[SSG Wargame fortune_cookie]

```
v12 = &argc;
v11 = *MK_FP(__GS__, 20);
setvbuf(stdout, 0, 2, 0);
loadkey();
v9 = 100;
selector = 0;
puts("==             [Fortune Cookie]           ==");
puts("==  ADVANCED Memory Corruption Detector. ==");
puts("==          Can you break this one?      ==");
seed = time(0);
srand(seed);
while ( 1 )
{
  print_menu();
  memset(&s, 0, 0x64u);
  v3 = rand();
  v4 = rand() * v3;
  v5 = rand();
  v10 = v4 * v5;
  g_canary = v4 * v5;
  printf("> ");
  __isoc99_scanf("%d", &selector);
  __fpurge(stdin);
  if ( selector != 1 )
    break;
  printf("Input your string : ");
  input_wrap((int)&s, v9);
  printf("This is your string : %sWn", &s);
  sleep(1u);
  if ( check_canary((int)&g_canary, (int)&v10, 4) != 1 )
  {
    puts("[!] Attack Detected.WnBye :pp");
    sleep(1u);
    exit(0);
  }
}
if ( selector == 2 )
  printf("Good bye :p");
else
  puts("Wrong Number..");
sleep(1u);
result = 0;
v7 = *MK_FP(__GS__, 20) ^ v11;
return result;
```

메인함수도 짧고 메뉴도 몇 개 안되는데 심볼까지 주는 친절한 문제이다.
[ebp-0x78]에 있는 char형 변수에 input을 받으므로 여기서 stack overflow 가 일어날 것 같다. 먼저 문자열을 입력받는 input_wrap 함수를 살펴보자.

```
int __cdecl input_wrap(int a1, int a2)
{
  char v3; // [sp+7h] [bp-11h]@2
  int v4; // [sp+8h] [bp-10h]@0
  int i; // [sp+Ch] [bp-Ch]@1

  for ( i = 0; i <= a2; ++i )
  {
    v3 = getchar();
    if ( v3 == 10 )
      return i;
    *(_BYTE *)(a1 + i) = v3;
  }
  return v4;
}
```

arg1에 arg2+1만큼 입력을 받는다. 이때 arg1으로 들어오는 s는 100byte 크기의 char형 배열이고 arg2로 들어오는 값은 [ebp-0x14]에 있는 값이다. [ebp-0x14]에는 100이 들어있는데 input_wrap함수에서 한 바이트 오버플로우가 일어나 [ebp-0x14]부분을 바꿀 수 있다. [ebp-0x14]는 입력받는 사이즈를 의미하므로 0xff등 큰 값으로 덮어씌우면 s에 0x78보다 훨씬 더 많은 양의 데이터를 써넣어 ret를 덮어쓸 수 있다.

이때 스택에는 canary가 두 개가 있는데, 첫 번째 canary는 [ebp-0x10]에 있고 rand()함수로 연산한 값으로 만들어진다. 두 번째 canary는 [ebp-0xc]에 있다. 첫 번째 canary는 rand()함수를 이용해서 만들어 우회하면 되고, 두 번째 canary는 메모리 릭을 통해 우회할 수 있다.

마침 flag를 전역변수에 저장하는 함수가 들어있다.

```
int loadkey()
{
  FILE *stream; // ST1C_4@1

  stream = fopen("/home/fortune_cookie/flag", "r");
  fread(&key, 0x64u, 1u, stream);
  return fclose(stream);
}
```

```
.bss:0804A0A0                 public key
.bss:0804A0A0 key             db    ? ;
```

