DEFCON CTF 2009
Binary Leetness 100-500 Solutions

박 찬암 (hkpco)

chanam.park@hkpco.kr

http://hkpco.kr/

2009. 7. 4

Contents

Capture The Flag의 소개와 변화

- 매년 라스베가스에서 개최되는 DEFCON의 한 행사로, 상대방의 시스템을 공격하고 자신의 시스템을 방어하는 해킹대회

- CTF라고도 불린다

- 4년마다 운영 단체가 바뀌며 작년까지는 kenshoto,
  올해부터는 Diutinus Defense Technologies Corp(ddtek or DDTech)

- 6월경에 예선을 거치며, 상위 10개팀(작년 우승팀 포함)이 본선에 진출
  본선은 8월 초 전후를 기점으로 매년 라스베가스에서 개최된다

Capture The Flag의 소개와 변화

- 예선 문제는 빙고판에 공개
- 각 분야별로 100점부터 500점까지 5개의 문제가 제공됨

- 초록색 빙고칸은 현재 푼 문제
- 보라색 빙고칸은 우리가 현재 선택한 문제
- 파란색 빙고칸은 아직 풀지 못한 문제
- 황토색 빙고칸은 어떠한 팀도 풀지 못한 문제
- 회색은 아직 공개되지 않은 문제

- 공개된 문제를 한 팀 이상이 풀어야 그 다음 문제를 추가적으로 공개
- 만약, 아무도 문제를 풀지 못하면 다음 문제가 공개되지 않음

Capture The Flag의 소개와 변화

Capture The Flag의 소개와 변화

< 예선 문제 형식 >

- Binary, Pwn, Forensics, Trivia, Web
- Binary, Pwn, Forensics, Trivia, Realworld
- Binary, Pwn, Forensics, Trivia, Crypto, Packet

* 전체적인 시스템이나 문제 형식 등은 크게 바뀌지 않고 유지됨

Capture The Flag의 소개와 변화

Binary

주어진 바이너리를 분석하여 문제를 푸는 분야

Pwn

주어진 바이너리 분석 뒤, 리모트 공격을 수행하여 문제를 푸는 분야

Forensics

포렌식 분야

Trivia

문화적 혹은 넌센스적인 문제를 푸는 분야. 하지만, 올해부터는 조금 바뀌었음

Capture The Flag의 소개와 변화

Web

웹 상의 환경을 공격하여 문제를 푸는 분야. 2007년을 마지막으로 사라짐.

Realworld

상대적으로 좀 더 실제환경과 유사한 컨셉으로 구성되어진 분야.

2008년을 마지막으로 사라짐.

Crypto

암호학적 지식을 문제와 결부시켜 풀이를 해나가는 분야. 올해부터 생겨남.

Packet

제공되는 패킷 등을 바탕으로 분석을 통해 문제를 푸는분야. 올해부터 생겨남.

Binary Leetness 100

What is the password?

문제를 위한 바이너리가 주어졌으며, 분석을 통하여 패스워드를 찾아야 함

- **Windows**
  - IDA, Ollydbg
- **Unix-like System**
  - GDB, Objdump

Binary Leetness 100

- file 명령을 통한 기본 조사

[hkpco@servera122 CTF]$ file bin100

bin100: **ELF** 32-bit LSB executable, Intel 80386, version 1 **(GNU/Linux)**, **statically linked**, **stripped**

- ELF 파일포멧의 Linux 바이너리이며
- 정적 컴파일(static compile) 하였고
- Stripped(디버깅 정보가 삭제) 되어있다.

Binary Leetness 100

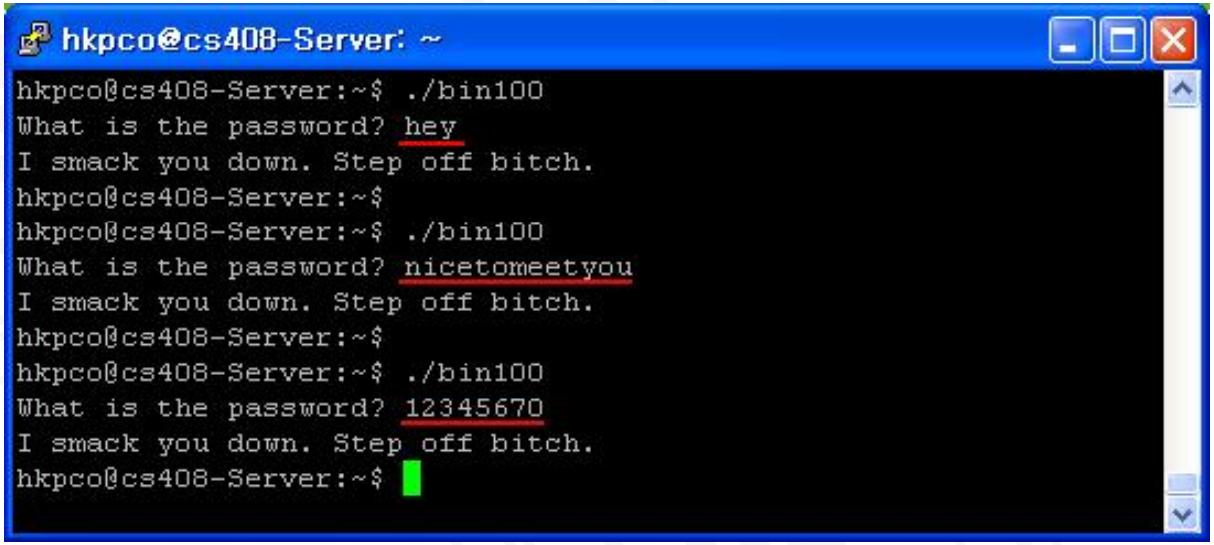
- strings 명령을 통한 추가 정보 수집

```
......
/X3H
GCC: (Debian 4.3
u_lO_stdin_used$
checklong_real
h_im
......
$Info: This file is packed with the UPX executable packer http://upx.sf.net $
$Id: UPX 3.01 Copyright (C) 1996-2007 the UPX Team. All Rights Reserved. $
PROT_EXEC|PROT_WRITE failed.
/proc/se
lf/exe
```

Binary Leetness 100

```
hkpco@cs408-Server:~$ ./upx -d bin100
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2008
UPX 3.03        Markus Oberhumer, Laszlo Molnar & John Reiser   Apr 27th
  2008

        File size         Ratio      Format      Name
   --------------------   ------   ----------   ----------
upx: bin100: Exception: checksum error

Unpacked 1 file: 0 ok, 1 error.
```

Binary Leetness 100

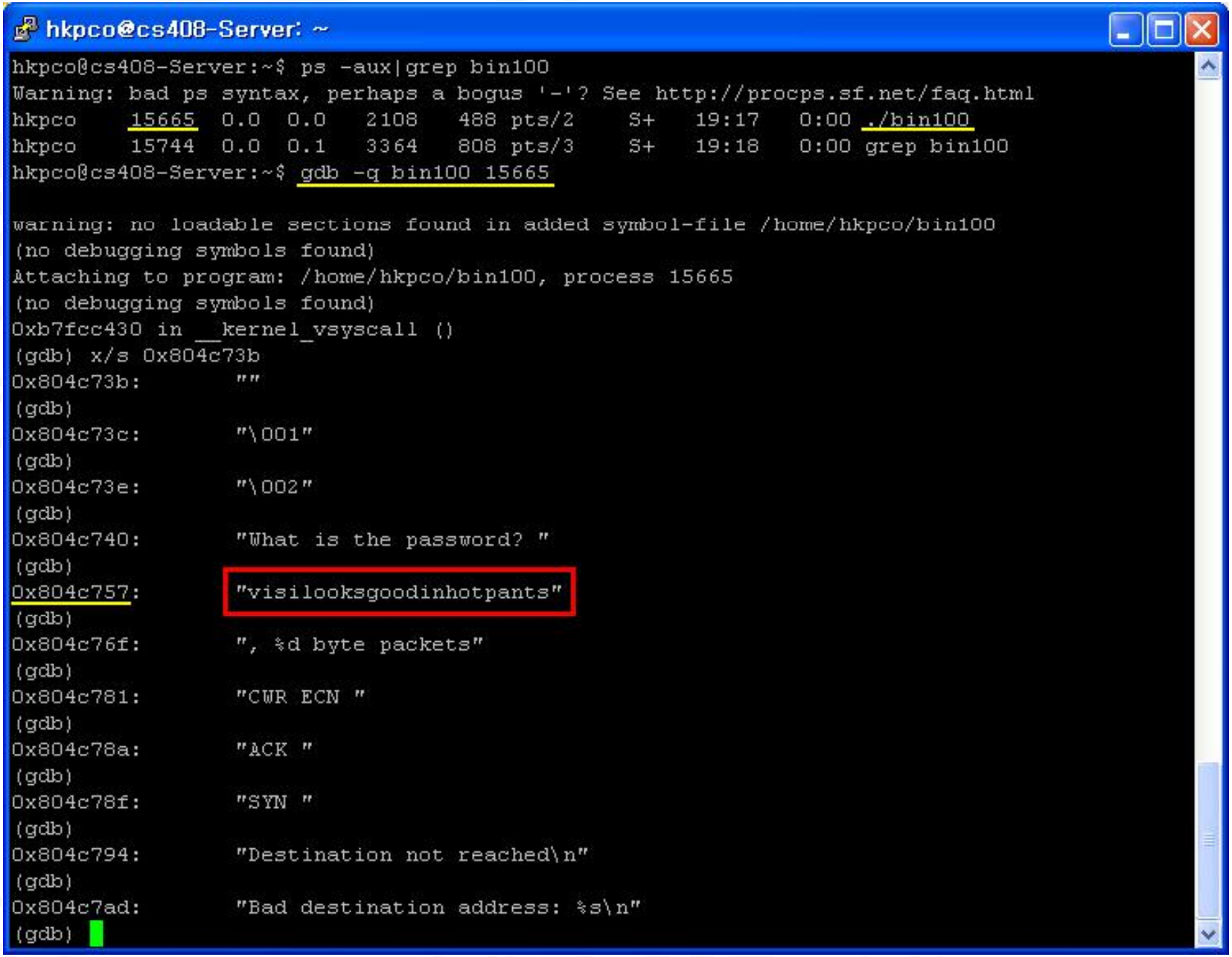


- 모든 패킹, 코드 암호화 등은 실행 과정에서 정상적인 형태를 취해야 함

- 그러므로, bin100 바이너리 실행 시 우선적으로 패킹이 풀릴 것이며
- 입력을 받는 과정에서 이미 메모리에 해당 패스워드가 존재할 것이라 예상

Binary Leetness 100

Binary Leetness 100



Binary Leetness 200

Name the algorithm implemented in this file. Note that several smaller algorithms may be utilized in implementing the overall solution.
The answer is the overall algorithm.

- 주어진 바이너리를 분석하여 알고리즘의 이름을 맞추는 문제

< 3가지 방법 >

  - Brute force
  - Signature
  - Analysis

Binary Leetness 200

- Bruteforce
  - 문제 출제자의 원래 의도가 바이너리 파일을 정석으로 분석하여 해당 알고리즘을 도출하는 것이라면, 굉장히 생소한 알고리즘을 문제로 내지는 않았을 것이기 때문에 대표적인 알고리즘 리스트를 이용하여 무차별 대입
- Signature
  - 각 알고리즘마다 루프 횟수, 할당 크기, 주 사용 자료구조 등 구분 지을만한 특정한 시그니처가 있을 것이기 때문에 이를 이용하여 구글 등의 검색으로 도출
- Analysis
  - 바이너리를 정식으로 분석하여 해당 알고리즘을 찾아냄

Binary Leetness 200

Huffman coding

In computer science and information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

From Wikipedia, the free encyclopedia

Binary Leetness 200

< Huffman Code >

- 무손실 데이터 압축 알고리즘
- 데이터에 특성에 따라 20 ~ 90% 까지 절약 가능
- 문자의 빈도 수에 따라 그 크기가 달라지는 *가변 길이 코드(variable-length code)*를 사용하여 기존 *고정 길이 코드(fixed-length code)*를 절약

Binary Leetness 200

- 가변 길이 코드(variable-length code) 테이블

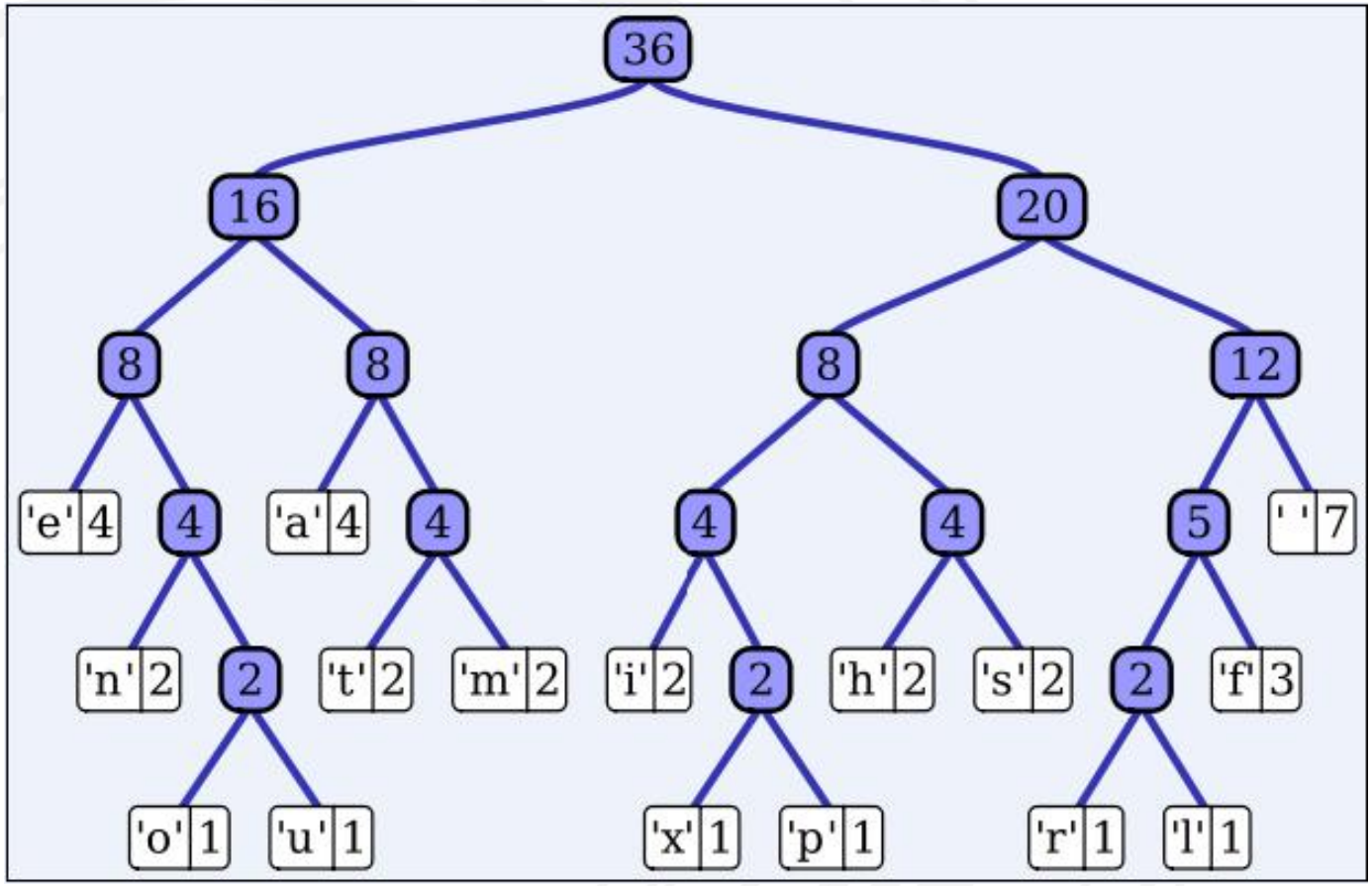
	a	b	c	d	e	f
빈도수	4500	1300	1200	1600	900	500
고정 길이 코드	000	001	010	011	100	101
가변길이 코드	0	101	100	111	1101	1100

- 고정 길이 코드: (4500 + 1300 + 1200 + 1600 + 900 + 500) * 3 = 300,000
- 가변 길이 코드: (4500*1 + 1300*3 + 1200*3 + 1600*3 + 900*4 + 500*4) = 224,000

  (약 25% 절약)

Binary Leetness 200

< Huffman Tree >



Binary Leetness 200

< 간단한 과정 >

- 1. 대상 파일의 모든 문자들의 빈도수를 구한다.
- 2. 빈도 수에 따라 코드의 길이를 재지정한다.
  - 이 과정에서 tree가 사용됨
    - 트리는 linked-list를 통하여 구현

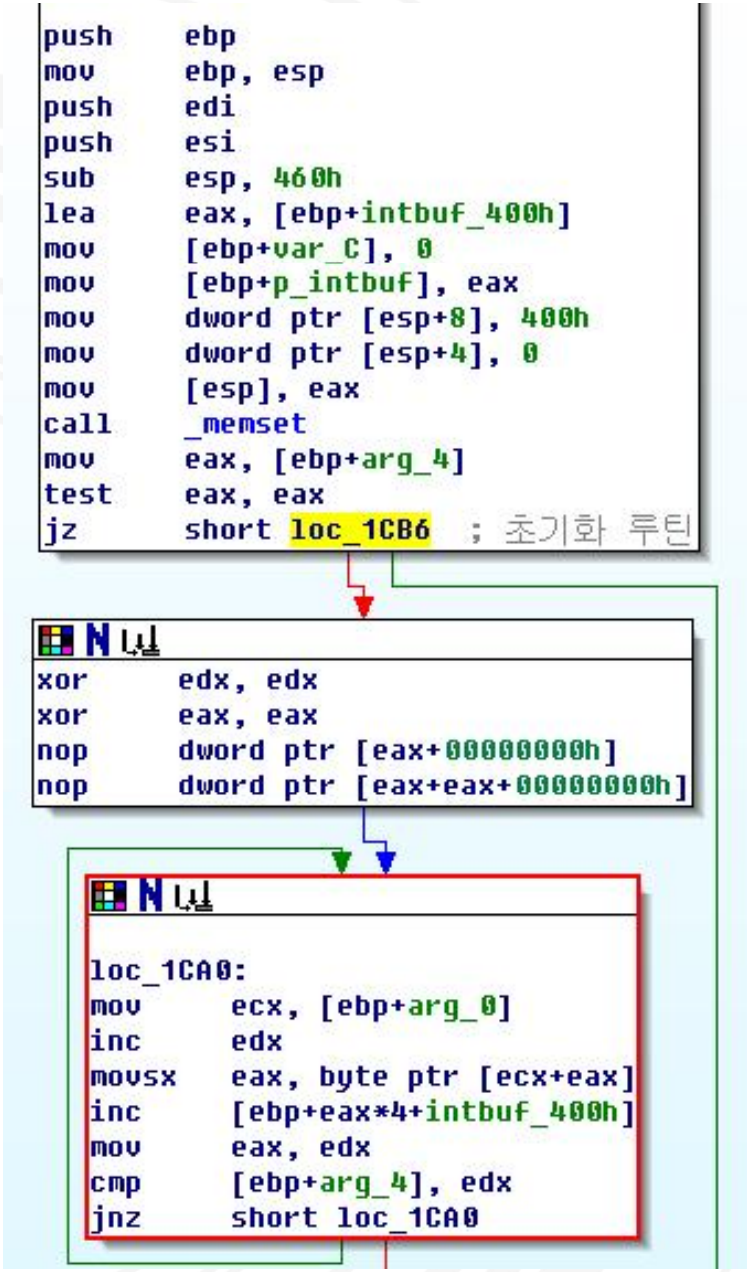
Binary Leetness 200

< 간단한 과정 >

- 1. 대상 파일의 모든 문자들의 빈도수를 구한다.

