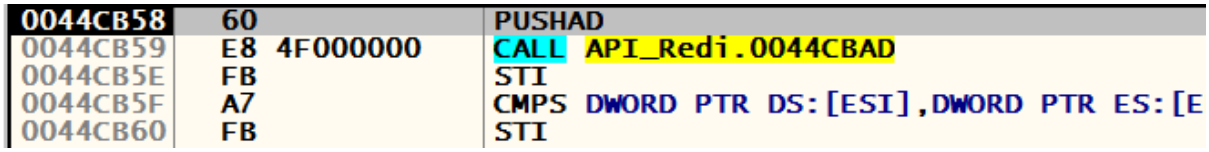
악성코드 분석 보고서

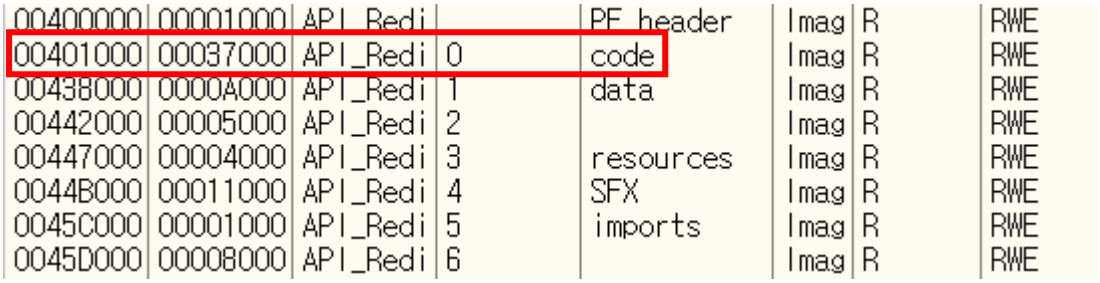
(sand-reversingwithlena-tutorials)

2025.09.19

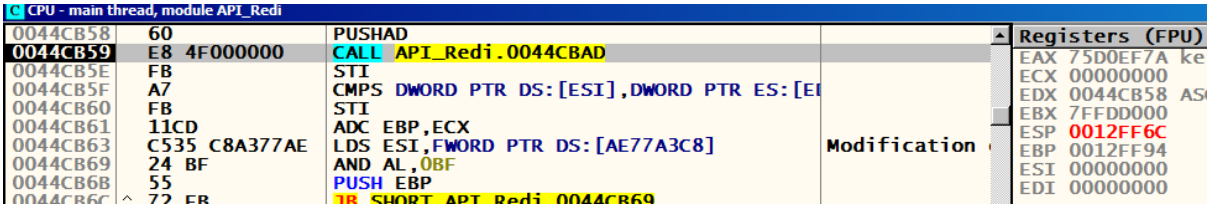
22

1. API Redirection Tutorial.exe

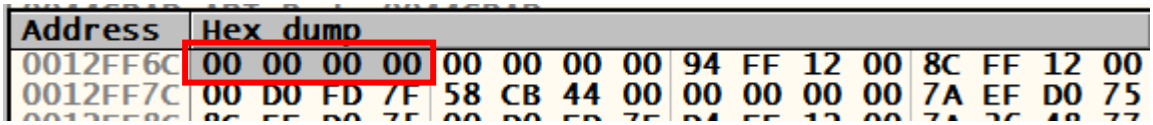




처음 시작하는 부분이 code영역에 속하지 않는 걸 알 수 있다. 그럼 OEP를 찾아야하는데 ESP 변하는 걸 이용해서 찾아본다.



'F8'을 눌러서 하나 내려가 보면 ESP가 바뀌는 걸 볼 수 있는데 dump로 넘어가서 HW BP를 지정해준다.



처음에 여기에 00 00 00 00이 있어서 잘못된 거인 줄 알았지만 HW BP할 때는 상관이 없다. 왜냐하면 저렇게 범위를 지정하는거 자체가 0x0012FF6C에서 4byte의 범위를 지정하는거라서 안에 있는 값은 상관이 없다.

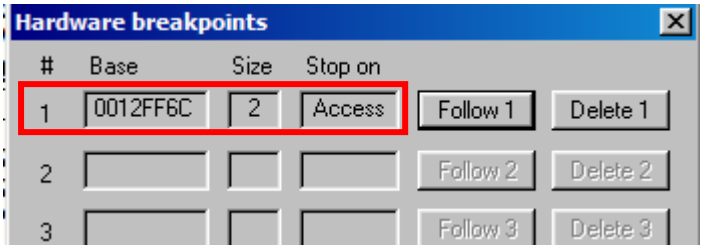
* HW BP

- on access : 주소를 읽거나 쓰기만 하면 중단, Size(1/2/4) 꼭 맞춰줘야함.

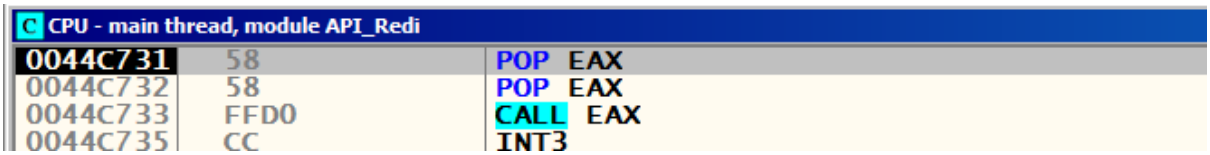
- on write : 주소에 쓰기만 하면 중단, Size(1/2/4) 꼭 맞춰줘야함.

