The basics technic of Shellcode 요약

12191604 박세웅

1. shellcode 란?

shellcode라고 불리는 이유 : 명령 shell을 실행하여 공격자가 해당 시스템을 제어하기 때문이다. shellcode는 machine code로 작성된 작은 크기의 프로그램,

shellcode는 어셈블리어 작성 후 기계어로 변경

shellcode는 세포에 침투하는 생물학적 바이러스처럼 실행중인 프로그램에 삽입된 코드를 뜻함

shellcode는 실제로 실행 가능한 프로그램은 아니다. 그러므로 shellcode를 작성할 때 메모리상 배치라 든지 메모리 세그먼트 등에 신경 쓰지 않는다.

2. The basic of shellcode

Shellcode를 할때는 c언어와 같은 고수준의 언어를 컴파일 과정에 의해 assembly, machine code와 같은 저수준 언어로 변경한다. 이렇게 machine code는 메모리에 로드되어 코드를 실행하게 된다. 즉, shelcode는 assembly, machine code와 같은 저수준 언어가 개발되어야 한다.

3. Assembly code

Assembly 명령어 공부!!

Intel syntax

Mov destination, source : 목표 피연산자에 소스 피연산자를 복사한다.

PUSH value : stack에 value 값을 저장한다.

Pop register : stack 상위의 값을 레지스터에 저장한다.

Call function_name(address) : 리턴을 위해 call 명령어의 다음 명령주소를 스택에 저장 한 후 함수의 위치로 점프를 한다.

Ret : 스택으로 부터 리턴 주소를 팝하고 그 곳으로 점프하여 함수에서 리턴 한다.

Inc destination : 목표 피연산자를 1증가 시킨다.

Dec destination : 목표 피연산자를 1감소 시킨다

add destination, value : 목표 피연산자에 value 값을 더한다.

sub destination, value : 목표 피연산자에 value 값을 뺀다.

or destination, value : 비트 or 논리 연산을 한다. 최종 결과는 목표 피연산자에 저장된다.

and destination, value : 비트 and 논리 연산을 한다. 최종 결과는 목표 피연산자에 저장된다.

xor destination, value : 비트 xor 논리 연산을 한다. 최종 결과는 목표 피연산자에 저장된다..

lea destination, source : 목표 피연산자에 소스 피연산자의 유효 주소를 로드합니다.

Assembly code 에서 시스템 함수를 호출하기 위해서는 int 0x80, syncall 명령어를 사용할 수 있습니다. Int 명령어의 피연산자 값으로 0X80을 전달하면 EAX(피연산자의 연산결과 저장소)에 저장된 시스템함수를 호출한다. Syncall 명령어를 호출하면 Rax에 저장된 시스템 함수를 호출한다.

Linux system call in assembly

Shellcode를 개발하기 위해 assembly code에서 사용 가능한 system함수들이 필요하다.

c언어 에서는 편의성과 호환성을 위해 표준 라이브러리가 제공된다.

어셈블리 언어는 어느 특정 프로세서 아키텍쳐용이라고 정해져 있다.

System call 정보는 운영체제, 프로세서의 아키텍처 마다 다르다

해당 파일에 사용한 가능한 리눅스 시스템 함수의 이름과 시스템 콜 번호가 나열돼 있다.

어셈블리로 시스템 함수를 호출 할 때는 시스템 콜 번호를 이용한다..

Save argument value in registers

System call 번호는 EAX, RAX 에 저장한다.

Kernel 의 경우 기본적으로 cdecl 함수 호출 규약을 사용하지만, 동작의 유연성을 위해 "System V ABI" Calling Convention 도 사용한다. (cdecl 함수 =c 를 선언하는 함수)

Shellcode 를 작성하는 경우 "System V ABI" Calling Convention 을 사용한다.

Build assembly code

```
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.
```

ELF 바이너리를 생성하려면 링커에게 어셈블리 명령이 어디서부터 시작하는지 알려주는 global _start 줄이 필요하다.

