기 술 문 서

Unpacking

박지훈

sonicpj@nate.com

ezbeat.tistory.com

Content

1. 패킹

1.1 패킹이란

먼저 언패킹을 하기 전에 패킹이 무엇인지부터 알아보겠습니다. 패킹이라는 것은 실행 압축이라고도 말할 수 있습니다. 실행 압축이라는 것은 실행 파일 내부에 코드를 압축/해제 하는 코드를 포함하고 있어서 평소에는 코드가 압축 상태로 존재하다가 파일을 실행 시키면 해당 파일이 메모리에 올라가게 되면서 메모리에서 압축을 해제 시킨 후 파일을 실행시키는 기술입니다. 실행 압축이 될 수 있는 파일은 PE파일 뿐입니다. 여기서 잠깐 PE파일이 어떠한 확장자를 뜻하는 것인지 보고 넘어가겠습니다.

PE파일의 종류
EXE, SCR, DLL, OCX, SYS, OBJ

[그림 1]

위와 같은 확장자를 가진 파일들이 PE파일이 되겠습니다.
즉, JPG같은 그림파일은 패킹을 할 수 없습니다.
패킹을 하는 도구를 패커라고 부르는데 이러한 패커 중 하나를 가지고 패킹이 안 된 파일을 패킹을 해봐서 패킹 되었을 때와 안 되었을 때의 코드를 봐보겠습니다. 먼저 패킹을 하지 않은 파일입니다.

[그림 2]

빨간색 네모 친 부분을 보시면 패킹을 하지 않았기 때문에 무엇으로 컴파일을 했는지 확인이 가능했습니다. 이번에는 가장 기본적인 패커 인 UPX패커로 패킹을 하겠습니다.

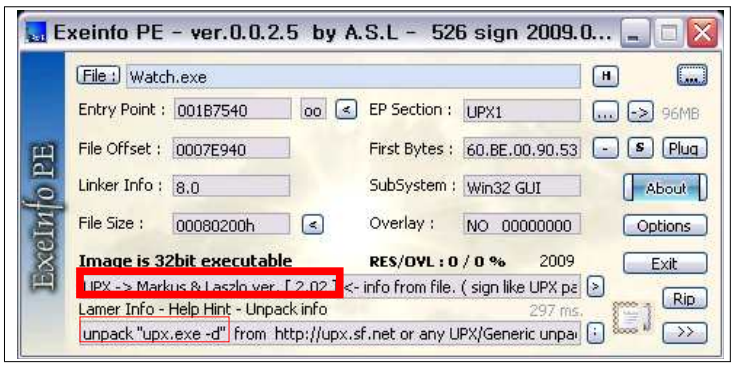
```
C:₩Documents and Settings₩EZ2DJ₩바탕 화면>upx Watch.exe
                       Ultimate Packer for eXecutables
          Copyright (C) 1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006
UPX 2.02w       Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 13th 2006

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   1785856 ->    524800   29.39%    win32/pe     Watch.exe

Packed 1 file.
```

[그림 3]

단순히 패킹이 되었다는 것을 보여주는 그림입니다. 전 UPX패커를 system32폴더에 넣어두어서 사용하고 있습니다.
다시 PE확인 툴로 다시 해당 파일을 보겠습니다.



[그림 4]

무엇으로 컴파일 되었는지에 대한 정보는 안 나와 있고 UPX로 패킹이 되어 있다고 나타나 있습니다. 전 PEID도 사용하지만 이 툴도 많이 사용합니다. 아래 빨간색 네모 친 부분을 보면 해당 파일을 무엇으로 언패킹을 해야 하는지 까지 어느 정도 힌트를 주기 때문입니다.

이제 패킹을 하는 것은 우리의 목적이 아니었고 해당 파일의 코드가 어떻게 변했는지 알아보려고 했으므로 해당 파일을 Hex Edit으로 봐보겠습니다. Hex Edit으로 볼 때 PE Header부분에서 Section Header부분 만 봐보겠습니다. 왜냐하면 패킹을 해도 결국은 윈도우에서 실행되어야 합니다. 즉, 실행되려면 PE구조를 깨서는 안 됩니다. 그렇기 때문에 PE 형식은 그대로 유지하고 있으면서 코드 부분만 압축을 하는 형식이기 때문에 Section에 대한 정보를 가지고 있는 Section Header만 보는 것입니다.

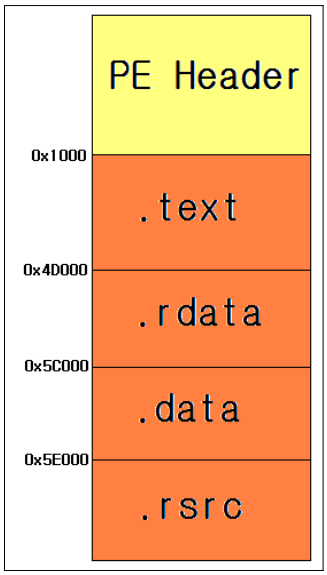
패킹을 하지 않은 파일의 Section Header부분을 봐보겠습니다.

```
000001e0h: 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 ; .........text...
000001f0h: 43 BB 04 00 00 10 00 00 00 C0 04 00 00 10 00 00 ; C?......?.....
00000200h: 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ; ............ ..`
00000210h: 2E 72 64 61 74 61 00 00 4E EA 00 00 00 D0 04 00 ; .rdata..N?..?.
00000220h: 00 F0 00 00 00 D0 04 00 00 00 00 00 00 00 00 00 ; .?..?.........
00000230h: 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 ; ....@..@.data...
00000240h: E0 34 00 00 00 C0 05 00 00 20 00 00 00 C0 05 00 ; ?...?.. ...?.
00000250h: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 ; ............@..?
00000260h: 2E 72 73 72 63 00 00 00 40 53 15 00 00 00 06 00 ; .rsrc...@S......
00000270h: 00 60 15 00 00 E0 05 00 00 00 00 00 00 00 00 00 ; .`...?.........
00000280h: 00 00 00 00 40 00 00 40 00 00 00 00 00 00 00 00 ; ....@..@........
```

[그림 5]

PE구조에 대한 문서는 많이 있으므로 잘 이해가 안되시는 분들은 PE구조를 보시고 오시면 되겠습니다.

이제 위 Section Header를 가지고 Section부분을 나타내보겠습니다.
총 4개의 섹션이 존재하며 메모리상에서가 아닌 파일 상에서만 나타내도록 하겠습니다.
(패딩 부분은 그림에서 뺐습니다.)