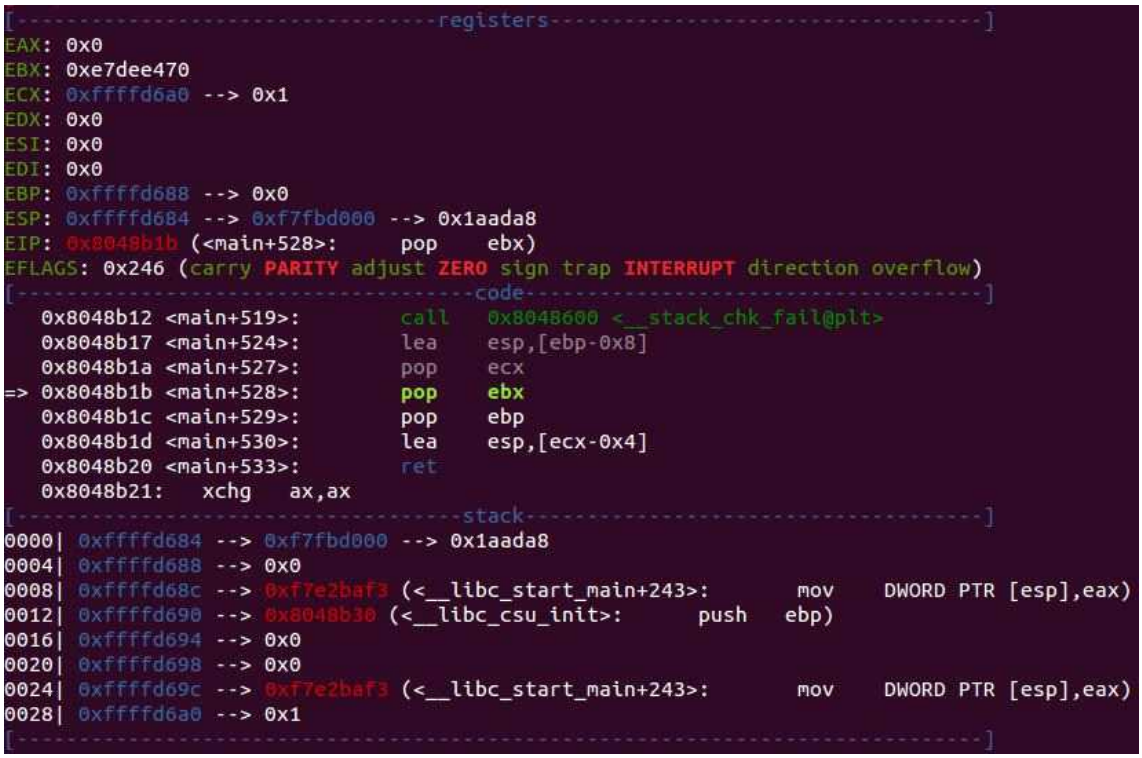
.bss영역에 flag가 들어있으므로 puts()함수 등을 이용해 flag를 출력할 수 있다.

이때 main함수가 #leave #ret를 통해 종료되는 것이 아니라 처음보는 방식으로 리턴한다는 것을 발견했다.

```
.text:08048B17                lea     esp, [ebp-8]
.text:08048B1A                pop     ecx
.text:08048B1B                pop     ebx
.text:08048B1C                pop     ebp
.text:08048B1D                lea     esp, [ecx-4]
.text:08048B20                retn
.text:08048B20 main           endp
```

위 부분의 내용으로 보아 ecx에 들어갈 부분을 수정해 주거나 [ecx-4]부분을 수정하면 eip를 잡을 수 있을 것 같다. 입력할 수 있는 값이 매우 크므로 두 번째 방법을 사용하도록 하자. 이때 ecx에 들어가는 값을 보존해 주기 위해서 메모리 릭을 통해 그 값을 알아낸 뒤에 그 자리에 덮어써 줘야 한다.(ebp-8)

ecx에 들어가는 값을 gdb로 열어보자.



원래 ret부분([ebp+4])에 들어있는 값이 [ebp+20]에도 들어있고 실제 #RET명령어를 실행할 때에 [ebp+4]가 아닌 [ebp+20]이 가리키는 곳으로 리턴하는 것을 알 수 있다. 즉, [ebp+20]부분에 우리가 원하는 주소를 넣으면 해당 주소 부분을 실행시킬 수 있다는 것이다.

Exploit 과정은 다음과 같다.
[1byte overwrite] -> [stack canary leak] -> [ecx에 들어갈 값 leak] -> [ebp+20 부분까지 overwrite] -> [get flag]

(exploit 파일은 따로 첨부)