이제 이걸 build 하면!!

```
lazenca0x0@ubuntu:~/ASM$ nasm -f elf ASM32.asm
lazenca0x0@ubuntu:~/ASM$ ld -m elf_i386 -o hello ASM32.o
lazenca0x0@ubuntu:~/ASM$ ./hello
Hello, world!
lazenca0x0@ubuntu:~/ASM$ file ./hello
./hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
lazenca0x0@ubuntu:~/ASM$
```

-f elf인자를 nasm 어셈블리를 사용해 helloworld.asm을 어셈블해서 ELF 바이너리로 링크할 수 있는 목적파일로 만들 것이다. 링커 프로그램 ld는 이 어셈블된 목적 파일에서 실행 가능한 a.out 바이너리를 만들어 낸다.

Change to shellcode format

앞에서 개발한 프로그램은 혼자서 동작되지 않으며 링킹 과정도 필요하기 때문에 Shellcode 가 아니다.

독자적으로 동작 가능하게 변경하기 위해서 텍스트, 데이터 세그먼트를 사용하지 않는다. 함수의 호출은 call 명령어를 사용하여 “hellowworld”함수를 호출한다. Write 시스템 함수에 의해 출력 될 문자열은 해당 명령어 뒤에 작성한다.

이 부분에서 문제 발생 : write 시스템 함수에 2 번째 인자로 출력할 메시지가 저장된 주소를 전달해야 한다. Assembly Code 에서는 데이터 세그먼트를 사용하지 않기 때문에 mov 명령어를 이용해 값을 전달 할 수 없습니다. >> 해당문제는 call, ret 명령어를 사용한다.

call 명령어에 의해 함수가 호출될 때 call 명령어 다음 명령어의 주소를 스택에 저장한다.

함수의 사용이 끝난 후에 ret 명령어를 이용해 스택에 저장된 주소를 eip 레지스터에 저장한다.

이러한 구조에 의해 RET 명령어가 실행 되기전에 스택에 저장된 리턴 주소를 변경하면 프로그램의 실행 흐름을 변경 할 수 있다.

즉, call 명령어 뒤에 출력할 메시지를 저장하면 pop 명령어를 이용해 ecx 레지스터에 주소값을 전달한다.

Test

shellcode 에 생성한 shellcode 를 저장한다.

strlen() 함수에 의해 shellcode 에 저장된 문자열의 길이를 출력한다..

strcpy() 함수에 의해 shellcode 의 내용이 code 영역으로 복사된다.

code 변수의 형태를 함수 형태로 형 변화하여 "void (*function)()" 변수에 저장한다.

function() 함수를 호출하면 data 영역에 저장된 shellcode 가 실행 된다.

Debugging

```
lazenca0x0@ubuntu:~/ASM$ gcc -o shellcode -fno-stack-protector -z execstack --no-pie -m32 shellcode.c
test.c:5:15: warning: array 'code' assumed to have one element
 unsigned char code[];
               ^
lazenca0x0@ubuntu:~/ASM$ ./shellcode
Shellcode len : 2
Segmentation fault (core dumped)
lazenca0x0@ubuntu:~/ASM$
```

위의 test 방법으로 build 하면 위 그림처럼 오류가 뜬다.

디버깅을 통해 에러의 원인을 확인하자.

strcpy()함수가 호출되는 부분에 breakpoint 를 설정하고 실행한다.

strcpy()함수가 호출되기 전이기 때문에 code 변수 영역(0x804a074)에는 아무런 값이 저장되어 있지 않다..

shellcode 변수 영역에는 앞에서 작성한 값이 정상적으로 저장되어 있다.

그렇다면 shellcode 가 정상적으로 실행되지 않는 이유는 무엇일까?

```
gdb-peda$ ni

0x080484b4 in main ()
gdb-peda$ x/32bx 0x804a074
0x804a074 <code>: 0xe8    0x0f    0x00    0x00    0x00    0x00    0x00    0x00
0x804a07c:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a084:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a08c:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
gdb-peda$
```

위처럼 shellcode 에 포함되어 있는 null byte(0x00) 때문이다.