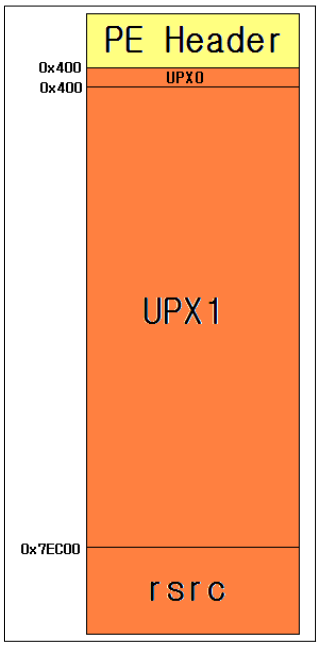
[그림 6]

4개의 섹션으로 파일이 구성되어 있는 것을 보실 수 있습니다. 이제 UPX패킹을 한 PE파일의 Section Header부분을 봐보겠습니다.

```
000001e0h: 00 00 00 00 00 00 00 00 55 50 58 30 00 00 00 00 ; ........UPX0....
000001f0h: 00 80 13 00 00 10 00 00 00 00 00 00 00 04 00 00 ; .□..............
00000200h: 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 E0 ; ............□..?
00000210h: 55 50 58 31 00 00 00 00 00 F0 07 00 00 90 13 00 ; UPX1.....?..?.
00000220h: 00 E8 07 00 00 04 00 00 00 00 00 00 00 00 00 00 ; .?..............
00000230h: 00 00 00 00 40 00 00 E0 2E 72 73 72 63 00 00 00 ; ....@..?rsrc...
00000240h: 00 20 00 00 00 80 1B 00 00 16 00 00 00 EC 07 00 ; . ...□.......?.
00000250h: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 ; ............@..?
```

[그림 7]

위 정보를 가지고 마찬가지로 그림으로 표현해 보겠습니다.



[그림 8]

[그림 6]에서 본 것과는 차이가 있다는 것을 바로 느끼실 수가 있을 것입니다.
먼저 섹션의 개수부터가 달려졌고 섹션의 이름도 바뀌었습니다.
패커의 종류도 엄청나게 다향한테 각 패커마다 실행 압축된 파일의 모양은 전부 다릅니다.)

※UPX패킹을 하게 되면 UPX1섹션에 압축된 원본 소스와 압축 해제 소스가 존재하며 해당 파일을 실행 시키는 순간 UPX0섹션에 풀어버리게 됩니다. 그래서 UPX0섹션의 크기를 보시면 아시겠지만 비워둔 것을 볼 수 있습니다.

1.2 패킹을 하는 이유

패킹이 무엇인지 위에서 간략하게 보고 와봤습니다. 그러면 왜 패킹을 하는 것일까요?
컴퓨터 입장에서 보자면 압축이 안 된 코드를 바로 메모리에 올리고 명령어들을 CPU로 보내서 처리하면 될 것을 코드를 압축 시키고 메모리에서 다시 해제를 한 다음 처리하는 번거로움이 있을 탠데 말이죠.

패킹을 하는 이유로는 크게 2가지를 생각해 볼 수 있습니다.
첫 번째는 방대한 파일의 크기를 줄일 수 있다는 것입니다. 알집이나 빵집과 같은 경우는 압축을 하게 되면 압축을 당한 파일은 그 상태로는 실행이 불가능 합니다. 하지만 실행압축을 하게 되면 실행이 가능한 상태로 파일이 압축이 되는 것입니다. 더 이상 설명은 하지 않고 파일을 패킹 해 봄으로써 용량 체크만 하고 넘어가도록 하겠습니다.
대상 파일은 다들 실험 해볼 수 있도록 전부 가지고 있는 NATEONMain.exe파일을 가지고 하겠습니다.

NATEONMain.exe NateOn Messenger SK Communications	크기: 8,53MB (8,945,664 바이트)	Microsoft Visual C++ ver. 8.0 / Visual Studio 2005 - no MSCab
	크기: 2,75MB (2,886,144 바이트)	UPX -> Markus & Laszlo ver. [2.02]

[그림 9]

패킹 했을 때와 안 했을 때 인데 용량차이가 거의 3~4배 가량 차이가 나는 것을 보실 수 있습니다. (실행압축 한 파일도 exe파일이므로 당연히 실행 가능)

두 번째는 여러 크래커들로부터 자신의 파일이 역분석 당하는 것으로부터 보호 받을 수 있다는 것입니다. 일단 패킹이 되게 되면 코드가 압축이 되어 있으므로 정적분석으로는 절대로 main함수를 찾아갈 수 없을 것입니다. 또한 해당 파일을 분석하려면 main함수를 찾아서 분석을 시작해야 하는데 동적 분석으로 해도 main함수를 찾기란 엄청 힘들 것입니다. 사실 위에서 보여준 UPX패킹은 크래커들로부터 파일을 보호하는 목적보다는 방대한 파일의 용량을 줄이는 것이 목적이어서 쉽게 OEP(Original Entry Point)를 찾을 수 있습니다. 다음 장에서 언패킹을 설명할 것인데 이렇게 정상적인 OPE를 찾아서 덤프를 뜨고 해당 파일의 IAT를 복구 시켜주게 되면 언패킹을 성공하게 되는 것입니다. 언패킹을 설명하고 시연하면서 더욱 자세하게 설명을 하도록 하겠습니다.

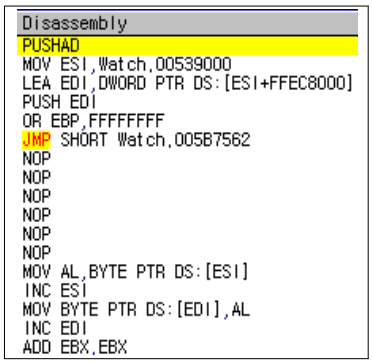
2. 언패킹

2.1 언패킹이란..

언패킹이라는 것은 위에서 설명 했듯이 패킹했던 파일을 푸는 것입니다. 자세히 설명해보면 압축된 코드가 메모리에 올라가서 실행을 하게 되면 압축을 해제하는 루틴을 거친 다음 원본 소스를 실행하게 되는데 그때 원본 소스 시작 점을 잡아서 해당 부분에서 덤프를 뜨는 것입니다. 즉, 압축이 해제된 상태의 코드를 덤프 뜨는 것이기 때문에 언패킹 성공 했다고 볼 수 있는 것입니다. 하지만 이렇게 시작 점(OEP)를 찾는 다는 것은 강력한 패커 일수록 더욱 어렵습니다. 이번 절에서는 몇몇 간단한 패커들을 언패킹 하도록 하겠습니다.

