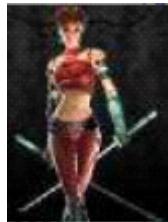


# 03.Basic nag removal + header problems

2012년 1월 27일 금요일  
오후 9:49

## Lena Reversing



03.Basic nag removal + header problems

번역자 : re4lfl0w / re4lfl0w@gmail.com

Last updated : 2012.03.28

Hello everybody.

안녕 모두들.

Welcome to this Part3 in my series about reversing for newbies/beginners.

나의 reversing 초보자를 위한 series Part 3에 온 것을 환영해.

This "saga" is intended for complete starters in reversing, even for those without any programming experience at all.

이 saga의 대상은 reversing에서 Programming 경험조차 없는 완벽한 초보자다.

Set your screen resolution to 1152\*864 and press F11 to see the movie full screen !!!

Again, I have made this movie interactive.

You screen 해상도를 1152\*864로 설정해 그리고 full screen으로 movie를 보기 위해 F11를 눌러

So, if you are a fast reader and you want to continue to the next screen, just click here on this invisible hotspot. You don't see it, but it IS there on text screens.

그래서, 네가 이것을 빨리 읽고 다음 screen을 보고 싶다면, 보이는 hotspot 여기를 눌러. 보고 싶지 않을 때는 여기에 두지마.

Then the movie will skip the text and continue with the next screen.

Movie는 text와 다음 screen을 skip할 수 있다.

If something is not clear or goes too fast, you can always use the control buttons and the slider below on this screen.

무언가 명확하지 않거나 빨리 넘기고자 할 때, 항상 control button과 이 screen 밑에 있는 slider 바를 사용해.

He, try it out and click on the hotspot to skip this text and to go to the next screen now!!!

도전해봐. 그리고 이 text와 다음 screen을 보기 위해 hotspot을 click해.

During the whole movie you can click this spot to leave immediately

이 movie 어디에서나 즉시 떠나기 위해 이 spot을 click 할 수 있다.

Click here as soon as you finished reading(on each screen !)

네가 읽기가 끝났을 때 이 곳을 클릭해. (이번 screen에서)

## 1. Abstract

In this Part 3, we will reverse a RegisterMe. Intention is to kill both nags, as well the starting as the closing nag.

이번 Part 3에서는, 우리는 RegisterMe를 reverse 할 것이다. 목적은 시작할 때, 끝날 때 2개의 쓸데없는 nag을 제거하는 것이다.

A nag is a window that does not show when the program is registered. You must understand that a program (often) already at startup verifies if it is registered or not.

Program을 등록하면 Nag은 보여지지 않는 window다. Program이 이미 시작할 때 등록이 되어있는지 검증한다는 것을 이해할 것이다.

If it is unregistered, the nag asking to register is shown. If registered, the nag is not shown. 만약에 등록이 되어있지 않다면, nag은 등록하라고 한다. 만약에 등록이 되어 있다면 nag은 보여지지 않는다.

This already makes clear that this is a perfect indication to find the registration references!

이것은 이미 등록하는 reference를 찾는 것이라는 것을 명확하게 알려준다.

The goal is NOT to study this particular case but to try to teach some basic techniques.

이번 부분 case에서는 Study 하는 게 목표가 아니다. 그러나 약간의 기본적인 기술은 가르치겠다.

In case you are a complete starter, I advise you to first see the previous parts in this series before seeing this movie: this Part really relies on previous Parts.

이번 케이스는 네가 완전히 처음 시작하는 사람이라면, 이 movie를 보기 전에 먼저 지난번 parts를 보고 와라.

Exploit your new gained knowledge in a positive way.

얻은 지식으로는 긍정적인 방향으로 이용해라.

in this matter, I also want to refer to Part 1.

이 문제에 대해, Part 1을 참고하기를 원한다.

As always, some of these writings are mine, others have been collected by me from different sources. Thanks to the authors.

항상, 이들 작품 중에서 일부는 내 거다. 나에 의해 다른 source에서 글쓰기 위한 재료들을 수집했다. 그 authors에게 감사한다.

## 2. Tools and target

The tools for today are : Ollydebug and ... your brain.

오늘의 툴은 : Ollydebug와 너의 두뇌다.

The first can be obtained for free at

첫번째로 얻을 것은 무료다.

<http://www.ollydbg.de>

... the second is your responsibility ;)

두번째는 너의 책임감이다.

For your convenience, I included the target (RegisterMe) in this package. It is a very basic and easy to reverse "RegisterMe".

너의 편리함을 위해, 나는 이번 target을 이 package에 포함했다. 그것은 매우 기본적이고 "RegisterMe"를 reverse 하기 쉽다.

I also included the second target "RegisterMe.Ooops.exe" in this package.

나는 또한 두번째 target인 "RegisterMe.Ooops.exe"를 이 package에 포함했다.

## 3. Study of the target

Study of the target is always extremely important. It can give you detailed info on how to start. Good research on working of the application can drastically reduce the time you spend on it afterwards.

Target 공부는 항상 절대적으로 중요하다. 그것은 너에게 어떻게 시작해야 할지 자세한 정보를 준다. Application 행동을 연구하는 것은 나중에 많은 시간을 줄여준다.

So, let's do that together. Like you can see, I have already opened the RegisterMe in Olly.

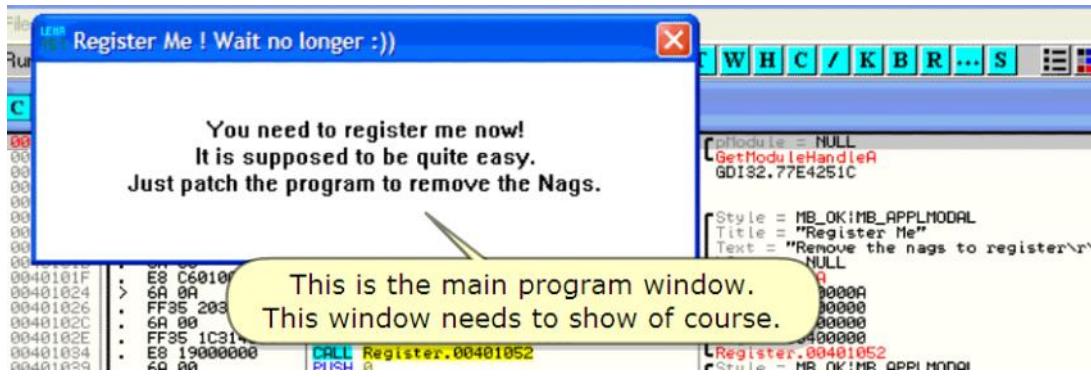
From now on, I will suppose you know how to work with Olly's basics.

그래서, 함께 시작하자. 네가 보는 거와 같이, 나는 이미 RegisterMe를 Olly에 열어놨다. 그럼 이제, 네가 어떻게 Olly's의 기본을 배우는지 알고 있을 것이라고 생각할 것이다.



This is the starting nag telling you what needs to be accomplished. :)

Starting nag은 너에게 무엇을 성취해야 하는지 말해준다.

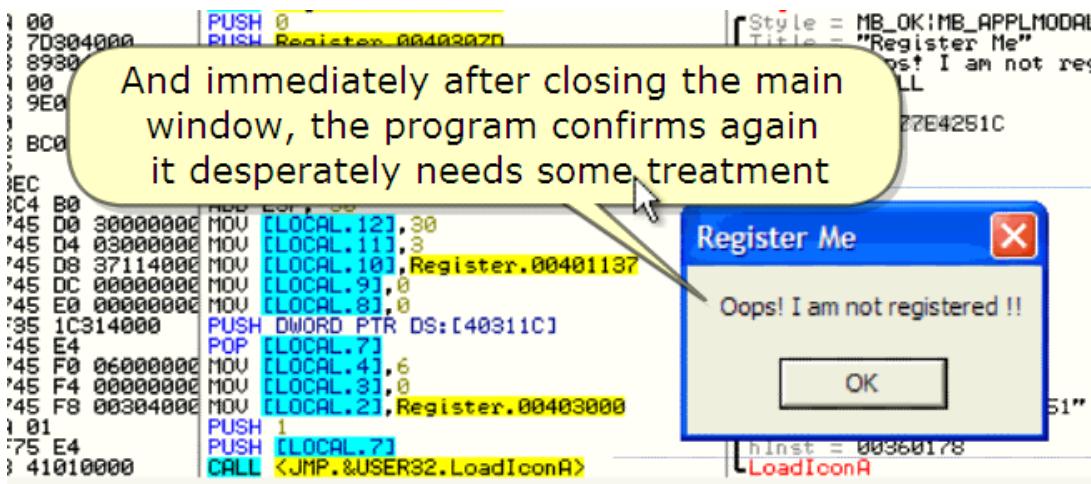


This is the main program window.

This window needs to show of course.

이것이 main program window다.

이 Window는 물론 보여줘야 됩니다.



And immediately after closing the main window, the program confirms again it desperately needs some treatment

그리고 즉시 main windows를 닫을 때, program은 지독하게 등록이 필요하다고 다시 보여준다.

Keep your mouse pointer here and click whenever you are ready reading on each textscren.

너의 mouse pointer는 여기를 유지하고 네가 textscren을 다 읽었을 때 click 해라.

And the RegisterMe exits. Let's restart and study in the code what we've seen  
Restart!

그리고 RegisterMe는 종료됐다. 다시 시작하고 우리가 봤던 code를 공부하자.

#### 4. Finding the patches

Beginning reversers often completely trust searching everything through text strings. I mean that their first reflex is always to search for recognizable stuff and then depart from there.

처음 시작하는 reverser는 자주 text strings을 통해 검색하는 모든 것이 완벽하다고 믿는다. 나는 첫번째로 검색 되어지는 것은 항상 검색하기 위해 사용할 수 있는 재료이고 그곳에서부터 검색을 시작한다.

However, the day they don't find text strings (due to packing, encryption, etc) ... they are completely lost.

그러나, 어떤 날은, 그들은 text strings을 찾을 수 없습니다. (packing, encryption, etc에 의해) 그들은 완벽히 사라진다.

So, let's NOT search text strings, but learn to think like a reverser.

Remark : of course you may use it, but as confirmation is preferred.

그래서, text strings을 검색하지 않는다, 그러나 reverser 같이 생각하기 위해 배워라.

주목 : 물론 너는 우선적으로 확인할 때 사용할 것이다.

The screenshot shows the CPU pane of Immunity Debugger with assembly code. A yellow callout box highlights the instruction at address 00401024, which is a `JE SHORT Register.00401024`. The assembly code is as follows:

```
00401000: $ 6A 00    PUSH 0
00401002: . E8 0020000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007: . A3 1C314000 MOU DWORD PTR DS:[40311C],ERX
0040100C: . 83F8 00    CMP EAX,0
0040100F: . 74 13      JE SHORT Register.00401024
00401011: . 6A 00      PUSH 0
00401013: . 68 7D304000 PUSH Register.00403070
00401018: . 68 34304000 PUSH Register.00403034
0040101D: . 6A 00      PUSH 0
0040101F: . E8 C010000 CALL <JMP.&USER32.MessageBoxA>
00401024: > 6A 00      PUSH 0
00401025: FF35 20    ADD EDI,20
0040102C: 6A 00      PUSH 0
0040102E: FF35 20    ADD EDI,20
00401034: 6A 00      PUSH 0
00401039: 68 71000000 PUSH 0
0040103B: 6A 00      PUSH 0
00401040: 68 83000000 PUSH 0
00401045: 6A 00      PUSH 0
00401047: E8 9E010000 CALL <JMP.&USER32.MessageBoxA>
0040104C: 6A 00      PUSH 0
0040104D: E8 BC010000 CALL <JMP.&KERNEL32.ExitProcess>
00401052: 55          PUSH EBP
00401053: 8BEC        MOV EBP,ESP
00401055: 83C4 B0    ADD ESP,-50
00401058: C745 D0 30000000 MOU [LOCAL.12],30
0040105F: C745 D4 03000000 MOU [LOCAL.11],3
00401066: C745 D8 37114000 MOU [LOCAL.10],Register.0040113?
0040106D: C745 D0 00000000 MOU [LOCAL.9],0
00401074: C745 E0 00000000 MOU [LOCAL.8],0
0040107B: FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401081: 8F45 E4    POP [LOCAL.4],6
00401084: C745 F0 06000000 MOU [LOCAL.3],6
00401088: C745 F4 00000000 MOU [LOCAL.2],0
00401092: C745 F8 00384000 MOU [LOCAL.1],Register.00403000
00401099: 6A 01      PUSH 1
0040109B: FF75 E4    PUSH [LOCAL.7]
0040109E: E8 41010000 CALL <JMP.&USER32.LoadIconA>
```

A yellow callout box contains the text: "Ok. What have we here? Here is already a compare and a JE. Let's step F8 and see..." A tooltip also points to the `JE` instruction.

The Registers pane shows the following values:

Module = NULL
GetModuleHandleA
Style = MB_OK MB_APPLMODAL
Title = "Register Me"
Text = "Remove the nags to register\r\nTI
hOwner = NULL
MessageBoxA
Arg4 = 00000000
Arg3 = 00000000
Arg2 = 00000000
Arg1 = 00000000
Register.00401052
Style = MB_OK MB_APPLMODAL
Title = "Register Me"
Text = "Oops! I am not registered !!"
hOwner = NULL
MessageBoxA
ExitCode = 0
ExitProcess

The Stack pane shows the following values:

kernel32.7C81604F
ASCII "RegisterMelena151"
RsrcName = 1,
hInst = 8054A6ED
LoadIconA

Ok. What have we here? Here is already a compare and a JE.

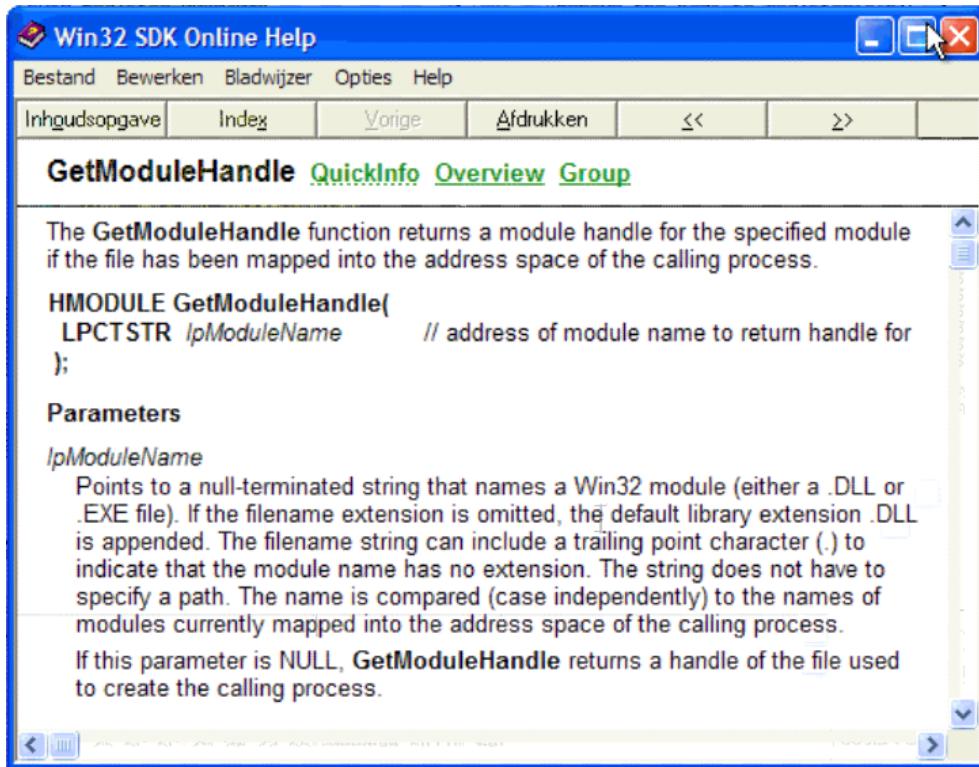
Let's step F8 and see...

Ok. 우리는 어디에 있는가? 이곳은 이미 JE에 비교됐다.

F8을 누르고 봐라.

Before the compare is a GetModuleHandleA API. Let's have a look in Win32.hlp

비교하는 것은 GetModuleHandleA API입니다. Win32.hlp를 봐라.



GetModuleHandleA returns the imagebase (address) of the RegisterMe.

Let's see it in the code.

GetModuleHandleA는 RegisterMe의 ImageBase(address)를 돌려준다.

Code를 봠라.

INFO : Imagebase == the address at which the windows loader maps the program into memory.

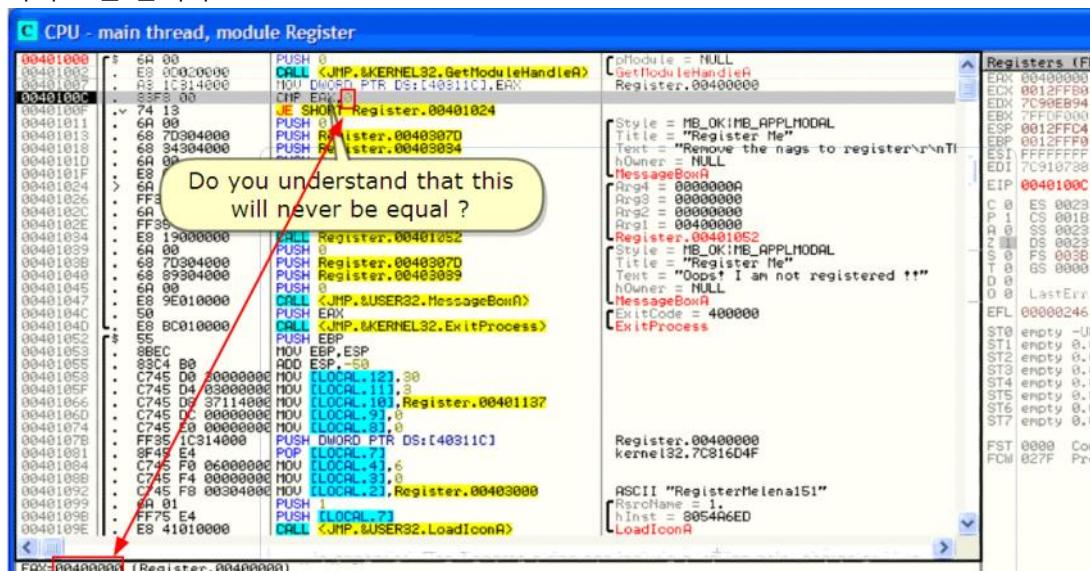
ImageBase == windows loader가 program을 memory에 일치시키는 address다.

번역 주 이해가 안 되도 일단 넘어가자. 그래도 모르겠으면 처음부터 다시 보고 다른 책들을 찾 아보면 이!! 하는 순간이 온다. 나를 믿으라.

All right. EAX == 400000 and what happens after the compare ? Step again F8

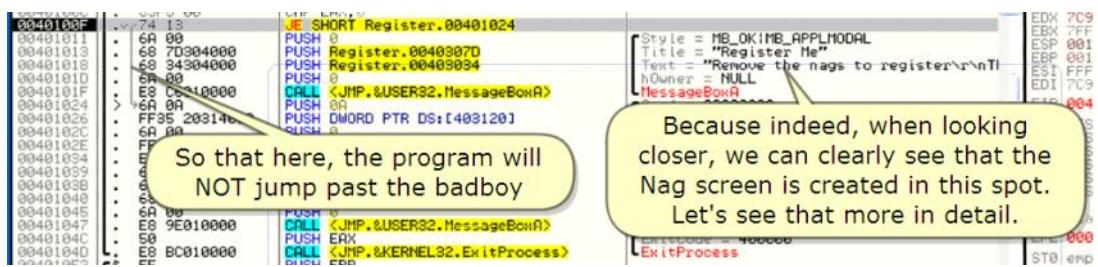
맞다. EAX == 400000 그리고 비교 후에 무슨 일이 일어났나?

다시 F8을 눌러라.



Do you understand that this will never be equal?

결코 같아지지 않는다는 것을 이해했어?



So that here, the program will NOT jump past the badboy

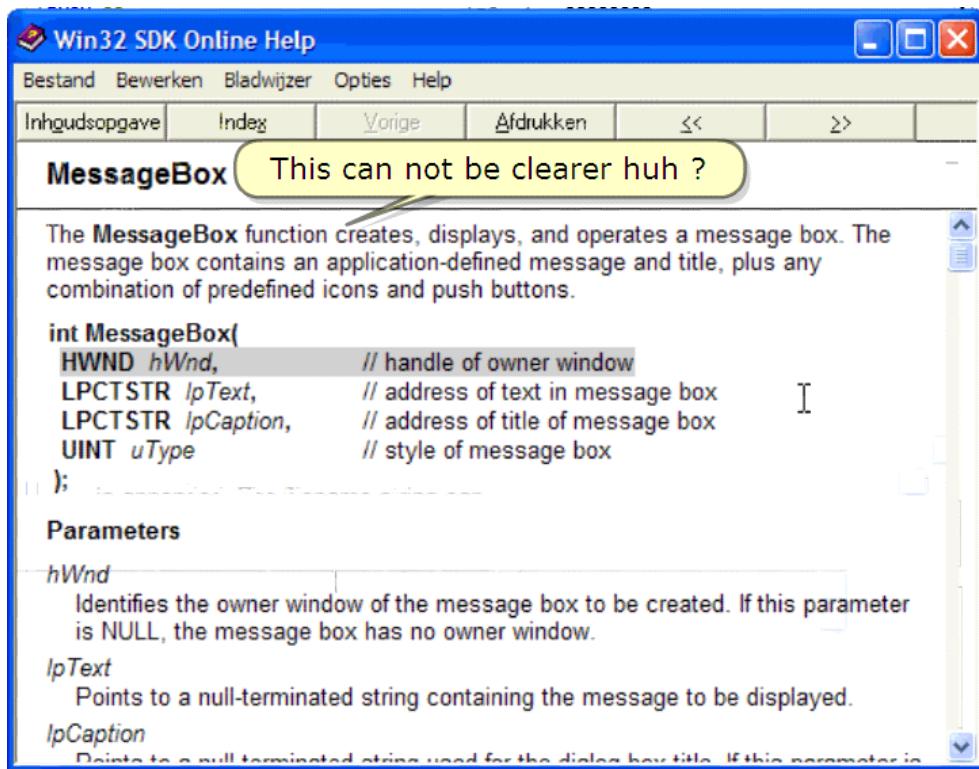
Because indeed, when looking closer, we can clearly see that the Nag screen is created in this spot.

Let's see that more in detail.

그래서, program은 과거의 badboy를 jump 하지 않는다.

그래서, 가까이서 볼 때, 우리는 명확히 이 위치에서 nag screen이 생성된 것을 볼 수 있다.

좀 더 자세히 보자.



This can not be clearer huh ?

이것은 명확하지 않느냐?

The handle from the owner window

Owner Windows의 handle 이다.

And these will be the title and the text in the messagebox.

그리고 이것들은 messagebox에서 title과 text를 나타낸다.

There are more "types" of messageboxes. (info, ...) but that is not important for us here.

Let's see in the code how we can avoid running to this nag screen :)

그곳에는 messagebox들의 type을 많다. 그러나 이것은 현재 우리에게 중요한 것은 아니다.

우리가 어떻게 naq screen을 회피할 수 있는지 Code를 봐라.

