



# BUFFER OVERFLOW

BLM 4011 Bil. Sis. Gv.

**What you are expected to do:** You need to have a good understanding of the "buffer overflow" topic by reviewing this document and other necessary resources. Program.c code is shared, you are expected to write a C program similar to this example. There should be a variable that holds the information whether there is an admin or not in the program. As a result of the buffer overflow attack, without entering the correct admin pass and username, the variable holding the admin should be turned to true and it should be shown that there is an admin in the system. When you come to the lab in your respective lab section, all the necessary programs must be installed and ready to run on your computer. It is your responsibility to ensure that the versions of the relevant programs (Linux, gcc, etc.) are compatible and ready to run. You may also be asked to change some lines in your code to test your knowledge on the subject.

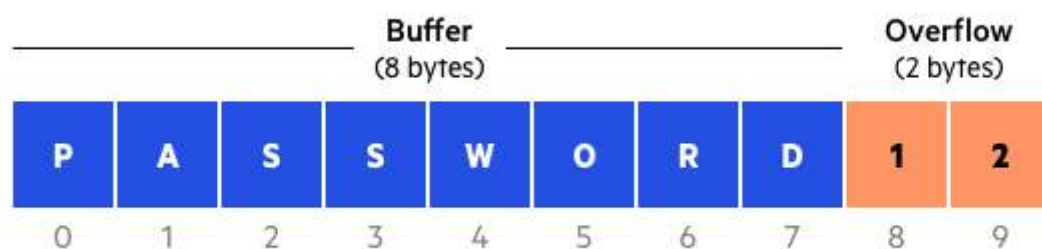
**Ne yapmanız bekleniyor:** Bu dokümanı ve gerekli başka kaynakları inceleyerek "buffer overflow" konusuna hakim olmanız gerekiyor. Program.c kodu paylaşılmıştır, bu örneğe benzer bir C programı yazmanız beklenmektedir. Programda admin olup olmadığını tutan bir değişken bulunmalıdır. Buffer overflow atağı sonucunda doğru admin pass ve username girmeden adminliği tutan değişkenin true ya çevrilip sistemde admin olduğu gösterilmelidir. Laba geldiğinizde gerekli tüm programların bilgisayarınızda kurulu ve çalıştırılmaya hazır olması gerekmektedir. İlgili programların (Linux, gcc, vb.) versiyonlarının uyumlu ve çalışmaya hazır olması sizin sorumluluğunuzdadır. Lab saatinizde sadece programları çalıştıracaksınız, ayrıca konuya ve koda hakimiyetiniz açısından bazı yerleri değiştirmeniz istenebilir.

## Brief Explanation of Buffer Overflow

Buffers are memory storage regions. When the data is transferring to another location buffers hold the data temporarily. When the volume of data exceeds the storage capacity of the memory buffer a buffer overflow occurs. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, so if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary.

Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.



## Brief Explanation of Buffer Overflow Attack

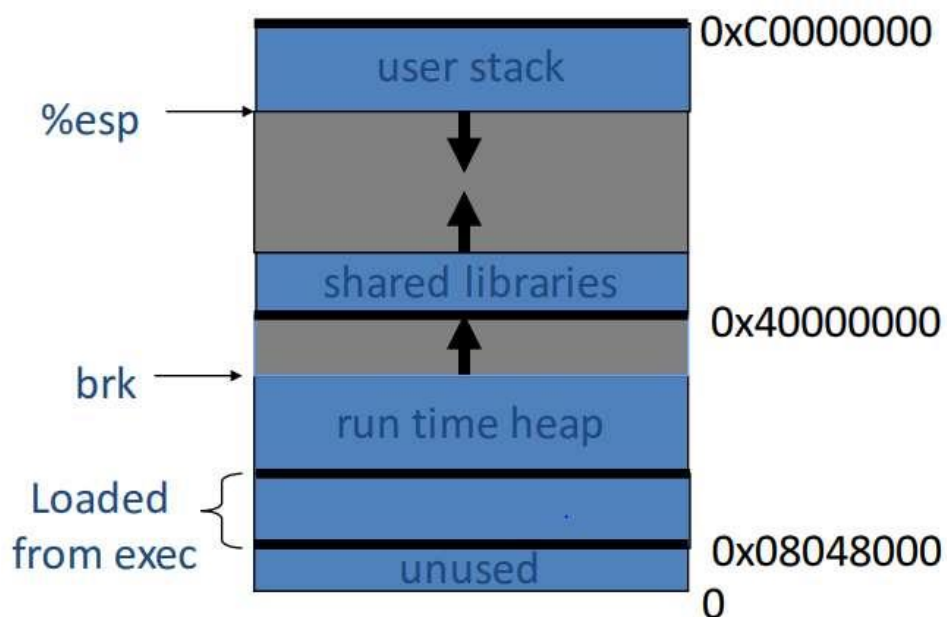
Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. For example, an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems. If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For example, 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.

