악성코드 분석 보고서

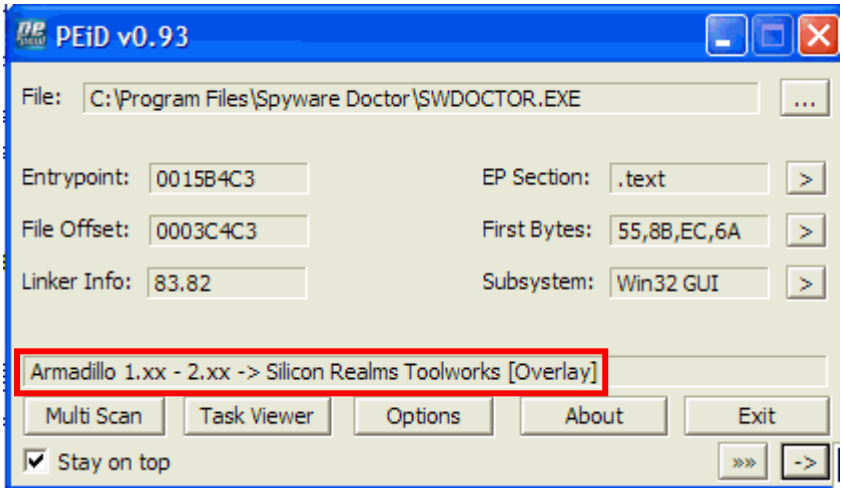
(sand-reversingwithlena-tutorials)

2025.10.13

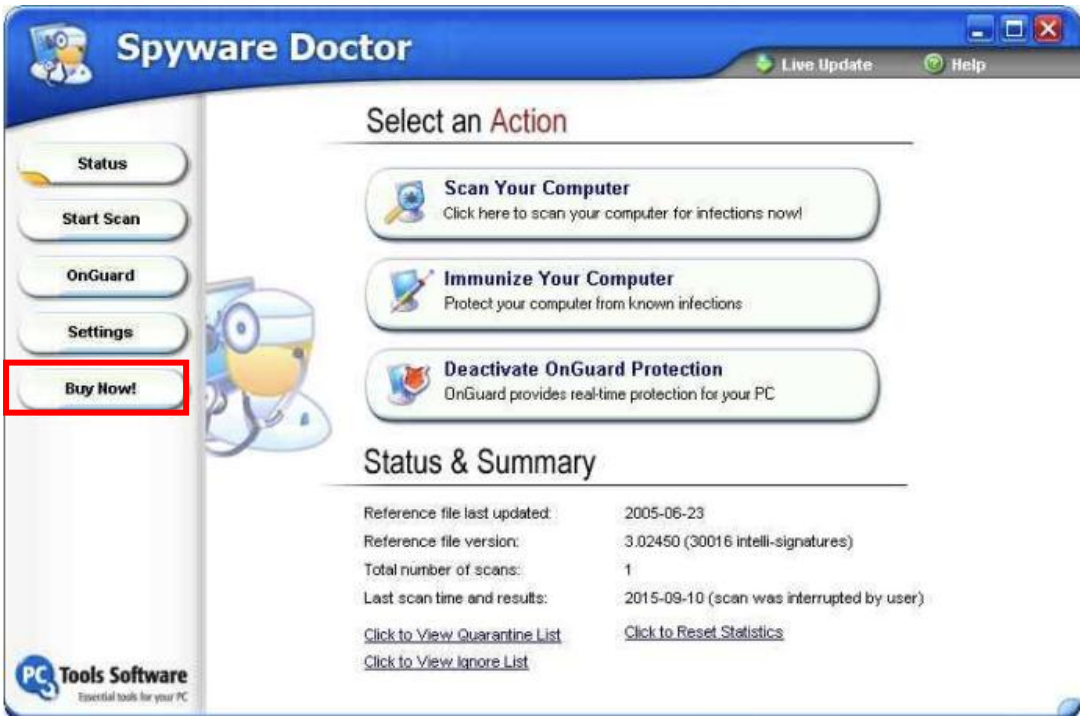
25

24와 마찬가지로 실행이되지 않아 lena영상 및 다른 사용자의 글을 참고하여 작성하였다.