C CPU - main thread, module Register

```

00401000  $ 6A 00      PUSH 0
00401002  . E8 00200000 CALL QJMP.&KERNEL32.GetModuleHandleA>
00401007  . E8 1C314000 NOV DWORD PTR DS:[40311C],EAX
0040100C  . E8 34384000 CMP EAX,0
0040100F  > 74 13      JE SHORT Register.00401024
00401011  . E8 00
00401013  . E8 7D904000 PUSH Register.0040307D
00401018  . E8 34384000 PUSH Register.00403034
0040101F  > E8 C6010000 CALL QJMP.&USER32.MessageBoxA>
00401024  . E8 00
00401026  . FF35 20314000 PUSH DWORD PTR DS:[403120]
0040102C  . E8 00
0040102E  . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401034  . E8 19000000
00401039  . E8 00
0040103B  . E8 7D904000
00401040  . E8 89304000
00401045  . E8 00
00401047  . E8 9E010000
0040104C  . E8 BC010000
00401052  > E8 00
00401053  . E8 EC000000
00401055  . E8C4 B0    MOU EBP,ESP
00401059  . C745 D0 30000000 MOU [LOCAL_12],30
0040105F  . C745 D4 03000000 MOU [LOCAL_11],30
00401066  . C745 D8 37114000 MOU [LOCAL_10],30 Register.00401137
0040106D  . C745 DC 00000000 MOU [LOCAL_9],30
00401074  . C745 E0 00000000 MOU [LOCAL_8],30
00401078  . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401081  . C745 E4    POP [LOCAL_7]
00401084  . C745 F0 00000000 MOU [LOCAL_4],30
00401088  . C745 F4 00000000 MOU [LOCAL_3],30 Register.00403000
00401092  . C745 F8 00304000 MOU [LOCAL_2],30 Register.00401137
00401099  . E8 01
0040109B  . FF75 E4    PUSH I
0040109E  . E8 41010000 CALL QJMP.&USER32.LoadIconA>

```

Jump is NOT taken  
00401024=Register.00401024

It is clear that we want to jump the nag screen and I suppose that ...

... you remember how to do this temporarily (from previous parts in this series), right ?

It is clear that we want to jump the nag screen and I suppose that ...

... you remember how to do this temporarily (from previous parts in this series), right?

이것은 명확하다. 우리가 nag screen을 jump 하는 것을 원하고 내가 추정했다.

어떻게 임시적으로 사용할지 기억하고 있어? (이전 series의 parts에서), 그렇지?

00401000 \$ 6A 00 PUSH 0
00401001 . E8 00200000 CALL QJMP.&KERNEL32.GetModuleHandleA>
00401007 . E8 1C314000 NOV DWORD PTR DS:[40311C],EAX
0040100C . E8 34384000 CMP EAX,0
0040100F > 74 13 JE SHORT Register.00401024
00401011 . E8 00
00401013 . E8 7D904000 PUSH Register.0040307D
00401018 . E8 34384000 PUSH Register.00403034
0040101F > E8 C6010000 CALL QJMP.&USER32.MessageBoxA>
00401024 . E8 00
00401026 . FF35 20314000 PUSH DWORD PTR DS:[403120]
0040102C . E8 00
0040102E . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401034 . E8 19000000
00401039 . E8 00
0040103B . E8 7D904000
00401040 . E8 89304000
00401045 . E8 00
00401047 . E8 9E010000
0040104C . E8 BC010000
00401052 > E8 00
00401053 . E8 EC000000
00401055 . E8C4 B0 MOU EBP,ESP
00401059 . C745 D0 30000000 MOU [LOCAL\_12],30
0040105F . C745 D4 03000000 MOU [LOCAL\_11],30
00401066 . C745 D8 37114000 MOU [LOCAL\_10],30 Register.00401137
0040106D . C745 DC 00000000 MOU [LOCAL\_9],30
00401074 . C745 E0 00000000 MOU [LOCAL\_8],30
00401078 . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401081 . C745 E4 POP [LOCAL\_7]
00401084 . C745 F0 00000000 MOU [LOCAL\_4],30
00401088 . C745 F4 00000000 MOU [LOCAL\_3],30 Register.00403000
00401092 . C745 F8 00304000 MOU [LOCAL\_2],30 Register.00401137
00401099 . E8 01
0040109B . FF75 E4 PUSH I
0040109E . E8 41010000 CALL QJMP.&USER32.LoadIconA>

So, we jump the nag screen for this time. But remember for later that we will need to think of a permanent solution for this. Let's continue

So, we jump the nag screen for this time. But remember for later that we will need to think of a permanent solution for this. Let's continue

그래서, 우리는 이번 시간에 nag screen을 jump 할 것이다. 그러나 나중을 위해 기억해라. 우리는 영원한 풀이를 생각하기 위해 필요하다. 계속하자.

번역 주)쉽게 말해서, 이건 임시적으로 쓰는 거다. 왜냐하면 나중에 영원히 고치기 위해서 임시적으로 이 길이 맞는지 test 해봐야지 않겠어요?

We have jumped the nag screen.

Now follows the main program's window. Look carefully.

우리는 nag screen을 jump 했다.

이제 main program window를 따라가자. 조심해라.

The program runs fine. Now, let's again look further in the code at what's next

Program 실행이 잘 됐다. 이제, 다시 다음에 무엇이 있는지 코드를 보자.

A screenshot of the Immunity Debugger interface. The assembly pane shows a sequence of instructions starting with a CALL to Register\_00401052. A tooltip box highlights the next instruction, which is a CALL to `<JMP.&USER32.MessageBoxA>`. The tooltip text reads: "Ah ! Look, we land immediately in the next messagebox ... in the creation of the next nag screen. How are we going to handle this ... there simply is NO conditional jump to change ? Let's step the nag first to see it appear". The registers pane shows various CPU register values. The stack pane displays memory starting with the string "OKIMB\_APPLMODAL". The bottom status bar shows the current assembly address as 00401052.

Ah! Look, we land immediately in the next messagebox... in the creation of the next nag screen. How are we going to handle this...

아! 봐라, 우리는 즉시 다음 messagebox에 도착했다. 다음 nag screen을 만드는 곳이다. 어떻게 이것을 조종하지?

there simply is NO conditional jump to change? Let's step the nag first to see it appear

이것은 매우 간단하다. 조건 jump를 어떻게 바꿔? 먼저 Nag이 보이는지 보자.

The screenshot shows a debugger interface with assembly code and a UI dialog.

**Assembly Code:**

```
00401047 E8 3C100000 CALL <JMP.&USER32.MessageBoxA>
0040104C 50 PUSH EAX
0040104D E8 BC010000 CALL <JMP.&KERNEL32.ExitProcess>
00401052 55 PUSH EBP
00401053 BEC6
00401055 8BC6 B0
00401058 C745 D0 800000
0040105F C745 D4 000000
00401066 C745 D8 871140
0040106D C745 DC 000000
00401074 C745 E8 00000000 MOU [LOCAL_81],0
0040107B FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401084 8F45 E4 POP [LOCAL_7]
0040108B C745 F0 00000000 MOU [LOCAL_41],6
00401098 C745 F4 00000000 MOU [LOCAL_34],0
0040109E C745 F8 00304000 MOU [LOCAL_21],Register.00403000
004010A4 6A 81 PUSH [LOCAL_7]
004010A9 FF75 E4 PUSH [LOCAL_7]
004010B4 F3 00 JNSW <JMP.&USER32.MessageBoxA>
```

**UI Dialog:**

A modal dialog box titled "Register Me" contains the message "Oops! I am not registered !!".

**Registers:**

Register	Value
EAX	0012EC4
EBP	0012ECC4
ECX	00000000
EDX	00000000
ESI	00000000
EDI	00000000
ESP	0012ECC0
EBP	0012ECC4
ECR	00000000
ECX	00000000
EDX	00000000
ESI	00000000
EDI	00000000
ESP	0012ECC0
EBP	0012ECC4
ECR	00000000
ECX	00000000
EDX	00000000
ESI	00000000
EDI	00000000
ESP	0012ECC0

Mmm, let's think of a cure to remedy this...

음, 어떻게 해결할지 생각해 봐야겠다.

See what will happen if we keep stepping...

But let's restart and solve this thing ;)

우리가 계속 진행한다면 무엇이 일어날까 봐라.

그러나, 우리는 restart 하고 이 문제를 해결하겠다.

The question is : "What can we do to make the RegisterMe always jump the nag?".

In fact, this is quite simple and there are several solutions possible.

질문 : "어떻게 하면 RegisterMe가 항상 naq를 jump할 수 있을까?"

사실, 꽤 간단하고 여러가지 가능한 해결책들이 있다.

...so that we always jump over the nag,  
no matter what

For example : we could  
assemble this JE into JMP

But we also could NOP the  
messagebox (see further)

For example : we could assemble this JE into JMP

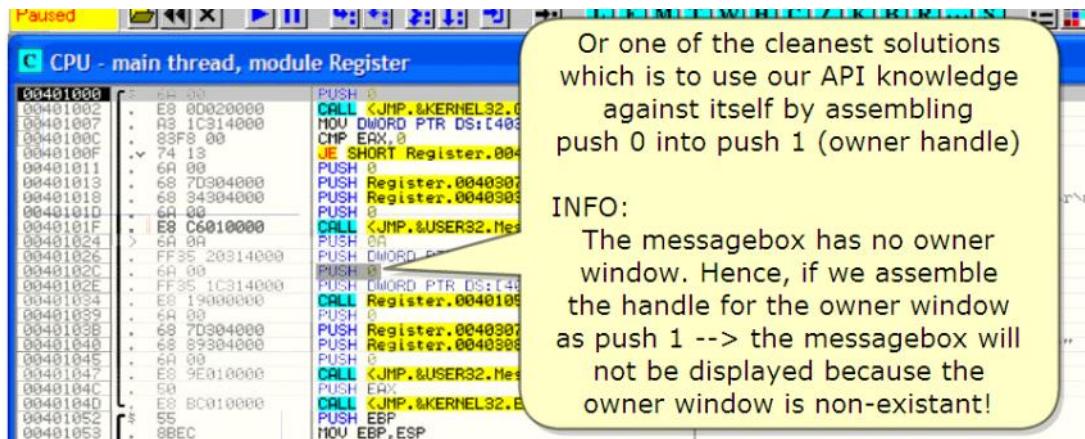
...so that we always jump over the nag, no matter what

But we also could NOP the messagebox (see further)

예 : JE를 JMP로 assemble 할 수 있다.

그래서 우리는 nag을 항상 jump 할 수 있다.

그러나 우리는 또한 messagebox를 NOP 할 수 있다. (나중에 보자)



Or one of the cleanest solutions which is to use our API knowledge against itself by assembling push 0 into push 1 (owner handle)

또한 제일 깨끗한 해결책은 API 지식을 사용하는 것이다. PUSH 0을 PUSH 1로 바꾼다.

#### INFO:

The messagebox has no owner window. Hence, if we assemble the handle for the owner window as push 1 --> the messagebox will not be displayed because the owner window is non-existent!

MessageBox는 owner window가 없다. 그리하여, 우리가 own window의 handle을 PUSH 1로 assemble 하면 --> messagebox는 나타나지 않는다. 왜냐하면 owner window는 존재하지 않기 때문이다.

However, I want to show another solution for this first nag screen, another "elegant one" ;) --> changing the entry point to 00401024 will skip the nag too.

그러나, 우리는 이번 nag screen을 위해 다른 해결 방법을 원한다. 다른 "지능적인" --> entry point를 00401024로 바꿔 nag을 skip 할 것이다.

We can do that because the code before the nag is not important. I trust you still remember from Part 1 what the EP is

nag은 중요한 code가 아니기 때문에 할 수 있다. 나는 네가 아직까지 Part1에서 배운 EP가 무엇인지 기억하고 있다는 것을 믿는다.

Though first, I need to explain a little more about the Portable Executable and the PE header

먼저, 나는 좀 더 설명하기 위해 Portable Executable과 PE header가 필요하다.

Fasten your seat belts ...

and let's go !!!

빨리 너의 안전 벨트를 매!

그리고 가자!!!

#### INFO :

the theoretical display that follows, is not strictly necessary yet. Here, you can do easily without. But a cook... wants to know his ingredients, and thus ...

이론을 따라가자면, 아직 절대적으로 필요하지 않다. 너는 쉽게 할 수 있다. 그러나, 만들어봐.

그의 구성요소를 알기 원한다면

These next pages will become necessary later anyway, especially when dealing with protected files. But I promise that I won't go too deep in detail yet :)

다음 page 에서는 필요하게 될 것이다. 나중에 어디에서든지, 특별히, 보호된 files과 거래할 때. 그러나 내가 약속한다. 아직 깊게 파고들 필요없다.

## 5. The PE File Structure

INFO:

The Portable Executable (PE) format is a file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems.

PE format은 32bit와 64bit Windows 운영체제에서 실행 가능하고 object code, DLL file format 이다.

The term "Portable" refers to the format's portability across all 32-bit (and by extension 64-bit) Windows operating systems.

"Portable"은 참조한다. 모든 32bit Windows OS에서 이식성이 있다. (그리고 64bit로 확장된다)

The PE format is basically a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code.

PE format은 기본적인 data 구조다. 그것은 Windows OS loader가 실행 가능한 code를 관리하기 위해 요약된 정보가 필요하다.

This includes dynamic library references for linking, API export and import tables, resource management data and TLS data.

The format was designed by Microsoft and then in 1993 standardized.

이것은 연결하기 위해 API export와 import table, resource, management data와 TLS data의 동적 library 참조를 포함한다.

이 포맷은 Microsoft에 의해 1993년 표준으로 디자인 됐다.

INFO:

The term "Portable Executable" was chosen because the intent was to have a common file format for all flavors of Windows, on all supported CPUs.

용어로 "Portable Executable" 선택된 이유는 공통적인 file format 이기 때문이다. 모든 종류의 Windows에서 지원하고 모든 CPU에서 지원하기 때문이다.

To a large extent, this goal has been achieved with the same format used on Windows NT and descendants, Windows 95 and descendants, and Windows CE.

매우 큰 규모다, 이 목적은 Windows NT와 밑의 버전들, Windows 95와 밑의 버전들, 그리고 Window CE에서 같은 format으로 사용되고 있다.

INFO :

A very handy aspect of PE files is that the Data Structures on disk are the same data structures used in memory.

매우 편리한 PE file의 측면은 disk상의 data 구조가 똑같은 구조로 메모리에서도 사용된다.

Loading an executable into memory is primarily a matter of mapping certain ranges of a PE

file into the address space.

Thus, a data structure is identical on disk and in memory.

사용 가능하게 메모리로 loading 하는 것은 주로 Address space로 mapping 하는 것과 같이 명확한 PE file의 범위이다.

그리하여, data 구조는 disk와 memory에서 동일하다.

The key point is that if you know how to find something in a PE file, you can almost certainly find the same information after the file is loaded in memory.

Key point는 너도 알다시피 어떻게 PE file에서 찾느냐는 것이다. 너는 file이 memory에 load 된 후에 거의 비슷한 정보를 찾을 수 있다.

It's important to note that PE files are not just mapped into memory as a single memory-mapped file.

이것은 중요하다. 주목할 것은 PE file이 single memory-mapped file일 때 메모리로 mapped 되지 않는다.

Instead, the Win32 loader looks at the PE file and decides what portions of the file to map in.

대신에, Win32 loader는 PE file을 보고 어느 부분으로 file이 map될지 결정한다.

This mapping is consistent in that higher offsets in the file correspond to higher memory addresses when mapped into memory.

이 mapping은 높은 file offset에서 memory의 높은 memory address로 mapped 될 때 일치한다.

The offset of an item in the disk file may differ from its offset once loaded into memory.

Disk에서 item의 offset은 memory로 loaded 될 때의 offset과 다르다.

However, all the information is present to allow you to make the translation from disk offset to memory offset.

그러나 모든 정보는 disk에서 offset에서 memory offset으로 변경될 수 있다.

번역 주)offset은 시작된 위치에서 상대적으로 떨어진 위치를 뜻한다. 해석하는데 더 헛갈려.

Reversecore 참고 하시라.

INFO :

A module in memory represents all the code, data, and resources from an executable file that is needed by a process.

Memory에서 module은 executable file에서 code, data, resource를 대표한다. PE는 process에 의해 필요하다.

Other parts of a PE file may be read, but not mapped in (for instance, relocations).

PE file의 다른 부분은 읽히지만 mapped 되지 않을 수 있다. (대신에, 재배치).

Some parts may not be mapped in at all, for example, when debug information is placed at the end of the file.

몇 부분에서 예를 들어, debug 정보가 file의 끝에 놓일 경우 몇몇 부분에서 모든 것이 mapped 되지 않는다.

A field in the PE header tells the system how much memory needs to be set aside for mapping the executable into memory.

PE header의 field는 system에게 얼마나 많은 executable의 memory가 memory에 mapping 되기 위해 필요한지 말한다.

Data that won't be mapped in, is placed at the end of the file, past any parts that will be

mapped in.

Data는 mapped 되지 않는 것이 file의 끝에 놓여진다. 지나온 모든 part는 mapped 될 것이다.

The PE data structures are:

PE data 구조는

DOS header

DOS stub

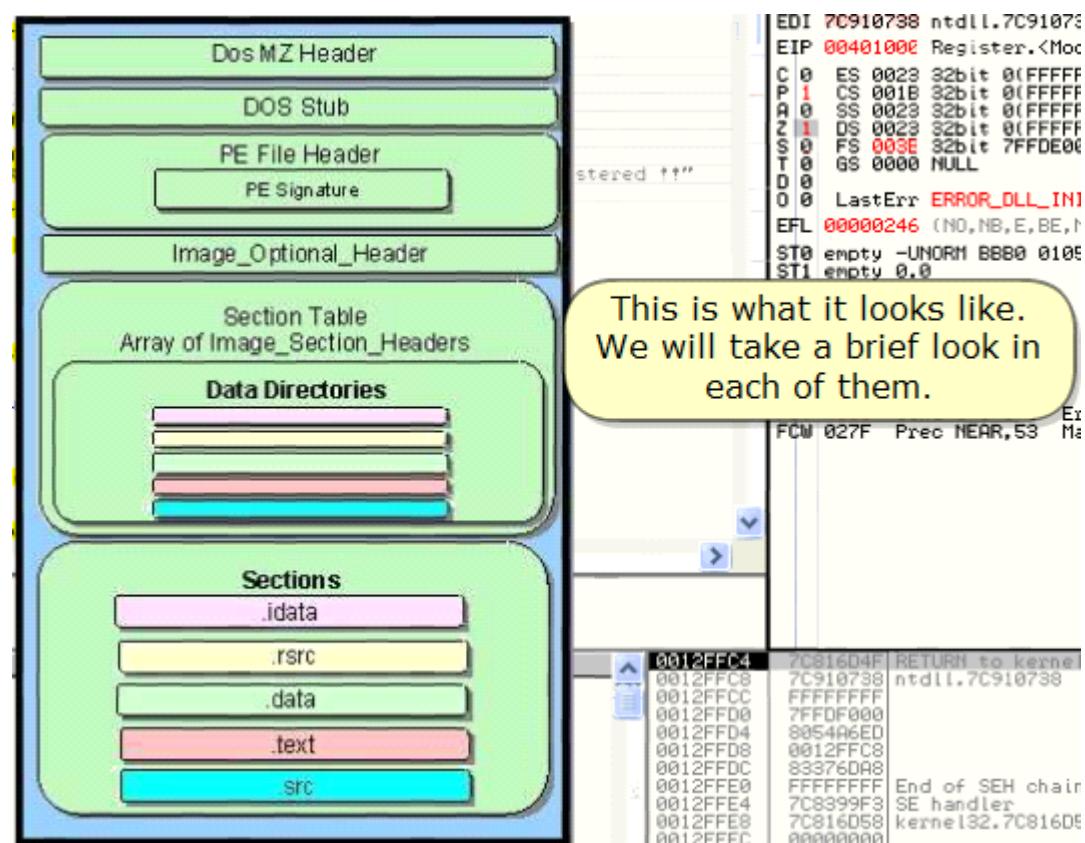
PE File Header

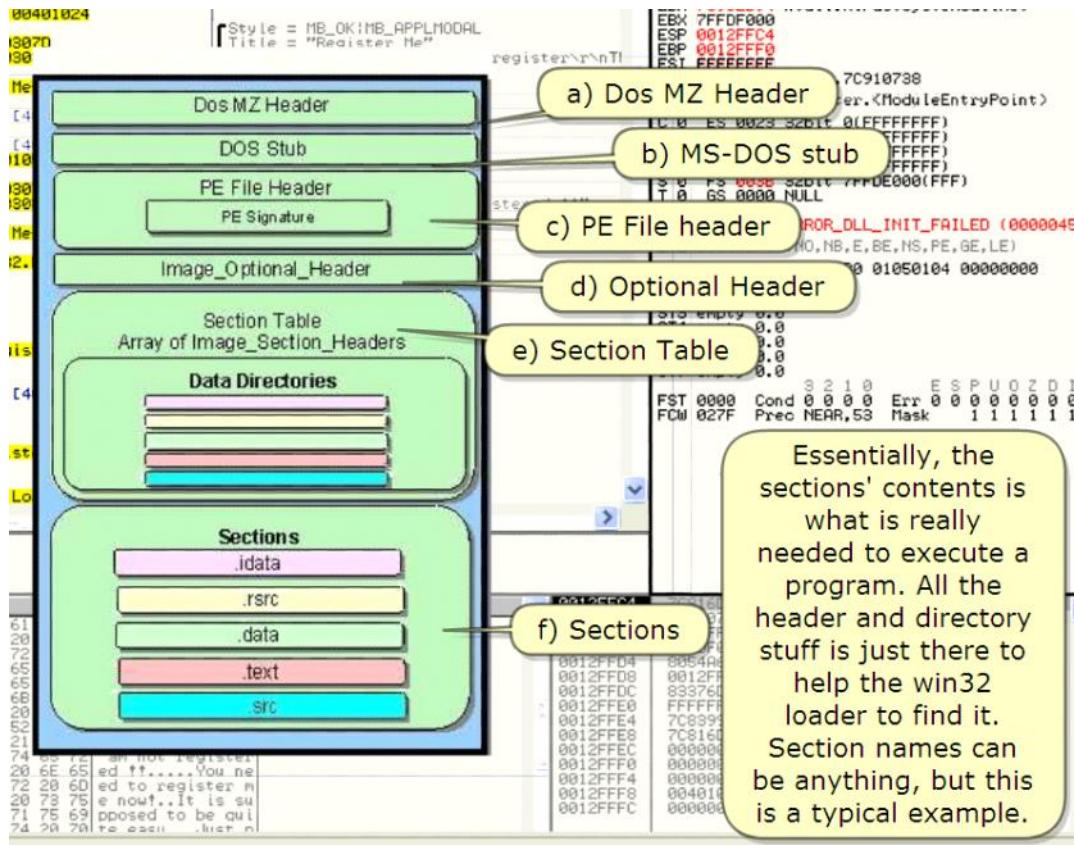
Image Optional Header

Section Table

Data Directories

Sections





This is what it looks like.

We will take a brief look in each of them.

보는 바와 같다.

우리는 간단히 각각에 대해 알아보겠다.

- a) Dos MZ Header
- b) MS-DOS stub
- c) PE File Header
- d) Optional Header
- e) Section Table
- f) Sections

Essentially, the sections' contents is what is really needed to execute a program.

All the header and directory stuff is just there to help the win32 loader to find it.

특별히, section's content는 program을 실행하는데 정말로 필요하다.