## The purpose of the attacker

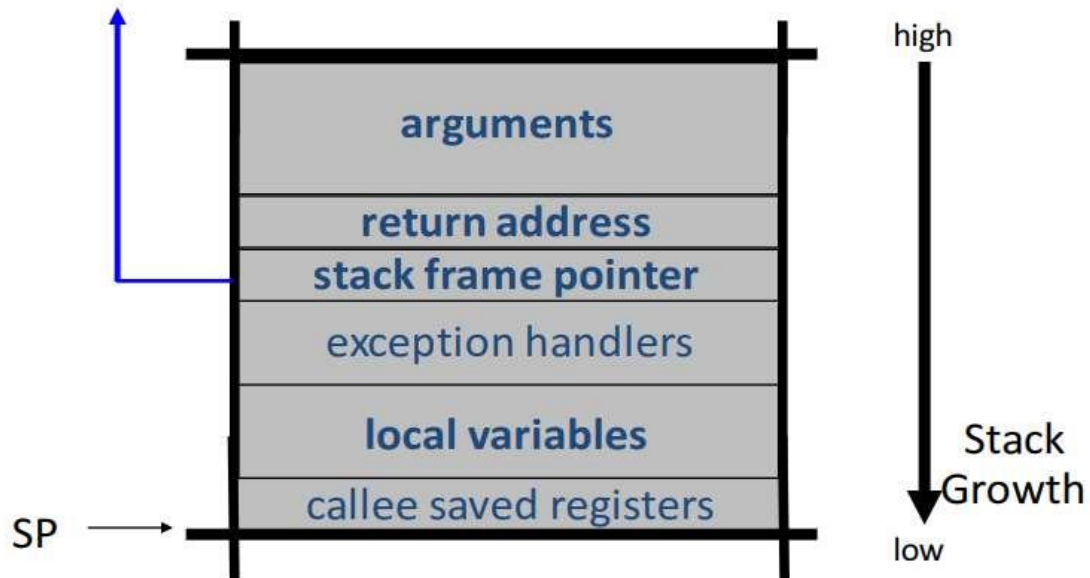
- Take control of the target machine (e.g. web server).
- Implementing pipeline execution methods by controlling the flow of the application on the target machine.
- Attacks are commonly performed on errors in C/C++ programs.
  - First major vulnerability: 1988 Internet Worm. Fingerd.
- What you need to do the attack:
  - C functions, stack and heap memory information
  - Knowledge of how to make system calls (exec() system call)
  - The attacker must know what processor architecture and operating system is on the target machine.