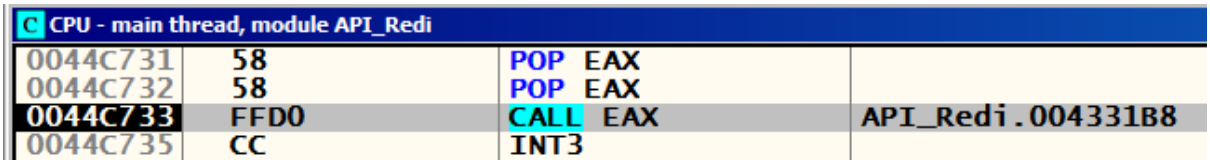
- on execution : EIP가 그 주소의 명령어를 실행하려고 딱 올라올 때 중단, Size 설정 의미 없음.

※ 굳이 범위를 지정하고 할 필요 없이 사이즈를 적어줘도 된다.

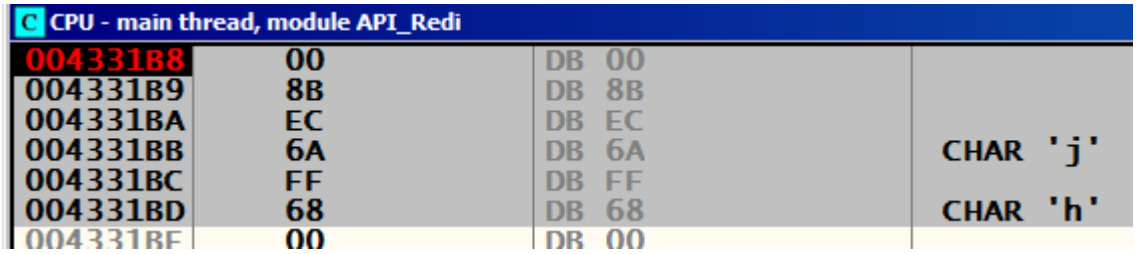


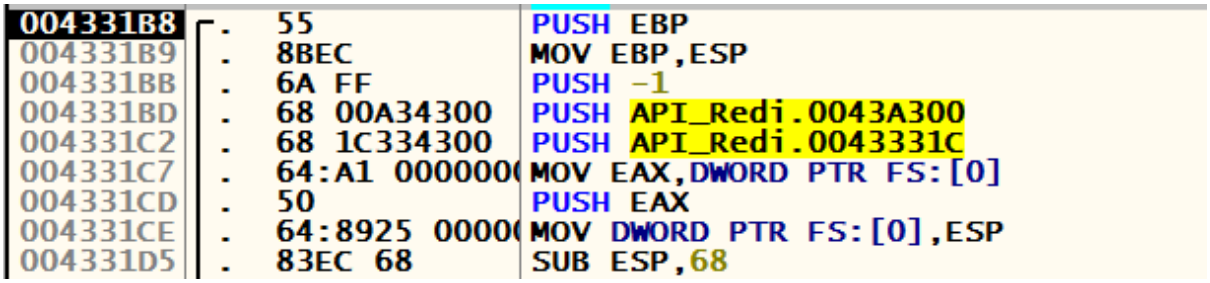
이렇게 BP한 걸 확인할 수 있다. (Word를 선택했기 때문에 2byte)



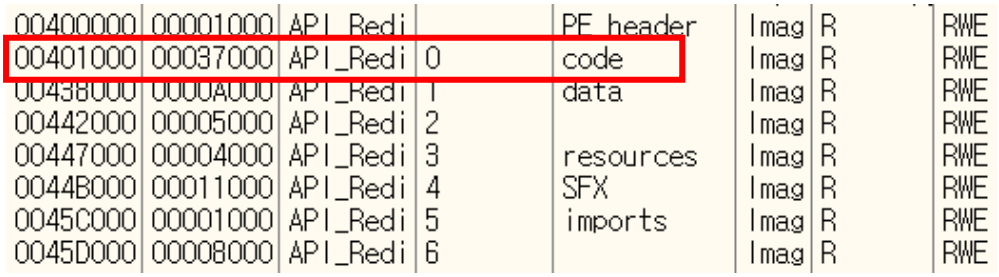


'F9'를 실행하면 0x0044C731에서 멈추는 걸 볼 수 있고 좀 내려오면 CALL EAX에서 0x004331B8에서 점프하는 걸 볼 수 있다.