모든 header와 directory stuff는 그것을 찾고 Win32 loader를 돋기 위해 존재한다.

Section names can be anything, but this is a typical example.

Section name은 어떤 것이라도 괜찮다. 그러나 이것은 일반적으로 예처럼 한다.

Let's take a look.

The PE header is (normally) located at [imagebase] till [imagebase + 1000].

In our case, this is 400000 till 401000.

Remember that the imagebase from this RegisterMe was 400000.

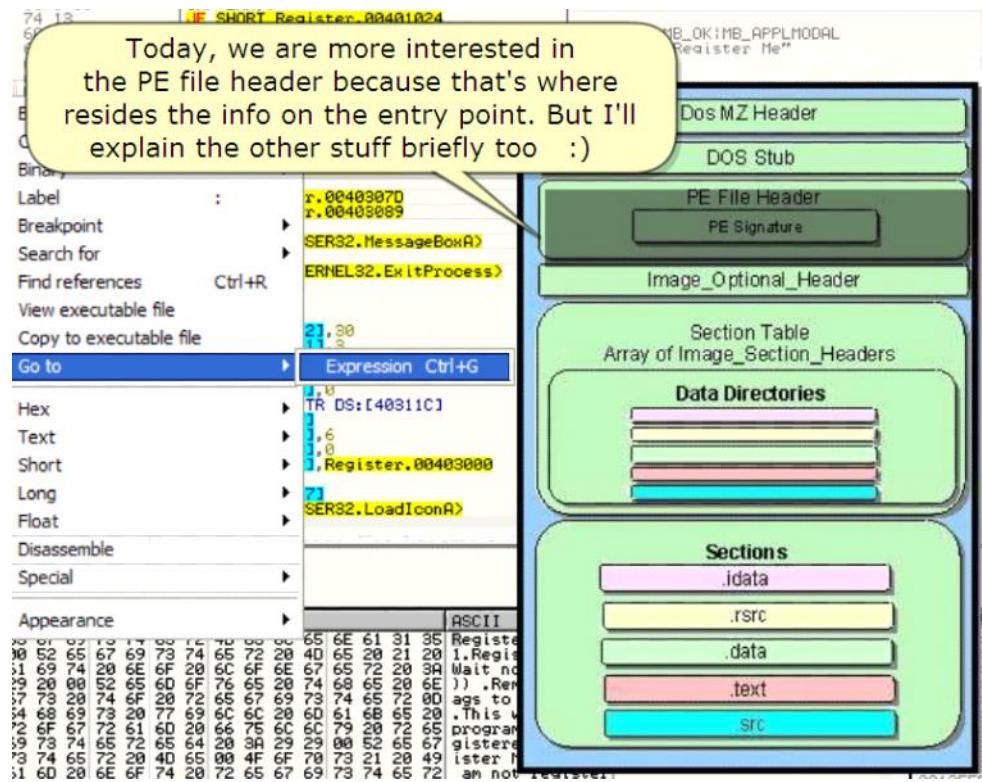
봐라.

PE header는 [ImageBase]에서 [ImageBase + 1000]까지 위치해 있다. 우리의 경우에는 400000 ~ 401000 이다.

기억해 이 RegisterMe에서 ImageBase는 4000000 이다.

the PE header is formed by all this

PE header는 모든 것에 의해 형성됐다.



Today, we are more interested in the PE file header because that's where resides the info on the entry point. But I'll explain the other stuff briefly too :)

오늘, 많은 흥미로운 것이 PE 파일에 있다. 왜냐하면 그것은 entry point에 존재한다. 그러나 나는 간단히 다른 재료를 설명할 것이다.

let's see the header

Header를 보자.

C CPU - main thread, module Register

```

00401000 E8 00020000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007 A3 C314000 HOU DWORD PTR DS:[40811C],ERX
0040100C 83F8 00 CMP EAX, 0
0040100F 74 13 JE SHORT Register.00401024
00401011 6A 00 PUSH 0
00401013 68 7D304000 PUSH Register.0040397D
00401018 68 34304000 PUSH Register.00403934
0040101D 6A 00 PUSH 0
0040101F E8 C6010000 CALL <JMP.&USER32.MessageBoxA>
00401024 6A 00 PUSH 0
00401026 FF35 20314000 PUSH DWORD PTR DS:[408120]
0040102C 6A 00 PUSH 0
0040102E FF35 1C314000 PUSH DWORD PTR DS:[40811C]
00401034 E8 19000000 CALL Register.00401052
00401039 6A 00 PUSH 0
0040103B 68 7D304000 PUSH Register.0040397D
00401048 68 89304000 PUSH Register.00403989
00401045 6A 00 PUSH 0
00401047 E8 9E010000 CALL <JMP.&USER32.MessageBoxA>
0040104C 50 PUSH ERX
0040104D E8 BC010000 CALL <JMP.&KERNEL32.ExitProcess>
00401052 55 PUSH EBP
00401053 8BE8 MOV EBP,ESP
00401055 83C4 B0 ADD ESP,-50
00401058 C745 D0 30000000 MOV ELOCAL.121,30
0040105F C745 D4 03000000 MOV ELOCAL.111,3
00401066 C745 D8 37114800 MOV ELOCAL.101,Register.00401137
0040106D C745 DC 00000000 MOV ELOCAL.91,0
00401074 C745 E0 00000000 MOV ELOCAL.81,0
00401078 FF35 1C314000 PUSH DWORD PTR DS:[40811C]
00401081 8F45 E4 POP ELOCAL.71
00401084 C745 F0 06000000 MOV ELOCAL.41,6
00401088 C745 F4 00000000 MOV ELOCAL.31,0
00401092 C745 F8 00004000 MOV ELOCAL.21,Register.00403000
00401099 6A 01 PUSH 1
0040109B FF75 E4 PUSH I
0040109E E8 41010000 CALL ...

```

... and we get a view on the beginning of the PE header ...

Registers <ModuleEntryPoint>	
Address	Hex dump
00400000	4D 5A 90 00 00 00 00 00 00 00 00 00 00 00 00 00
00400010	B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400048	00 1F 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400053	69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6F
00400058	is program cannot be run in DOS mode.
00400060	74 20 62 65 20 72 75 6E 20 59 6E 20 44 4F 53 20
00400068	60 6F 65 2E 00 00 00 24 00 00 00 00 00 00 00 00
00400078	E3 E2 11 00 R7 83 7F 88 R7 83 7F 88 00 00 00 00
00400088	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400098	R7 83 7F 88 00 00 7F 88 5B 03 60 88 R6 88 00 00
004000A8	68 85 79 28 A6 00 00 00 00 00 00 00 00 00 00 00
004000B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000C8	5B 45 00 00 4C 01 04 00 1E 29 01 99 00 00 00 00
004000D8	PE...L0..A1 00 00 00 00 00 00 00 00 00 00 00 00 00
004000E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

...and we get a view on the beginning of the PE header ...

그리고 우리는 PE header의 시작점을 얻었다.

Registers <ModuleEntryPoint>	
Address	Hex dump
00400000	4D 5A 90 00 00 00 00 00 00 00 00 00 00 00 00 00
00400010	B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400048	00 1F 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400053	69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6F
00400058	is program cannot be run in DOS mode.
00400060	74 20 62 65 20 72 75 6E 20 59 6E 20 44 4F 53 20
00400068	60 6F 65 2E 00 00 00 24 00 00 00 00 00 00 00 00
00400078	E3 E2 11 00 R7 83 7F 88 R7 83 7F 88 00 00 00 00
00400088	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00400098	R7 83 7F 88 00 00 7F 88 5B 03 60 88 R6 88 00 00
004000A8	68 85 79 28 A6 00 00 00 00 00 00 00 00 00 00 00
004000B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004000C8	5B 45 00 00 4C 01 04 00 1E 29 01 99 00 00 00 00
004000D8	PE...L0..A1 00 00 00 00 00 00 00 00 00 00 00 00 00
004000E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### a) Dos MZ Header

INFO : This makes a PE file an MS-DOS executable. The first 2 bytes are always : 4D 5A -->

"MZ" is the dos exe signature.

PE file은 MS-DOS executable에서 만들었다. 처음 2 bytes는 항상 : 4D 5A --> "MZ" 는 dos exe signature(특징)다.

번역 주)Mark Zbikowski의 약자다. 1951년 생으로 Detroit 생이다. Microsoft의 유명한 Architect 그리고 일찍이 computer hacker 였다. 그는 몇 년간 회사에서 일을 시작했다. MS-DOS, OS/2, Cairo and Windows NT를 이끄는 노력을 했다. 2006년 그는 25년간 service를 회사에 명예롭게 했다. Bill Gates and Steve Ballmer 다음으로 3번째의 위치까지 도달했다. 그는 현재 technical advisor를 여러 회사와 University of Washington에서 강사들에게 한다. 그는 DOS executable file format을 design 했다. 그것은 MS-DOS executable files로 사용됐다. 그리고 file format headers 들은 그의 initial로 시작한다. : magic number는 ASCII character로 "MZ" (0x4D, 0x5A) 다.

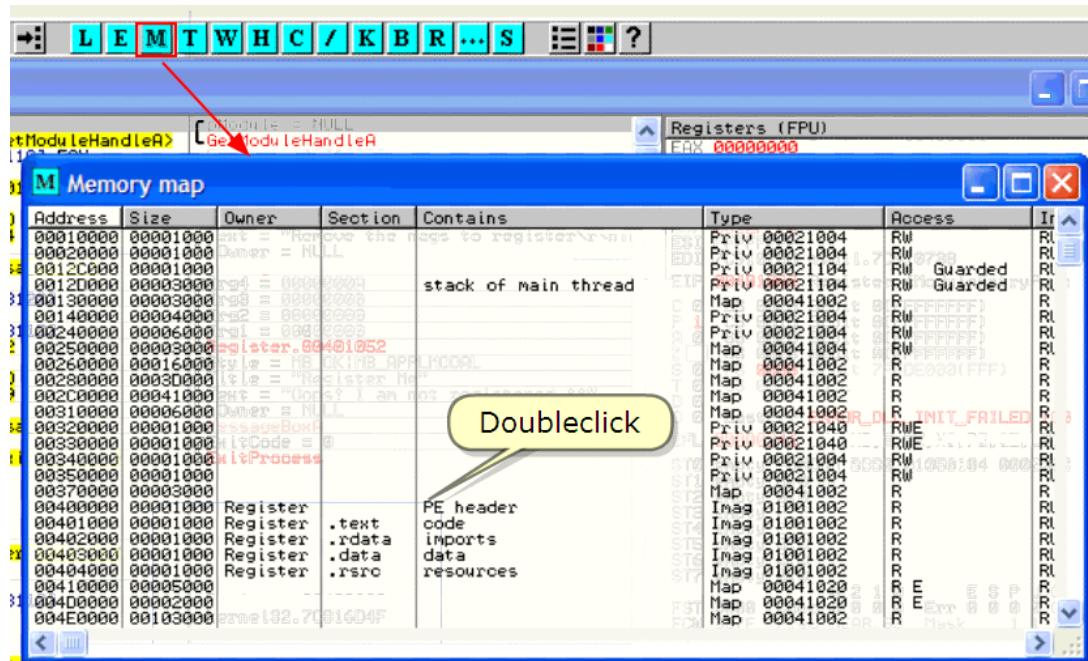
After that comes a typical sequence of Last Page Size,, Toaal Pages in File, Relocation Items,

etc. The last dword is the PE file header Pointer. We will use this Pointer to find the PE file header.

일반적인 Last Page Size의 순서가 온 후에, File의 Total pages, 재배치 Items, etc. 마지막 dword는 PE file header Pointer다. 우리는 이 Pointer를 PE file header를 찾는데 사용할 것이다.

But let's take a look at the header in the memory modules window, it will be a lot clearer because Olly is really helpfull here...

그러나 우리는 memory module windows의 header를 보자. 그것은 명확하게 해줄 것이다. 왜냐하면 Olly가 정말로 이곳에서 도와준다.

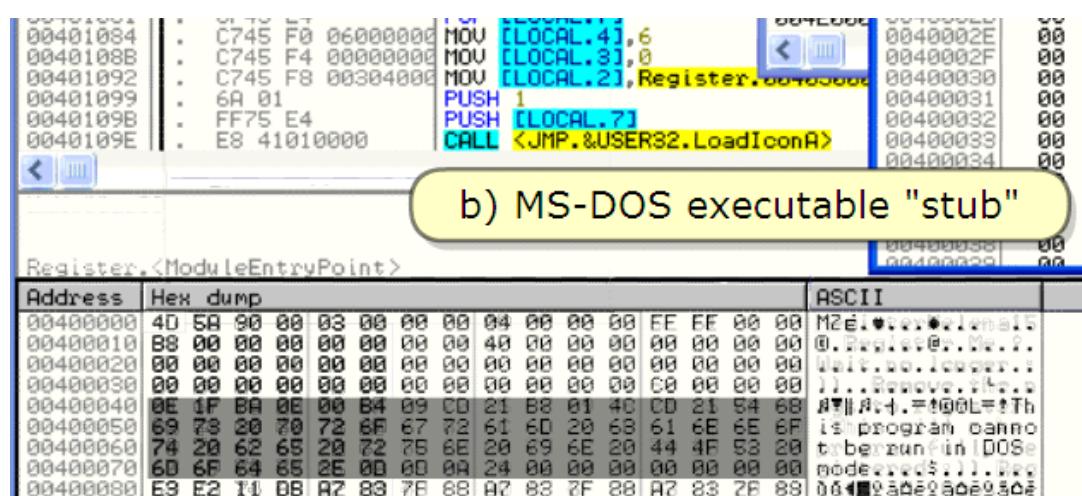


Doubleclick

Quite a bit clearer huh? But not important for us, let's continue.

더블클릭

꽤 명확해졌지? 그러나 이것은 우리에게 중요한 게 아니다. 계속 해보자.



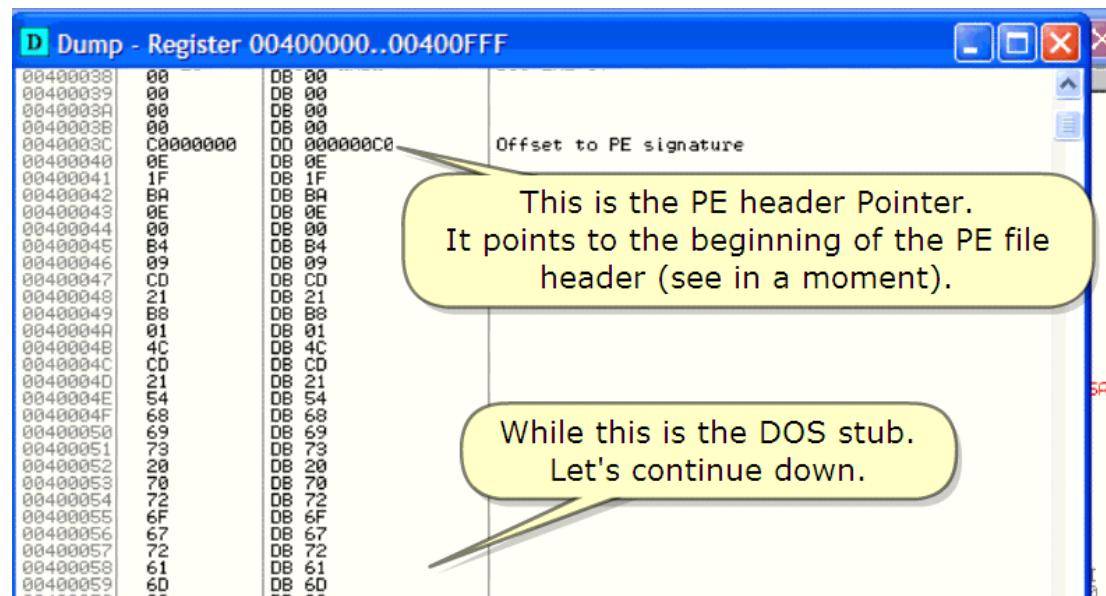
b) MS-DOS executable "stub"

The DOS stub is actually a valid EXE for PE-files, it is a MS-DOS 2.0 compatible executable

that almost always consists of a small number of bytes that output an error message.

DOS stub은 PE-files을 위해 실제로 exe가 맞는지 검증한다. 그것은 MS-DOS 2.0 compatible executable 이다. 그것은 거의 항상 적은 bytes로 되어있다. 그것은 error를 내보낸다.

It can simply display a string like "This program requires Windows" or "Cannot be run in DOS mode" or similar. In a Win32 system the PE loader just skips such a MS-DOS Stub 이것은 간단히 "This program은 Windows에서 동작한다." 나 "DOS mode에서 실행할 수 없다" 나 비슷하게 보여준다. Win32 system에서 PE loader는 MS-DOS Stub을 skip 한다.



This is the PE header Pointer.

이것은 PE header Pointer다.

It points to the beginning of the PE file header (see in a moment).

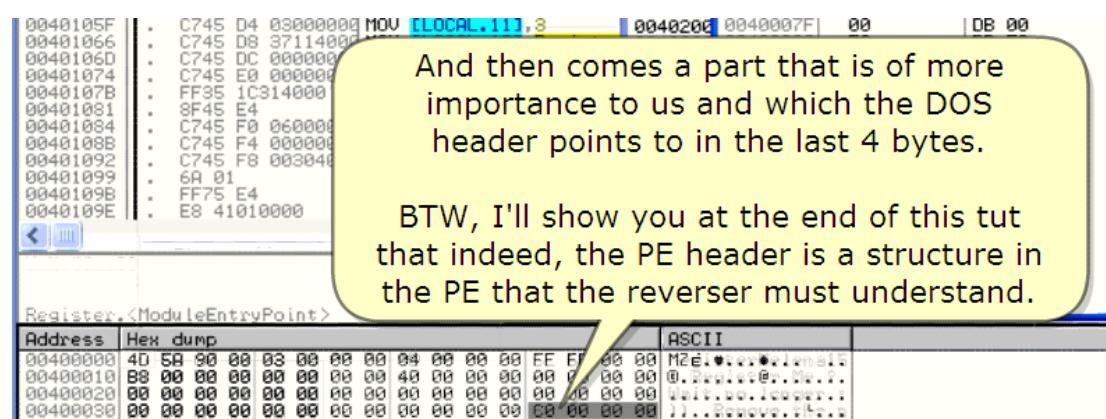
While this is the DOS stub.

Let's continue down.

이것은 PE file header의 처음을 가리킨다. (곧 바로 보자)

This is Dos stub다.

내려봐.



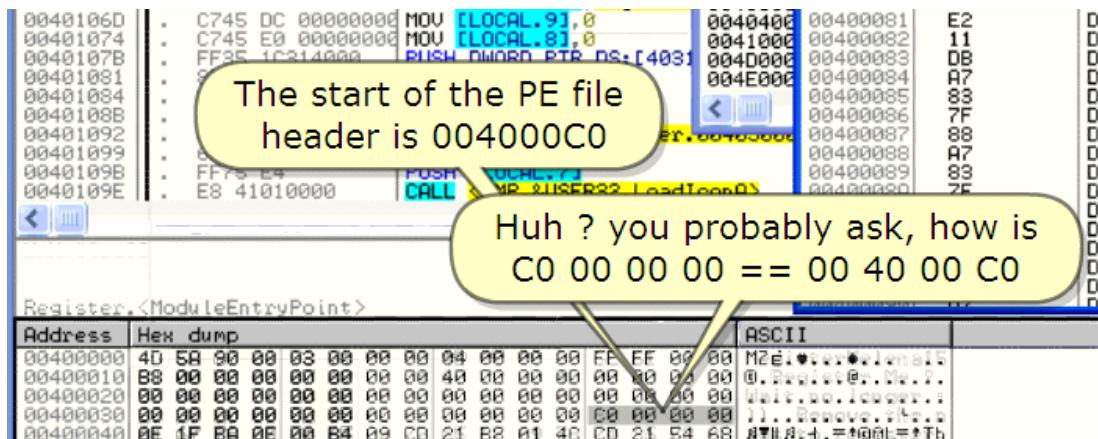
And then comes a part that is of more importance to us and which the DOS header points to in the last 4 bytes.

이 부분이 우리에게 매우 중요하다. 그리고 DOS header의 마지막 4bytes를 가리킨다.

BTW, I'll show you at the end of this tut that indeed, the PE header is a structure in the PE

that the reverser must understand.

너에게 tutorial의 마지막에 보여줄 것이다. PE에서 PE header는 reverser가 꼭 이해해야 할 구조다.



The start of the PE file header is 0040000C0

PE file header의 시작은 0040000C0 다.

huh? you probably ask, how is C0 00 00 00 == 00 40 00 C0

너는 물을 것이다, 어떻게 C0 00 00 00 == 00 40 00 C0 되는지.

Well, first of all, there is the endians. This means that you need to read the opcodes (per byte) in reverse order.

좋아, 우선, endian이 있습니다. 이것은 반대순서로 opcodes를(byte 씩) 읽을 필요가 있다는 것을 의미한다.

--> 00 00 00 C0 becomes C0 00 00 00

**번역 주)** little endian, big endian

Second : remember that the data in the header is info for the windows loader.

두번째 : 기억해라. Header 속의 data는 windows loader를 위한 정보다.

The windows loader automatically adds the ImageBase (here 400000) to the offset that it finds in the header and maps this into memory at that location (here at 400000)

Windows loader는 자동으로 ImageBase(여기는 400000)을 offset에 추가한다. offset을 header에서 찾고 그 위치에서(여기는 400000) memory로 map 한다.

--> C0 00 00 00 in the dump window

becomes "VA" 00 40 00 C0

Dump window 속에서 C0 00 00 00 이면

VA는 00 40 00 C0 이 된다.

INFO : VA and RVA addresses : The PE format makes heavy use of so-called RVAs.

가상주소와 상대적인 가상주소 : PE format은 많은 RVA라 불리우는 것을 사용한다.

An RVA, aka "relative virtual address", is used to describe a memory address if you don't know the image base address.

만약에 네가 ImageBase address를 모를 때 RVA는 "상대적인 가상 주소", memory address를 설명할 때 쓰인다.

It is the value you need to add to the image base address to get the actual linear address.

이것은 값이다. 네가 ImageBase address에 추가할 때 실제로 선형적인 address가 필요하다.

The base address is the address the PE image is loaded to in RAM, and may vary from one invocation to the next.

Base address는 address다. PE image는 RAM에 load 됐다. 그리고 한 번의 기도로 다음까지 서로 다르다.

번역 주)base address는 address다. 뭔가 이상함..

Example: suppose an executable file is loaded to address 400000 and program execution starts at RVA 4000.

예: executable file은 400000 address와 program 실행할 때 RVA 4000으로 load 됐다.

The effective execution starts at RVA 4000. The effective execution start (EP) will then be at the address 404000.

효과적인 execution은 RVA 4000 에 시작되는 것이다. 효과적인 execution은 address 404000에서 시작되는 것이다.(EP)

If the executable were loaded to 1000000, the execution start address (aka entry point or EP) would then become 01004000.

만약에 executable이 1000000 에 load 됐다면, execution start address(entry point)는 01004000 이 되었을 것이다.

