

# 악성코드 분석 보고서

(sand-reversingwithlana-tutorials)

2025.10.09

“**Armadillo**”는 프로그램을 실행하기 전에 압축·암호화·난독화해서 소스코드나 내부 동작을 숨기고, 크랙이나 역공학을 어렵게 만드는 프로텍터다.

※ 패커와 프로텍터 차이

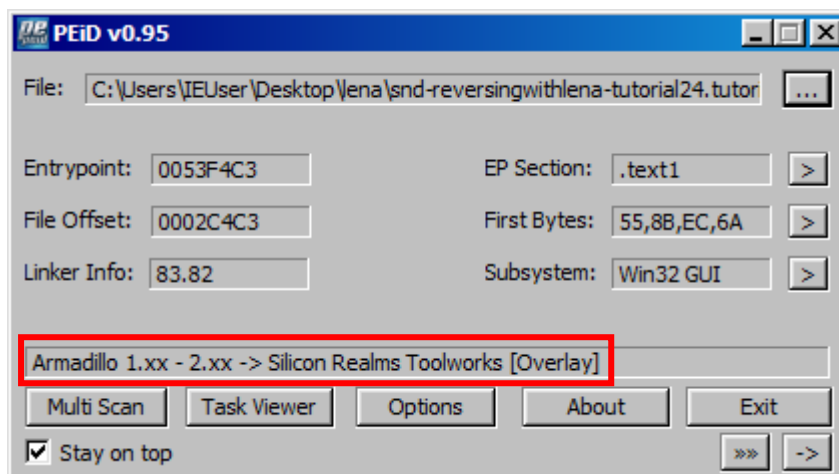
- 패커 : 실행 파일 압축기
- 프로텍터 : PE 파일을 리버싱으로부터 보호하기 위한 유틸리티

### ▶ Armadillo의 주요 특징

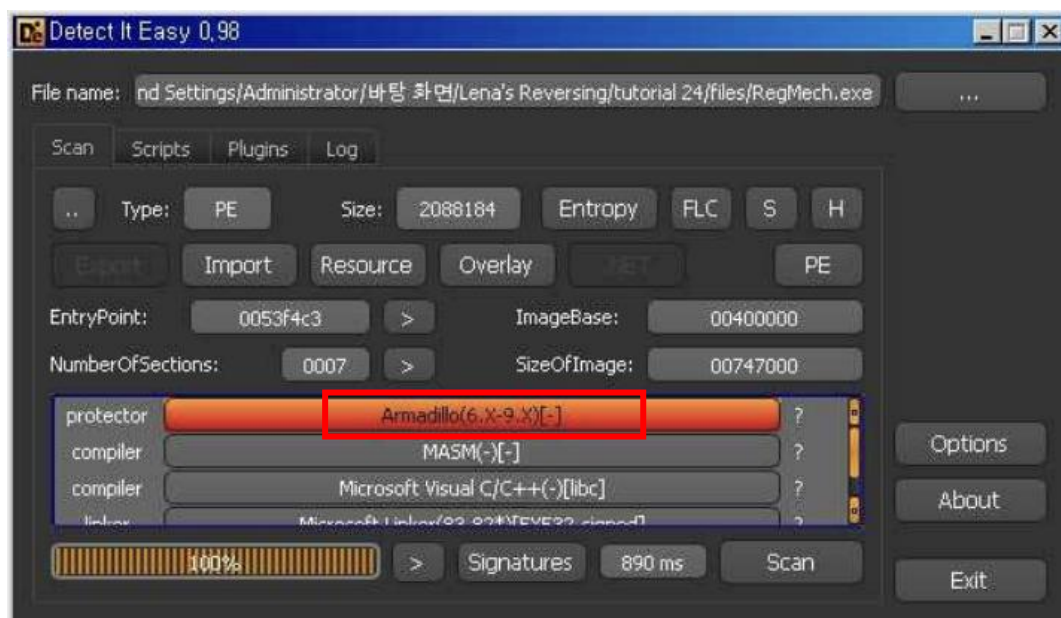
1. 코드 압축/암호화
2. Import Table(IAT) 은폐
  - 실행 파일이 사용하는 Windows API 목록을 숨기거나 난독화 → 리버스 엔지니어가 분석하기 어렵게 만들.
3. Anti-Debugging 기법
  - 디버거(OllDbg, SoftICE 등)로 실행 시 탐지해서 실행을 중단하거나 잘못된 흐름으로 유도.
4. Anti-Dumping 기법
  - 실행 중에 메모리를 덤프해도 원본이 온전히 나오지 않도록 설계.
5. 버전별 진화
  - 구버전: 비교적 쉽게 언패킹 가능.
  - 신버전: 가상화, 더 강력한 난독화, 복잡한 암호화 루틴 적용.

Armadillo 프로텍터는 "**OutputDebugStringA**" API 함수를 사용하여 올리디버그를 탐지하고 크래시를 유발시킨다. 다만 올리디버그에서 적절한 옵션을 세팅해주면 아무 일도 발생하지 않고 분석할 수 있다.

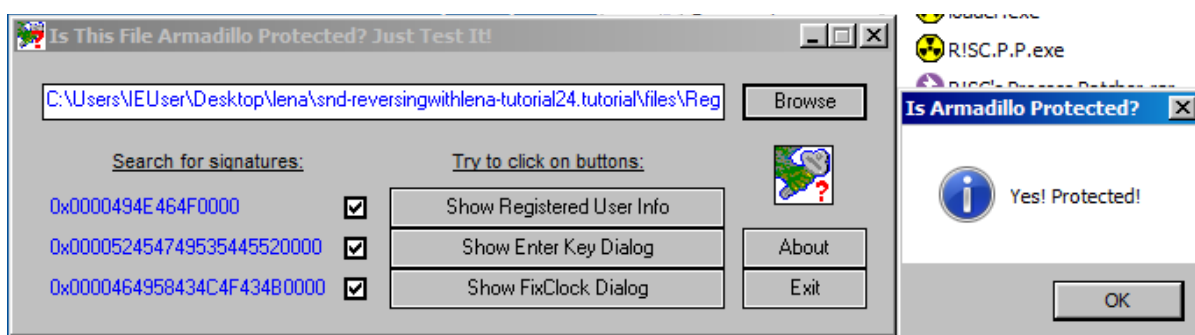
Armadillo는 "**Is it Arma ?**"라는 보조 툴을 사용해 Armadillo를 확인한다.



