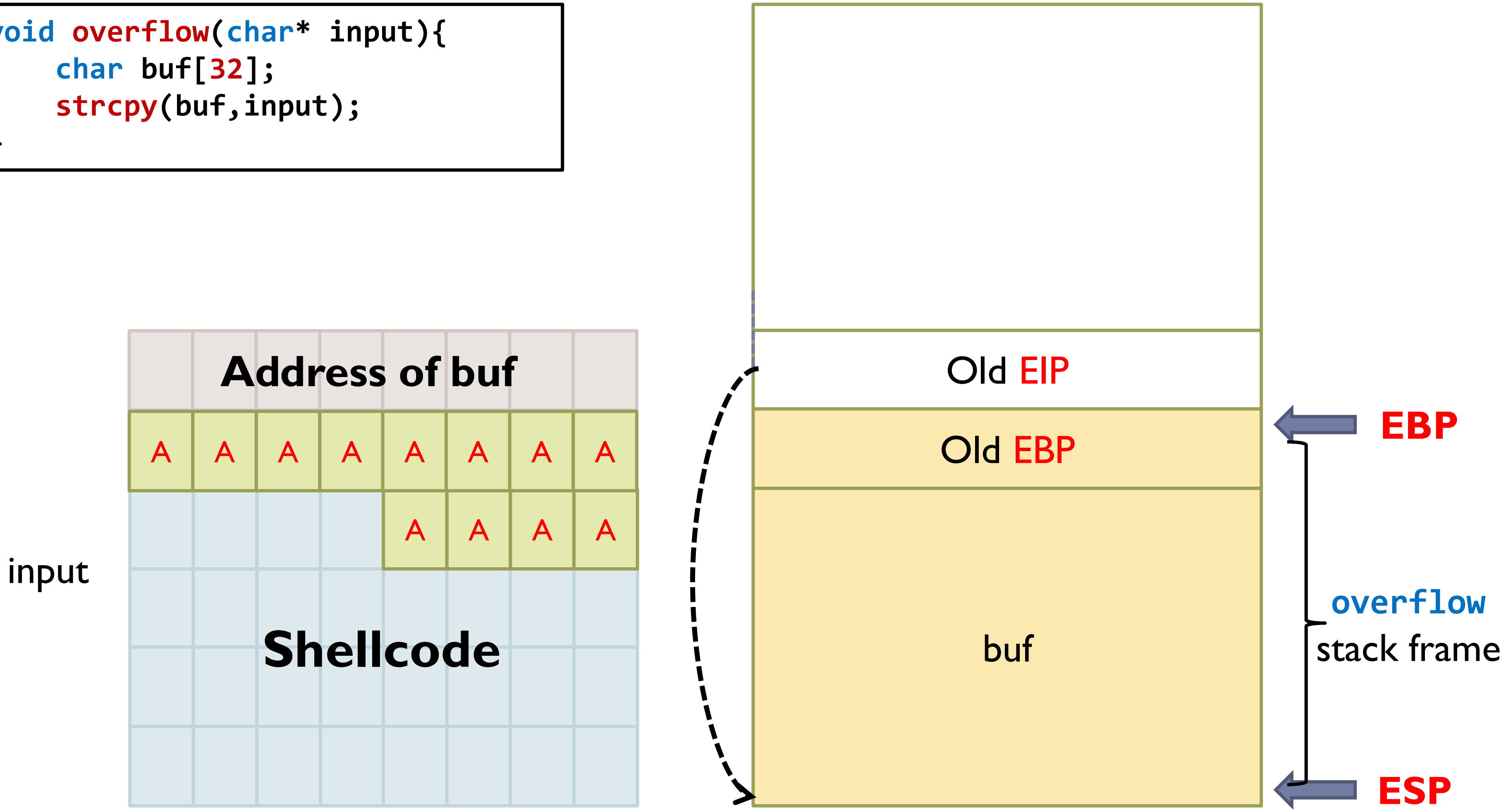
```asm
section .text
    global _start

_start:
    xor rdi, rdi
    push rdi
    mov rbx, 0x68732f2f6e69622f
    push rbx
    mov rdi, rsp
    xor rsi, rsi
    xor rdx, rdx
    mov al, 59
    syscall
```

```
48 31 ff
57
48 bb 2f 62 69 6e 2f
2f 73 68
53
48 89 e7
48 31 f6
48 31 d2
b0 3b
0f 05
```

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
 unsigned char shellcode[] =
