Lecture 5
Virus, Malware and Buffer Overflow

CY201-01 Intro to Cybersecurity

Reshmi Mitra, PhD

Southeast Missouri State University

Spring 2020

Table of Contents

Table of Contents

Malicious code

Malicious = intending to do harm

- What is a malicious code?
- How can it take control of a system?
- How can it lodge in a system?
- How does malicious code spread?
- How can it be recognized?
- How can it be stopped?

Malware or Malicious Code - Introduction

Malware is a catch-all term for any type of malicious software, regardless of how it works, its intent, or how it's distributed.

Virus is a specific type of **malware** that self-replicates by inserting its code into other programs.

What is a malicious code?

- Unexpected or undesired effects in program or data caused by an agent intent on damage.
- Agent is the writer of the code
- Mistakes are not malicious (human error)

Malware or Malicious Code - Types

- **Trojan Horse** - decoy as legitimate application while disguising its intent to spy for malicious effects such as:
  - spying on the victim's computer
  - access files
  - extract sensitive data
- **Logic Bomb** - Becomes active only on when a condition becomes true
- **Time Bomb** – Becomes active only at certain time
- **Trapdoor (backdoor)** – other means of gaining privileged access to machine or program; bypasses normal access control and authentication
- **Worm** – propagates copies of itself via network
- **Rabbit** – replicates to exhaust resources

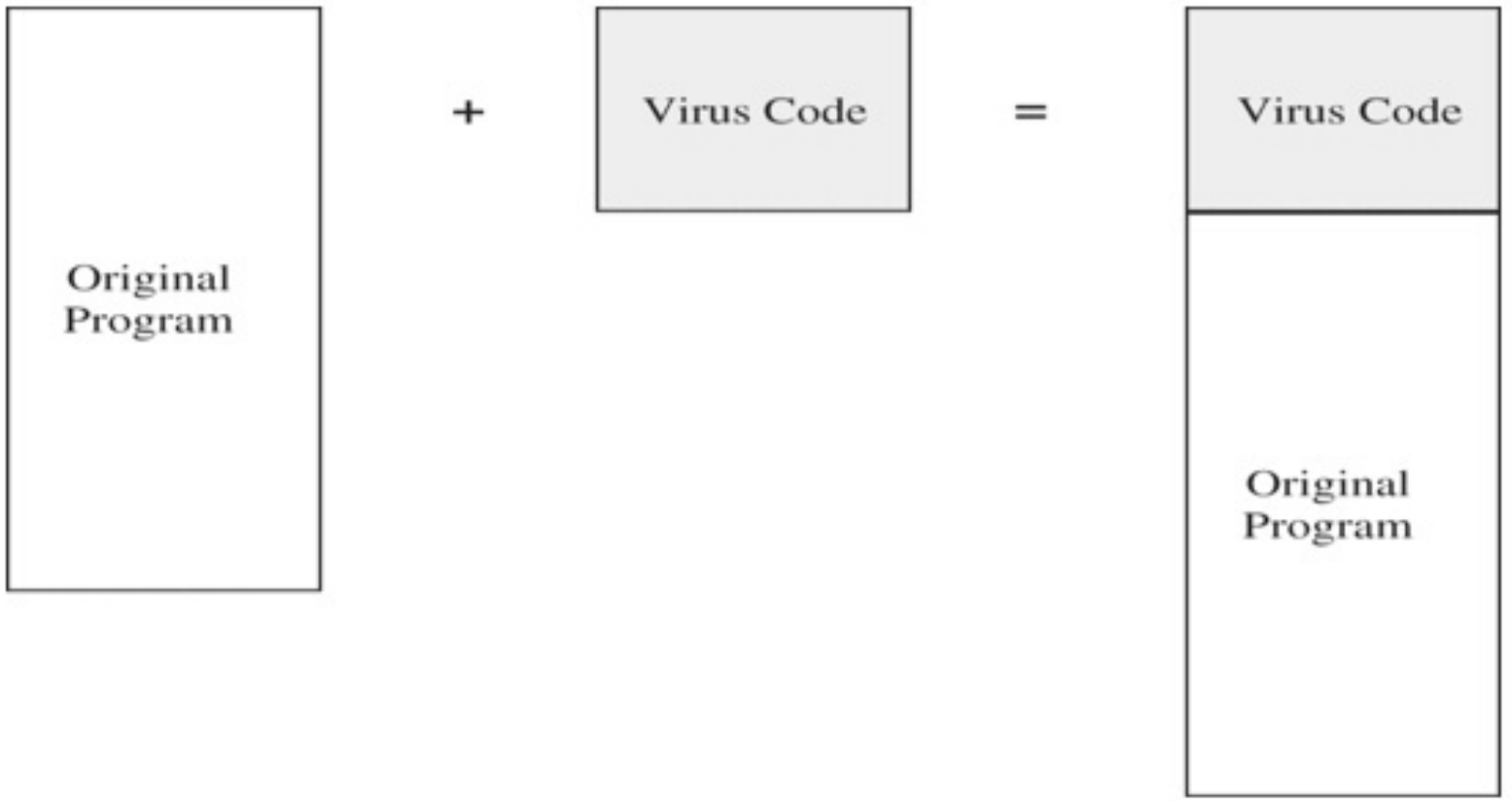
Viruses - Malicious code

All viruses are not the same.

One way to categorize is based on when they become active:

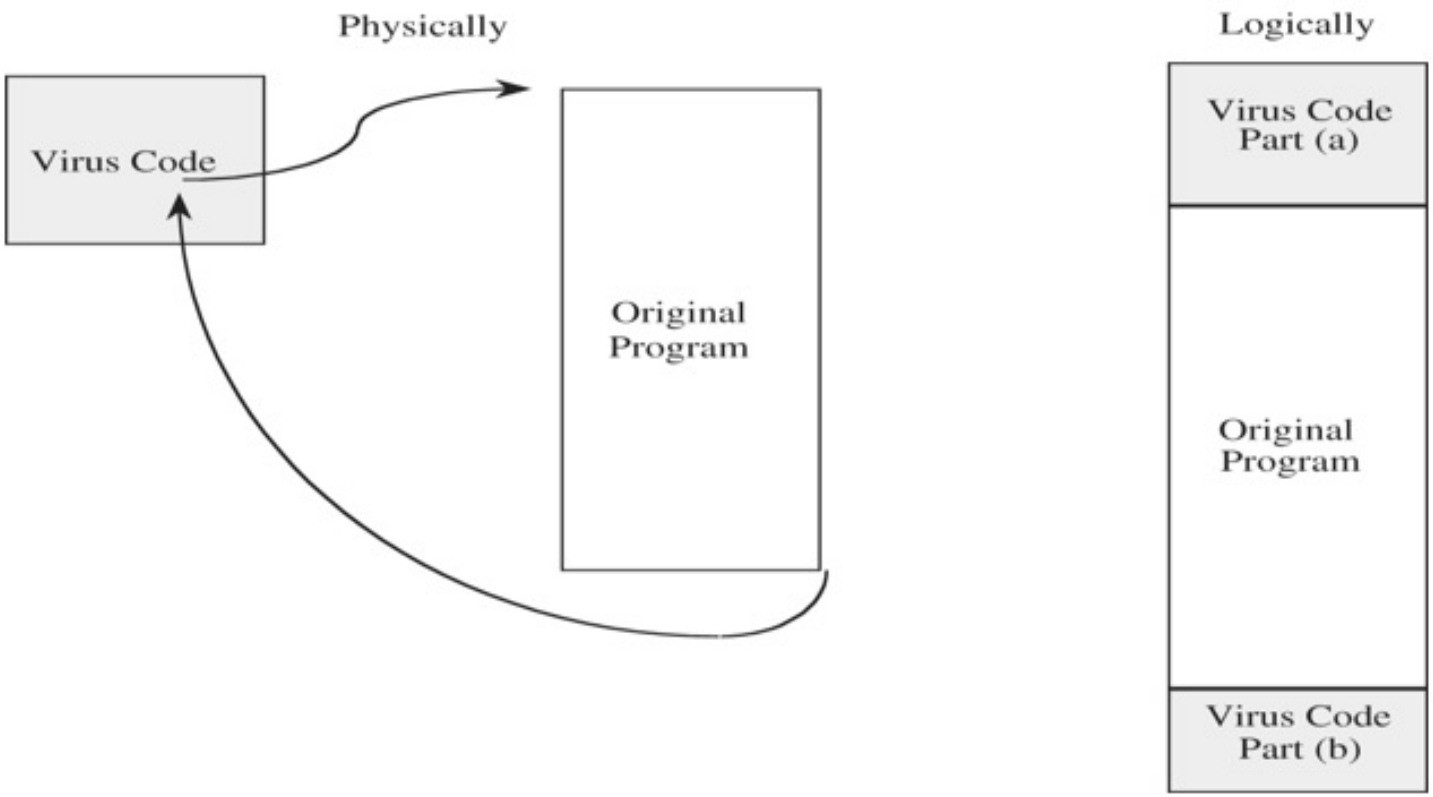
- **Transient virus** – runs when host runs
- **Resident virus** – resides in memory (active as a stand alone)