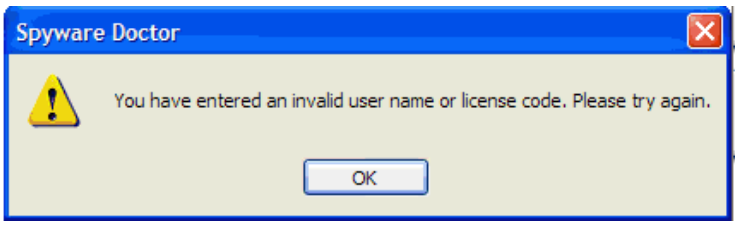
1. SpywareDocter.exe – Loader 사용



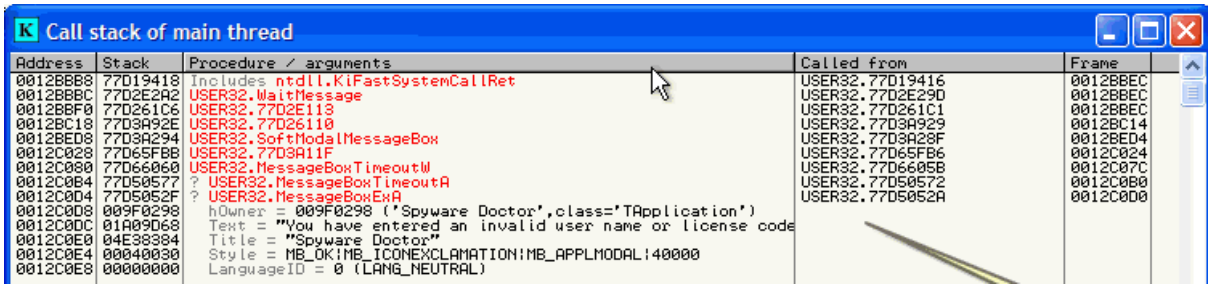
아르마딜로 패커를 사용한 걸 볼 수 있다.



레지스터 값을 입력하는 창이 생기는데 값을 입력해주고 'Register Now'를 눌러준다.



이렇게 입력한 값이 틀리다는 걸 알 수 있다.



올리디버그로 돌아간 후 콜스택을 보면 정보가 없는 걸 확인할 수 있다.

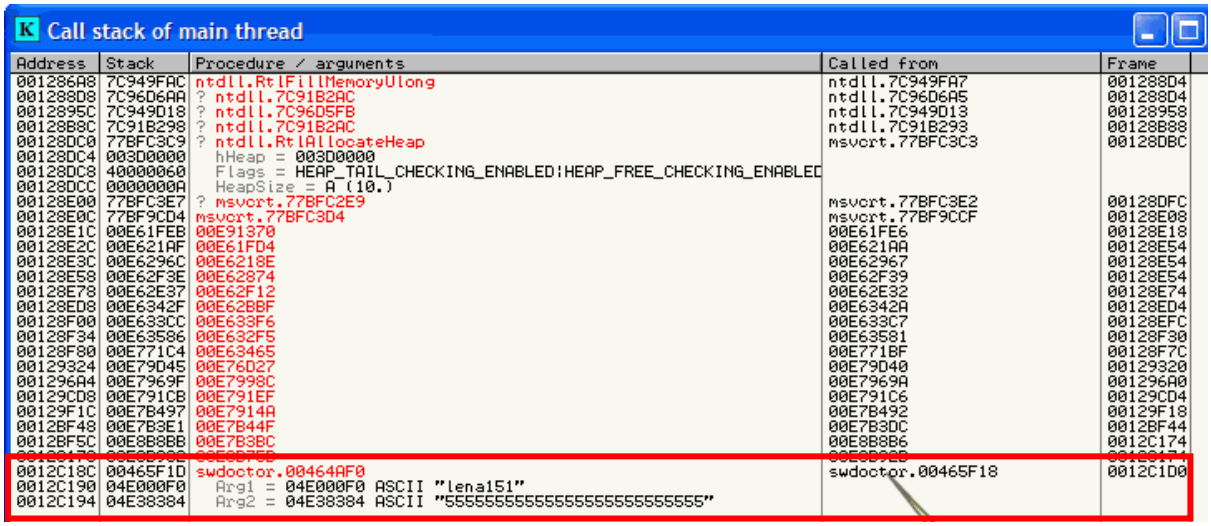
하지만 키 값을 입력한 후 'Register Now'를 누르면 bad boy창이 늦게 뜨는 걸 알 수 있다. 그래서 오래 걸리는 시간을 이용하여 일시정지를 해주고 콜 스택을 열어보도록 한다.

