이민규

1. The basics technic of Shellcode

1.1. Shellcode

Shellcode란, 명령 shell을 실행하여 공격자가 해당 시스템을 제어하기 때문이다. Machine code로 작성된 작은 크기의 프로그램이며 일반적으로 어셈블리어로 작성 후 기계어로 변환한다.

1.2. Assembly-Language

"int 0x80", "syscall" : "int" 명령어의 피연산자 값으로 0x80을 전달하면 eax에 있는 시스템 함수를 호출, "syscall" 명령어를 호출하면 RAX에 있는 시스템 함수를 호출한다. 32bit, 64bit 마다 시스템 함수의 콜 번호가 다르다.

1.3. Assembly code example

ASM32.asm

```
section .data                           ; 데이터 세그먼트
    msg db  "Hello, world!",0x0a, 0x0d  ; 문자열과 새 줄 문자, 개행 문자 바이트

section .text                           ; 텍스트 세그먼트
    global  _start                      ; ELF 링킹을 위한 초기 엔트리 포인트

_start:
    ; SYSCALL: write(1,msg,14)
    mov eax, 4      ; 쓰기 시스템 콜의 번호 '4' 를 eax 에 저장합니다.
    mov ebx, 1      ; 표준 출력를 나타내는 번호 '1'을 ebx에 저장합니다.
    mov ecx, msg    ; 문자열 주소를 ecx에 저장니다.
    mov edx, 14     ; 문자열의 길이 '14'를 edx에 저장합니다.
    int 0x80        ; 시스템 콜을 합니다.

    ; SYSCALL: exit(0)
    mov eax, 1      ; exit 시스템 콜의 번호 '1'을 eax 에 저장합니다.
    mov ebx, 0      ; 정상 종료를 의미하는 '0'을 ebx에 저장 합니다.
    int 0x80        ; 시스템 콜을 합니다.
```

1.4. Change to Shellcode

위의 예제는 혼자서 동작하지 않기 때문에 Shellcode가 아니다. 독립적으로 동작하기 위해서는 텍스트, 데이터 세그먼트를 사용하지 않아야 한다. 여기서 hello world 문자열을 전달하기 위해서는 데이터 세그먼트가 필요한데, 이는 call 명령어를 응용해 해결할 수 있다.

ASM32.s

```
BITS 32                         ; nasm에게 32비트 코드임을 알린다

call helloworld                 ; 아래 mark_below의 명령을 call한다.
db "Hello, world!", 0x0a, 0x0d  ; 새 줄 바이트와 개행 문자 바이트

helloworld:
    ; ssize_t write(int fd, const void *buf, size_t count);
    pop ecx         ; 리턴 주소를 팝해서 exc에 저장합니다.
    mov eax, 4      ; 시스템 콜 번호를 씁니다.
    mov ebx, 1      ; STDOUT 파일 서술자
    mov edx, 15     ; 문자열 길이
    int 0x80        ; 시스템 콜: write(1,string, 14)

    ; void _exit(int status);
    mov eax,1       ;exit 시스템 콜 번호
    mov ebx,0       ;Status = 0
    int 0x80        ;시스템 콜: exit(0)
```

call helloworld를 수행하면 스택에 그 다음 명령어의 주소를 스택에 push 하므로 db “Hello world!”, 0x0A, 0x0D의 주소를 스택에 넣게 된다. helloworld 함수에서 pop ecx를 하게 되면 문자열의 주소가 ecx에 들어가게 되어 정상적으로 문자열을 전달 할 수 있게 된다.