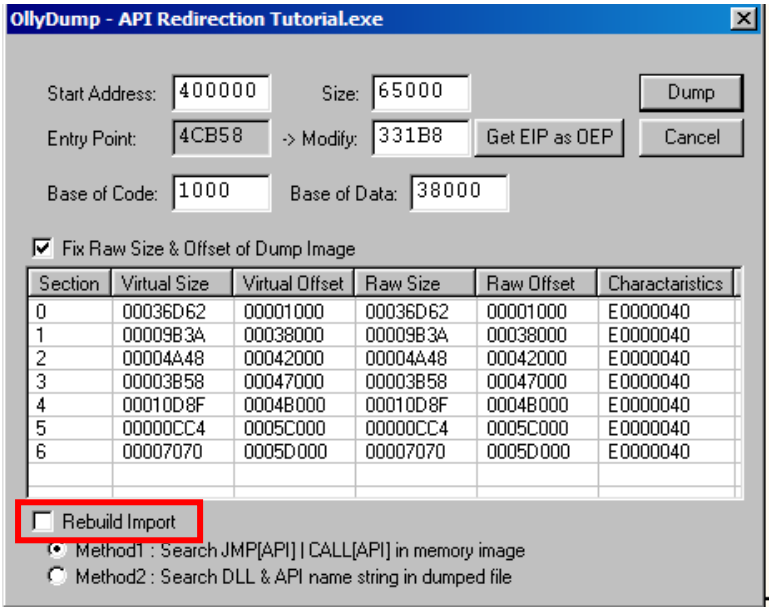




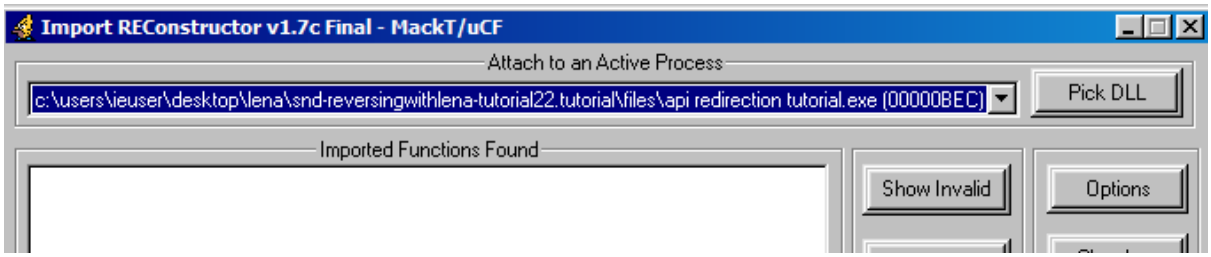
근데 'F7'을 눌러서 0x004331B8로 넘어가주면 처음 사진 처럼 나오는데 'Ctrl + a'을 눌러서 코드를 분석해주면 아래 사진처럼 나오는 걸 볼 수 있다.



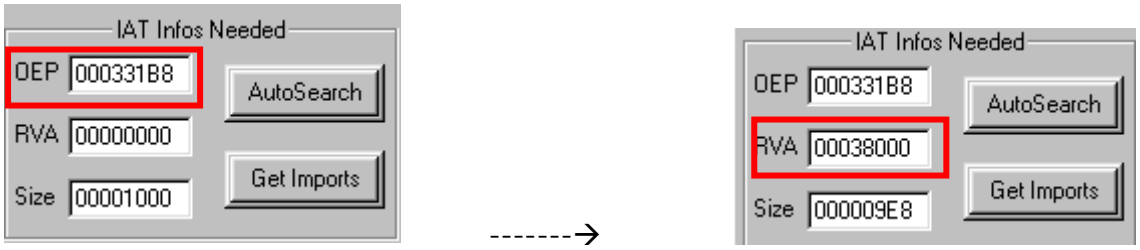
해당 부분이 code에 포함되는 걸로 봐 OEP인 걸 알 수 있다.



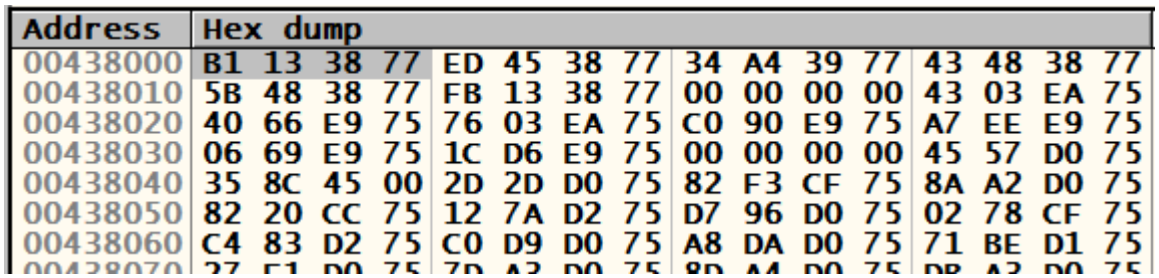
해당 부분에서 덤프를 해주면 된다.



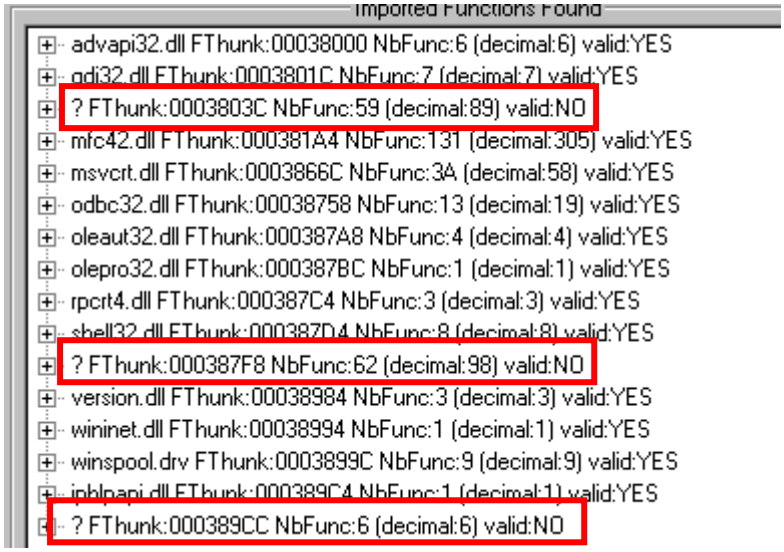
ImpRec를 이용하여 실행 중인 파일을 가져온다.



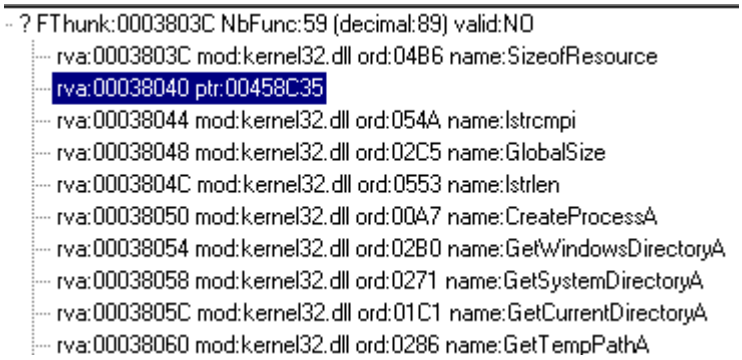
OEP를 바꿔주고 'AutoSearch'를 넣어주면 IAT를 찾아주는 걸 알 수 있다.