- 2. 빈도 수에 따라 코드의 길이를 재지정한다.
  - 이 과정에서 tree가 사용됨
    - 트리는 linked-list를 통하여 구현

Binary Leetness 200



Binary Leetness 200

```
text:00001C8D                xor     edx, edx
text:00001C8F                xor     eax, eax
; 레지스터 초기화

text:00001CA0 loc_1CA0:
text:00001CA0                mov     ecx, [ebp+arg_0]
; 대상 base address저장

text:00001CA3                inc     edx
; 루프를 위한 edx 레지스터 증가
```

Binary Leetness 200

```
text:00001CA4                   movsx   eax, byte ptr [ecx+eax]
 ; base address에서 루프 카운터 eax 만큼 더한 offset을 eax에 저장

text:00001CA8                   inc     [ebp+eax*4+intbuf_400h]
 ; 대상 문자에 해당하는 변수공간의 카운터 1 증가
 ; 카운터를 저장하는 변수가 4byte이기 때문에 *4

/*
  intbuf_400h는 4byte 변수 100h(256)개가 저장되어 있는데,
  이는 ASCII 코드의 문자 집합이 총 2^8(256)개이기 때문
*/
```

Binary Leetness 200

```
text:00001CAF                mov     eax, edx
; 카운터 값을 eax에 저장

text:00001CB1                cmp     [ebp+arg_4], edx
text:00001CB4                jnz     short loc_1CA0
; 함수 호출 당시 지정된 길이와 비교하여 같지 않다면 루프 재수행
```

! 결국, **intbuf_400h** 공간에 전체 문자의 카운터 값이 저장됨

Binary Leetness 200

< 간단한 과정 >

- 1. 대상 파일의 모든 문자들의 빈도수를 구한다.

- 2. 빈도 수에 따라 코드의 길이를 재지정한다.
  - 이 과정에서 tree가 사용됨
    - 트리는 linked-list를 통하여 구현

Binary Leetness 200

- Huffman Coding에 사용되는 구조체

```
typedef struct _huf
{
        long count;       // 빈도수
        int  data;        // 문자
        struct _huf *left, *right
} huf;
```

-> 총 16byte

Binary Leetness 200

```
void construct_trie(void)
{
…
  /* 초기 단계 */
  for ( i = nhead = 0; i < 256; i++)
  {
…
        if((h = (huf*)malloc(sizeof(huf))) == NULL)          // 16byte 할당
…
        h->count = freq[i];          // 빈도수 저장
        h->data = i;                 // 문자 저장
    …
      }
  }
…
…
```

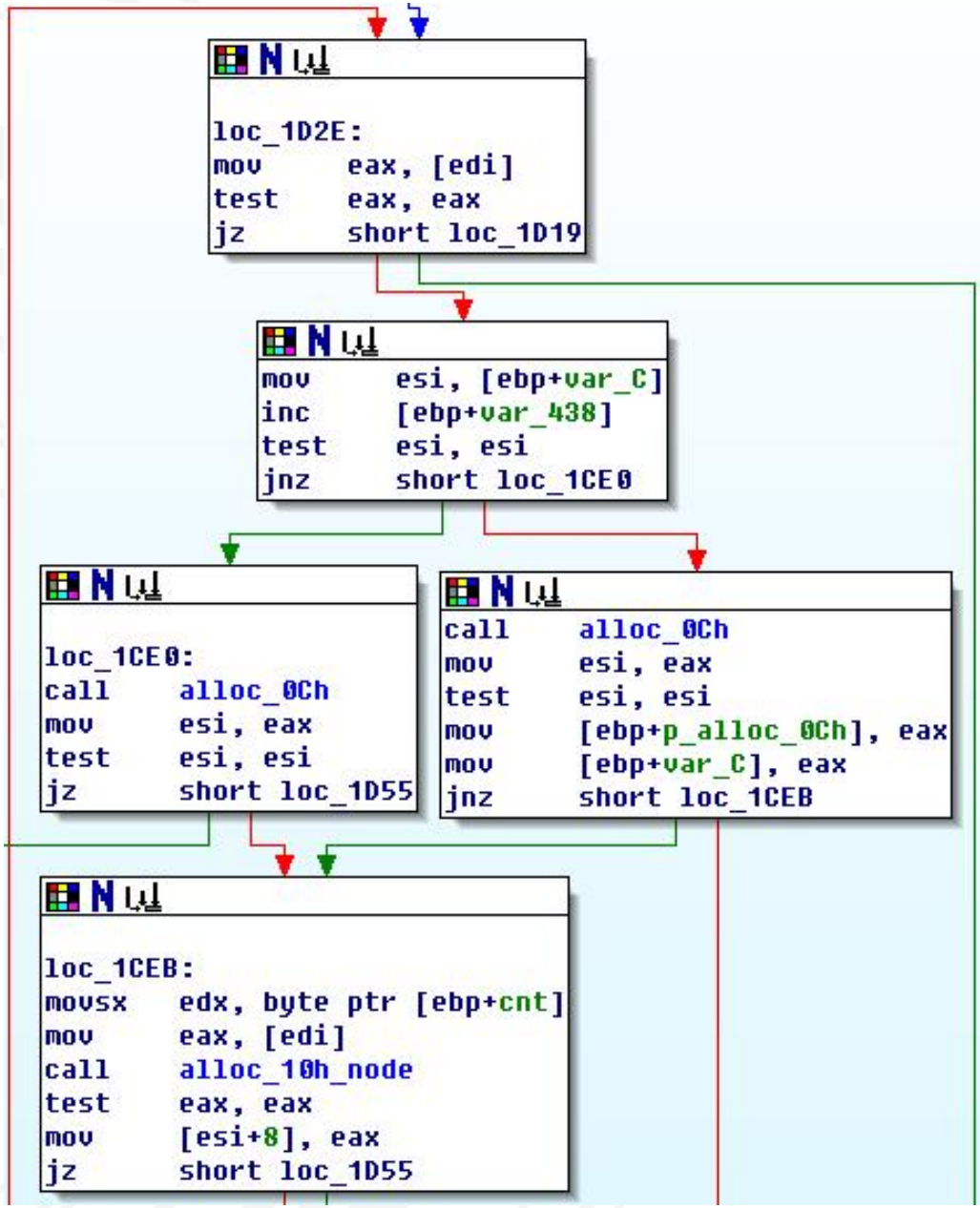
Binary Leetness 200

다음 범위가

Huffman Tree를

위한 초기화 과정

[00001D2E ~ 00001D2C]



Binary Leetness 200

```
text:00001CB6                mov     edi, [ebp+p_intbuf]
; 각 문자의 빈도수가 존재하는 intbuf 배열의 주소를 edi에 저장

text:00001CBC                mov     [ebp+cnt], 0
; 카운터 값 초기화
```

Binary Leetness 200

```
text:00001CEB                movsx   edx, byte ptr [ebp+cnt]
; 카운터 값을 edx 레지스터에 저장

text:00001CF2                mov     eax, [edi]
; 현재 가리키고 있는 intbuf의 특정 오프셋 주소를 eax에 저장

text:00001CF4                call    alloc_10h_node
; alloc_10h_node 함수 호출
```

Binary Leetness 200

- **(Alloc_10h_node 함수 내부)**

```
text:00001A06                mov     dword ptr [esp], 10h
…
text:00001A17                call    _malloc
; huffman tree를 위한 구조체 크기인 10h 만큼 공간 할당

text:00001A1C                test    eax, eax
text:00001A1E                jz      short loc_1A3C
; 실패시 점프
```

Binary Leetness 200

- **(Alloc_10h_node 함수 내부)**

```
text:00001A20                mov     edx, edi
; intbuf의 특정 오프셋을 가리키는 주소를 edx로 이동

text:00001A22                mov     dword ptr [eax+0Ch], 0
text:00001A29                mov     dword ptr [eax], 0
text:00001A2F                mov     dword ptr [eax+4], 0
; 구조체 초기화
```

Binary Leetness 200

- **(Alloc_10h_node 함수 내부)**

```
text:00001A36                   mov     [eax+8], esi
; esi 레지스터(intbuf의 특정 오프셋을 가리키고 있으며, 빈도수 값)의 값을
; 구조체의 8h번째 오프셋에 저장

text:00001A39                   mov     [eax+0Ch], dl
; dl 레지스터(문자 카운터 값)의 값을 구조체의 0Ch번째 오프셋에 저장
```

Binary Leetness 200

```
text:00001D19                inc     [ebp+cnt]
; 카운터 증가

text:00001D1F                add     edi, 4
; 각 문자의 빈도수를 저장하는 배열 카운터 증가

text:00001D22                cmp     [ebp+cnt], 100h
text:00001D2C                jz      short loc_1D61
; 100h(256)개가 아니라면 루프의 처음으로 점프
```

Binary Leetness 200

- 모든 문자에 대한 빈도수를 구함
- 트리를 사용하며 해당 구조체는 16byte
- 트리의 초기 과정에서 구조체에 문자와 해당 문자의 빈도수를 저장하며 이는 Huffman Tree의 초기 과정과 동일
- 이러한 과정을 총 256번(아스키의 빈도수와 동일) 수행

Binary Leetness 200

It’s you!, Huffman baby~

Binary Leetness 300

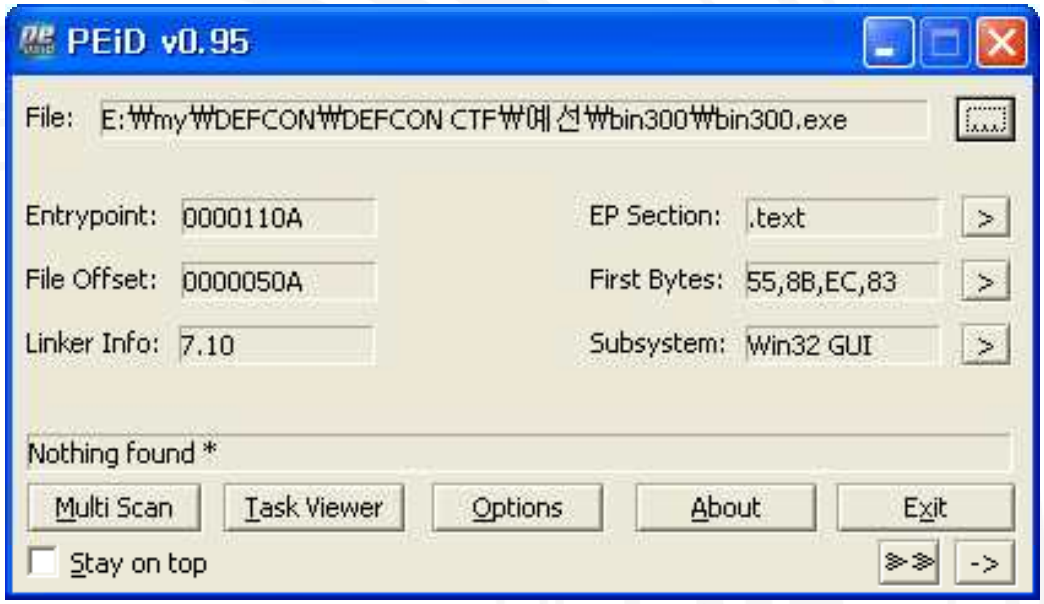
The answer is the 16 byte decryption key used in the file linked here. Enter in the form: \xaa\xbb\xcc…

바이너리에서 사용되는 16byte Decryption Key를 찾는 문제

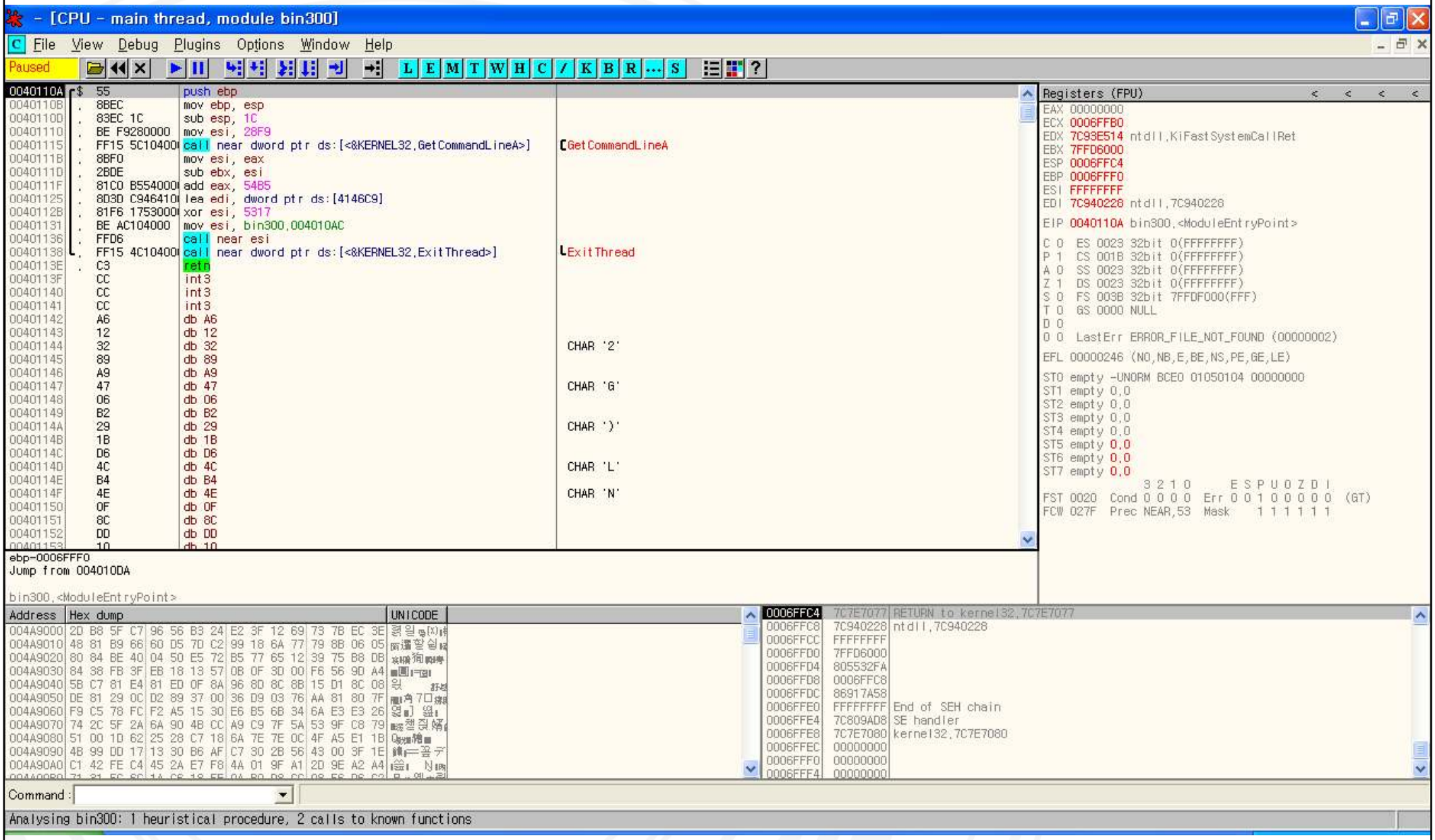
- Windows 실행파일

```
[hkpco@servera122 ~]$ file bin300
bin300: MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-bit
```

Binary Leetness 300



Binary Leetness 300



Binary Leetness 300

< binary 300의 개략적 실행 흐름 **>**

① 930000, 990000 메모리 할당

② 930000, 990000에 데이터를 써넣음

③ 930000 영역 내부 코드(9303BD)가 990000 영역 메모리 복호화

④ 400000 메모리 할당

⑤ 930000루틴이 990400의 데이터를 (407000~407000*(E00*4))에 써넣음(0040A590는 UPX 구문)

⑥ CreateRemoteThread로 해당 UPX 루틴 실행

⑦ helloworld.txt 파일에 “Hello world!”라는 문자열을 기록

사용된 기술 - *코드 난독화, 코드 암호화, 패킹*

Binary Leetness 300

< binary 300의 개략적 실행 흐름 **>**

① 930000, 990000 메모리 할당

② 930000, 990000에 데이터를 써넣음

③ 930000 영역 내부 코드(9303BD)가 990000 영역 메모리 복호화

④ 400000 메모리 할당

⑤ 930000루틴이 990400의 데이터를 (407000~407000*(E00*4))에 써넣음(0040A590는 UPX 구문)

⑥ CreateRemoteThread로 해당 UPX 루틴 실행

⑦ helloworld.txt 파일에 “Hello world!”라는 문자열을 기록

사용된 기술 - *코드 난독화, 코드 암호화, 패킹*

Binary Leetness 300

< binary 300의 개략적 실행 흐름 >

① A, B 메모리 할당

② A, B에 각각 데이터를 써넣음

③ A 영역이 암호화 된 B 영역 데이터를 복호화

④ C 메모리 할당

⑤ A 루틴이 복호화 된 B의 데이터를 C에 써넣음

⑥ CreateRemoteThread로 C영역 루틴 실행

⑦ helloworld.txt 파일에 “Hello world!”라는 문자열을 기록

사용된 기술 - *코드 난독화, 코드 암호화, 패킹*

Binary Leetness 300

< 분석 시 장애물 >

- 코드 난독화가 이루어져 있기 때문에 분석 툴이 어셈블리 코드를 제대로 해석하지 못함
- 프로그램의 엔트리 포인트를 엉뚱한 곳으로 지정
- 패킹

Binary Leetness 300

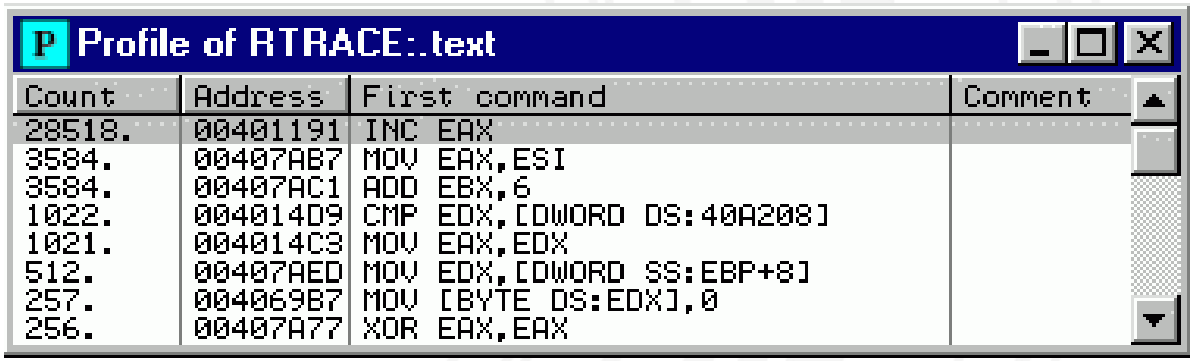
< 분석 방법론 >

- 코드 난독화, 암호화, 패킹 등 어떠한 방법이든 실행 시에는 모두 그 원래의 형태로 진행되어야 함
- 여기서는 그러한 특성을 이용하여 동적 분석으로 쉽게 해결할 수 있음

Binary Leetness 300

두 가지 방법(더 있겠지만 여기서 언급하는것에 해당)

