

악성코드 분석 보고서

(sand-reversingwithlana-tutorials)

2025.09.19

1. API Redirection Tutorial.exe

0044CB58	60	PUSHAD
0044CB59	E8 4F000000	CALL API_Redi.0044CBAD
0044CB5E	FB	STI
0044CB5F	A7	CMPS DWORD PTR DS:[ESI],DWORD PTR ES:[E
0044CB60	FB	STI

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

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

CPU - main thread, module API_Redi				Registers (FPU)			
0044CB58	60	PUSHAD		EAX	75D0EF7A	ke	
0044CB59	E8 4F000000	CALL API_Redi.0044CBAD		ECX	00000000		
0044CB5E	FB	STI		EDX	0044CB58	AS	
0044CB5F	A7	CMPS DWORD PTR DS:[ESI],DWORD PTR ES:[E		EBX	7FFDD000		
0044CB60	FB	STI		ESP	0012FF6C		
0044CB61	11CD	ADC EBP,ECX		EBP	0012FF94		
0044CB63	C535 C8A377AE	LDS ESI,FWORD PTR DS:[AE77A3C8]	Modification	ESI	00000000		
0044CB69	24 BF	AND AL,0BF		EDI	00000000		
0044CB6B	55	PUSH EBP					
0044CB6C	72 FB	JR SHORT API_Redi.0044CB69					

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

Address	Hex dump
0012FF6C	00 00 00 00 00 00 00 00 94 FF 12 00 8C FF 12 00
0012FF7C	00 D0 FD 7F 58 CB 44 00 00 00 00 00 7A EF D0 75

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

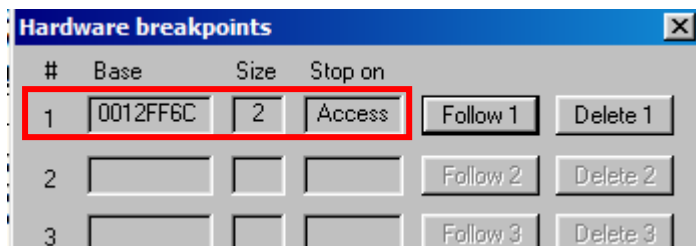
* HW BP

- 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)

CPU - main thread, module API_Redi			
0044C731	58	POP EAX	
0044C732	58	POP EAX	
0044C733	FFD0	CALL EAX	
0044C735	CC	INT3	

CPU - main thread, module API_Redi			
0044C731	58	POP EAX	
0044C732	58	POP EAX	
0044C733	FFD0	CALL EAX	API_Redi.004331B8
0044C735	CC	INT3	

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

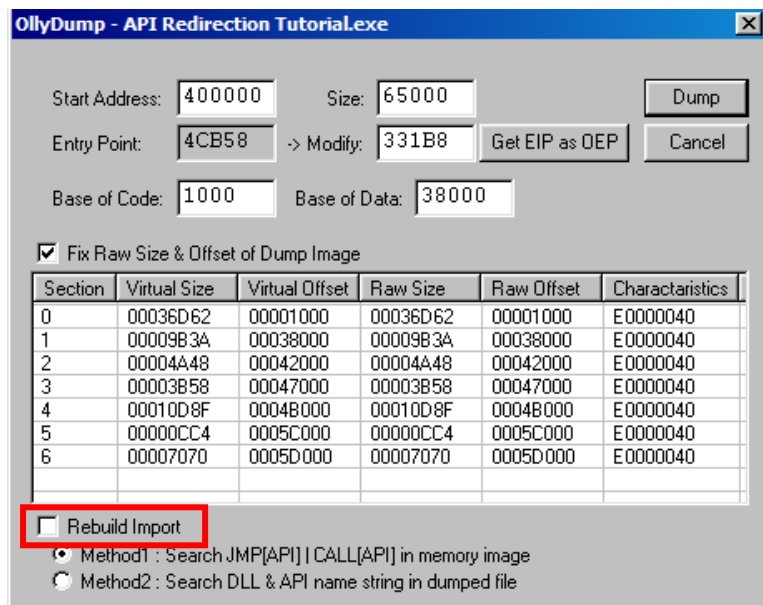
CPU - main thread, module API_Redi			
004331B8	00	DB 00	
004331B9	8B	DB 8B	
004331BA	EC	DB EC	
004331BB	6A	DB 6A	CHAR 'j'
004331BC	FF	DB FF	
004331BD	68	DB 68	CHAR 'h'
004331BE	00	DB 00	