* 올리디버그 단축키

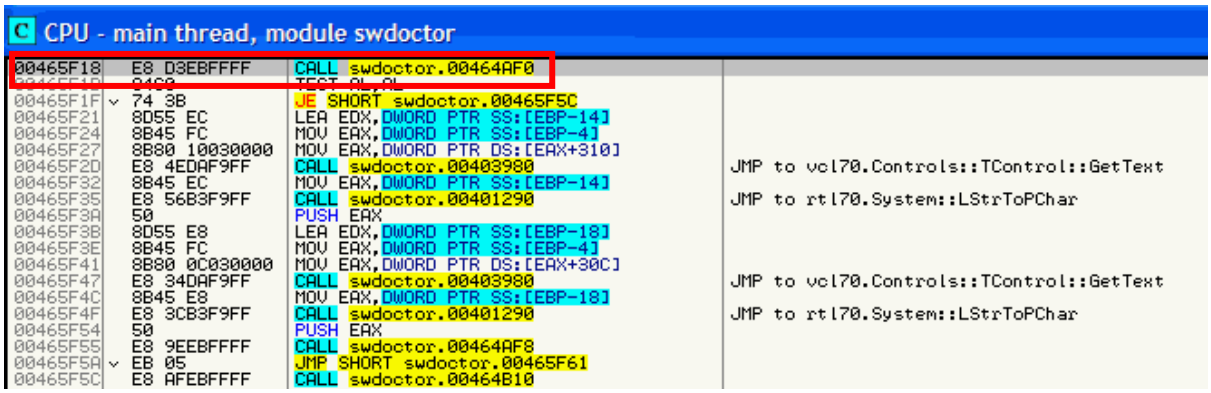
- Alt + F9 : 시스템/패커 루틴을 건너뛰고 사용자 코드 진입점으로 실행 (커서가 있는 부분에서 멈춤)