## Memory Layout

### Linux Process Memory Layout



## Stack Frame

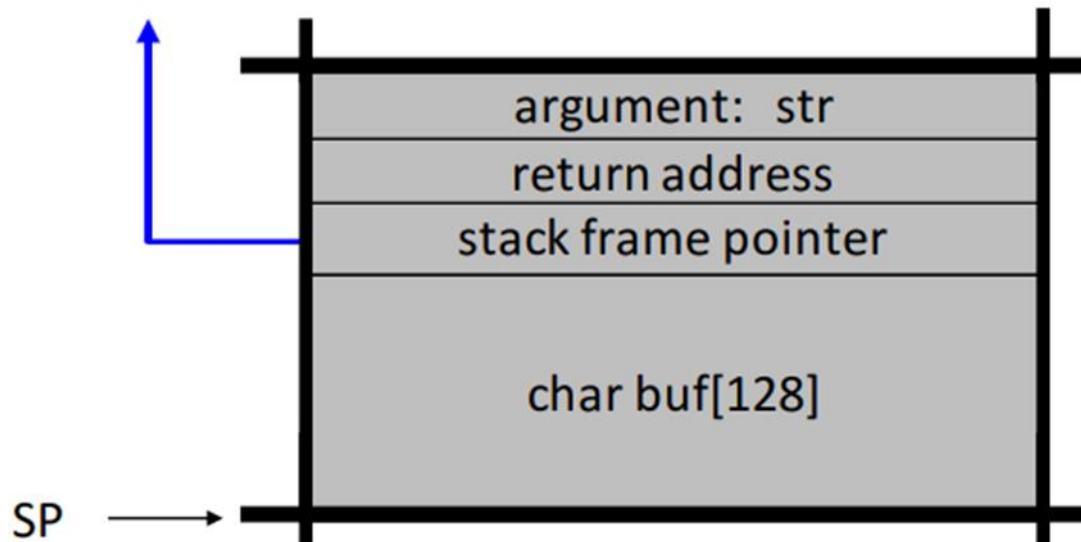


## Buffer Overflow

- Let's have a web server and a function on this server is as follows.

```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    do-something(buf);  
}
```

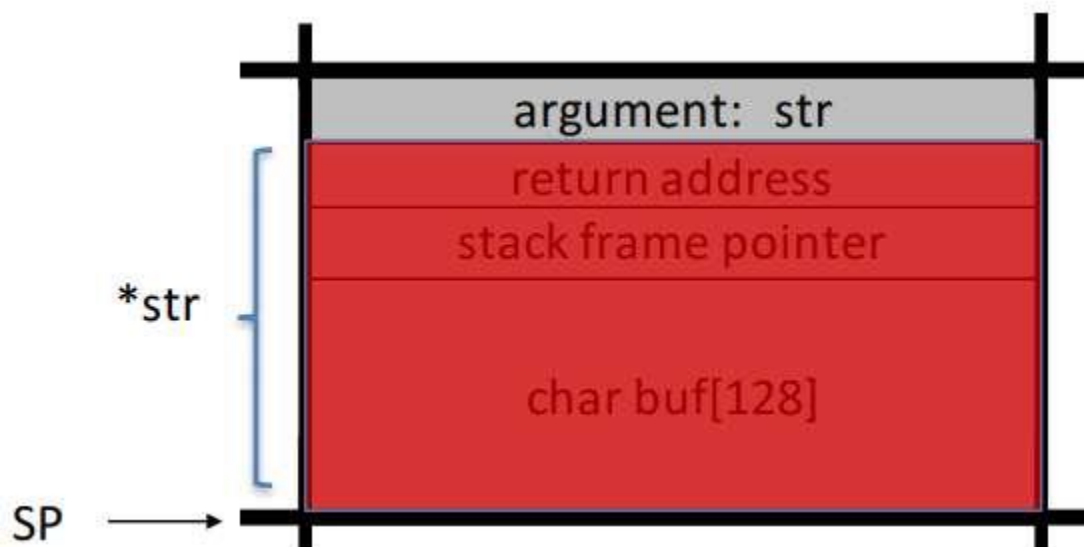
- When the following function runs, the stack frame will be like the one on the below.



