악성코드 분석 보고서

(sand-reversingwithlena-tutorials)

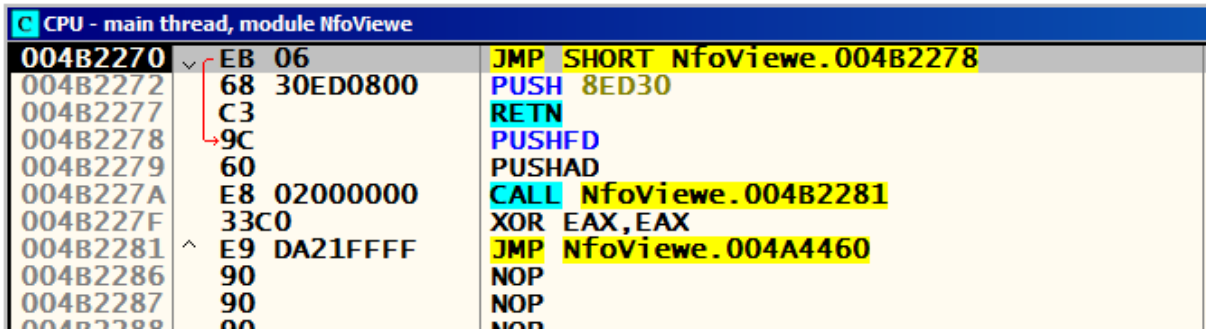
2025.09.29

23

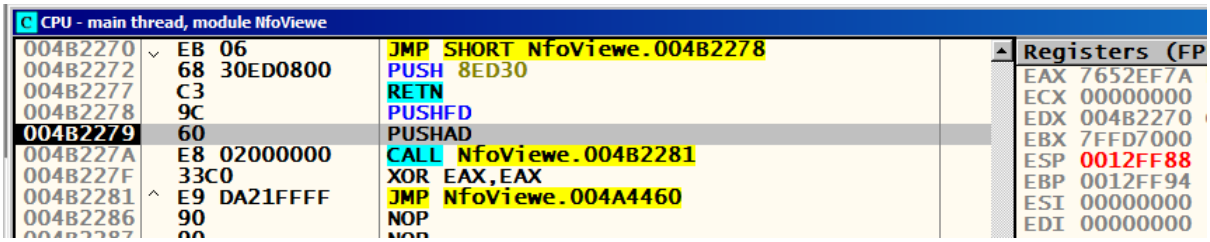
"Stolen Byte"란 원래의 바이너리에서는 사라졌지만, 패커의 일부분처럼 동작하기 위해 패커 코드내에 사용된 코드를 말한다. OEP를 찾은 뒤 바이너리를 덤프하면 이런 바이트들은 덤프된 바이너리에 존재하지 않거나 원래 있어야 할 자리에 존재하지 않고 이상한 자리에 존재한다. 이렇게 돼 버리면 바이너리를 실행하면 크래시가 발생 될 것이다. 즉 일종의 안티 디버깅 기법이라 할 수 있는 것이다.

그러나 패킹된 상태에서는 크래시가 발생하지 않는데 이유는 이런 "Stolen Byte" 들이 OEP에 도달하기 전에 패커 코드 내에서 사용되기 때문이다.

1.NfoViewer.exe

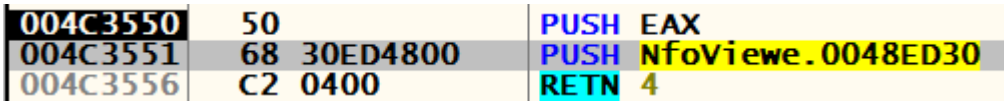


ESP가 바뀔 때를 이용하여 OEP를 찾아준다.