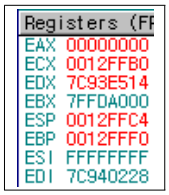
2.2 UPX v2.02w 언패킹

위에서 UPX패커를 가지고 패킹을 설명하였으므로 가장 처음으로 UPX언패킹을 하도록 하겠습니다. 언패킹을 할 때는 2가지가 있습니다. 첫 번째로는 언패킹 툴을 사용해서 언패킹을 하는 것입니다. 두 번째로는 직접 어셈블리 코드와 IAT EAT등등을 보면서 언패킹을 하는 것입니다. 하지만 여기서는 첫 번째 방법은 사용하지 않을 것입니다. 왜냐하면 툴을 사용해서 푸는 것은 원리 같은 것도 아에 모르는 사람들도 풀 수 있기 때문입니다.
UPX언패킹 같은 경우는 패커에서 옵션 주면 바로 풀 수 있습니다. 이제 UPX언패킹을 시작해보도록 하겠습니다. 대상 파일은 아까 UPX패킹했던 파일로 하겠습니다.
먼저 해당 파일을 올리디버거로 열어보겠습니다.



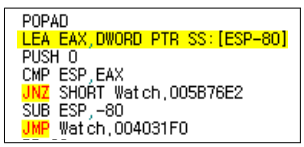
[그림 10]

가장 먼저 보이는 명령어가 PUSHAD 명령어입니다. 해당 명령어는 아래 그림에 나와있는 8개의 레지스터 값을 EAX부터 차례로 스택에 넣으라는 명령어입니다.



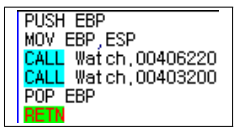
[그림 11]

왜 레지스터의 값을 전부 스택에 넣는 것일까요?? 그것은 현재 상태의 레지스터 값을 저장해 놨다가 압축이 풀린 코드에서 해당 레지스터 값을 불러서 사용하기 때문입니다. 그러면 분명히 해당 레지스터 값들을 스택에서 다시 레지스터로 불러오는 POPAD명령어가 있을 것입니다. 그 부분이 어디인지 찾아보도록 하겠습니다. 처음에 PUSHAD를 실행 시켜 줍니다. 그 다음에 그때 설정 된 ESP에 접근하면 멈추게 하는 “Hardware, on access” 하드웨어 브레이크포인트를 걸어줍니다. 그 다음에 F9를 눌러서 실행을 시켜주면



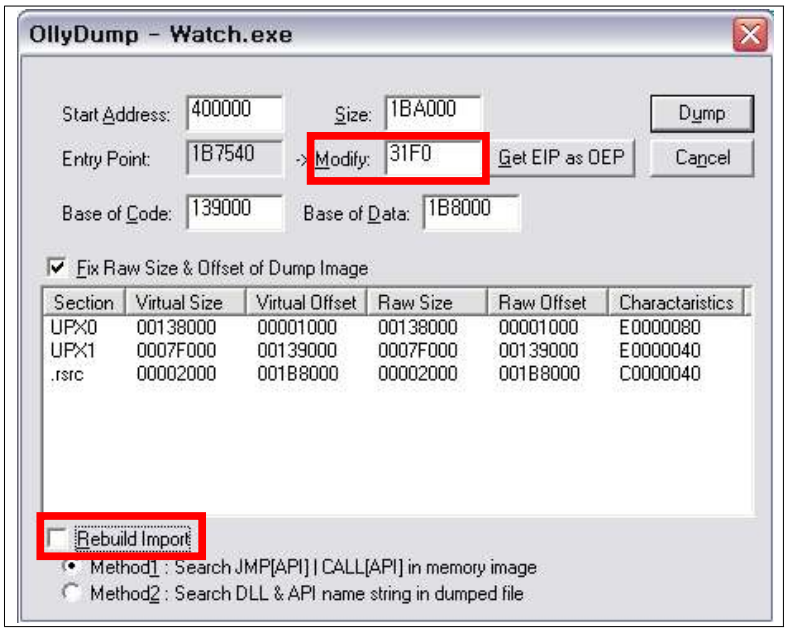
[그림 12]

위와 같은 상태로 멈추게 됩니다. 코드를 보니 POPAD가 존재하는 것을 확인할 수 있습니다. 가장 아래 있는 JMP문까지 실행을 시켜주면 OEP로 갈 수 있습니다. JMP문을 실행 시킨 후의 코드를 봐보겠습니다.



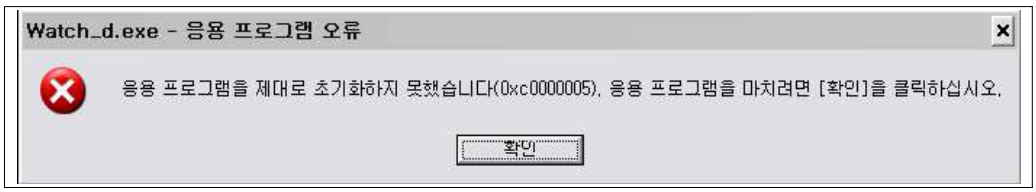
[그림 13]

제대로 된 OEP를 찾은 것 같습니다. 만약 아니라면 언패킹 한 후 실행 시킬 때 오류가 나겠죠?? 이제 저 부분에 덤프를 떠줍니다.



[그림 14]

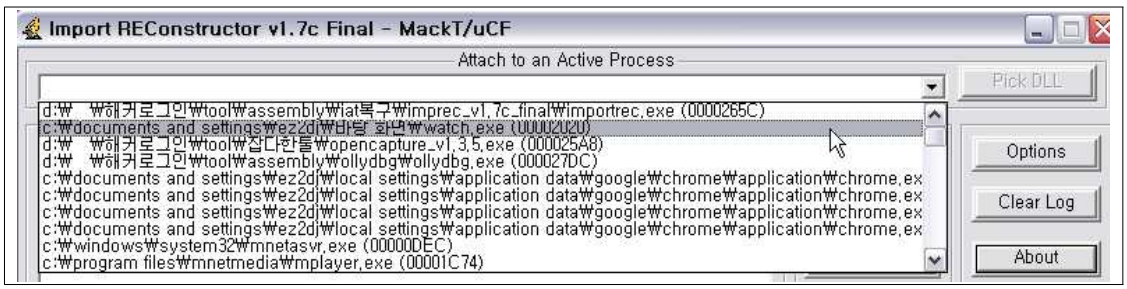
OllyDump라는 플러그인을 사용한 것입니다. Entry point를 31F0으로 수정하겠다는 것을 볼 수 있습니다. 그리고 아래 네모 친 Rebuild Import부분은 IAT를 복구 시켜주겠다는 것인데 IAT를 전문적으로 복구시켜주는 툴을 사용할 것이므로 해당 부분은 체크를 해제해 주겠습니다. 그리고 Dump를 해줍니다. 이제 저장 한 파일을 실행시키도록 하겠습니다.