- What would happen if \*str was 136 bytes long?

```
void func(char *str) {  
    char buf[128];  
  
    strcpy(buf, str);  
    do-something(buf);  
}
```

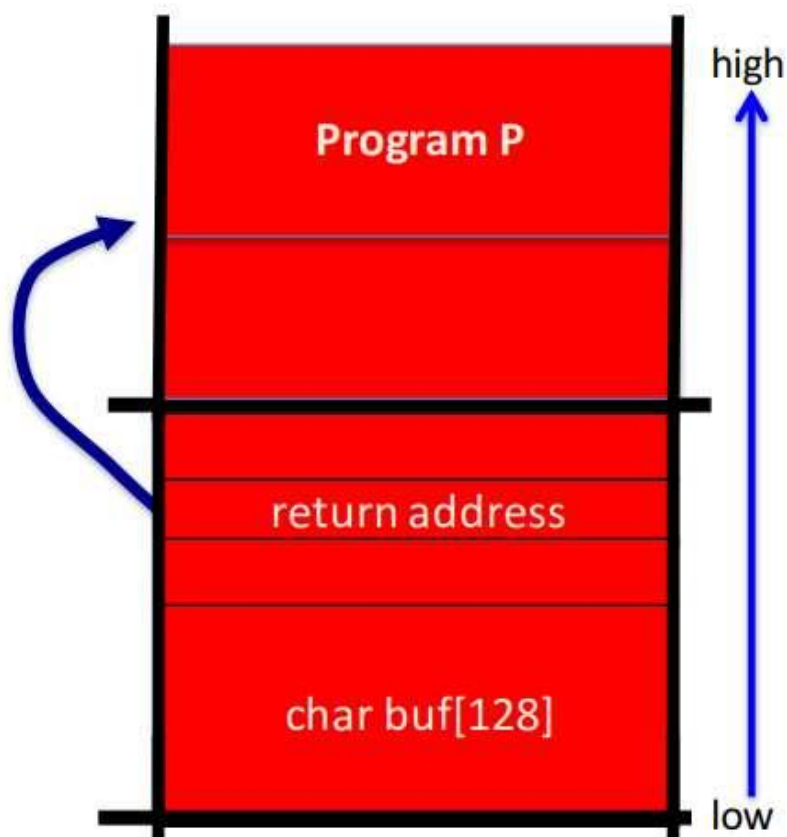
- After the strcpy function is called, the stack status is as on below;



- What is the problem with the program?
  - String lengths are not checked when the strcpy function is called.

## Exploiting the stack vulnerability

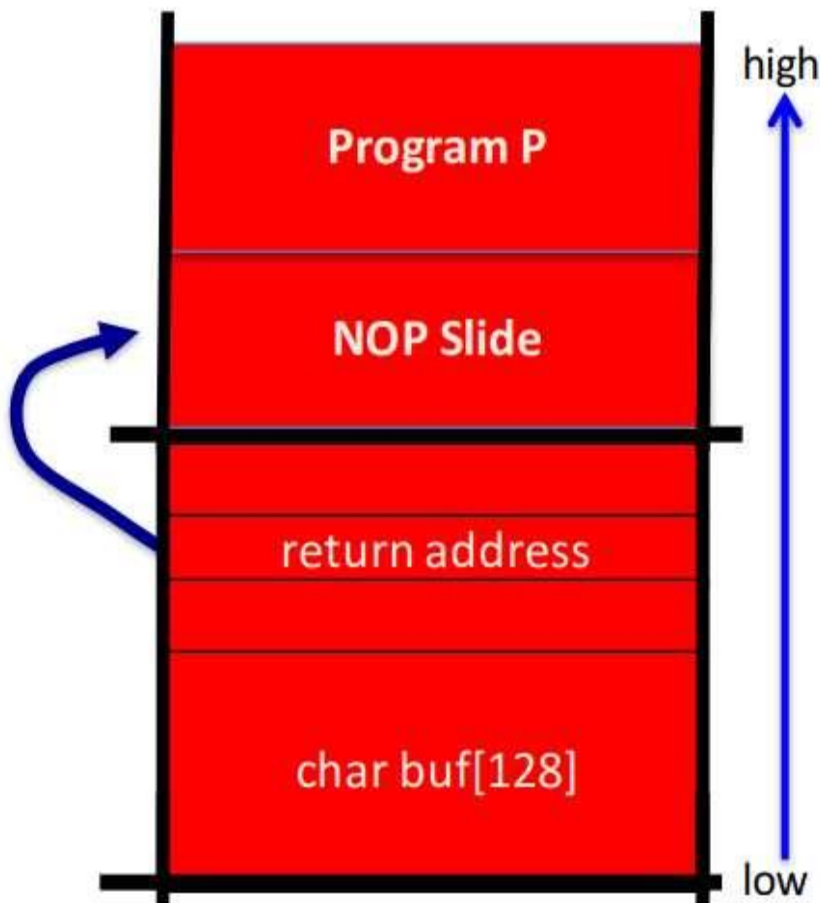
- Scenario
  - Let's set the size of \*str in such a way that the stack after the strcpy function looks like the figure.
  - If Program P running on the Stack is an application running `exec("/bin/sh")`.
  - An attacker can now access the shell with Program P and make system calls.