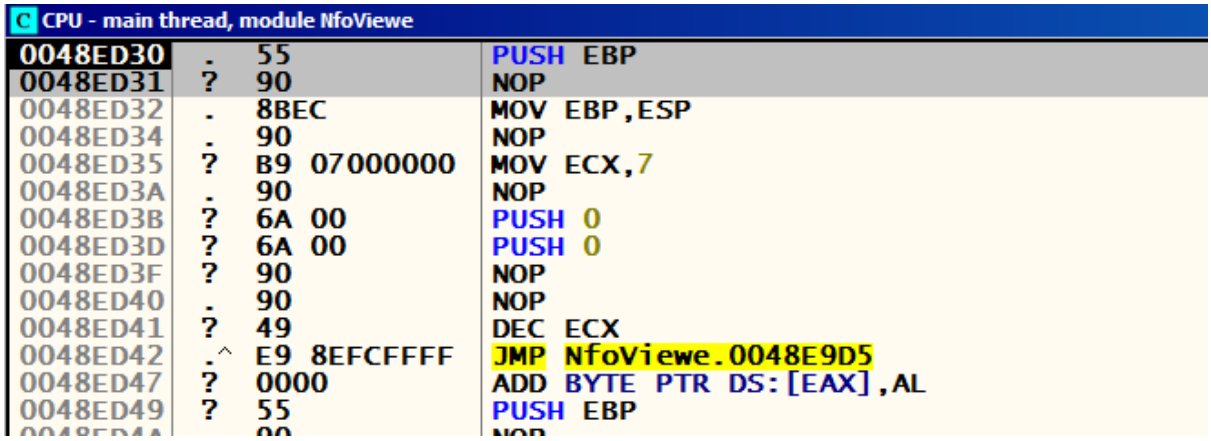
해당 부분에서 바뀌는 걸 알 수 있고 ESP값에 HW BP를 걸어준다.

Breakpoint > Hardware, on access > Dword

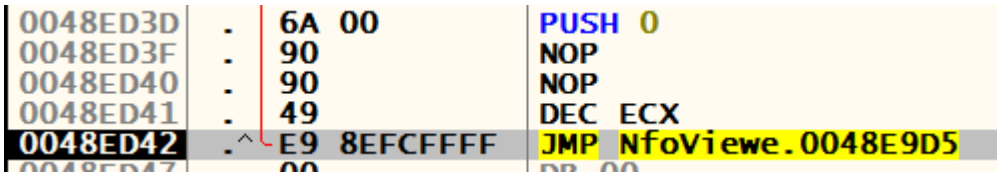


'F9'로 실행하다보면 해당 부분에서 멈추는 걸 볼 수 있다.

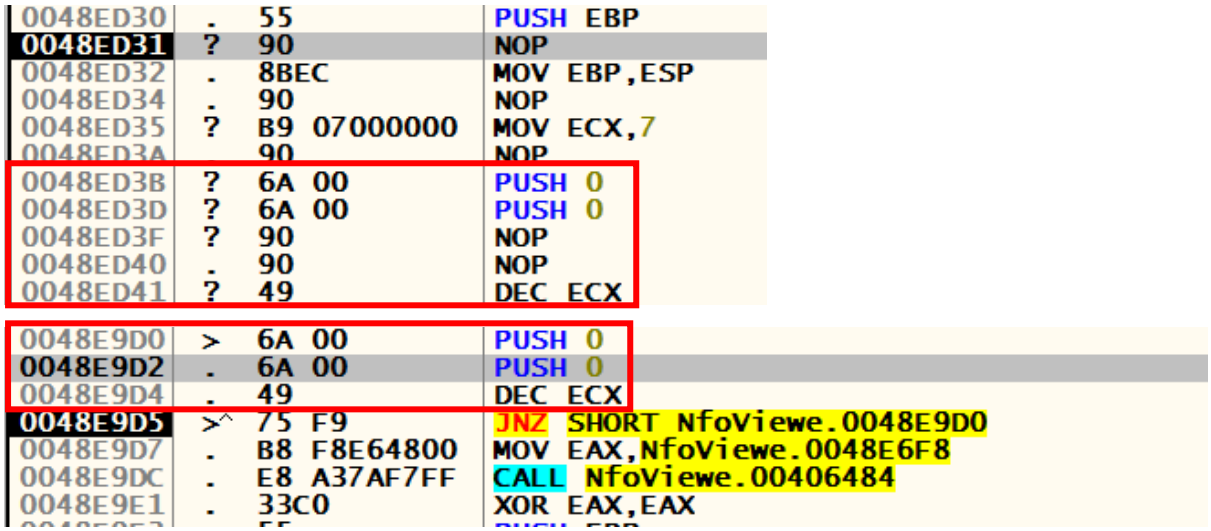
"PUSH XXXXXXXX + RETURN" 형태의 명령어가 나타나면 이는 "JMP XXXXXXXX"와 동일한 의미로 해석할 수 있다. 따라서 위 사진의 명령에서는 "48ED30" 주소로 분기되는 걸 볼 수 있다.



이부분이 OEP라고 유추할 수 있지만 좀 더 살펴본다.



'F8'을 이용해서 내려가다보면 JMP가 있는 걸 볼 수 있다. 한 번 더 눌러 넘어가본다.



해당 부분이 아까와 비슷한 걸 알 수 있다.

이렇게 NOP이라는 쓰레기 값이 코드 중간중간 많이 들어가는 걸 볼 수 있다. 이걸 stolen bytes(도난 바이트)라고한다.

