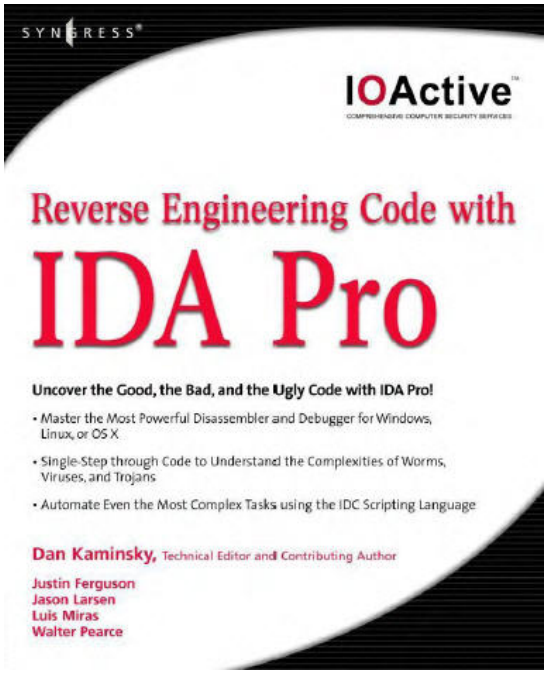
Reverse Engineering Code with IDA Pro



By Dan Kaminsky, Justin Ferguson, Jason Larsen, Luis Miras, Walter Pearce

정리: vangelis(securityproof@gmail.com)

이 글은 ***Reverse Engineering Code with IDA Pro***(2008년 출판)라는 책을 개인적으로 공부하면서 정리한 것입니다. 목적이 책 소개가 아니라 공부이므로 글의 내용은 형식에 국한되지 않고 경우에 따라 책의 내용 일부를 편역하거나 또는 철저하게 요약해서 올리고, 경우에 따라 필자가 내용을 추가할 것입니다. 내용에 오류 및 오타가 있다면 지적 부탁 드립니다. 이 글은 이 글이 필요한 사람들을 위해 정리된 것입니다.

4장 Walkthroughs One and Two

이 장에서는 구체적인 예를 통해 실제 리버싱 과정을 분석한다. 분석에 사용되는 프로그램 StaticPasswordOverflow.exe은 Securityproof 게시판에 이 글과 함께 첨부할 것이다. 참고로 필자가 사용하는 IDA 버전은 Hex-Rays 플러그인이 설치된 5.2.0.908 버전이다.

Following Execution Flow

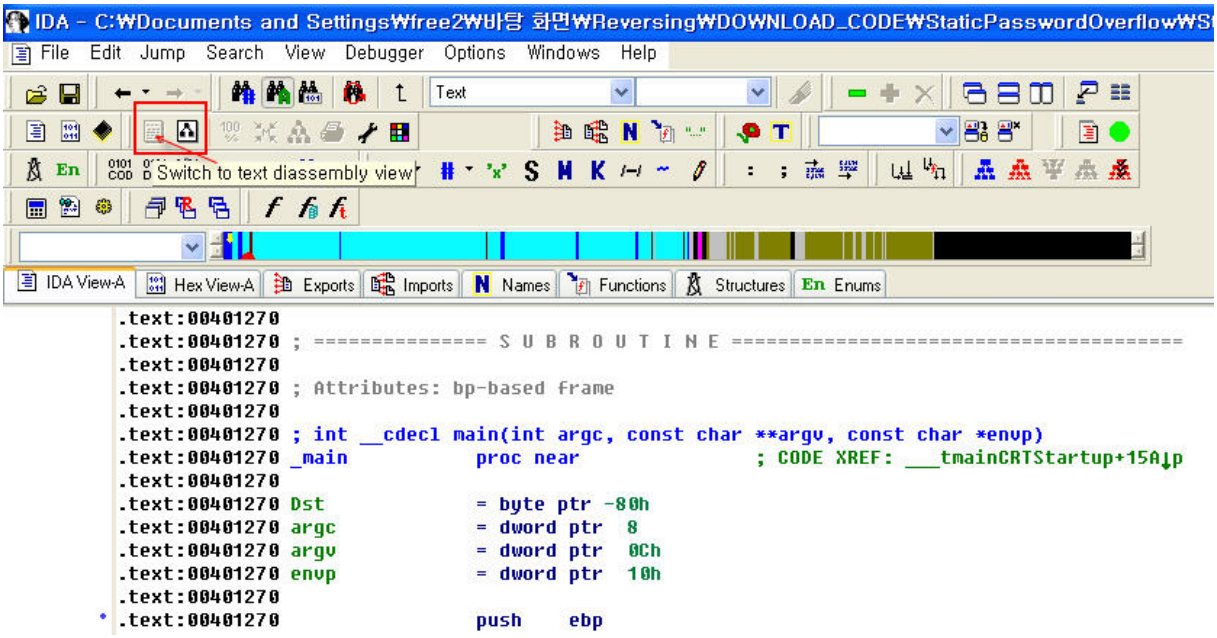
바이너리를 리버싱하는 첫 번째 단계는 바이너리가 어떤 일을 하고, 어떻게 하는지 확인하는 것이다. 먼저 우리가 분석할 `staticpasswordoverflow.exe`라는 파일을 실행해보자.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\free2\바탕 화면\Reversing>staticpasswordoverflow.exe
Reverse Engineering with IDA Pro
Example #1 & #2 - Static Password in Executable
       This example demonstrates a binary file with a
       hardcoded (static) password required to
       continue past a certain point in execution.

[*] Please Provide the password to continue
Password: reversing

******* INVALID PASSWORD *******
You failed. Goodbye.

C:\Documents and Settings\free2\바탕 화면\Reversing>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

결과를 보면 패스워드를 입력해야 다음 단계로 넘어갈 수 있는 것을 볼 수 있다. 그리고 만약 패스워드가 요구하는 것과 맞지 않을 경우 "`You failed. Goodbye.`"란 말과 함께 프로그램이 종료된다. 이제 IDA로 분석을 해보기로 한다.

다음은 StaticPasswordOverflow.exe를 디스어셈블링한 것이다. graph disassembly view로 디스어셈

블링한 것을 나타낼 경우 메모리 상의 주소를 바로 확인하는 것이 힘들다. 그래서 text disassembly view로 보는 것이 편리할 수 있다. 물론 전체 흐름을 한 눈에 파악하는 것에는 graph disassembly view가 더 편할 수 있다.