[그림 15]

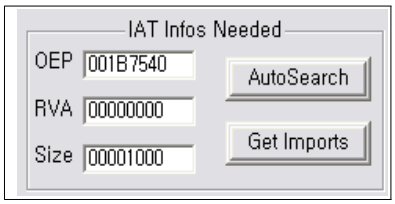
깜짝 놀라셨겠지만 위와 같은 에러창이 뜨는 이유는 IAT가 제대로 복구 되질 않았기 때문입니다. IAT에서는 해당 파일이 사용할 dll파일들의 정보를 가져오는데 dll파일들을 읽지 못하기 때문입니다. IAT복구는 ImportREC라는 툴을 사용해 복구하겠습니다.

ImportREC를 켜서 패킹이 되어 있는 파일로부터 IAT정보를 얻어온 다음 그 정보를 덤프 뜬 파일에 적용시켜 주는 것입니다.



[그림 16]

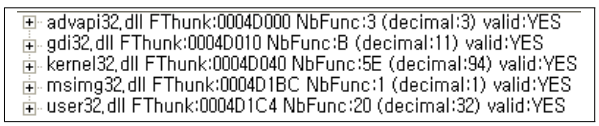
저 watch.exe라는 파일이 패킹 된 파일입니다. 그 다음에 입력을 해줄 부분을 봐보겠습니다.



[그림 17]

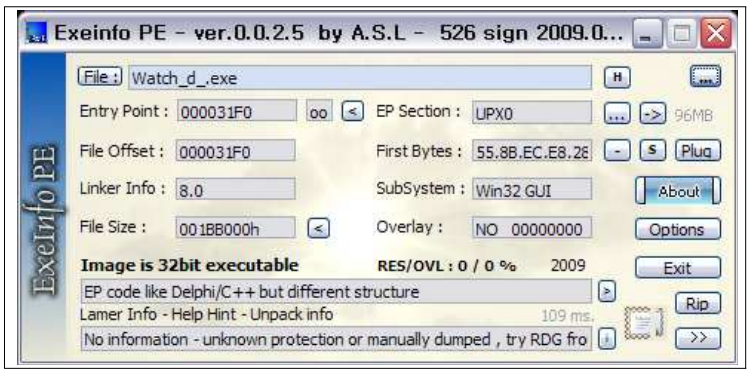
OEP를 적는 부분에는 저희가 제대로 찾은 OEP를 입력해 줍니다. 아까 덤프 뜰 때 수정 될 OEP를 적어주면 됩니다. (0x31F0 이었습니다.)

그 후 AutoSearch를 해주고 제대로 된 OEP를 찾았다면 Get Imports를 해서 IAT정보를 얻고 만약 제대로 된 OEP를 찾지 못했다면 Get Imports를 해도 정상적인 dll정보를 가져오지 못할 것입니다. 우리는 제대로 된 OEP를 찾았으므로 GetImports한 상태를 봐보겠습니다.



[그림 18]

전부 vaild에 YES라고 뜨므로 성공 한 것 같습니다. 만약 NO라고 뜬 부분이 있으면 다시 봐봐야 합니다. 이제 아래 있는 Fix Dump를 클릭해 아까 덤프 뜬 파일에 덮어 씌워줍니다. 그리고 PE확인 툴로 확인을 해보면



[그림 19]

언패킹이 성공했고 실행을 해보면 실행 또한 제대로 되고 있는 것을 확인 할 수 있습니다.

지금까지 UPX언패킹을 해보았는데 자동화 된 언패킹 툴을 사용하지 않고 언패킹을 해보았습니다. 지금부터는 다른 언패킹을 계속 해볼 것인데 방금처럼 자세히 설명은 하지 않고 언패킹을 하도록 하겠습니다.

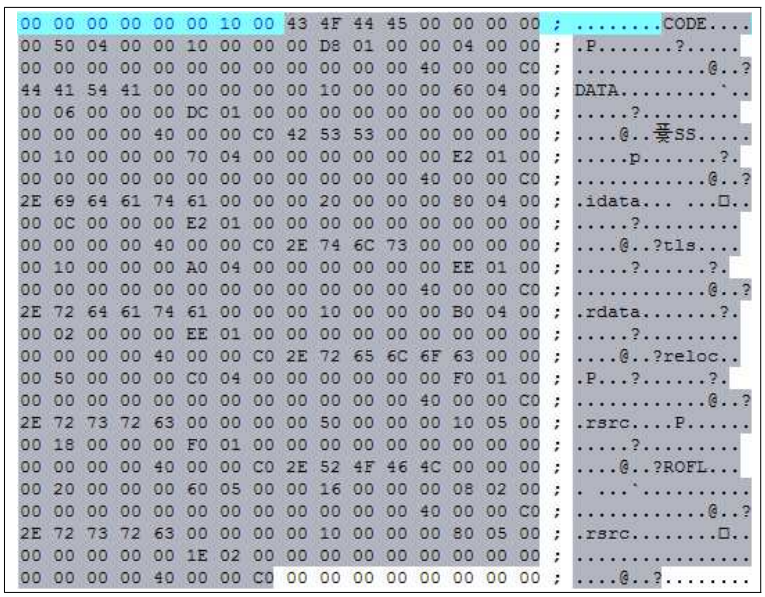
2.3 Aspack v2.0/2.001언패킹

이미 Unpacking문제들을 보면 여러 가지로 패킹이 되어 있으므로 Unpacking문제들을 가지고 해보겠습니다.

무엇으로 패킹이 되어 있는지 먼저 확인한 다음 올리디버거로 열어봅니다.

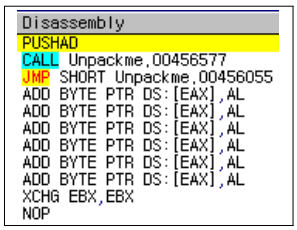
[그림 20]

Aspack으로 패킹이 되었다는 것을 확인 했습니다.