Virus Infection of Software - Appending



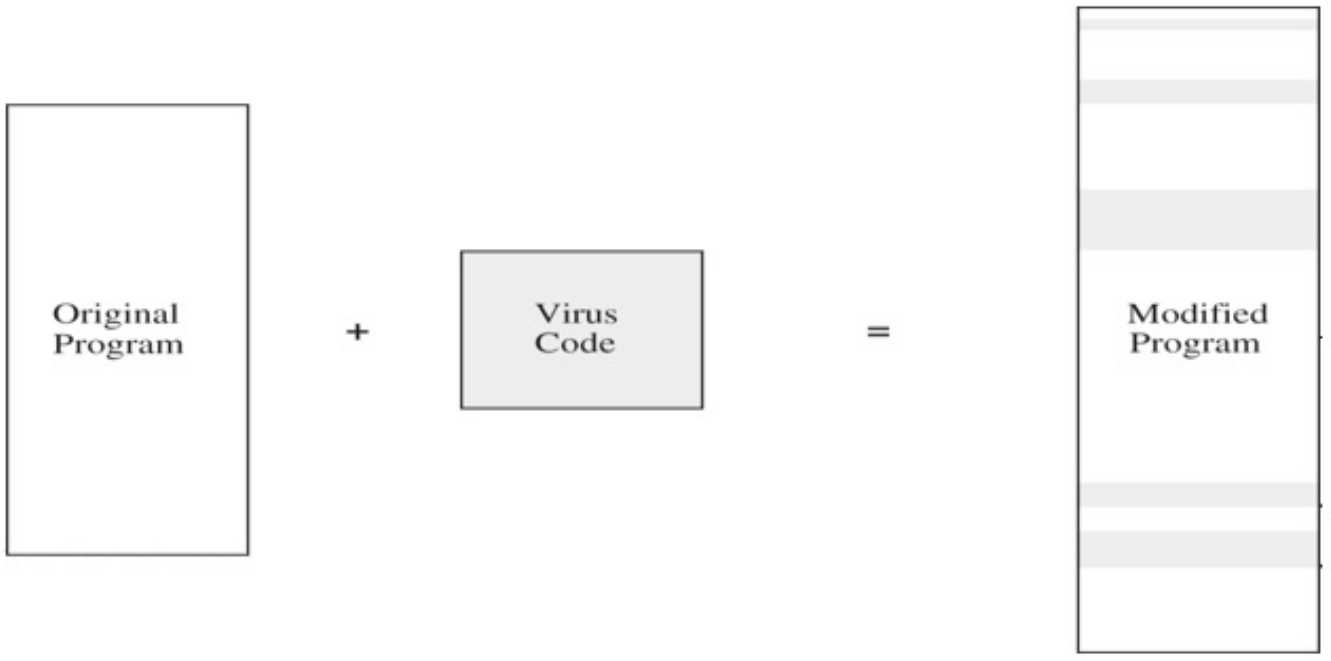
Virus Appended to a Program.

Virus Infection of Software - Surrounding



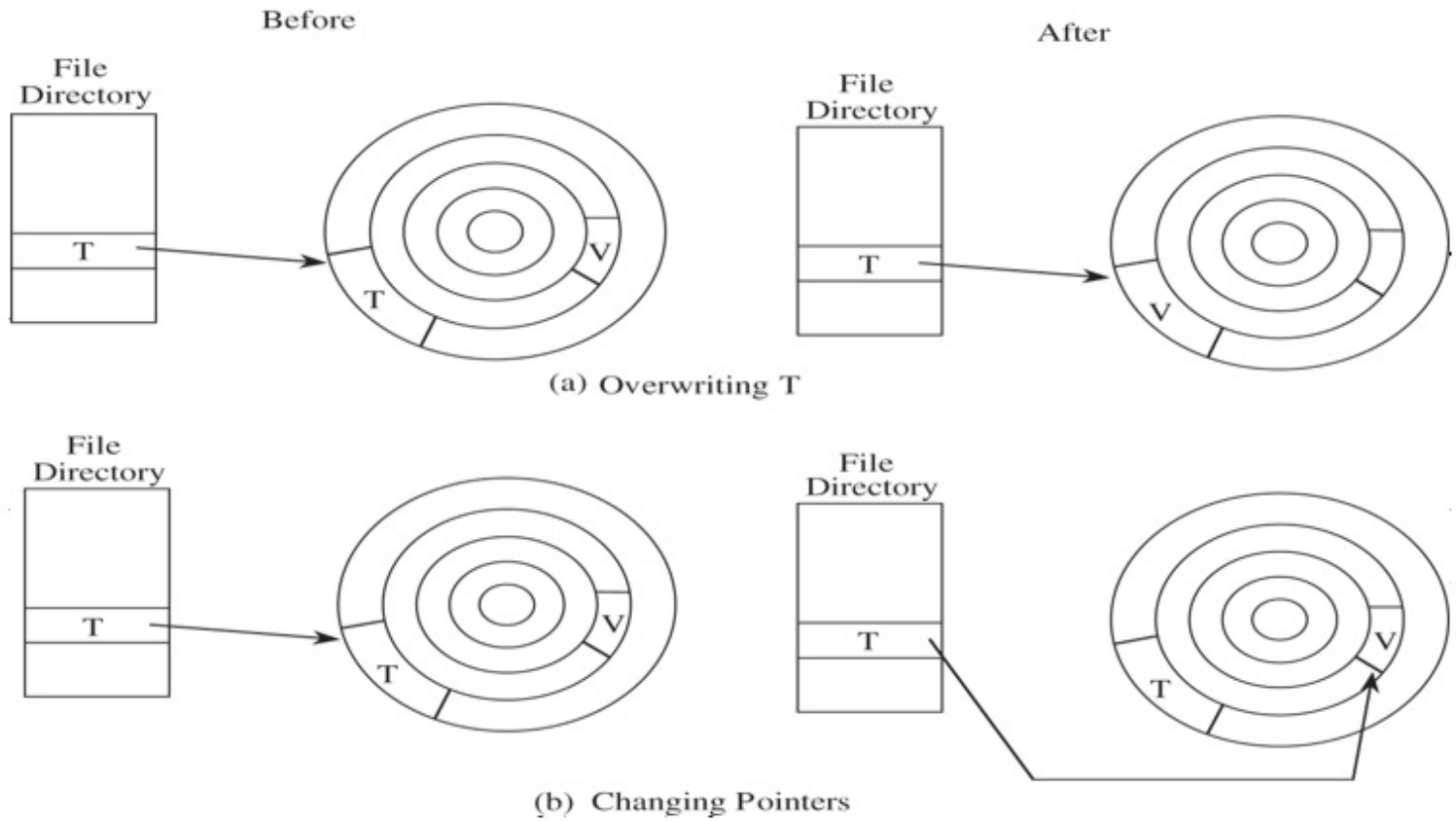
Virus Surrounding a Program.

Virus Infection of Software - Integrating



Virus Integrated into a Program.

Virus Infection of Software - Replacing



Virus Completely Replacing a Program.

Virus Writers

If you are a virus writer ...

- What is the meaning of an 'effective' code?
- How will you make it 'malicious'?
- Which of the prior viruses is hardest to detect?
- Which of the prior viruses is easiest to propagate?

Virus Writers

How will one write an 'effective' malicious code?

Appealing Qualities for Virus Writers:

- Easy to create
- Hard to detect - stealth operation
- Not easily destroyed
- Spreads widely
- Re-infects easily
- Machine and OS independent

Understanding Virus Behavior

In order to develop detection and protection mechanisms, we need to understand how viruses lodge in a system

Where do viruses live?