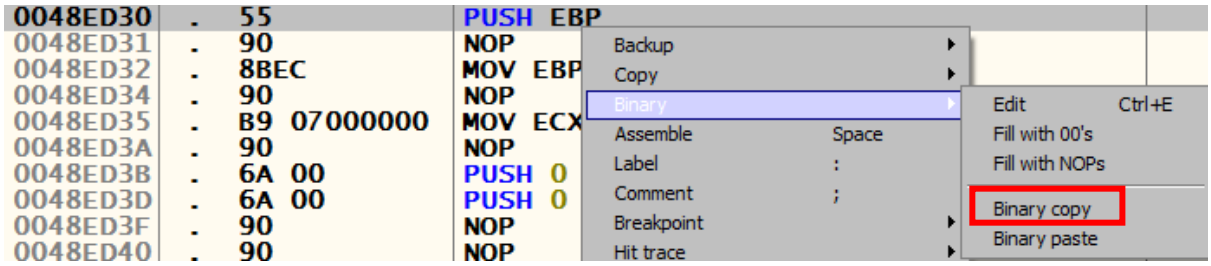
0x0048DE35에서 ECX에 7이 저장되는 걸 알 수 있고 ECX가 1씩 감소하는 걸 볼 수 있다. 그리고 PUSH 0이 2번씩 들어가고 ECX가 0이 될 때까지 발생하니까 PUSH 0이 총 14번 실행되는 걸 알 수 있다. 이러한 의미없는 스택 채우기는 디버거로 보는 사람에게 OEP(실제 시작 지점)까지의 추적을 헷갈리게 하려는 트릭으로 볼 수 있다.

```
0048ED30   .   55              PUSH EBP
0048ED31   ?   90              NOP
0048ED32   .   8BEC            MOV EBP,ESP
0048ED34   .   90              NOP
0048ED35   ?   B9 07000000     MOV ECX,7
0048ED3A   .   90              NOP
0048ED3B   ?   6A 00           PUSH 0
0048ED3D   ?   6A 00           PUSH 0
0048ED3F   ?   90              NOP
0048ED40   .   90              NOP
0048ED41   ?   49              DEC ECX
```

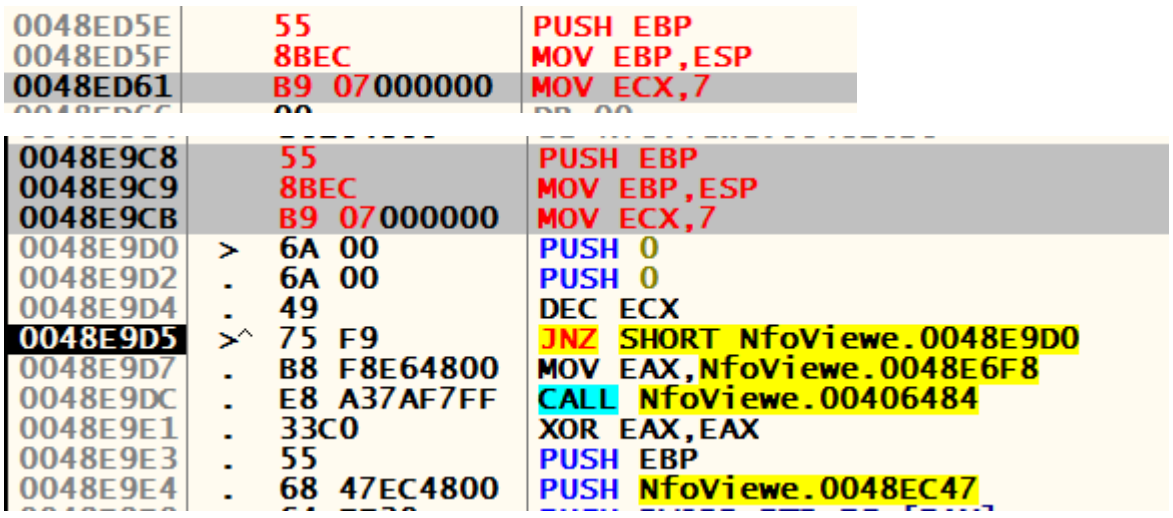
0x0048E9D0위에 잉여 공간이 있는 걸 알 수 있고 0x0048ED42에서 밑으로 내려가 보면 잉여 공간이 있는 걸 알 수 있다.

이 부분들을 이용해서 원래의 OEP를 재건할려고한다.

0x0048ED42 밑에 빨간 네모의 코드를 복사해서 넣어준다.