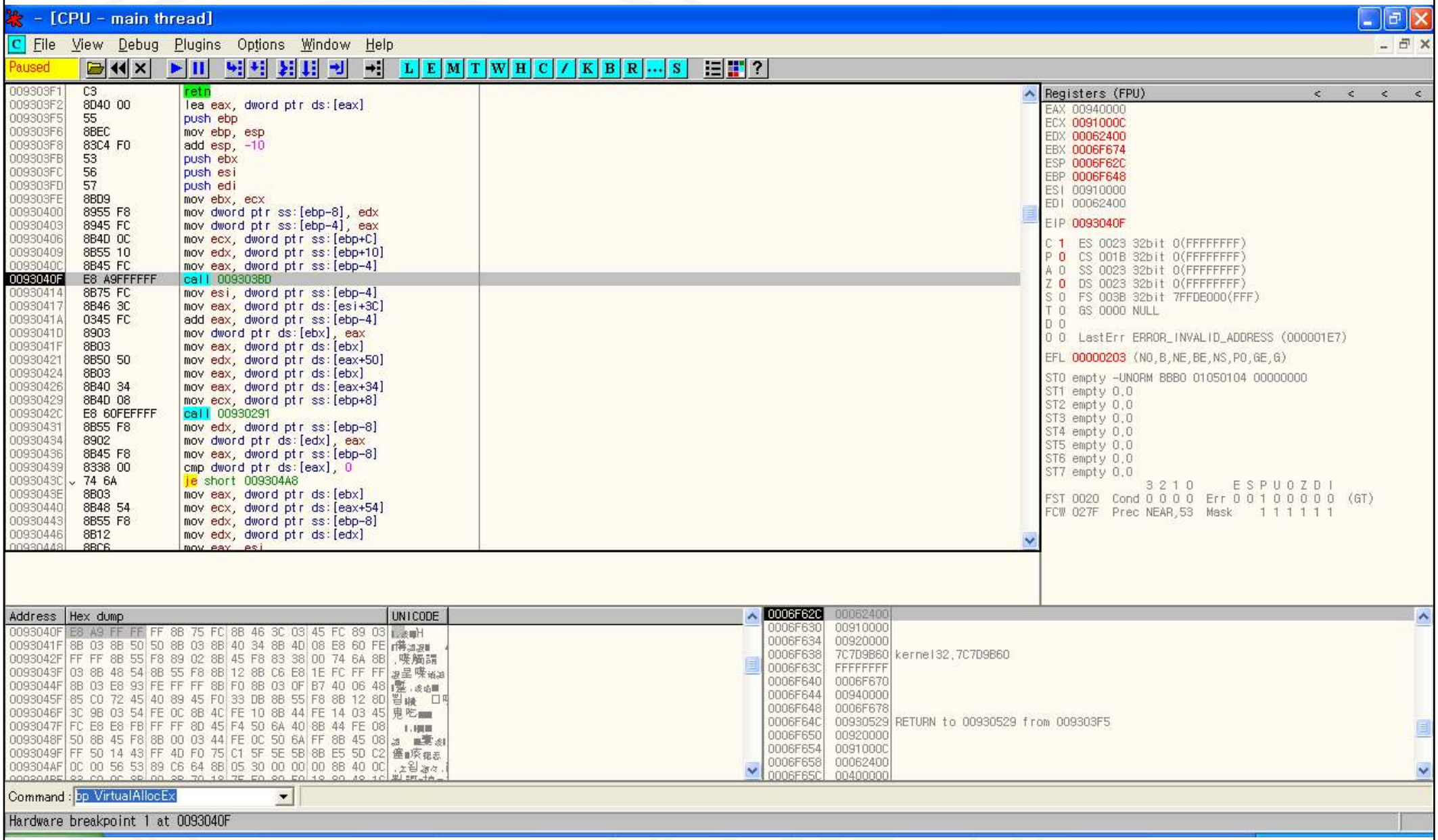
- Run trace
  - 프로그램을 직접 실행하며 해당 명령의 실행 카운터 등을 구해주는 Ollydbg의 기능
  - 참가 팀중 한 명이 해당 방법으로 풀이한 보고서를 작성
  - 복호화 과정 시 동일한 명령의 반복 루틴이 많을것이라 예상하고 Run trace를 이용하여 빈도수가 많은 명령 근처에서 복호화 루틴을 쉽게 찾음



Binary Leetness 300

- Breakpoint의 활용
  - 바이너리 내부의 API가 아닌 DLL 파일 자체의 API에 breakpoint를 걸어두고 실행 시 breakpoint가 걸리는 지점 근처의 루틴을 분석
  - 300번 문제의 경우는 패킹을 푸는 과정에서 암호화 된 루틴을 복호화 시키는 등의 진행을 거치고 있음
  - 바이너리가 실행압축 되어있다고 가정하고 그에 자주 사용되는 API인 VirtualAllocEx()에 breakpoint를 걸어둔 뒤 분석

Binary Leetness 300

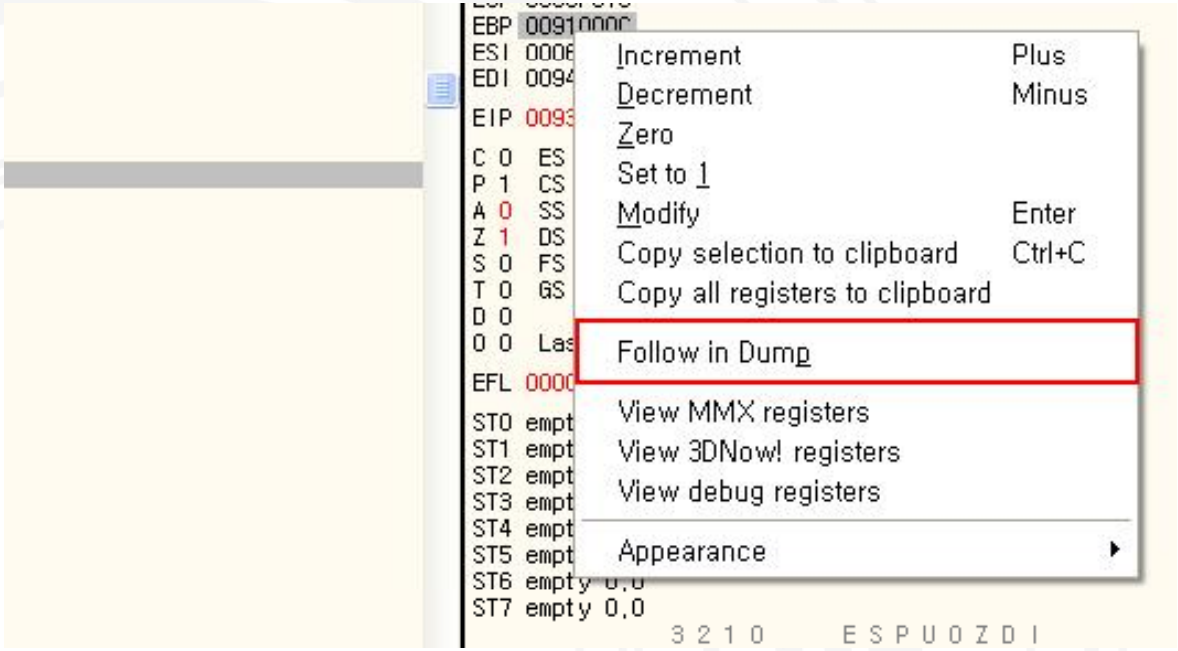


Binary Leetness 300

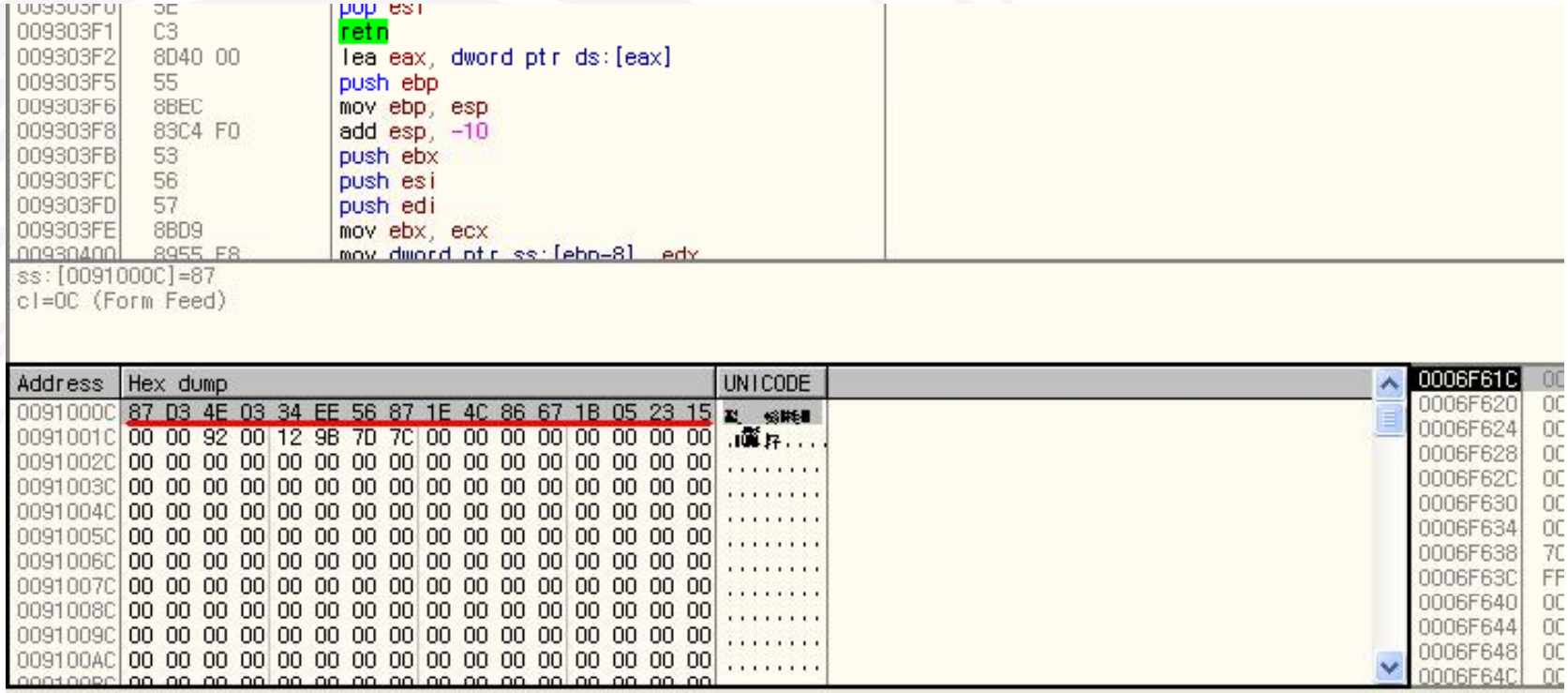
```
009303C2  74 2A       je short 009303EE
009303C4  8BF8        mov edi, eax
009303C6  8BE9        mov ebp, ecx
009303C8  33C0        xor eax, eax
009303CA  8BF2        mov esi, edx
009303CC  4E          dec esi
009303CD  85F6        test esi, esi
009303CF  72 1D       jb short 009303EE
009303D1  46          inc esi
009303D2  33D2        xor edx, edx
009303D4  8A4C05 00   mov cl, byte ptr ss:[ebp+eax]
009303D8  300C17      xor byte ptr ds:[edi+edx], cl
009303DB  8BCA        mov ecx, edx
009303DD  32C8        xor cl, al
009303DF  300C17      xor byte ptr ds:[edi+edx], cl
009303E2  40          inc eax
009303E3  83F8 10     cmp eax, 10
009303E6  75 02       jnz short 009303EA
009303E8  33C0        xor eax, eax
009303EA  42          inc edx
009303EB  4E          dec esi
009303EC  75 E6       jnz short 009303D4
009303EE  5D          pop ebp
009303EF  5F          pop edi
009303F0  5E          pop esi
009303F1  C3          retn
009303F2  8D40 00     lea eax, dword ptr ds:[eax]
009303F5  55          push ebp
009303F6  8BEC        mov ebp, esp
009303F8  83C4 F0     add esp, -10
009303FB  53          push ebx
009303FC  56          push esi
009303FD  57          push edi
009303FE  8BD9        mov ebx, ecx
00930400  8955 E8     mov dword ptr ss:[ebp-8], edx
```

```
Registers (FPU)
EAX 00000000
ECX 0091000C
EDX 00000000
EBX 0006F674
ESP 0006F61C
EBP 0091000C
ESI 00062400
EDI 00940000

EIP 009303D4
```

Binary Leetness 300



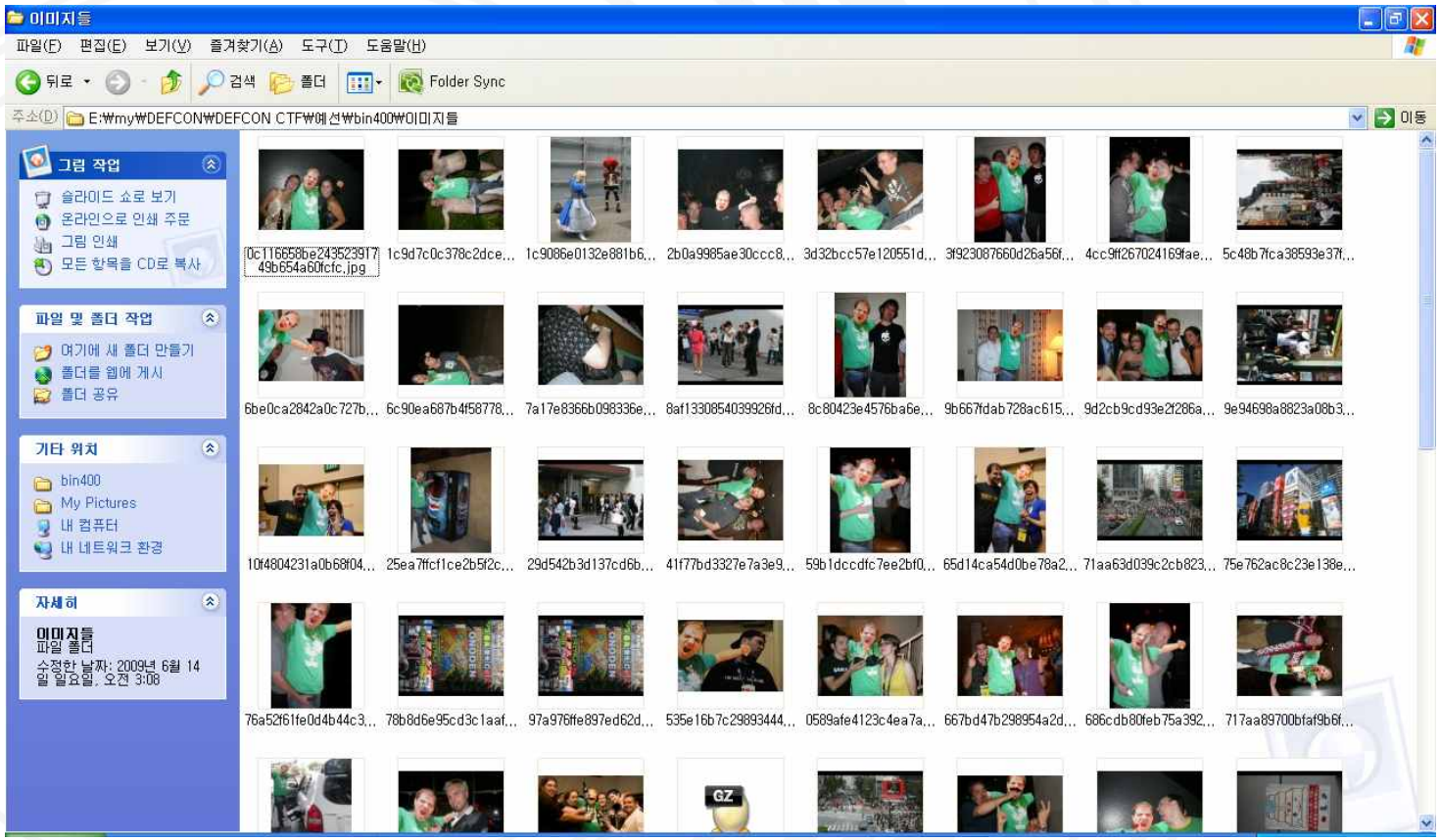
Binary Leetness 300



Binary Leetness 400

The answer to your second questions is ‘by hash’.

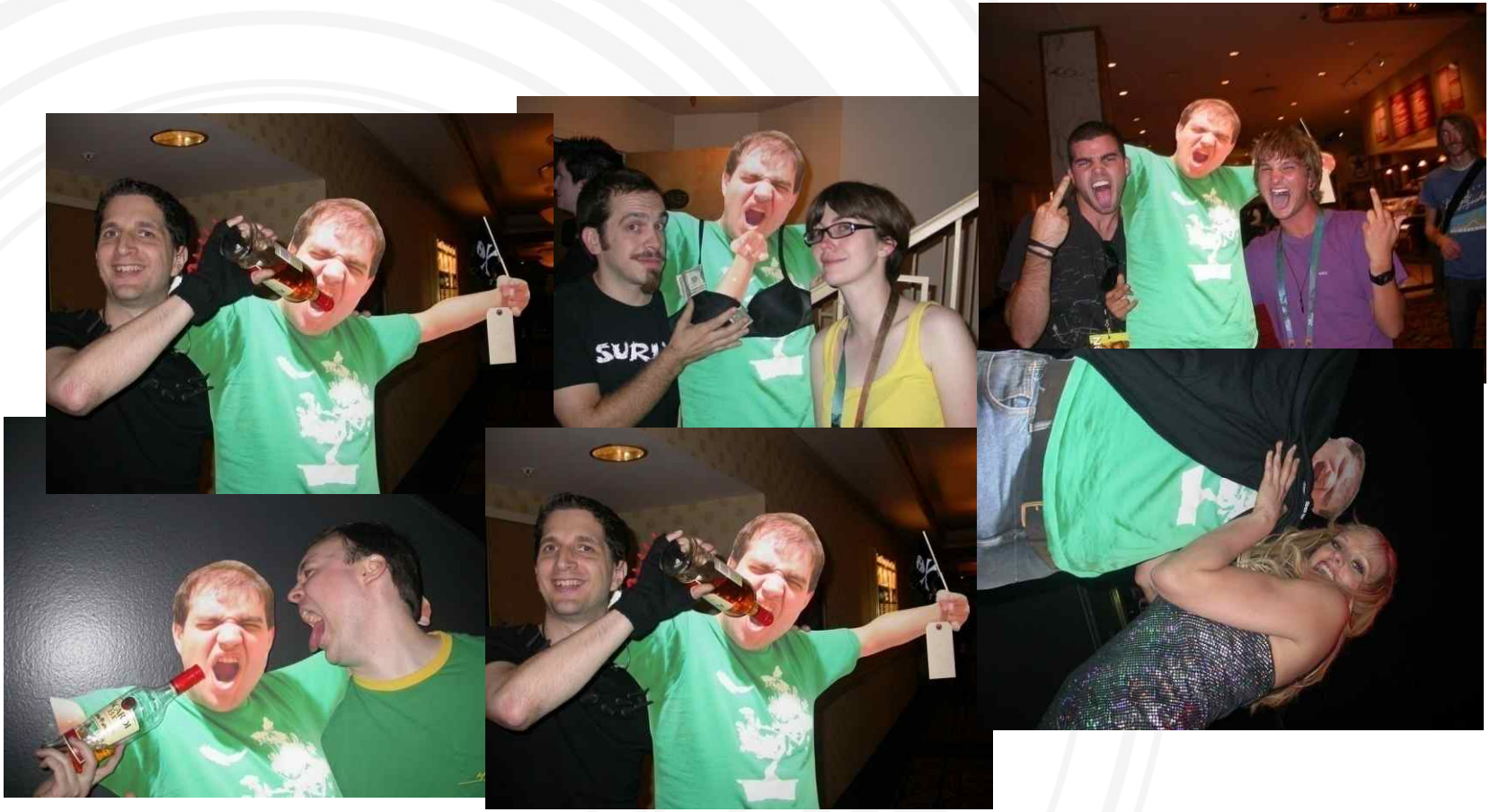
- 압축 파일이 주어지고, 해당 압축을 풀어보면 이미지 파일 수십개가 나옴



Binary Leetness 400

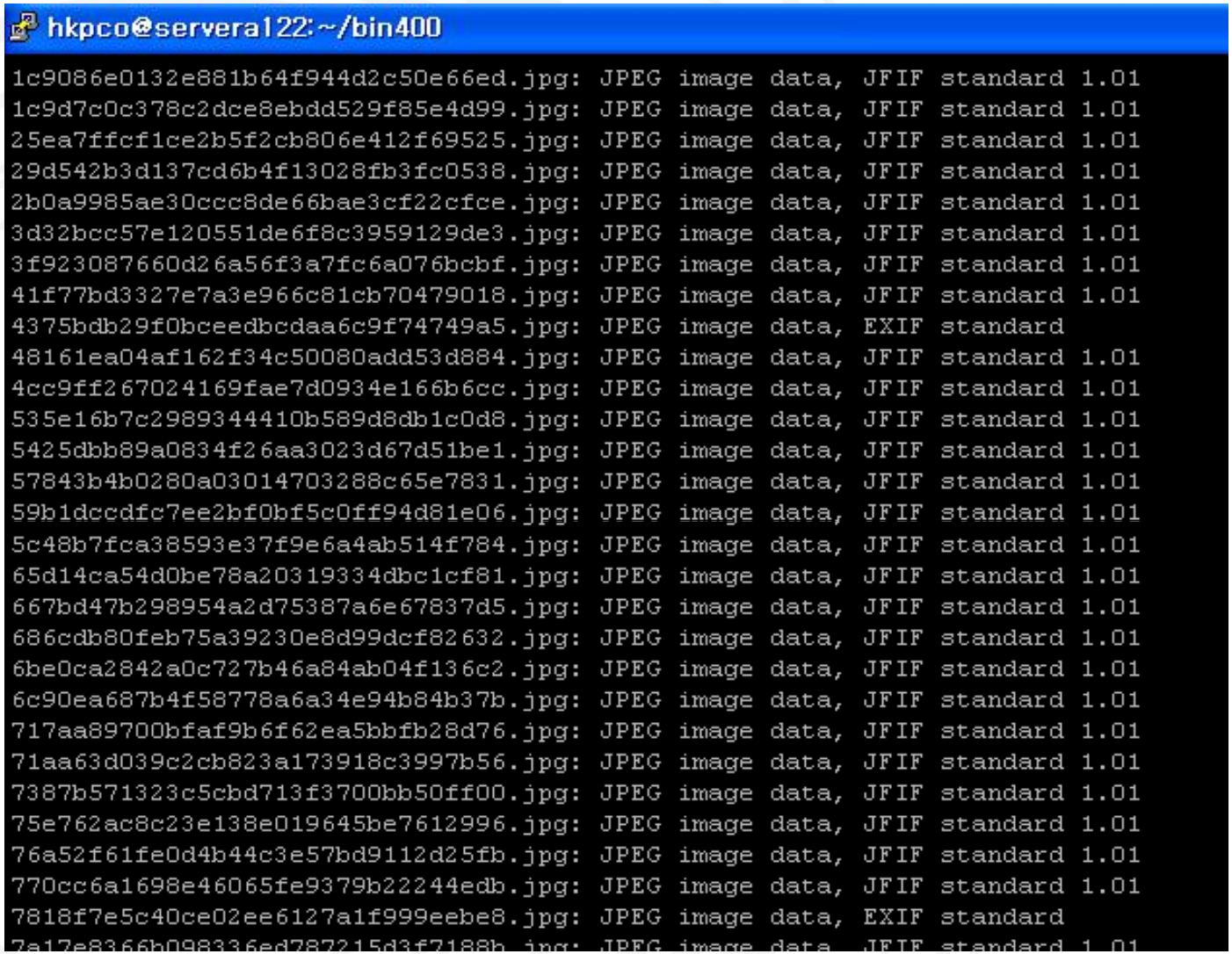
Dan Kaminsky?