- Ctrl + F9 : 현재 함수가 RET 명령으로 끝날 때까지 실행

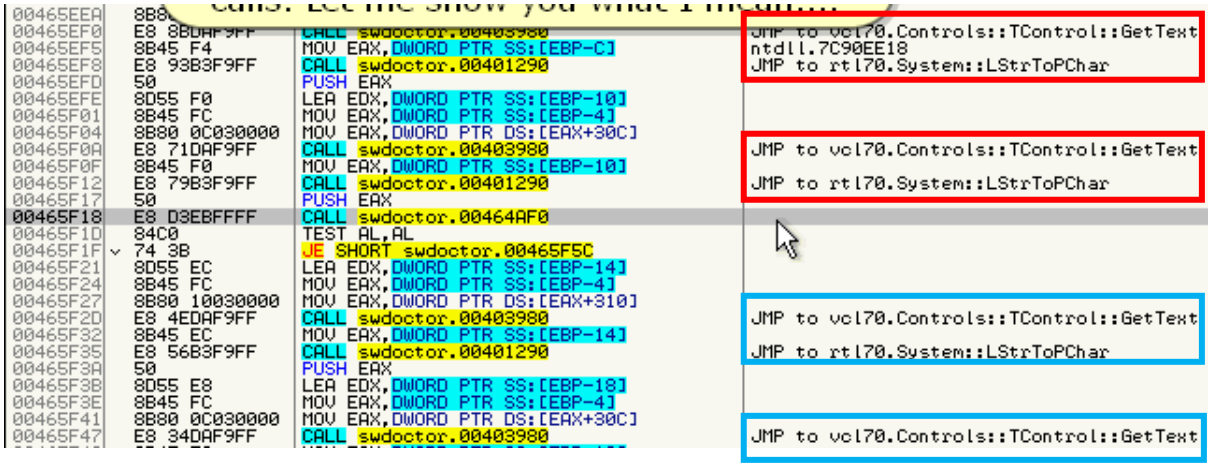
- F9 : 프로그램 전체 계속 실행 (다음 BP까지)



이렇게 입력한 값이 swdocotr.00464AF0에 저장된 걸 볼 수 있고 0x00465F18에서 해당 함수가 불려진 걸 볼 수 있다. 0x00465F18 위치로 들어가본다.



※ 리버싱(역공학)은 약간의 '시행착오(trial and error)'가 섞여 있다. 하지만 리버싱의 진짜 묘미는 '추측'을 가능한 한 줄이고 논리적인 관찰로 추측해야한다. 조금만 살펴보면 (지금처럼 압축 해제된 소프트웨어에서도 볼 수 있듯이), 이 부분이 등록 루틴(registration routine) 이라는 걸 꽤 높은 확신으로 알 수 있다.



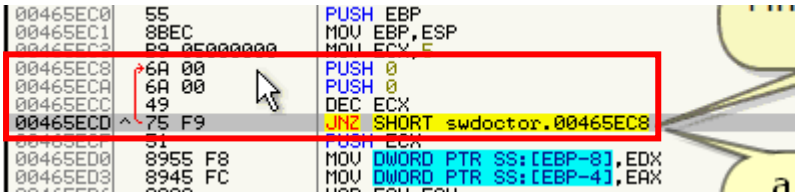
스크롤을 올리면 빨간색의 첫 번째 GetText에서는 시리얼을 읽는 부분, 두 번째 GetText는 이름을 읽는 부분이라는 걸 추정해볼 수 있고 그 후 값을 받아서 0x00464AF0으로 넘어가는 걸 볼 수 있고 그 후 JE 분기문이 실행되는 걸 볼 수 있다.

그리고 파란색의 첫 번째와 두 번째도 시리얼과 이름을 읽는 부분이라는 걸 추정할 수 있다. 하지만 JE가 실행된다면 이 부분은 실행되지 않을 것이다.