올리디버그로 들어가서 확인해보면 0x00438000이 IAT의 시작인 걸 알 수 있다. 'Get Imports'를 눌러서 IAT의 값들이 제대로 복구되었는지 확인한다.



몇 개의 IAT가 Valid:NO로 되어 있는 걸 볼 수 있다.



- 예: kernel32.dll FTThunk:0003803C NbFunc:59 valid:YES
  - FTThunk: 그 DLL 블록의 FirstThunk RVA (IAT 블록 시작 위치)
  - NbFunc: 그 블록 안에 함수 개수
- 예: rva:00038044 mod:kernel32.dll ord:054A name:lstrcmpi
  - rva: 이 IAT 슬롯의 RVA
  - mod: 이 슬롯이 가리키는 DLL.
  - ord: 그 DLL에서의 ordinal 번호
  - name: API 이름
- 예: rva:00038040 ptr:00458C35
  - rva: IAT 슬롯의 위치
  - ptr: 슬롯 안에 현재 들어있는 포인터 값

-> 원래는 mod:kernel32.dll name:... 가 붙어야 정상인데, 지금은 내부 주소라서 매칭 실패 → ptr 표시.

00400000	00001000	API_Redi		PE header	Imag	R	RWE
00401000	00037000	API_Redi	0	code	Imag	R	RWE
00438000	0000A000	API_Redi	1	data	Imag	R	RWE
00442000	00005000	API_Redi	2		Imag	R	RWE
00447000	00004000	API_Redi	3	resources	Imag	R	RWE
0044B000	00011000	API_Redi	4	SFX	Imag	R	RWE
0045C000	00001000	API_Redi	5	imports	Imag	R	RWE
0045D000	00008000	API_Redi	6		Imag	R	RWE

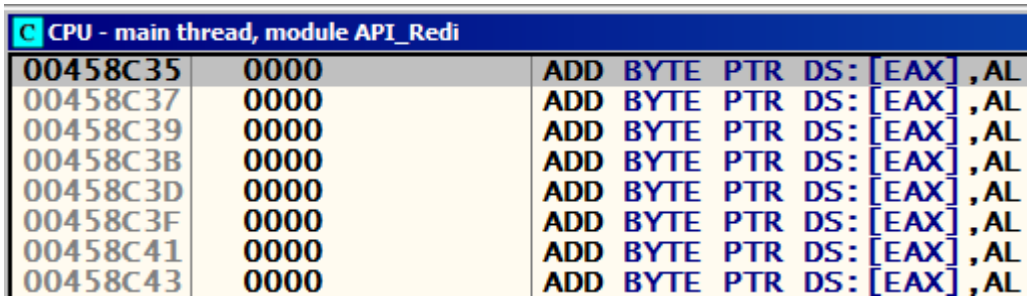
rva:00038040 ptr:00458C35가 있으면 ptr을 Memory Map에서 봐야한다. 왜냐하면 rva는 IAT의 위치이고 ptr은 그 함수가 어디있는지의 위치이기 때문이다.

그래서 ptr:00458C35를 보면 SFX에 위치하는 걸 볼 수 있다, 이런 경우에는 자기코드인 걸 알 수 있다.