Damn it Kaminsky!!



Binary Leetness 400

- 해당 이미지 파일 포멧은 EXIF, JFIF 둘 중 하나



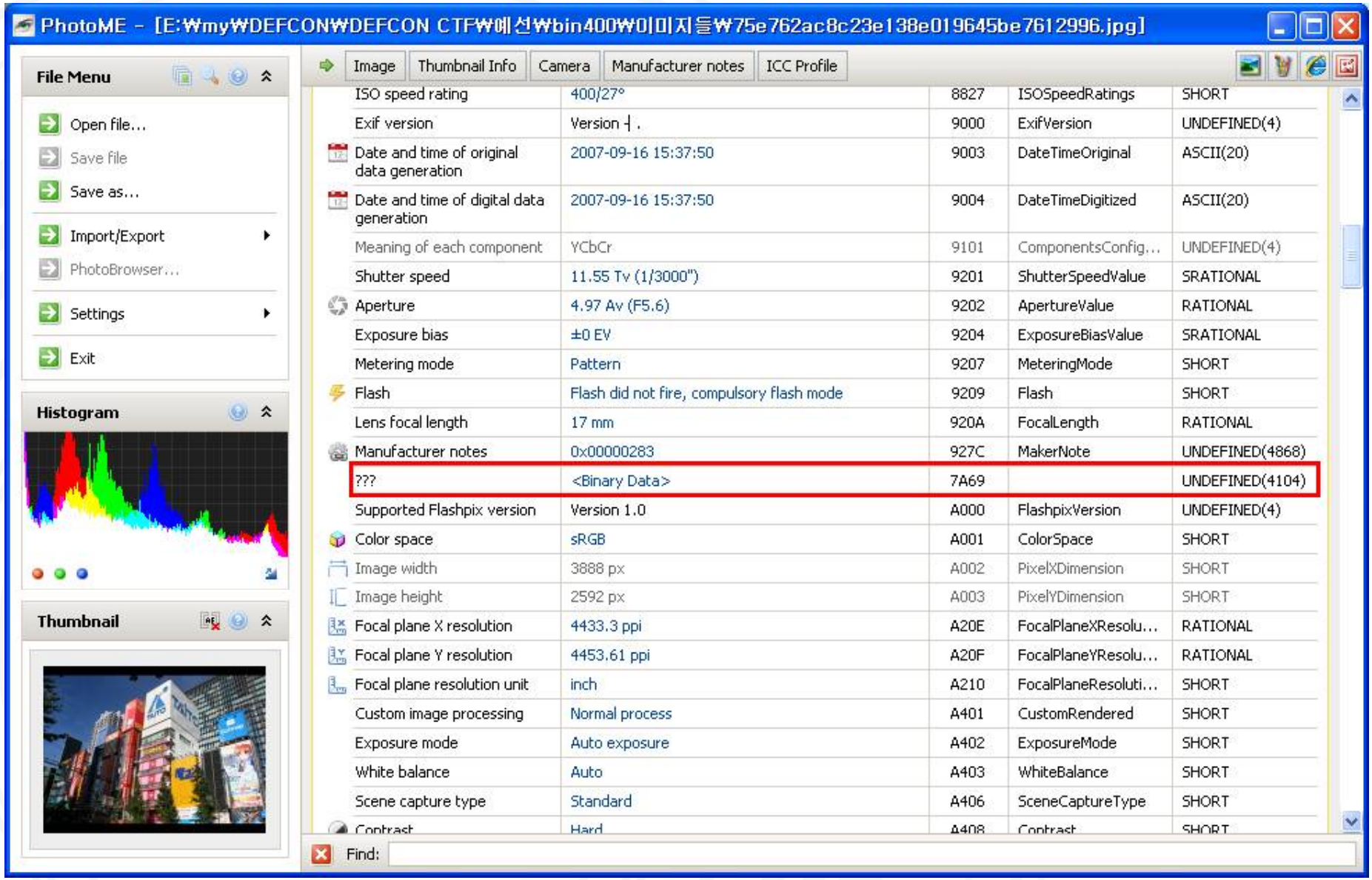
Binary Leetness 400

- Exif(Exchangeable image file format)
  - Exchangeable image file format (Exif) is a specification for the image file format used by digital cameras.
- JFIF(JPEG File Interchange Format)
  - The JPEG File Interchange Format (JFIF) is an image file format standard.

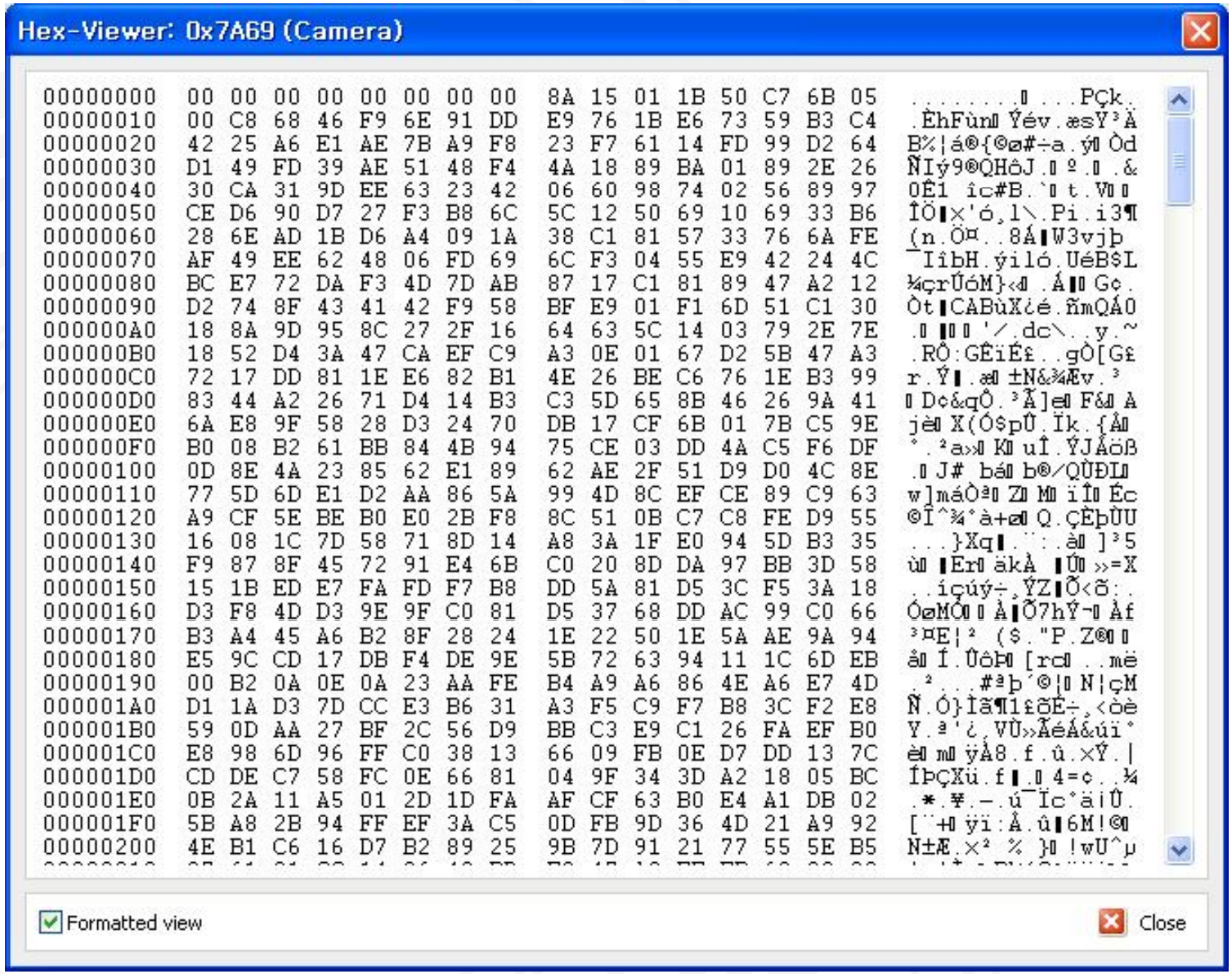
- 이미지 파일 내에 특정 데이터가 숨겨져 있을것으로 예상
- 하지만, 헤더 구조를 잘 알지않는 이상 이를 구별해 내기가 힘듦

- 관련 파일 정보 등을 분석해 주는 툴을 이용하면 편리
  - Ex) PhotoME

Binary Leetness 400



Binary Leetness 400



Binary Leetness 400

- Python의 Python Imaging Library를 이용하여 해당 데이터 추출
- 이미지 파일 안에 숨겨진 데이터는 Tag-ID 7A69에 존재하며
- PIL의 Exif 관련 함수로 추출 가능
- 참고로, Exif 표준은 JFIF의 거의 모든 기능을 지원하므로 JIFF를 고려해 줄 필요는 없음

Binary Leetness 400

- 추출된 데이터들은 압축(ZIP) 파일의 조각들이며
- 이를 특정 규칙으로 merge 해야함

- 이는 jpg 파일 전체 내용의 SHA-1 값 정렬 순으로 합치면 됨
- 다양한 hash를 기준으로 정렬해 보면 그 시작 부분이 ZIP 파일의 헤더로 나오는것을 보고 알 수 있음

Binary Leetness 400

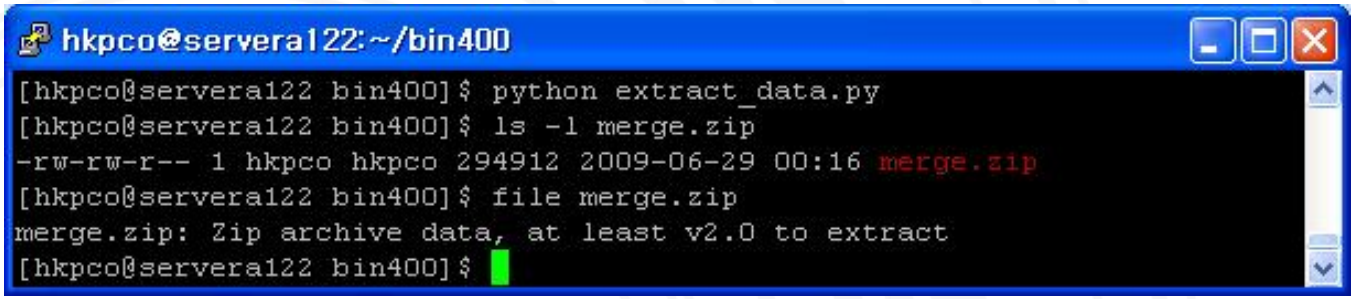
이미지 파일에서 데이터를 추출하여 SHA-1 순으로 합치는 스크립트

extract_data.py - 메모장

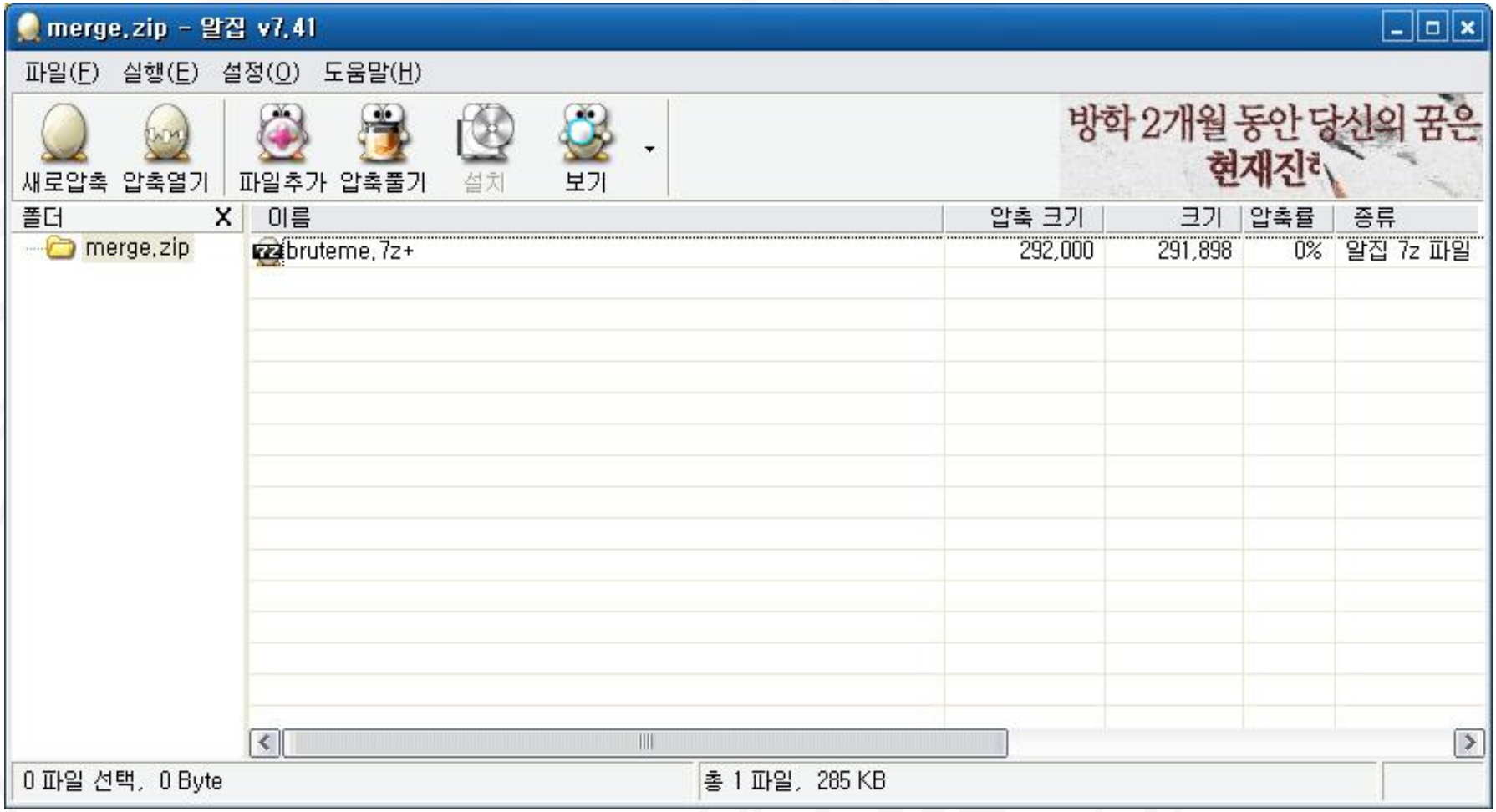
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
import Image
import ExifTags
import glob
import hashlib

files_jpg = glob.glob('./*.jpg')
for fname in files_jpg:
        im = Image.open(fname)
        exinfo = im._getexif()
        val = exinfo[31337]
        val = val[8:]
        fname = fname.replace( 'jpg', 'slice' )
        fp = open(fname, 'wb')
        fp.write(val)
        fp.close()

files_zip = glob.glob('./*.jpg')
dicts = {}
for fname in files_zip:
        fp = open( fname, 'rb' )
        data = fp.read()
        dicts[fname] = hashlib.sha1(data).hexdigest()
        fp.close()

diclist = []
for i in dicts.keys():
        diclist.append( (i, dicts[i]) )

diclist.sort( key = lambda(x, y):y )

merge_zip = open( 'merge.zip', 'ab' )
for file in diclist:
        ff = file[0].replace( 'jpg', 'slice' )
        f_tmp = open( ff, 'rb' )
        slice_data = f_tmp.read()
        merge_zip.write(slice_data)
        f_tmp.close()

