1. 쉘 코드

- 이 코드 조각은 리눅스 시스템에서 실행될 수 있는 쉘코드(Shellcode) 이다. 쉘코드는 보통 저수준의 어셈블리 언어로 작성되어 있으며, 특정 기능을 수행하기 위해 메모리 내에서 직접 실행되는 작은 코드 단위이다.

```
char shell[]=
"₩xeb₩x2a₩x5e₩x89₩x76₩x08₩xc6₩x46₩x07₩x00₩xc7₩x46₩x0c₩x00₩x00₩x00₩x00"
"₩xb8₩x0b₩x00₩x00₩x00₩x89₩xf3₩x8d₩x4e₩x08₩x8d₩x56₩x0c₩xcd₩x80₩xb8₩x01"
"₩x00₩x00₩x00₩xbb₩x00₩x00₩x00₩x00₩xcd₩x80₩xe8₩xd1₩xff₩xff₩xff"
"₩x2f₩x62₩x69₩x6e₩x2f₩x73₩x68"

void main() {
int *ret;
ret =(int *)&ret+2;
(*ret)=(int)shell;
}
```

- char shell[] = ...: : 시스템 호출을 통해 /bin/sh 셸을 실행하는 기능을 수행한다.
  ① ₩xb8₩x0b₩x00₩x00₩x00: 시스템 콜(execve)을 담고있는 부분
  ② ₩x2f₩x62₩x69₩x6e₩x2f₩x73₩x68: /bin/sh을 의미

- int *ret; : 정수가 저장된 메모리 주소를 가리키는 변수(리턴 주소)

- ret =(int *)&ret+2;
  ① (int *)&ret: ret 변수 자체의 메모리 주소를 가져와서(&ret) 그 주소를 '정수형 포인터'라고 알림(int *)
  ② +2 : 정수 크기(보통 4바이트)의 2배인 8바이트만큼 주소가 뒤로 이동 (포인터에 숫자를 더하면, 그 숫자에 포인터 타입의 크기를 곱한 만큼 주소가 이동함)
  ③ 이 코드는 ret 변수 자체의 주소에서 8바이트 떨어진 메모리 위치를 ret 포인터가 가리키게 만듦 - 지역 변수 ret의 주소에서 '정수 크기(4바이트)의 2배', 즉 8바이트 떨어진 곳에 함수가 끝난 후 돌아가야 할 '리턴 주소'가 저장

  * 이 +2는 절대로 고정된 값이 아님. 컴파일러가 다르면 변수들의 순서나 스택 구조가 달라질 수 있고, 운영체제나 아키텍처가 바뀌거나 심지어 컴파일러 최적화 옵션이 달라져도 리턴 주소의 위치는 변할 수 있음. 실제 공격에서는 이 'offset' 값(여기서는 +2)을 알아내려고 프로그램을 디버깅하거나 여러 번 실행해보면서 찾아냄.

- (*ret) = (int)shell;
  ① *ret: ret 포인터가 가리키는 메모리 위치(값을 저장), 즉 리턴 주소라고 생각하는 공간에 접근

② (int)shell: shell 배열 (셸코드)이 시작되는 메모리 주소를 가져와서 정수형으로 변경
③ ret 포인터가 가리키는 메모리 공간, 즉 **'리턴 주소' 값을 셸코드의 시작 주소로 변경함.**

- 함수가 끝나면 저장해뒀던 리턴 주소로 돌아가서 다음 코드를 실행. 그런데 이 코드는 리턴 주소가 저장된 메모리 공간에, 그곳에 원래 돌아가야 할 주소 대신 셸코드의 주소를 강제로 입력함. main 함수가 실행을 마치면, 프로그램은 원래 돌아가야 할 곳이 아니라 넣어둔 셸 코드의 시작 주소로 점프해서 셸 코드를 실행함.

2. int main(int argc, char *argv[])

- C 언어에서 프로그램의 진입점인 main 함수의 정의이다. 이 함수는 두 개의 매개변수를 받는다: argc와 argv.

- 이러한 매개변수를 통해 프로그램은 실행 시 외부에서 전달된 인자를 받아 처리할 수 있다. 이는 프로그램의 유연성을 높이고, 다양한 입력에 따라 동작을 변경할 수 있게 해준다.

① argc(Argument Count): argc는 프로그램이 실행될 때 전달된 인자의 개수를 나타내는 정수이다. 예를 들어, 프로그램이 ./myprogram arg1 arg2와 같이 실행되면, argc의 값은 3이 된다. (실행 파일 이름 ./myprogram과 두 개의 인자 arg1, arg2를 포함한다)
② argv(Argument Vector): argv는 문자열 포인터 배열로, 각 인자의 내용을 저장한다. argv[0]는 항상 실행된 프로그램의 이름을 가리킨다. argv[1]는 첫 번째 인자의 내용을, argv[2]는 두 번째 인자의 내용을 가리킨다. 예를 들어, 위의 예시에서 argv[0]는 ./myprogram, argv[1]는 arg1, argv[2]는 arg2가 된다.

3. 문자열 버퍼

- char *buffer = "wishfree₩n%x₩n는 문자열을 가리키는 포인터 buffer를 선언하고 초기화한다. 이 문자열은 "wishfree"라는 텍스트와 줄 바꿈 문자 ₩n, 그리고 %x 포맷 지정자와 또 다른 줄 바꿈 문자를 포함하고 있다.

- printf(buffer)는 buffer에 저장된 문자열을 출력한다. 여기서 %x는 포맷 지정자로, 정수를 16진수로 출력하는 데 사용된다. 그러나 printf에 인수로 전달된 값이 없기 때문에, 이 코드는 정의되지 않은 동작을 일으킬 수 있다. 즉, %x에 해당하는 값이 없기 때문에 프로그램이 예기치 않게 동작할 수 있다.

4. 포맷 스트링 공격의 원리

① 변수 선언 및 초기화:
- long i=0x00000064: 변수 i를 long 타입으로 선언하고, 16진수 0x64 (10진수로 100)로 초기화한다.
- long j=1: 변수 j를 long 타입으로 선언하고, 1로 초기화한다.

② 주소 출력: printf("i의 주소 : %x₩n",&i);는 변수 i의 주소를 16진수 형식으로 출력한다. &i는 i의 메모리 주소를 나타낸다.

③ 값 출력: printf("i의 값 : %x₩n",i);는 변수 i의 값을 16진수 형식으로 출력한다. 이 경우 i의 값은 0x64이므로, 출력 결과는 64가 된다.

④ 포맷 지정자 사용: printf("%64d%n₩n", j, &i);는 두 가지 작업을 수행한다.
- %64d는 j의 값을 64자리로 출력하되, 오른쪽 정렬을 한다. j의 값은 1이므로, 출력 결과는 63개의 공백 후에 1이 나타난다.
- %n은 현재까지 출력된 문자 수를 &i가 가리키는 메모리 주소에 저장한다. 이 경우, i의 값은 64자리 출력에서 64가 된다. 따라서 i의 값이 10진수 64로 변경된다.
- **i가 공격하고자 하는 취약 함수의 ret 주소 값인데, %64 부분에 공격 셀의 주소값을 계산해 넣으면 버퍼 오버플로우와 동일한 효과를 갖음**

⑤ 변경된 값 출력: printf("변경된 i의 값 : %x₩n",i);는 변경된 i의 값을 16진수 형식으로 출력한다. 이전에 i의 값이 64로 변경되었으므로, 출력 결과는 40이 된다 (16진수로 64는 40이다).