004331B8	. 55	PUSH EBP	
004331B9	. 8BEC	MOV EBP,ESP	
004331BB	. 6A FF	PUSH -1	
004331BD	. 68 00A34300	PUSH API_Redi.0043A300	
004331C2	. 68 1C334300	PUSH API_Redi.0043331C	
004331C7	. 64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
004331CD	. 50	PUSH EAX	
004331CE	. 64:8925 00000000	MOV DWORD PTR FS:[0],ESP	
004331D5	. 83EC 68	SUB ESP,68	

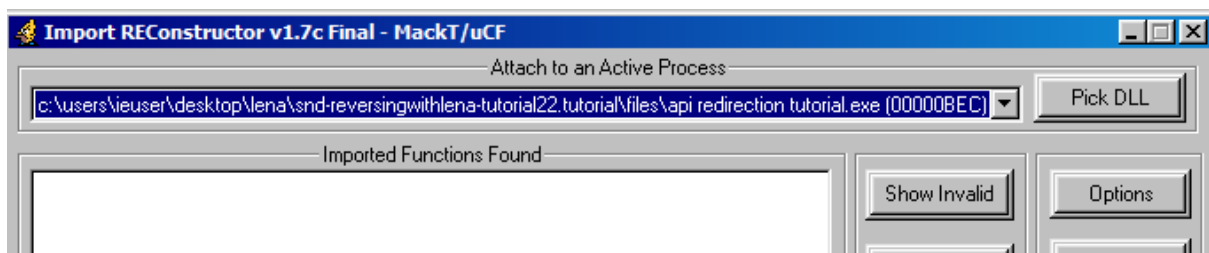
근데 'F7'을 눌러서 0x004331B8로 넘어가주면 처음 사진 처럼 나오는데 'Ctrl + a'을 눌러서 코드를 분석해주면 아래 사진처럼 나오는 걸 볼 수 있다.

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

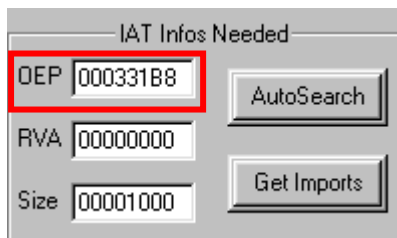
해당 부분이 code에 포함되는 걸로 봐 OEP인 걸 알 수 있다.



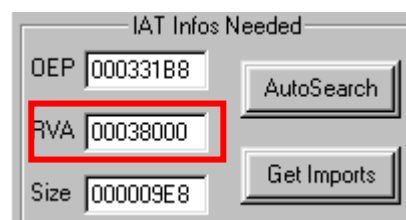
해당 부분에서 덤프를 해주면 된다.



ImpRec를 이용하여 실행 중인 파일을 가져온다.