- Memory-Resident Viruses
  - Terminate and Stay Resident (TSR)
  - Infects Windows System Registry to reload
- Applications
- Macros
- Scripts
- Libraries
- Images
- Documents

Remember: detection (and protection) mechanism will change based on its location.

Virus Detection

Virus Signatures

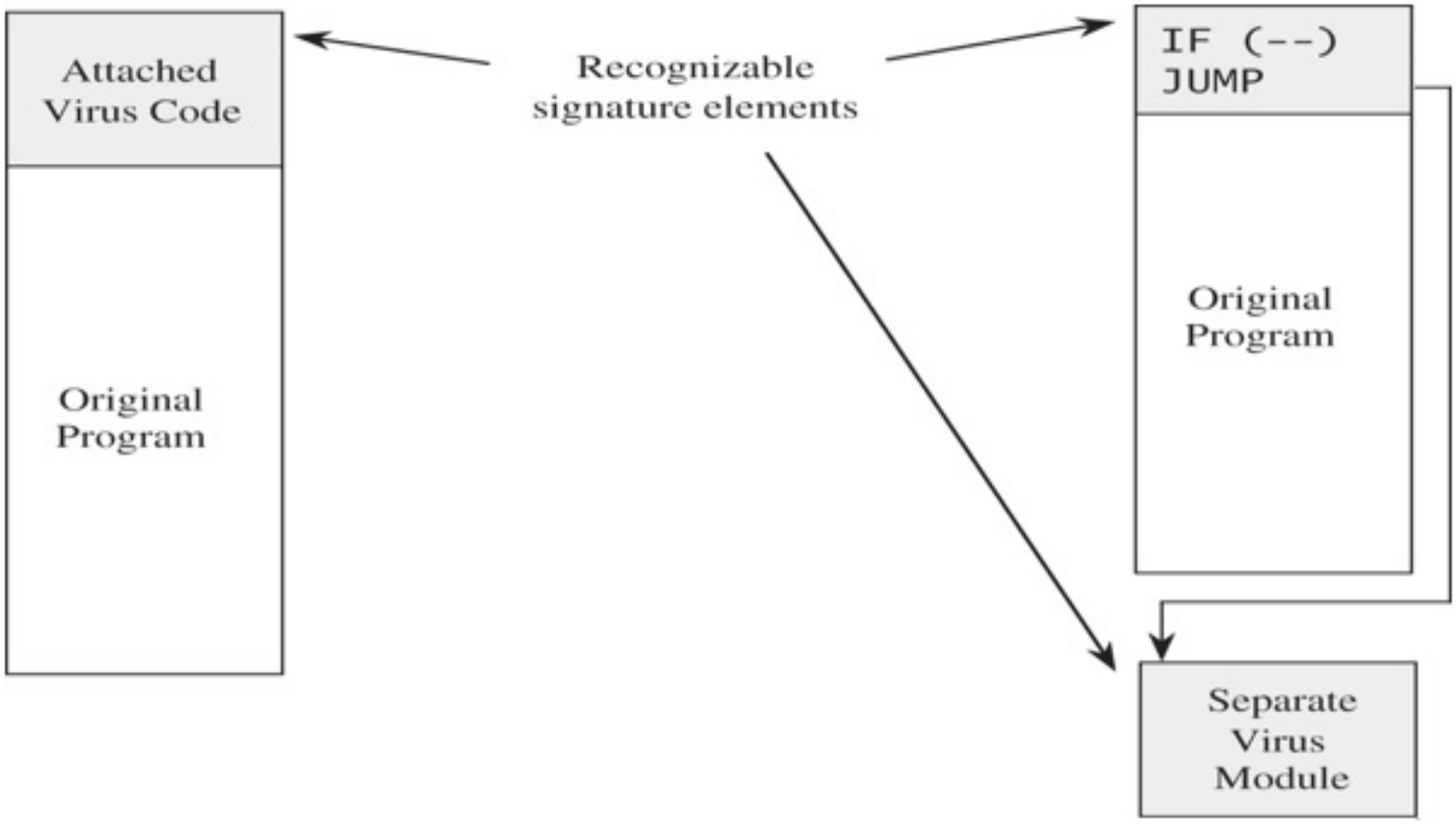
- Viruses are no completely invisible
- They all leave a signature pattern (DNA)
- Patterns are found with Virus Scanners

Virus patterns

- Always at same location
- Top of file location
- File size grows
- Strange code; *jump* statements
- Hash or checksum change (later chapters)

Think about Anti-virus scans...

Recognizable Patterns in Viruses



Properties of Effective Virus

- **Transmission Patterns** - Effective means of transmission from one location to another is critical
- **Polymorphic Virus** – virus that can change its appearance and hence pattern
- **Encrypting Virus** – tries to hide

Think of stealth appearance and escaping detection

Virus Prevention Mechanisms

How can one prevent virus from propagating:

- Commercial software applications
- Test all software
- Opening attachments
- Make system images
- Keep copies of executable files and data files
- Virus Detection Software

Virus - Review Questions

Truths and Misconceptions about Viruses

- Viruses infect only Windows
- Viruses can modify “hidden” or “read-only” files
- Files only appear in executable files
- Viruses spread only on disks or only through email
- Viruses cannot remain in memory when power is off
- Viruses cannot infect hardware
- Viruses can be malevolent, benign or benevolent

Justify your answer with reason(s).

Virus - Review Questions

Truths and Misconceptions about Viruses

- Viruses infect only Windows (False)
- Viruses can modify "hidden" or "read-only" files (True)
- Files only appear in executable files (False)
- Viruses spread only on disks or only through email (False)
- Viruses cannot remain in memory when power is off (True/False)
- Viruses cannot infect hardware (True/False)
- Viruses can be malevolent, benign or benevolent (True)

Justify your answer with reason(s).

Section Review: Malicious code

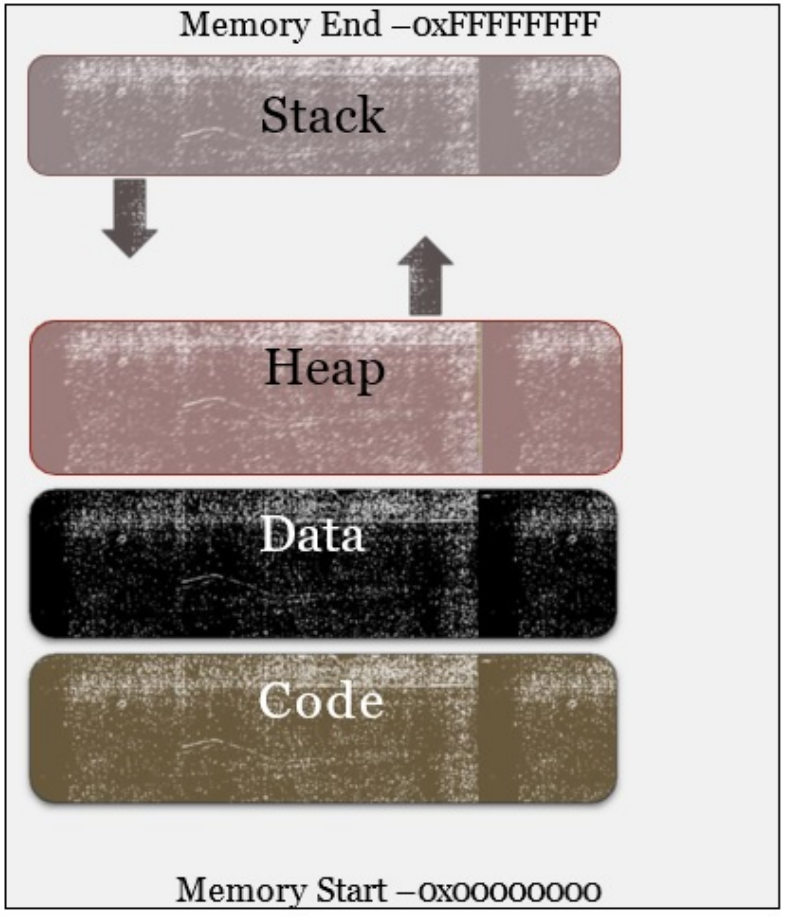
- What is a malicious code?
- How can it take control of a system?
- How can it lodge in a system?
- How does malicious code spread?
- How can it be recognized?
- How can it be prevented or detected?
- Difference between benign and malicious code