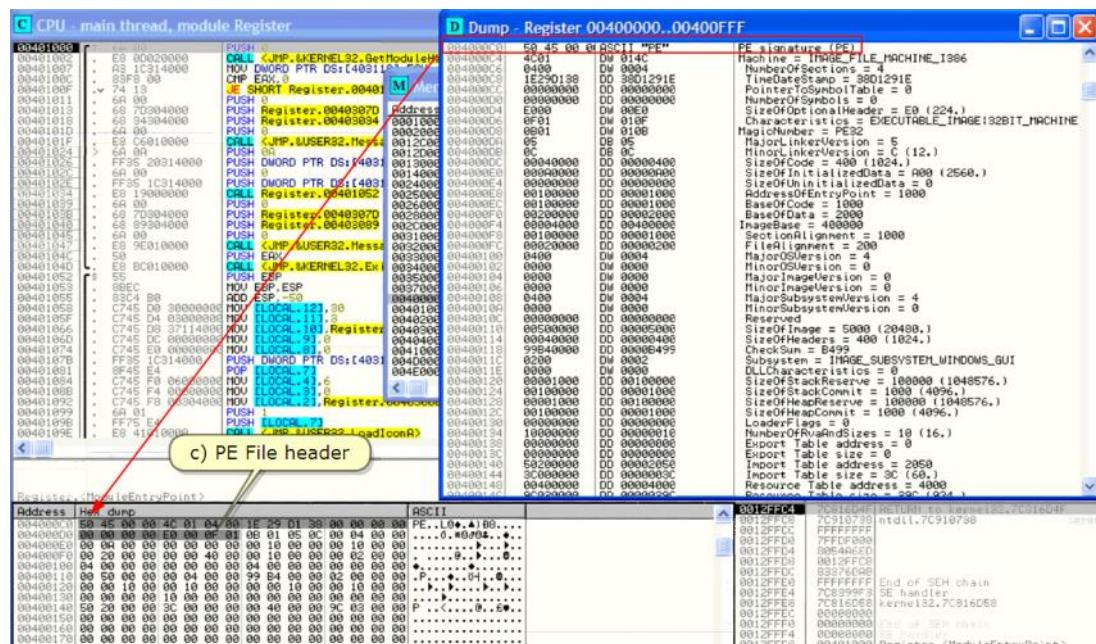
INFO:

It is crucial to understand that a RVA is a value relative to the ImageBase.

이것은 이해하는데 중대하다. RVA는 ImageBase와 상대적이다.

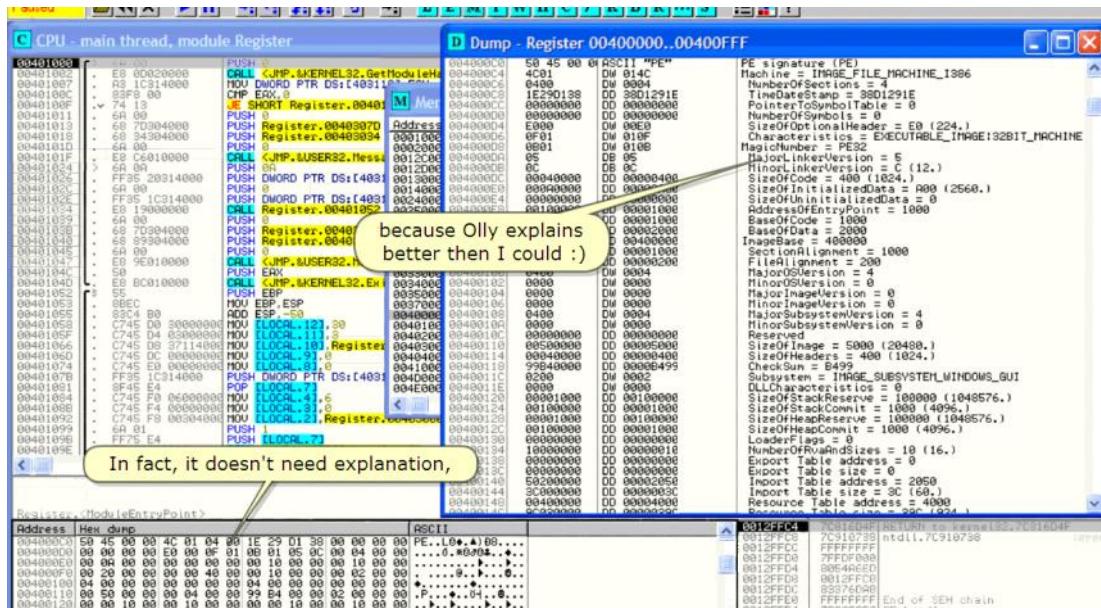
A VA is the address in memory (when run) whilst an offset (see more later) is the location in the file on disk

Disk의 file이 상대적으로(나중에 더 보겠다) 위치해 있는 동안 VA는 memory의 address다.



So, let's go to VA 004000C0 and scroll the dump window too to have a look at the ...

그래서 VA 004000C0 로 가자 그리고 dump window를 이동시켜 봐.

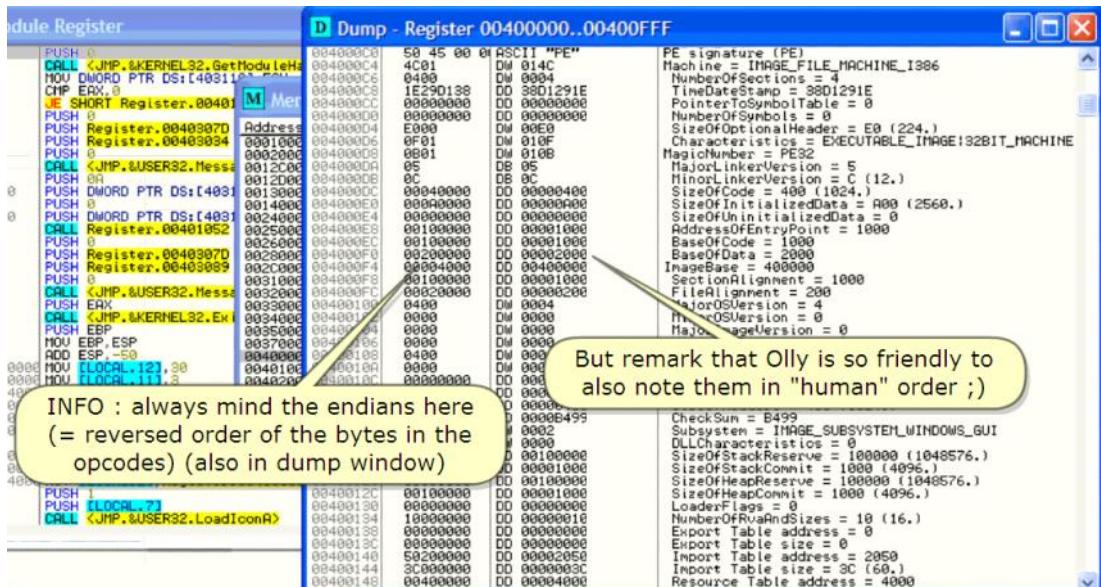


### c) PE File Header

In fact, it doesn't need explanation,  
because Olly explains better than I could :)

사실, 더 이상 설명은 필요없다,

왜냐하면 Olly가 내가 하는 것보다 설명을 더 잘하기 때문에.

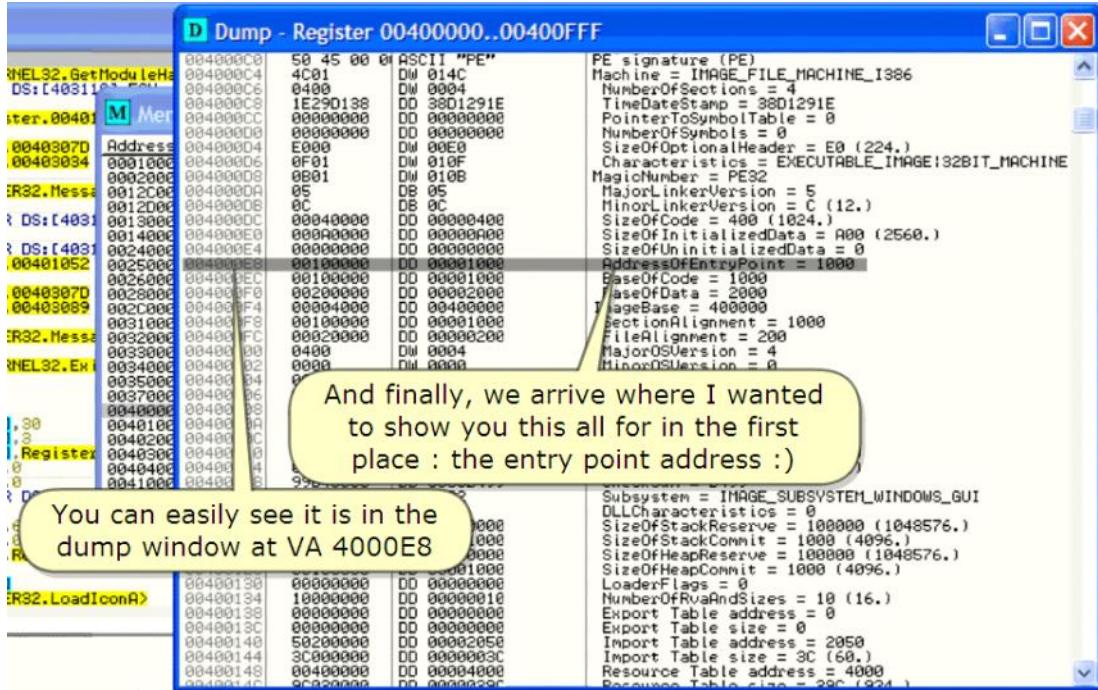


INFO : always mind the endians here (= reversed order of the bytes in the opcodes) (also in dump window)

항상 이곳은 endian이라는 것을 기억하자.(= opcode들은 byte들의 반대 순서다.) (또한 이것은 dump window에 있다)

But remark that Olly is so friendly to also note them in "human" order :)

주목해라. 이 *v*는 치적하게 이가의 명령으로 전여놓는다.

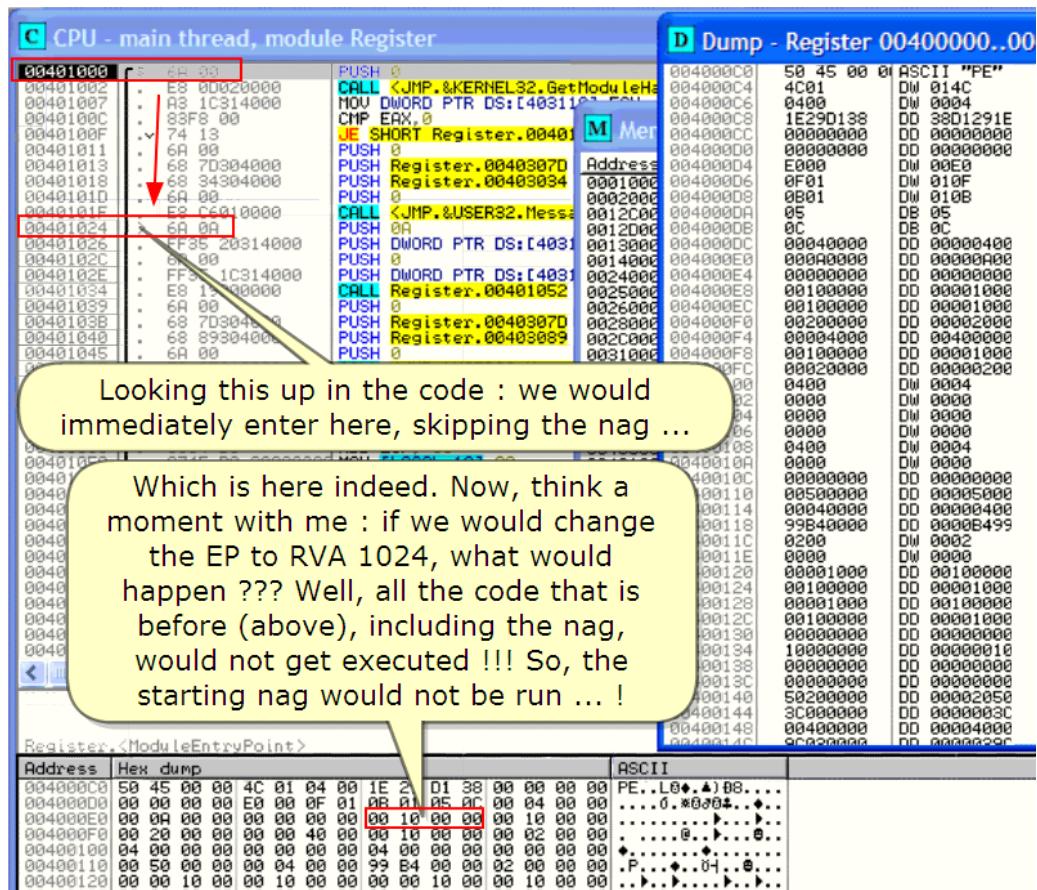


And finally, we arrive where I wanted to show you this all for in the first place : the entry point address :)

마지막으로, 우리는 첫번째 장소에서 이것을 너에게 보여주기 위해 도착한다. : the entry point address :)

You can easily see it is in the dump window at VA 4000E8

Dump window의 VA 4000E8에서 쉽게 볼 수 있다.



Which is here indeed. Now, think a moment with me : if we would change the EP to RVA 1024, what would happen ???

이곳이다. 이제, 나와 같이 잠시 생각해 보자. : 우리가 만약에 EP를 바꾸기 위해 RVA를 1024로

바꾸면 무슨 일이 일어날까 ???

Well, all the code that is before(above), including the nag, would not get executed !!! So, the starting nag would not be run ... !

좋아, nag를 포함하는 code는 아직 실행되지 않았다. 그래서 starting nag을 실행하지 않을 것이다.

Looking this up in the code : we would immediately enter here, skipping the nag ...

Let's do that, but first, let's finish this tour.

Code를 보자. : 우리는 즉시 이곳으로 들어갈 수 있고 nag을 skip 한다.

자, 해보자, 그러나 이 여행은 끝내고

End of the Optional Header, immediately followed by the ...

Optional Header의 끝이다, 즉시 Section Table이 따라왔다.

#### e) Section Table

Between the PE headers and the raw data for the image's Sections lies the Section Table.

PE headers와 Image's Section에 대한 raw data 사이에 Section Table이 있다.

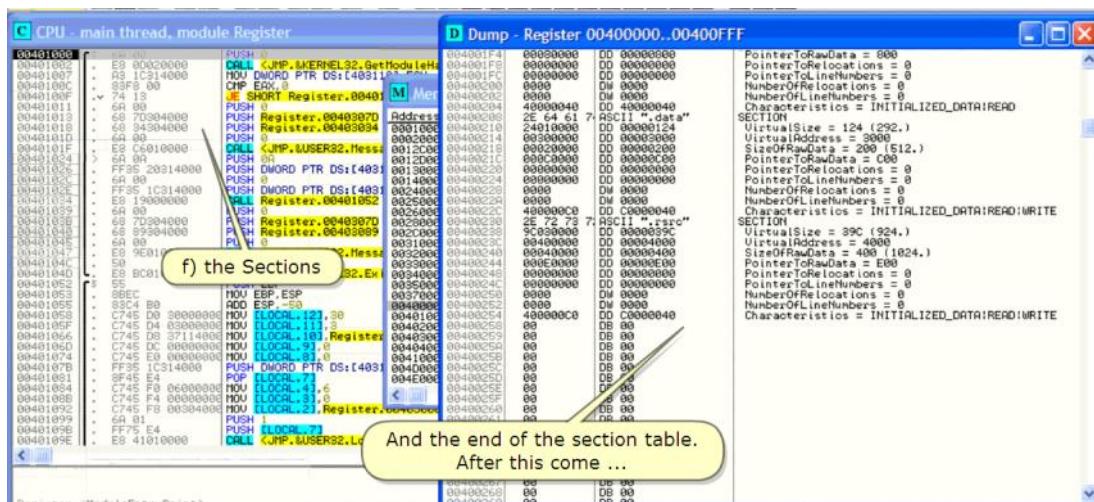
There is one section header for each section, and each data directory will point to one of the sections. Several data directories may point to the same section, and there may be sections without a data directory pointing to them.

각 section을 위해 하나의 section header가 있다, 그리고 각 data directory는 sections 중 하나를 가리킨다. 여러가지 data directories은 똑같은 section을 가리킬 수 있다. 그리고 data directory가 section을 가리키는 것이 없을 때 sections이 있을 수도 있다.

The section in the image are sorted by their starting address (RVAs), rather than alphabetically.

Image안의 section은 그들의 starting address(RVAs)에 의해 ABC 순으로 정렬 됐다.

번역 주) 이 PE쪽은 이론적으로 부족해서인지 나조차도 어렵게 느껴진다. 계속 반복해서 보는 수 밖에 없다. 제 번역이 시원찮다면 한글로 된 다른 강의를 많이 보시라.



And the end of the section table.

After this come ...

Section table의 끝이다.

이제 다가오는 것은 Sections 이다.

#### f) the Sections

.... usually starting at RVA 1000, we have the sections, normally the code (text) section first.

보통 starting으로 RVA 1000이다. 우리는 sections을 가지고 있다. 보통 code(text) section이 먼저 온다.

INFO :

Sections have two alignment values, one within the file "on this" (Pointer to Raw Data) and the other in memory (Virtual Address).

Sections은 2가지 정렬 값을 갖는다. 하나는 file에 있는(Pointer to Raw Data)이고 다른 하나는 memory에 있다.(Virtual Address)

번역 주: 어렵게 생각하지 말자. 뭘 그대로 Raw Data를 가리키는 것이다. 즉 주소를 뜻한다.

The PE file header specifies both of these values, which can differ.

PE file header는 분명한 양쪽의 값을 가지고 있다. 다를 수 있다.

Each section starts at an offset that's some multiple of the alignment value.

각 section은 offset에서 시작한다. 약간의 다양한 alignment value가 있다.

For instance, in the PE file, a typical alignment would be 200.

이 사례는, PE file에서는 일반적으로 200을 alignment로 잡는다.

Thus, every section begins at a file offset that's a multiple of 200. Once mapped into memory, sections always start on at least a page boundary.

File offset에서 매번 section 시작할 때마다 200씩 중복 된다. 한번 Memory로 map 될 때, Section은 항상 최소한 한 page의 경계로 시작한다.

That is, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page.

PE section이 memory로 map 될 때, 각 section의 첫번째 byte는 memory page와 일치한다.

INFO :

The Section Table is an array of IMAGE\_NUMBER\_OF\_DIRECTORY ENTRIES (16 spaces reserved for entries) IMAGE\_DATA\_DIRECTORYs.

Section Table은 IMAGE\_NUBER\_OF\_DIRECTORY ENTRIES IMAGE\_DATA\_DIRECTORYs의 배열이다.(16 space는 바뀐다)

Each of these directories describes the location and size of a particular piece of information, which is located in one of the sections that follow the directory entries.

각 directories들은 location과 위치와 특정 부분의 size를 설명한다. Directory entries를 따르는 sections 중 하나에 위치해 있습니다.

This array allows the loader to quickly find a particular section of the image 이 배열은 빠르게 image의 특정 부분을 찾을 수 있도록 loader에게 허락된다.

INFO :

When you use code or data from a DLL, you're importing it.

When any PE file loads, one of the jobs of the Win32 loader is to locate all the imported functions and data and make those addresses available to the file being loaded.

네가 DLL에서 code나 data를 사용할 때 이것은 중요하다.

어떤 PE file을 load 할 때, Win32 loader는 불러온 function에 위치하고 data와 그것들의 address를 File에 처음 load 되었을 때 사용할 수 있게 한다.

When you link directly against the code and data of another DLL, you're implicitly linking against the DLL.

다시 네가 직접적으로 반대되는 code와 다른 DLL의 data에 연결할 때, 암암리에 반대되는 DLL에 연결한다.

You don't have to do anything to make the addresses of the imported APIs available to your code.

너는 불러온 code에서 API가 사용할 수 있게 만들기 위해 할 수 있는 게 없다.

Within a PE file, there's an array of data structures, one per imported DLL.

PE file에서, 불러온 DLL에 data 구조 배열이 하나씩 있다.

Each of these structures gives the name of the imported DLL and points to an array of function pointers.

각 구조들은 불러온 DLL 이름들과 function pointer의 배열을 가리키는 것을 알려준다.

The array of function pointers is known as the Import Address Table(IAT).

function pointer의 배열은 Import Address Table이 알고 있다.

Each imported API has its own reserved spot in the IAT where the address of the imported function is written by the Win32 loader.

각 불러온 API들은 IAT에서 반대되는 위치를 가지고 있다. IAT는 불러온 function이 Win32 loader에 의해 쓰여져 있다.

This last point is particularly important: once a module is loaded into RAM, the IAT contains the address that is invoked when calling imported APIs.

마지막 point는 특별히 중요하다: 하나의 module이 RAM에 load 될 때, IAT는 불러온 API를 부를 때 언급된 address를 포함한다.

The beauty of the IAT is that there's just one place in a PE file when loaded into RAM where an imported API's address is stored.

아름다운 IAT는 RAM으로 load할 때 PE file의 한 장소에 위치한다. RAM은 불러온 API 주소가 저장되어 있다.

All the calls go through the same function pointer in the IAT.

모든 calls은 IAT에서 같은 function pointer를 통합니다.

INFO :

The important parts of an import table are the imported DLL name and the two arrays of IMAGE\_IMPORT\_BY\_NAME pointers.

중요한 parts의 import table은 불러온 DLL name과 2가지 IMAGE\_IMPORT\_BY\_NAME 배열 pointer다.

In the EXE file, the two arrays (pointed to by the Characteristics and FirstThunk fields) run parallel to each other, and are terminated by a NULL pointer entry at the end of each array.

Exe file에서, 2가지 배열은(지표와 FirstThunk filed에 의해 가리킨다.) 병렬적으로 각각 실행된다. 그리고 NULL pointer에 의해 각 배열의 끝에서 종료된다.

The pointers in both arrays point to an IMAGE\_IMPORT\_BY\_NAME structure. Why are there two parallel arrays of pointers to the IMAGE\_IMPORT\_BY\_NAME structures?

Pointers는 양쪽 배열에서 IMAGE\_IMPORT\_BY\_NAME 구조를 가리킨다. 왜 2가지 병렬 pointers는 IMAGE\_IMPORT\_BYNAME 구조를 가리키나?

The array pointed at by the Characteristics field is left alone, and never modified.

It's sometimes called the hint-name table. The array pointed at by the FirstThunk field is overwritten by the PE loader.

특성 filed에 의해 가리킨 배열은 남게 됐다. 그리고 절대 변경이 안 된다.

가끔 hint-name table을 불렀다. FirstThunk filed에 의해 가리킨 배열은 PE loader에 의해 덮어 씌워졌다.

The loader iterates through each pointer in the array and finds the address of the function that each IMAGE\_IMPORT\_BY\_NAME structure refers to.

Loader는 배열에서 각 pointer를 통하여 반복한다. 그리고 각 IMAGE\_IMPORT\_BY\_NAME 구조가 참조하는 function address를 찾는다.

The loader then overwrites in RAM the pointer with the found function's address.

Loader는 RAM pointer에 찾은 function address를 덮어 씁니다.

Since the array of pointers that are overwritten by the loader eventually holds the addresses of all the imported functions, it's called the Import Address Table(IAT)

Pointers의 배열은 결국 loader에 의해 불러온 function의 주소를 얻기 위해 덮어 써진다. 이것 은 Import Address Table 이라고 불린다.

#### INFO :

All this is an extremely important matter when packers/protectors come in question.

모든 것은 packer와 protector가 있을 경우 극도로 중요하다.