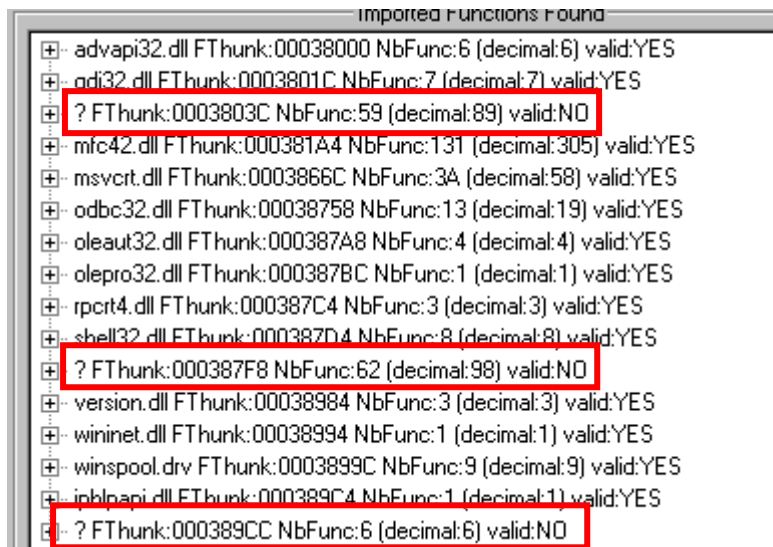
----->



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

Address	Hex dump															
00438000	B1	13	38	77	ED	45	38	77	34	A4	39	77	43	48	38	77
00438010	5B	48	38	77	FB	13	38	77	00	00	00	00	43	03	EA	75
00438020	40	66	E9	75	76	03	EA	75	C0	90	E9	75	A7	EE	E9	75
00438030	06	69	E9	75	1C	D6	E9	75	00	00	00	00	45	57	D0	75
00438040	35	8C	45	00	2D	2D	D0	75	82	F3	CF	75	8A	A2	D0	75
00438050	82	20	CC	75	12	7A	D2	75	D7	96	D0	75	02	78	CF	75
00438060	C4	83	D2	75	C0	D9	D0	75	A8	DA	D0	75	71	BE	D1	75
00438070	27	F1	D0	75	7D	A2	D0	75	8D	A4	D0	75	DB	A2	D0	75

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.

```

- ? FThunk:0003803C NbFunc:59 (decimal:89) valid:NO
  rva:0003803C mod:kernel32.dll ord:04B6 name:SizeofResource
  rva:00038040 ptr:00458C35
  rva:00038044 mod:kernel32.dll ord:054A name:lstrcmpi
  rva:00038048 mod:kernel32.dll ord:02C5 name:GlobalSize
  rva:0003804C mod:kernel32.dll ord:0553 name:lstrlen
  rva:00038050 mod:kernel32.dll ord:00A7 name>CreateProcessA
  rva:00038054 mod:kernel32.dll ord:02B0 name:GetWindowsDirectoryA
  rva:00038058 mod:kernel32.dll ord:0271 name:GetSystemDirectoryA
  rva:0003805C mod:kernel32.dll ord:01C1 name:GetCurrentDirectoryA
  rva:00038060 mod:kernel32.dll ord:0286 name:GetTempPathA

```

- 예: kernel32.dll FTThunk:0003803C NbFunc:59 valid:YES
 - o FTThunk: 그 DLL 블록의 FirstThunk RVA (IAT 블록 시작 위치)
 - o NbFunc: 그 블록 안에 함수 개수
- 예: rva:00038044 mod:kernel32.dll ord:054A name:lstrcmpi
 - o rva: 이 IAT 슬롯의 RVA
 - o mod: 이 슬롯이 가리키는 DLL.
 - o ord: 그 DLL에서의 ordinal 번호
 - o name: API 이름
- 예: rva:00038040 ptr:00458C35
 - o rva: IAT 슬롯의 위치
 - o 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	81FC 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.

CPU - main thread, module API_Redi			
00458C35	0000	ADD	BYTE PTR DS:[EAX], AL
00458C37	0000	ADD	BYTE PTR DS:[EAX], AL
00458C39	0000	ADD	BYTE PTR DS:[EAX], AL
00458C3B	0000	ADD	BYTE PTR DS:[EAX], AL
00458C3D	0000	ADD	BYTE PTR DS:[EAX], AL
00458C3F	0000	ADD	BYTE PTR DS:[EAX], AL
00458C41	0000	ADD	BYTE PTR DS:[EAX], AL
00458C43	0000	ADD	BYTE PTR DS:[EAX], AL

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.

Address	Hex dump	Appearance	ASCII
00438040	D6 CB B3 84	B3 C9 A5 4E FB 75 BB EE F5 D9 A8 71	OE³,,
00438050	E5 DD 25 5A	85 9E 1C 98 59 E4 66 7A 01 53 87 E4	âY%Z

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.

CPU - main thread, module API_Redi		
004536A6	EB 2C	JMP SHORT API_Redi.004536D4
004536A8	8B55 F4	MOV EDX,DWORD PTR SS:[EBP-C]
004536AB	8B02	MOV EAX,DWORD PTR DS:[EDX]
004536AD	25 FFFF0000	AND EAX,0FFFF
004536B2	8945 D0	MOV DWORD PTR SS:[EBP-30],EAX
004536B5	8B4D D0	MOV ECX,DWORD PTR SS:[EBP-30]
004536B8	51	PUSH ECX

Address	Hex dump
00438040	4C 4E C9 76 D0
00438050	A4 0E 04 00 8C
00438060	4F 0F 04 00 2A

'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.