## How to find the return address?

- Answer: NOP slide (No operation Command)
  - Stack state can be predicted after func() is called (possible with operating system, processor architecture and C programming knowledge.)
  - It is possible to squash the return address by entering a large number of NOP commands before Program P (Hijack application, i.e. the application that runs the shell command) is executed.



## Some of the vulnerable functions

- strcpy (char \*dest, const char \*src)
- strcat (char \*dest, const char \*src)
- gets (char \*s)
- scanf (const char \*format, ... )

## Examples to be applied

1. Changing the value of variables in a function with buffer overflow.
  2. Executing a shell using the exec system call with buffer overflow.
- Architectures, Operating Systems and Compilers used in the examples. (You don't have to use these versions, but if you are using different versions you need to make necessary modifications).
    - x86 i686 processor architecture.
    - Ubuntu 16.04 operating system with security measures turned off system, Linux kernel 4.15.0-45.
    - gcc 5.4.0 C compiler is used.

## Test System

→ Modern operating systems and compilers offer effective solutions for buffer overflow.

→ In order for the examples to be applied to be understandable and easily applicable;

- Operating system and compiler settings have been changed to the format used in the early 2000s
- 32 bit operating system (Ubuntu 16.04) is used
- Address space layout randomization is turned off
  - `sudo sysctl -w kernel.randomize_va_space=0`
- Stack protection feature in C compiler is turned off
  - `-fno-stack-protector`

- The executable stack feature is activated while compiling the program.
  - -z execstack

→ Compile the sample application with the following command.

- gcc -o program -z execstack -fno-stack-protector program.c

→ Create the user folders in the directory where the program is located.

- mkdir user-1 && mkdir user-2 && mkdir admin

→ Copy the txt files to the folder.

- cp a.txt b.txt user-1/

→ Run the program.

- ./program

## Example 1: Changing another variable with buffer overflow(1)

- When the code below is run, is it possible to log into the system using a different Stack Status without knowing the user and password?

```
int main() {
    int isAdmin;
    char password[20];
    char user[20];

    while (1) {
        isAdmin = 0;

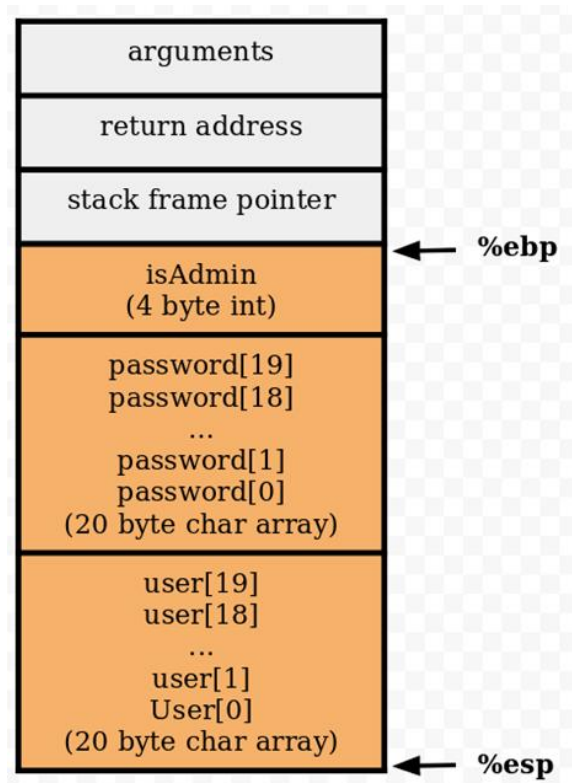
        printf("username:");
        scanf("%s", user);

        printf("password:");
        scanf("%s", password);

        if (strcmp(user, "admin") == 0 &&
            strcmp(password, "admin1234") == 0) {
            isAdmin = 1;
        }

        if (isAdmin) {
            admin_menu();
        } else {
```