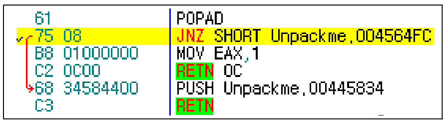
[그림 21]

Section header를 봐도 아까 UPX와는 다르다는 것을 알 수 있습니다.



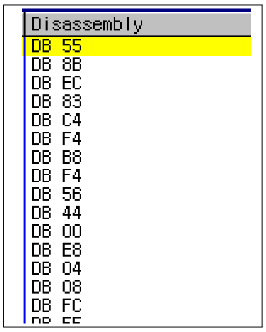
[그림 22]

처음 코드를 보니 PUSHAD로 시작을 하면서 UPX와 비슷하다는 것을 느꼈습니다. 위에서 했던 방법대로 해당 코드를 실행해 주고 ESP에 하드웨어 브포를 걸어 준 다음 F9로 실행을 시켜주겠습니다.



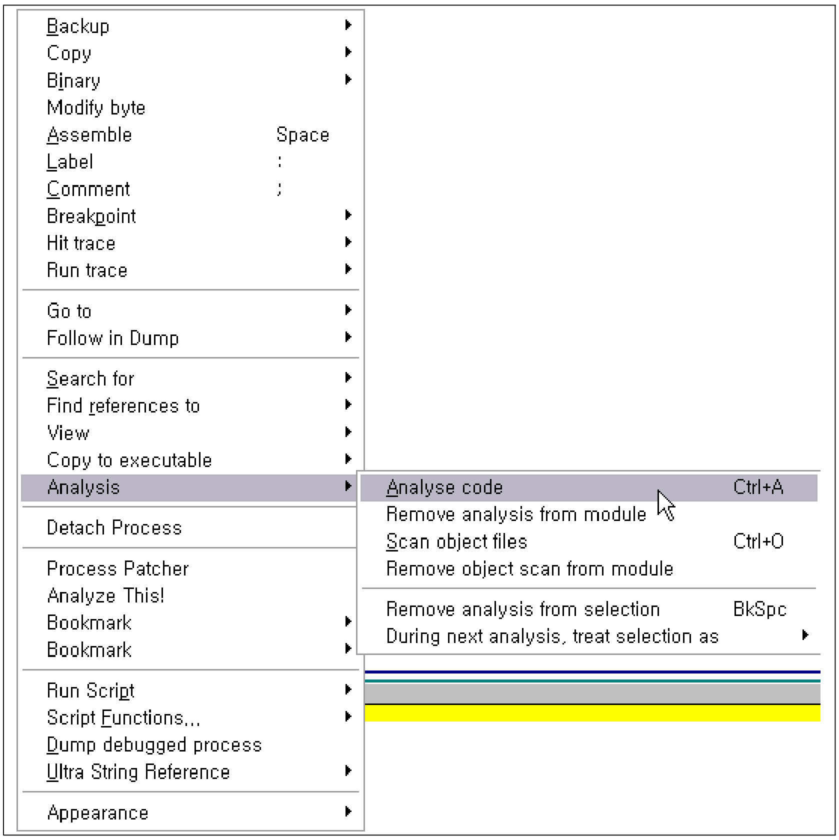
[그림 23]

예상대로 POPAD가 존재했으며 해당 루틴을 계속 따라가보면



[그림 24]

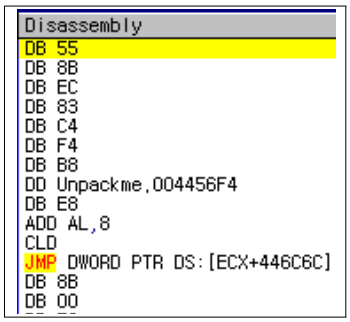
위와 같은 부분으로 갈 것입니다. 여기서 보면 처음 보는 사람들은 OEP가 아니구나 라고 생각할 수도 있는데 이와 같은 경우에 맞닥드리면 당황하지 말고 아래와 같이 Analyse code를 눌러줍니다.



[그림 25]

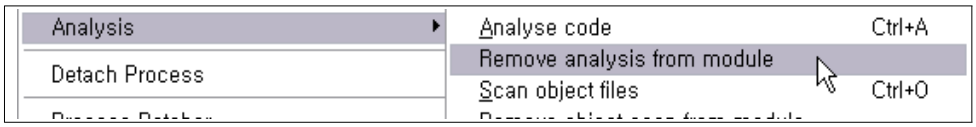
올리디버거에서 코드를 1Byte씩 읽어버려서 이러한 현상이 일어난다고 합니다.

다시 읽혀도 이렇게 나오네요. 그래서 그냥 Disassembly Viewer로 Hex값을 쳐서 봐보도



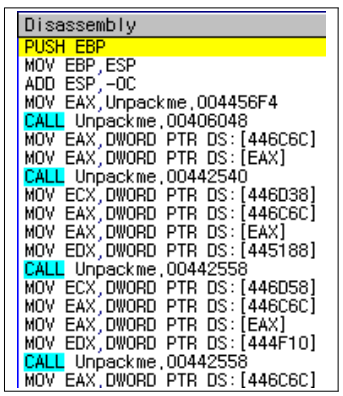
[그림 26]

되지만 올리디버거 기능에서



[그림 27]

이것을 눌러주시면 코드가 제대로 나오는 것을 확인 할 수 있습니다.

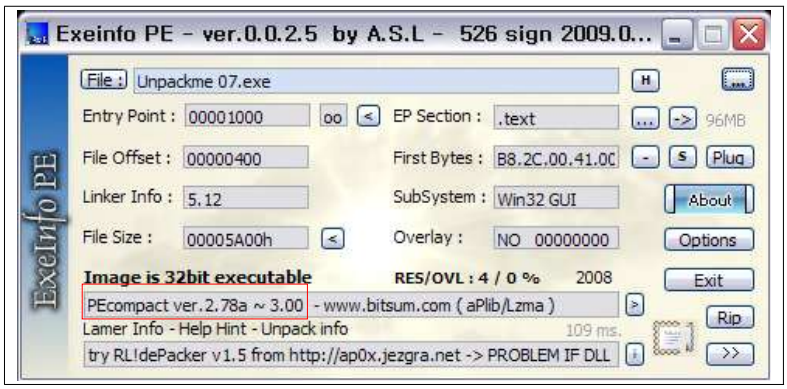


[그림 28]

이제 제대로 된 OEP를 찾은 것 같으니 해당 주소로 EP를 바꿔주고 IAT를 복구 시켜주면 언패킹이 끝나게 됩니다. 제대로 된 OEP를 찾았기 때문에 언패킹이 성공하였습니다.