PEID를 보면 구 버전 Armadillo가 사용된 걸 볼 수 있다.



하지만 Lena는 신버전을 사용한다고 추정하여 다른 툴로 분석한 결과 6.x – 9.x가 사용된 걸 볼 수 있다.



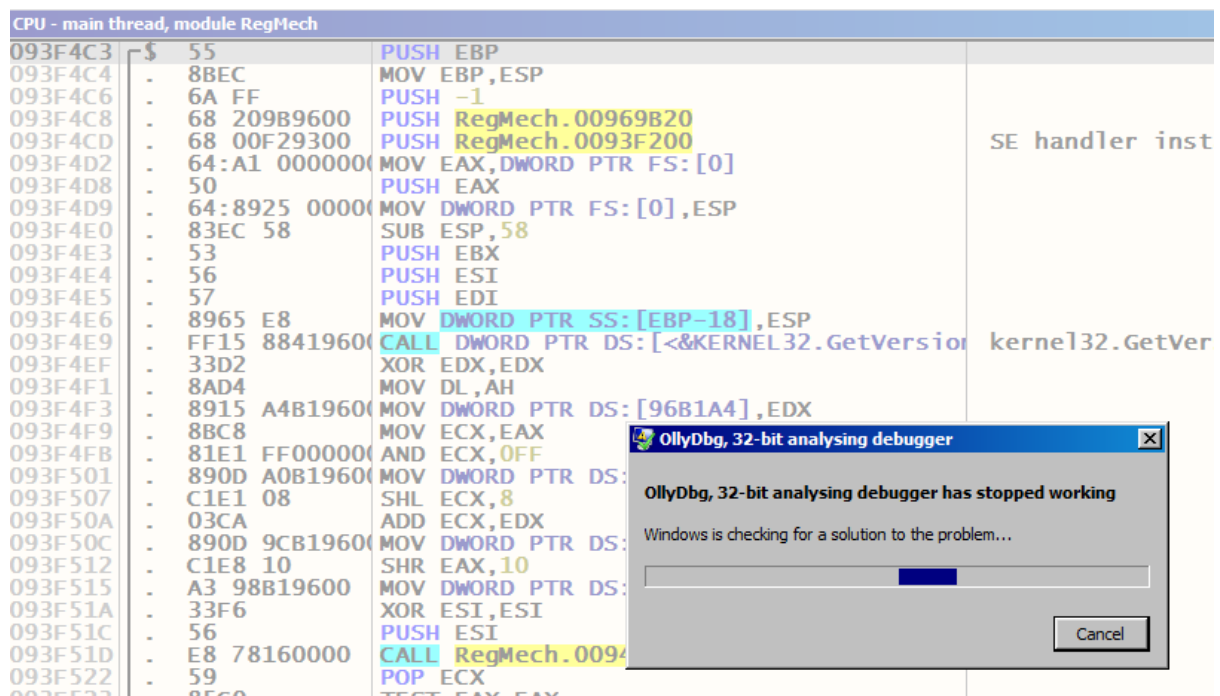
Is it Arma ? 보조 툴을 이용해서 확인한 결과 Armadillo가 사용된 걸 확인할 수 있다.

“Loader”는 로더의 대상 프로그램이 시작되고 언패킹이 될 때 까지 기다린다. 언패킹이 되면(원래 코드) 해당 프로세스의 메모리에서 사용자가 원하는 부분을 패치 시킨다.

로더의 단점은 프로그램을 실행하려고 할 때 항상 필요하다는 것이다. 일반적으로 대상 프로그램과 같은 폴더 내에 로더가 존재해야 한다. 로더를 사용함으로써 얻은 장점은 대상 프로그램이 언패킹 된 상태가 아니어도 된 다는 점이다. (상당히 많은 시간이 단축된다.) 로더와 프로그램의 관계는 종종 "부모-자식" 관계로 서술되기도 한다. (로더-부모, 대상 프로그램-자식) 다양한 GUI 툴을 이용하여 로더를 쉽게 만들 수 있다. 코딩할 필요까지 없고 그냥 "어떤 주소에서 어떤 바이트를 어떻게 패치하겠나" 라는 정도의 데이터만 입력하면 된다.

"dUP and ABEL" 같은 툴이 로더 생성에 사용되는 대표적인 툴이다.

주의할점은 패치 를 진행하고자 하는 타이밍이다. (언패킹 시간이 지난 뒤 진행) 만약 로더가 성공적 으로 작동하지 않았다면 패치 타이밍을 변경하면서 계속 시도해야 한다. (대부분의 로더 생성 툴의 타이밍은 "8000h" 정도이다.) 그리고 일반적인 로더 외에도 특별한 로더가 존재하는데 이런 로더들은 어떤 프로그램이 로더를 탐지하고 회피하기 위해 메모리 주소를 바꾸는 경우도 있기 때문에 특별하게 코딩된다.