"\x48\x31\xff\x57\x48\xbb\x2f\x62\x69\x6
e\x2f\x2f\x73\x68\x53\x48\x89\xe7\x48\x3
1\xf6\x48\x31\xd2\xb0\x3b\x0f\x05";
    ((void(*)()) shellcode)();
}
```

Overwrite EIP with the Shellcode Address

```
void overflow(char* input){
    char buf[32];
    strcpy(buf,input);
}
```

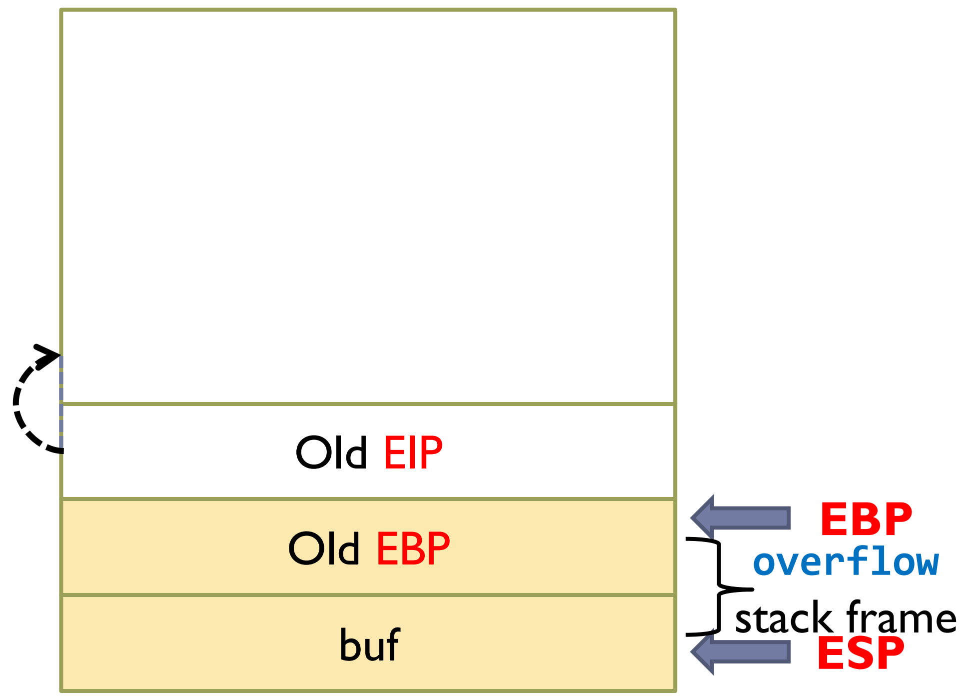


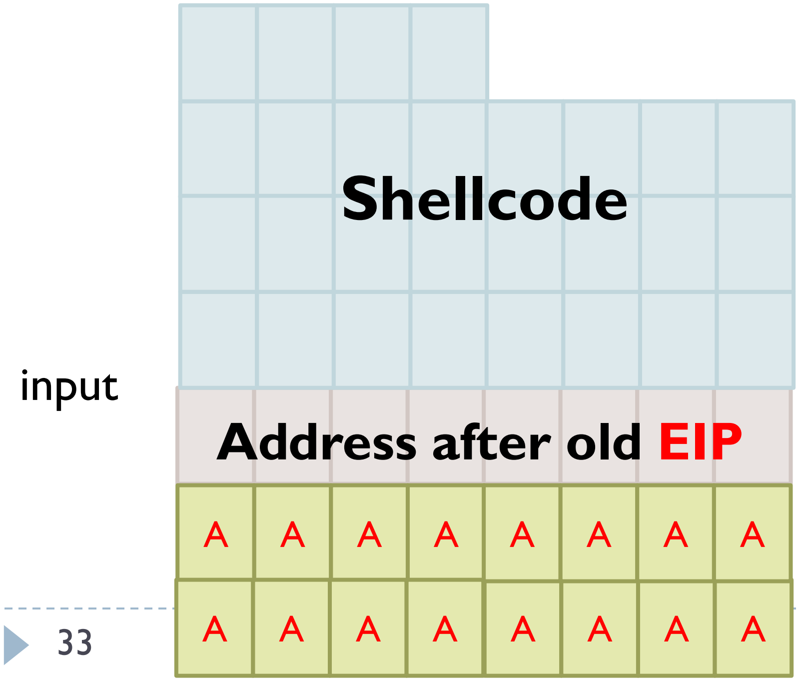
Overwrite EIP with the Shellcode Address

What if buf is smaller than shellcode?