Build & Disassemble

```
lazenca0x0@ubuntu:~/ASM$ nasm ASM32.s
lazenca0x0@ubuntu:~/ASM$ ndisasm -b32 ASM32
00000000  E80F000000        call dword 0x14
00000005  48                dec eax
00000006  656C              gs insb
00000008  6C                insb
00000009  6F                outsd
0000000A  2C20              sub al,0x20
0000000C  776F              ja 0x7d
0000000E  726C              jc 0x7c
00000010  64210A            and [fs:edx],ecx
00000013  0D59B80400        or eax,0x4b859
00000018  0000              add [eax],al
0000001A  BB01000000        mov ebx,0x1
0000001F  BA0F000000        mov edx,0xf
00000024  CD80              int 0x80
00000026  B801000000        mov eax,0x1
0000002B  BB00000000        mov ebx,0x0
00000030  CD80              int 0x80
lazenca0x0@ubuntu:~/ASM$ hexdump -C ASM32
00000000  e8 0f 00 00 00 48 65 6c  6c 6f 2c 20 77 6f 72 6c  |.....Hello, worl|
00000010  64 21 0a 0d 59 b8 04 00  00 00 bb 01 00 00 00 ba  |d!..Y...........|
00000020  0f 00 00 00 cd 80 b8 01  00 00 00 bb 00 00 00 00  |................|
00000030  cd 80                                            |..|
00000032
lazenca0x0@ubuntu:~/ASM$
```

이를 파이썬으로 읽어 Machine code를 얻을 수 있다. 다시 이를 아래와 같은 코드를 통해 테스트할 수 있다.

shellcode.c

```
#include<stdio.h>
#include<string.h>

unsigned char shellcode [] = "\xe8\x0f\x00\x00\x00Hello, world!\n\rY\xb8\x04\x00\x00\x00\xbb\x01\x00
unsigned char code[];

void main(){
    int len = strlen(shellcode);
    printf("Shellcode len : %d\n",len);
    strcpy(code,shellcode);
    (*(void(*)()) code)();
}
```

이 프로그램을 실행해 보면 오류가 나는데, 그 원인은 Shellcode 안에 들어있는 0x00 때문이다.

가장 먼저 call 명령어에 널 바이트가 존재하므로 helloworld보다 아래에 있는 함수에 먼저 점프 한 후, helloworld를 호출하게 되면 상대적 위치가 음수가 되므로 2의 보수 표현에 의해 0이 없어진다.

ASM32-2.s

```
BITS 32         ; nasm에게 32비트 코드임을 알린다

jmp short last  ; 맨 끝으로 점프한다.
helloworld:
    ; ssize_t write(int fd, const void *buf, size_t count);
    pop ecx     ; 리턴 주소를 팝해서 ecx에 저장합니다.
    mov eax, 4  ; 시스템 콜 번호를 씁니다.
    mov ebx, 1  ; STDOUT 파일 서술자
    mov edx, 15 ; 문자열 길지
    int 0x80    ; 시스템 콜: write(1,string, 14)

    ; void _exit(int status);
    mov eax,1   ;exit 시스템 콜 번호
    mov ebx,0   ;Status = 0
    int 0x80    ;시스템 콜: exit(0)

last:
    call helloworld ; 널 바이트를 해결하기 위해 위로 돌아간다.
    db "Hello, world!", 0x0a, 0x0d  ; 새 줄 바이트와 개행 문자 바이트
```

이 이후에 없애야 할 널 바이트는 mov명령어에 있는 0x00이다. 이는 넣는 값보다 레지스터의 크기가 크기 때문에 발생하는데, 무조건 al에 숫자를 넣게 되면 그전에 값이 남아있게 되므로 sub이나 xor을 이용해 레지스터를 0으로 초기화 해 줘야 한다. 이 때 sub 명령어보다 xor명령어가 플래그를 덜 수정하기 때문에 xor을 사용하는 것이 안전하다. 이렇게 다시 코드를 작성하면 다음과 같다.