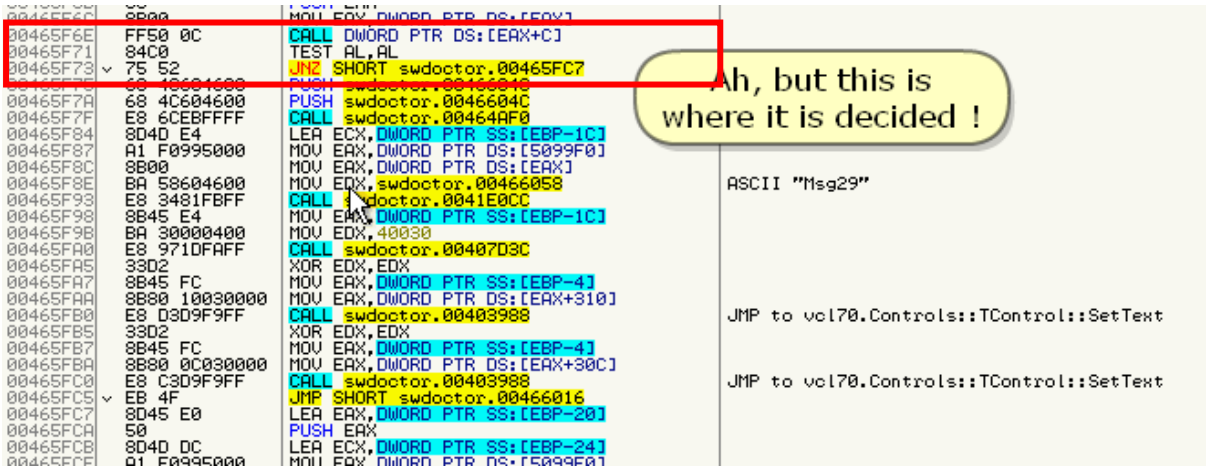
- GetText의 역할

GetText는 거의 항상 텍스트 속성을 읽어오는 메서드.

즉, 프로그램의 UI 요소(텍스트 박스, 레이블, 버튼 등)에 표시된 문자열을 가져오는 함수.



스크롤을 올리면 해당 부분을 볼 수 있는데 단지 스택을 준비하는(pushing 0) 동작을 하는 부분이다.



콜 스택에서 0x00465F18 기준으로 밑으로 내려가보면 또 하나의 분기문이 있는 걸 볼 수 있다. 해당 분기문이 점프를 하면 SetText가 실행 안되는 걸 볼 수 있다.

해당 부분은 실패 메시지를 보여주는 함수이므로 점프를 해야 성공 메시지를 띄울 수 있다는 것을 추정할 수 있다.

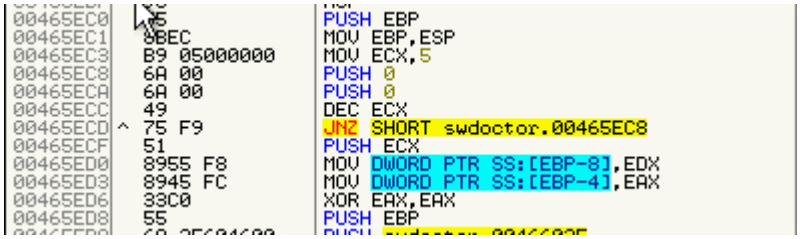
해당 분기문에서 메시지의 방향성을 선택하니까 그 위의 call 함수인 0x00465F6E는 시리얼과/이름을 비교해서 서로 맞으면 1, 다르면 0을 나오게 하는 걸 알 수 있다.

리턴 값이 1일 경우 분기되는 "465FC7" 주소를 등록 성공이라고 추정하였으니 실제로

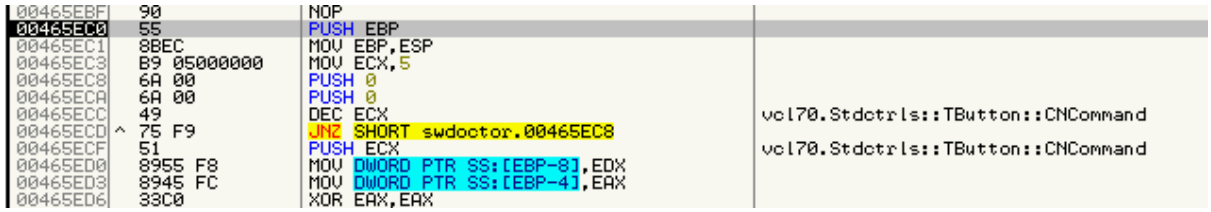
맞는지 안 맞는지 테스트 해봐야 한다.