문자열을 다루는 함수들은 전달된 문자열에 null byte(0x00) 값을 발견하면 문자열의 끝이라고 판단한다.

strlen() 함수도 shellcode 의 길이가 2 라고 출력하며, strcpy()함수 또한 null byte(0x00) 이전까지의 값을 code 영역에 복사한다..

즉, shellcode 의 내용이 code 영역에 복사되지 않아 발생한 문제이다.

해당 문제를 해결하기 위해 shellcode 에 포함된 null byte 를 제거해야한다.

Remove null byte

0xE80F000000 : call dword 0x14 0xB804000000 : mov eax, 4

0xBB01000000 : mov ebx, 1 0xBA0F000000 : mov edx, 15

0xB801000000 : mov eax,1 0xBB00000000 : mov ebx,0 를 통해 null byte 제거

Call instruction

Call 명령어에 첫번째 널바이트가 존재한다. Call 명령어의 피연산자에서 사용 가능한 값의 크기는 dword 이다. "0X14"와 같이 작은 값이 사용될 경우 남은 부분에 null byte 가 포함

다음과 같이 코드를 작성함으로써 해결

Jump 명령어를 사용해 helloworld 함수를 지나 last 함수로 이동한다.

Last 함수에서는 helloworld 함수를 호출하고, 출력할 메시지를 저장한다.

```
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
```

다음과 같이 동일한 call 명령어를 사용했음에도 해당 코드에서 null byte가 제거된 이유는 음수때문이다.

jmp short 명령어에 의해 0x20 byte로 이동한다.

jmp 명령어의 피연산에서 사용 가능한 값의 크기는 dword(32bits) 이다..

jmp short 명령어의 피연산자에서 사용 가능한 값의 크기는 -128 ~ 127 이다.

call 명령어는 helloworld 함수를 호출하기 위해 -0x22(0x2 - 0x24)로 이동 해야 한다.

여기서 -0x22 를 표현하기 위해 2 의 보수로 값(0xFFFFFFDD)을 표현하게 된다..

즉, 'jmp short', 'call 음수' 명령어로 인해 null byte 가 제거 되는 것이다.

Register .

call 명령어 다음으로 null byte 가 포함되어 있는 명령어는 Register 에 값을 저장하는 부분이다.

64 bit, 32bit, 16 bit 레지스터에 표현 가능한 값보다 작은 값을 저장하게 되면 나머지 공간은 null byte 로 채워지게 된다.

shellcode 에서 64 bit, 32 비트 레지스터 영역을 사용하기 전에 레지스터의 모든 영역을 0 으로 변경하는 것이 좋다.

write 시스템 콜의 경우 인자 값을 전달 받기 위해 ebx, ecx, edx 레지스터를 사용한다.

해당 사항을 고려하지 않고 null byte 를 제거하기 위해 bl, cl, dl 레지스터에 값을 저장하면 안된다..

그 이유는 아래 예제와 같이 shellcode 가 실행되기 전에 해당 레지스터에 어떠한 값이 저장되어 있을 수 있기 때문이다.

하위 레지스터에 값을 저장한다고 해서 앞에 3 바이트가 null byte 로 변경되는 것이 아니기 때문이다.

레지스터 초기화 하기!!

sub 명령어를 이용한 초기화

sub 명령어를 이용해 자기 자신의 레지스터의 값을 뺀다.

sub 명령어는 쉘 코드 앞부분에서 사용하면 잘 동작한다.

sub 명령어는 연산 결과에 따라 OF, SF, ZF, AF, PF, CF flag 의 값이 설정된다..

이로 인해 코드가 예상과 다르게 동작할 수 있다

xor 명령어를 이용해 초기화

xor 명령은 전달된 2 개의 피연산자 값을 배타적 논리합을 수행한다.

배타적 논리합은 어떤 값이든 자신의 값을 연산하면 0 이 된다..

OF, CF flag 의 값이 지워지며, SF, ZF, PF flag 는 결과에 따라 결정된다.

AF flag 는 정의 되지 않는다.

xor 명령어가 sub 명령어 보다 flag 에 영향을 덜 주기 때문에 xor 을 이용해 레지스터 값을 초기화 하는 것이 효율적이다.

.