- Place the shellcode after EIP

```
void overflow(char* input){
    char buf[8];
    strcpy(buf,input);
}
```





Summary of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program (e.g., strcpy from users' input without checking boundaries)

2. Inject shellcode into a known memory address

3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address. Normally this step can be combined with step 2 using one input.

4. Return from the vulnerable function.

5. Start to execute the shellcode.

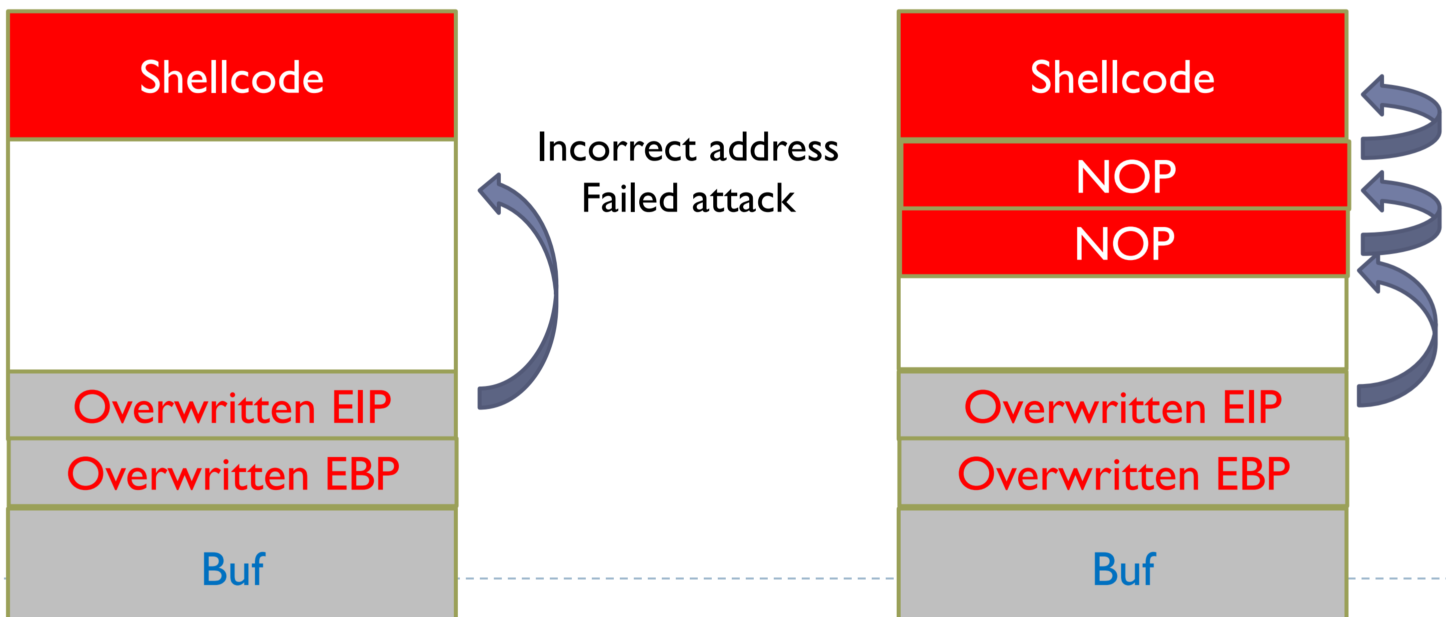
Shellcode Address is Unknown

Need to guess the address of shellcode.