Recommended reading: Sec 3.2 (166 - 196)

Table of Contents

Memory Allocation

The memory of a typical program is divided into:



Memory Allocation - functions of different segments

The memory of a typical program is divided into:

- **Code Segment** == 'text'
- **Data Segment** (Holds global data)
- **Stack**
  - "scratch paper"
  - static memory allocation
  - local variables, parameters to functions, return address
  - where temporary information is stored
- **Heap**
  - dynamic memory allocation

Heap vs Stack

Heap

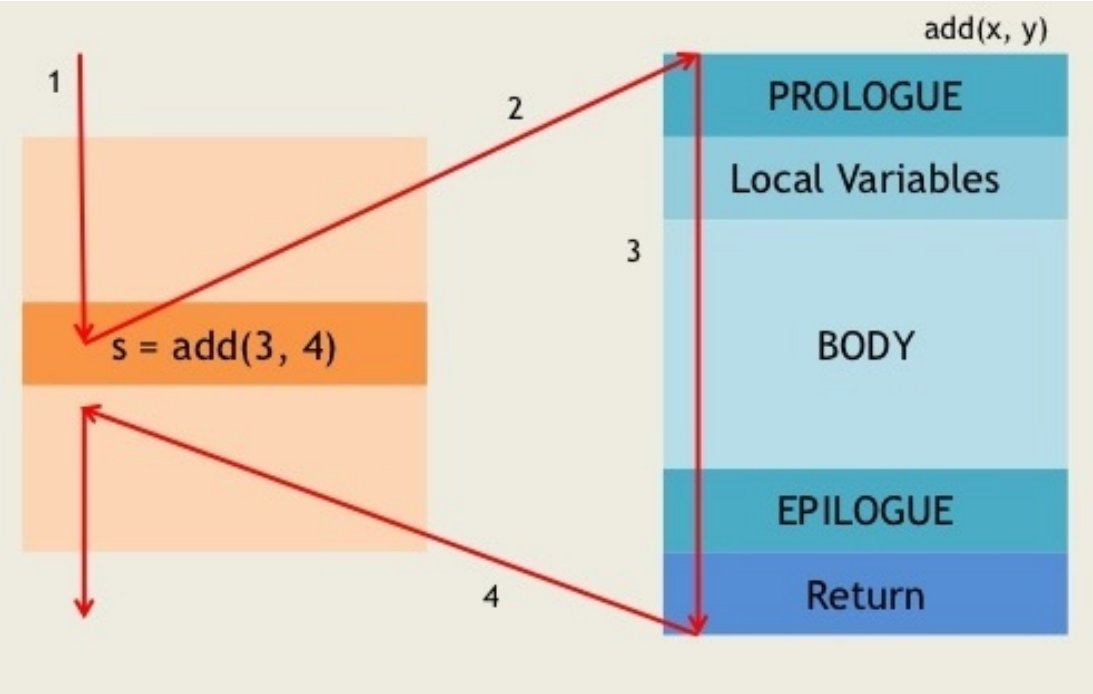
- Allocated at run time.
- Dynamically allocated variables are found in the heap
- Typically, a pointer refers to a heap address, if it is returned by a call to the *malloc* function.

Stack

- The stack is used to store function arguments, local variables, or some information allowing to retrieve the stack state before a function call...
- This stack is based on a LIFO (Last In, First Out) access system, and grows toward the low memory addresses.

Function call example

What is the sequence of events when you call **'add (3, 4)'**?



How will this change when there are recursive function calls?

Function call

Memory allocation becomes important during function call as program control moves to a different memory region.

A function call may be broken up in three steps:

- **prologue**: the current frame pointer is saved. A frame can be viewed as a logical unit of the stack and contains all the elements related to a function. The amount of memory which is necessary for the function is reserved.
- **call**: the function parameters are stored in the stack and the instruction pointer is saved, in order to know which instruction must be considered when the function returns.
- **return(or epilogue)**: the old stack state is restored.

Buffer introduction

In C language, strings OR buffers are:

- Represented by a pointer to the address of their first byte,and
- we consider we have reached the end of the buffer when we see a NULL byte.

Implications:

- No way to set precisely the amount of memory reserved for a buffer
- Depends on the number of characters.

Buffer example

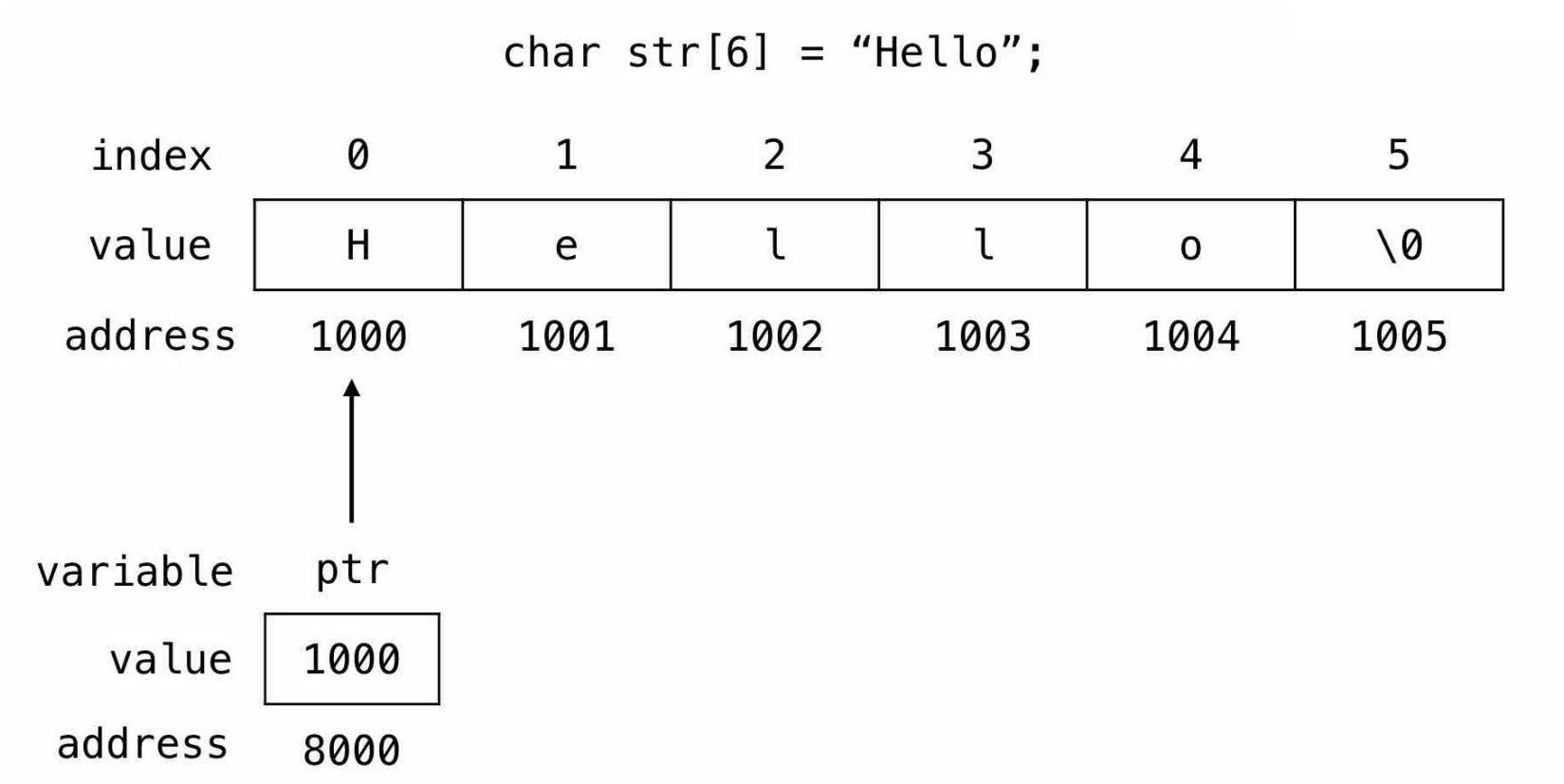
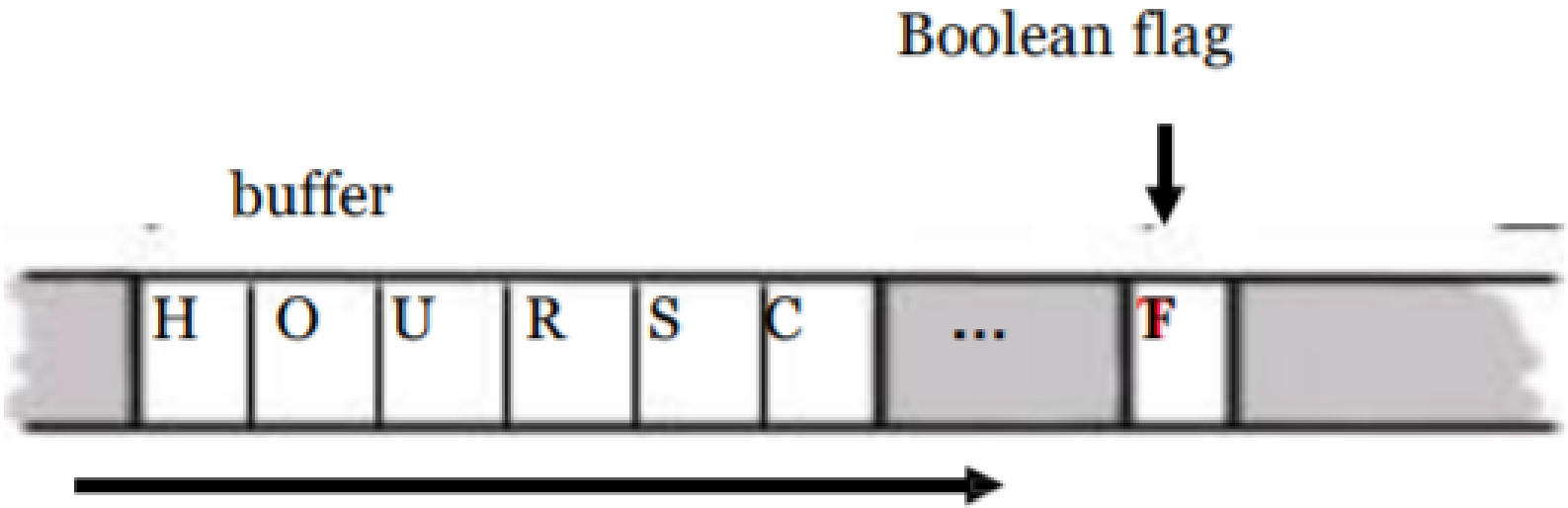