004536F0	E8 9B070000	CALL API_Redi.00453E90
004536F5	83C4 0C	ADD ESP,0C
004536F8	E9 3EFFFFFF	JMP API_Redi.0045363B
004536FD	8B45 E8	MOV EAX,DWORD PTR SS:[EBP-18]
00453700	A3 C0EB4500	MOV DWORD PTR DS:[45EBC0],EAX
00453705	8B4D F8	MOV ECX,DWORD PTR SS:[EBP-8]
00453708	5B	MOV EBX,ECX

Address	Hex dump
00438040	35 8C 45 00 D0
00438050	A4 0E 04 00 8C
00438060	4F 0F 04 00 2A

'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 쓰여진 곳이고 "453E90" 함수가 리다이렉트와 관련된 함수 인 것을 알 수 있다.

=====

00453E90	55	PUSH EBP	
00453E91	8BEC	MOV EBP,ESP	
00453E93	83EC 10	SUB ESP,10	
00453E96	C745 FC 000000	MOV DWORD PTR SS:[EBP-4],0	
00453E9D	833D 34404600	CMP DWORD PTR DS:[464034],0	
00453EA4	75 0A	JNZ SHORT API_Redi.00453EB0	
00453EA6	B9 0A0000EF	MOV ECX,EF00000A	
00453EAB	E8 822B0000	CALL API_Redi.00456A32	
00453EB0	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
00453EB3	8B08	MOV ECX,DWORD PTR DS:[EAX]	
00453EB5	51	PUSH ECX	
00453EB6	8B0D 34404600	MOV ECX,DWORD PTR DS:[464034]	
00453EBC	E8 8B5C0000	CALL API_Redi.00459B4C	
00453EC1	8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	
00453EC4	837D F8 00	CMP DWORD PTR SS:[EBP-8],0	
00453EC8	74 45	JE SHORT API_Redi.00453F0F	
00453ECA	8D55 F0	LEA EDX,DWORD PTR SS:[EBP-10]	
00453ECD	52	PUSH EDX	
00453ECE	6A 04	PUSH 4	
00453ED0	6A 04	PUSH 4	
00453ED2	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
00453ED5	50	PUSH EAX	
00453ED6	FF15 2CF74500	CALL DWORD PTR DS:[45F72C]	kernel32.VirtualProtect
00453EDC	85C0	TEST EAX,EAX	
00453EDE	75 0A	JNZ SHORT API_Redi.00453EEA	

여기서 CALL함수를 들어가보면 VirtualProtect함수가 실행되는 걸 볼 수 있다.

00453ECD	52	PUSH EDX	
00453ECE	6A 04	PUSH 4	
00453ED0	6A 04	PUSH 4	
00453ED2	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
00453ED5	50	PUSH EAX	
00453ED6	FF15 2CF74500	CALL DWORD PTR DS:[45F72C]	kernel32.VirtualProtect

0012FD7C	00438040	Address = API_Redi.00438040
0012FD80	00000004	Size = 4
0012FD84	00000004	NewProtect = PAGE_READWRITE
0012FD88	0012FD8C	OldProtect = 0012FD8C

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.