위의 text disassembly view로 본 내용을 아래와 같이 발췌했다.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
.text:00401270 ; =============== S U B R O U T I N E =======================================
.text:00401270
.text:00401270 ; Attributes: bp-based frame
.text:00401270
.text:00401270 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401270 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401270
.text:00401270 Dst             = byte ptr -80h
.text:00401270 argc            = dword ptr  8
.text:00401270 argv            = dword ptr  0Ch
.text:00401270 envp            = dword ptr  10h
.text:00401270
.text:00401270                 push    ebp
.text:00401271                 mov     ebp, esp
.text:00401273                 sub     esp, 80h
.text:00401279                 push    offset aReverseEnginee ; "Reverse Engineering with IDA Pro₩nExampl"...
```

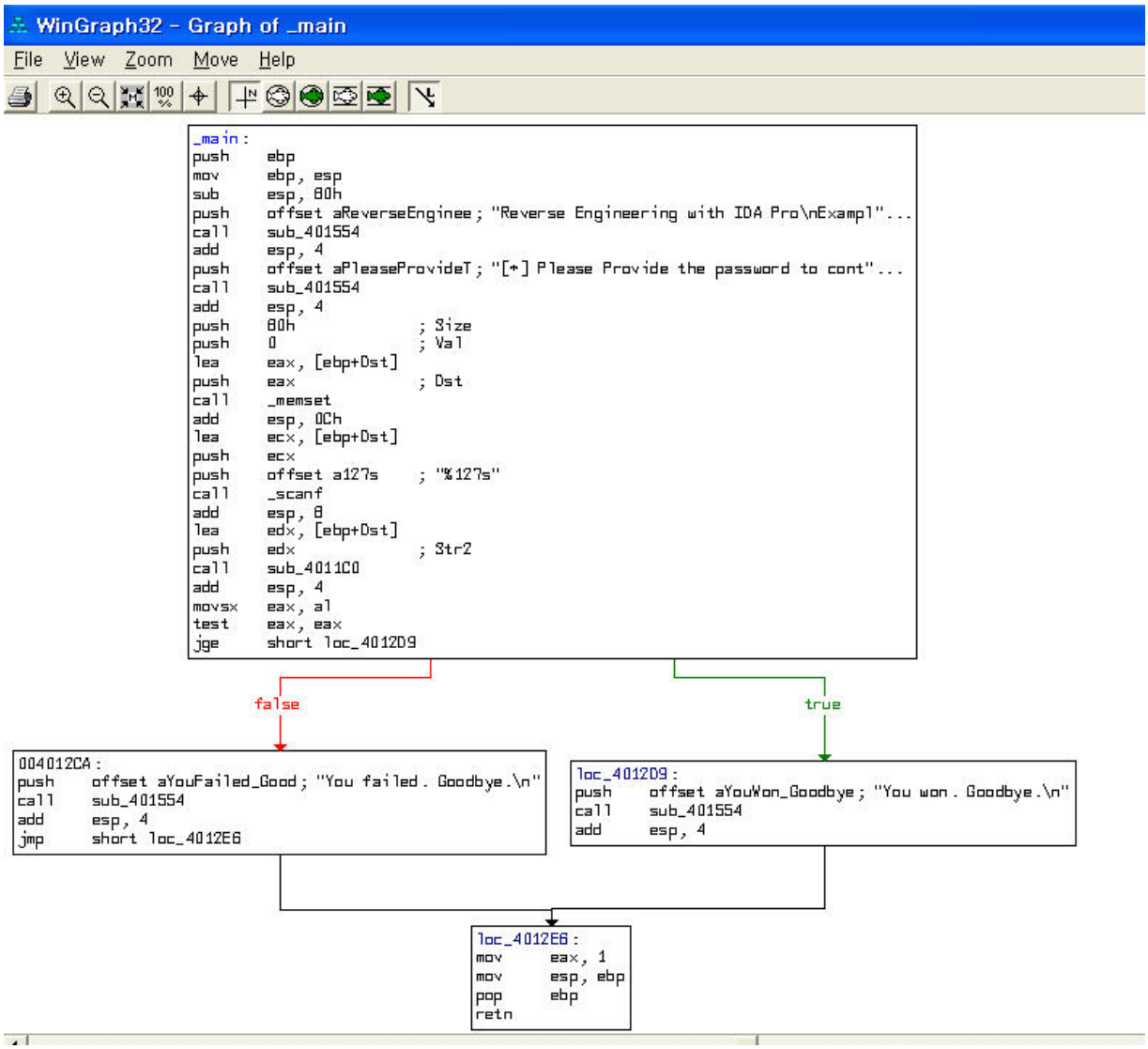
```
.text:0040127E                call    sub_401554
.text:00401283                add     esp, 4
.text:00401286                push    offset aPleaseProvideT ; "[*] Please Provide the password to cont"...
.text:0040128B                call    sub_401554
.text:00401290                add     esp, 4
.text:00401293                push    80h             ; Size
.text:00401298                push    0               ; Val
.text:0040129A                lea     eax, [ebp+Dst]
.text:0040129D                push    eax             ; Dst
.text:0040129E                call    _memset
.text:004012A3                add     esp, 0Ch
.text:004012A6                lea     ecx, [ebp+Dst]
.text:004012A9                push    ecx
.text:004012AA                push    offset a127s    ; "%127s"
.text:004012AF                call    _scanf
.text:004012B4                add     esp, 8
.text:004012B7                lea     edx, [ebp+Dst]
.text:004012BA                push    edx             ; Str2
.text:004012BB                call    sub_4011C0
.text:004012C0                add     esp, 4
.text:004012C3                movsx   eax, al
.text:004012C6                test    eax, eax
.text:004012C8                jge     short loc_4012D9
.text:004012CA                push    offset aYouFailed_Good ; "You failed. Goodbye.₩n"
.text:004012CF                call    sub_401554
.text:004012D4                add     esp, 4
.text:004012D7                jmp     short loc_4012E6
.text:004012D9 ; ---------------------------------------------------------------------------
.text:004012D9
.text:004012D9 loc_4012D9:                            ; CODE XREF: _main+58↑j
.text:004012D9                push    offset aYouWon_Goodbye ; "You won. Goodbye.₩n"
.text:004012DE                call    sub_401554
.text:004012E3                add     esp, 4
.text:004012E6
.text:004012E6 loc_4012E6:                            ; CODE XREF: _main+67↑j
.text:004012E6                mov     eax, 1
.text:004012EB                mov     esp, ebp
.text:004012ED                pop     ebp
.text:004012EE                retn
```

```
.text:004012EE _main            endp
.text:004012EE
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

위의 디스어셈블링한 결과는 main 함수 부분이다. 앞에서 프로그램을 실행한 후 간단히 어떤 일을 하는지 알아보았다. 대략 보면 main 함수에서 사용되고 있는 함수들은 sub_401554, memset, scanf, sub_4011C0 이렇게 4 개가 보이고, jge 명령어를 통해 if문이 사용되고 있음을 알 수 있다.

호출되는 함수들 중에서 sub_401554와 sub_4011C0는 정확한 함수명이 사용되고 있지 않다. 그러나 sub_401554 함수의 경우 printf 함수인 것으로 쉽게 추측할 수 있을 것 같다. 이는 앞에서 프로그램을 실행한 후 특정 문자열들이 출력되는 것과 IDA로 디스어셈블링한 부분에서 특정 문자열이 스택에 push된 후 sub_401554 함수가 호출되는 것을 보면 알 수 있다.

그리고 조건문이 사용되고 있는 것도 쉽게 알 수 있다. 이는 **View** -> **Ghraphs** -> **Flow chart** 과정을 통해 분명하게 볼 수 있다. **Flow chart**는 **F12**를 클릭하면 바로 사용할 수 있다.



위의 flow chart를 보면 false일 경우 "You failed. Goodbye."가 출력되고, true일 경우 "You won. Goodbye."란 문자열이 출력된다. 핵심적인 부분은 **sub_4011C0** 함수 부분이다. 이 함수를 분석하면 실마리를 찾을 수 있을 것 같다.

대략적으로 확인을 했으니 이제 본격적인 분석에 들어간다.

~~~~~~

```
1 ; Attributes: bp-based frame

2 ; int __cdecl main(int argc, const char **argv, const char *envp)
3 _main proc near

4 Dst= byte ptr -80h
5 argc= dword ptr  8
6 argv= dword ptr  0Ch
7 envp= dword ptr  10h

8  push    ebp
9  mov     ebp, esp
10 sub     esp, 80h
11 push    offset aReverseEnginee ; "Reverse Engineering with IDA ProWnExampl"...
12 call    sub_401554
13 add     esp, 4
14 push    offset aPleaseProvideT ; "[*] Please Provide the password to cont"...
15 call    sub_401554
16 add     esp, 4
17 push    80h             ; Size
18 push    0               ; Val
19 lea     eax, [ebp+Dst]
20 push    eax             ; Dst
21 call    _memset
22 add     esp, 0Ch
23 lea     ecx, [ebp+Dst]
24 push    ecx
25 push    offset a127s    ; "%127s"
26 call    _scanf
27 add     esp, 8
28 lea     edx, [ebp+Dst]
29 push    edx             ; Str2
```
~~~~~~

```
30 call    sub_4011C0
31 add     esp, 4
32 movsx   eax, al
33 test    eax, eax
34 jge     short loc_4012D9


35 push    offset aYouFailed_Good ; "You failed. Goodbye.₩n"
36 call    sub_401554
37 add     esp, 4
38 jmp     short loc_4012E6


39 loc_4012D9:             ; "You won. Goodbye.₩n"
40 push    offset aYouWon_Goodbye
41 call    sub_401554
42 add     esp, 4


43 loc_4012E6:
44 mov     eax, 1
45 mov     esp, ebp
46 pop     ebp
47 retn
48 _main endp
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

; int __cdecl main(int argc, const char **argv, const char *envp)

이 부분은 C 언어의 소스에서는 다음과 같이 표현된다.