- SetText의 역할

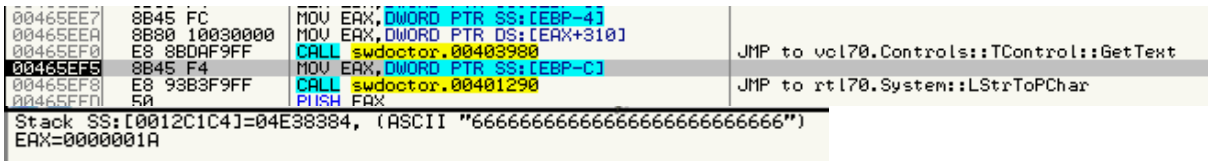
UI 컨트롤의 "Text" 속성 값을 바꾸는 함수.



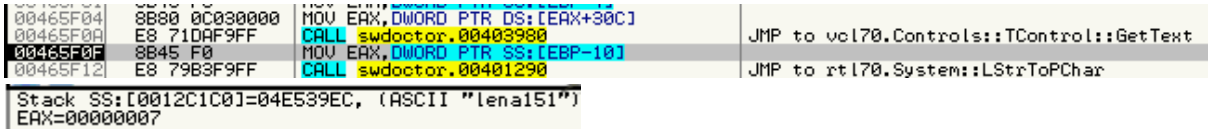
추정한 것들이 맞는지 확인하기 위해서 처음부터 실행해봐야한다. 0x00465EC0 부분에 HW BP를 건 후 실행해준다.



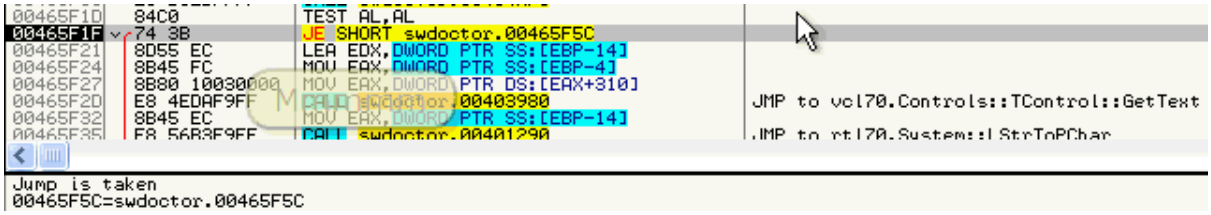
값들을 입력 후 'Register Now'를 눌러주면 BP 부분에서 멈추는 걸 볼 수 있다.



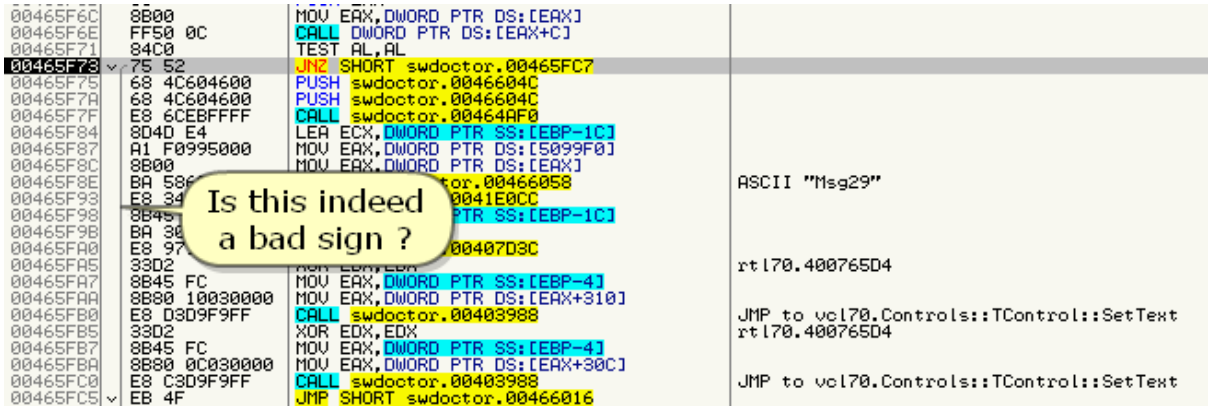
아까 빨간색의 첫 번째 부분을 보면 시리얼 값을 읽어드리는 걸 볼 수 있다.