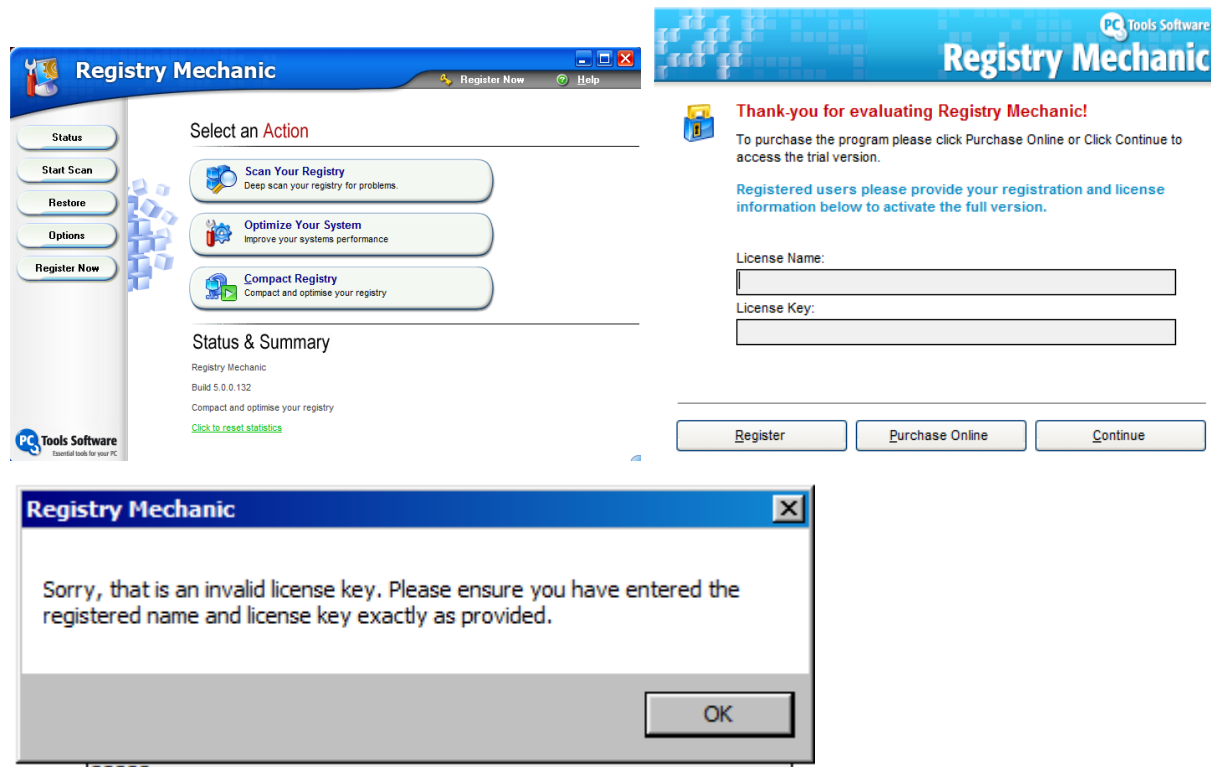


RegMe.exe를 올리디버그를 실행 후 'F9'을 눌러 실행해보면 이렇게 멈추는 걸 볼 수 있다. "OutputDebugStringA" API 때문에 멈추는 걸 알 수 있다.

근데 해당 문제가 32bit 환경이면 실행이 안되는 걸 볼 수 있다.. 하지만 Win 7, XP 둘

다 32bit만 가지고 있기 때문에 해당 문제는 풀 수가 없어 Lena 영상과 다른 사람이 풀  
걸로 공부하기로 한다.

## 1) RegMech.exe – Loader 사용



올리디버그로 실행 후 'F9'을 실행하면 해당 창이 나오는 걸 알 수 있고 레지스터 키 입력 창이 나오는 걸 알 수 있다. 키 입력 후 'Register'를 누른 후 값이 틀리면 실패 창이 나오는 걸 볼 수 있다.

그럼 올리디버그로 돌아가서 Alt+'F9' 눌러서 현재 실행되고 있는 코드로 넘어가 일시 중지해준다.

프로그램 분석을 위해 등록 루틴을 리버싱 할 것이다. ( 물론 다른 부분을 리버싱 해도 되지만 패킹된 바이너리가 실행 될 때 패치를 하는 것이 가능하다는 것을 보여주기 위해 등록 루틴을 선택했다. )

등록 루틴을 찾기 위해서는 여러 가지 방법이 존재한다.

- "Return to user" ( 바이너리 일시 중지 후 => Alt + F9 )
- "Call Stack" ( Alt + K )

이번에는 콜스택을 이용하여 등록루틴을 찾을 것이다.

Address	Stack	Procedure / arguments	Called from
0012B2B8	77D049C4	USER32.77D0757B	USER32.77D049BF
0012B2E0	77D1A956	USER32.77D0490E	USER32.77D1A951
0012B5A0	77D1A2BC	USER32.SoftModalMessageBox	USER32.77D1A2B7
0012B6F0	77D1A10B	USER32.77D1A147	USER32.77D1A106
0012B75C	733DF6B2	Includes USER32.77D1A10B	MSUBUM60.733DF6B0
0012B79C	733DF52E	Includes MSUBUM60.733DF6B2	MSUBUM60.733DF52B
0012B7C4	733DF829	MSUBUM60.733DF414	MSUBUM60.733DF824
0012B7F4	733D3BF0	MSUBUM60.733DF798	MSUBUM60.733D3BEB
0012B858	7344D07A	MSUBUM60.733D3963	MSUBUM60.7344D075
0012B8D0	0088BD5B	? MSUBUM60.rtcMsgBox	RegMech.0088BD55

현재 실패했다는 메시지 박스가 띄어진 상태에서 스택을 보니까 rtcMsgBox 함수가 사용된 걸 볼 수 있다.

0088BCF5	. 0F84 81000000	JE 0088BD7C	0088BD7C
0088BCF8	. C745 FC 090000	MOV DWORD PTR [EBP-4],9	
0088BD02	. C745 90 040000	MOV DWORD PTR [EBP-70],80020004	
0088BD09	. C745 88 0A0000	MOV DWORD PTR [EBP-78],0A	
0088BD10	. C745 A0 040000	MOV DWORD PTR [EBP-60],80020004	
0088BD17	. C745 98 0A0000	MOV DWORD PTR [EBP-68],0A	
0088BD1E	. C745 B0 040000	MOV DWORD PTR [EBP-50],80020004	
0088BD25	. C745 A8 0A0000	MOV DWORD PTR [EBP-58],0A	
0088BD2C	. C785 60FFFFFF	MOV DWORD PTR [EBP-A0],906134	
0088BD36	. C785 58FFFFFF	MOV DWORD PTR [EBP-A8],4008	
0088BD40	. 8D45 88	LEA EAX,[EBP-78]	
0088BD43	. 50	PUSH EAX	
0088BD44	. 8D40 98	LEA ECX,[EBP-68]	
0088BD47	. 51	PUSH ECX	
0088BD48	. 8D55 A8	LEA EDX,[EBP-58]	
0088BD4B	. 52	PUSH EDX	ntdll.KiFastSystemCallRet
0088BD4C	. 6A 00	PUSH 0	
0088BD4E	. 8D85 58FFFFFF	LEA EAX,[EBP-A8]	
0088BD54	. 50	PUSH EAX	
0088BD55	. FF15 18114000	CALL [401110]	MSUBUM60.rtcMsgBox

0x0088BD5B로 이동 후 위로 올려보면 JE 조건문이 나오는 걸 볼 수 있다. 근데 rtcMsgBox함수가 나오지 않기 하기 위해서 ZF=1이 되어야 점프해서 뛰어넘는 걸 알 수 있다.