CPU - main thread, module API_Redi

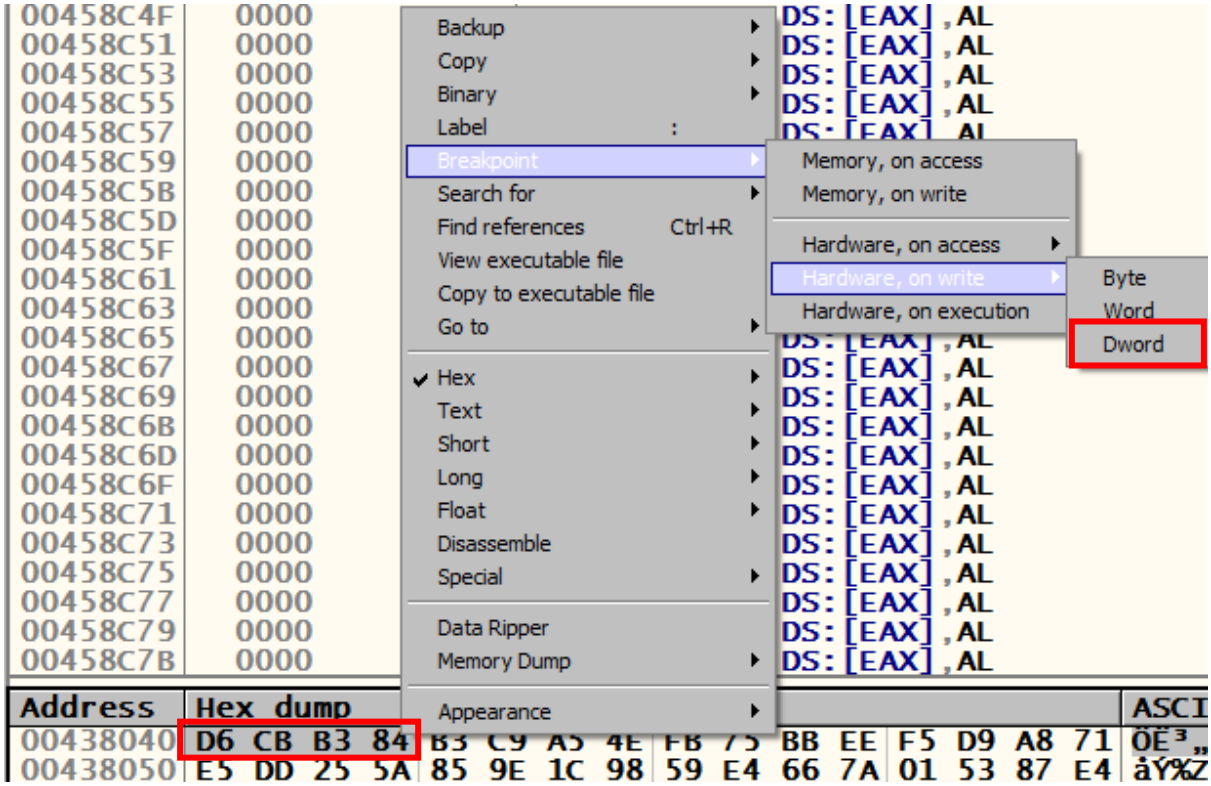
```
00458C35   55              PUSH EBP
00458C36   8BEC            MOV EBP,ESP
00458C38   51              PUSH ECX
00458C39   8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
00458C3C   50              PUSH EAX
00458C3D   FF75 08         PUSH DWORD PTR SS:[EBP+8]
00458C40   E8 F35BFFFF     CALL API_Redi.0044E838
00458C45   85C0            TEST EAX,EAX
00458C47   75 0C           JNZ SHORT API_Redi.00458C55
00458C49   FF75 08         PUSH DWORD PTR SS:[EBP+8]
00458C4C   FF15 4CF64500   CALL DWORD PTR DS:[45F64C]      kernel32.FindClose
00458C52   8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00458C55   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00458C58   C9              LEAVE
00458C59   C2 0400         RETN 4
00458C5C   55              PUSH EBP
00458C5D   8BEC            MOV EBP,ESP
00458C5F   51              PUSH ECX
00458C60   8365 FC 00      AND DWORD PTR SS:[EBP-4],0
00458C64   8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
00458C67   50              PUSH EAX
00458C68   FF75 0C         PUSH DWORD PTR SS:[EBP+C]
00458C6B   FF75 08         PUSH DWORD PTR SS:[EBP+8]
00458C6E   E8 8B5AFFFF     CALL API_Redi.0044E6FE
00458C73   85C0            TEST EAX,EAX
00458C75   75 0F           JNZ SHORT API_Redi.00458C86
00458C77   FF75 0C         PUSH DWORD PTR SS:[EBP+C]
00458C7A   FF75 08         PUSH DWORD PTR SS:[EBP+8]
00458C7D   FF15 54F64500   CALL DWORD PTR DS:[45F654]      kernel32.FindNextFileA
00458C83   8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00458C86   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00458C89   C9              LEAVE
00458C8A   C2 0800         RETN 8
00458C8D   55              PUSH EBP
00458C8E   8BEC            MOV EBP,ESP
00458C90   81EC 48010000   SUB ESP,148
```

이렇게 코드에서 보면 kernel32의 FindClose 함수를 나타내는 걸 알 수 있고 잘 가르키고 있는 걸 볼 수 있다. 하지만 코드를 재시작했을 때 왜 이걸 잘 못 가져오는지 알 수 있다.



코드를 재시작하고 0x00458C35를 보면 아무런 값이 존재하지 않는 걸 알 수 있다. 즉, 리다이렉트 코드는 바이너리가 실행될 때 언패킹 루틴에서 작성되기 때문에 OEP를 찾아서 덤프해버리면 위 사진과 같이 특정 API들의 주소가 찾아지지 않아 크래시가 발생하는 것이다.

rva:00038040를 봐야한다. rva가 IAT의 위치이므로 0x00438040의 값이 변할 때를 봐야한다.

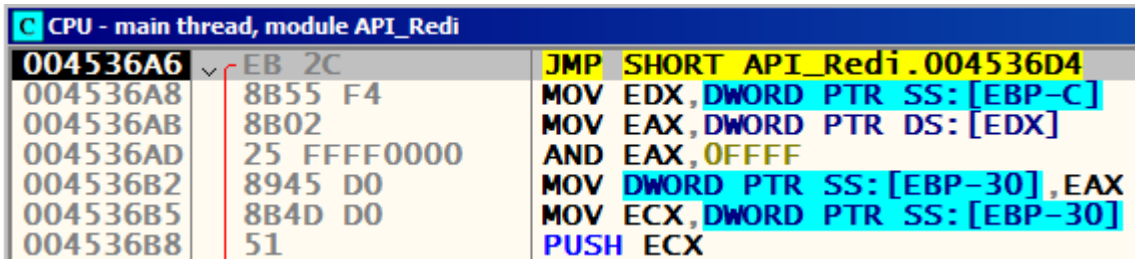


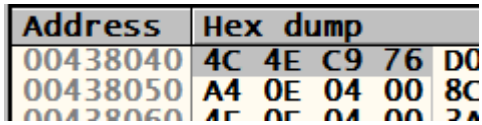
0x00438040의 값이 변할 때를 봐야하니까 on write로 해서 봐야한다.

* 프로그램은 바이너리에 존재하는 모든 임포트들을 찾아내기 위해서 딱 2개의 API 함수가 필요하다.