※ 코드의 크기를 주의해가면서 붙여넣기 해준다.

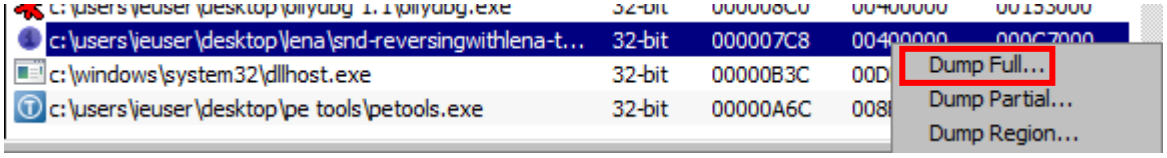


이렇게 다 붙여넣기 해준 코드를 다시 복사해준 후 0x0048E9D0 위에 있는 잉여공간에 넣어준다.

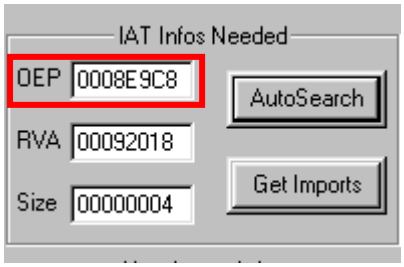
```
0048E9C4      D0E64800      DD NfoViewe.0048E6D0
0048E9C8      55            PUSH EBP
0048E9C9      8BEC          MOV EBP,ESP
0048E9CB      B9 07000000   MOV ECX,7
0048E9D0  >   6A 00         PUSH 0
0048E9D2  .   6A 00         PUSH 0
0048E9D4  .   49            DEC ECX
0048E9D5  >^  75 F9         JNZ SHORT NfoViewe.0048E9D0
0048E9D7  .   B8 F8E64800   MOV EAX,NfoViewe.0048E6F8
0048E9DC  .   E8 A37AF7FF   CALL NfoViewe.00406484
0048E9E1  .   33C0          XOR EAX,EAX
```

이렇게 새로운 OEP를 만들었으니까 해당 주소를 이제 OEP로 인식시켜야 한다.

이렇게 한 후 이번에는 PETools를 이용하여 덤프를 만들것이다.



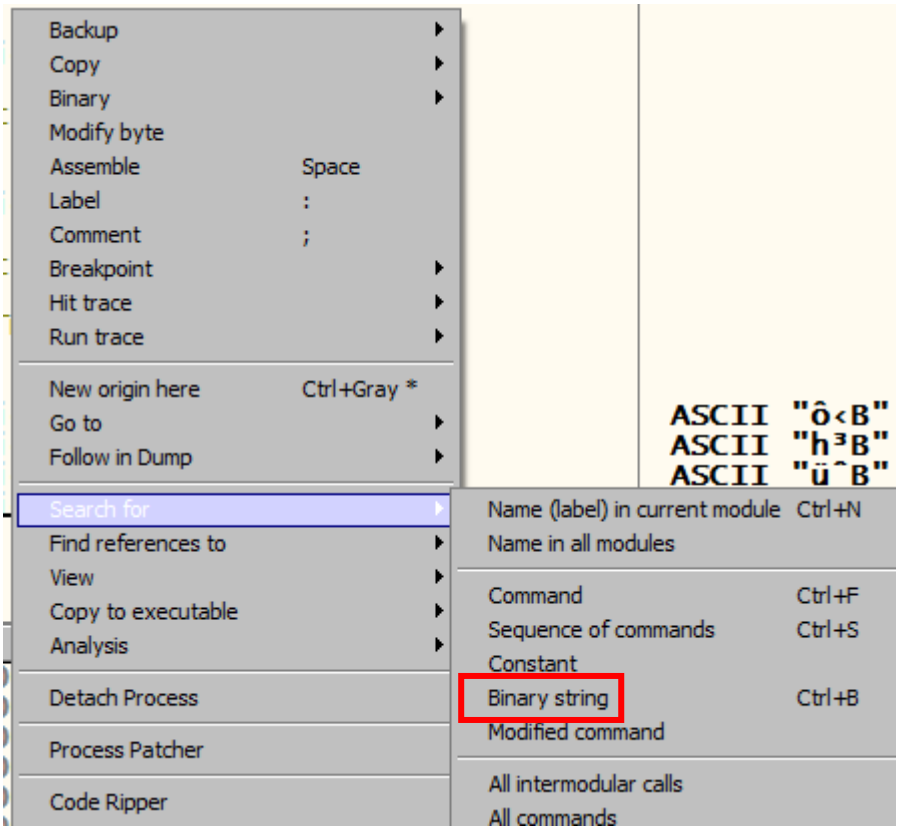
Dump Full을 눌러서 덤프를 만들어준다.