merge_zip.close()
```

Binary Leetness 400



Binary Leetness 400

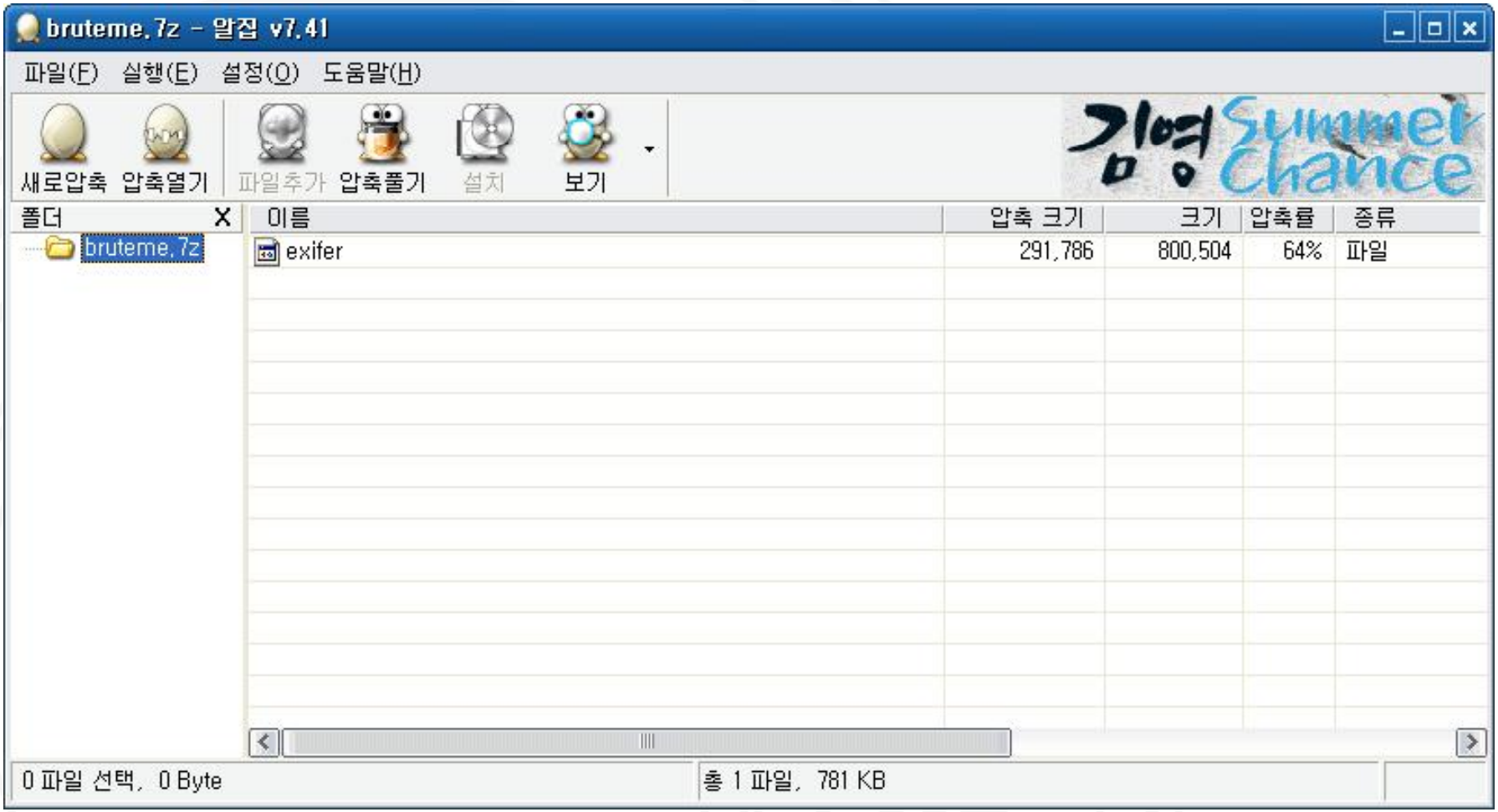


Binary Leetness 400

- 압축 파일에 패스워드가 걸려있음
- 이는 압축 툴(알집 등) 자체의 암호 찾기 기능이나 Advanced Archive Password Recovery 프로그램 등을 사용하여 풀 수 있음

- 압축 파일의 패스워드는 weak

Binary Leetness 400



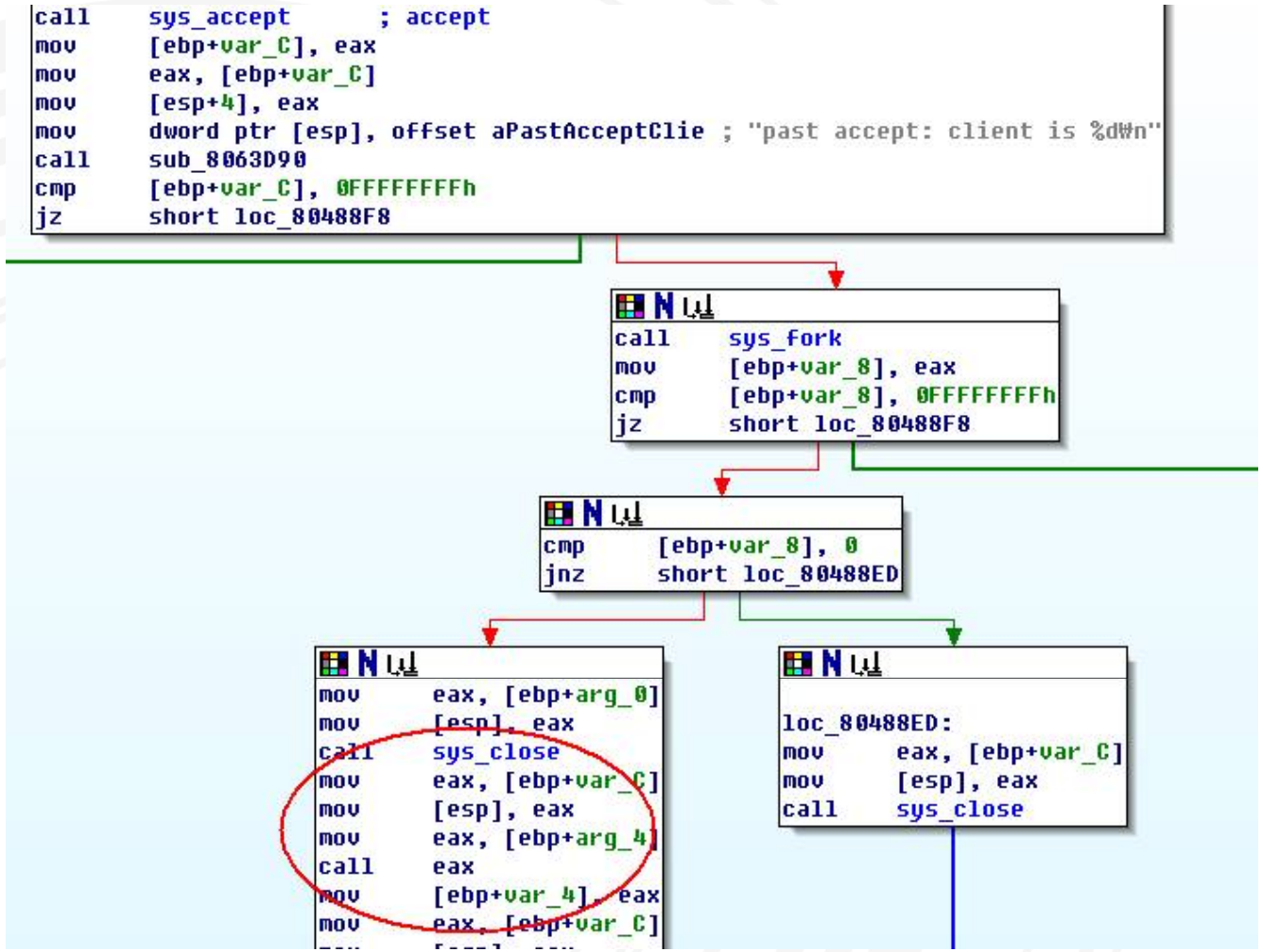
Binary Leetness 400

- 압축을 풀면 바이너리가 나오며, 간략한 정보는 다음과 같음

[hkpco@servera122 bin400]$ file exifer

exifer: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, for GNU/Linux 2.6.9, stripped

Binary Leetness 400

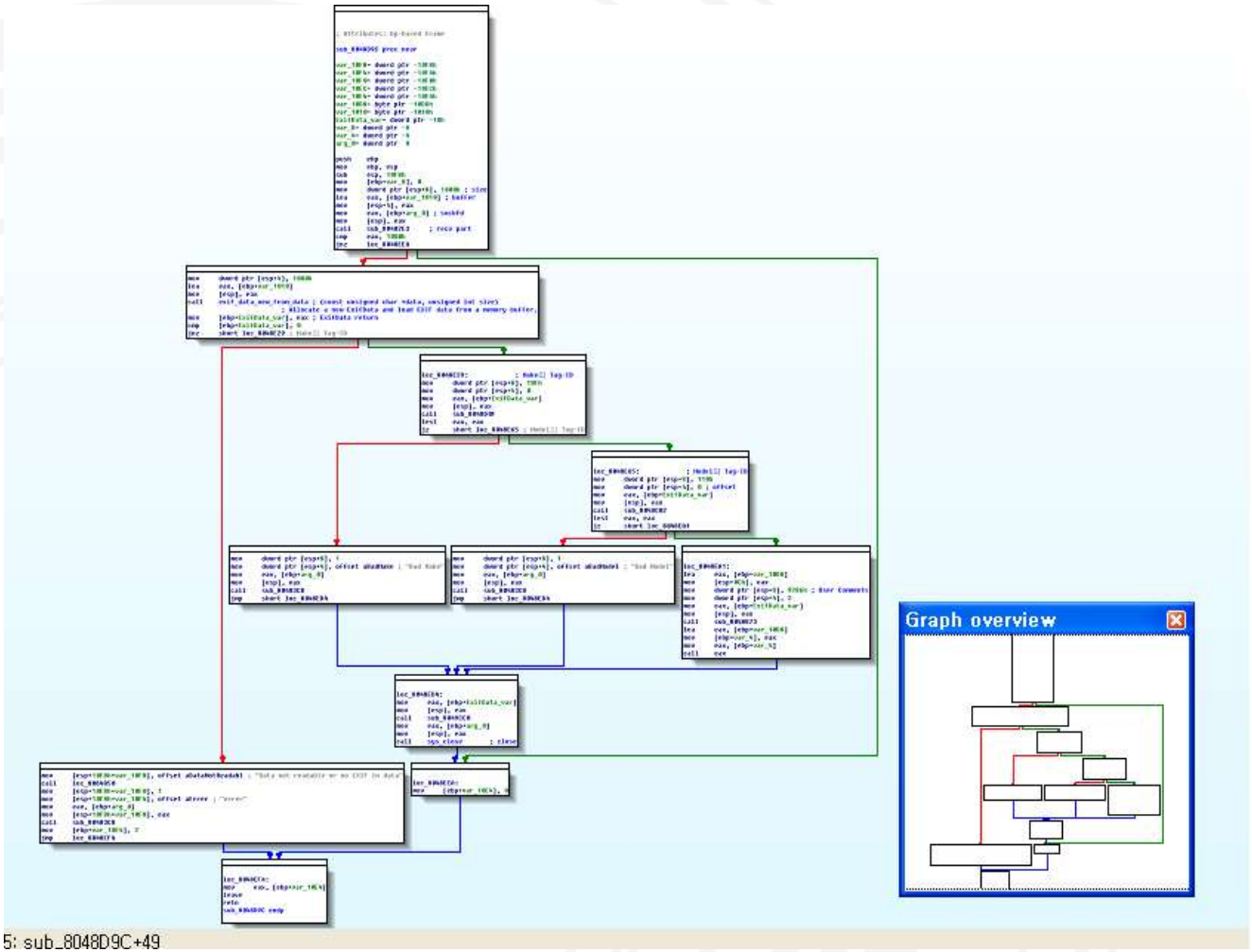


Binary Leetness 400

- 서버 프로그램의 루틴은 보통 accept() 함수로 연결을 수락한 다음, 스레드 혹은 자식 프로세스를 생성하여 주요 루틴을 수행하도록 작성이 됨.
- 대회 문제는 특히 안정성을 위하여 스레드 보다는 자식 프로세스 생성으로 처리를 많이 하는편임
- 그래서 accept() 함수 또는 fork() 함수 등의 호출 구문 이후의 루틴을 살펴보면 서버에서 처리하는 핵심 루틴을 쉽게 찾을 수 있음

- 여기서는 두 번째 인자의 주소값을 호출하는것을 볼 수 있음

Binary Leetness 400

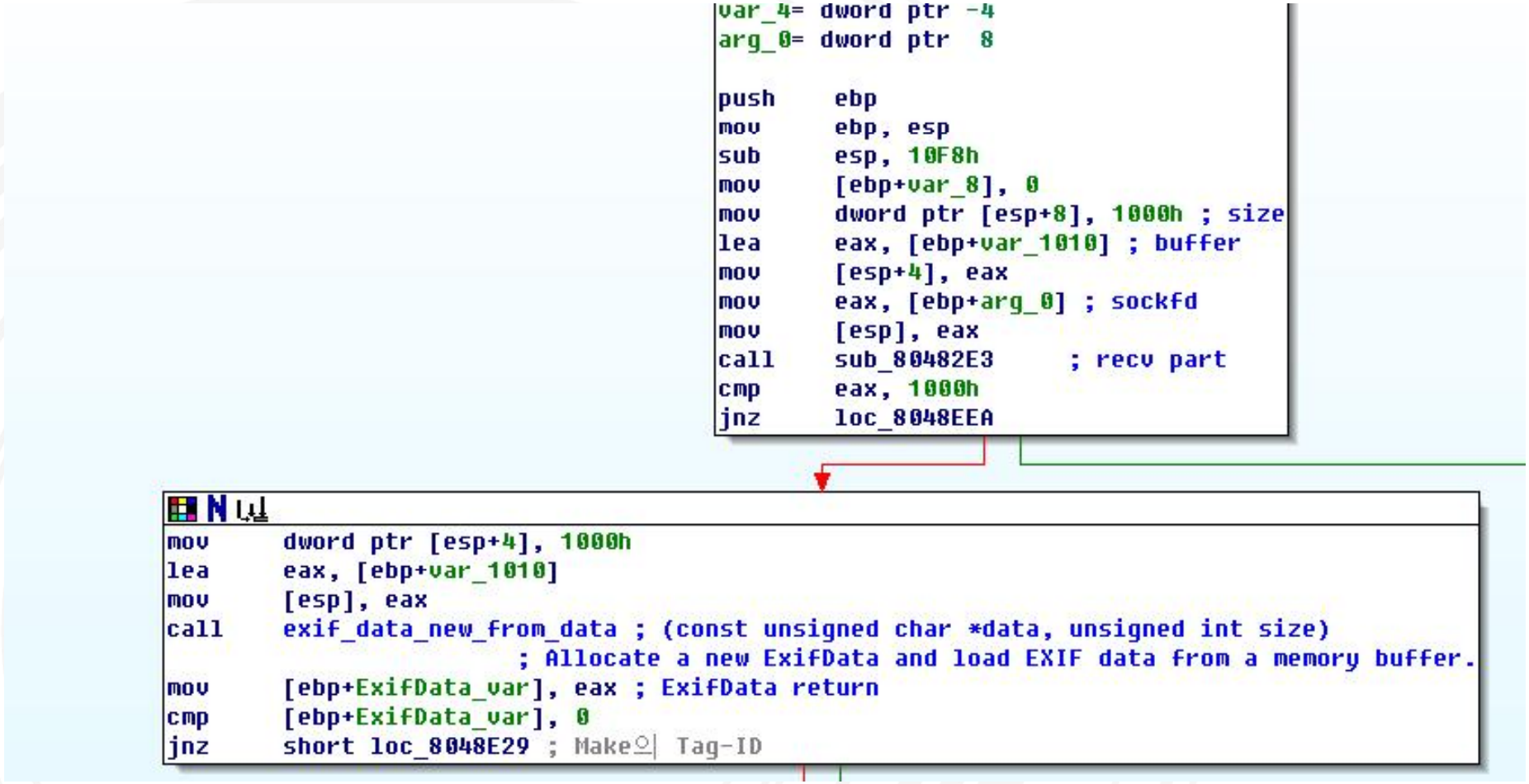


Binary Leetness 400

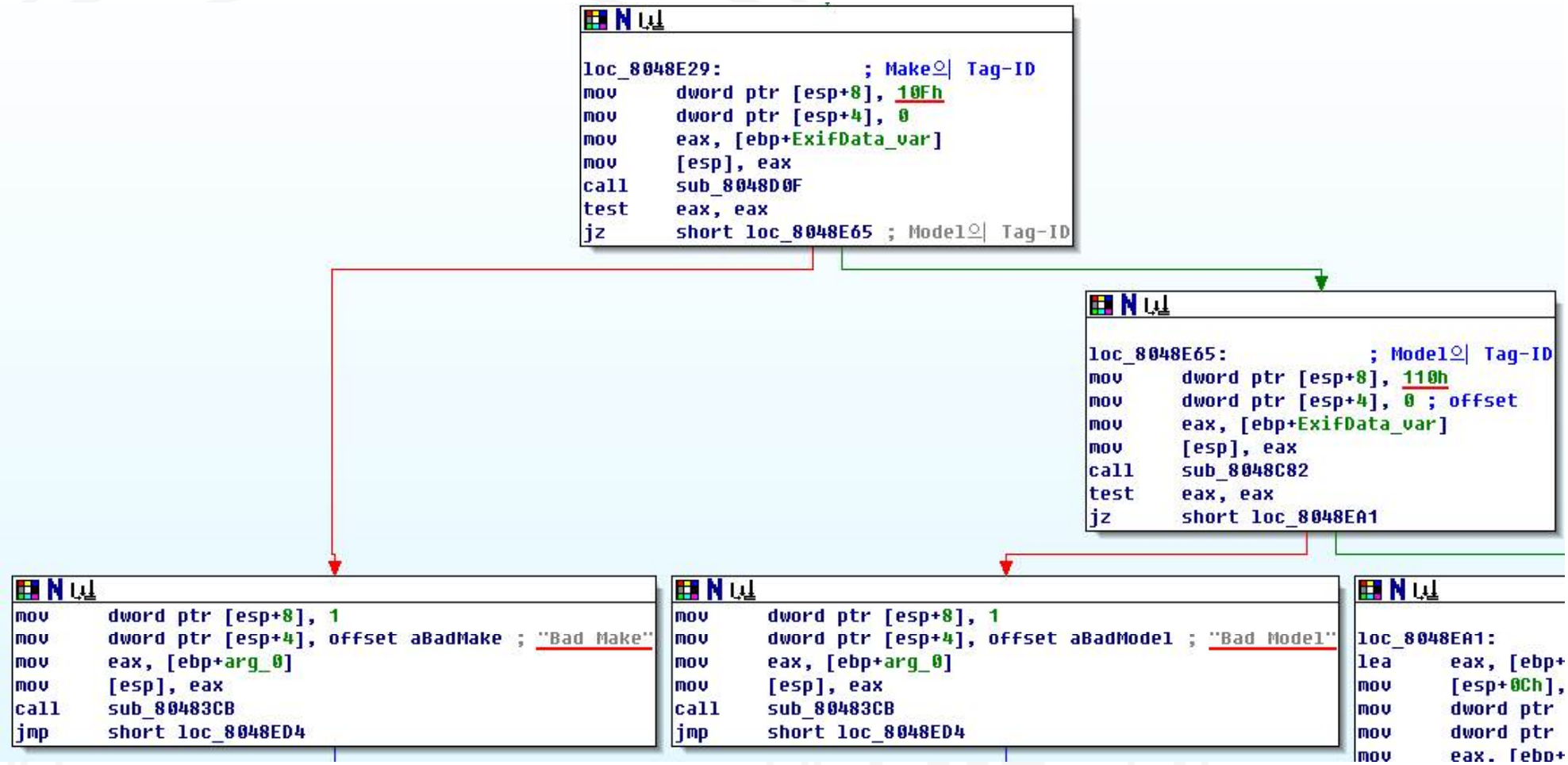
- 파일 이름 등 조금만 살펴보면 해당 바이너리는 **Exif** 관련 라이브러리를 사용한다는 것을 알 수 있음

- 문제 바이너리는 **statically**에 디버깅 정보까지 삭제되어 있기 때문에 기본적으로 함수명을 알 수 없으며, 대신 인자값의 개수나 함수 내의 고유 스트링 등을 **libexif(Exif** 라이브러리)에 존재하는 함수 원형이나 소스코드 등의 내용과 비교하여 쉽게 추측할 수 있음(그 이외에도 방법은 많음)

Binary Leetness 400



Binary Leetness 400



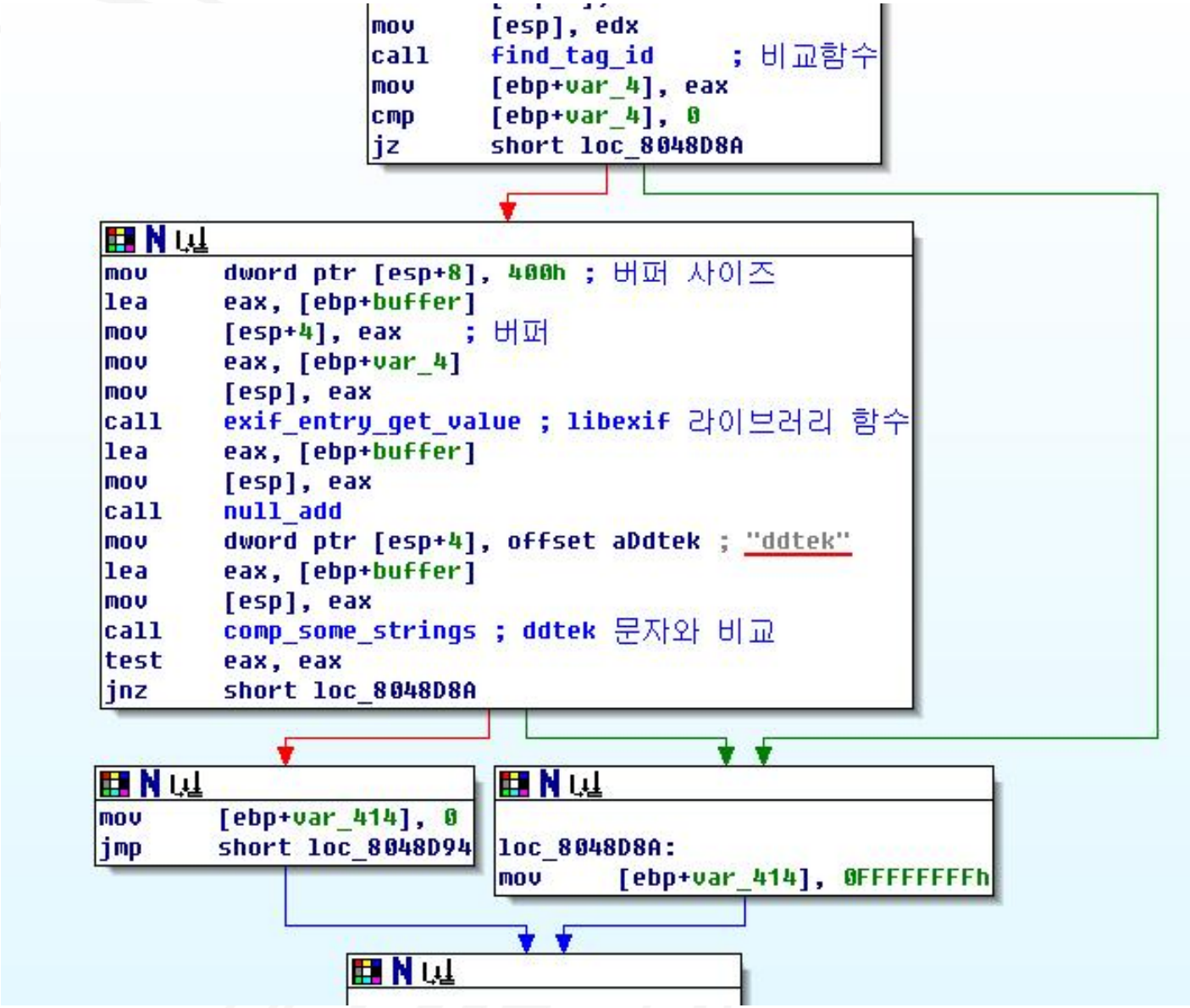
- 함수 호출 이후의 에러 메시지나 인자로 주어지는 값만 보아도 어디를 비교하는지 쉽게 알 수 있음

Binary Leetness 400