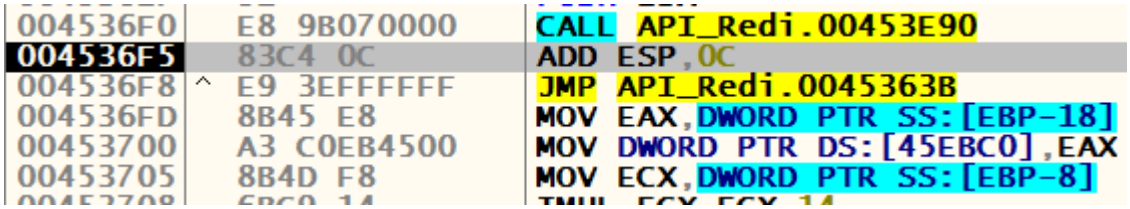
- LoadLibraryA : 특정 dll을 호출

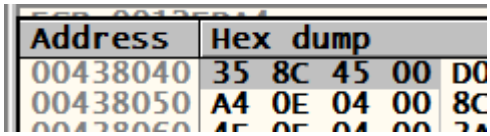
- GetProcAddress : 호출하고자 하는 임포트된 각각의 API 함수 주소를 알아낸다.





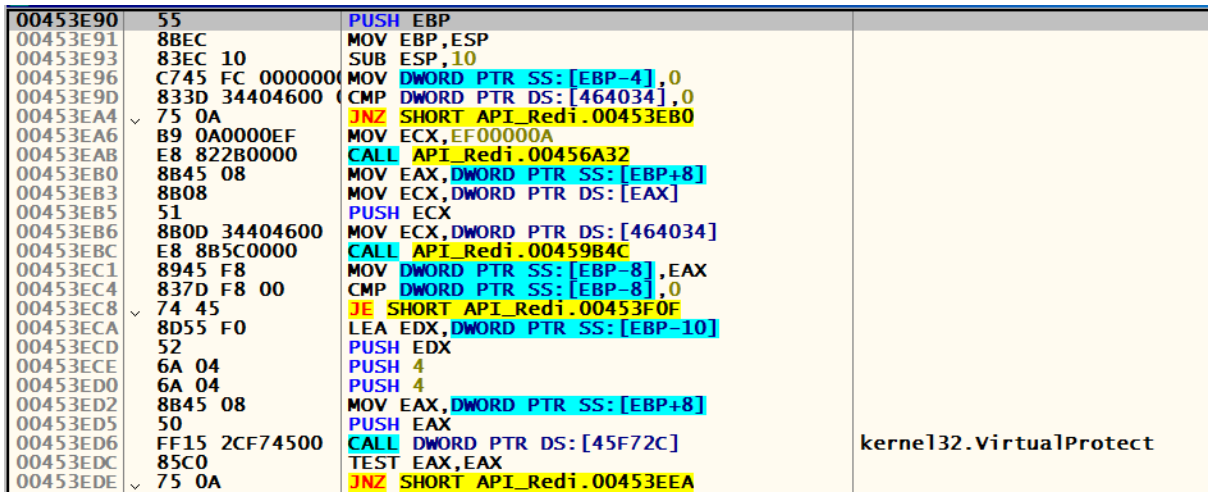
'F9'를 실행하다 보면 해당 부분에서 값이(76xxx)로 변하는 걸 볼 수 있다. 즉 해당 부분이 원래 함수의 원본 주소인 걸 알 수 있다.



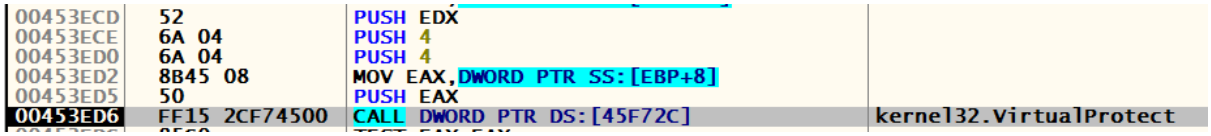


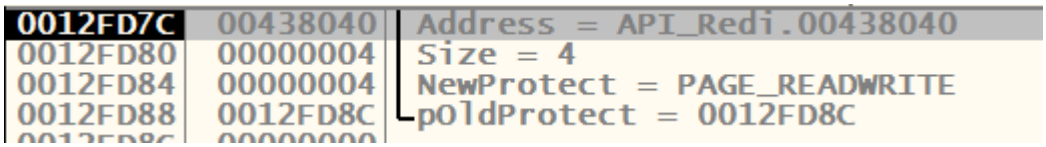
'F8'을 이용해서 좀 더 실행해보면 CALL이 실행되면서 0x00438040 값이 원래의 ptr:00458C35로 바뀌는 걸 볼 수 있다. 즉, 이 부분이 원본 주소 대신에 리다이렉트 된 주소가 씌여진 곳이고 "453E90" 함수가 리다이렉트와 관련된 함수 인 것을 알 수 있다.

==========



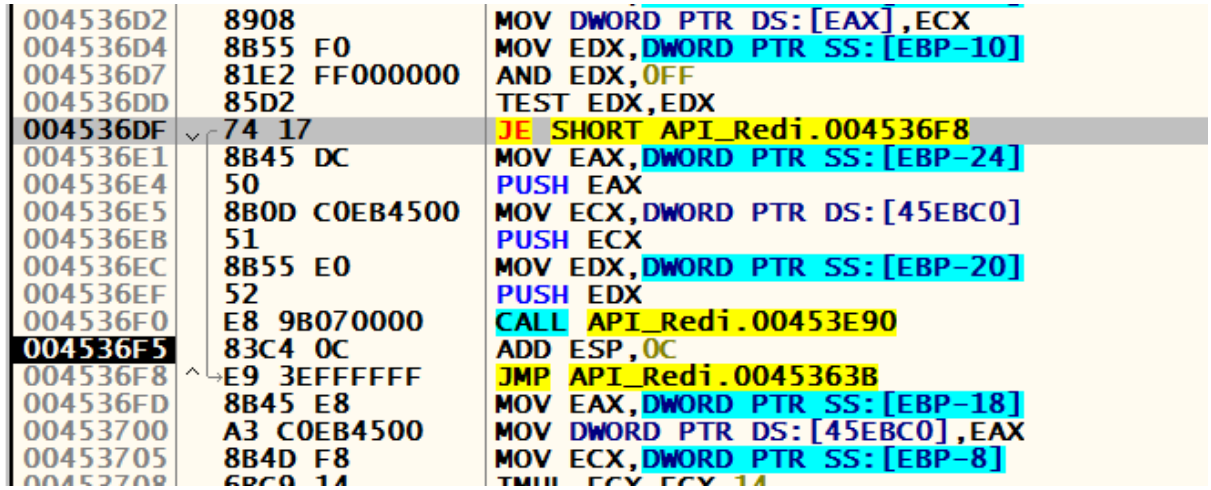
여기서 CALL함수를 들어가보면 VirtualProtect함수가 실행되는 걸 볼 수 있다.