```
int main(int argc, char *argv[])
```

그리고 "; int __cdecl main(int argc, const char **argv, const char *envp)"에서 "**__cdecl**"는 함수를 호출하는 방법을 나타낸다. 보통 **Calling Convention**라고 부르는데, 여기에는 __cdecl, pascal, _stdcall, _fastcall 4가지 방법이 있다. 그 중 **__cdecl**는 다음과 같이 4가지 특징이 있다.

- Arguments Passed from Right to Left
- Calling Function Clears the Stack
- 'this' pointer is passed via stack last in case of Programs using OOP
- Functions using _cdecl are preceded by an "_"

이 4가지 함수 호출 방법들 사이의 큰 차이점은 스택 정리 방법에 있는데, __cdecl은 스택 정리를 위해서 add 명령어가 추가된다. 이것의 예는 13번째 라인의 "add esp, 4"에서 볼 수 있다.

라인 3 부분을 살펴보자.

```
3 _main proc near
```

여기서는 **proc**라는 표준 어셈블러의 directive가 사용되고 있다. 이 proc이라는 디렉티브는 프로시저의 시작을 알리는 것이다. 참고로 프로시저의 끝을 알리는 디렉티브는 **endp**이다. **near**는 현재 세그먼트 안에 위치한 목적지를 의미한다. 이와 대조적인 것이 **far**로, 이는 다른 세그먼트의 위치를 가리킨다. 그래서 라인 3은 현재 세그먼트 내에서 main 함수가 시작됨을 나타낸다.

4 ~ 7번까지는 ebp로부터 각 값의 위치를 나타내고 있다. 4 ~7까지에서 나오는 dword는 32비트를 나타내고, **ptr**은 연산과정에서 피연산자의 크기에 구애 받지 않고자 할 때 사용하는 어셈블러 연산자이다.

```
4 Dst= byte ptr -80h   // 스택의 크기는 128 바이트
5 argc= dword ptr  8   // ebp로부터 8 바이트 떨어진 위치
6 argv= dword ptr  0Ch // ebp로부터 12 바이트 떨어진 위치
7 envp= dword ptr  10h // ebp로부터 16 바이트 떨어진 위치
```

이를 도식도로 나타내면 다음과 같다.

buffer (128 바이트)	**saved ebp** (4 바이트)	**return address** (4 바이트)	**argc** (4 바이트)	**argv** (4 바이트)	**envp** (4 바이트)

라인 8 ~ 10번까지를 살펴보자. 8 ~ 10번까지는 함수의 호출 과정에서 필수인 procedure prelude 과정으로써, 이를 통해 현재의 stack pointer를 새로운 frame pointer로 만든다. 하나의 새로운 스택 프레임이 만들어지는데, 이 과정에서 로컬 변수를 위한 공간이 할당된다. procedure prelude에 대해서는 이 글을 읽는 사람들에게 별도 설명이 필요하지 않을 것이다.

```
8  push    ebp
9  mov     ebp, esp
10 sub     esp, 80h // 128 바이트
```

라인 2 ~ 10까지 C 언어로 나타내면 대략 다음과 같을 것이다.

```
int main(int argc, char *argv[])
{
        int array[128];
         …
```

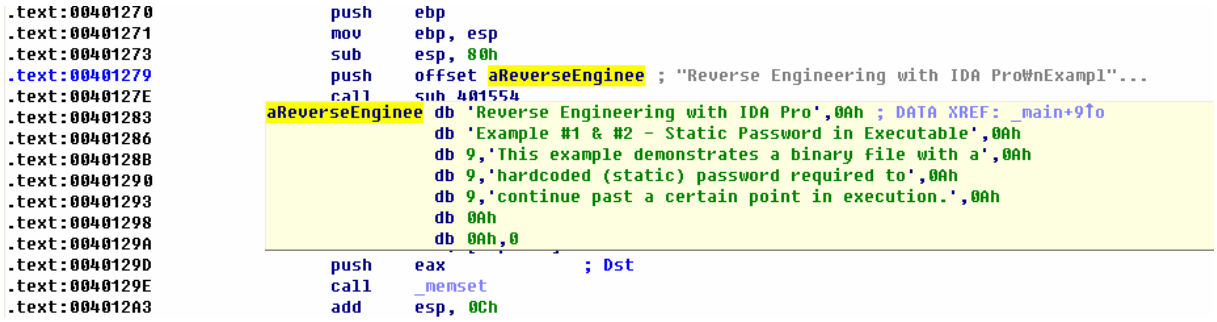
이제 다음 라인들을 살펴본다.