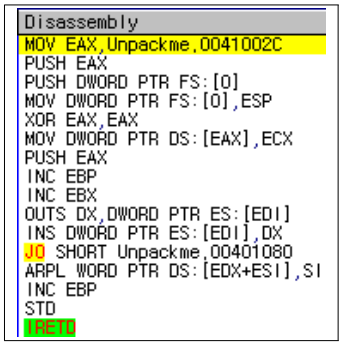
[그림 29]

2.4 PEcompact ver.2.78a ~ 3.00 언패킹



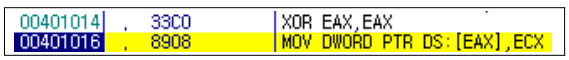
[그림 30]

무엇으로 패킹이 되어있는지 확인했으므로 이제 올리디버거로 열어보겠습니다.



[그림 31]

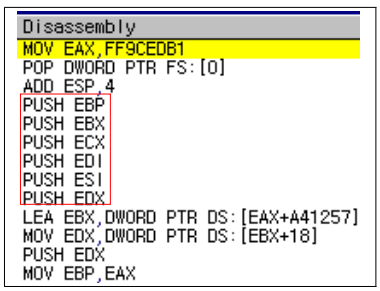
지금까지 봐왔던 PUSHAD라는 명령어가 없습니다. 하지만 어디선가 레지값들을 저장하는 부분이 있을 것이므로 계속 트레이스를 해보겠습니다.



[그림 32]

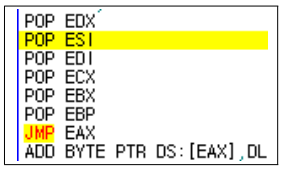
해당 부분에서 예외처리가 발생 했습니다. 당연히 EAX는 0인데 0이 가리키고 있는 곳에 ECX값을 넣으라고 했으니 예외처리가 발생 했습니다.

Shift+ F9를 눌러서 예외처리 루틴으로 들어가겠습니다.



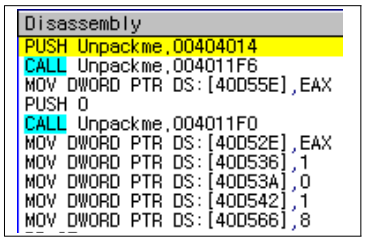
[그림 33]

코드를 보니 PUSHAD가 여전히 없습니다. 하지만 빨간색 네모 친 부분을 보니 레지스터 값들을 PUSH하고 있습니다. 그래서 마지막 PUSH까지 실행 시킨 후 하드웨어브포를 걸고 실행 시켜보았습니다.



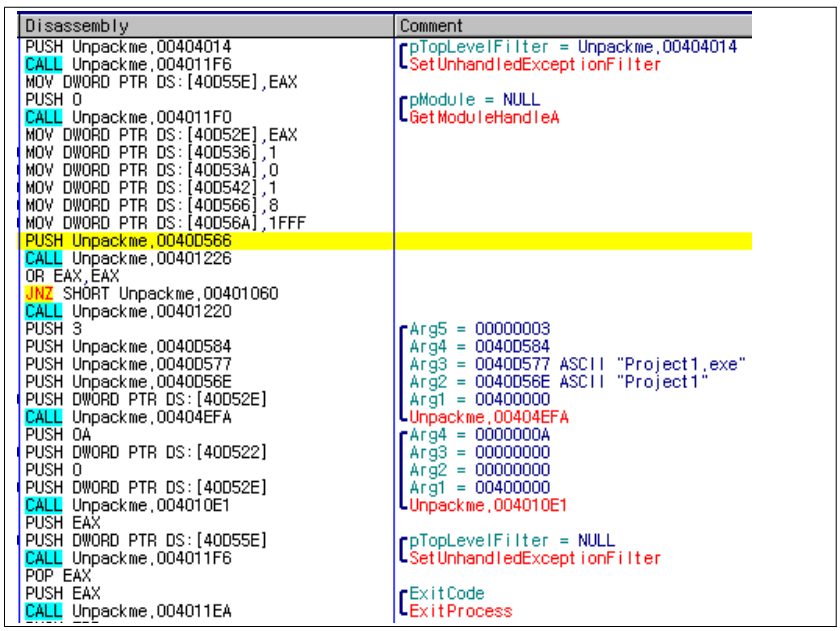
[그림 34]

또 다시 그 값들이 POP되고 있었습니다. 느낌상 거의 OEP를 찾은 것 같군요. JMP문 까지 실행 시키도록 하겠습니다.



[그림 35]

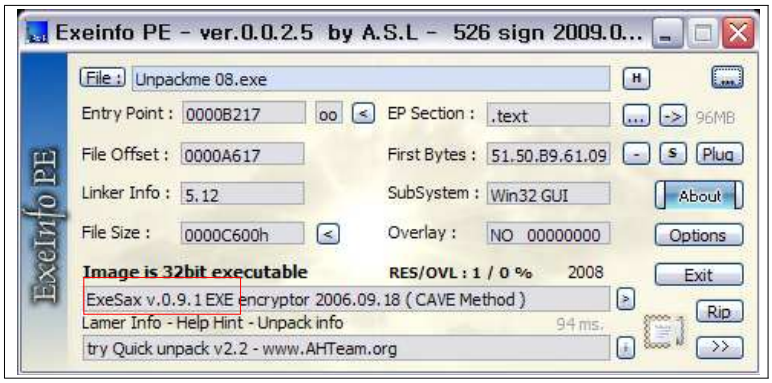
OEP처럼 생긴 부분이네요. 코드가 쫌 이상하다 싶으로 다시 한번 분석하도록 위에서 해줬던 방법을 사용 해보겠습니다.



[그림 36]

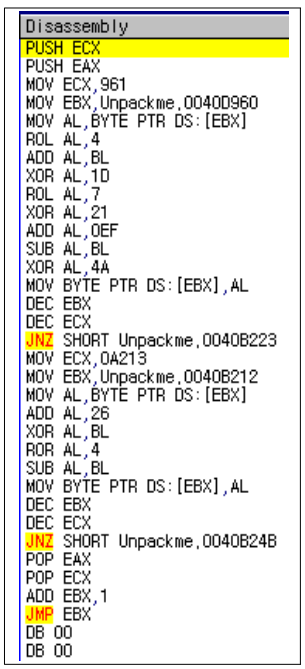
OEP를 제대로 찾은 것 같군요. EP를 바꿔주고 IAT를 복구 시켜주면 끝납니다.