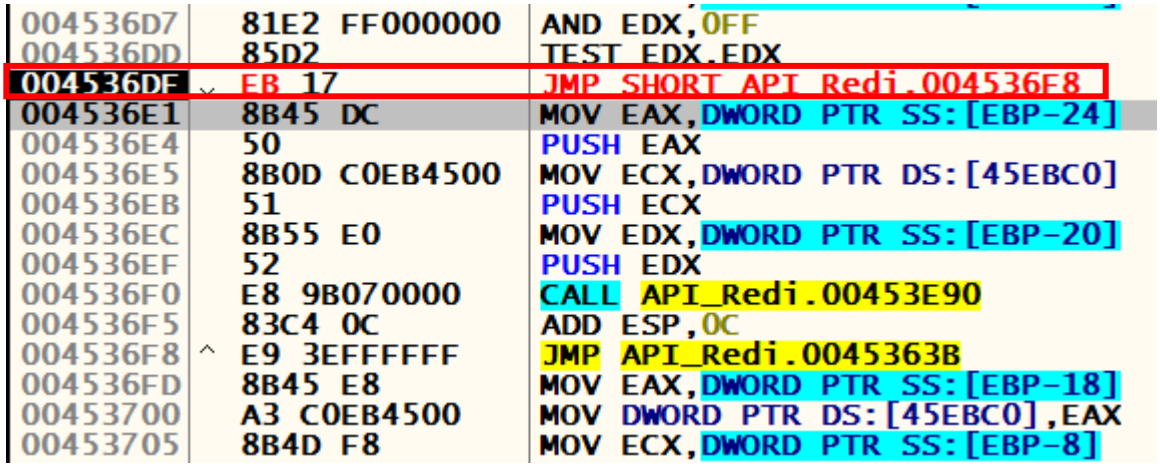


스택을 확인하면 004388C0 의 주소가 인자로 들어감을 확인할 수 있다. 즉, 함수가 리다이렉트 될 때마다 VirtualProtect 가 호출된다. VirtualProtect 함수는 프로세스의 가상 주소 공간안의 commit 된 페이지의 액세스 보호 설정을 변경한다.