```
11 push    offset aReverseEnginee ; "Reverse Engineering with IDA ProWnExampl"...
12 call    sub_401554
13 add     esp, 4
14 push    offset aPleaseProvideT ; "[*] Please Provide the password to cont"...
15 call    sub_401554
16 add     esp, 4
```

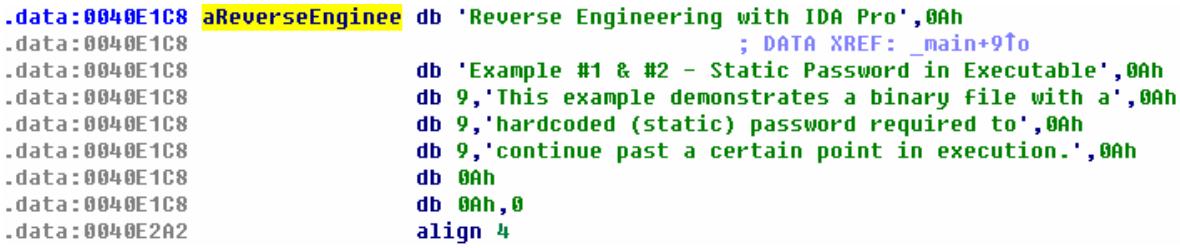
11 ~ 16 라인은 앞에서 프로그램을 실행했을 때 보았던 startup header 출력 부분이다. 즉, 다음 출력 부분이다.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Reverse Engineering with IDA Pro
        Example #1 & #2 - Static Password in Executable
                This example demonstrates a binary file with a
                hardcoded (static) password required to
                continue past a certain point in execution.

        [*] Please Provide the password to continue
        Password: reversing
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

11라인에서 스택에 aReverseEnginee 부분에 해당하는 데이터를 push하고 있다. 여기서 offset은 스택 세그먼트를 기준으로 한다. aReverseEnginee 부분에 마우스를 올리면 다음과 같이 해당 문자열들이 보인다.

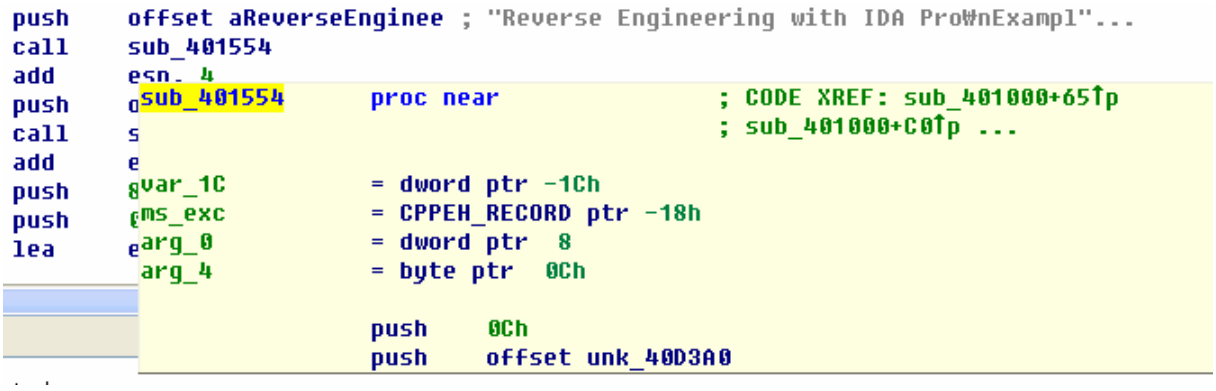


자세한 내용 확인을 위해 Enter를 이용해 해당 위치로 이동해보자.



aReverseEnginee에 해당되는 데이터는 .data 섹션 ".data:0040E1C8"에 위치해 있는 것을 볼 수 있다. .data 섹션에 있는 startup header 내용을 보았다. 다시 원래의 entry point로 돌아가기 위해 Esc키를 누른다.

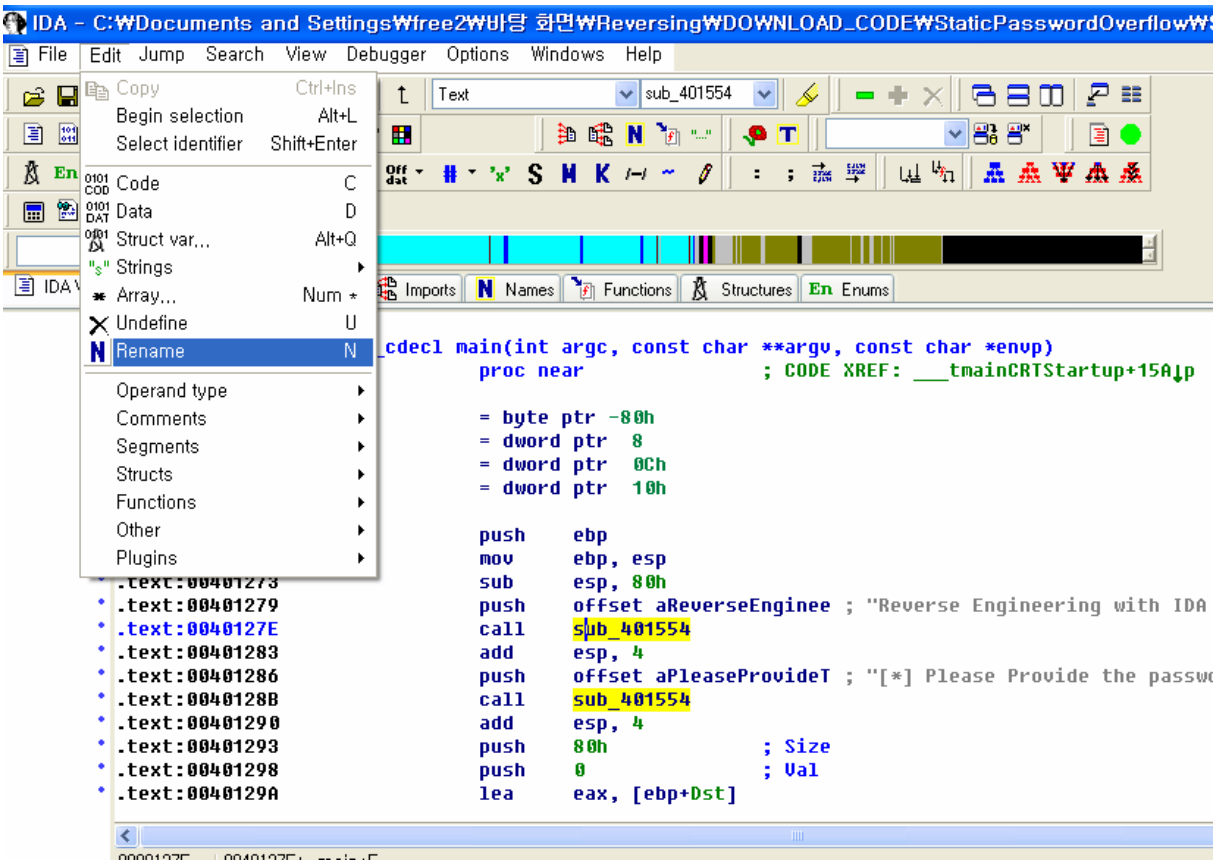
라인 **11**에서 출력될 데이터를 push한 후, 라인 **12**에서는 **sub_401554**라는 함수를 호출하고 있다. 프로그램을 실행시켰을 때 startup header 내용이 출력된 것으로 보아 **sub_401554**라는 함수는 printf 함수라는 것을 쉽게 추측할 수 있다. 확인을 위해 **sub_401554**라는 서브 루틴 안으로 들어가본다.



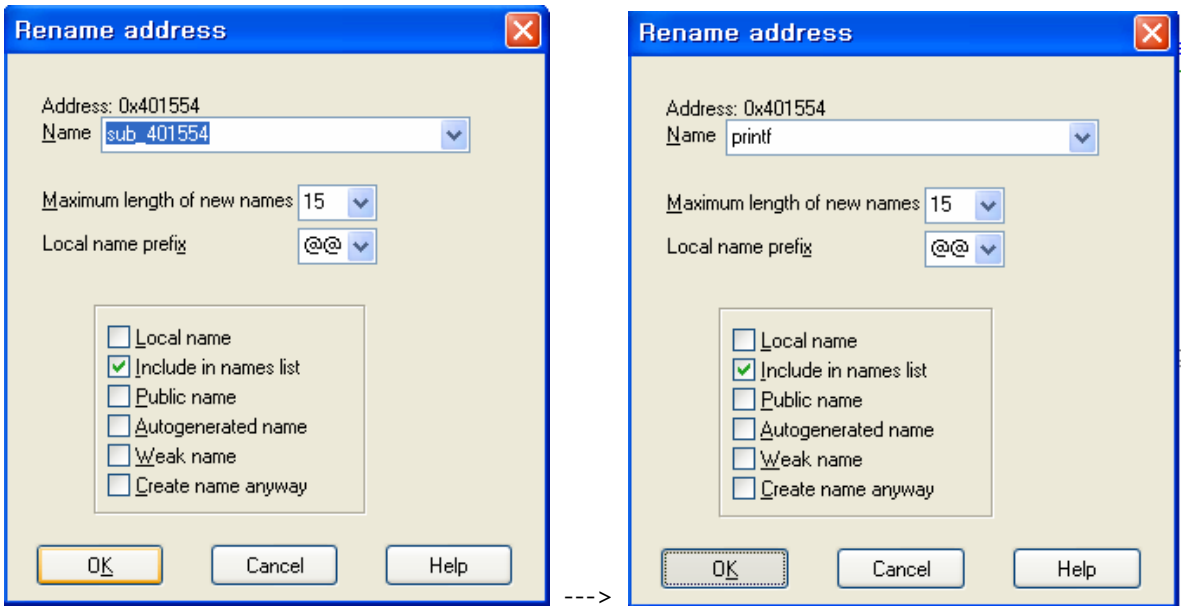

```
.text:00401554
.text:00401554 ; =============== S U B R O U T I N E =======================
.text:00401554
.text:00401554 ; Attributes: bp-based frame
```

```
.text:00401554
.text:00401554 sub_401554       proc near               ; CODE XREF: sub_401000+65↑p
.text:00401554                                          ; sub_401000+C0↑p ...
.text:00401554
.text:00401554 var_1C           = dword ptr -1Ch
.text:00401554 ms_exc           = CPPEH_RECORD ptr -18h
.text:00401554 arg_0            = dword ptr  8
.text:00401554 arg_4            = byte ptr  0Ch
.text:00401554
.text:00401554                  push    0Ch
.text:00401556                  push    offset unk_40D3A0
.text:0040155B                  call    __SEH_prolog4
```

내용을 보면 보면 **sub_401554** 함수가 printf 함수라는 부분은 나오지 않는다. 그러나 쉽게 추측할 수 있고, 앞으로의 편의를 위해 이 함수를 printf 함수로 이름을 수정한다. 수정하지 않아도 되지만 안으로 깊게 들어갈수록 헷갈릴 수 있으므로 여기서 아예 함수명을 수정한다. 수정 방법은 다음과 같다. 수정할 함수명 위에 커서를 올려두고, Edit 부분의 Rename을 먼저 선택한다.



그런 후 다음 창에서 수정하면 된다.



다음과 같이 함수명이 printf로 수정되었다.