Table of Contents

Simple Buffer Overflow

Consider boolean flag for authentication

- Buffer overflow could overwrite flag allowing anyone to authenticate!
- In some cases, attacker need not be so lucky as to have overflow overwrite flag



Buffer Exploitation

The general idea is to give servers very large strings that will overflow a buffer.

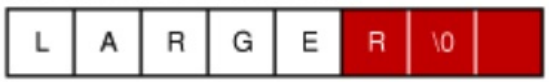
For a server with sloppy code - it's easy to crash the server by overflowing a buffer (SEGV typically).

It's sometimes possible to make the server do whatever you want (instead of crashing).

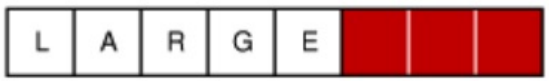
Buffer Exploitation - practical problems

Avoid using strcpy() because it does not perform bounds checking. Use either an alternative function that performs the same function with bounds checking or check bounds manually for any use of strcpy()

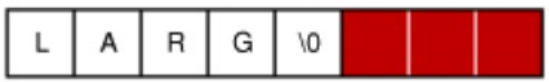
```
Char destination[5]; char *source = "LARGER";

strcpy(destination, source);
```




```
strncpy(destination, source, sizeof(destination));
```

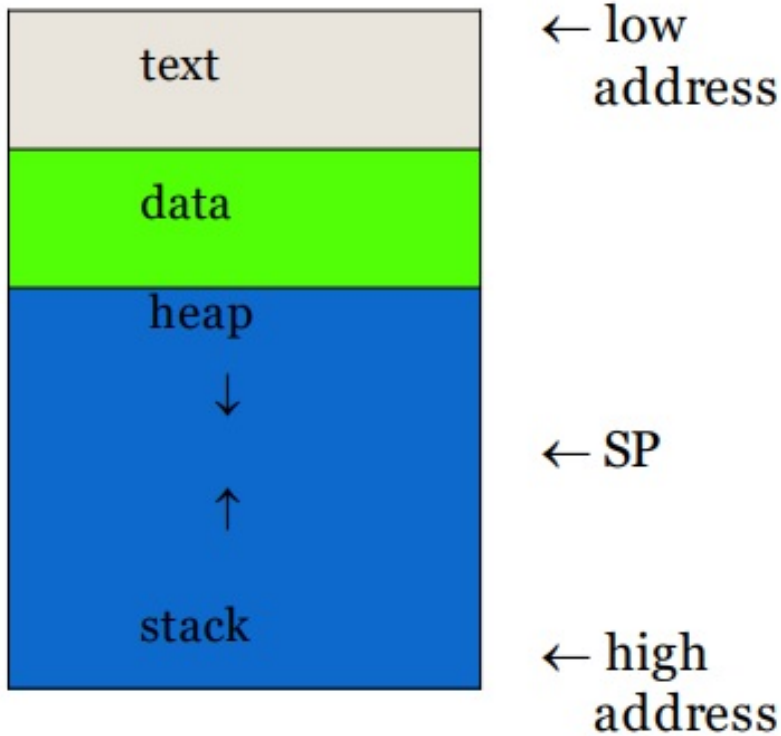



```
strlcpy(destination, source, sizeof(destination));
```



Memory organization - Review

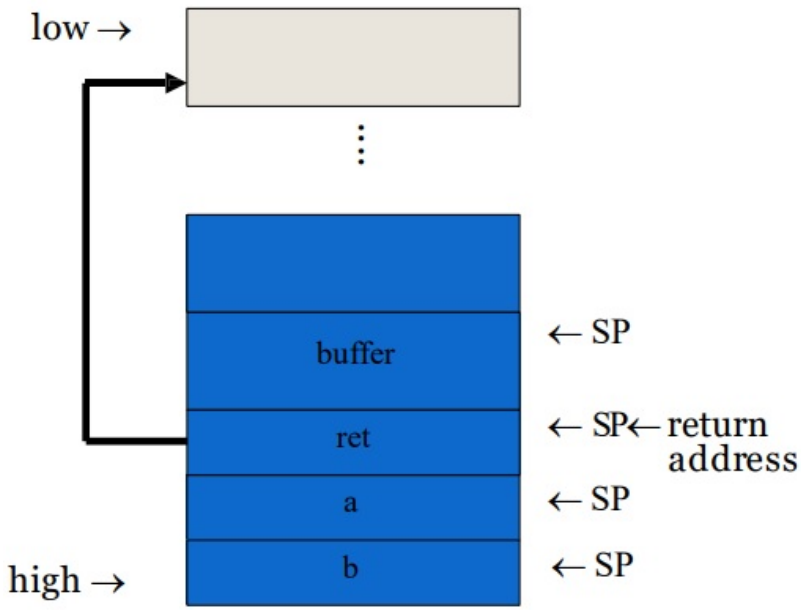
- **Code Segment** == 'text'
- **Data Segment** (Holds global data)
- **Stack**
  - "scratch paper"
  - static memory allocation
  - local variables, parameters to functions, return address
  - where temporary information is stored