Lately, there is a tendance of virii to (ab) use packers/protectors to obfuscate their presence.

최근에, 그것은 virii를 치료한다. packers/protectors를 사용하고 혼란스럽게 하기 위해 존재한다.

Painful result of this is that reversers will have to deal with those packers/protectors too.

Reverse에게 빠아픈 결과다. Reverse는 packers/protectors를 처리해야 할 것이다.

If you are a beginning reverser, all this is probably extremely confusing.

만약에 네가 처음 시작하는 reverser라면, 아마 극도로 혼란스럽다.

No problem, its ok if you have retained some of the stuff for now. It will all become clearer when we deal with protectors.

문제없다, 현재 상태만 유지해. 우리가 protector와 거래할 경우, 이것은 곧 명확해 질 것이다.

#### INFO :

Anyway, so far the tour in the PE file structures. If something is not clear or if you want some good winters night reading,

어디서든, 지금까지 PE file 구조에 대해 여행했다. 만약에 명확하지 않거나 하면 밤에 시간날때 읽어봐라.

I once more advise you to read Goppit's tutorial "Portable Executable File Format Compendium". See part one for DL-link.

한 가지 더 충고를 하자면, Goppit's tutorial "Portable Executable File Format 개요서"를 읽어 봐. Part를 보면 DL-link가 있다.

But for now, let's continue this RegisterMe and do as we said : change the EP so that we jump the nag already at startup.

그러나 현재, RegisterMe를 계속 하자. EP를 바꾸자. 우리는 시작할 때 이미 nag을 jump 할 수 있다.

번역 주)PE 구조는 lena의 text 설명만으로는 이해하기 힘듭니다. PE Header 부분은 Reversecore를 참조하면 좋습니다.

## 6. Making the patches

Many reversers would immediately grab one of the many tools we have to change the EP.

많은 reverses는 즉시 많은 tool중에 하나를 잡는다. 우리는 EP를 바꾸겠다.

I think it's better to know what one is doing, and where to find it.

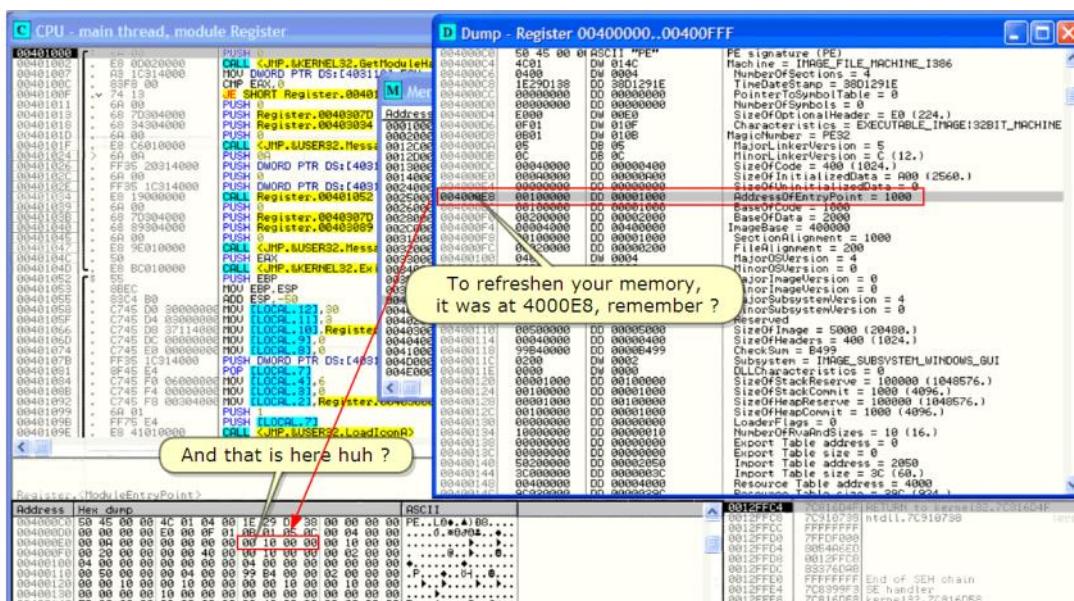
That's also why we made this little trip in the PE file structures.

내 생각에 무슨 일을 하는지 알기 위해 그리고 그것을 찾기 위해 이것이 더 좋은 생각이다.

왜 PE file 구조에서 약간의 이동을 만들었을까?

So, let's simply do it in Olly ;)

Olly에서 매우 간단하게 할 수 있다.

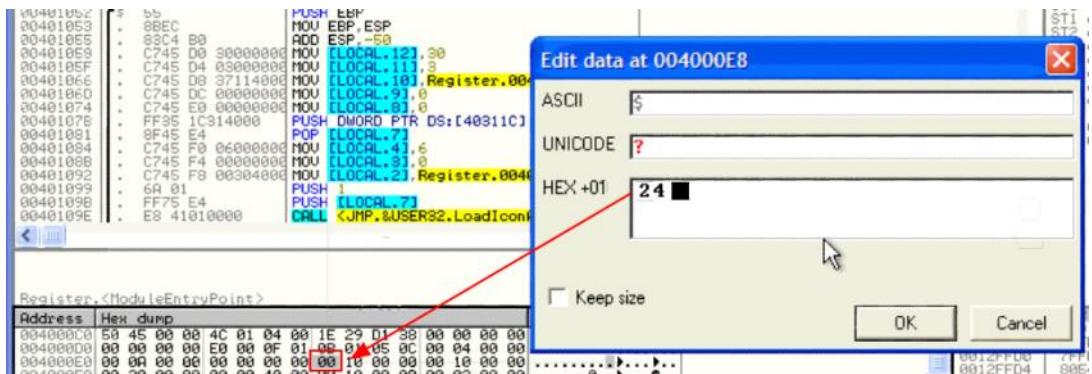


To refreshen your memory, it was at 4000E8, remember?

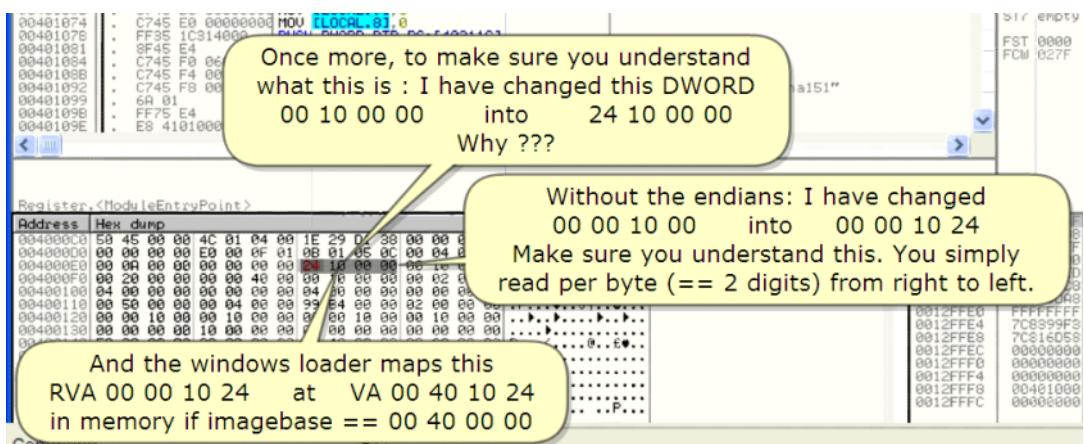
너의 memory를 새롭게 한다. 4000E8 이었다. 기억해?

그리고 이것이 무엇이냐?

:)  
:)



:)



Once more, to make sure you understand what this is : I have changed this DWORD 00 10 00 00 into 24 10 00 00 Why?

한가지 더, 물론 이것이 무엇인지 이해할 수 있게 만들었다 : 나는 DWORD 00 10 00 00을 24 10 00 00 으로 바꿨다. 왜?

Without the endians: I have changed 00 00 10 00 into 00 00 10 24

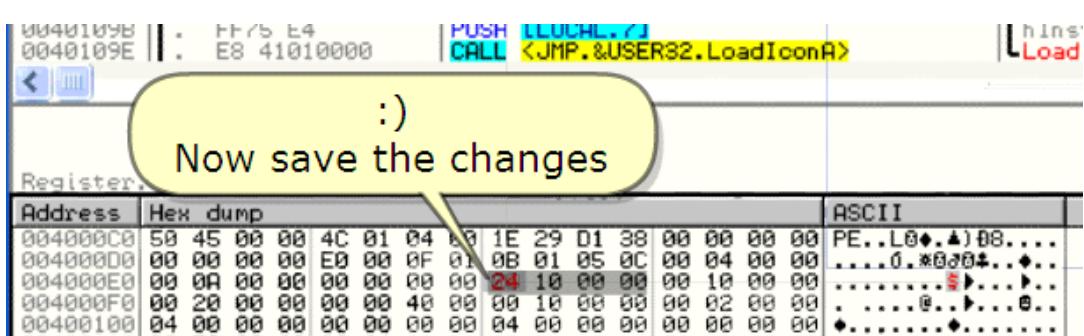
Make sure you understand this. You simply read per byte(== 2digits) from right to left.

Endian이 없을 때 : 00 00 10 00 에서 00 00 10 24로 바꾼다.

물론 너는 이것을 이해했을 것이다. 너는 간단히 byte를 오른쪽에서 왼쪽으로 읽는다.

And the window loader maps this RVA 00 00 10 24 at VA 00 40 10 24 in memory if ImageBase == 00 40 00 00

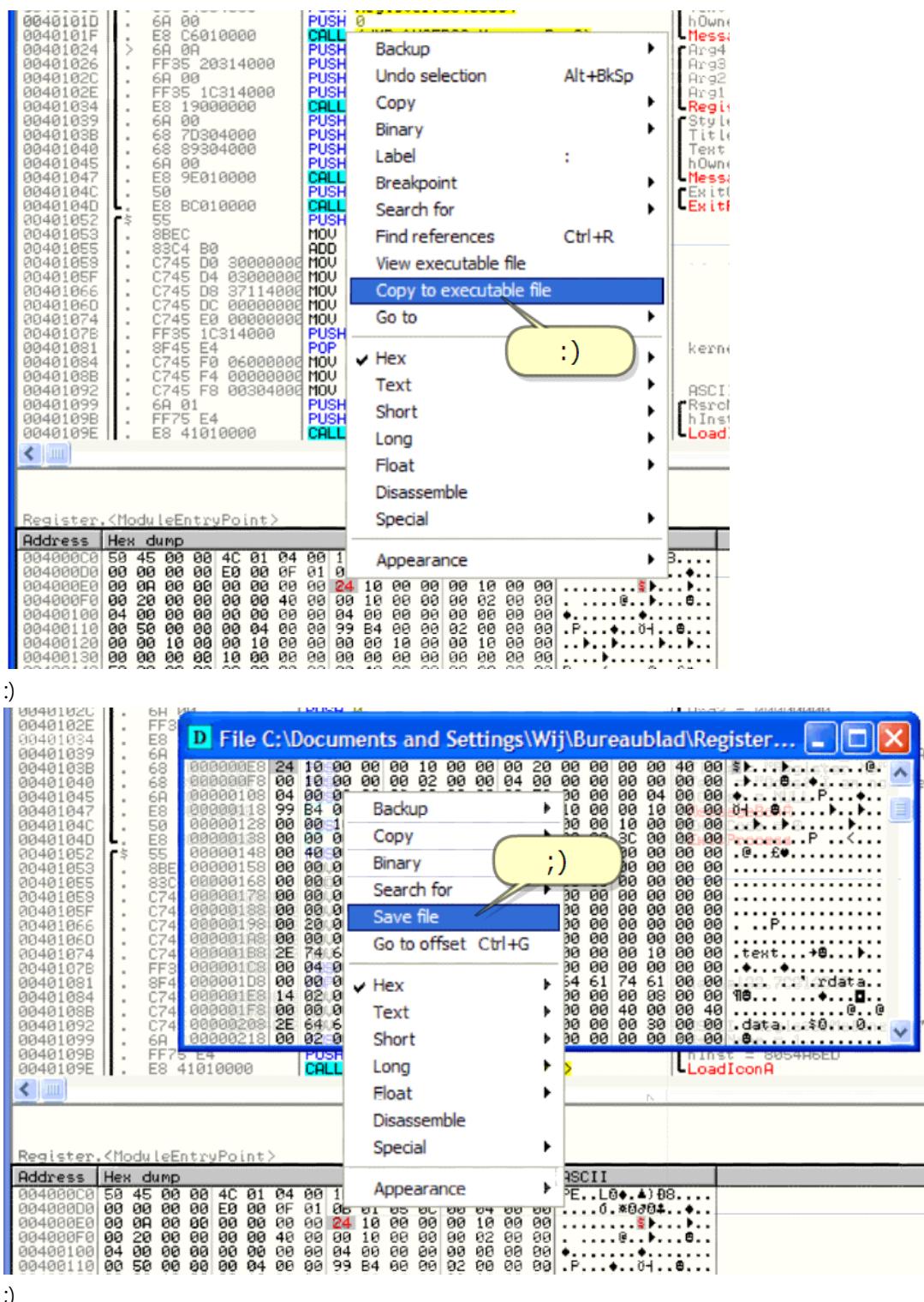
그리고 만약에 ImageBase == 00 40 00 00이라면 window loader는 메모리에서 RVA 00 00 1024를 VA 00 40 10 24로 map 한다.

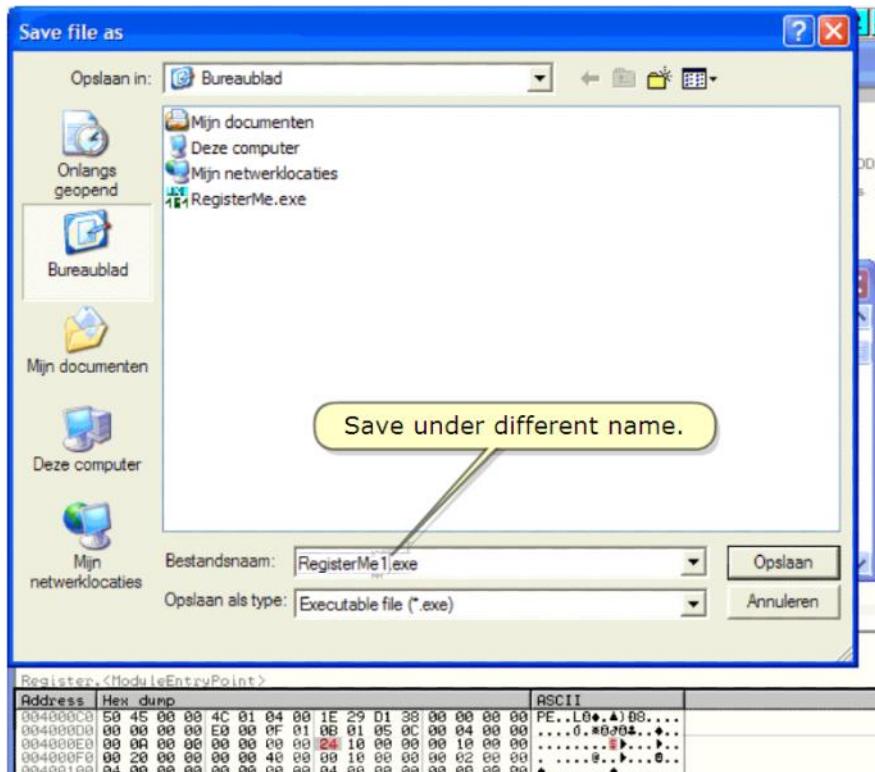


:)

Now save the changes

이제 바뀐 것을 저장해라.

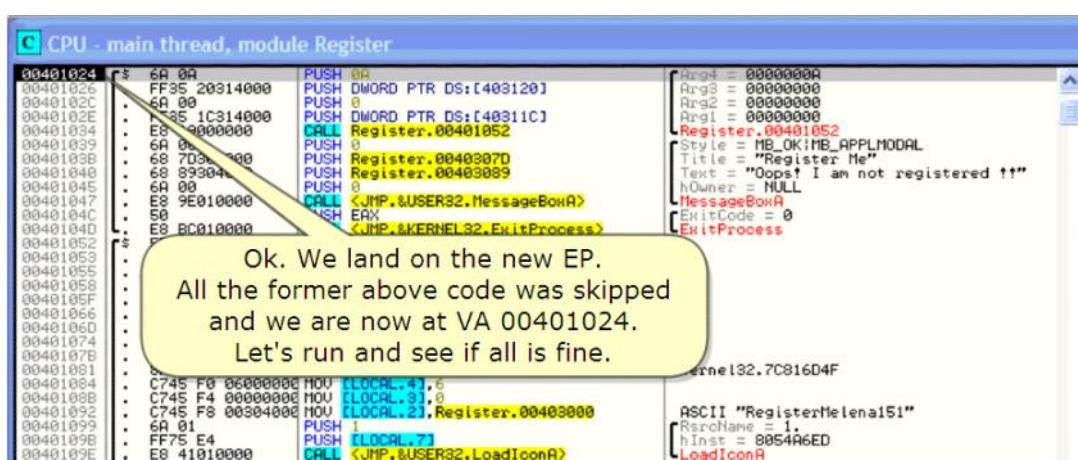




Save under different name.

다른 이름으로 저장해라.

:)



Ok. We land on the new EP.

All the former above code was skipped and we are now at VA 00401024.

Let's run and see if all is fine.

Ok. 우리는 새로운 EP에 도착했다.

모든 예전의 위의 code는 skip 됐다 그리고 우리는 이제 VA 00401024에 있다.

시작하자 그리고 모두 괜찮다는 것을 보자.

All right. There is no nag.

That's fine. Now, let's find something for the second nag too because ... the second nag is till there of course. Just look.

좋아. Nag이 없다.

좋아. 이제 second nag을 띄우는 무언가를 찾아보자. 왜냐하면 second nag도 물론 있다. 곧 본다.

```

MOV [LOCAL_3], 0
MOV [LOCAL_2], Register.00403000
PUSH 1
PUSH [LOCAL_7]
CALL <JMP.&USER32.LoadIconA>
MOU [LOCAL_6], EAX
MOU [LOCAL_11], EAX
PUSH 7F00
PUSH 0
CALL <JMP.&USER32.LoadCursorA>
MOU [LOCAL_5], EAX
LEA EAX, [LOCAL_12]
PUSH EAX
CALL <JMP.&USER32.RegisterClassExA>
PUSH 0

```

:(  
:(

Killing the nag ?

The solution is easy : again there are different solutions (see before) but let's just NOP the call to the messagebox. Do it properly and NOP the arguments too. Like this.

Nag를 죽이자.

이 해결방법은 쉽다. : 다시 다른 해결방법을 보자( 전에 봤다) 그러나 NOP 을 messagebox에 부른다. 올바로 하자. 그리고 NOP을 인자로 보내자. 이렇게.

Let's kill the messagebox by NOPing the 5 lines of its code.

우리는 NOPing에 의해 messagebox가 사용된 5line의 code를 없앤다.

We saw before that the 4 arguments for the messagebox are pushed on the stack first before the call to the API processes them.

우리는 첫번째로 API processes를 호출한 후에 MessageBox를 위해 4 arguments가 stack 넣어 진 것을 봤다.

However, assembling

ADD ESP, 10

Instead of the call to MessageBoxA at line 401047 (and two NOP's) will also kill the messagebox.

그러나, assembling을 하면

ADD ESP, 10

401047 line의 MessageBoxA를 호출하는 대신에(2개의 NOP) 우리는 messagebox를 죽일 것이다.

This is because this line re-balances the stack for the 4 arguments.

Anyway, and like proposed before: let's kill the messagebox by NOPing all of its code

그래서 이 line은 다시 4 arguments를 위해 stack에서 balance를 맞춘다.

언제든지, 그리고 그전처럼 제안한다 : messagebox를 NOP에 의해 모든 code를 kill 한다.

**C CPU - main thread, module Register**

```

00401024 $ 6A 0A PUSH 0A
00401026 . FF35 20314000 PUSH DWORD PTR DS:[403120]
0040102C . 6A 00 PUSH 0
0040102E . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401034 . E8 19000000 CALL Register.00401052
00401039 . 6A 00 PUSH 0
0040103B . 68 7D304000 PUSH Register.00403070
00401040 . 68 89304000 PUSH Register.00403070
00401045 . 6A 00 PUSH 0
00401047 . E8 9E010000 CALL <JMP>
0040104C . E8 BC010000 PUSH EAX
00401052 . 55 PUSH EBP
00401053 . 8BEC MOU EBP,E
00401055 . 89C4 B8 ADD ESP,8
00401058 . C745 D0 30000000 MOU ELOC
0040105F . C745 D4 03000000 MOU ELOC
00401066 . C745 D8 37114000 MOU ELOC
0040106D . C745 DC 00000000 MOU ELOC
00401074 . C745 E0 00000000 MOU ELOC
00401078 . FF35 1C314000 PUSH DWORD
00401081 . 8F45 E4 POP ELOC
00401084 . C745 F0 06000000 MOU ELOC
0040108B . C745 F4 00000000 MOU ELOC
00401092 . C745 F8 00304000 MOU ELOC
00401099 . 6A 01 PUSH 1
0040109B . FF75 E4 PUSH ELOC
0040109E . E8 41301000 CALL <JMP>
004010A3 . 8945 E8 MOU ELOC
004010A6 . 8945 FC MOU ELOC
004010A9 . 68 00700000 PUSH 7F00
004010E6 . 6A 03 PUSH 0
004010B0 . E8 29010000 CALL <JMP>
004010B5 . 8945 EC MOU ELOC
004010B8 . 8D45 D0 LEA EAX,
004010B9 . 50 PUSH EAX
004010BEC . E8 35010000 CALL <JMP>
004010C1 . 6A 00 PUSH 0

```

Rightclick :)

Register, <ModuleEntryPoint>+15

Address | Hex dump

00401052=Register.00401052

Register, <ModuleEntryPoint>+15

Address | Hex dump

0012FFC4

**C CPU - main thread, module Register**

```

00401024 $ 6A 0A PUSH 0A
00401026 . FF35 20314000 PUSH DWORD PTR DS:[403120]
0040102C . 6A 00 PUSH 0
0040102E . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401034 . E8 19000000 CALL Register.00401052
00401039 . 90 NOP
0040103A . 90 NOP
0040103B . 90 NOP
0040103C . 90 NOP
0040103D . 90 NOP
0040103E . 90 NOP
0040103F . 90 NOP
00401040 . 90 NOP
00401041 . 90 NOP
00401042 . 90 NOP
00401043 . 90 NOP
00401044 . 90 NOP
00401045 . 90 NOP
00401046 . 90 NOP
00401047 . 90 NOP
00401048 . 90 NOP
00401049 . 90 NOP
0040104A . 90 NOP
0040104B . 90 NOP
0040104C . 50 E3 BC010000 PUSH EBP
0040104D . 00 CALL Register.00401052
00401052 . 55 PUSH EBP
00401053 . 8BEC MOU EBP,E
00401055 . 89C4 B8 ADD ESP,8
00401058 . C745 D0 30000000 MOU ELOC
0040105F . C745 D4 03000000 MOU ELOC
00401066 . C745 D8 37114000 MOU ELOC
0040106D . C745 DC 00000000 MOU ELOC
00401074 . C745 E0 00000000 MOU ELOC
00401078 . FF35 1C314000 PUSH DWORD PTR DS:[40311C]
00401081 . 8F45 E4 POP ELOC
00401084 . C745 F0 06000000 MOU ELOC
0040108B . C745 F4 00000000 MOU ELOC

```

NOP == No Operation == do nothing.  
 Do you understand that where the messagebox code was, there is now "it's ok, do nothing" ??