아까 빨간색의 두 번째 부분을 보면 이름 값을 읽어드리는 걸 볼 수 있다.

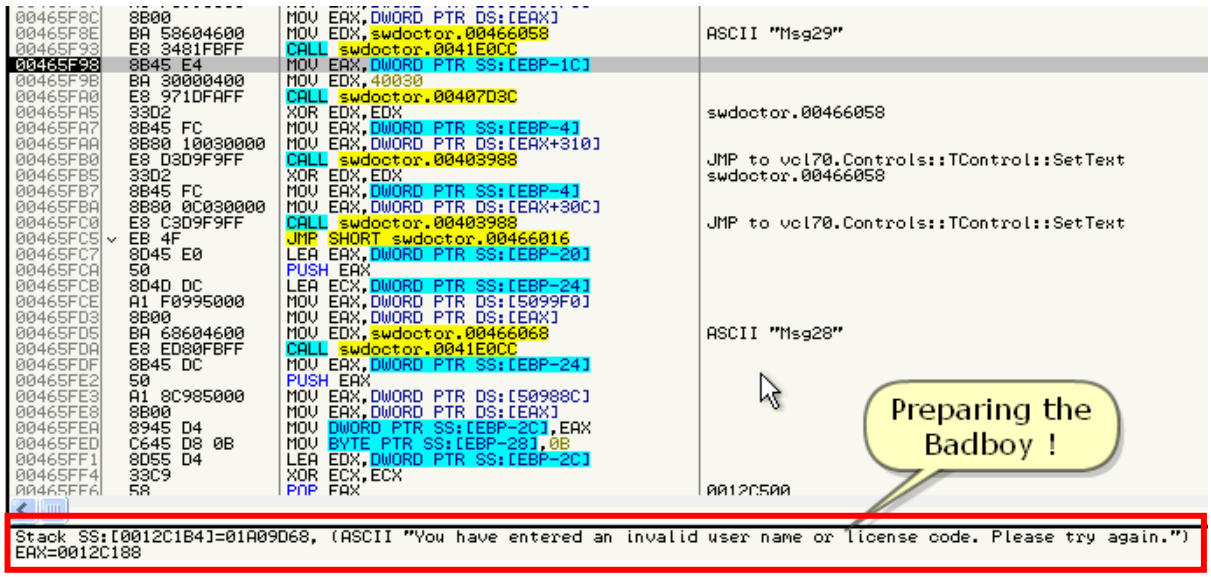


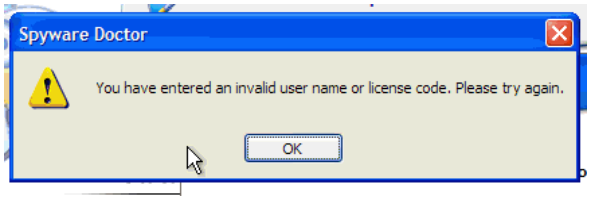
그 후 파란색 부분인데 이 부분은 점프하는 걸 볼 수 있다. 해당 부분도 GetText를 사용하는 걸 볼 수 있는데 이 부분은 값을 입력 안하면 발생되는 코드로 알 수 있다.



실행하다보면 SetText를 실행하는지 안하는지 판단하는 분기문에 도달하는 걸 볼 수 있다. 전에는 점프를 하면 성공 메시지, 점프를 하지 않으면 실패 메시지를 발생시킨다고 추정을 했는데 이번에 실행해보니까 점프가 실행안되는 걸 판단할 수 있다.

즉, 해당 부분은 추정하는게 맞았고 call함수가 판단하는 부분이라는 걸 알았다.