SP = stack pointer

Simplified stack example

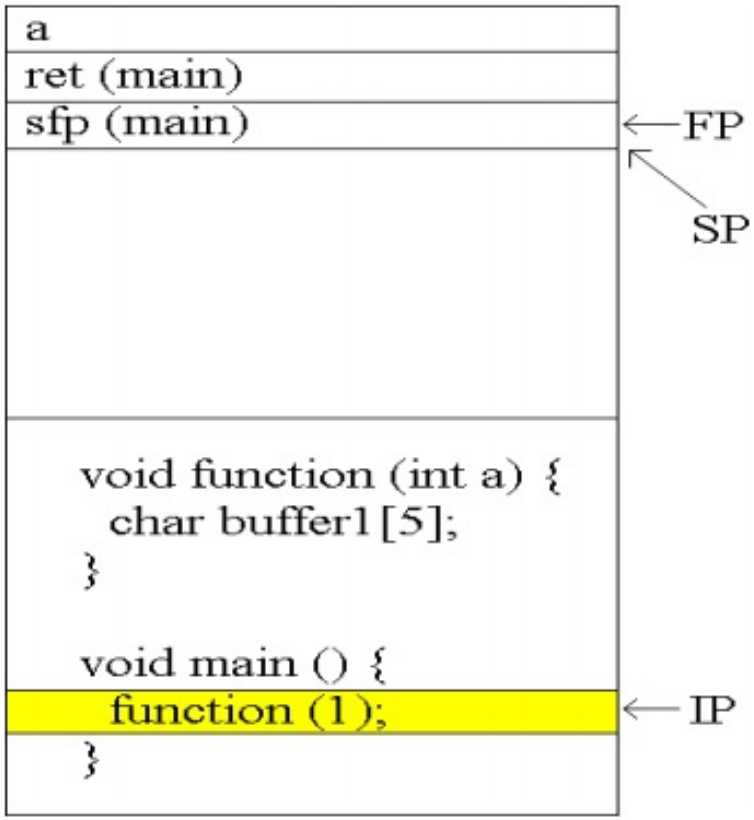
```
void my_func(int a, int b){
     char buffer[10];
}

void main(){
     my_func(1, 2);
}
```



SP = stack pointer

How will the stack look like for func(1, 2, 3, 4)?

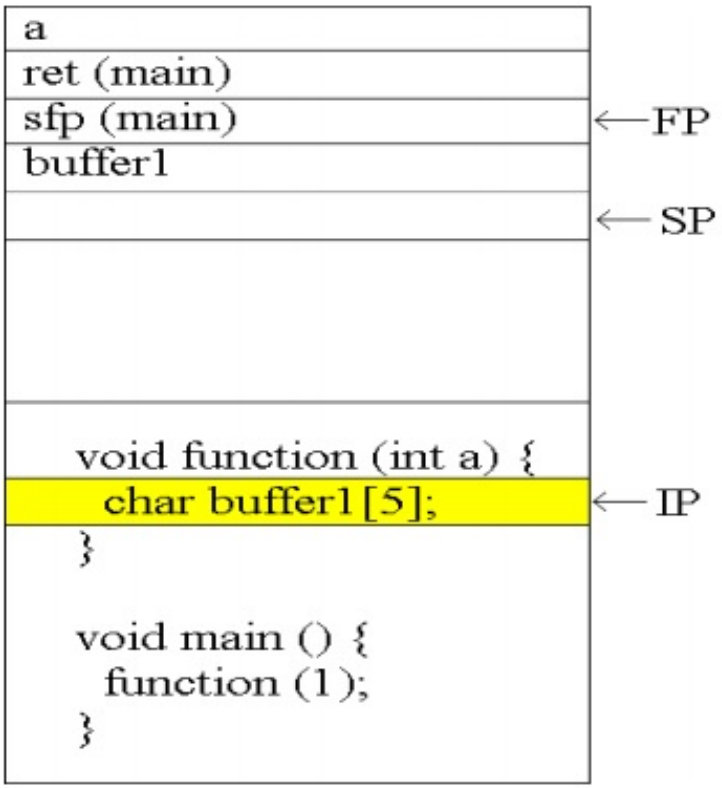
Function call - **in main()**



- FP = Frame pointer
  points to the current active 'frame'
- SP = Stack pointer
  points to the top of the stack
- IP = instruction pointer
  points to the next instruction to be executed

Which are the code and stack segments in the diagram?

Function call - **in func()**



- Notice that local variable 'buffer1' is inserted in the stack
- SP points to next empty location

Buffer Overflow - Sample code

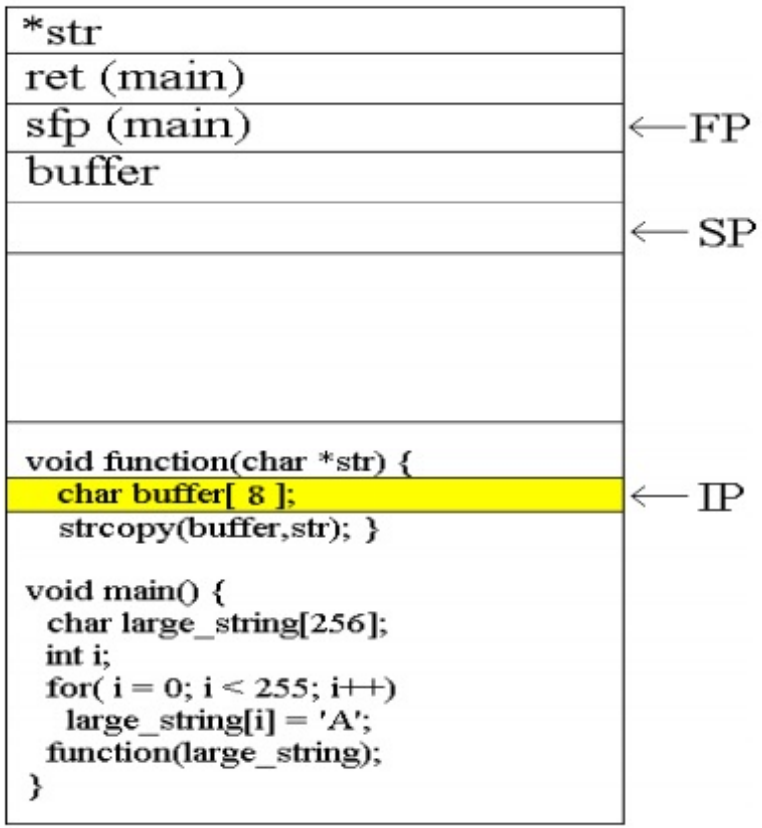
```
void function(char *str) {
    char buffer[8];
    strcpy(buffer,str);
}
void main(){
   char large_str[256];
   int i;
   for(i = 0; i<255; i++){
      large_str[i] = 'A';
    }
    function(large_str);
}
```

What is this code doing?

- main - copy the same character 255 times in 'large_str'
- function - copy 'str' in 'buffer'

Which part of the code is causing buffer overflow problem?

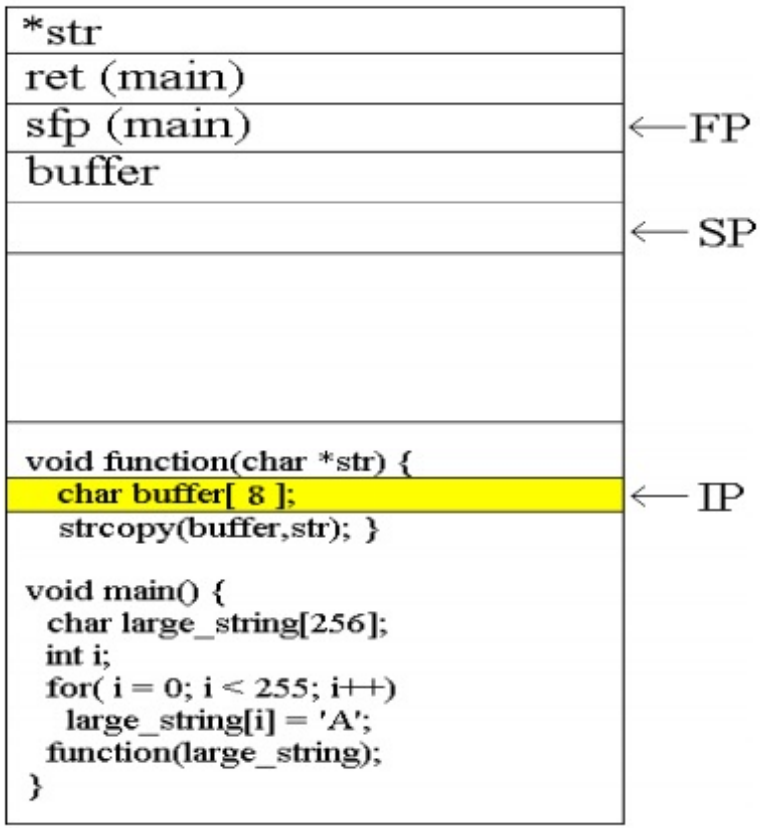
Buffer Overflows - 1



Notice that we are already in function

1. formal parameter - string pointer '*str' pushed into the stack
2. return address (ret) and stack frame pointer (sfp) pushed into the stack.
3. Local variable 'buffer' is inserted in the stack. From here it is all about contents of the function
4. SP points to next empty location in the stack