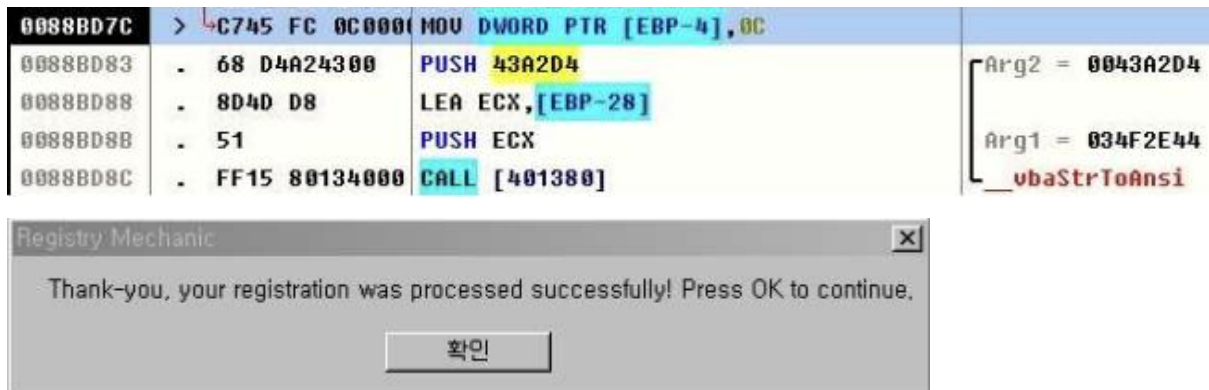
0088BCEC	. 0FBF95 0CFFFFFF	MOVSX EDX,WORD PTR [EBP-F4]	
0088BCF3	. 85D2	TEST EDX,EDX	ntdll.KiFastSystemCallRet
0088BCF5	. 0F84 81000000	JE 0088BD7C	0088BD7C

좀 더 위 코드를 보면 EDX 레지스터에 저장되어 조건문에 사용되는 것을 볼 수 있다. 0x0088BCEC에 BP를 설치하고 프로그램을 재시작하여 분석을 진행해야 한다.









ZF=1로 바꾼 후 실행하면서 JE가 실행되어 점프하는 걸 볼 수 있다.

여기서 "JE" => "JMP"로 패치하면 되지만 해당 바이너리의 경우가 패킹된 상태이기 때문에 그냥 패치해 버리면 동작하지 않게 된다.(패킹된 상태인 걸 어떻게 아냐면 처음에 실행할 때와 현재 위치까지 왔을 때 코드를 비교해보면 다르기 때문에 패킹된 상태인 걸 알 수 있다.) 이런 경우에는 간단한 로더를 이용하는 것이 제일 좋은 방법이다.

※ 여기서 바이너리 코드인지 언패킹 코드인지 아는 방법

- 언패킹 코드(unpacking code): 패커가 원본 프로그램을 메모리로 복호화하는 루프나 스텝 코드.
- 바이너리 코드(binary code): 복호화된 뒤의 원래 실행 프로그램(실제 명령어) 부분

002B0000	00006000				Priv	RW	RW
00400000	00001000	RegMech		PE header	Imag	R	RWE
00401000	00505000	RegMech	.text		Imag	R	RWE
00906000	0000E000	RegMech	.data		Imag	R	RWE
00914000	00040000	RegMech	.text1	code	Imag	R	RWE
00954000	00010000	RegMech	.adata		Imag	R	RWE
00964000	00020000	RegMech	.data1	data, imports	Imag	R	RWE
00984000	00130000	RegMech	.pdata		Imag	R	RWE
00AB4000	00093000	RegMech	.rsrc	resources	Imag	R	RWE
00B50000	00101000				Map	R	R

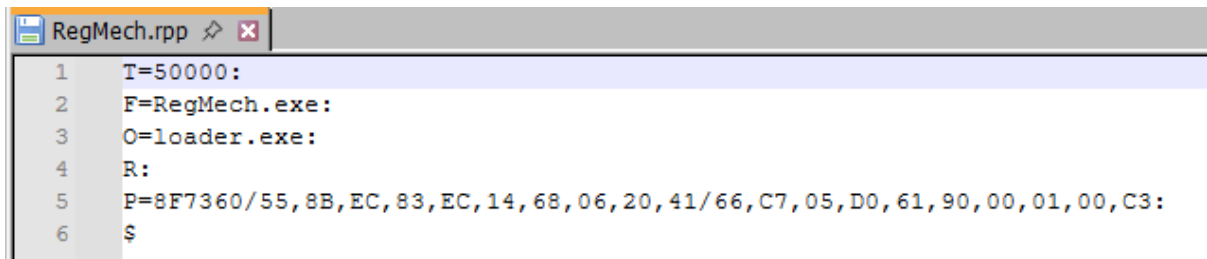
실패 메시지박스가 나오는 기준으로 볼 때 주소 0x0088BD5B가 .text에 속하기 때문에 바이너리 코드인지 알 수 있다.

현재 .exe파일의 EP을 보면 0x93F4C8인 걸 알 수 있다. .text1에 속하는 걸 알 수 있다. 그럼 .text1이 전체다 스텝이라고 생각 할 수 있다. 하지만 스텝+실행코드가 포함될 수 있다. 하지만 contains가 .text1에 code라고 적혀있어 이 부분에 실행 코드가 속해있을 가능성이 크다.

PE파일의 기본 섹션은 .text, .data, .rdata, .rsrc, .edata, .idata, .reloc 이다.

이 목록에 없는 섹션 이름이 보이면, "링커가 추가했거나 패커/프로텍터가 새로 만든 섹

션" 이라고 보면된다.



```
1 T=50000:
2 F=RegMech.exe:
3 O=loader.exe:
4 R:
5 P=8F7360/55,8B,EC,83,EC,14,68,06,20,41/66,C7,05,D0,61,90,00,01,00,C3:
6 $
```

RegMech.exe과 같은 곳에 RegMech.rpp파일을 만들어 놓는다.

- T = 패치할 타이밍 ( 단위 : ms )
- F = 패치 대상 바이너리
- O = 생성 될 로더 바이너리 이름
- R = 쓰레드 재시작
- P = 패치 진행 주소 / 원본 명령어 / 변경하고자 하는 명령어

즉, 위와 같은 스크립트 파일을 이용하면 "88BCF5" 주소의 "0F8481000000" 명령어가 "E98200000090" 명령어로 패치된다.