NOP == 명령어 없다 == 아무것도 안 한다.

MessageBox code가 아무것도 안 한다는 것을 이해했어?

INFO:

I'm only showing some possibilities whilst bringing proof there are always different solutions to a problem in assembler.

Usually though, the reverser will go for the smallest changes/patches brought to the code.

Assembler에서 문제에 대한 다른 해결책은 여러 가지 있을 수 있다. 그 중에서 몇 가지만 보여 준 것뿐이야.

보통의 경우, reverser는 최대한 작은 patche를 code에 해야 한다.

C CPU - main thread, module Register

```

00401024 5A 0A PUSH ECX
00401026 . FF35 20014000 PUSH DWORD PTR DS:[403120]
0040102C . 6A 00 PUSH 0
0040102E . FF35 1C014000 PUSH DWORD PTR DS:[40311C]
00401034 E8 19000000 CALL Register.00401052
00401039 90 NOP
0040103A 90 NOP
0040103B 90 NOP
0040103C 90 NOP
0040103D 90 NOP
0040103E 90 NOP
0040103F 90 NOP
00401040 90 NOP
00401041 90 NOP
00401042 90 NOP
00401043 90 NOP
00401044 90 NOP
00401045 90 NOP
00401046 90 NOP
00401047 90 NOP
00401048 90 NOP
00401049 90 NOP
0040104A 90 NOP
0040104B 90 NOP
0040104C L: 59 BC010000 PUSH EOX
00401053 55 PUSH EBP
00401055 . 8BEC MOV EBP,ESP
00401056 . 89C4 B0 ADD ESP,-58
00401058 . C745 D0 30000000 MOU [LOCAL..12],30
0040105E . C745 D4 03000000 MOU [LOCAL..11],30
00401066 . C745 D8 37114000 MOU [LOCAL..10],30 Ret
0040106D . C745 D0 00000000 MOU [LOCAL..9],30
00401074 . C745 E9 00000000 MOU [LOCAL..8],30
00401078 . FF35 1C014000 PUSH DWORD PTR DS:
00401081 . F45 E4 POP [LOCAL..7]
00401084 . C745 F0 06000000 MOU [LOCAL..4],6
00401088 . C745 F4 00000000 MOU [LOCAL..3],0

```

Register.00401052

Backup Copy Binary Undo selection Assemble Label Comment Breakpoint Hit trace Run trace Go to Follow in Dump Search for Find references to View Copy to executable Selection Analysis

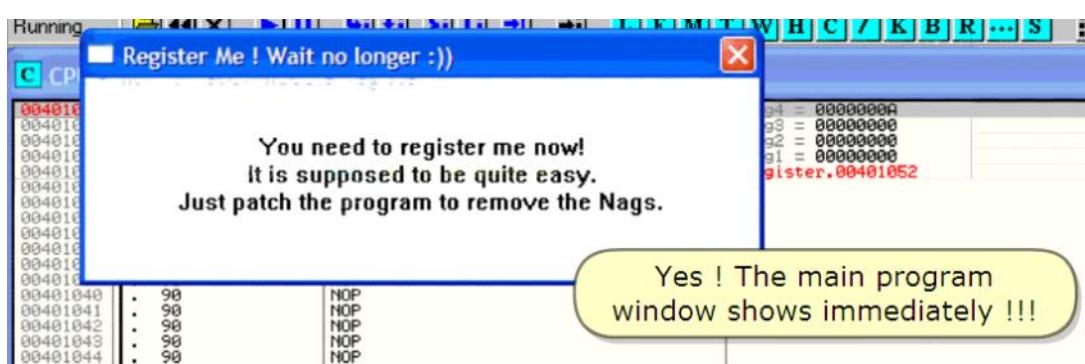
:( )

Saving

All that rests now is ...

모든 것을 이제 새롭게 시작하자.

## 7. Testing the RegisterMe



Yes! The main program window shows immediately !!!

예! Main program window가 즉시 보인다.

Terminated

```

C CPU - main thread, module ntdll
7C90EB94 C3
7C90EB95 8D424 00000000
7C90EB9C 8D424 00
7C90EBA0 90
7C90EBA1 90
7C90EBA2 90
7C90EBA3 90
7C90EBA4 90
7C90EBA5 8D5424 08
7C90EBA9 CD 2E
7C90EBAB C3
7C90EBAC 55
7C90EBAD 8BEC
7C90EBAF 9C
7C90EBB0 81EC D0020000
7C90EBB6 8985 DCFDFFFF
7C90EBBC 898D D8FDFFFF
7C90EBC2 8B45 08
7C90EBC5 8B4D 04
7C90EBC8 8948 0C
7C90EBCB 8085 2CFDFFFF
7C90EBD1 8988 B8000000
7C90EBD7 8998 A4000000
7C90EBDD 8990 A8000000
7C90EBE3 8980 A0000000
7C90EBE9 8988 9C000000
7C90EBEF 8D4D 0C
7C90EFE2 8988 C4000000
RETN
LEA ESP, DWORD PTR SS:[ESP]
LEA ESP, DWORD PTR SS:[ESP]
NOP
NOP
NOP
NOP
NOP
NOP
INT 2E
RETN
PUSH EBP
MOV EBP, ESP
PUSHFD
SUB ESP, 200
MOV DWORD PTR SS:[EBP-224], EAX
MOV DWORD PTR SS:[EBP-228], ECX
MOV EAX, DWORD PTR SS:[EBP+8]
MOV ECX, DWORD PTR SS:[EBP+4]
MOV DWORD PTR DS:[EAX+C], ECX
LEA EAX, DWORD PTR SS:[EBP-20]
MOV DWORD PTR DS:[EAX+B8], ECX
MOV DWORD PTR DS:[EAX+A4], EBX
MOV DWORD PTR DS:[EAX+A8], EDI
MOV DWORD PTR DS:[EAX+A0], ES
MOV DWORD PTR DS:[EAX+9C], EDI
LEA ECX, DWORD PTR SS:[EBP+C]
MOV DWORD PTR DS:[EBX+A4], ECX
!!!  

No nags anymore  

Mission completed.  

Congratulations !!!

```

!!!

No nags anymore

Mission completed.

Congratulations !!!

더 이상 nag이 없다.

목표 성공했다.

축하해 !!!

Ok. But I also promised to show you how important it is to know something about the PE structure, especially the PE header.

Ok. 그러나 나는 너에게 특별히 PE header에서 PE 구조를 알기 위해 중요한 것을 보여주기로 약속했다.

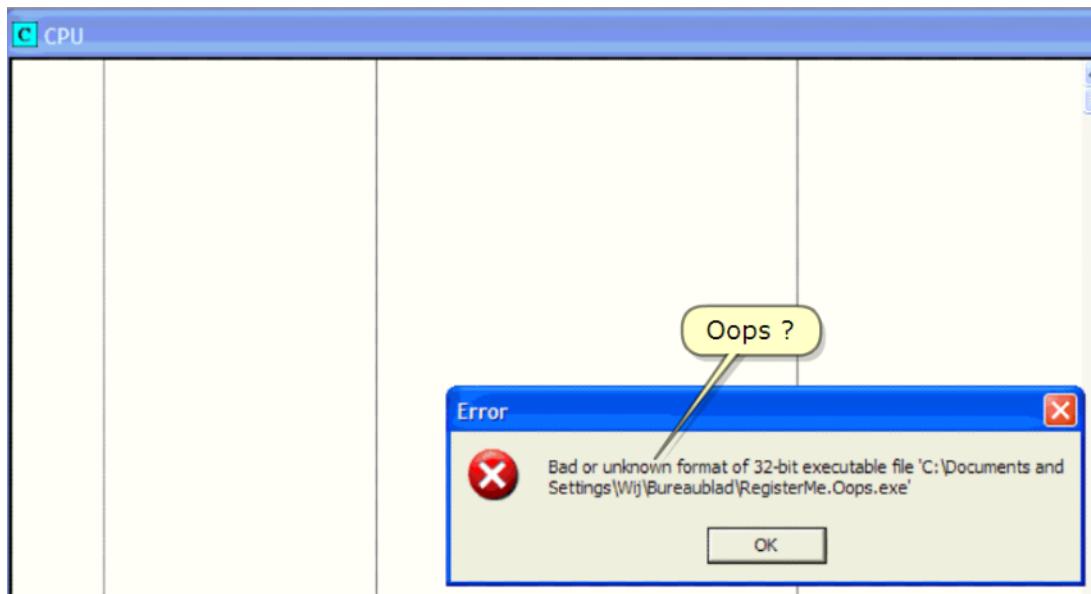
For that, I have included RegisterMe.Ooops.exe.

나는 RegisterMe.Ooops.exe를 포함했다.

Outside our debugger, it runs fine (XP), and behaves exactly like the other RegisterMe, but load it in Olly and ....

Debugger 바깥에서, XP에서는 실행이 잘된다. 그리고 행동이 정확히 다른 RegisterMe, 그러나 Olly에서 load된다. 그리고 ...

Loading the Registerme.Ooops.exe



Oops ?

A screenshot of a debugger interface showing assembly code for the main thread of the ntdll module. The assembly code is as follows:

```
CPU - main thread, module ntdll
7C901231 C3          RETN
7C901232 8BFF        MOU EDI,EDI
7C901234 90          NOP
7C901235 90          NOP
7C901236 90          NOP
7C901237 90          NOP
7C901238 90          NOP
7C901239 CC          INT3
7C90123A C3          RETN
7C90123B 90          NOP
7C90123C 8BFF        MOU EDI,EDI
7C90123E 90          NOP
7C90123F 90          NOP
7C901240 90          NOP
7C901241 90          NOP
7C901242 90          NOP
7C901243 8B4424 04   MOV EAX,DWORD PTR SS:[ESP+4]
7C901247 CC          INT3
7C901248 C2 0400      RETN 4
7C90124B 90          NOP
7C90124C 90          NOP
7C90124D 90          NOP
7C90124E 90          NOP
7C90124F 90          NOP
7C901250 64:A1 10000000 MOV EAX,DWORD PTR FS:[10]
7C901251 C3          RETN
7C901252 90          NOP
7C901253 90          NOP
7C901254 90          NOP
7C901255 90          NOP
7C901256 90          NOP
7C901257 90          NOP
7C901258 90          NOP
7C901259 90          NOP
7C90125A 90          NOP
7C90125B 90          NOP
7C90125C 57          PUSH EDI
7C90125D 8B7C24 0C   MOV EDI,DWORD PTR SS:[ESP+C]
7C90125E 8B5424 08   MOV EDX,DWORD PTR SS:[ESP+8]
7C90125F C702 00000000 MOV DWORD PTR DS:[EDX],0
7C901260 897A 04   MOV DWORD PTR DS:[EDX+4],EDI
7C901261 00FF        OR EDI,EDI
7C901262 74 1E       JE SHORT ntdll.7C901290
7C901263 83C9 FF     OR ECX,FFFFFF
7C901264 33C0        XOR EAX,EAX
7C901265 F2:AE      REPNE SCAS BYTE PTR ES:[EDI]
7C901266 F7D1        NOT ECX
7C901267 -----
```

A yellow speech bubble with the text 'Oops ?' points to the assembly instruction at address 7C901250.

Return to 7C93EDC0 (ntdll.7C93EDC0)

Oops?

```

7C901259 90 NOP
7C90125A 90 NOP
7C90125B 90 NOP
7C90125C 57 PUSH EDI
7C90125D 8B7C24 0C MOV EDI,DWORD PTR SS:[ESP+C]
7C901261 8B5424 08 MOV EDX,DWORD PTR SS:[ESP+8]
7C901265 C702 00000000 MOV DWORD PTR DS:[EDX],0
7C90126B 897A 04 MOV DWORD PTR DS:[EDX+4],EDI
7C90126E 0BF0 OR EDI,EDI
7C901270 74 1E JE SHORT ntdll.7C901290
7C901272 83C9 FF OR ECX,FFFFFF
7C901275 93C0 XOR EAX,EAX
7C901277 F2:AЕ REPNE SCAS BYTE PTR ES:[EDI]
7C901279 F7D1 NOT ECX

```

Return to 7C93EDC0 (ntdll.7C93EDC0)

Address	Hex dump	ASCII

Oops?

Perhaps you wonder what all this is. This is not the usual view for a disassembled exe in Olly. Right?

아마 너는 이것이 무엇인지 궁금할 것이다. 이것은 Olly에서 일반적으로 disassembled된 exe 상황이 아니다. 그렇지?

INFO :

Indeed, it is not. I have done with this exe what some packers/protectors tend to do to keep away newbie reversers: using some debugger "bugs" and stuff in the header.

정말, 이것은 아니다. 나는 이미 exe를 끝냈다. 약간의 packers/protectors의 성향을 보인다.  
Newbie reverse가 떠나는 것을 지키기 위해 해야 한다. : 몇 개의 debugger를 사용할 때 "bugs" 와 header에서 쓰일 때

Olly gets "confused", and to protect himself from crashing, he stops at system breakpoint instead of at EP.

Olly는 "혼란"을 얻는다. 그리고 그것을 충돌로부터 지키기 위해, 그는 EP 대신에 system breakpoint을 걸어 멈춘다.

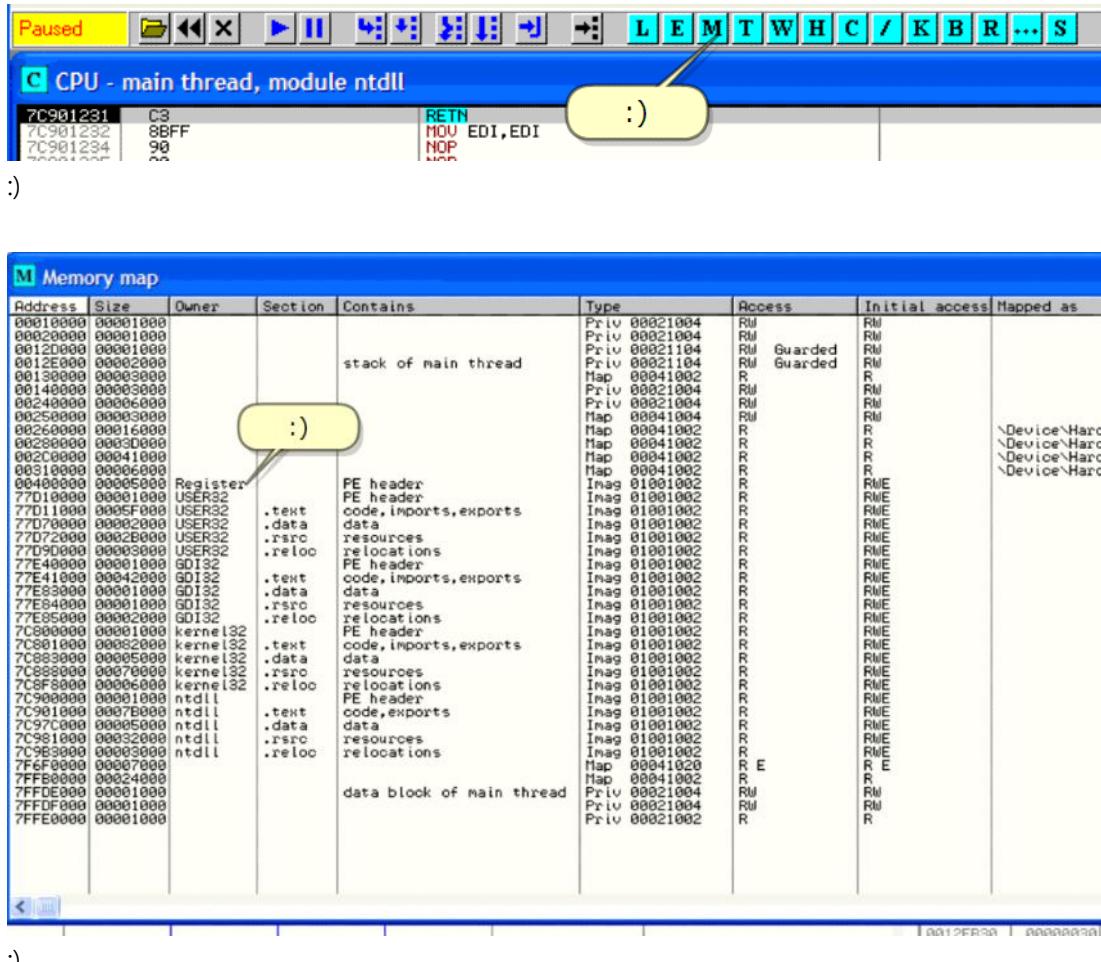
This however can be easily bypassed. Allow me to quickly show you how to do this and I will explain more detailed afterwards.

그러나 우회도로를 만들기 쉽다. 어떻게 해야 하는지 빠르게 보여주기를 위해 허락한다. 그리고 나는 후에 좀 더 자세하게 설명할 것이다.

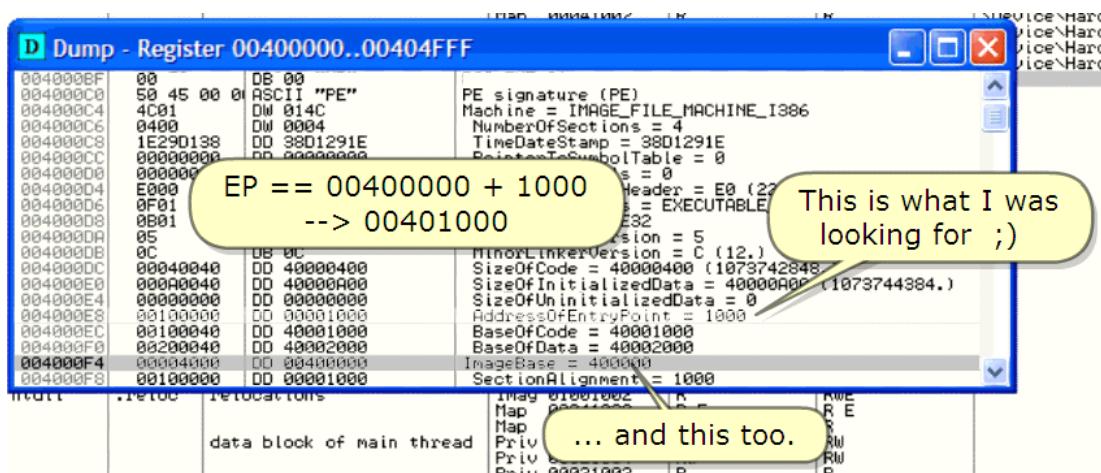
INFO :

Sometimes, it is necessary to make Olly pause at system breakpoint. You can find this possibility under "Options" --> "Debugging options" --> "Events" --> check "System breakpoint"

가끔, Olly가 system breakpoint를 걸 때가 필요하다. 너는 이곳에서 찾을 수 있다.



:)



This is what I was looking for ;)

내가 무엇을 보고 있는지 봐.

...and this too.

EP == 00400000 + 1000

---&gt; 00401000

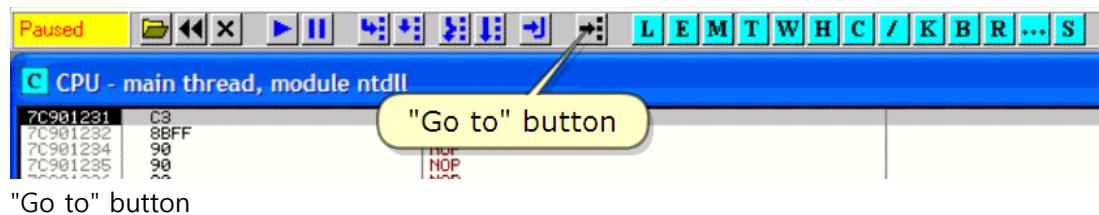
INFO :

Once again : I will explain all this more in detail later. Just allow me to show you how to quickly deal with this first.

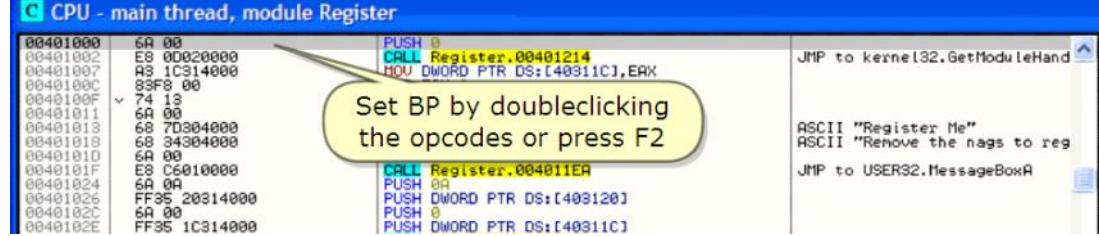
--&gt; go to EP (at VA 00401000) and set BP and run till BP.

다시 한 번: 나는 이 모든 것을 나중에 자세히 설명할 것이다. 지금은 어떻게 빠르게 처리하는지 보여주겠다

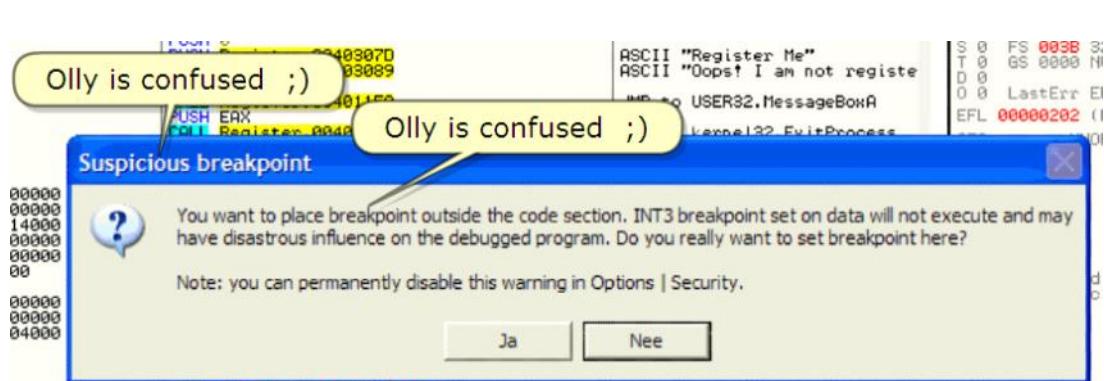
EP로 가라(at VA 00401000) 그리고 BP를 설치하고 BP까지 실행하라.



"Go to" button



Opcodes를 doubleclick 하거나 F2를 눌러서 BP를 설치해.