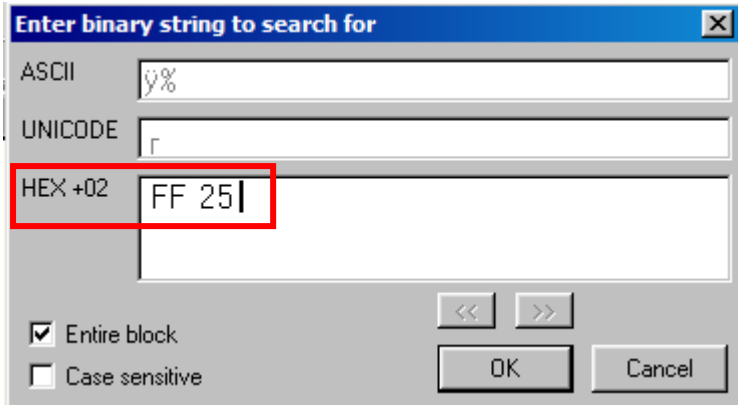


ImpRec를 이용해서 IAT를 수정해준다. 새로 만든 OEP를 넣고 'AutoSearch'를 눌러 IAT를 찾아준다.

```
Address   Hex dump
00492018  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00492028  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00492038  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00492048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00492058  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

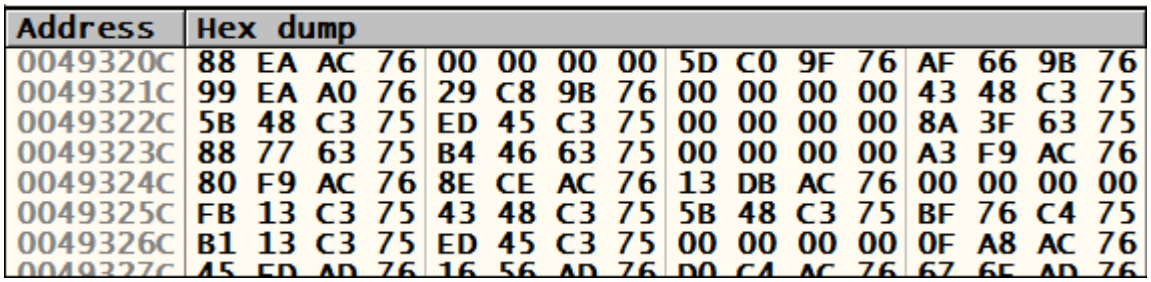
0x00492018이 IAT의 시작 부분이다. 올리디버그를 이용해서 해당 부분이 맞는지 찾아보면 00으로 가득차서 아무것도 없는 걸 볼 수 있다. 아마도 패커에서 임포트 테이블을 조작한 듯 하다. 다른 방법을 통해서 임포트 테이블을 찾아야 하는데 "jump" 테이블을 기억할지 모르겠지만, 이 테이블은 opcode "FF 25"로 시작한다. 따라서 "FF 25"를 바이너리 검색 하는 것 또한 하나의 방법이다.





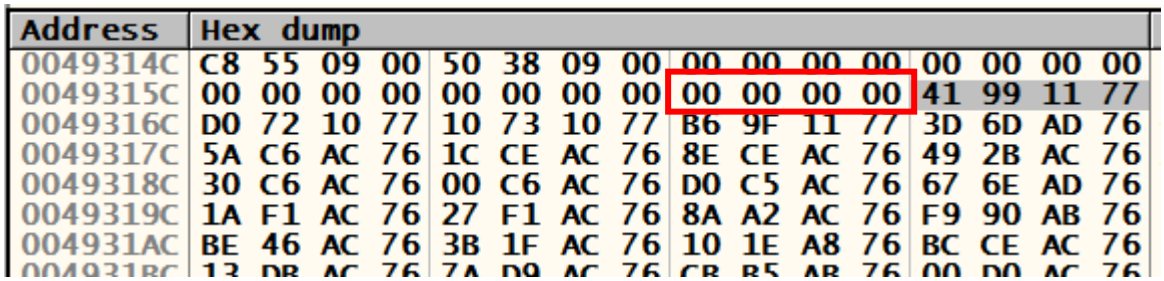