그리고 R!SC.P.P.exe(로더)와 RegMech.exe, RegMech.rpp를 같은 곳에 위치한 후 로더를 이용하여 RegMech.rpp를 불러와 loader.exe 파일을 생성한다. loader.exe파일을 실행하면 RegMech 프로그램이 실행된다.

register 버튼을 눌러 아무값이나 넣고 등록해도 성공적으로 등록이 되는 것을 볼 수 있다.

## 2) RegMech.exe – 바이너리 언패킹

패커 코드 내에 숨겨진 무언가가 있을 수도 있기 때문에 리버서는 패킹된 바이너리를 언패킹 하는 법을 알아야 한다고 했다.

"Armadillo" 프로텍터는 해당 API를 이용하여 디거버를 종료시키는 기능을 지원한다. 일종의 안티 디버깅인데 우선 이를 우회하기 위해서는 올리디버그의 "Olly Advacned" 플러그인을 사용하면 된다. 만약 플러그인을 사용하지 않고 직접 하려면

"OutputDebugStringA" 함수에 BP를 설치하고, 내부 분석을 하여 **"PUSH XXXX"** 대신에 **"RET"**로 패치하면 된다.

00400000	00001000	RegMech		PE header	Imag	R	RWE
00401000	00505000	RegMech	.text		Imag	R	RWE
00906000	0000E000	RegMech	.data		Imag	R	RWE
00914000	00040000	RegMech	.text1	code	Imag	R	RWE
00954000	00010000	RegMech	.adata		Imag	R	RWE
00964000	00020000	RegMech	.data1	data, imports	Imag	R	RWE
00984000	00130000	RegMech	.pdata		Imag	R	RWE
00AB4000	00093000	RegMech	.rsrc	resources	Imag	R	RWE

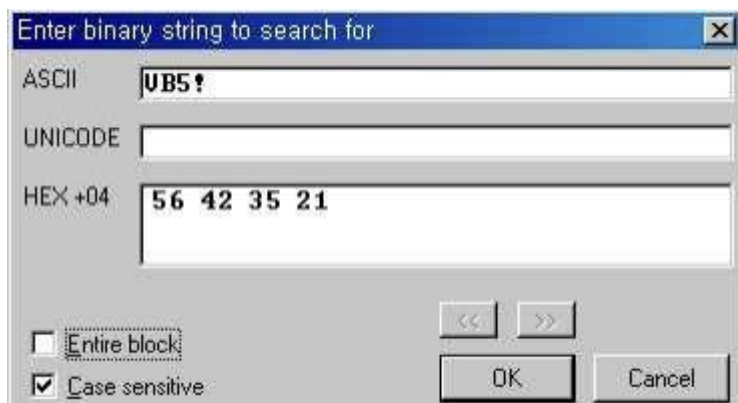
추가 생성한 섹션(test1,adata,data1,pdata)을 볼 수 있는데 추가적으로 어떤 dll 파일들이 임포트 되는지 보기 위해 스크롤을 아래로 내리면서 분석한다.

73370000	00001000 (4096.)	MSVBUM60		PE header
73371000	000FC000 (1032192.)	MSVBUM60	.text	code,imports,exports
73460000	00000000 (53248.)	MSVBUM60	ENGINE	data
73470000	00008000 (32768.)	MSVBUM60	.data	
73482000	00031000 (200704.)	MSVBUM60	.rsrc	resources
73483000	00010000 (65536.)	MSVBUM60	.reloc	relocations

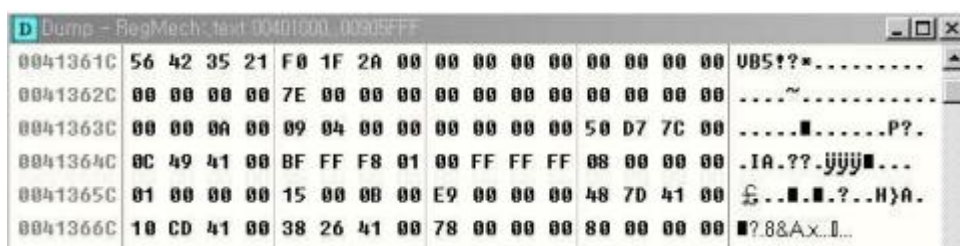
"MSVBVM60.DLL" 파일이 임포트 되는 것을 볼 수 있다. 해당 DLL 이 임포트 되는 것으로 보아 해당 바이너리는 "Visual Basic(이하 VB)" 언어로 작성된 것을 알 수 있다.

바이너리가 VB로 작성되어 있다면 VB 언어의 특성 때문에 OEP를 찾는 것은 굉장히 쉬워진다. 모든 VB 바이너리는 다음과 같은 OEP 형태를 가진다.

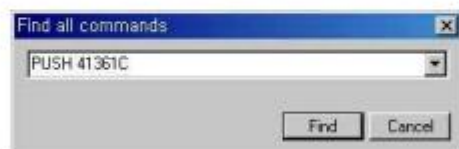
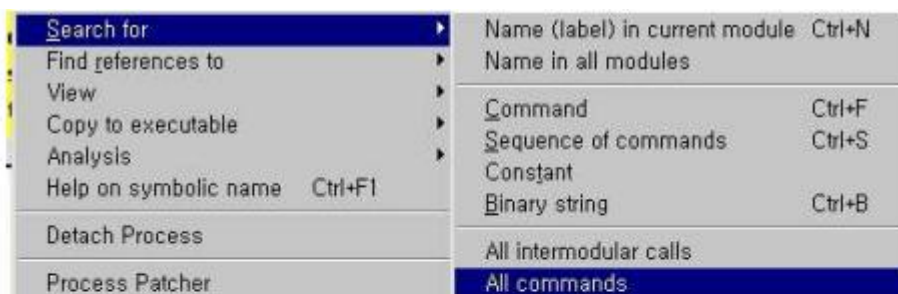
- "VB" 문자열이 스택에 저장된다.
- 저장된 뒤 VB DLL로 분기하기 위해 "ThunRTMain" 이 호출된다.



메모리 맵 창에서 위와 같이 "VB5!" 문자열을 검색한다. 사용되는 VB DLL 파일에 따라 약간씩 문자열이 달라서 검색 결과가 없을 수도 있는데 그럴 때는 "VB"를 검색 하면 된다.



"VB5!" 문자열의 주소를 저장함으로써 "OEP" 코드에서 해당 문자열이 스택에 저장될 것이다. 즉 OEP를 찾기 위해 "PUSH 41361C" 명령어를 찾으려면 된다.