Buffer Overflows - 2

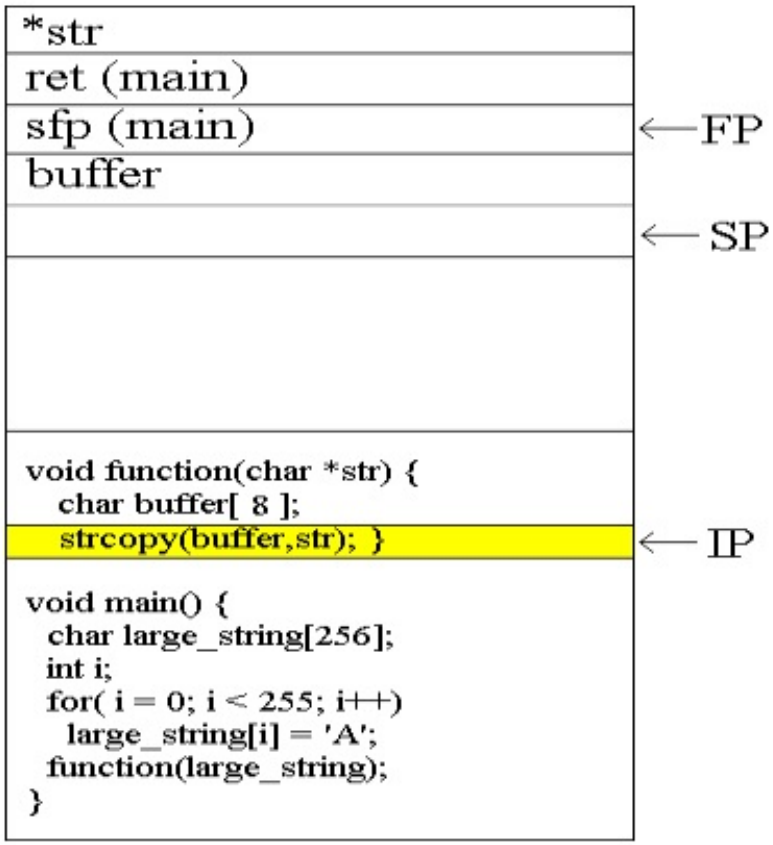


How will the stack look like if:

1. formal parameter changes to function (char *s1, char *s2)?
2. local variables change to 'char a, b'?

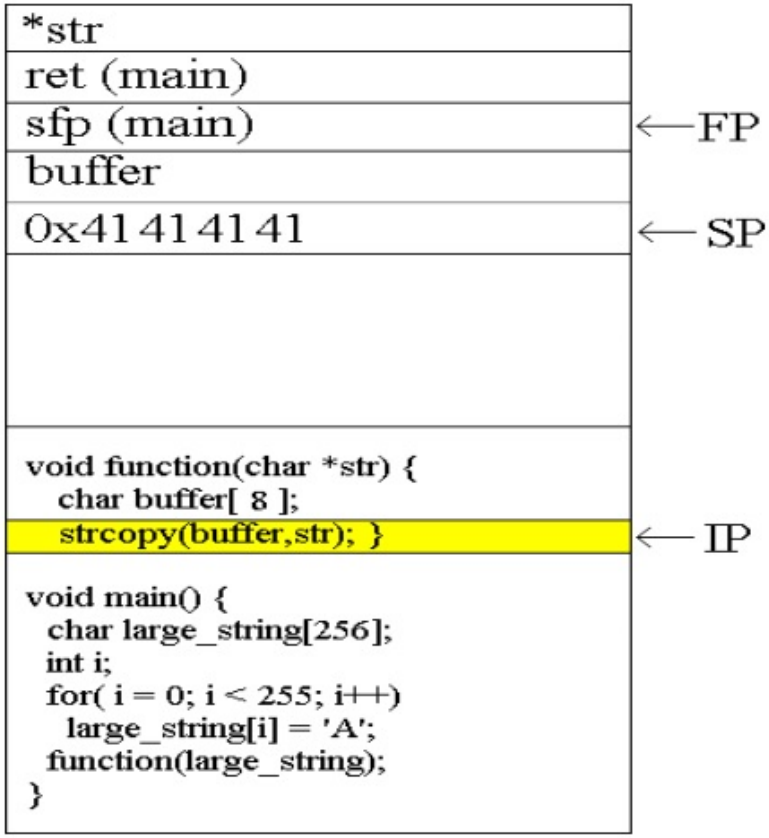
Can you redraw the stack?

Buffer Overflows - 3



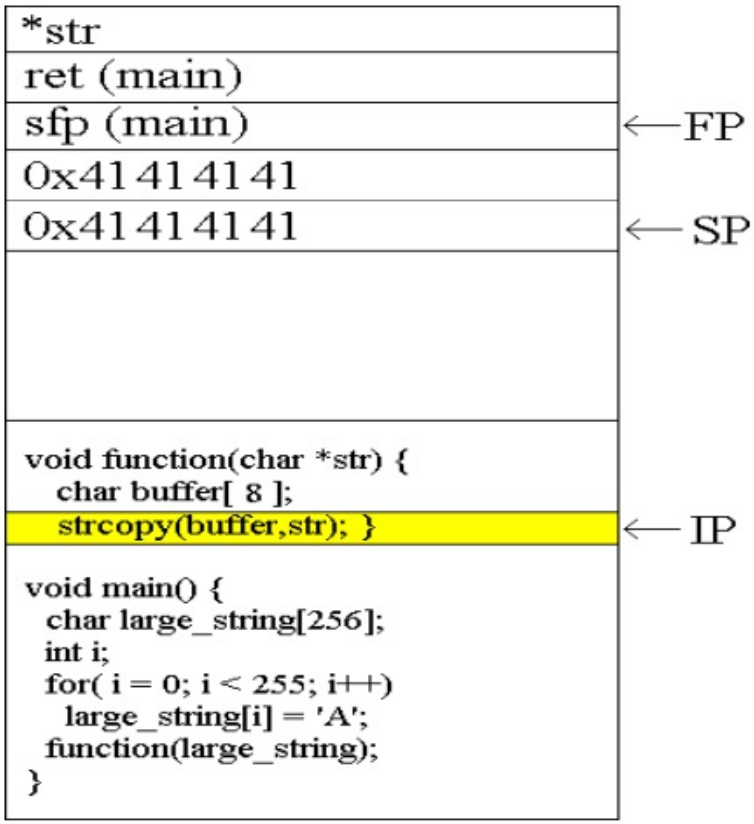
- Move to string copy instruction
- It is not executed yet, so you do not see the changes in the stack.

Buffer Overflows - 4



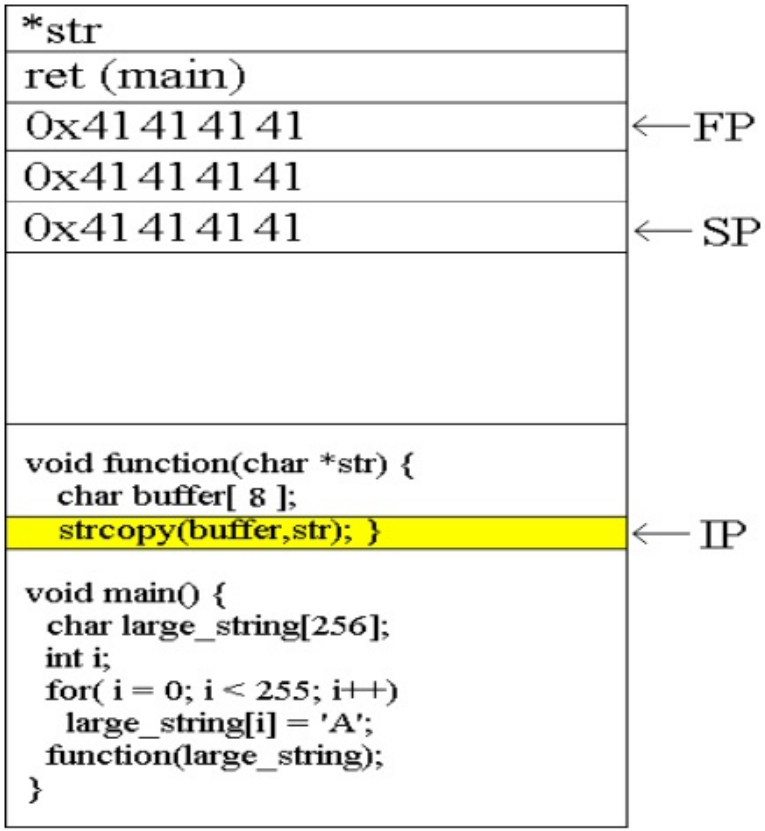
- ASCII code for ‘A’ is 65 (decimal) or 41 (hex)
- large_string = ‘AAA ... A’ (256 times!)
- strcopy the 1st byte from ‘str’ to ‘buffer’
- ‘buffer’ size is just 8 bytes