```
0040120C  $- FF25 0C324900 JMP DWORD PTR DS:[49320C]   kernel32.CloseHandle
00401212     8BC0         MOV EAX,EAX
00401214  $- FF25 08324900 JMP DWORD PTR DS:[493208]   kernel32.CreateFileA
0040121A     8BC0         MOV EAX,EAX
0040121C  $- FF25 04324900 JMP DWORD PTR DS:[493204]   kernel32.GetFileType
00401222     8BC0         MOV EAX,EAX
00401224  $- FF25 00324900 JMP DWORD PTR DS:[493200]   kernel32.GetFileSize
0040122A     8BC0         MOV EAX,EAX
0040122C  $- FF25 FC314900 JMP DWORD PTR DS:[4931FC]   kernel32.GetStdHandle
00401232     8BC0         MOV EAX,EAX
00401234  .- FF25 F8314900 JMP DWORD PTR DS:[4931F8]   kernel32.RaiseException
0040123A     8BC0         MOV EAX,EAX
0040123C  $- FF25 F4314900 JMP DWORD PTR DS:[4931F4]   kernel32.ReadFile
00401242     8BC0         MOV EAX,EAX
00401244  .- FF25 F0314900 JMP DWORD PTR DS:[4931F0]   kernel32.RtlUnwind
0040124A     8BC0         MOV EAX,EAX
0040124C  $- FF25 EC314900 JMP DWORD PTR DS:[4931EC]   kernel32.SetEndOfFile
00401252     8BC0         MOV EAX,EAX
00401254  $- FF25 E8314900 JMP DWORD PTR DS:[4931E8]   kernel32.SetFilePointer
0040125A     8BC0         MOV EAX,EAX
0040125C  $- FF25 E4314900 JMP DWORD PTR DS:[4931E4]   kernel32.UnhandledExceptionFil
00401262     8BC0         MOV EAX,EAX
00401264  $- FF25 E0314900 JMP DWORD PTR DS:[4931E0]   kernel32.WriteFile
0040126A     8BC0         MOV EAX,EAX
0040126C  $- FF25 20324900 JMP DWORD PTR DS:[493220]   user32.CharNextA
00401272     8BC0         MOV EAX,EAX
00401274  .- FF25 DC314900 JMP DWORD PTR DS:[4931DC]   kernel32.ExitProcess
```

이렇게 들어가보면 Import Stub이 있는 걸 볼 수 있다.