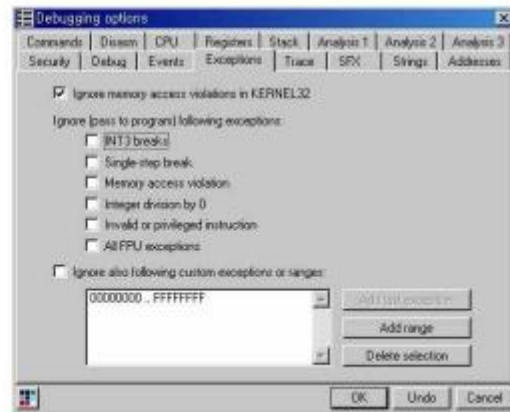
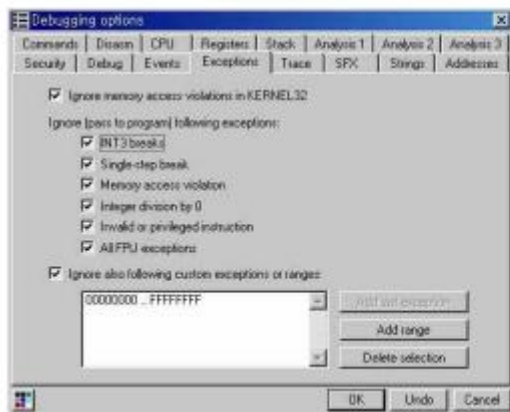
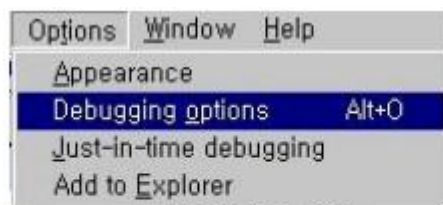
Address	Disassembly	Comment
00401000	SBB [EDX+98417343],EBX	(Initial CPU selection)
0041262C	PUSH 41361C	

위 3개의 사진에서 보듯이 "문자열 검색 메뉴 - 특정 명령어 검색 - 검색 결과"를 하면 최종적으로 "0041262C" 주소에서 해당 명령어를 실행하고 있는 것을 볼 수 있다. 해당 주소로 이동하면 바이너리의 "OEP"를 찾을 수 있을 것이다.

0041260C	- FF25 5C104000	JMP [40105C]	MSVBUM60.EVENT_SINK_Invoke
00412612	- FF25 38114000	JMP [401138]	MSVBUM60.GetMemObj
00412618	- FF25 4C124000	JMP [40124C]	MSVBUM60.PutMemObj
0041261E	- FF25 64124000	JMP [401264]	MSVBUM60.SetMemObj
00412624	- FF25 58134000	JMP [401358]	MSVBUM60.ThunRTMain
0041262A	0000	ADD [EAX],AL	
0041262C	68 1C364100	PUSH 41361C	
00412631	E8 EFFFFFFF	CALL 00412624	JMP to MSVBUM60.ThunRTMain
00412636	0000	ADD [EAX],AL	
00412638	68 00000030	PUSH 30000000	

이동된 주소를 보니 VB로 작성된 바이너리의 OEP 형태임을 알 수 있다. 대부분의 VB OEP는 "PUSH XXXX -> CALL ThunRTMain" 형태를 띈다.

"VB OEP"를 찾기 위해 위 사진과 같이 간단한 방법을 이용할 수 있다. 하지만 레벨 20에서 "OEP"를 찾는 일반적인 방법은 3가지가 존재한다고 했었다. 지금까지는 트레이싱과 "ESP Trick"을 이용하였다. 이번에 3번째 방법을 보여줄 것이다. (Exceptions 을 이용한 OEP 찾기)



디버거 옵션에 들어가서 "Exception" 탭을 보면 원래는 좌측 사진처럼 모든 것이 체크되어 있을 것이다. "Exception"을 이용하여 OEP를 찾으려면 우측 사진처럼 변경해야 한다.

0093F4C3	\$ 55	PUSH EBP
0093F4C4	. 8BEC	MOV EBP,ESP
0093F4C6	. 6A FF	PUSH -1
0093F4C8	. 68 209B9600	PUSH 969B20
0093F4CD	. 68 00F29300	PUSH 93F200

이제 프로그램을 재시작하여 위 사진과 같이 "EP" 부분에서 멈춘다. 그리고 프로그램을



정상적으로 실행하기 위해 "Shift+F9"를 누르면서 발생하는 "Exceptions"을 무시한다. 단, 몇번 눌렀는지 기억해야 한다. 분석 환경 마다 다르지만 이번에는 "23"번 눌렀을 때 프로그램 로딩이 시작되었다.

그리고 다시 프로그램을 재시작 한 뒤 위 사진에서 볼 수 있는 "EP" 까지 실행하였다. OEP를 쉽게 찾기 위해서 이전과 동일한 작업을 하지만 "Shift+F9"를 한 번 덜 누를 것이다. 그러면 실행하고자 하는 명령어가 언패킹 루틴과 "OEP" 사이 쯤 위치하게 된다. 그리고 코드 섹션에 BP를 설치하고 프로그램을 실행하면 OEP에 멈추게 될 것이다.

0115CA3E	8900	MOV [EAX],EAX
0115CA40	90	NOP
0115CA41	E9 57010000	JMP 0115CB9D
0115CA46	FF75 EC	PUSH DWORD PTR [EBP-14]
0115CA49	E8 42F5FFFF	CALL 0115BF90
0115CA4E	59	POP ECX
0115CA4F	C3	RETN
0115CA50	8B65 E8	MOV ESP,[EBP-18]
0115CA53	70 07	J0 SHORT 0115CA5C
0115CA55	7C 03	JL SHORT 0115CA5A
0115CA57	EB 05	JMP SHORT 0115CA5E
0115CA59	E8 74FBEBF9	CALL FB01C5D2
0115CA5E	A1 A4231701	MOV EAX,[11723A4]
0115CA63	85C0	TEST EAX,EAX
0115CA65	0F84 0C010000	JE 0115CB77
0115CA6B	8B50 04	MOV EDX,[EAX+4]
0115CA6E	8B0D 90241701	MOV ECX,[1172490]

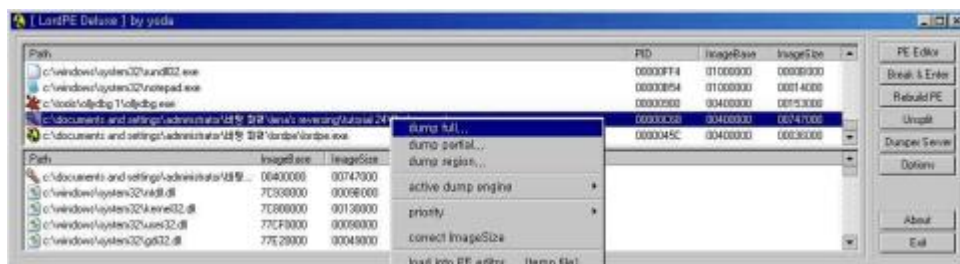
프로그램 재시작 - "Shift + F9" 22번 클릭하면 위 사진의 주소에서 멈추게 된다.