Buffer Overflows - 5



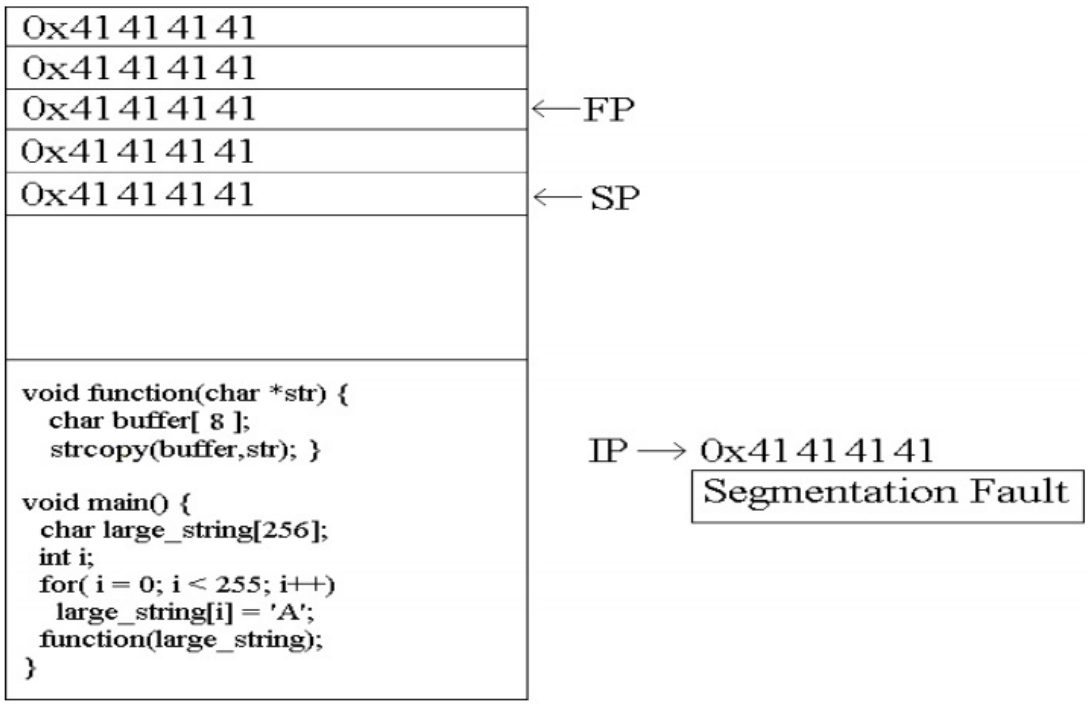
- strcopy the 2nd byte from 'str' to 'buffer'
- Is this buffer overflow yet?

Buffer Overflows - 6



- strcopy the 3rd byte from 'str' to 'buffer'
- How many times will the copy instruction be executed?

Buffer Overflows - 7



- Data is overwriting code, definitely segmentation fault!
- IP pointing to memory address 0x414141
- Is there a valid instruction in 0x414141?

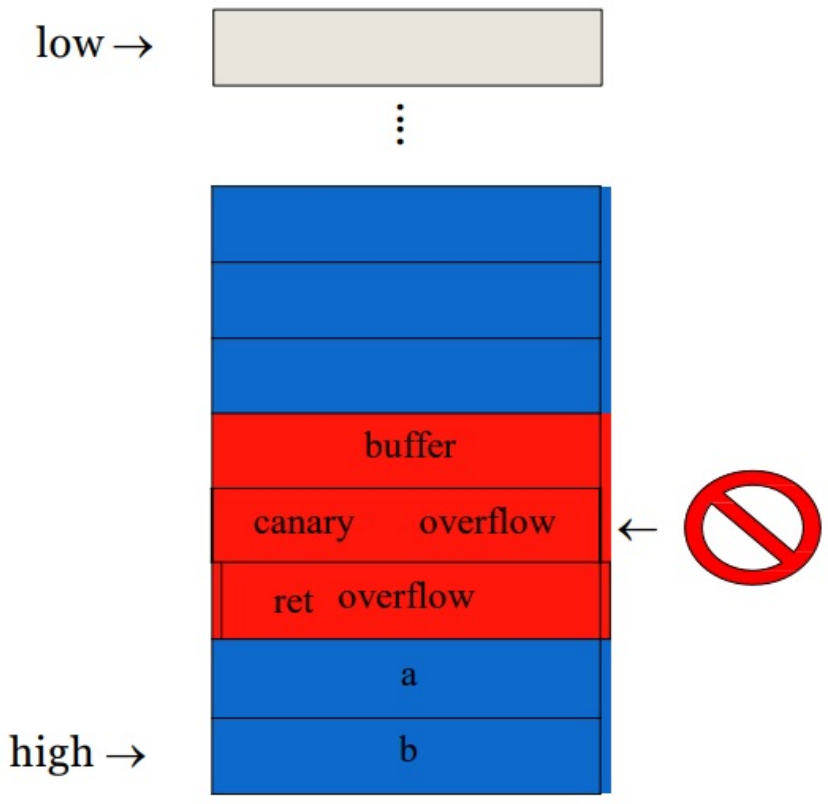
Stack smashing

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done, it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

How to prevent stack smashing problem?

Table of Contents

Canary - introduction



- Run-time stack check
- Push canary on to the stack
- Canary value is set constant OR depends on return address

Marking stack as non-execute

Basic stack exploit can be prevented by marking stack segment as non-executable.

- Support in Windows SP2. Code patches exist for Linux, Solaris.

Problems:

- Does not defend against 'return-to-libc' exploit.
- Some apps need executable stack (e.g.LISP interpreters).
- Does not block more general overflow exploits:
- Overflow on heap, overflow funcpointer.

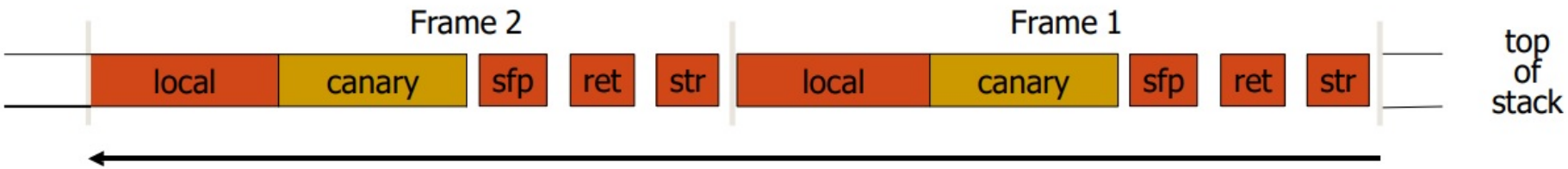
Microsoft's canary

- Microsoft added buffer security check feature to C++ with /GS compiler flag
- Uses canary (or "security cookie")
- Q: What to do when canary dies?
- A: Check for user-supplied handler
- Handler may be subject to attack
- Claimed that attacker can specify handler code
- If so, "safe" buffer overflows become exploitable when /GS is used!

Run time checking: Stackguard

There are many run-time checking techniques . . .

- StackGuard tests for stack integrity.
- Embed “canaries”in stack frames and verify their integrity prior to function



Canary types

Random canary:

- Choose random string at program startup.
- Insert canary string into every stack frame.
- Verify canary before returning from function.
- To corrupt random canary, attacker must learn current random string.

Terminator canary:

- Canary = 0, newline, linefeed, EOF
- String functions will not copy beyond terminator.
- Hence, attacker cannot use string functions to corrupt stack.

ASLR - Address Space Layout Randomization

- Randomize place where code is put into memory
- Makes most buffer overflow attacks probabilistic
- Vista does this (256 random layouts)
- Similar thing is done in Mac OS X