2.5 ExeSax v.0.9.1 언패킹



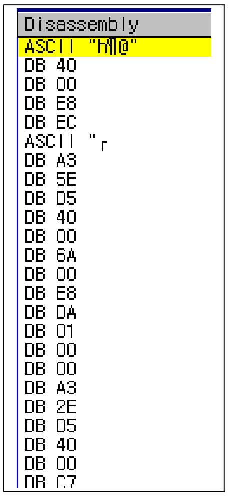
[그림 37]

무엇으로 패킹이 되었는지 확인했으므로 올디로 열어보겠습니다. 대부분 PUSHAD가 처음에 나오는 경우는 거의 없습니다. 언패킹의 목표는 제대로 된 OEP를 찾아서 IAT복구를 시켜주면 되는 것인데 제대로 된 OEP를 찾는 다는 것이 노가다성을 띄기도 하고 어렵기도 합니다. 한번에 OEP를 찾는 경우도 있는데 못 찾는 경우는 여러 번 시도를 해야합니다.



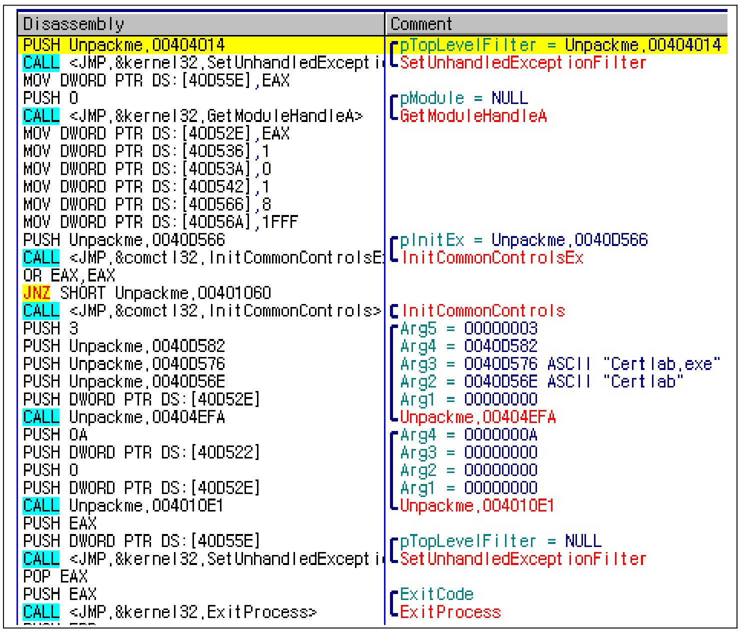
[그림 38]

이 패커도 처음 딱 봐서는 잘 모르겠지만 점프문들이 있으므로 희망을 걸고 트레이스를 해보겠습니다.
트레이스를 해보니 마지막 JMP문에서 어디론가 가는데 해당 부분을 봐보겠습니다.



[그림 39]

이것도 코드가 잘못 읽은 것 같습니다. 이제는 습관적으로 코드를 재분석 시켜줍니다.



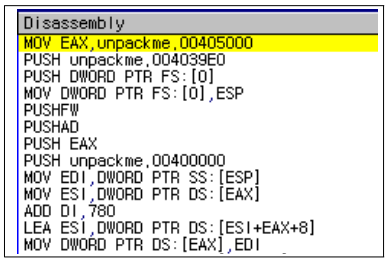
[그림 40]

제대로 된 OEP를 찾은 것 같군요. 해당 부분으로 EP를 바꾸고 IAT복구 시켜주면 끝납니다.

2.6 PEtite 2.x 언패킹

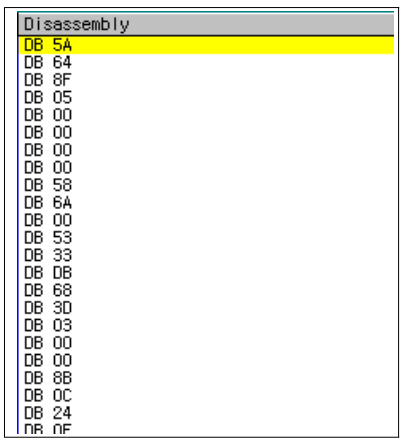
[그림 41]

패킹이 무엇으로 되어 있는지 간단한게 확인 후 올디로 열어보았습니다.



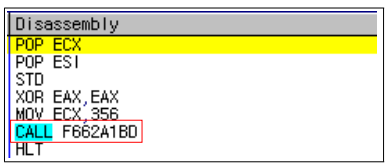
[그림 42]

처음 열어 본 화면인데 PUSHAD가 보이는 군요. 하지만 무작정 위에서 했던 방법대로 했습니다.



[그림 43]

그랬더니 이런 화면이 나와서 당연하다는 듯이 재분석을 시키고 코드를 읽다보면

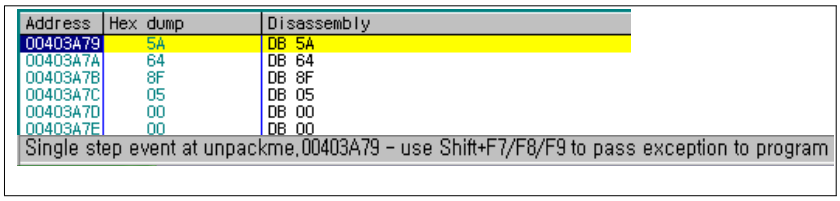


[그림 44]

위와 같은 코드가 나오게 됩니다. 함수를 호출하고 있는데 F662A1BD라는 주소는 커널 영역 주소입니다. (0x80000000부터 커널영역 주소)

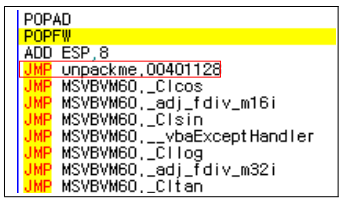
즉, 커널영역 디버깅이 불가능한 OllyDebuger로는 아마 해당 명령어를 실행시키면 오류가 날것이 뻔합니다. 무언가 제가 못 본 것이 있다고 생각하여 다시 한번 실행시켜보았습니다. (똑같이 PUSHAD끝나고 스택에 Hardware BP를 걸어준 상태입니다.)

아까와 똑같은 자리에서 멈추지만 코드 재분석을 하지 않은 상태를 봐보면 예외처리가 발생한 것을 볼 수 있었습니다.