00400000	00001000 (4096.)	RegMech		PE header	Inag 0100100 R
00401000	00505000 (5263360.)	RegMech	.text	code	Actualize
00906000	0000E000 (57344.)	RegMech	.data		View in Disassembler Enter
00914000	00040000 (262144.)	RegMech	.text1	SFX	Dump in CPU
00954000	00010000 (65536.)	RegMech	.adata		Dump
00964000	00020000 (131072.)	RegMech	.data1	data,import	Search: Ctrl+B
00984000	00130000 (1245184.)	RegMech	.pdata		Search next: Ctrl+L
00A04000	00093000 (602112.)	RegMech	.rsrc	resources	Set break-on-access F2
00B50000	00005000 (20480.)				Set memory breakpoint on access

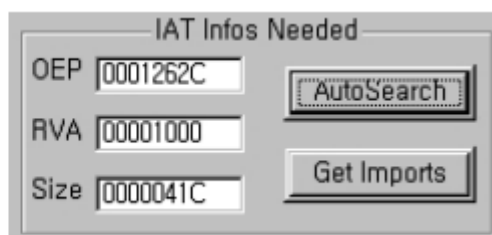
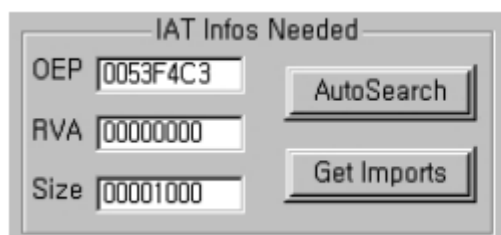
멈추면 메모리 맵 창에서 코드 섹션 부분에 BP를 설치하고 프로그램을 계속 실행시킨다.

0041261E	- FF25 64124000	JMP [401264]	HSUBUH60.SetHenObj
00412624	- FF25 58134000	JMP [401358]	HSUBUH60.ThunRTMain
0041262A	0000	ADD [EAX],AL	
0041262C	68 1C364100	PUSH 41361C	
00412631	E8 EFFFFFFF	CALL 00412624	JMP to HSUBUH60.ThunRTMain
00412636	0000	ADD [EAX],AL	
00412638	68 00000030	PUSH 30000000	

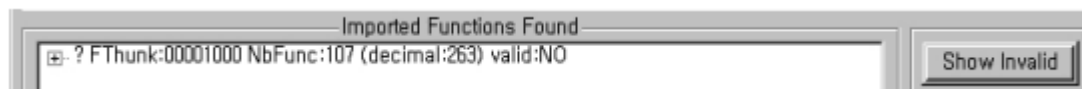
실행되면 위 사진에서 보는 것처럼 OEP 부분에서 멈추게 된다. OEP를 찾았으니 이제 덤 프를 뜨고 임포트 테이블을 수정해야 한다.



"LordPE" 툴에서 덤프를 뜨고자 하는 프로세스를 선택하고 "dump full" 기능을 이용하여 덤프를 뜬다. 뜨고 나으면 이제 임포트 테이블을 수정해야 한다.



"ImportREC"에서 대상 프로세스를 선택하면 초기 세팅 값은 좌측 사진처럼 되어있다. OEP 값을 이전에 찾았던 "OEP" 주소로 변경하고 "AutoSearch" 버튼을 누르면 우측 사진처럼 "RVA / Size" 값이 세팅된다.



자동으로 값이 세팅되고 "Get Imports" 버튼을 누르면 위 사진처럼 임포트 되는 모듈 한 개가 "valid : NO" 상태가 된다. 우측의 "Show Invalid" 버튼을 누르면 어떤 주소가 문제인지 보여준다.

```
rva:00001044 mod:msvbvm60.dll ord:01B5 name:__vbaStrVarMove
rva:00001048 ptr:0113F261
rva:0000104C mod:msvbvm60.dll ord:00F5 name:__vbaFreeVarList
```

위 사진이 문제 되는 부분이다. 이제 디버거에서 문제가 되는 "API"를 조사해야 한다. "00401048 (ImageBase 00400000 + 1048)"을 호출하는 주소는 "0113F261" 이다. 직접 해당 주소로 가도 되지만 점프 테이블 부터 조사하여 어떤 API가 문제되는지 볼 것이다.

```
00412246 - FF25 48104000 JMP [401048]
```

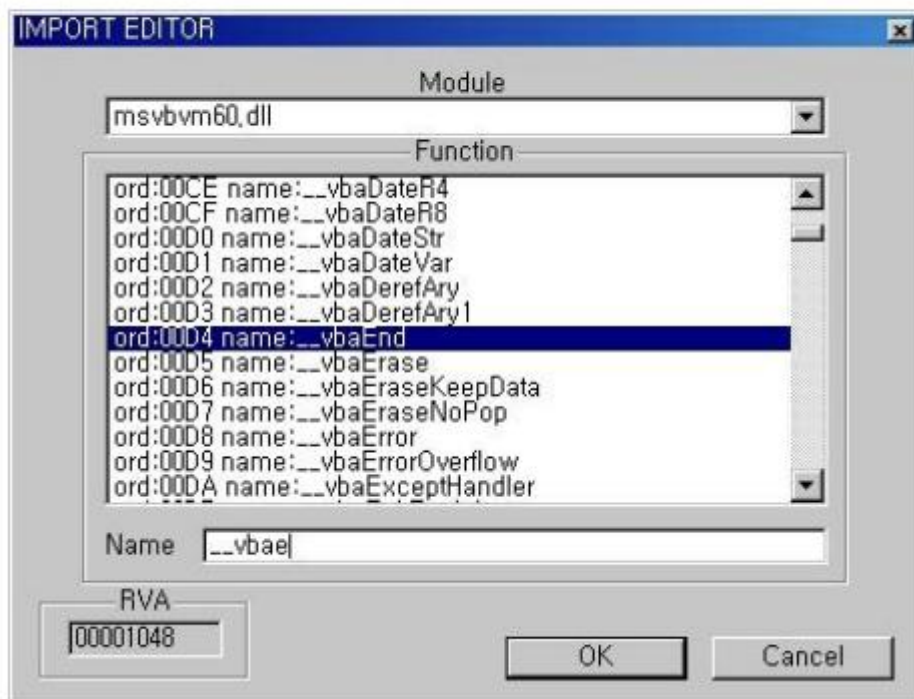
점프 테이블을 보니 "412246" 주소에서 "401048" 함수로 분기되는 것을 볼 수 있다.