Make(Tag-ID = 010Fh)

필드의 값이 ddtek인지

비교하는 루틴

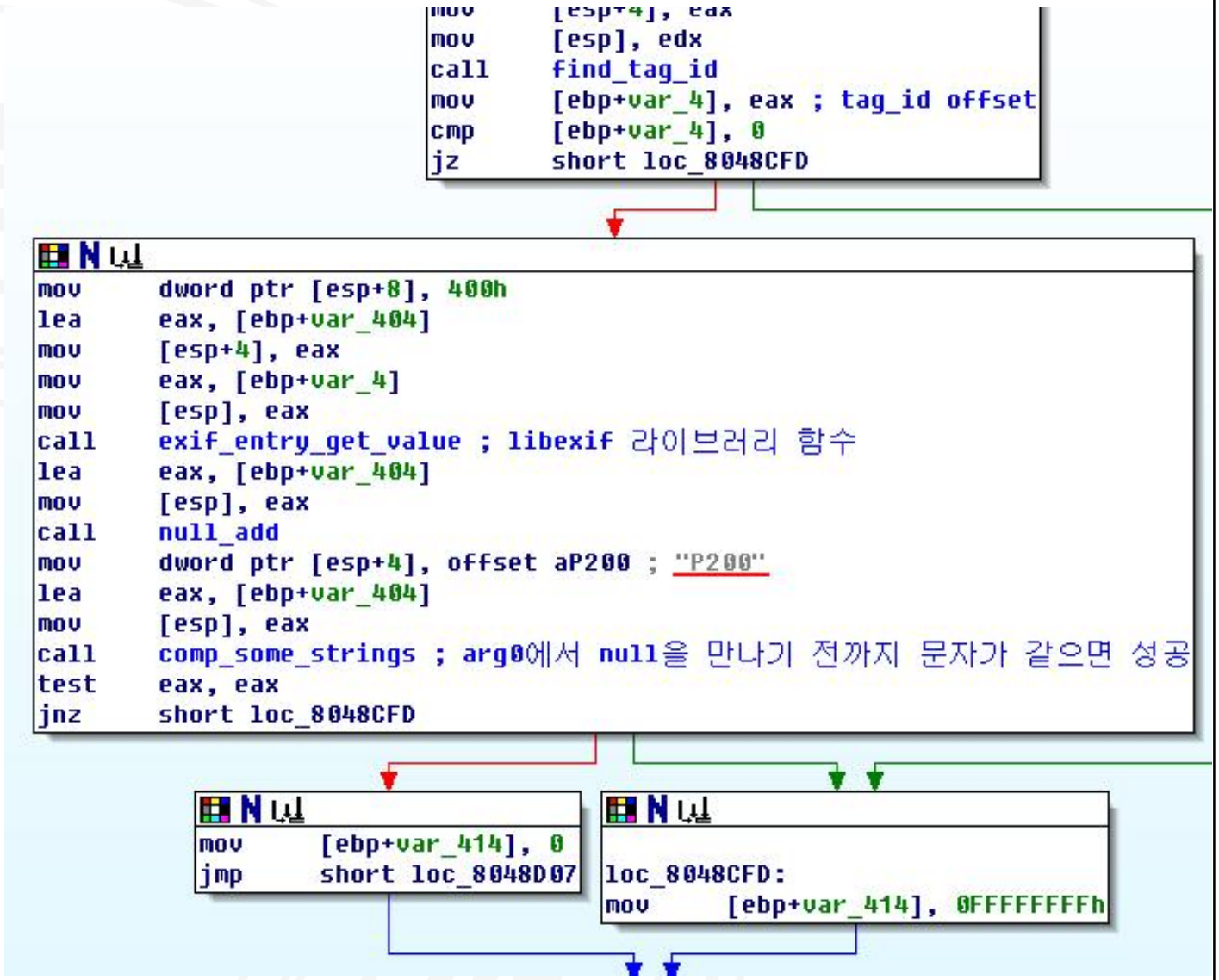


Binary Leetness 400

Model(Tag-ID = 0110h)

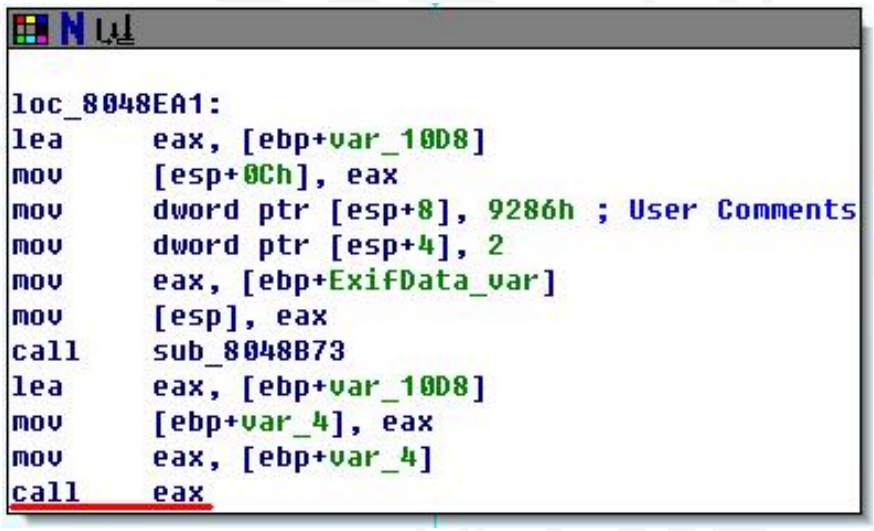
필드의 값이 P200 인지

비교하는 루틴



Binary Leetness 400

- User Comments의 값을 가져와서 실행하는 루틴



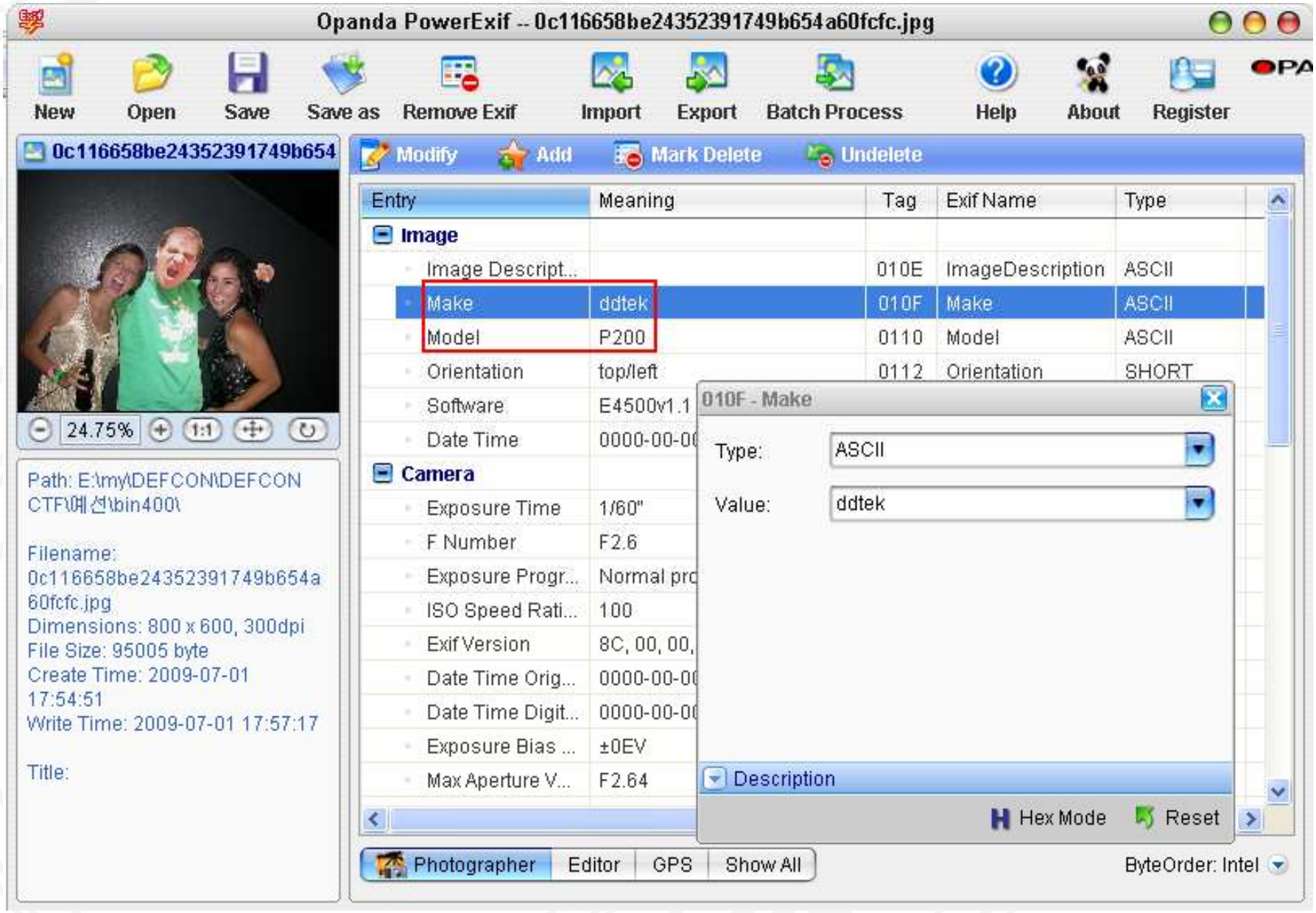
```
loc_8048EA1:
lea     eax, [ebp+var_10D8]
mov     [esp+0Ch], eax
mov     dword ptr [esp+8], 9286h ; User Comments
mov     dword ptr [esp+4], 2
mov     eax, [ebp+ExifData_var]
mov     [esp], eax
call    sub_8048B73
lea     eax, [ebp+var_10D8]
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    eax
```

Binary Leetness 400

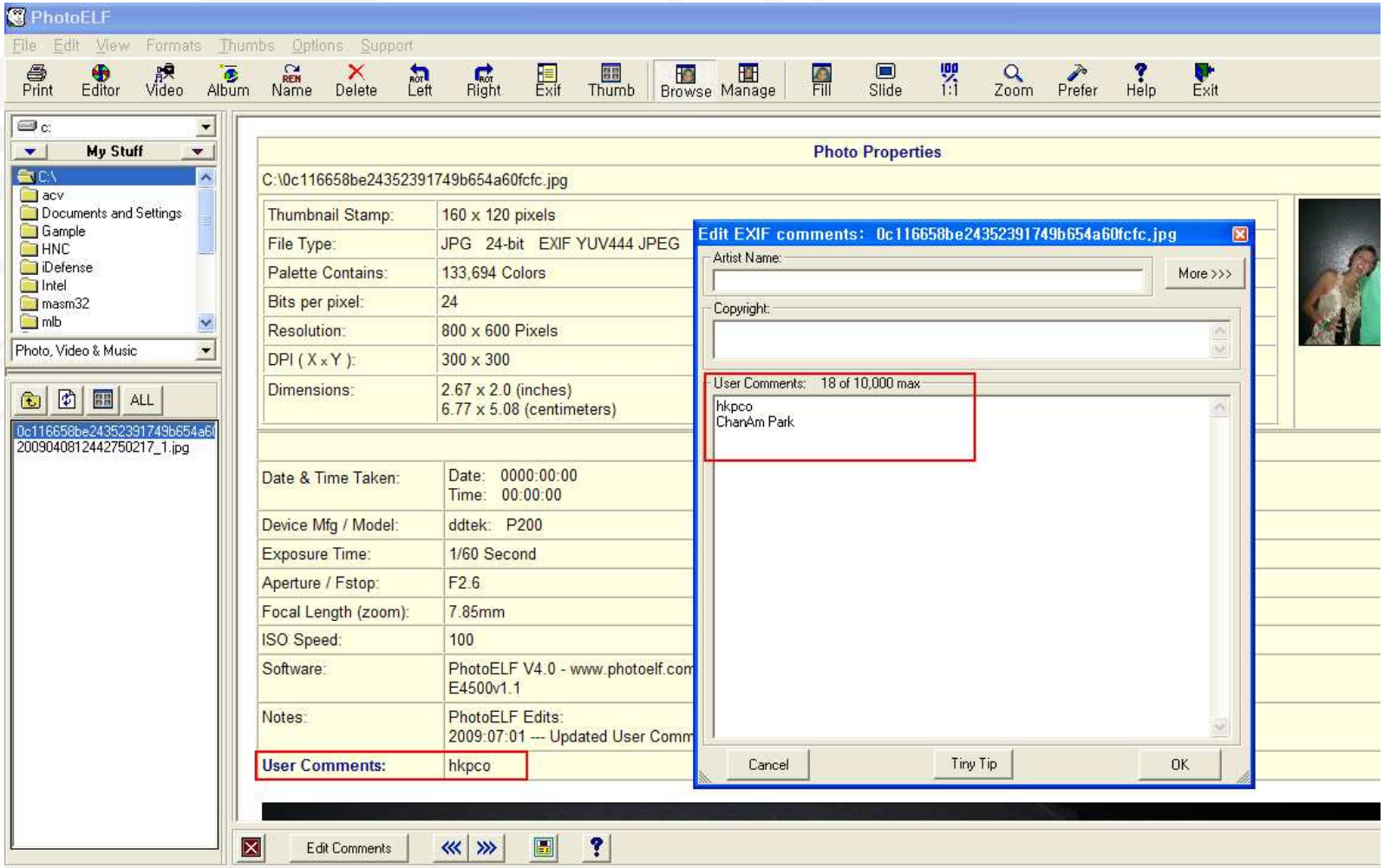
- Exif의 Make, Model 필드와 User Comments를 각각 *ddtek, P200, 실행을 원하는 코드*로 조작해야 하며, 그러한 방법은 많고 특히, 각종 툴이 잘 만들어져 있기 때문에 쉽게 바꿀 수 있음

- Make, Model은 **Opanda PowerExif**
- User Comments는 **PhotoELF**

Binary Leetness 400



Binary Leetness 400



Binary Leetness 400

최종적으로 다음과 같이 이미지 파일을 조작하여 문제를 풀 수 있음

- Make(010Fh): ddtek
- Model(0110h): P200
- User Comments: 실행을 원하는 기계어 코드

Binary Leetness 500

Help, we lost our assembler and manuals.

Assemble and run the given code on the given interpreter

and you will get a prize.

< 문제에서 제공되는 두 가지 >

a. 정체모를 어셈블리 코드

b. 정체모를 바이너리

< 이들의 정체 >

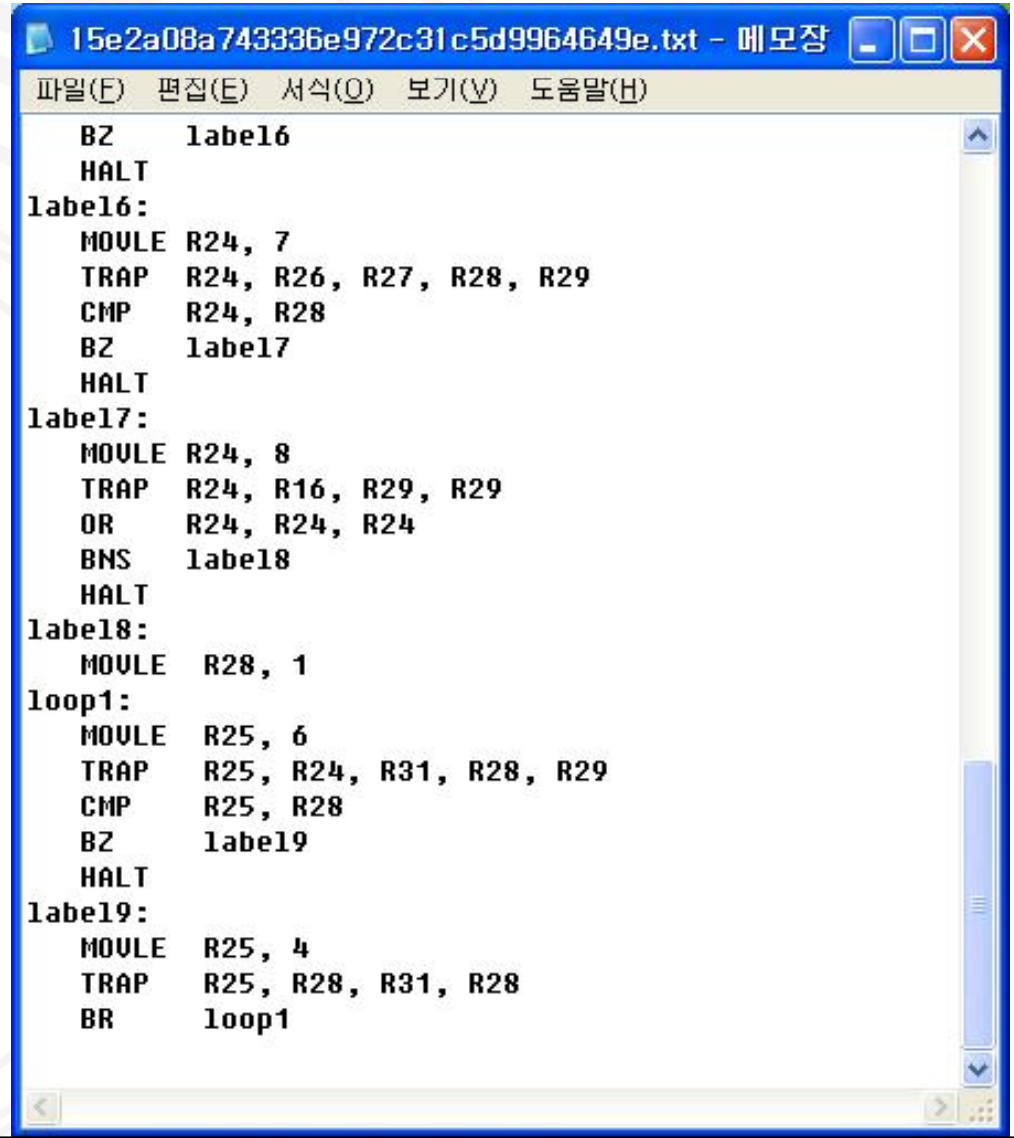
a. 임의로 작성된 가상 어셈블리 코드

b. 가상 어셈블리 코드를 해석하는 인터프리터

Binary Leetness 500

- 문제에서 제공된 asm 코드