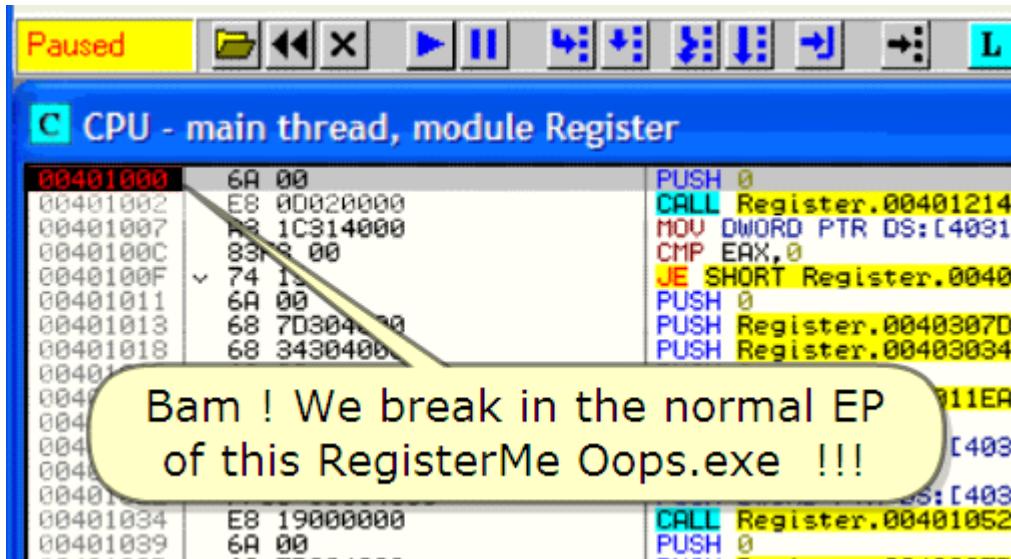
Olly is confused ;)

올리는 혼잡해졌어.

:)

:)

:)



Bam! We break in the normal EP of this RegisterMe Oops.exe !!!

우리는 일반적인 EP로 RegisterMe Oops.exe를 멈췄다.

INFO :

You can also avoid most of this with the plugin "ollyadvanced.dll" from MaRKuSTH\_DJM.

(Check the right options in the plugin). You can download oollyadvanced

너는 이것을 MaRKuSTH\_DJM이 만든 "ollyadvanced.dll" plugin과 함께라면 피할 수 있다.

(plugin에서 right options을 check 해.) 너는 oollyadvanced를 download 할 수 있다.

<http://tuts4you.com/download/php?view.75>

And extract it to your Plugins dir in Olly. Or perform a search for the latest version there.

그리고 Olly에서 너의 plugins dir로 추출할 수 있다. 또는 최신 버전을 검색할 수 있다.

Now restart Olly and load the ReverseMe Oops.exe, but it may take quite some time before Olly has allocated all (superfluous) bytes etc. (Olly will "eat" lots of memory)

이제 Olly를 재시작하고 ReverseMe Ooops.exe를 load 해라, Olly가 모든 bytes가(불필요한) 할당되기 전에 이것은 꽤 시간을 얻을 수 있다. (Olly는 많은 memory를 차지한다)

UPDATE:

MaRKuS TH\_DJM informs me that he has added extra features to his plugin : now is also taken care of "Handle Base of Code, Size of Code and Base of Data",

MaRKuS TH\_DJM는 추가적인 특징들을 그의 plugin에 추가했다는 사실을 알려줬다. : 이제 조심히 "Handle Base of Code, Size of Code and Base of Data"를 얻을 수 있다.

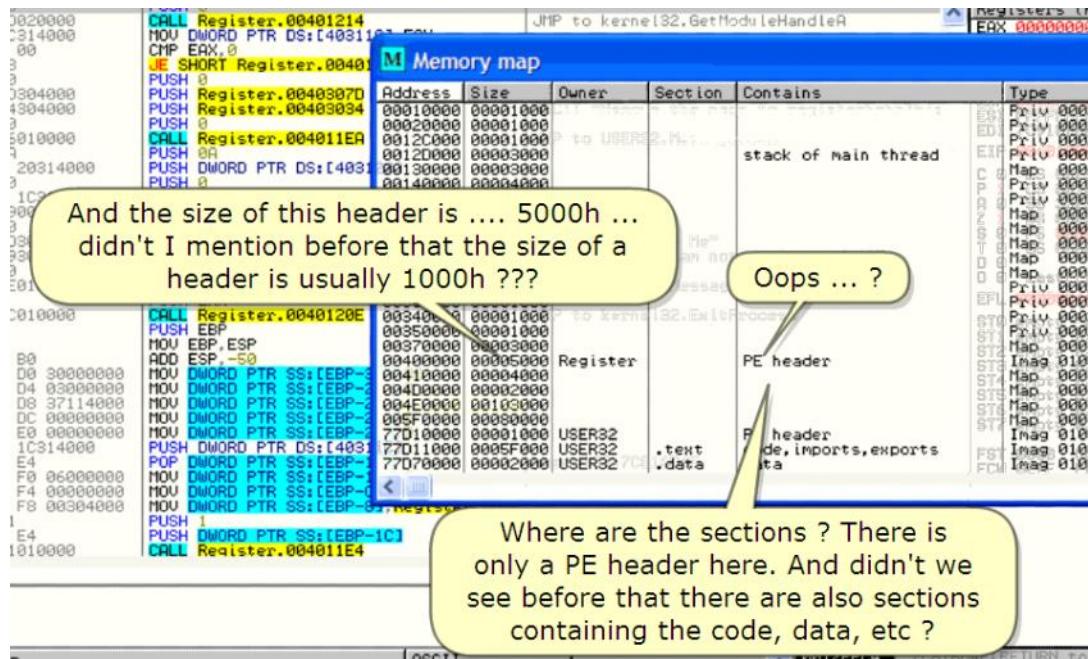
so that ReverseMeOoops.exe is now also correctly loaded. Thanks maRKuS TH\_DJM

그래서 ReverseMeoops.exe는 이제 정확히 load 됐다. 고마워 maRKuS TH\_DJM.

Ok, let me explain all this a little : some virii but also protectors (see later tuts, I'll come back to this) can deliberately manipulate data in the PE header as anti-debug tricks (...etc). Ok. 이제 모든 것에 대해서 약간 설명한다 : 약간의 virii 그러나 protector는 (나중에 배울 것이다. 이 주제로 돌아올 것이다.) anti-debut trick을 사용할 때 PE header에서 고의로 data를 조종한다.

"Why and how" you ask? Well, let's take a look in the header...

"왜 그리고 어떻게?" 질문하겠지? Header를 살펴보자.



Oops ...?

Where are the sections ? There is only a PE header here. And didn't we see before that there are also sections containing the code, data, etc?

Sections들이 어디 갔어? 이곳은 오직 PE header만 있다. 그리고 그곳에서 sections들이 code, data, etc를 포함하는 것을 우린 보지 못했다.

And the size of this header is .... 5000h ....

Didn't I mention before that the size of a header is usually 1000h ???

그리고 header size가 50000h ....

전에 header size는 보통 1000h 아니었나? 언급하지 않을 수 없다.

Now, to turn a long story into a brief explanation : some stuff in the PE header has been changed without making a really fundamental change (exactly like some virii, protectors, etc can do).

이제 long story를 간단히 설명 하겠다 : PE header의 약간의 구성들은 바꼈다. 정말 필수적인 것이 바뀌지는 않았다.(정확히 약간 virii, protectors, etc가 하는 것과 비슷하다)

The result is that the exe still runs (winXP) but Olly gets confused by all these anomalies (and can be busy for a while, trying to find everything and allocating memory for it). Let's see the header ...

이 결과로 아직까지 실행은 된다.(winXP) 그러나 Olly는 모든 변칙적인 것에 의해 혼란이 된다.(그리고 이것은 일시적으로 매우 바쁘다. 모든 것을 찾기 위해 노력하고 그것을 위해 memory를 할당한다)

Header를 보자.

Address	Size	Owner	Section	Contains	Type	Access	Ir
00010000	00001000				Priv	00021004	RW
00020000	00001000				ED	00021004	RW[0738]
0012C000	00001000			stack of main thread	EIP	00021004	Guarded
00130000	00003000				CS	00041002	R
00140000	00004000				CS	00041004	RW
00240000	00006000				DS	00041004	RW
00250000	00003000				DS	00041002	RW
00260000	00016000				DS	00041002	RW
00280000	00030000				DS	00041002	R
				DLL "Register Me"	Map	00041002	R
				DLL "Oops! I am not registered :)"	Map	00041002	R
				so USER32.MessageBoxA	Map	00041002	R
00340000	00001000				O	00021040	RW[INIT_FAILED]
00350000	00001000				EFL	00021040	NB, RWE, NS, PE, OE,
00370000	00003000				ST0	00021004	EDB01000104 000
00400000	00005000	Register		PE header	ST1	00021004	RW
00410000	00004000				ST2	00041002	R
004D0000	00002000				ST3	00041002	R
004E0000	00103000				ST4	00041020	R
005F0000	00080000				ST5	00041020	R
77D10000	00001000	USER32		PE header	ST6	00041020	R
77D11000	0005F000	USER32		code, imports, exports	ST7	01001002	R
77D70000	00002000	USER32		data	FS1	01001002	R
					FCM	01001002	Mask

Doubleclick

And scroll down

스크롤 내려봐

D Dump - Register 00400000..00404FFF			
004000C6	0400	DW 0004	NumberOfSection
004000C8	1E290138	DD 3801291E	TimeDateStamp = 3801291E
004000CC	00000000	DD 00000000	PointerToSymbolTable = 0
004000D0	00000000	DD 00000000	NumberOfSymbols = 0
004000D4	E000	DW 00E0	SizeOfOptionalHeader = E0 (4.0)
004000D6	0F01	DW 010F	Characteristics = EXECUTABLE_IMAGE 32BIT_MACHINE
004000D8	0001	DW 0108	MagicNumber = PE32
004000DA	05	DB 05	MajorLinkerVersion = P
004000DC	0C	DB 0C	MinorLinkerVersion = C (12.)
004000DC	00040040	DD 40000400	SizeOfCode = 40000400 (10373742848.)
004000E0	00000004	DD 04000000	SizeOfInitializedData = 4000000 (67111424.)
004000E4	00000000	DD 00000000	SizeOfUninitializedData = 0
004000E8	00100000	DD 00001000	AddressOfEntryPoint = 1000
004000EC	00100000	DD 40001000	BaseOfCode = 40001000
004000F0	00200040	DD 40002000	BaseOfData = 40002000
004000F4	00004000	DD 00400000	ImageBase = 400000
004000F8	00100000	DD 00001000	SectionAlignment = 1000
004000FC	00020000	DD 00002000	FileAlignment = 200
00400100	0400	DW 0004	MajorOSVersion = 4
00400102	0000	DW 0000	MinorOSVersion = 0

400000400 ???

Size of code should be 400 instead of 40000400

400000400 ???

400000400 대신에 code의 size는 400 이다.

D Dump - Register 00400000..00404FFF			
004000C6	0400	DW 0004	NumberOfSections = 4
004000C8	1E290138	DD 3801291E	TimeDateStamp = 3801291E
004000CC	00000000	DD 00000000	PointerToSymbolTable = 0
004000D0	00000000	DD 00000000	NumberOfSymbols = 0
004000D4	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
004000D6	0F01	DW 010F	Characteristics = 32BIT_MACHINE
004000D8	0001	DW 0108	MagicNumber = PE32
004000DA	05	DB 05	MajorLinkerVersion = P
004000DC	0C	DB 0C	MinorLinkerVersion = C (12.)
004000DC	00040040	DD 40000400	SizeOfCode = 40000400 (10373742848.)
004000E0	00000004	DD 04000000	SizeOfInitializedData = 4000000 (67111424.)
004000E4	00000000	DD 00000000	SizeOfUninitializedData = 0
004000E8	00100000	DD 00001000	AddressOfEntryPoint = 1000
004000EC	00100000	DD 40001000	BaseOfCode = 40001000
004000F0	00200040	DD 40002000	BaseOfData = 40002000
004000F4	00004000	DD 00400000	ImageBase = 400000
004000F8	00100000	DD 00001000	SectionAlignment = 1000
004000FC	00020000	DD 00002000	FileAlignment = 200
00400100	0400	DW 0004	MajorOSVersion = 4
00400102	0000	DW 0000	MinorOSVersion = 0

--> should be A00

A00 이어야 한다.

D Dump - Register 00400000..00404FFF

004000C6	0400	DW 0004	NumberOfSections = 4 TimeDateStamp = 38D1291E PointerToSymbolTable = 0 NumberOfSymbols = 0 SizeOfOptionalHeader = E0 (224.) Characteristics = EXECUTABLE_IMAGE 32BIT_MACHINE MagicNumber = PE32 MajorLinkerVersion = MinorLinkerVersion = SizeOfCode = 40000400 SizeOfInitializedData = 40003400 (67111424.) SizeOfUninitializedData = 0 AddressOfEntryPoint = 1000 BaseOfCode = 40001000
004000C8	1E290138	DD 38D1291E	
004000CC	00000000	DD 00000000	
004000D0	00000000	DD 00000000	
004000D4	E000	DW 00E0	
004000D6	0F01	DW 010F	
004000D8	0B01	DW 010B	
004000DA	05	DB 05	
004000DB	0C	DB 0C	
004000DC	00040040	DD 40000400	
004000E0	000A0004	DD 04000A00	
004000E4	00000000	DD 00000000	
004000E8	00100000	DD 00001000	
004000EC	00100040	DD 40001000	
004000F0	00200040	DD 40002000	
004000F4	00004000	DD 04000000	
004000F8	00100000	DD 00001000	
004000FC	00020000	DD 00000200	
00400100	0400	DW 0004	MajorOSVersion = 4 MinorOSVersion = 0
00400102	0000	DW 0000	

--> should be 1000

D Dump - Register 00400000..00404FFF

004000C6	0400	DW 0004	NumberOfSections = 4 TimeDateStamp = 38D1291E PointerToSymbolTable = 0 NumberOfSymbols = 0 SizeOfOptionalHeader = E0 (224.) Characteristics = EXECUTABLE_IMAGE 32BIT_MACHINE MagicNumber = PE32 MajorLinkerVersion = MinorLinkerVersion = SizeOfCode = 40000400 SizeOfInitializedData = 40003400 (67111424.) SizeOfUninitializedData = 0 AddressOfEntryPoint = 1000 BaseOfCode = 40001000
004000C8	1E290138	DD 38D1291E	
004000CC	00000000	DD 00000000	
004000D0	00000000	DD 00000000	
004000D4	E000	DW 00E0	
004000D6	0F01	DW 010F	
004000D8	0B01	DW 010B	
004000DA	05	DB 05	
004000DB	0C	DB 0C	
004000DC	00040040	DD 40000400	
004000E0	000A0004	DD 04000A00	
004000E4	00000000	DD 00000000	
004000E8	00100000	DD 00001000	
004000EC	00100040	DD 40001000	
004000F0	00200040	DD 40002000	
004000F4	00004000	DD 04000000	Should be 2000 Scroll down
004000F8	00100000	DD 00001000	
004000FC	00020000	DD 00000200	
00400100	0400	DW 0004	
00400102	0000	DW 0000	

Should be 2000

Scroll down

D Dump - Register 00400000..00404FFF

0040011C	0200	DW 0002	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI DLLCharacteristics = 0 SizeOfStackReserve = 100000 (1048576.) SizeOfStackCommit = 1000 (4096.) SizeOfHeapReserve = 100000 (1048576.) SizeOfHeapCommit = 1000 (4096.) LoaderFlags = 0 NumberOfRvaAndSizes = 40000004 (1073741828.)
0040011E	0000	DW 0000	
00400120	00001000	DD 00100000	
00400124	00100000	DD 00010000	
00400128	00001000	DD 00100000	
0040012C	00100000	DD 00001000	
00400130	00000000	DD 00000000	
00400134	04000040	DD 40000004	should be 00000010
00400138	00005000	DD 00050000	
0040013C	00000500	DD 00005000	
00400140	50200000	DD 00002050	
00400144	3C000000	DD 0000003C	
00400148	00400000	DD 00004000	
0040014C	9C030000	DD 0000039C	
00400150	00000000	DD 00000000	
00400154	00000000	DD 00000000	
00400158	00 00 00	0 ASCII " "	
00400160	00006000	DD 00600000	
00400164	00006000	DD 00060000	
00400168	00000000	DD 00000000	

Should be 00000010

BTW, especially this one is bothering Olly ;)

By the way, 특히 이것은 Olly에게 지루하다. ;)

D Dump - Register 00400000..00404FFF

0040011C	0200	DW 0002	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI DLLCharacteristics = 0 SizeOfStackReserve = 100000 (1048576.) SizeOfStackCommit = 1000 (4096.) SizeOfHeapReserve = 100000 (1048576.) SizeOfHeapCommit = 1000 (4096.) LoaderFlags = 0 NumberOfRvaAndSizes = 40000004 (1073741828.)
0040011E	0000	DW 0000	
00400120	00001000	DD 00100000	
00400124	00100000	DD 00010000	
00400128	00001000	DD 00100000	
0040012C	00100000	DD 00001000	
00400130	00000000	DD 00000000	
00400134	04000040	DD 40000004	--> 0
00400138	00005000	DD 00050000	
0040013C	00000500	DD 00005000	
00400140	50200000	DD 00002050	
00400144	3C000000	DD 0000003C	
00400148	00400000	DD 00004000	
0040014C	9C030000	DD 0000039C	
00400150	00000000	DD 00000000	
00400154	00000000	DD 00000000	
00400158	00 00 00	0 ASCII " "	
00400160	00006000	DD 00600000	
00400164	00006000	DD 00060000	
00400168	00000000	DD 00000000	

--> 0

Dump - Register 00400000..00404FFF			
00400011C	0200	DW 0002	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI
00400011E	0000	DW 0000	DLLCharacteristics = 0
004000120	00001000	DD 00100000	SizeOfStackReserve = 100000 (1048576.)
004000124	00100000	DD 00001000	SizeOfStackCommit = 1000 (4096.)
004000128	00001000	DD 00100000	SizeOfHeapReserve = 100000 (1048576.)
00400012C	00100000	DD 00001000	SizeOfHeapCommit = 1000 (4096.)
004000130	00000000	DD 00000000	LoaderFlags = 0
004000134	04000004	DD 40000004	NumberOfRvaAndSizes = 40000004 (1073741828.)
004000138	00005000	DD 00050000	Export Table address = 500000
<b>00400013C</b>	<b>00005000</b>	<b>DD 00050000</b>	<b>Export Table size = 50000 (327680.)</b>
004000140	50200000	DD 00002050	Import Table address = 2050
004000144	3C000000	DD 0000003C	Import Table size = 3C (60.)
004000148	00400000	DD 00004000	Resource Table address = 4000
00400014C	9C030000	DD 0000039C	Resource Table size = 39C (924.)
004000150	00000000	DD 00000000	Exception Table address = 0
004000154	00000000	DD 00000000	Exception Table size = 0
004000158	00 00 00 01	ASCII ""	SECTION
004000160	00006000	DD 00060000	VirtualSize = 600000 (6291456.)
004000164	00006000	DD 00060000	VirtualAddress = 60000
004000168	00000000	DD 00000000	SizeOfRawData = 0

--> 0

I want to remark that there are better tools then Olly to find the right values for this.

We will come to these tools in later parts.

나는 주목하기 위해. 그것은 Olly로 옳은 값을 찾는 것보다 좋은 tool 이다.

우리는 이 tools을 다음 parts에서 볼 수 있다.

For now, let's edit the values. I'll do it in Olly in the dump window.

현재는, value를 수정하겠다. 나는 Olly의 dump window를 사용하겠다.

Remark : you can also edit them here by doubleclicking and binary editing the lines that need changing. MIND THE ENDIANS !!!

주목 : 너는 그것들을 여기에서 doubleclicking과 binary 수정으로 바꾸는 것들이 필요한 line들을 수정할 수 있다. ENDIAN 개념을 잡고 있어라.

After that, you will also need to save the changed opcodes from the dump window (see further).

그 후에, 너는 바꾸는 opcodes를 dump windows에서 최대한 적게 변경하기 위해 필요할 것이다.(나중에 보자)

Once again : there are also many tools that can do this job.

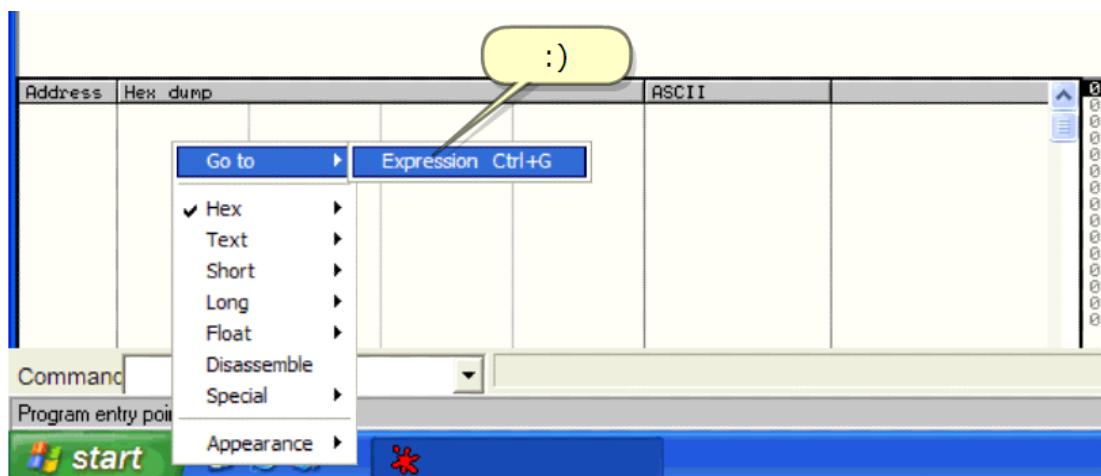
다시 한 번 : 그 일을 할 수 있는 많은 tools이 있다.

However, I think it is important to understand what you are doing.... So, follow along ...

그러나, 내 생각으로 네가 무엇을 하는 것인지 이해하는 것이 중요하다. 따라와라.

번역 주)ENDIAN도 2가지 종류가 있다. Big endian, Little endian. 그런데 lena는 여기에서 endian만 이야기 해준다. 나중에 헷갈릴텐데...인터넷에서 정리가 잘 된 문서입니다.