```
.text:00401279          push    offset aReverseEnginee ; "Reverse Engineering with IDA Pro
.text:0040127E          call    printf
.text:00401283          add     esp, 4
.text:00401286          push    offset aPleaseProvideT ; "[*] Please Provide the password
.text:0040128B          call    printf
.text:00401290          add     esp, 4
.text:00401293          push    80h             ; Size
```

라인 13에서는 라인 12에서 함수가 호출되고, 그 작업이 끝나자 스택을 클리어하는 작업이 나와 있다. 앞에서도 살펴보았지만 함수 호출 방법에서 __cdecl은 스택 정리를 위해서 add 명령어가 추가된다고 했다. "add esp, 4" 부분은 **sub_401554** 함수가 그 역할을 하기 위해 필요한 주소 공간을 위해 할당된 4 바이트를 원상복귀 시킨다. 스택은 아래로 자라므로 4 바이트를 'add'했다. 4 바이트를 add 한 것은 **sub_401554** 함수의 아규먼트 1개 aReverseEnginee의 주소 값 때문이다.

라인 14 ~ 16까지는 라인 11~13까지와 같은 형태의 작업이므로 별도의 설명은 하지 않는다. 이제 라인 11 ~ 16을 C 언어로 나타내면 다음과 같을 것이다.

```
printf("Reverse Engineering with IDA Pro\n"
       "Example #1 & #2 - Static Password in Executable\n"
       "\tThis example demonstrates a binary file with a\n"
       "\thardcoded (static) password required to\n"
       "\tcontinue past a certain point in execution.\n\n\n");

printf("[*] Please Provide the password to continue\nPassword: ");
```

이제 라인 17 ~ 22까지 살펴보자.

```
17 push    80h             ; Size
18 push    0               ; Val
19 lea     eax, [ebp+Dst]
20 push    eax             ; Dst
21 call    _memset
22 add     esp, 0Ch
```

이 부분을 보면 memset 함수가 사용되고 있음을 알 수 있다. 라인 **17 ~ 18**에서는 memset이 초기화할 공간 배열 array의 크기는 80h, 즉 128 바이트("push 80h")로 잡고, 그 공간을 null로 채운다("push 0"). 이는 다음에서 살펴보겠지만 초기화된 buffer에 scanf 함수를 이용해 데이터를 읽어들이기 위한 것이다. 라인 **19 ~20**에서는 ebp+Dst의 주소를 eax에 로딩하여 스택에 push한다. Dst의 값은 라인 4에서 볼 수 있다. memset 함수를 호출하는데 필요한 값들이 모두 push된 후 라인 21에서 memset 함수가 호출된다. 이 전체 과정을 C 언어로 나타내면 다음과 같다.

```
memset(array, 0x00, 128);
```

```
.text:00401620
.text:00401620 ; Attributes: library function
.text:00401620
.text:00401620 ; void *__cdecl memset(void *Dst, int Val, size_t Size)
.text:00401620 _memset         proc near               ; CODE XREF: sub_401000+2E↑p
.text:00401620                                         ; sub_401000+44↑p ...
.text:00401620
.text:00401620 Dst             = dword ptr  4
.text:00401620 Val             = byte ptr  8
.text:00401620 Size            = dword ptr  0Ch
.text:00401620
.text:00401620                 mov     edx, [esp+Size]
.text:00401624                 mov     ecx, [esp+Dst]
.text:00401628                 test    edx, edx
.text:0040162A                 jz      short toend
.text:0040162C                 xor     eax, eax
.text:0040162E                 mov     al, [esp+Val]
.text:00401632                 test    al, al
.text:00401634                 jnz     short loc_40164C
.text:00401636                 cmp     edx, 100h
.text:0040163C                 jb      short loc_40164C
```

memset 함수의 작업이 끝난 후 다시 스택을 클리어하기 위해 라인 22에서 "add esp, 0Ch" 작업이 이루어진다. 여기서 **0CH**, 즉 12바이트를 add한 것은 위의 그림에서 보듯 memset의 아규먼트 3개 **array(void *Dst), 0x00(int Val), 128(size_t Size)**의 주소 값 때문이다.

이제 scanf 함수가 호출되는 라인 23 ~ 27을 살펴보자.

```
23 lea     ecx, [ebp+Dst]
```

```
24 push    ecx
25 push    offset a127s    ; "%127s"
26 call    _scanf
27 add     esp, 8
```

먼저 ecx에 ebp+Dst의 주소를 로딩하는데, 이곳은 이미 memset 함수에 의해 array[]의 공간이 0으로 초기화되어 있는 곳이다. 그리고 그 값이 저장된 ecx를 스택에 push하고, 그런 다음 "%127s"를 push한다.

```
.data:0040E2DC ; char a127s[]
.data:0040E2DC a127s          db '%127s',0          ; DATA XREF: _main+3A↑o
```

```
push    offset a127s    ; "%127s"
call    _scanf
add     es; char a127s[]
lea     ed a127s          db '%127s',0          ; DATA XREF: _main+3A↑o
```

scanf 함수에서 사용되는 '%127s'에서 알 수 있듯 버퍼에 127 바이트를 저장한다. 라인 23~27까지를 C 언어로 나타내면 다음과 같다.

```
scanf("%127s", &array);
```

라인 27의 "add esp, 8"를 통해 스택을 클리어한다. 여기서 8 바이트를 클리어하는 이유는 scanf의 아규먼트 2개 **"%127s"**와 **&array**의 주소 값이 들어가 있기 때문이다.

라인 1 ~ 27 여기까지는 실마리를 푸는 특별한 내용은 없다. 핵심적인 부분인 라인 28부터 시작된다. 28 ~ 34에서 sub_4011C0 함수와 조건문이 사용되고 있으며, 앞에서 **Flow chart**를 통해 살펴보았듯이 라인 34에서 분기가 된다. 일단 28 ~ 34까지의 내용을 살펴보자.

```
28 lea     edx, [ebp+Dst]
29 push    edx             ; Str2
30 call    sub_4011C0
31 add     esp, 4
32 movsx   eax, al
33 test    eax, eax
34 jge     short loc_4012D9
```

라인 28~29에서는 edx에 ebp+Dst의 주소 값을 로딩하고, 그 값을 push한다. 그런 다음 라인 30에서 sub_4011C0 함수가 호출된다. 라인 30 이후를 보면 sub_4011C0 함수가 가장 핵심적인 역할을

하고 있음을 알 수 있다. 그렇다면 이제 sub_4011C0 함수에 대해 알아볼 필요가 있다. 하지만 sub_4011C0 함수에 대해 알아보기 전에 main 함수 전체에 대해 먼저 간단하게 알아보는 것이 분석을 위해 더 효율적일 것이다.

라인 30에서 sub_4011C0 함수가 호출되고, 라인 31에서는 스택이 다시 클리어된다. 그런데 여기서 클리어되는 스택이 4바이트이다. 이는 sub_4011C0 함수가 아규먼트를 주소 값 하나만 가진다는 것을 알 수 있다. 이에 대해 확인을 해보면 다음과 같다.

```
.text:004011C0 ; =============== S U B R O U T I N E =======================================
.text:004011C0
.text:004011C0 ; Attributes: bp-based frame
.text:004011C0
.text:004011C0 ; int __cdecl sub_4011C0(char *Str2)
.text:004011C0 sub_4011C0      proc near               ; CODE XREF: _main+4B↓p
.text:004011C0
.text:004011C0 Dst             = byte ptr -80h
```

라인 32에서 사용되고 있는 movsx 명령은 "move with sign-extedn"를 의미하며, 레지스터의 상위 비트들을 모두 부호로 채운다. 여기서는 eax에 al의 값을 복사한다. main 함수 부분만으로는 al의 값을 알 수가 없다. al 값은 sub_4011C0 함수에 대해 알아볼 때 자세히 다루도록 하자. sub_4011C0 함수에서 해당 부분은 다음과 같다.