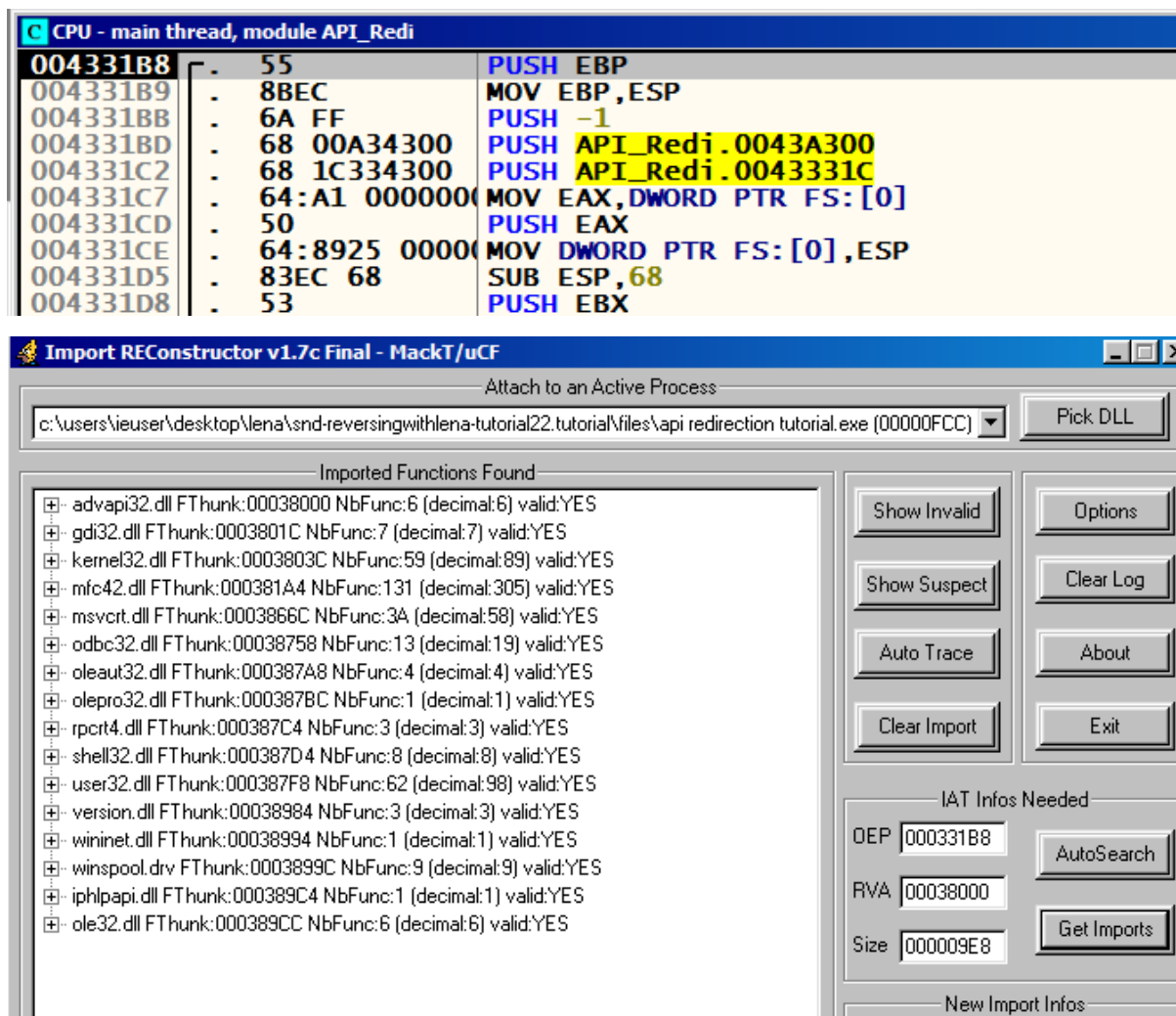
=====

004536D2	8908	MOV DWORD PTR DS:[EAX],ECX
004536D4	8B55 F0	MOV EDX,DWORD PTR SS:[EBP-10]
004536D7	81E2 FF000000	AND EDX,0FF
004536DD	85D2	TEST EDX,EDX
004536DF	74 17	JE SHORT API_Redi.004536F8
004536E1	8B45 DC	MOV EAX,DWORD PTR SS:[EBP-24]
004536E4	50	PUSH EAX
004536E5	8B0D C0EB4500	MOV ECX,DWORD PTR DS:[45EBC0]
004536EB	51	PUSH ECX
004536EC	8B55 E0	MOV EDX,DWORD PTR SS:[EBP-20]
004536EF	52	PUSH EDX
004536F0	E8 9B070000	CALL API_Redi.00453E90
004536F5	83C4 0C	ADD ESP,0C
004536F8	E9 3EFFFFFF	JMP API_Redi.0045363B
004536FD	8B45 E8	MOV EAX,DWORD PTR SS:[EBP-18]
00453700	A3 C0EB4500	MOV DWORD PTR DS:[45EBC0],EAX
00453705	8B4D F8	MOV ECX,DWORD PTR SS:[EBP-8]
00453708	8B4D F8	MOV ECX,DWORD PTR SS:[EBP-8]