RemoveNullbyte.s

```
BITS 32         ; nasm에게 32비트 코드임을 알린다

jmp short last  ; 맨 끝으로 점프한다.
helloworld:
    ; ssize_t write(int fd, const void *buf, size_t count);
    pop ecx     ; 리턴 주소를 팝해서 exc에 저장합니다.
    xor eax,eax ; eax 레지스터의 값을 0으로 초기화합니다.
    mov al, 4   ; 시스템 콜 번호를 씁니다.
    xor ebx,ebx ; ebx 레지스터의 값을 0으로 초기화합니다.
    mov bl, 1   ; STDOUT 파일 서술자
    xor edx,edx ; edx 레지스터의 값을 0으로 초기화합니다.
    mov dl, 15  ; 문자열 길지
    int 0x80    ; 시스템 콜: write(1,string, 14)

    ; void _exit(int status);
    mov al,1    ;exit 시스템 콜 번호
    xor ebx,ebx ;Status = 0
    int 0x80    ;시스템 콜: exit(0)

last:
    call helloworld ; 널 바이트를 해결하기 위해 위로 돌아간다.
    db "Hello, world!", 0x0a, 0x0d  ; 새 줄 바이트와 개행 문자 바이트
```

Removed Null byte

```
lazenca0x0@ubuntu:~/ASM$ nasm RemoveNullbyte.s
lazenca0x0@ubuntu:~/ASM$ ndisasm RemoveNullbyte
00000000  EB15              jmp short 0x17
00000002  59                pop cx
00000003  31C0              xor ax,ax
00000005  B004              mov al,0x4
00000007  31DB              xor bx,bx
00000009  B301              mov bl,0x1
0000000B  31D2              xor dx,dx
0000000D  B20F              mov dl,0xf
0000000F  CD80              int 0x80
00000011  B001              mov al,0x1
00000013  31DB              xor bx,bx
00000015  CD80              int 0x80
00000017  E8E6FF            call word 0x0
0000001A  FF                db 0xff
0000001B  FF4865            dec word [bx+si+0x65]
0000001E  6C                insb
0000001F  6C                insb
00000020  6F                outsw
00000021  2C20              sub al,0x20
00000023  776F              ja 0x94
00000025  726C              jc 0x93
00000027  64210A            and [fs:bp+si],cx
0000002A  0D                db 0x0d
lazenca0x0@ubuntu:~/ASM$
```

shellcode2.c

```
#include<stdio.h>
#include<string.h>

unsigned char shellcode [] = "\xeb\x15\x59\x31\xc0\xb0\x04\x31\xdb\xb3\x01\x31\xd2\xb2\x0f\xcd\x80\x
unsigned char code[] = "";

void main()
{
    int len = strlen(shellcode);
    printf("Shellcode len : %d\n",len);
    strcpy(code,shellcode);
    (*(void(*)()) code)();
}
```

위 코드를 실행하게 되면 제대로 Hello world!가 출력된다.

2. Return to Shellcode

2.1. call & ret

먼저 call과 ret 명령어의 이해가 필요하다.

Instruction	Processing(?)
Call <Operation>	PUSH ReturnAddress JMP <Operation>
ret	POP RIP JMP RIP

즉, ret이 실행되기 전에 스택의 ret 주소를 변경하면 프로그램의 흐름을 변경할 수 있다는 것이다

2.2. Permissions in memory

메모리에는 권한이 설정되어 있는데, 일반적으로 gcc에서는 DEP가 설정되어 있기 때문에 데이터가 저장되는 영역에는 실행권한이 없다.

2.3. Proof of concept

poc.c

```c
#include <stdio.h>
#include <unistd.h>

void vuln(){
    char buf[50];
    printf("buf[50] address : %p\n",buf);
    read(0, buf, 100);
}

void main(){
    vuln();
}
```

위의 코드는 buf[50]에 100개 문자열까지 입력을 받으므로 Stack Overflow가 일어날 수 있다. 이 프로그램을 디버깅 하면서 ret을 실행할 순서에 rsp가 가리키고 있는 스택의 최상위 메모리는 해당 함수가 종료 후 돌아갈 주소의 값을 담고 있다. main의 ret주소와 vul의 buf 주소의 차이는 72바이트이므로 문자를 72개 이상 입력하면 main의 ret주소를 덮어 쓸 수 있다.