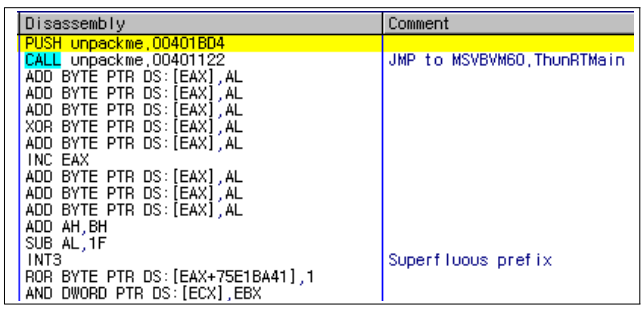
[그림 45]

해당 부분에서 Shift+F9를 눌러서 예외처리 부분으로 가보겠습니다.



[그림 46]

아까 Hardware BP를 걸어둔 상태이기 때문에 계속 진행하지 않고 멈춘 것을 볼 수 있었습니다. 코드를 보니 POPAD가 보이네요. 그 아래로 특정 주소로 점프를 하고 있는데 느낌이 좋습니다. 해당 부분으로 점프를 하게 되면 올디가 잘못 읽어서 1Byte씩 나와 있지만 재분석을 시켜주겠습니다.

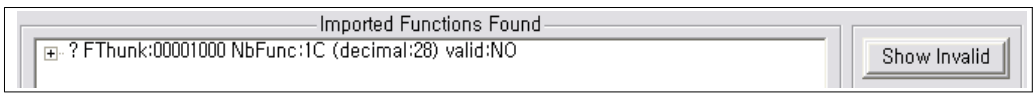


[그림 47]

OEP를 찾은 것 같네요. 해당 부분에 덤프 뜨고 IAT복구 시켜주겠습니다.

IAT복구 시키는 과정을 잠깐 보겠습니다.

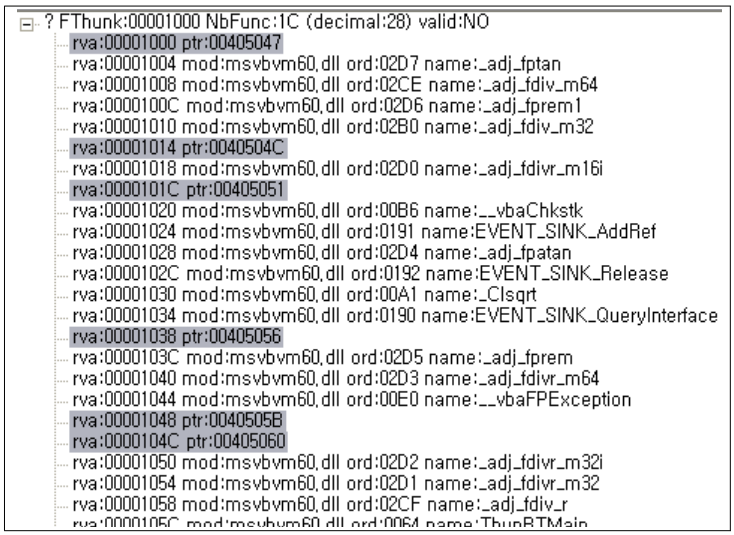
OEP를 수정시켜 주고 Get Imports를 해주었더니



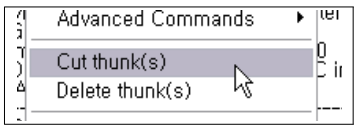
[그림 48]

임포트 함수를 찾을 수 없다고 나옵니다. 이럴 땐 Show Invaild를 눌러줍니다.

회색으로 되어 있는 부분을 전부 마우스 우 클릭을 해서

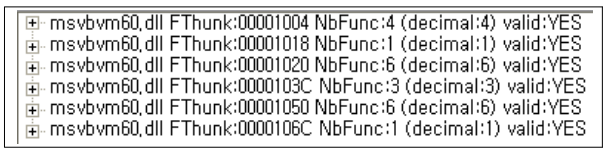


[그림 49]



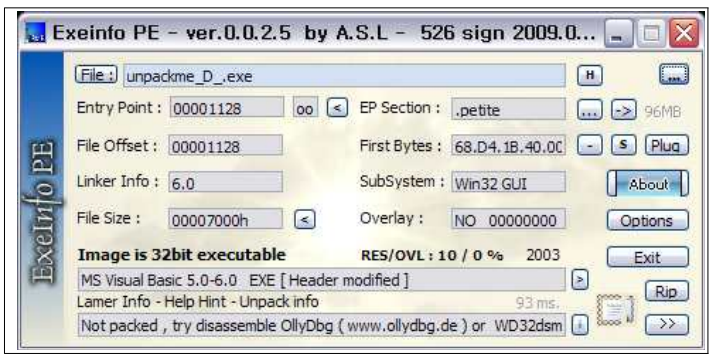
[그림 50]

Cut thunk를 해줍니다.



[그림 51]

위와 같이 제대로 IAT가 잡힙니다. 이제 Fix Dump를 눌러서 적용을 시켜줍니다.



[그림 52]

정상적으로 언패킹에 성공하였습니다.

3. 결론

이번 문서에서는 간단하게 패킹과 언패킹 개념에 대해서 살펴보았고 몇몇 간단한 패커로 패킹되어 있는 프로그램을 언패킹 해보는 과정을 봐보았습니다. 리버싱을 하다보면 피할 수 없는 분야가 이 언패킹 분야입니다. 바이러스를 분석하더라도 게임을 리버싱 하더라도 대부분의 프로그램들이 패킹이 되어있습니다. 위에서 보여준 패커들은 안티디버깅 기술도 크게 적용되지 않았고 OEP를 찾는 것도 그렇게 힘들지 않았습니다.

반면에 언패킹을 하기 힘든 패커들도 있습니다. 예를 들어서 Themida, Winlicense, NoobyProtect, VMProtect 등등 이러한 패커들은 다양한 기능과 엄청난 안티디버깅 기술 코드 난독화 등 많은 기술이 들어가있어서 언패킹을 한다고하면 많은 시간과 노력을 요구로 할 것입니다.

요새는 패킹과 언패킹이라는 분야가 리버싱이라는 분야에서 따로 떨어질 만큼 큰 비중을 차지하고 있다고 첫 번째 결론을 내리고 싶습니다.

두 번째는 개발자가 프로그램을 하나 개발했습니다. 하지만 배포하기 전에 어느정도 보안적 측면에 관심이 있다면 최소한 코드 속에 안티리버싱 기술을 적용시키진 못하더라도 완성된 프로그램에 패킹 정도는 해줘야하지 않을까 라는 저의 생각입니다.