## Stack Status



## Example 1: Changing another variable with buffer overflow(2)

```
int main() {  
    int isAdmin;  
    char password[20];  
    char user[20];  
  
    while (1) {  
        isAdmin = 0;  
  
        printf("username:");  
        scanf("%s", user);  
  
        printf("password:");  
        scanf("%s", password);  
  
        if (strcmp(user, "admin") == 0 &&  
            strcmp(password, "admin1234") == 0) {  
            isAdmin = 1;  
        }  
  
        if (isAdmin) {  
            admin_menu();  
        } else {
```

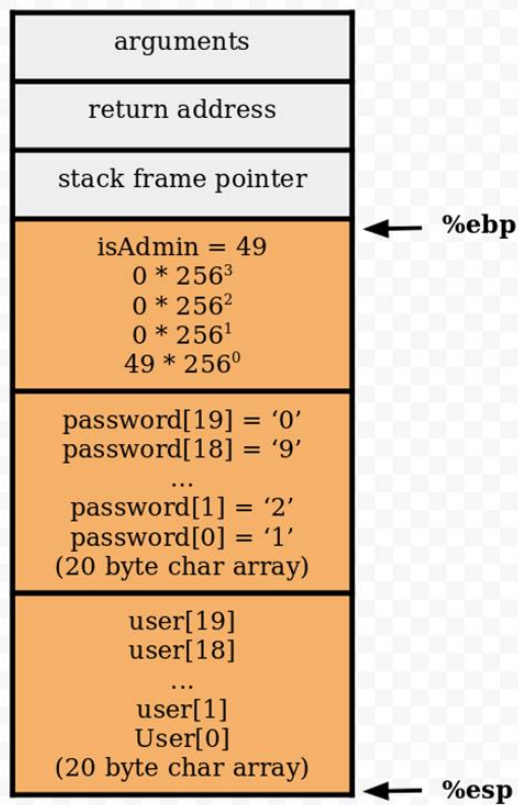
### Program Output

```
username:admin
password:aaa
incorrect username or password

username:admin
password:1234567890123456789
incorrect username or password

username:admin
password:123456789012345678901
commands:
  list
  read FILENAME
  reset
  exit
ADMIN MENU
>ADMIN MENU
>list
files:
file-1.txt
```

### Stack Status



### Example 1: Changing another variable with buffer overflow(3)

```
int main() {
    int isAdmin;
    char password[20];
    char user[20];

    while (1) {
        isAdmin = 0;

        printf("username:");
        scanf("%s", user);


        printf("password:");
        scanf("%s", password);

        if (strcmp(user, "admin") == 0 &&
            strcmp(password, "admin1234") == 0) {
            isAdmin = 1;
        }

        if (isAdmin) {
            admin_menu();
        } else {
```

#### Program Output

```
username:12345678901234567890123456789012345678901
password:a
commands:
    list
    read FILENAME
    reset
    exit
ADMIN MENU
>ADMIN MENU
>reset
delete all files & reset system!!!
ADMIN MENU
>
```

 **41 karakter**

## Stack Status