<http://recipes.egloos.com/4993723>

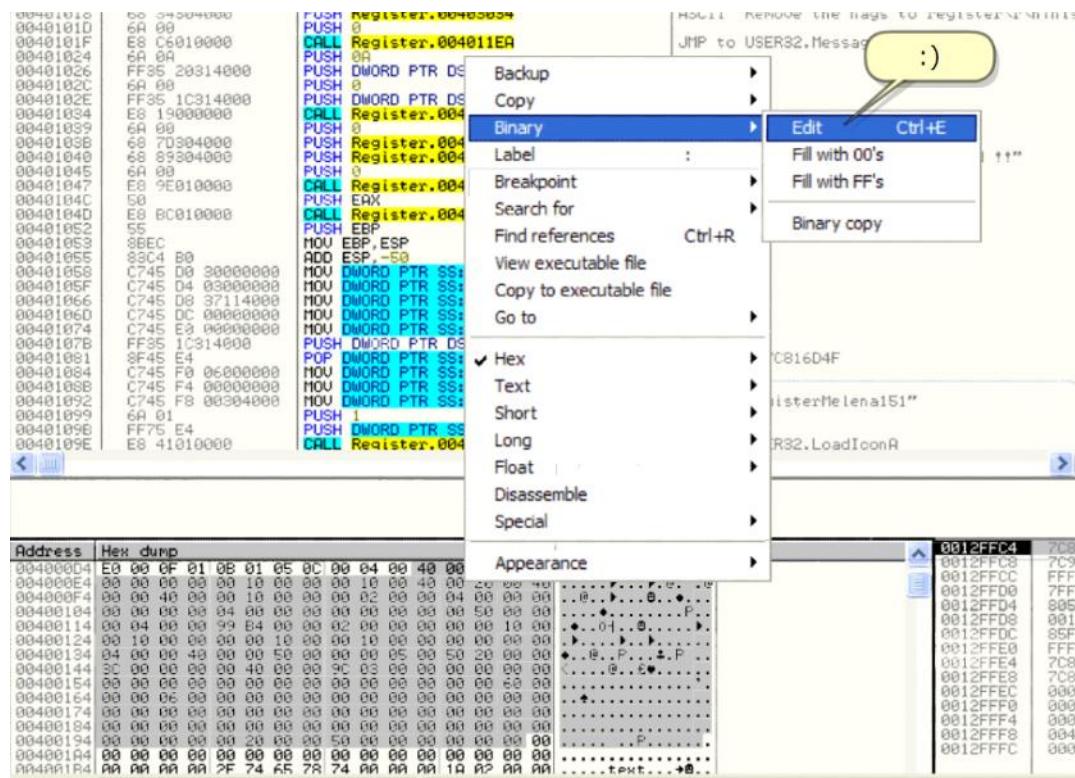


Rightclick

:)

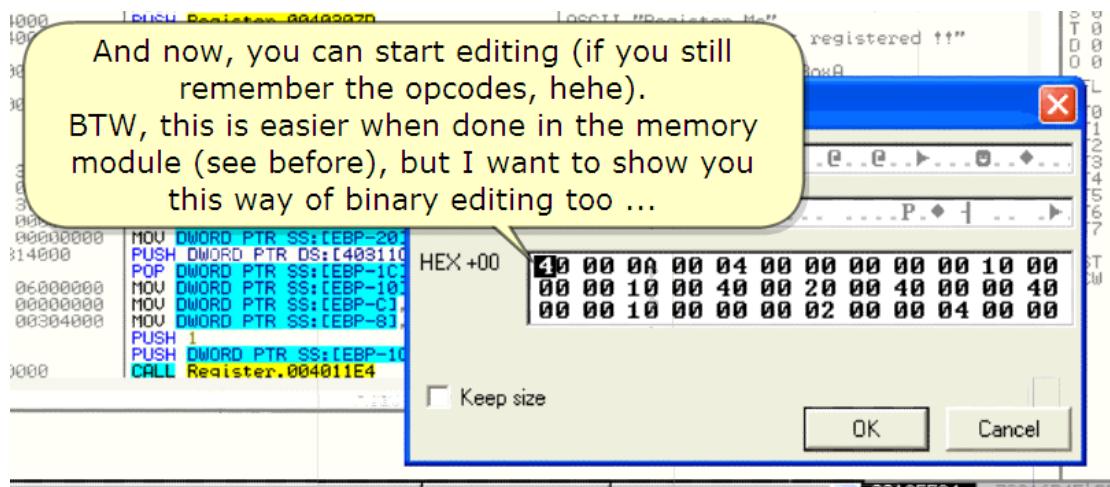
Go to the right place to make the changes

바꾸기 위해 정확한 장소로 이동한다.



Rightclick

:)



And now, you can start editing (if you still remember the opcodes, hehe).

이제 너는 editing을 시작할 수 있다.(만약에 네가 아직까지 opcodes를 기억한다면, 헤헤).

BTW, this is easier when done in the memory module (see before), but I want to show you this way of binary editing too ...

Memory module이 끝났을 때 이것은 쉽다.(전 것을 봐라) 그러나, 나는 너에게 binary editing 방법을 보여주기를 원한다.

Continue editing where necessary like this then click or till you land ....

필요한 공간에서 이렇게 click 하거나 네가 도달할 때까지 Editing을 계속하자.

The screenshot shows a debugger interface with two main panes. The top pane displays assembly code with various opcodes highlighted in red. A yellow callout box contains the following text:

.... here.  
I removed it from the movie to reduce its size but you can see all the changes I have made here. Now, save and test this exe.  
BTW, like I said before : if you have edited these from within the memory module, you will also need to save these changed opcodes like I will show you now (they will not appear in red then !!!). Just follow ...

The bottom pane shows a memory dump table with columns for Address, Hex dump, ASCII, and a list of addresses on the right. Several hex values in the dump are highlighted in red, corresponding to the changes mentioned in the callout box.

Address	Hex dump	ASCII	
004000C4	4C 01 04 00 00 1E 29 D1 98 00 00 00 00 00 00	L...A108...	0012FFC4
004000D4	E0 00 F1 0B 01 05 00 00 00 04 00 00 00 00 00	0...0004...*	0012FFC8
004000E4	00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00	*...P...*	0012FFCC
004000F4	00 00 40 00 00 10 00 00 00 02 00 00 00 04 00 00	*...P...*	0012FFD0
00400104	00 00 00 00 00 04 00 00 00 00 00 00 00 00 50 00	*...P...*	0012FFD4
00400114	00 04 00 00 99 B4 00 00 00 02 00 00 00 00 00 10 00	*...P...*	0012FFD8
00400124	00 10 00 00 00 00 00 00 10 00 00 00 10 00 00 00 00	*...P...*	0012FFDC
00400134	10 00 00 00 00 00 00 00 00 00 00 00 00 00 50 20	*...P...*	0012FFE0
00400144	00 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00	*...P...*	0012FFE4
00400154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	*...P...*	0012FFE8
00400164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	*...P...*	0012FFEC
00400174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	*...P...*	0012FFF0
			0012FFF4

... here.

I removed it from the movie to reduce its size but you can see all the changes I have made here. Now, save and test this exe.

나는 이 movie에서 size를 줄이기 위해 삭제했다. 그러나 너는 여기에서 내가 바꾼 것을 볼 수 있다. 이제 저장하고 test 하자.

BTW, like I said before : if you have edited these from within the memory module, you will also need to save these changed opcodes like I will show you now(they will not appear in red then !!!). Just follow.

By the way, 내가 예전에 말했던 것과 같아 : 네가 이것을 memory module에서 edit 했다면, 또한 너에게 보여준 것처럼 바뀐 opcodes를 저장해야 할 필요가 있다. (그들은 red에서 나타나지 않을 것이다.) 날 따라와.

Rightclick

:)

Rightclick

:)

Saving

Restart to test the exe (or press Ctrl-F2)

Test 하기 위해 Restart 하거나 (Ctrl-F2를 눌러라)

CPU - main thread, module Register

00401000	6A 00	PUSH <b>0</b>
00401002	E8 00020000	<b>CALL</b> <b>&amp;KERNEL32.GetModuleHandleA</b>
00401004	89 1C314000	MOU <b>DWORD PTR DS:[40311C],EAX</b>
00401006	89F8 00	CMP <b>EBX,0</b>
00401008	74 10	<b>JMP SHORT Register.00401024</b>
00401011	6A 00	PUSH <b>0</b>
00401013	E8 7003040000	PUSH <b>Register.00403070</b>
00401015	E8 3400040000	PUSH <b>Register.00403084</b>
00401017	6A 00	PUSH <b>0</b>
0040101F	E8 C0100000	<b>CALL</b> <b>&amp;USER32.MessageBoxA</b>
00401024	6A 00	PUSH <b>0</b>
00401026	FF35 1C314000	PUSH <b>DWORD PTR DS:[403120]</b>
00401028	6A 00	PUSH <b>0</b>
0040102A	FF35 1C314000	PUSH <b>DWORD PTR DS:[40311C]</b>
0040102C	E8 1900000000	<b>CALL</b> <b>Register.00401052</b>
0040102D	6A 00	PUSH <b>0</b>
0040102F	E8 7003040000	PUSH <b>Register.00403070</b>
00401030	E8 8300040000	PUSH <b>Register.00403084</b>
00401032	6A 00	PUSH <b>0</b>
00401034	E8 9E01000000	<b>CALL</b> <b>&amp;USER32.MessageBoxA</b>
00401036	6A 00	PUSH <b>0</b>
00401038	E8 BC01000000	<b>CALL</b> <b>&amp;KERNEL32.ExitProcess</b>
00401052	55	PUSH <b>EBP</b>
00401053	8BEC	MOU <b>EBP,ESP</b>
00401055	88C4 B0	ADD <b>ESP,-50</b>
00401058	C745 D0 30000000	
00401059	C745 D4 03000000	
00401060	C745 D8 37114000	
00401060	C745 DC 00000000	
00401074	C745 E0 00000000	
00401078	FF35 1C314000	
00401081	8F45 E4	
00401084	C745 F0 00000000	
00401088	C745 F4 00000000	
00401092	C745 F8 0000040000	
00401093	6A 01	
00401098	FF75 E4	
0040109E	E8 41010000	<b>CALL</b> <b>&amp;USER32.AddIconA</b>
004010A3	8945 E8	MOU <b>DWORD PTR SS:[EBP-18],EAX</b>
004010A6	8945 FC	MOU <b>DWORD PTR SS:[EBP-4],EAX</b>
004010A9	68 007F0000	PUSH <b>7F00</b>
004010B0	6A 00	
00402000	41 86 80 7C	
00402010	41 86 80 7E	
00402020	41 86 80 7F	
00402030	41 86 80 7E	
00402040	41 86 80 7F	
00402050	41 86 80 80	
00402060	41 86 80 80	
00402070	41 86 80 80	
00402080	41 86 80 80	
00402090	41 86 80 80	
004020A0	41 86 80 80	
004020B0	41 86 80 80	
004020C0	41 86 80 80	
004020D0	41 86 80 80	
004020E0	41 86 80 80	
004020F0	41 86 80 80	
00402100	41 86 80 80	
00402110	41 86 80 80	
00402120	41 86 80 80	
00402130	41 86 80 80	
00402140	41 86 80 80	
00402150	41 86 80 80	
00402160	41 86 80 80	
00402170	41 86 80 80	
00402180	41 86 80 80	
00402190	41 86 80 80	
004021A0	41 86 80 80	
004021B0	41 86 80 80	
004021C0	41 86 80 80	
004021D0	41 86 80 80	
004021E0	41 86 80 80	
004021F0	41 86 80 80	
00402200	41 86 80 80	
00402210	41 86 80 80	
00402220	41 86 80 80	
00402230	41 86 80 80	
00402240	41 86 80 80	
00402250	41 86 80 80	
00402260	41 86 80 80	
00402270	41 86 80 80	
00402280	41 86 80 80	
00402290	41 86 80 80	
004022A0	41 86 80 80	
004022B0	41 86 80 80	
004022C0	41 86 80 80	
004022D0	41 86 80 80	
004022E0	41 86 80 80	
004022F0	41 86 80 80	
00402300	41 86 80 80	
00402310	41 86 80 80	
00402320	41 86 80 80	
00402330	41 86 80 80	
00402340	41 86 80 80	
00402350	41 86 80 80	
00402360	41 86 80 80	
00402370	41 86 80 80	
00402380	41 86 80 80	
00402390	41 86 80 80	
004023A0	41 86 80 80	
004023B0	41 86 80 80	
004023C0	41 86 80 80	
004023D0	41 86 80 80	
004023E0	41 86 80 80	
004023F0	41 86 80 80	
00402400	41 86 80 80	
00402410	41 86 80 80	
00402420	41 86 80 80	
00402430	41 86 80 80	
00402440	41 86 80 80	
00402450	41 86 80 80	
00402460	41 86 80 80	
00402470	41 86 80 80	
00402480	41 86 80 80	
00402490	41 86 80 80	
004024A0	41 86 80 80	
004024B0	41 86 80 80	
004024C0	41 86 80 80	
004024D0	41 86 80 80	
004024E0	41 86 80 80	
004024F0	41 86 80 80	
00402500	41 86 80 80	
00402510	41 86 80 80	
00402520	41 86 80 80	
00402530	41 86 80 80	
00402540	41 86 80 80	
00402550	41 86 80 80	
00402560	41 86 80 80	
00402570	41 86 80 80	
00402580	41 86 80 80	
00402590	41 86 80 80	
004025A0	41 86 80 80	
004025B0	41 86 80 80	
004025C0	41 86 80 80	
004025D0	41 86 80 80	
004025E0	41 86 80 80	
004025F0	41 86 80 80	
00402600	41 86 80 80	
00402610	41 86 80 80	
00402620	41 86 80 80	
00402630	41 86 80 80	
00402640	41 86 80 80	
00402650	41 86 80 80	
00402660	41 86 80 80	
00402670	41 86 80 80	
00402680	41 86 80 80	
00402690	41 86 80 80	
004026A0	41 86 80 80	
004026B0	41 86 80 80	
004026C0	41 86 80 80	
004026D0	41 86 80 80	
004026E0	41 86 80 80	
004026F0	41 86 80 80	
00402700	41 86 80 80	
00402710	41 86 80 80	
00402720	41 86 80 80	
00402730	41 86 80 80	
00402740	41 86 80 80	
00402750	41 86 80 80	
00402760	41 86 80 80	
00402770	41 86 80 80	
00402780	41 86 80 80	
00402790	41 86 80 80	
004027A0	41 86 80 80	
004027B0	41 86 80 80	
004027C0	41 86 80 80	
004027D0	41 86 80 80	
004027E0	41 86 80 80	
004027F0	41 86 80 80	
00402800	41 86 80 80	
00402810	41 86 80 80	
00402820	41 86 80 80	
00402830	41 86 80 80	
00402840	41 86 80 80	
00402850	41 86 80 80	
00402860	41 86 80 80	
00402870	41 86 80 80	
00402880	41 86 80 80	
00402890	41 86 80 80	
004028A0	41 86 80 80	
004028B0	41 86 80 80	
004028C0	41 86 80 80	
004028D0	41 86 80 80	
004028E0	41 86 80 80	
004028F0	41 86 80 80	
00402900	41 86 80 80	
00402910	41 86 80 80	
00402920	41 86 80 80	
00402930	41 86 80 80	
00402940	41 86 80 80	
00402950	41 86 80 80	
00402960	41 86 80 80	
00402970	41 86 80 80	
00402980	41 86 80 80	
00402990	41 86 80 80	
004029A0	41 86 80 80	
004029B0	41 86 80 80	
004029C0	41 86 80 80	
004029D0	41 86 80 80	
004029E0	41 86 80 80	
004029F0	41 86 80 80	
00402A00	41 86 80 80	
00402A10	41 86 80 80	
00402A20	41 86 80 80	
00402A30	41 86 80 80	
00402A40	41 86 80 80	
00402A50	41 86 80 80	
00402A60	41 86 80 80	
00402A70	41 86 80 80	
00402A80	41 86 80 80	
00402A90	41 86 80 80	
00402AA0	41 86 80 80	
00402AB0	41 86 80 80	
00402AC0	41 86 80 80	
00402AD0	41 86 80 80	
00402AE0	41 86 80 80	
00402AF0	41 86 80 80	
00402B00	41 86 80 80	
00402B10	41 86 80 80	
00402B20	41 86 80 80	
00402B30	41 86 80 80	
00402B40	41 86 80 80	
00402B50	41 86 80 80	
00402B60	41 86 80 80	
00402B70	41 86 80 80	
00402B80	41 86 80 80	
00402B90	41 86 80 80	
00402BA0	41 86 80 80	
00402BB0	41 86 80 80	
00402BC0	41 86 80 80	
00402BD0	41 86 80 80	
00402BE0	41 86 80 80	
00402BF0	41 86 80 80	
00402C00	41 86 80 80	
00402C10	41 86 80 80	
00402C20	41 86 80 80	
00402C30	41 86 80 80	
00402C40	41 86 80 80	
00402C50	41 86 80 80	
00402C60	41 86 80 80	
00402C70	41 86 80 80	
00402C80	41 86 80 80	
00402C90	41 86 80 80	
00402CA0	41 86 80 80	
00402CB0	41 86 80 80	
00402CC0	41 86 80 80	
00402CD0	41 86 80 80	
00402CE0	41 86 80 80	
00402CF0	41 86 80 80	
00402D00	41 86 80 80	
00402D10	41 86 80 80	
00402D20	41 86 80 80	
00402D30	41 86 80 80	
00402D40	41 86 80 80	
00402D50	41 86 80 80	
00402D60	41 86 80 80	
00402D70	41 86 80 80	
00402D80	41 86 80 80	
00402D90	41 86 80 80	
00402DA0	41 86 80 80	
00402DB0	41 86 80 80	
00402DC0	41 86 80 80	
00402DD0	41 86 80 80	
00402DE0	41 86 80 80	
00402DF0	41 86 80 80	
00402E00	41 86 80 80	
00402E10	41 86 80 80	
00402E20	41 86 80 80	
00402E30	41 86 80 80	
00402E40	41 86 80 80	
00402E50	41 86 80 80	
00402E60	41 86 80 80	
00402E70	41 86 80 80	
00402E80	41 86 80 80	
00402E90	41 86 80 80	
00402EA0	41 86 80 80	
00402EB0	41 86 80 80	
00402EC0	41 86 80 80	
00402ED0	41 86 80 80	
00402EE0	41 86 80 80	
00402EF0	41 86 80 80	
00402F00	41 86 80 80	
00402F10	41 86 80 80	
00402F20	41 86 80 80	
00402F30	41 86 80 80	
00402F40	41 86 80 80	
00402F50	41 86 80 80	
00402F60	41 86 80 80	
00402F70	41 86 80 80	
00402F80	41 86 80 80	
00402F90	41 86 80 80	
00402FA0	41 86 80 80	
00402FB0	41 86 80 80	
00402FC0	41 86 80 80	
00402FD0	41 86 80 80	
00402FE0	41 86 80 80	
00402FF0	41 86 80 80	
00402000	41 86 80 80	
00402010	41 86 80 80	
00402020	41 86 80 80	
00402030	41 86 80 80	
00402040	41 86 80 80	
00402050	41 86 80 80	
00402060	41 86 80 80	
00402070	41 86 80 80	
00402080	41 86 80 80	
00402090	41 86 80 80	
004020A0	41 86 80 80	
004020B0	41 86 80 80	
004020C0	41 86 80 80	
004020D0	41 86 80 80	
004020E0	41 86 80 80	
004020F0	41 86 80 80	
00402100	41 86 80 80	
00402110	41 86 80 80	
00402120	41 86 80 80	
00402130	41 86 80 80	
00402140	41 86 80 80	
00402150	41 86 80 80	
00402160	41 86 80 80	
00402170	41 86 80 80	
00402180	41 86 80 80	
00402190	41 86 80 80	
004021A0	41 86 80 80	
004021B0	4	

Bam ... we land immediately at EP.

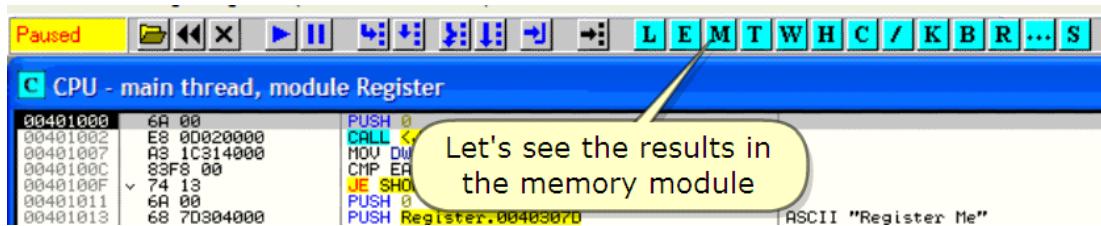
No "nags" about bad exe's anymore ...

And the opcodes in the dump window are shown immediately ...

우리는 즉시 EP에 도착했다.

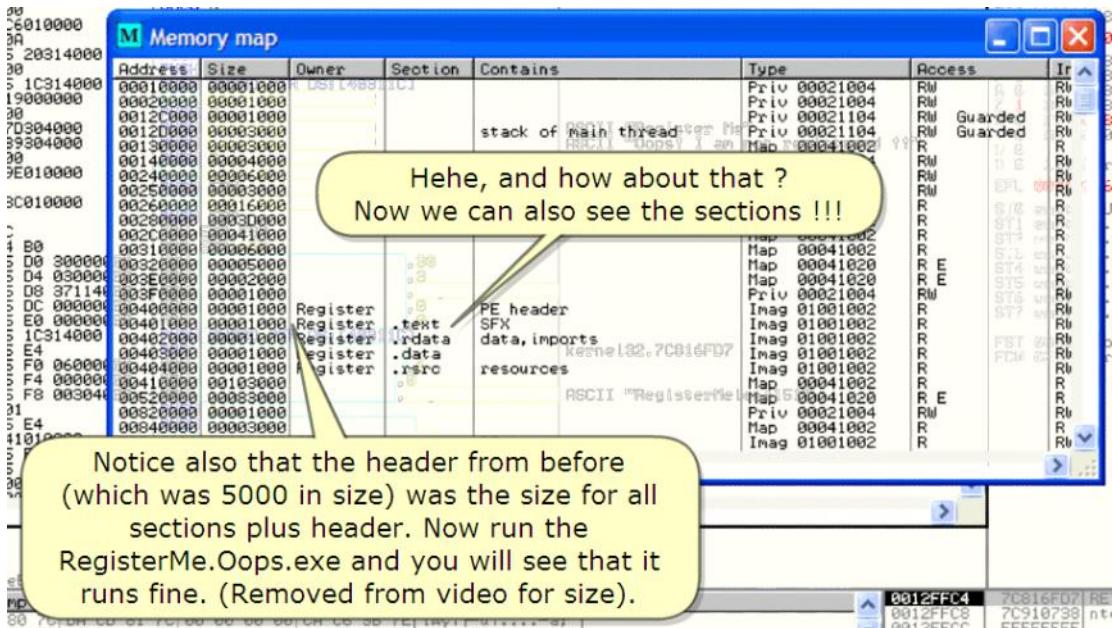
더 이상 Nags은 없다.

그리고 opcodes는 dump window에 즉시 보여진다.



Let's see the results in the memory module.

Memory module의 결과를 보자.



Hehe, and how about that ?

Now we can also see the sections !!!

헤헤, 어때?

이제 우리는 sections을 볼 수 있다.

Notice also that the header from before (which was 5000 in size) was the size for all sections plus header.

알린다. 이전의 header는(5000 size였다) section header size가 plus 되어 있었다.

Now run the RegisterMe.Ooops.exe and you will see that it runs fine. (Removed from video for size).

이제 RegisterMe.Ooops.exe를 실행하고 너는 잘 실행되는 것을 볼 수 있다. (video size 때문에 삭제했다)

## 9. Conclusion

In this part 3, we learned to kill some easy nags. There are more possibilities and other solutions to kill nags of course.

이번 part 3에서, 우리는 약간의 쉬운 nag을 없애는 것을 배웠다. 그것에는 많은 가능성이 있고 nags를 kill하는 다른 해결책이 물론 있다.

We also learned some basic stuff about the PE file structure, especially the PE header.

우리는 기본적인 PE file 구조의 구성을 특별히 PE header에서 배웠다.

My goal was to prove that the reverser needs to know the basics about the PE header, I think you will agree on that.

나의 목표는 reverser가 알아야 할 기본적인 PE header를 보여주는 것이었다. 네가 동의할 거라 생각한다.

I hope you understood everything fine and I also hope someone somewhere learned something from this.

네가 모든 것을 좋게 이해했고 또한 누구든지 어느 부분이던지 이것에서 뭔가를 배웠으면 한다.

See me back in Part 4 in this series ;)

나는 이 series의 Part 4에서 돌아오겠다.

The other parts are available at

다른 parts는 사용 가능하다.

<http://tinyurl.com/27dzdn> (tuts4you)

<http://tinyurl.com/r89zq> (SnD Filez)

<http://tinyurl.com/l6srv> (fixdown)

Regards to all and especially to you for taking the time to look at this tutorial.

Lena151 (2006, updated 2007)

모두에게 안부를 전하고 특별히 이 tutorial에 시간을 투자해준 너에게 감사한다.