```
.text:00401241                or      al, 0FFh
.text:00401243                jmp     short loc_40125D
```

라인 33에서 test 명령이 수행되는데, test 명령은 첫 번째와 두 번째 오퍼랜드의 bitwise AND(논리곱)를 수행하고, 그런 다음 EFLAGS 레지스터에 flag를 설정한다. 피연산자 내의 각각의 비트가 1인지를 알아볼 때 사용된다. test 명령 후 jge 명령이 실행된다. jge의 의미는 다음과 같다.

Table 2.12 Conditional Jump Registers

Instruction	EFLAGS condition	Description
ja	*CF = 0 && ZF = 0*	Jump if above
jae	*CF = 0*	Jump if above or equal
jb	*CF = 1*	Jump if below
jbe	*CF = 1 || ZF = 1*	Jump if below or equal
jc	*CF = 1*	Jump if carry
jcxz	*CX = 0*	Jump if CX is zero
jecxz	*ECX = 0*	Jump is ECX is zero
je	*ZF = 1*	Jump if equal
jg	*ZF = 0 && SF = OF*	Jump if greater than
jge	*SF = OF*	Jump if greater than or equal to
jl	*SF != OF*	Jump if less than
jle	*ZF = 1 || SF != OF*	Jump if less than or equal to

라인 34를 충족시키면 loc_4012D9로 jump하는데, loc_4012D9의 내용은 다음과 같으며, 라인 39 ~ 42 부분이다.

```
.text:004012D9 loc_4012D9:                             ; CODE XREF: _main+58↑j
.text:004012D9                 push    offset aYouWon_Goodbye ; "You won. Goodbye.₩n"
.text:004012DE                 call    printf
.text:004012E3                 add     esp, 4
```

이 부분은 스택에 aYouWon_Goodbye 부분의 데이터인 "You won. Goodbye."을 push하고, printf 함수를 호출한다.

```
.data:0040E2FC aYouWon_Goodbye db 'You won. Goodbye.',0Ah,0 ; DATA XREF: _main:loc_4012D9↑o
```

그러나 라인 33 ~ 34를 충족시키지 못하면 라인 35 ~38로 분기한다.

```
35 push    offset aYouFailed_Good ; "You failed. Goodbye.₩n"
36 call    sub_401554
37 add     esp, 4
38 jmp     short loc_4012E6
```

라인 35는 스택에 aYouFailed_Good의 데이터를 push한다. 그런 다음 sub_401554를 호출하는데, sub_401554 함수는 printf 함수로 이름을 수정 한 것이다. sub_401554 함수가 호출되면 "You failed. Goodbye."가 호출되고, 다시 라인 37에서 스택이 클리어 된다. 그리고 라인 38이 실행된다. 라인 38에서 loc_4012E6의 내용은 다음과 같다. 이는 라인 43 ~ 48까지의 내용과 일치한다.

```
.text:004012E6 loc_4012E6:                             ; CODE XREF: _main+67↑j
.text:004012E6                 mov     eax, 1
.text:004012EB                 mov     esp, ebp
.text:004012ED                 pop     ebp
.text:004012EE                 retn
.text:004012EE _main           endp
```

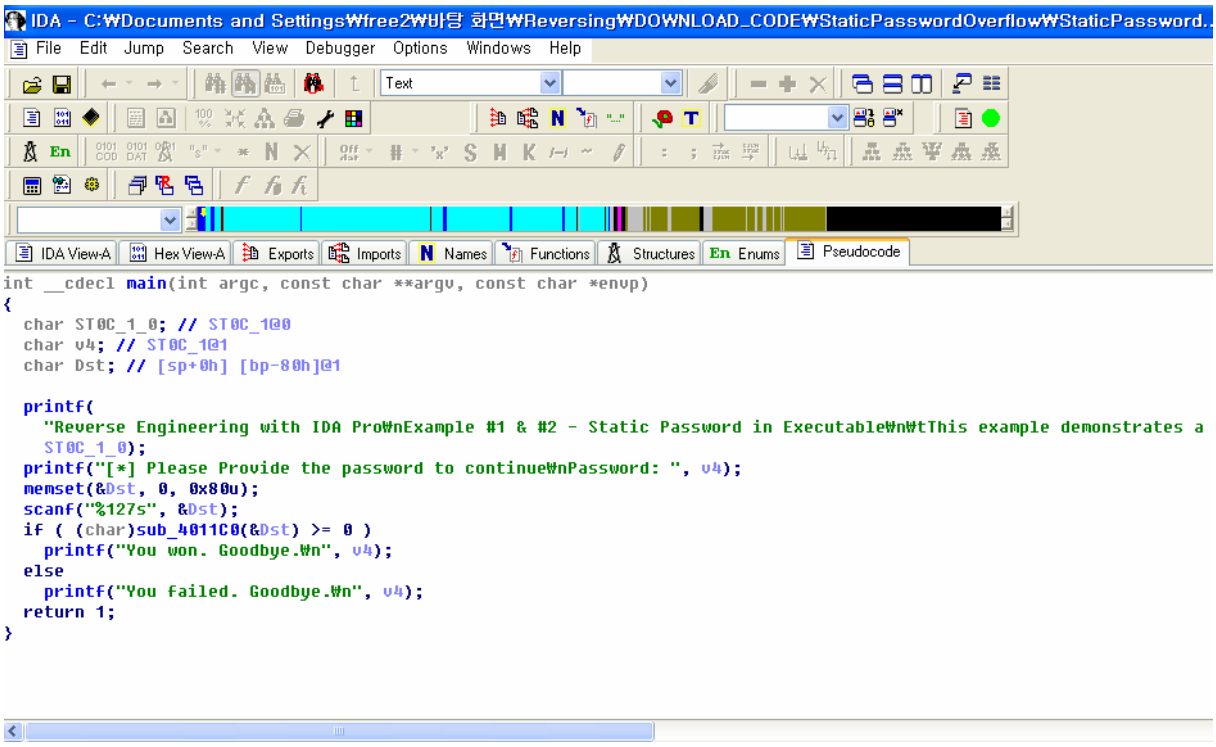
이 부분의 내용을 확인해보면, 먼저 eax에 1을 복사하는데, 이는 "return(1);"에서 사용되기 때문이다. 그런 다음 라인 45 ~ 46에서 procedure epilog가 실행되고, return(1); 한다. 그리고 프로시저의 끝을 알리는 디렉티브는 **endp**를 통해 main 함수가 종료된다.

```
.text:004012D7                 jmp     short loc_4012E6
.text:004012D9 ; ---------------------------------------------------------------------------
.text:004012D9
.text:004012D9 loc_4012D9:                             ; CODE XREF: _main+58↑j
.text:004012D9                 push    offset aYouWon_Goodbye ; "You won. Goodbye.\n"
.text:004012DE                 call    printf
.text:004012E3                 add     esp, 4
.text:004012E6
.text:004012E6 loc_4012E6:                             ; CODE XREF: _main+67↑j
.text:004012E6                 mov     eax, 1
.text:004012EB                 mov     esp, ebp
.text:004012ED                 pop     ebp
.text:004012EE                 retn
.text:004012EE _main           endp
.text:004012EE
```

loc_4012E6에 해당하는 부분을 IDA로 본 것이다. 이제 라인 28 ~ 42까지를 C 언어로 정리하면 다음과 같다.

```
if(check_user(&array) < 0) {
            printf("You failed. Goodbye.\n");
} else {
            printf("You won. Goodbye.\n");
}

return(1);
```

이상으로 main() 함수 전체를 살펴보았다. 마지막으로 막강한 성능을 보여주고 있는 Hex-Rays 플러그인을 이용한 Pseudocode를 알아보자. 책에서는 Hex-Rays에 대한 언급은 없다.



결과를 보니 우리가 여태 분석한 코드를 거의 그대로 보여주고 있다. 다소 차이가 있지만 핵심적인 부분은 변수, 함수의 아규먼트 등의 이름만 다를 뿐 그대로이다. 이는 취약점 찾기를 위해 바이너리 분석을 할 때 많은 도움이 될 것으로 보인다.

이제 **sub_4011C0(char *Str2)** 함수에 대해 분석을 해보자. 디스어셈블링한 결과는 책과 다소 다를 수 있다.