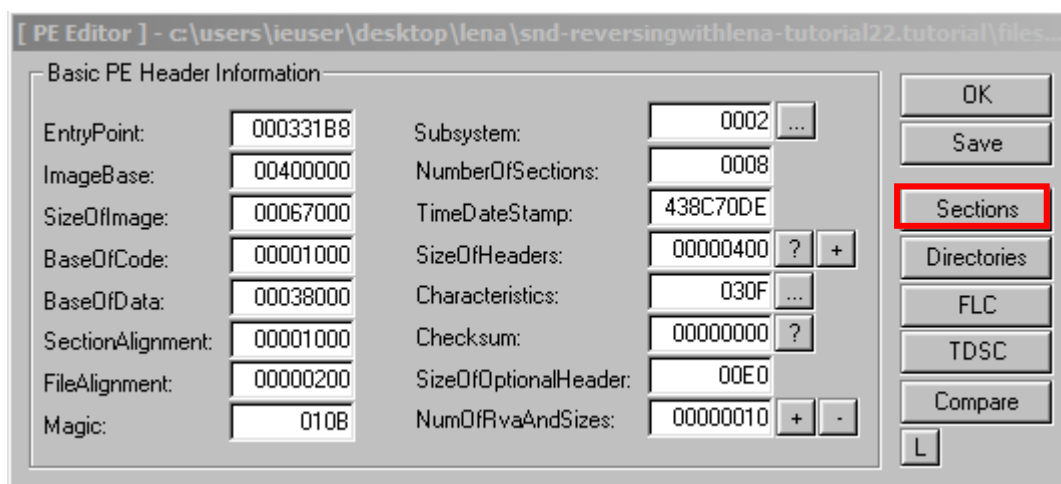
좀만 위에 올려보면 JE함수가 있는 걸 볼 수 있다. 이 함수가 실행되면 CALL함수를 뛰어넘는 걸 알 수 있다.

004536D7	81E2 FF000000	AND EDX,0FF
004536DD	85D2	TEST EDX,EDX
004536DE	EB 17	JMP SHORT API_Redi.004536F8
004536E1	8B45 DC	MOV EAX,DWORD PTR SS:[EBP-24]
004536E4	50	PUSH EAX
004536E5	8B0D C0EB4500	MOV ECX,DWORD PTR DS:[45EBC0]
004536EB	51	PUSH ECX
004536EC	8B55 E0	MOV EDX,DWORD PTR SS:[EBP-20]
004536EF	52	PUSH EDX
004536F0	E8 9B070000	CALL API_Redi.00453E90
004536F5	83C4 0C	ADD ESP,0C
004536F8	E9 3EFFFFFF	JMP API_Redi.0045363B
004536FD	8B45 E8	MOV EAX,DWORD PTR SS:[EBP-18]
00453700	A3 C0EB4500	MOV DWORD PTR DS:[45EBC0],EAX
00453705	8B4D F8	MOV ECX,DWORD PTR SS:[EBP-8]

JE -> JMP로 바꾸고 OEP로 찾아가준다.



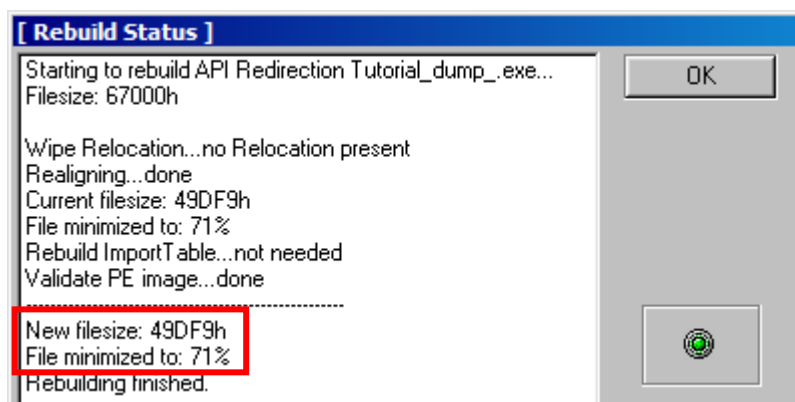
ImpRec를 실행해주면 전 부 YES로 바뀌는 걸 볼 수 있다. 해당 부분에서 Fix dump를 실행해준다.



LordPE를 이용하여 필요 없는 섹션을 정리해준다.

[Section Table]					
Name	VOffset	VSize	ROffset	RSize	Flags
0	00001000	00036D62	00001000	00036D62	E0000040
1	00038000	00009B3A	00038000	00009B3A	E0000040
2	00042000	00004A48	00042000	00004A48	E0000040
3	00047000	00003B58	00047000	00003B58	E0000040
4	0004B000	00010D8F	0004B000	00010D8F	E0000040
5	0005C000	00000CC4	0005C000	00000CC4	E0000040
6	0005D000	00007070	0005D000	00007070	E0000040
.mact	00065000	00002000	00065000	00002000	E0000060

“.mact” 섹션이 새로운 IAT를 포함하기 때문에 기존의 “4, 5, 6” 섹션은 삭제해도 무방하다.



삭제 후 리빌드 해주면 71%까지 줄어든 걸 볼 수 있다.