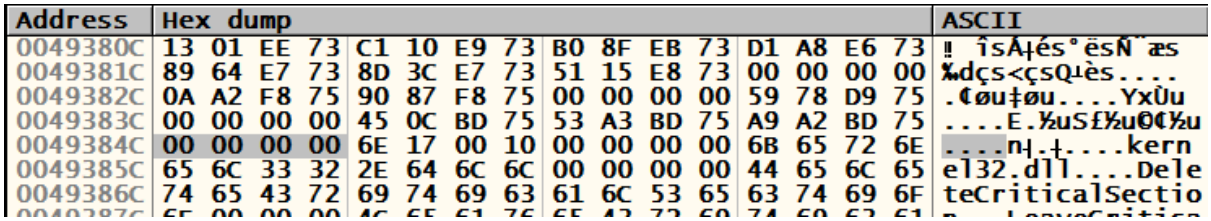
Address	Hex dump
0049320C	88 EA AC 76 00 00 00 00 5D C0 9F 76 AF 66 9B 76
0049321C	99 EA A0 76 29 C8 9B 76 00 00 00 00 43 48 C3 75
0049322C	5B 48 C3 75 ED 45 C3 75 00 00 00 00 8A 3F 63 75
0049323C	88 77 63 75 B4 46 63 75 00 00 00 00 A3 F9 AC 76
0049324C	80 F9 AC 76 8E CE AC 76 13 DB AC 76 00 00 00 00
0049325C	FB 13 C3 75 43 48 C3 75 5B 48 C3 75 BF 76 C4 75
0049326C	B1 13 C3 75 ED 45 C3 75 00 00 00 00 0F A8 AC 76

덤프에서 49320C를 들어가서 보면 IAT인 걸 볼 수 있다.



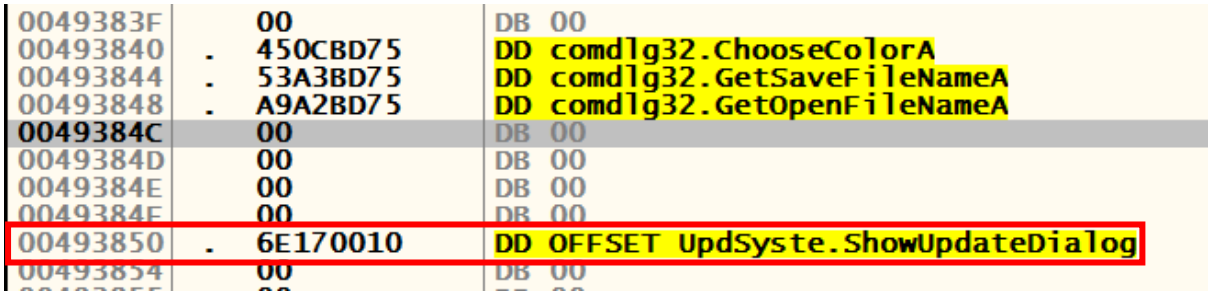
Address	Hex dump
0049314C	C8 55 09 00 50 38 09 00 00 00 00 00 00 00 00 00
0049315C	00 00 00 00 00 00 00 00 00 00 00 00 41 99 11 77
0049316C	D0 72 10 77 10 73 10 77 B6 9F 11 77 3D 6D AD 76
0049317C	5A C6 AC 76 1C CE AC 76 8E CE AC 76 49 2B AC 76
0049318C	30 C6 AC 76 00 C6 AC 76 D0 C5 AC 76 67 6E AD 76
0049319C	1A F1 AC 76 27 F1 AC 76 8A A2 AC 76 F9 90 AB 76
004931AC	BE 46 AC 76 3B 1F AC 76 10 1E A8 76 BC CE AC 76

위로 올라가서 시작 부분을 찾아보면 0x00493164인 걸 알 수 있다.



Address	Hex dump	ASCII
0049380C	13 01 EE 73 C1 10 E9 73 B0 8F EB 73 D1 A8 E6 73	‼ îsÁ┼és°ësÑ¨æs
0049381C	89 64 E7 73 8D 3C E7 73 51 15 E8 73 00 00 00 00	‰dçs<çsQ↕ès....
0049382C	0A A2 F8 75 90 87 F8 75 00 00 00 00 59 78 D9 75	.¢øu‡øu....YxÙu
0049383C	00 00 00 00 45 0C BD 75 53 A3 BD 75 A9 A2 BD 75	....E.½uS£½u©¢½u
0049384C	00 00 00 00 6E 17 00 10 00 00 00 00 6B 65 72 6E	....n┤.┼....kern
0049385C	65 6C 33 32 2E 64 6C 6C 00 00 00 00 44 65 6C 65	el32.dll....Dele
0049386C	74 65 43 72 69 74 69 63 61 6C 53 65 63 74 69 6F	teCriticalSectio

끝 부분을 찾아보면 0x0049384C인 걸 알 수 있다. 이게 끝부분이 맞는지 코드에서 찾아본다.

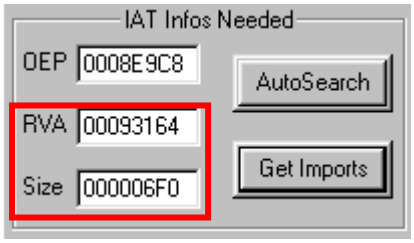