~~~~~~

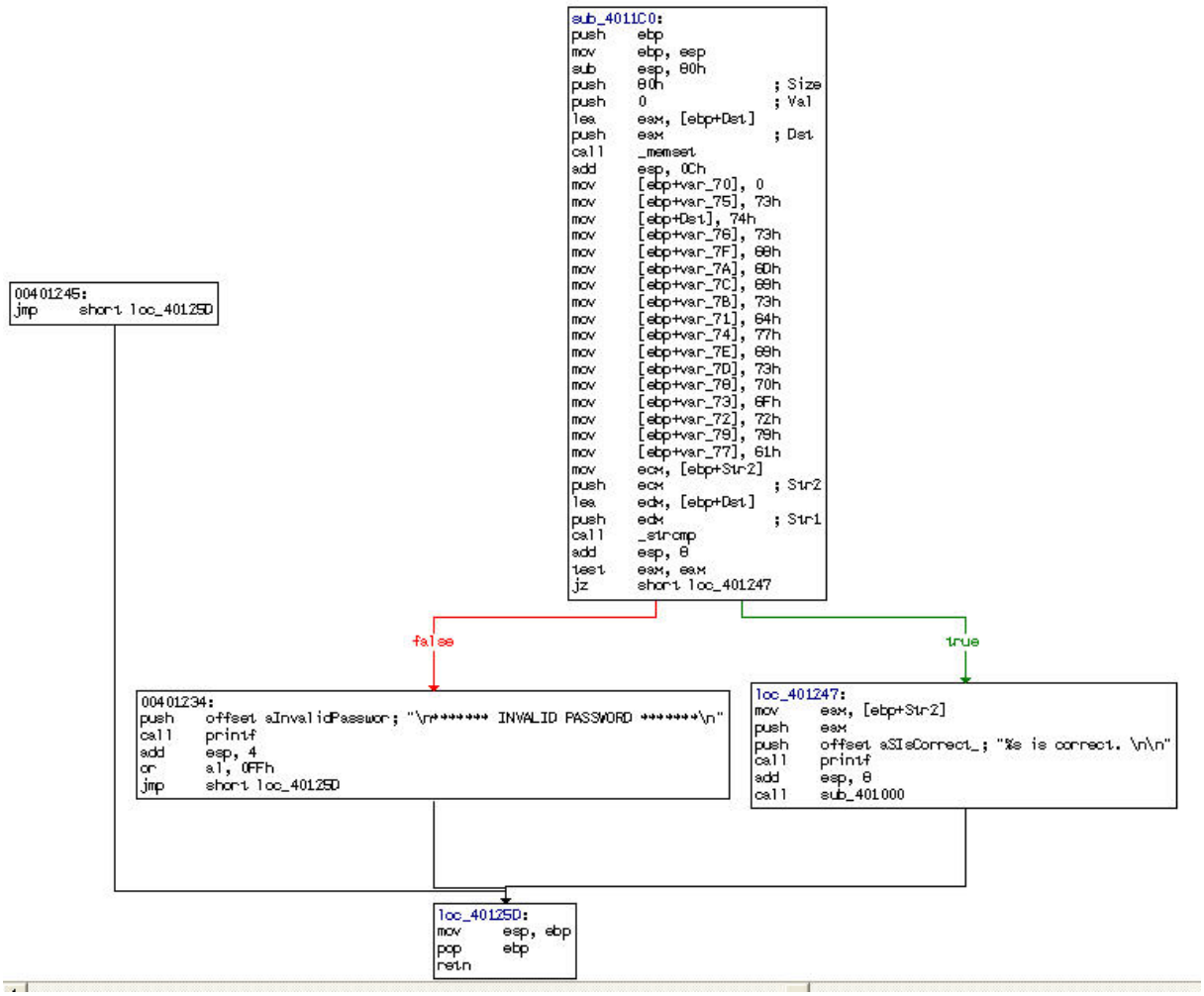
```
1   ; Attributes: bp-based frame

2   ; int __cdecl sub_4011C0(char *Str2)
3   sub_4011C0 proc near

4   Dst= byte ptr -80h
5   var_7F= byte ptr -7Fh
6   var_7E= byte ptr -7Eh
7   var_7D= byte ptr -7Dh
8   var_7C= byte ptr -7Ch
9   var_7B= byte ptr -7Bh
10  var_7A= byte ptr -7Ah
11  var_79= byte ptr -79h
12  var_78= byte ptr -78h
13  var_77= byte ptr -77h
14  var_76= byte ptr -76h
15  var_75= byte ptr -75h
16  var_74= byte ptr -74h
17  var_73= byte ptr -73h
18  var_72= byte ptr -72h
19  var_71= byte ptr -71h
20  var_70= byte ptr -70h
21  Str2= dword ptr  8

22  push    ebp
23  mov     ebp, esp
24  sub     esp, 80h
25  push    80h             ; Size
26  push    0               ; Val
27  lea     eax, [ebp+Dst]
28  push    eax             ; Dst
29  call    _memset
```
~~~~~~

```
30  add     esp, 0Ch
31  mov     [ebp+var_70], 0
32  mov     [ebp+var_75], 73h
33  mov     [ebp+Dst], 74h
34  mov     [ebp+var_76], 73h
35  mov     [ebp+var_7F], 68h
36  mov     [ebp+var_7A], 6Dh
37  mov     [ebp+var_7C], 69h
38  mov     [ebp+var_7B], 73h
39  mov     [ebp+var_71], 64h
40  mov     [ebp+var_74], 77h
41  mov     [ebp+var_7E], 69h
42  mov     [ebp+var_7D], 73h
43  mov     [ebp+var_78], 70h
44  mov     [ebp+var_73], 6Fh
45  mov     [ebp+var_72], 72h
46  mov     [ebp+var_79], 79h
47  mov     [ebp+var_77], 61h
48  mov     ecx, [ebp+Str2]
49  push    ecx             ; Str2
50  lea     edx, [ebp+Dst]
51  push    edx             ; Str1
52  call    _strcmp
53  add     esp, 8
54  test    eax, eax
55  jz      short loc_401247


56  push    offset aInvalidPasswor ; "Wn******* INVALID PASSWORD *******Wn"
57  call    printf
58  add     esp, 4
59  or      al, 0FFh
60  jmp     short loc_40125D


61 loc_401247:
62  mov     eax, [ebp+Str2]
63  push    eax
64  push    offset aSIsCorrect_ ; "%s is correct. WnWn"
```

```
65  call    printf
66  add     esp, 8
67  call    sub_401000


68  loc_40125D:
69  mov     esp, ebp
70  pop     ebp
71  retn
72  sub_4011C0 endp
```

~~~~~~

sub_4011C0(char *Str2) 함수의 흐름을 보면 다음과 같다.



이제 라인 별로 분석해보자. 라인 1 ~ 3은 별도의 설명이 필요하지 않을 것이다. 이 부분은 main 함수와 마찬가지이다. 라인 4 ~ 21까지 먼저 분석해보자. 4~21까지는 Dst와 변수들, 그리고 Str2라
~~~~~~

는 아규먼트의 위치를 나타내고 있다.

```
4  Dst= byte ptr -80h
5  var_7F= byte ptr -7Fh
6  var_7E= byte ptr -7Eh
7  var_7D= byte ptr -7Dh
8  var_7C= byte ptr -7Ch
9  var_7B= byte ptr -7Bh
10 var_7A= byte ptr -7Ah
11 var_79= byte ptr -79h
12 var_78= byte ptr -78h
13 var_77= byte ptr -77h
14 var_76= byte ptr -76h
15 var_75= byte ptr -75h
16 var_74= byte ptr -74h
17 var_73= byte ptr -73h
18 var_72= byte ptr -72h
19 var_71= byte ptr -71h
20 var_70= byte ptr -70h
21 Str2= dword ptr  8
```

라인 22~24까지는 procedure prolog에 해당되며, 라인 22~30까지는 앞에서 분석할 때 모두 언급한 부분이다. 여기서도 라인 25 ~ 29에서 **memset(array, 0x00, 128);** 과정이 나온다. 이 부분은 앞에서 자세하게 설명했으므로 여기서는 별도로 언급하지 않을 것이다. 라인 30은 스택을 클리어 하는 과정이다.