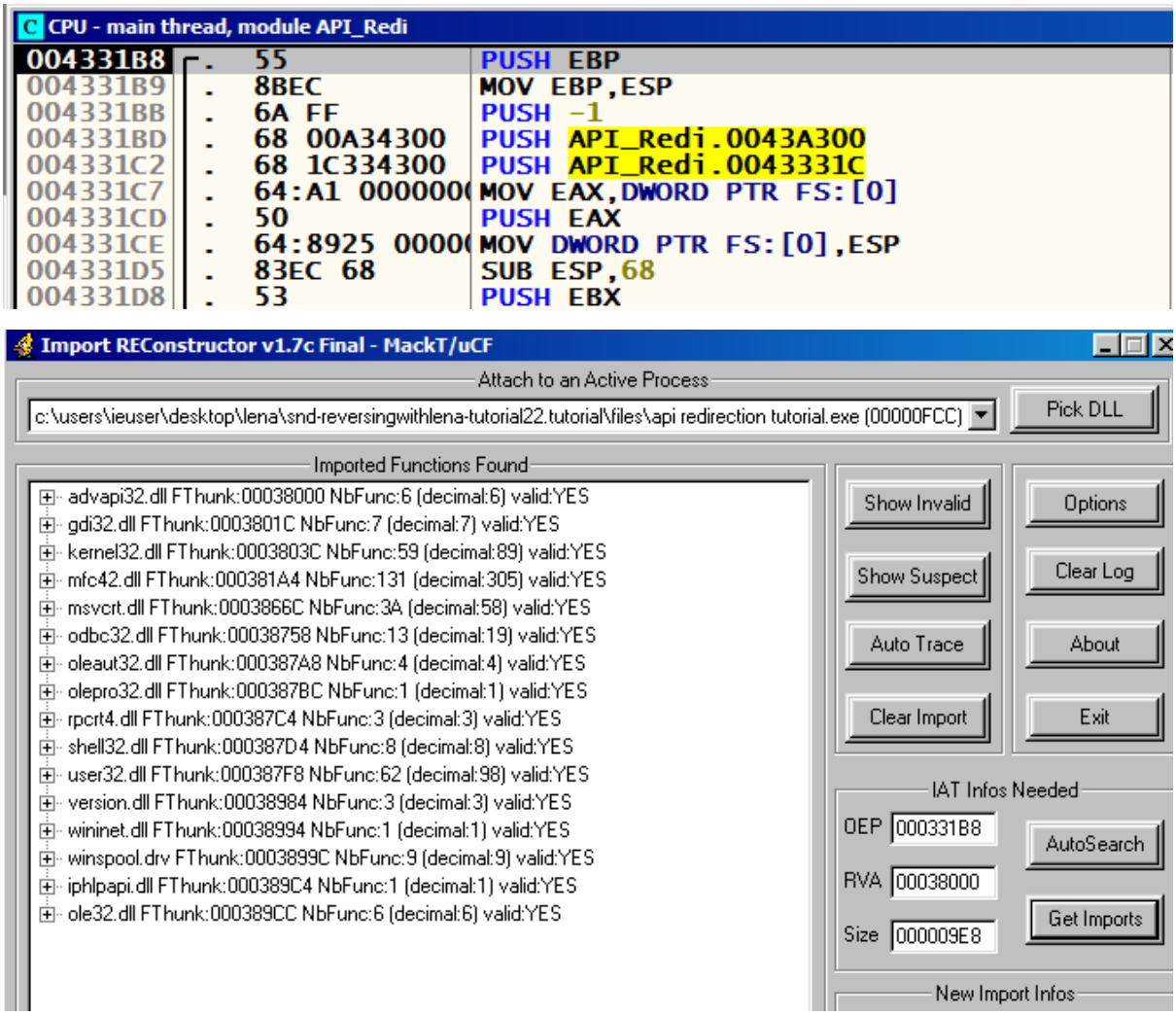
==========



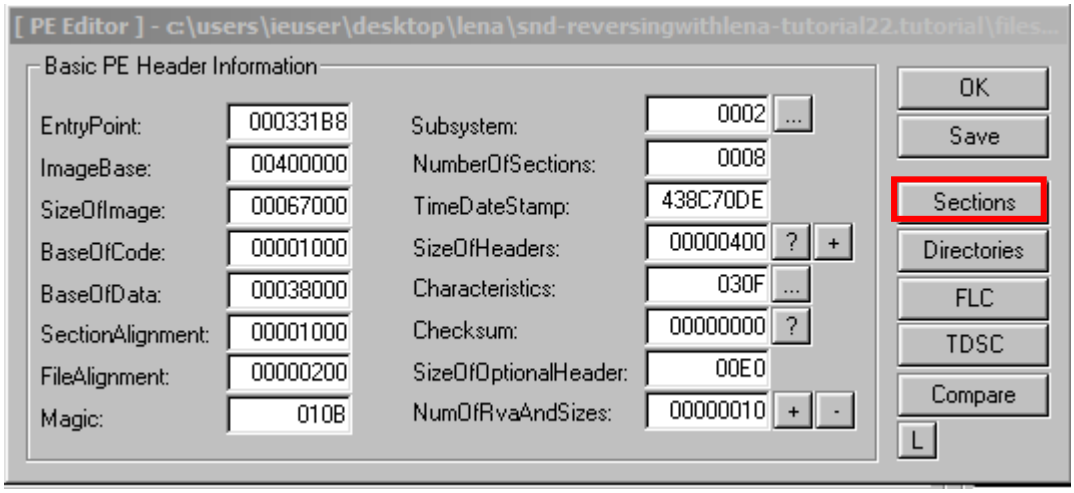
좀만 위에 올려보면 JE함수가 있는 걸 볼 수 있다. 이 함수가 실행되면 CALL함수를 뛰어넘는 걸 알 수 있다.



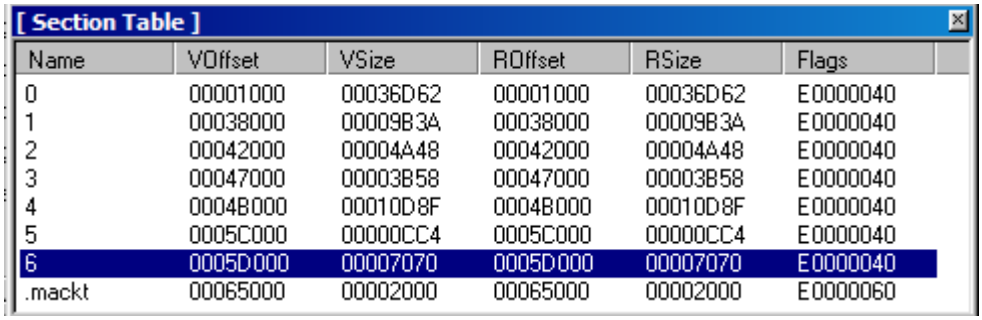
JE -> JMP로 바꾸고 OEP로 찾아가준다.



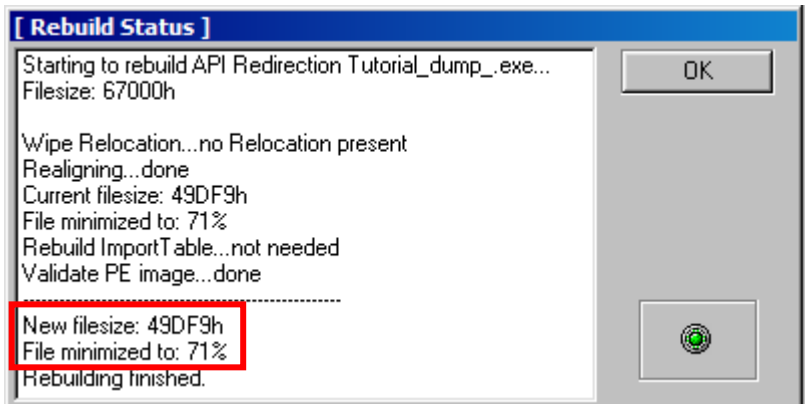
ImpRec를 실행해주면 전 부 YES로 바뀌는 걸 볼 수 있다. 해당 부분에서 Fix dump를 실행해준다.



LordPE를 이용하여 필요 없는 섹션을 정리해준다.



".mackt" 섹션이 새로운 IAT를 포함하기 때문에 기존의 "4, 5, 6" 섹션은 삭제해도 무방하다.



삭제 후 리빌드 해주면 71%까지 줄어든 걸 볼 수 있다.