```
start:
  EOR   R29,R29,R29
  MOVLE R28, 1
...
TRAP  R26, R27, R28, R29
  OR    R26, R26, R26
  BNS   label1
  HALT
label1:
  MOVLE R16, 5
...
MOVLE  R28, 1
loop1:
  MOVLE  R25, 6
  TRAP   R25, R24, R31, R28, R29
...
label9:
  MOVLE  R25, 4
...
```



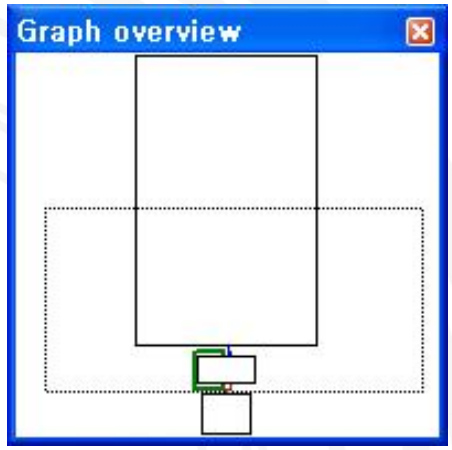
Binary Leetness 500

- 문제에서 제공한 바이너리

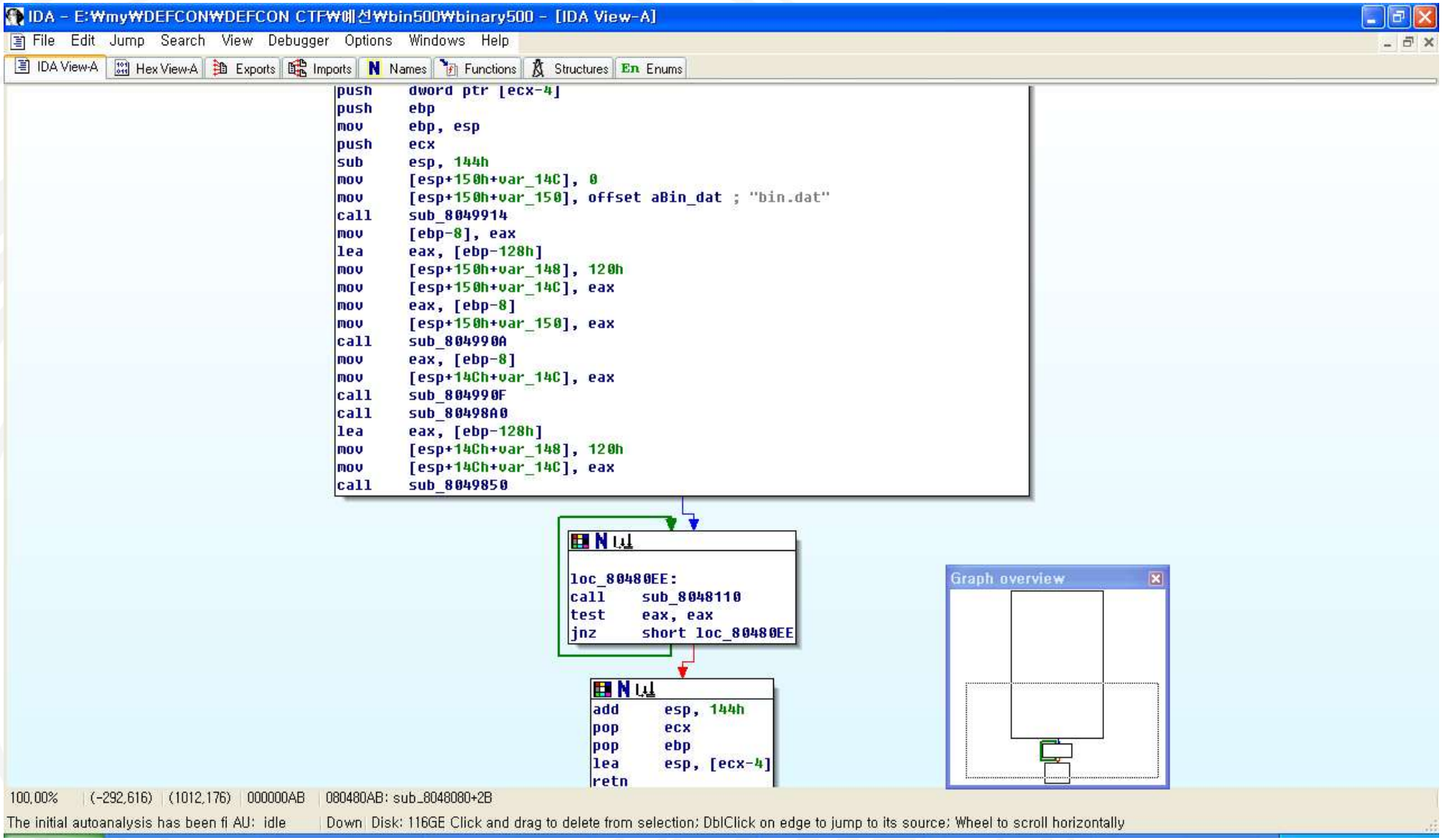
[hkpco@hkpco bin500]$ file bin500

bin500: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), statically linked, stripped

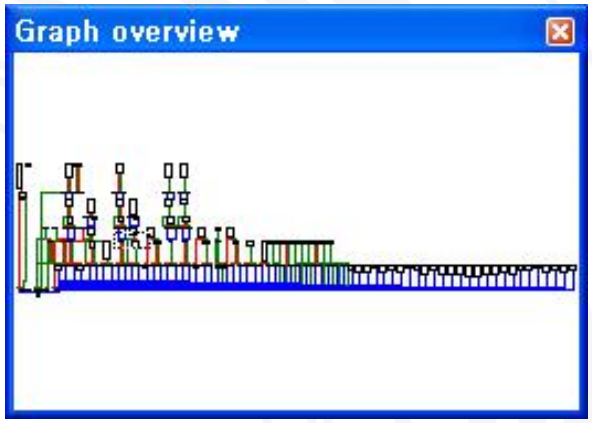
Binary Leetness 500



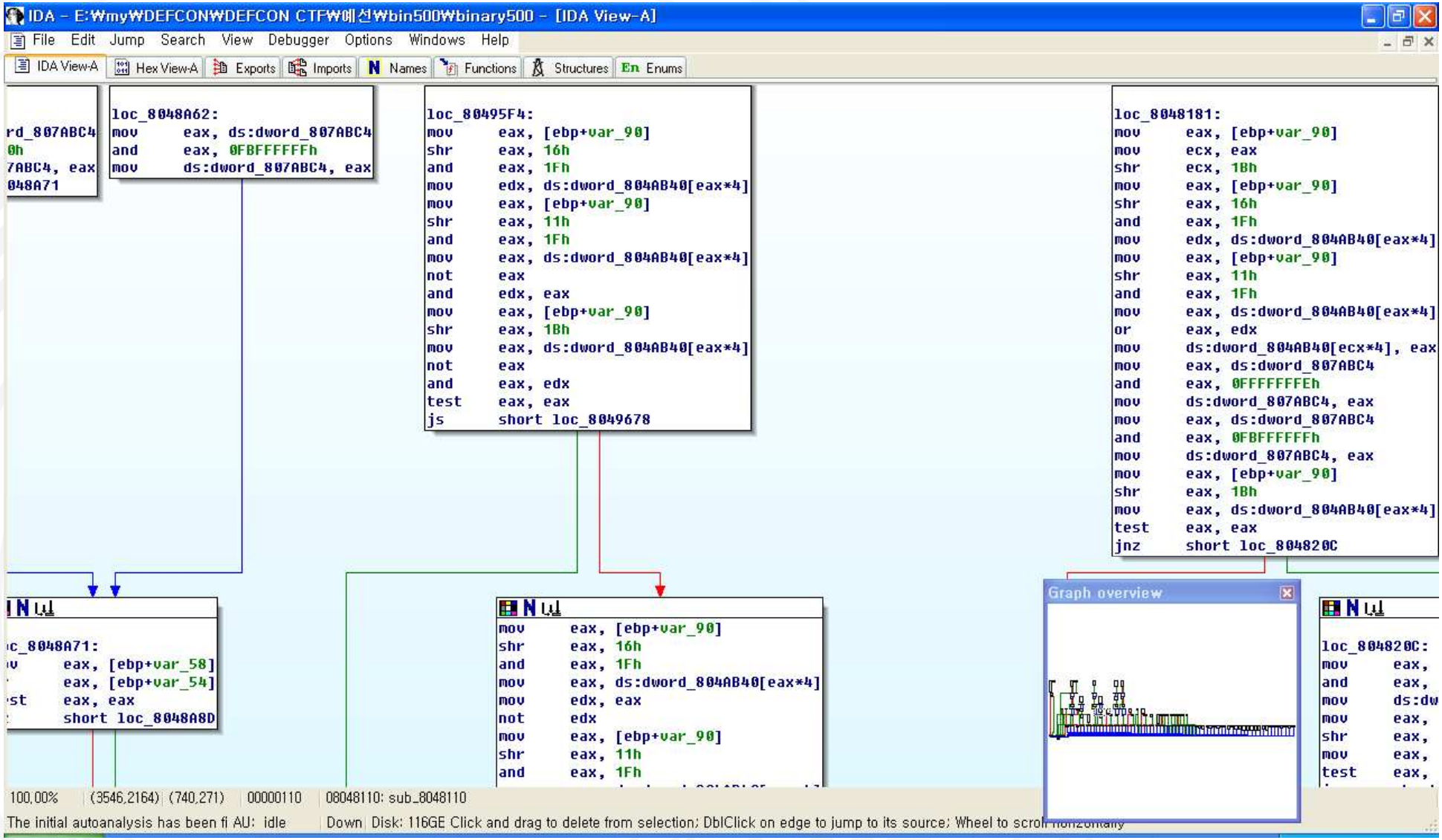
Binary Leetness 500



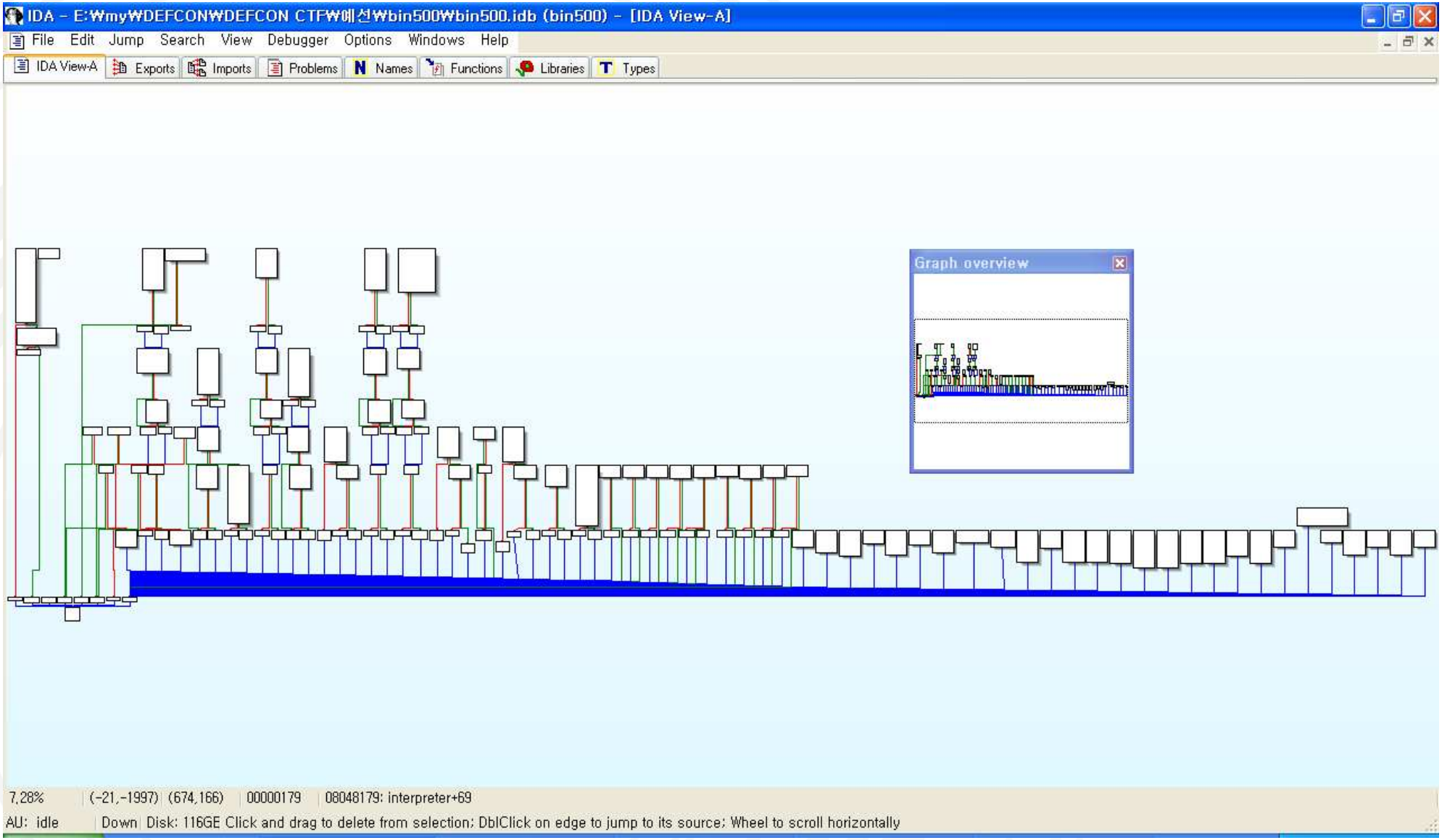
Binary Leetness 500



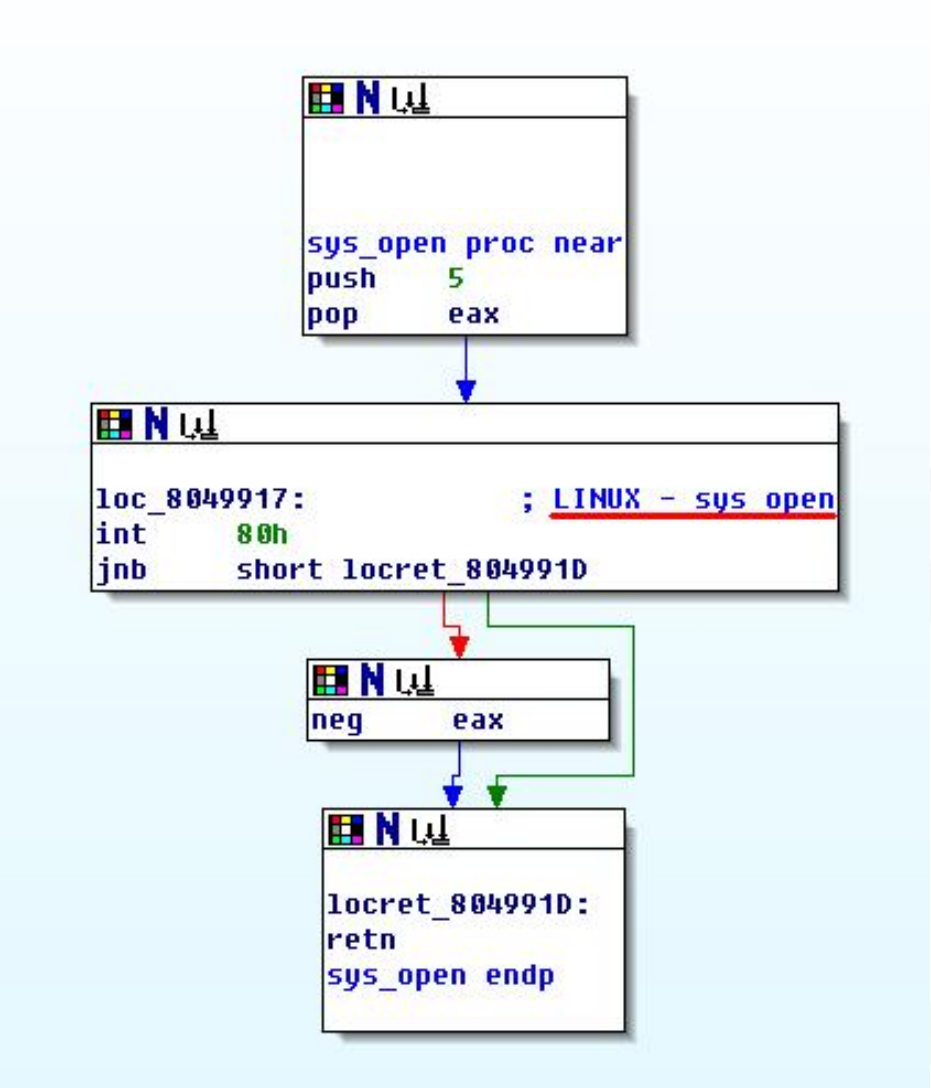
Binary Leetness 500



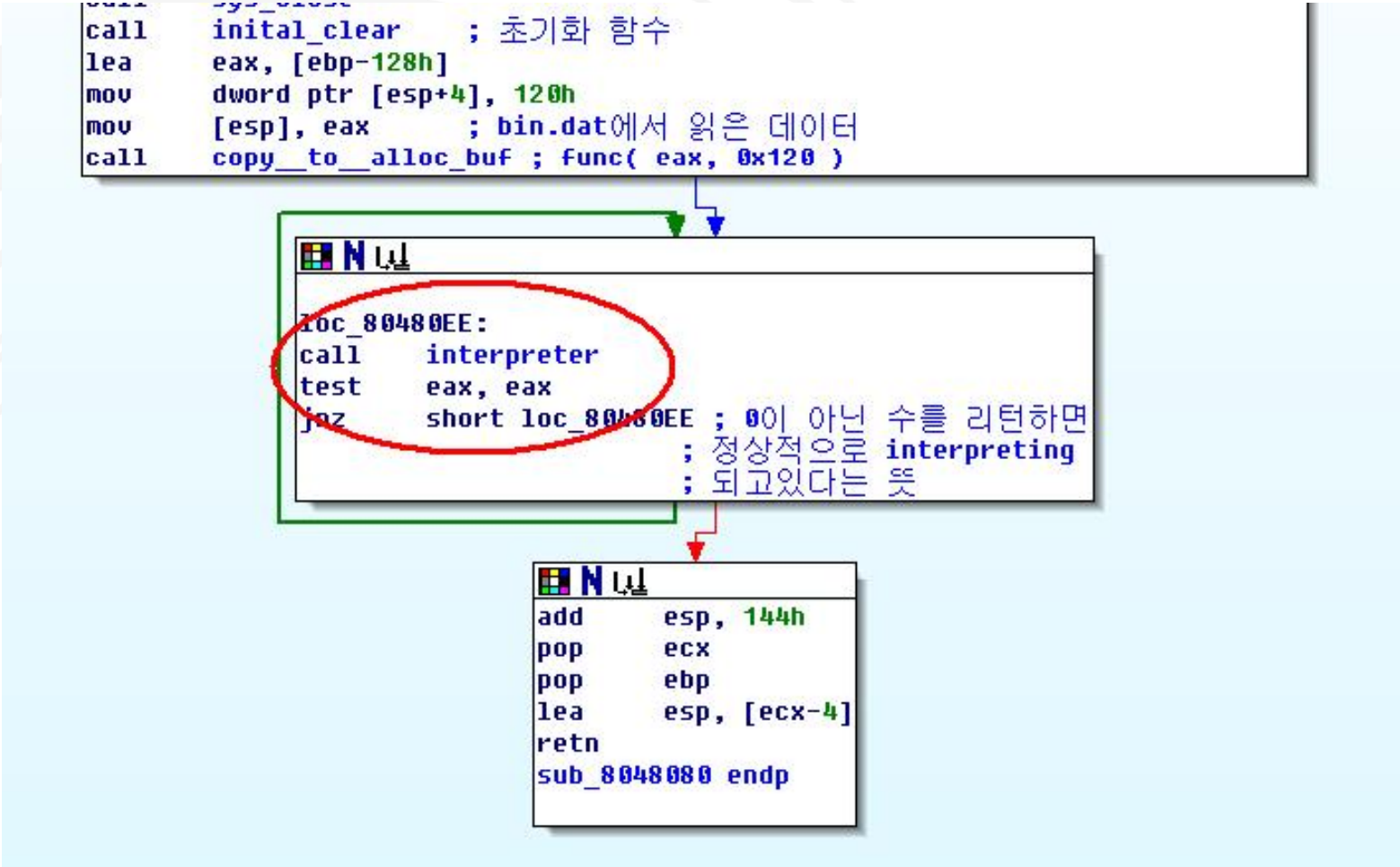
Binary Leetness 500



Binary Leetness 500

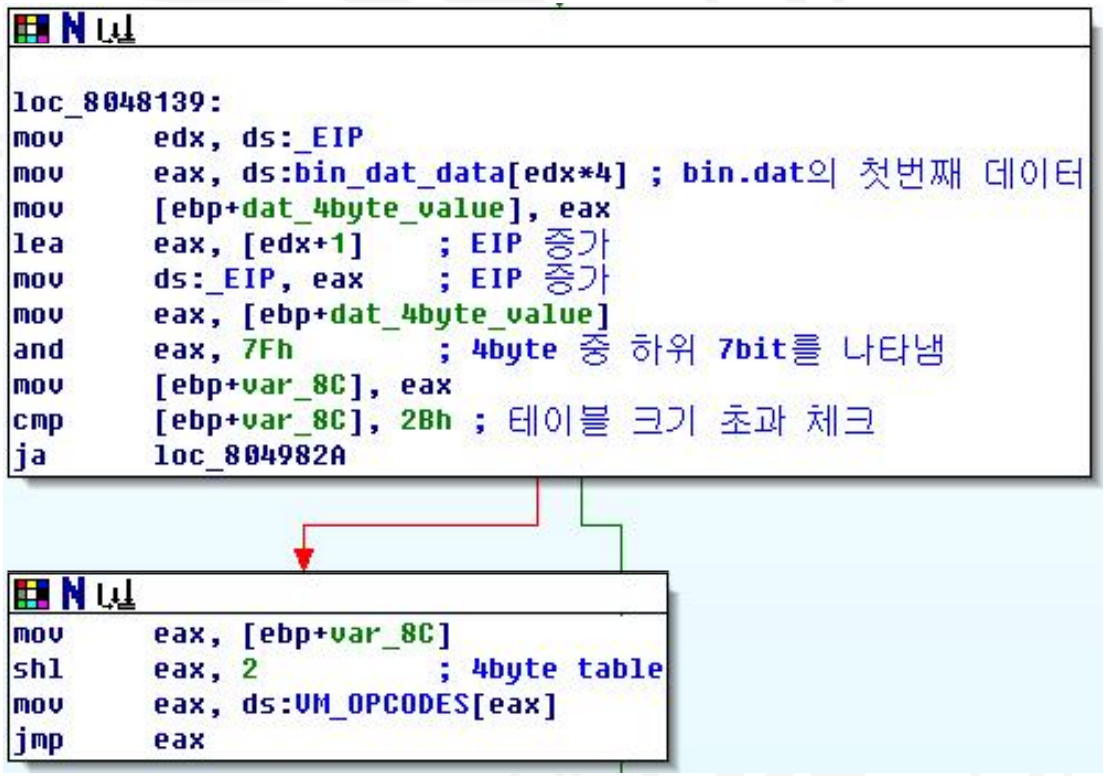


Binary Leetness 500



Binary Leetness 500

- Interpreter 함수의 초기 루틴



Binary Leetness 500