- Incorrect address can cause system crash: unmapped address, protected kernel code, data segmentation

Improve the chance: Insert many NOP instructions before shellcode

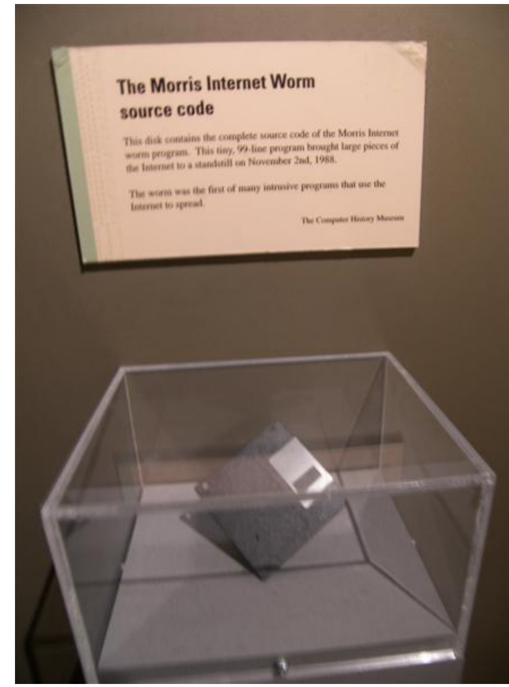
- NOP (No-Operation): does nothing but advancing to the next instruction.



Morris Worm: the First Buffer Overflow Vulnerability

History

- Released at 8:30pm, 2 November 1988 by Robert Tappan Morris, a graduate student at Cornell University
- Launched from the computer system of MIT, trying to confuse the public that this is written by MIT students, not Cornell.
- Buffer overflow in sendmail, fingerd network protocol, rsh/rexec, etc.



loppy disk containing the source code for the Morris Worm, at the Computer History Museum

Impact

- ~6,000 UNIX machines infected (10% of computers in Internet)
- Cost: $100,000 - $10,000,000

Robert Tappan Morris

What happens after Morris Worm

- Tried and convicted of violation of 1986 Computer Fraud and Abuse Act. This is the first felony conviction of this law.
- Sentenced to three years' probation, 400 hours of community service, and a fine of $10,050 (equivalent to $22,000 in 2023).
- Had to quit PhD at Cornell. Completed PhD in 1999 at Harvard.
- Cofounded Y Combinator in 2005
- Became a tenured professor at MIT in 2006. Elected to the National Academy of Engineering in 2019.

Robert Tappan Morris, Entrepreneur, professor at MIT

Following Morris Worm

2001

Code Red
Targeting Microsoft's IIS web server. Affected 359,000 machines in 14 hours

2003

SQL Slammer
Targeting Microsoft's SQL Server and Desktop Engine database. Affected 75,000 victims in 10 minutes

2005

Sasser
Targeting LSASS in Windows XP and 2000. Affected around 500,000 machines. Author: 18-year-old German Sven Jaschan. Received 21-month suspended sentence

2008

Conficker
Targeting Windows RPC. Affected around 10 million machines

2010

Stuxnet
Targeting industrial control systems, and responsible for causing substantial damage to the nuclear program of Iran

2012

Flame
Targeting cyber espionage in Middle Eastern countries

There are more…