```
22 push    ebp
23 mov     ebp, esp
24 sub     esp, 80h
25 push    80h             ; Size
26 push    0               ; Val
27 lea     eax, [ebp+Dst]
28 push    eax             ; Dst
29 call    _memset
30 add     esp, 0Ch
```

이제 가장 핵심적인 부분(라인 31 ~ 47)이 나온다. ebp를 기준으로 잡힌 주소 공간에 문자열 하나씩 복사한다. 그러나 아래를 보면 var의 순서가 제대로 정렬되지 않은 체 배열되어 있다.

```
31  mov     [ebp+var_70], 0
32  mov     [ebp+var_75], 73h
33  mov     [ebp+Dst],    74h
34  mov     [ebp+var_76], 73h
35  mov     [ebp+var_7F], 68h
36  mov     [ebp+var_7A], 6Dh
37  mov     [ebp+var_7C], 69h
38  mov     [ebp+var_7B], 73h
39  mov     [ebp+var_71], 64h
40  mov     [ebp+var_74], 77h
41  mov     [ebp+var_7E], 69h
42  mov     [ebp+var_7D], 73h
43  mov     [ebp+var_78], 70h
44  mov     [ebp+var_73], 6Fh
45  mov     [ebp+var_72], 72h
46  mov     [ebp+var_79], 79h
47  mov     [ebp+var_77], 61h
```

그래서 var를 순서대로 정렬하고, 16진수가 나타내는 문자를 표시하기로 하겠다.

```
[ebp+var_70], 0    null
[ebp+var_71], 64h   -->   d
[ebp+var_72], 72h   -->   r
[ebp+var_73], 6Fh   -->   o
[ebp+var_74], 77h   -->   w
[ebp+var_75], 73h   -->   s
[ebp+var_76], 73h   -->   s
[ebp+var_77], 61h   -->   a
[ebp+var_78], 70h   -->   p
[ebp+var_79], 79h   -->   y
[ebp+var_7A], 6Dh   -->   m
[ebp+var_7B], 73h   -->   s
[ebp+var_7C], 69h   -->   i
[ebp+var_7D], 73h   -->   s
[ebp+var_7E], 69h   -->   i
[ebp+var_7F], 68h   -->   h
[ebp+Dst], 74h      -->   t
```

스택은 아래로 자라기 때문에 위의 결과를 정상적으로 배열하기 위해 ebp+Dst에서부터 ebp+var_70 순으로 배열하여 나온 문자열은 "**thisismypassword**"이다. 프로그램을 실행하여 이 문자열을 입력해보자.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\free2\바탕 화면>StaticPasswordOverflow.exe
Reverse Engineering with IDA Pro
Example #1 & #2 - Static Password in Executable
       This example demonstrates a binary file with a
       hardcoded (static) password required to
       continue past a certain point in execution.


[*] Please Provide the password to continue
Password: thisismypassword
thisismypassword is correct.

Please select an option from the following menu:
       Select
       Drop
       Create
       Exit
       ^C
C:\Documents and Settings\free2\바탕 화면>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

정확한 패스워드임을 알 수 있다. 이로서 이 장의 핵심인 패스워드 찾기가 끝났다. 실제로 이렇게 단순하게 패스워드를 보호하고 있는 프로그램들은 많이 없을 것이다. 분석한 이 프로그램은 단순히 리버싱 공부를 위해 기본적이면서 다양한 분석거리를 제공하고 있을 뿐이다.

핵심적인 작업은 끝났지만 나머지 라인들도 살펴보기로 하겠다. 라인 52에는 strcmp 함수가 사용되고 있다. strcmp 함수는 scanf 함수를 이용해 받아들였던 문자열을 비교한다. ecx에 ebp+Str2를 복사하고, 그 값을 스택에 push한다. 그리고 ebp+Dst의 값을 edx에 로딩하고, 그 값을 스택에 push한다. 그런 다음 strcmp 함수가 호출된다. strcmp 함수는 다음과 같이 작업한다.

~~~~~~

```
int strcmp ( string str1, string str2)
```

str1 이 **str2** 보다 작다면 < 0 을 반환하고; **str1** 이 **str2** 보다 크다면 > 0 을 반환한다. 두 문자열이 같다면 0 을 반환한다.

~~~~~~

여기서는 edx와 ecx에 들어간 값이 비교된다. 스택이 아래 자란다는 것을 감안하면 edx에 들어간 것이 str1 값이 되고, ecx에 들어간 값이 str2가 된다. edx에 들어간 값은 ebp+Dst에 있던 것인데, 이 값은 scanf로 읽어 들인 문자열이다. Str2은 라인 21에 그 위치가 나와 있다. strcmp 함수의 작업이 끝나면 스택을 클리어한다. strcmp 함수의 아규먼트가 2개(8 바이트)이므로 라인 53에서 "add esp, 8"를 사용했다.

```
48  mov     ecx, [ebp+Str2]
49  push    ecx             ; Str2
50  lea     edx, [ebp+Dst]
51  push    edx             ; Str1
52  call    _strcmp
53  add     esp, 8
```

이제 첫 번째와 두 번째 오퍼랜드의 bitwise AND(논리곱)를 수행하는데, 그 값이 **0**이면 loc_401247로 jump한다. jz는 "Jump if zero"를 의미한다.

```
54  test    eax, eax
55  jz      short loc_401247
```

라인 54~55의 결과가 **true**이면 jump하는 loc_401247의 내용은 다음과 같다. 이 부분은 라인 61~67의 내용이다.

loc_401247:

```
mov     eax, [ebp+Str2]
push    eax
push    offset aSIsCorrect_ ; "%s is correct. WnWn"
call    printf
add     esp, 8
call    sub_401000
```

eax에 ebp+Str2의 값을 복사하고 스택에 push한다. 그런 다음 "%s is correct."를 push한 후 printf

함수가 호출되면 "**thisismypassword is correct.**"를 출력한다. 그리고 스택이 클리어 된다. 그런 다음 sub_401000 함수를 호출한다. sub_401000 함수의 내용은 다음과 같다.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; Attributes: bp-based frame

sub_401000 proc near

Dst= byte ptr -4D0h
var_450= byte ptr -450h
Dest= byte ptr -400h

push    ebp
mov     ebp, esp
sub     esp, 4D0h
push    esi
push    edi
mov     ecx, 13h
mov     esi, offset aPleaseSelectAn ; "Please select an option from the follow"...
lea     edi, [ebp+var_450]
rep movsd
movsw
movsb
push    80h             ; Size
push    0               ; Val
lea     eax, [ebp+Dst]
push    eax             ; Dst
call    _memset
add     esp, 0Ch
push    80h             ; Size
push    0               ; Val
lea     ecx, [ebp+Dest]
push    ecx             ; Dst
call    _memset
add     esp, 0Ch
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

sub_401000 함수의 내용 일부는 앞에서 프로그램 실행 시 확인했으며, 이 장에서 핵심적인 부분은

아니므로 그냥 넘어가도록 하겠다. 아 귀차니즘...

라인 54~55의 결과가 false이면 이동하는 라인 56~60을 살펴보자.

```
54  test    eax, eax
55  jz      short loc_401247

56  push    offset aInvalidPasswor ; "₩n******* INVALID PASSWORD *******₩n"
57  call    printf
58  add     esp, 4
59  or      al, 0FFh
60  jmp     short loc_40125D
```

먼저 스택에 aInvalidPasswor의 데이터를 push하고, printf 함수를 호출하여 aInvalidPasswor의 데이터 내용을 출력한다. aInvalidPasswor의 데이터 내용은 "******* INVALID PASSWORD *******"이다. 라인 58에서 스택을 클리어 한다. 라인 59에서는 두 오퍼랜드 사이에 논리 OR 연산을 수행하고, 그 결과를 al에 저장한다. 그리고 라인 60에서는 loc_40125D로 jump한다. loc_40125D의 내용은 다음과 같다.

```
loc_40125D:
mov     esp, ebp
pop     ebp
retn
sub_4011C0 endp
```

loc_40125D에서는 procedure epilog 과정을 거쳐 sub_4011C0(char *Str2) 함수가 종료된다. 이것은 라인 68~72까지의 내용이기도 하다.

```
68  loc_40125D:
69  mov     esp, ebp
70  pop     ebp
71  retn
72  sub_4011C0 endp
```

이 바이너리 분석의 핵심적인 부분은 이제 대략 끝낸 셈이다.