0113F261	55	PUSH EBP	
0113F262	8BEC	MOV EBP,ESP	
0113F264	6A FF	PUSH -1	
0113F266	68 08261601	PUSH 1162608	
0113F268	68 50001601	PUSH 1160050	JMP to msuirt._except_handl
0113F270	64:A1 00000000	MOV EAX,FS:[0]	
0113F276	50	PUSH EAX	RegMech.00964370
0113F277	64:8925 00000000	MOV FS:[0],ESP	
0113F27E	83EC 10	SUB ESP,10	
0113F281	53	PUSH EBX	
0113F282	56	PUSH ESI	RegMech.0096AA30
0113F283	57	PUSH EDI	RegMech.00A1262C
0113F284	8965 E8	MOV [EBP-10],ESP	
0113F287	33F6	XOR ESI,ESI	RegMech.0096AA30
0113F289	68 64741601	PUSH 1167464	ASCII "MSVBVM60.DLL"
0113F28E	FF15 E0201601	CALL [11620E8]	kerne132.GetModuleHandleA
0113F294	85C0	TEST EAX,EAX	RegMech.00964370
0113F296	74 0E	JE SHORT 0113F2A6	
0113F298	68 58741601	PUSH 1167458	ASCII "__vbaEnd"
0113F29D	50	PUSH EAX	RegMech.00964370
0113F29E	FF15 EC201601	CALL [11620EC]	kerne132.GetProcAddress
0113F2A4	8BF0	MOV ESI,EAX	RegMech.00964370

호출되는 "401048" 함수를 내부 분석해보니 위 사진처럼 "GetModuleHandleA" 함수를 통하여 "MSVBVM60.DLL" 파일 주소를 구하고 "GetProcAddress" 함수를 통하여 "\_\_vbaEnd"(즉시 중단 성격이라 악성/보호 코드에서 분석 방해 트리거로 쓰임) 함수의 주소를 구하고 있다. 해당 함수가 바로 문제되는 부분이었다.

```
rva:00001044 mod:msvbvm60.dll ord:01B5 name:__vbaStrVarMove
rva:00001048 ptr:0113F261
rva:0000104C mod:msvbvm60.dll ord:00F5 name:__vbaFreeVarList
```



문제가 되던 주소를 더블 클릭하면 직접 수정할 수 있는 창이 나타난다. 디버거에서 알아냈던 "\_\_vbaEnd" 로 세팅한다.

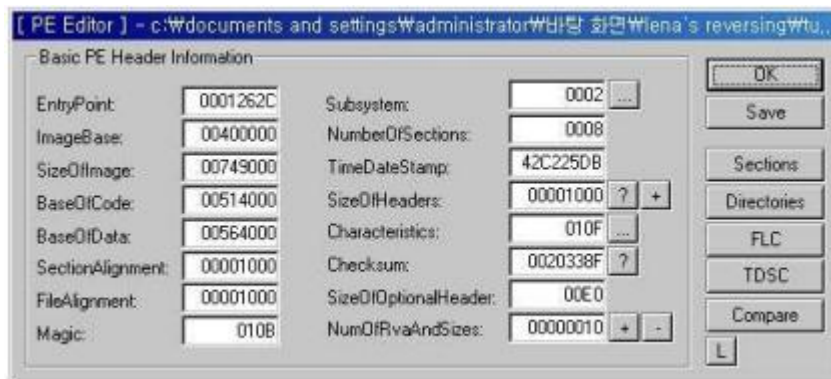
```

rva:00001044 mod:msvbvm60.dll ord:01B5 name:__vbaStrVarMove
rva:00001048 mod:msvbvm60.dll ord:00D4 name:__vbaEnd
rva:0000104C mod:msvbvm60.dll ord:00F5 name:__vbaFreeVarList

```



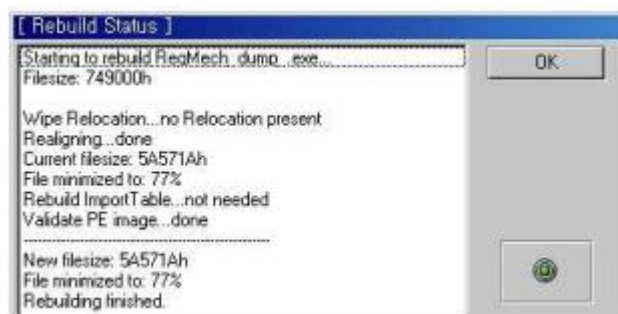
수정하고 다시 "Get Imports"를 누르면 문제되던 DLL 상태가 "valid : YES"로 된다. 이제 이전에 덤프를 떠났던 바이너리로 임포트 테이블을 수정한다.



수정하면 "LordPE"에서 "PE Editor" 기능으로 해당 바이너리를 열어서 프로텍터에서 추가했던 섹션을 제거해야 한다. (안 해도 되지만 사이즈를 줄이기 위해서 진행)

[ Section Table ]					
Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00504538	00001000	00504538	E0000020
.data	00506000	0000D800	00506000	0000D800	C0000040
.text1	00514000	00040000	00514000	00040000	E0000020
.adata	00554000	00010000	00554000	00010000	E0000020
.data1	00564000	00020000	00564000	00020000	C0000040
.pdata	00584000	00130000	00584000	00130000	C0000040
.rsrc	00684000	00093000	00684000	00093000	40000040
.mact	00747000	00002000	00747000	00002000	E0000060

"test1, adata, data1, pdata" 섹션을 제거한다. (마우스 우측 클릭 - Wipe Section)



섹션을 제거했으니 PE 구조를 리빌드해야 한다. 리빌드 하면 사이즈가 약 "77%" 정도로 줄어든다.