```
0049383F      00              DB 00
00493840   .  450CBD75        DD comdlg32.ChooseColorA
00493844   .  53A3BD75        DD comdlg32.GetSaveFileNameA
00493848   .  A9A2BD75        DD comdlg32.GetOpenFileNameA
0049384C      00              DB 00
0049384D      00              DB 00
0049384E      00              DB 00
0049384F      00              DB 00
00493850   .  6E170010        DD OFFSET UpdSyste.ShowUpdateDialog
00493854      00              DB 00
```

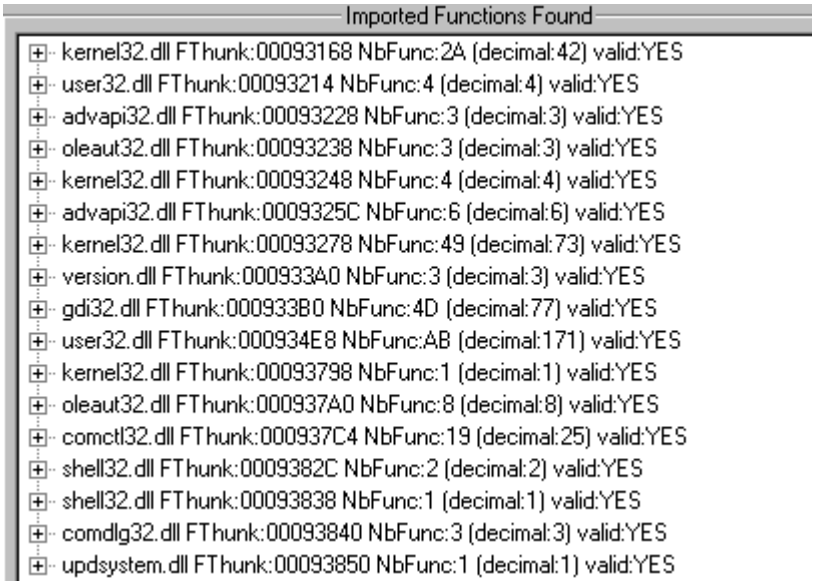
코드에서 보면 아래 하나 함수가 더 있는 걸 볼 수 있다. 즉, 끝 부분은 0x0049384C이 아니라 0x00493854인 걸 알 수 있다.

"Updsystem.dll(어플리케이션에서 제공하는 dll)" 로부터 임포트 되는 "ShowUpdateDialog" 함수 주소이다. "Import REC"에서 해당 dll을 임포트 하지 않아도 프로그램이 실행되면서 자동으로 임포트 될 것이다.

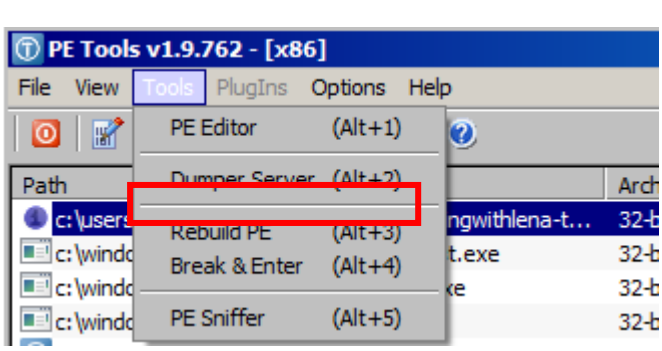
처음 시작 부분은 493164 끝 부분은 493854이다. 그럼 크기는 6F0이 나온다.

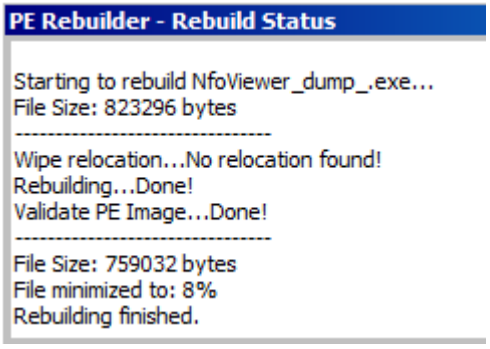


값을 넣고 'Get Imports'를 눌러준다.

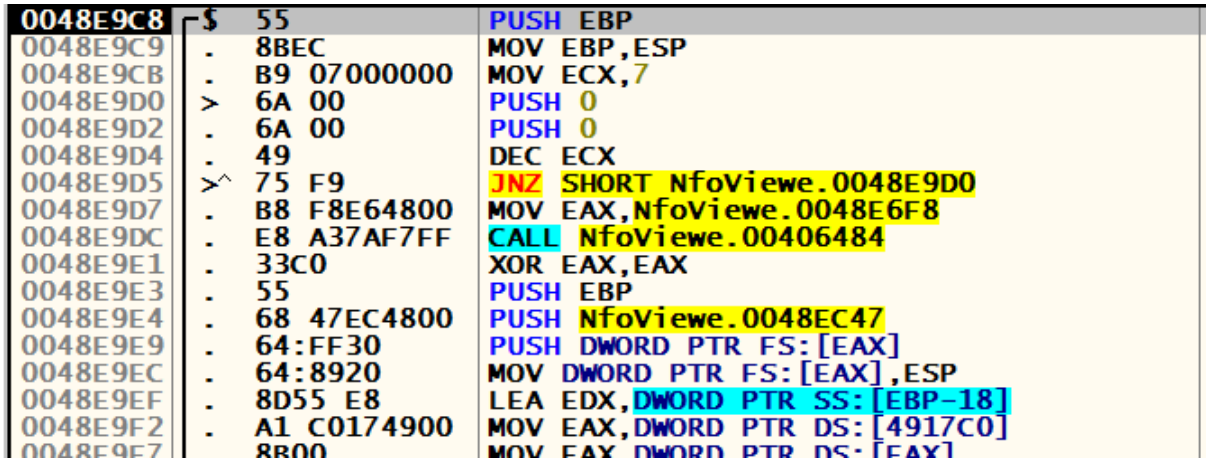


전부 YES로 나와 잘 찾은 걸 볼 수 있다. 이제 이전에 덤프 뜬 바이너리를 대상으로 IAT를 변경하면 된다.





PE Tools를 이용해서 리빌드를 해준다.



다시 실행해보면 OEP가 제대로 된 곳에서 실행되고 제대로 실행되는 걸 볼 수 있다.