```
.text:08048139          mov     edx, ds:_EIP
.text:0804813F          mov     eax, ds:bin_dat_data[edx*4]
; bin.dat의 edx*4 번째 데이터

.text:08048146          mov     [ebp+dat_4byte_value], eax
; bin.dat에서 _EIP 오프셋에 해당하는 값 4byte를 저장

.text:0804814C          lea     eax, [edx+1]
.text:0804814F          mov     ds:_EIP, eax
; EIP 증가
```

Binary Leetness 500

```
.text:08048154          mov     eax, [ebp+dat_4byte_value]
.text:0804815A          and     eax, 7로
; 4byte 중 하위 7bit를 나타냄

.text:0804815D          mov     [ebp+var_8C], eax
.text:08048163          cmp     [ebp+var_8C], 2Bh
; 테이블 크기 초과 체크

.text:0804816A          ja      loc_804982A
; 초과한다면, 종료
```

Binary Leetness 500

```
.text:08048170                mov     eax, [ebp+var_8C]
; 하위 7바이트

.text:08048176                shl     eax, 2
; table의 각 크기가 4byte이기 때문에, shift 연산으로 4를 곱함

.text:08048179                mov     eax, ds:VM_OPCODES[eax]
.text:0804817F                jmp     eax
; 해당 오프셋의 가상 코드 테이블로 점프
```

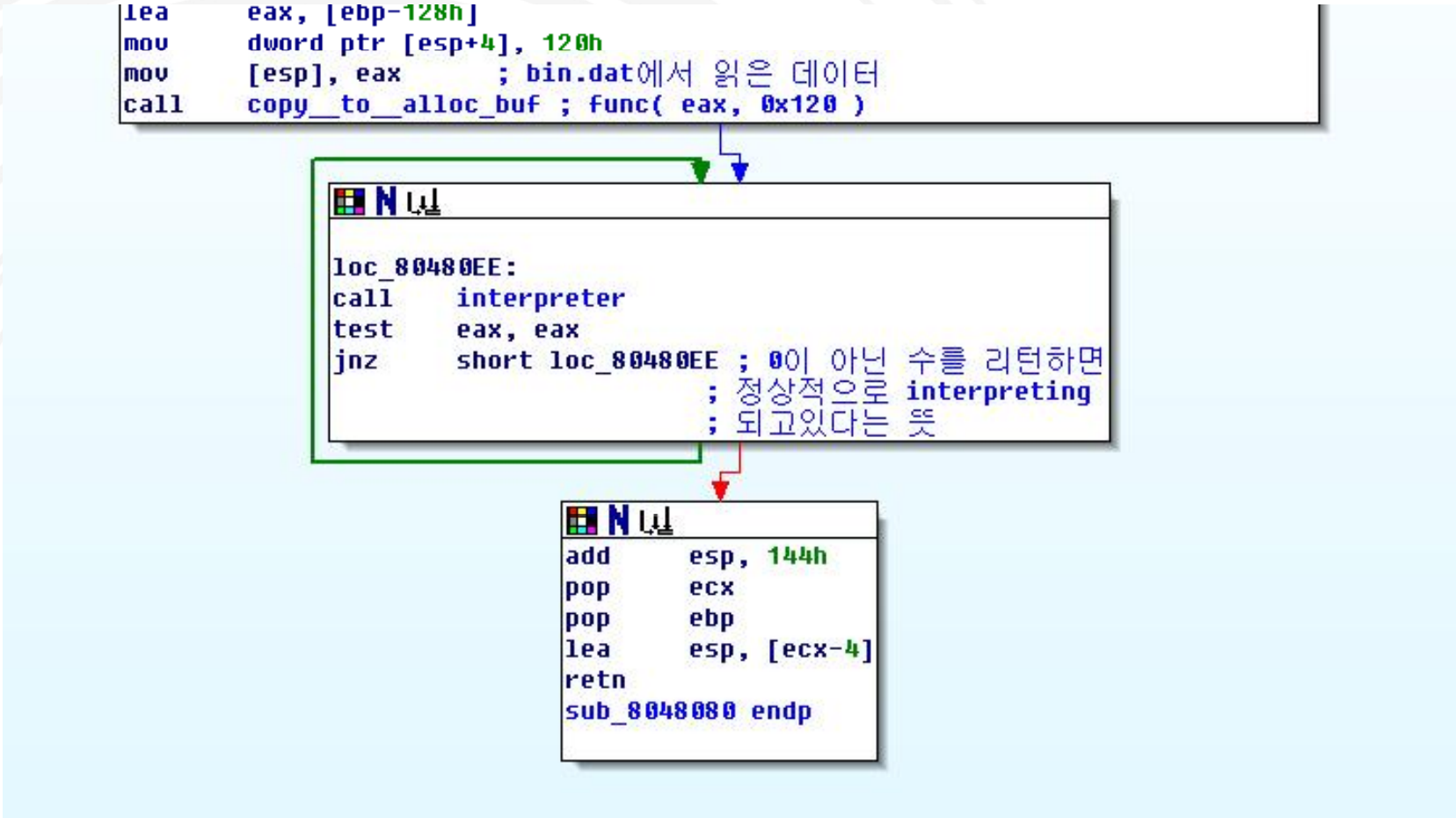
Binary Leetness 500

- 가상 명령 테이블의 일부

```
.rodata:080499C8 VM_OPCODES      dd offset __OR          ; DATA XREF: inter
.rodata:080499CC                 dd offset loc_8048255
.rodata:080499D0                 dd offset loc_8048264
.rodata:080499D4                 dd offset loc_80483BE
.rodata:080499D8                 dd offset loc_80483EB
.rodata:080499DC                 dd offset loc_8048427
.rodata:080499E0                 dd offset loc_8048481
.rodata:080499E4                 dd offset loc_80484B6
.rodata:080499E8                 dd offset loc_8048515
.rodata:080499EC                 dd offset __BR
.rodata:080499F0                 dd offset loc_8048580
.rodata:080499F4                 dd offset __MOVLE       ; 단순 mov
.rodata:080499F8                 dd offset loc_80485EB
.rodata:080499FC                 dd offset loc_804861E
.rodata:08049A00                 dd offset loc_8048775
.rodata:08049A04                 dd offset loc_80487B5
.rodata:08049A08                 dd offset __EOR
.rodata:08049A0C                 dd offset loc_8048946
.rodata:08049A10                 dd offset loc_8048982
.rodata:08049A14                 dd offset __INC
.rodata:08049A18                 dd offset loc_8048AC0
.rodata:08049A1C                 dd offset loc_8048AF3
.rodata:08049A20                 dd offset __MOVU
.rodata:08049A24                 dd offset __TRAP
.rodata:08049A28                 dd offset loc_8048FA7
```

Binary Leetness 500

- Interpreter함수가 성공하면 1을, 그렇지 않으면 0을 반환



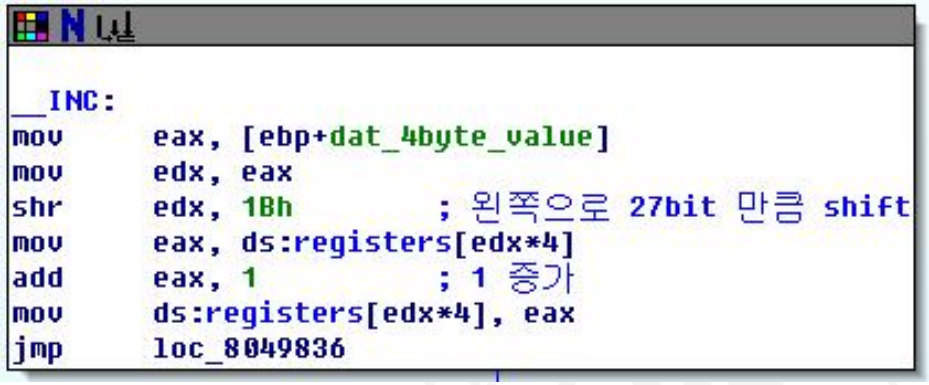
Binary Leetness 500

- VM_OPCODES 테이블에 있는 수십개의 루틴들을 분석 후,
- 주어진 어셈블리 명령에 해당하는 루틴을 찾아야 함

- 다음 단계는 가상 코드 몇 가지를 분석해본 후...

Binary Leetness 500

- 가상 테이블에 있는 increase 명령
  (가장 간단한 루틴)



```
__INC:
mov     eax, [ebp+dat_4byte_value]
mov     edx, eax
shr     edx, 1Bh        ; 왼쪽으로 27bit 만큼 shift
mov     eax, ds:registers[edx*4]
add     eax, 1          ; 1 증가
mov     ds:registers[edx*4], eax
jmp     loc_8049836
```

Binary Leetness 500

```
.text:08048A9F __INC:
.text:08048A9F                  mov     eax, [ebp+dat_4byte_value]
; bin.dat에서 특정 오프셋만큼 떨어진 위치의 4byte 데이터

.text:08048AA5                  mov     edx, eax
.text:08048AA7                  shr     edx, 1Bh
; 오른쪽으로 27bit 만큼 shift
; 이후, 특정 레지스터를 지정하는 오프셋이 됨
```

Binary Leetness 500

```
.text:08048AAA                mov     eax, ds:registers[edx*4]
; edx 오프셋의 레지스터 값을 eax에 복사

.text:08048AB1                add     eax, 1          ; 1 증가

.text:08048AB4                mov     ds:registers[edx*4], eax
; 증가한 값을 레지스터에 이동

.text:08048ABB                jmp     loc_8049836
; 1을 리턴하는 루틴으로 점프(종료)
```

Binary Leetness 500

- 특정 함수를 호출해주는 TRAP 명령

```
.text:08048B29 __TRAP:
.text:08048B29                 mov     eax, [ebp+dat_4byte_value]
.text:08048B2F                 shr     eax, 1Bh
; bin.dat의 특정 오프셋 4byte중 1Bh만큼 shift한 값을 eax에 저장

.text:08048B32                 mov     eax, ds:registers[eax*4]
.text:08048B39                 mov     [ebp+var_9C], eax
; 해당 오프셋의 레지스터 값을 eax에 저장 후 var_9C에 저장
```

Binary Leetness 500

```
.text:08048B3F          cmp     [ebp+var_9C], 0Bh
.text:08048B46          ja      loc_8048F98
; 레지스터 값이 trap table 사이즈인 0Bh(11)보다 크면 종료

.text:08048B4C          mov     edx, [ebp+var_9C]
.text:08048B52          mov     eax, ds:TRAP_TABLE[edx*4]
; 우리가 지정한 오프셋을 TRAP_TABLE에 적용하여 eax에 저장

.text:08048B59          jmp     eax
; 해당 trap 루틴으로 점프
```

Binary Leetness 500

- Trap 루틴이 저장된 테이블

```
.rodata:08049A78 TRAP_TABLE      dd offset wrap_exit
.rodata:08049A7C                 dd offset wrap_open
.rodata:08049A80                 dd offset wrap_close
.rodata:08049A84                 dd offset trap_read
.rodata:08049A88                 dd offset wrap_write
.rodata:08049A8C                 dd offset wrap_socket
.rodata:08049A90                 dd offset wrap_recv
.rodata:08049A94                 dd offset wrap_send
.rodata:08049A98                 dd offset wrap_accept
.rodata:08049A9C                 dd offset wrap_bind
.rodata:08049AA0                 dd offset wrap_listen
.rodata:08049AA4                 dd offset wrap_connect
.rodata:08049AA4 _rodata         ends
```

Binary Leetness 500

```
.text:08048BD5    wrap_close:
.text:08048BD5              mov     eax, [ebp+dat_4byte_value]
.text:08048BDB              mov     ebx, eax
.text:08048BDD              shr     ebx, 1Bh
; 4byte 중 특정 비트를 ebx 레지스터에 저장

.text:08048BE0              mov     eax, [ebp+dat_4byte_value]
.text:08048BE6              shr     eax, 16h
.text:08048BE9              and     eax, 1로
; 4byte 중 특정 비트를 eax 레지스터에 저장
```

Binary Leetness 500

```
.text:08048BEC          mov     eax, ds:registers[eax*4]
.text:08048BF3          mov     [esp], eax
.text:08048BF6          call    sys_close
; 특정 비트의 값을 close 함수의 인자로 전달

.text:08048BFB          mov     ds:registers[ebx*4], eax
; 리턴 값을 특정 레지스터에 저장

.text:08048C02          jmp     loc_8049836
; 종료
```

Binary Leetness 500

- <> = 1Bh
- [] = 16h
- {} = 11h
- () = 0Ch, in the traps
- // = 7h, !! = for traps, ** = for jump
- ~~ = 7Fh, jump to table

<32 31 30 29 28> [27 26 25 24 23] {22 21 20 19 18}

(17 16 /15 14 13) !12 *11 10 9 8*!/ ~7 6 5 4 3 2 1~

Binary Leetness 500

asm.txt - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
label2:
   PUSH  R29            ; 0
   PUSH  R29            ; 0
   PUSH  R29            ; 0                    (sin_addr = R29) -> htonl(INADDR_ANY)) == 0
   MOVL  R28, 2         ; R28 = 2              (sin_family = AF_INET)
   MOVU  R28, 0x5634    ; [R28+2] = 0x5634     (sin_port = 0x5634) -> ntohs = 13398
   PUSH  R28            ; R31 is a esp maybe.

   MOVLE R25, 9         ; R25 = 9
   MOVLE R24, 16        ; R24 = 16
   TRAP  R25, R16, R31, R24
// cli_sock(R25) = bind( sockfd_2, st_addr , 16 );

   OR    R25, R25, R25
   BNS   label3
   HALT

label3:
   MOVLE R25, 10
   TRAP  R25, R16, R24
// R25 = listen( sockfd_2, 16 );

   OR    R25, R25, R25
   BNS   label4
   HALT

label4:
   ADDIB R31, R31, 4

   MOVL  R28, 0xC58C            ; R28 = 0xC58C
   MOVU  R28, 0x586F            ; [R28+2] = 0x586F
   PUSH  R28                    ; for esp
                                ;                       111.88.140.197
                                ; 140.197.111.88 <- this one

   MOVL  R28, 2                 ; R28 = 2               (sin_family = AF_INET)
   MOVU  R28, 0x3456            ; [R28+2] = 0x3456      (sin_port = 0x3456) -> ntohs = 22068
   PUSH  R28                    ; for esp
   MOVLE R25, 11                ; R25 = 11
   MOVLE R24, 16                ; R24 = 16
   TRAP  R25, R26, R31, R24
```

Binary Leetness 500

```
sockfd_1(R26) = socket( 2, 1, 0 );
sockfd_2(R16) = socket( 2, 1, 0 );

/*
        (sin_addr = R29) -> htonl(INADDR_ANY)) == 0
        (sin_family = AF_INET)
        (sin_port = 0x5634) -> ntohs = 13398
*/
cli_sock(R25) = bind( sockfd_2, st_addr , 16 );

R25 = listen( sockfd_2, 16 );

/*
        (sin_family = AF_INET)
        (sin_port = 0x3456) -> ntohs = 22068
        target address - 140.197.111.88
*/
connect( sockfd_1, st_sock, 16 );
send( sockfd_1, 284, 4, 0 );
// 보낼 바이트 수를 넘겨줌

send( sockfd_1, bin.dat의시작, 284, 0 );
cli_sock(R24) = accept( sockfd_2, st_client, st_client );

loop:
        recv( cli_sock, buf, 1, 0 );
        write( 1, buf, 1 );
```

Binary Leetness 500

< 최종 과정 >

- 문제 바이너리를 분석하여 가상 코드를 기반으로 한 bin.dat 작성
- 바이너리에서 bin.dat을 해석

bin.dat의 수행 과정

- 로컬 서버 오픈
- 문제 서버로 접속하여 bin.dat 파일의 크기와 내용 전송
- 대상 서버에서 체크를 거친 후 일치하면 답을 로컬 서버로 전송

+@, funny pwn100

- 문제로 윈도우 바이너리와 서버가 주어짐
- 서버는 해당 바이너리를 실행시킨 것
- 취약성은 간단한 Buffer Overflow
- 하지만 일반적인 Windows Buffer Overflow로는 공격이 안됨
- 이유는…

+@, funny pwn100

- Nmap 등의 툴을 통하여 fingerprinting을 해보면 문제 서버의 OS가 FreeBSD라고 나옴
- 이는 즉, FreeBSD 기반의 서버에서 Wine으로 윈도우 바이너리를 실행시킨 것으로 볼 수 있음
- 결론적으로 공격을 위해서는 FreeBSD 쉘 코드를 써야함
- Wine? *Wine is a free software application that aims to allow Unix-like computer operating systems on the x86 or x86-64 architecture to execute programs written for Microsoft Windows.*

Thank you!!