실행을 해보면 발생시키는 걸 볼 수 있다. 다시 올리디버그로 돌아가본다.

```
00465FA0   E8 971DFAFF     CALL swdoctor.00407D3C
00465FA5   33D2            XOR EDX,EDX
00465FA7   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00465FAA   8B80 10030000   MOV EAX,DWORD PTR DS:[EAX+310]
00465FB0   E8 D3D9F9FF     CALL swdoctor.00403988          JMP to vcl70.Controls::TControl::SetText
00465FB5   33D2            XOR EDX,EDX
00465FB7   8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00465FBA   8B80 0C030000   MOV EAX,DWORD PTR DS:[EAX+30C]
00465FC0   E8 C3D9F9FF     CALL swdoctor.00403988          JMP to vcl70.Controls::TControl::SetText
00465FC5   EB 4F           JMP SHORT swdoctor.00466016
```

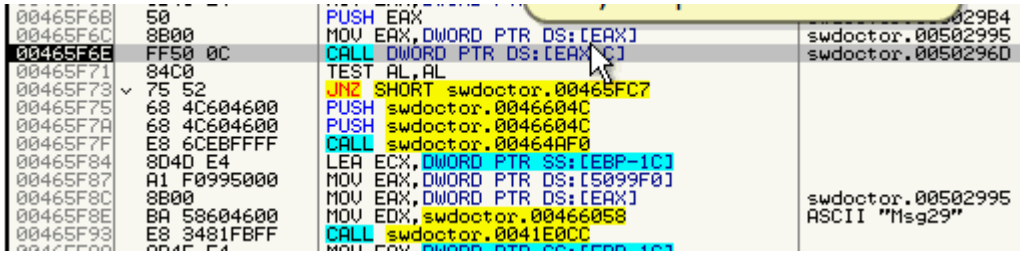
0x00465FA0에서 실패 메시지가 발생하는 걸 볼 수 있다. 아까 경고 창에서 'OK'버튼을 누르면 "SetText" 함수를 호출하여 입력 창을 비워서 사용자의 입력을 기다리는 걸 볼 수 있다.

```
00465F6B   50              PUSH EAX
00465F6C   8B00            MOV EAX,DWORD PTR DS:[EAX]
00465F6E   FF50 0C         CALL DWORD PTR DS:[EAX+C]
00465F71   84C0            TEST AL,AL
00465F73   75 52           JNZ SHORT swdoctor.00465FC7
00465F75   68 4C604600     PUSH swdoctor.0046604C
00465F7A   68 4C604600     PUSH swdoctor.0046604C
00465F7F   E8 6CEBFFFF     CALL swdoctor.00464AF0
00465F84   8D4D E4         LEA ECX,DWORD PTR SS:[EBP-1C]
00465F87   A1 F0995000     MOV EAX,DWORD PTR DS:[5099F0]
00465F8C   8B00            MOV EAX,DWORD PTR DS:[EAX]
00465F8E   BA 58604600     MOV EDX,swdoctor.00466058
00465F93   E8 3481FBFF     CALL swdoctor.0041E0CC
00465F98   8B45 E4         MOV EAX,DWORD PTR SS:[EBP-1C]
00465F9B   BA 30000400     MOV EDX,40030
```

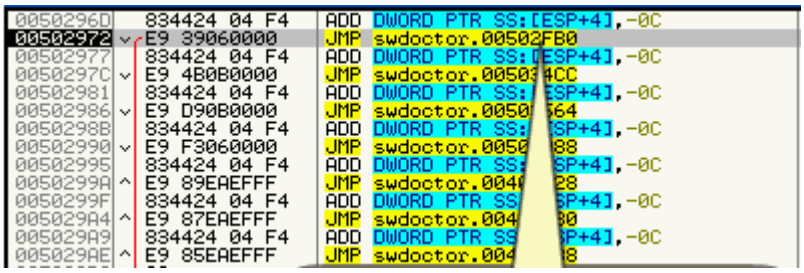
JNZ -> JMP로 바꾸면 되겠지만 이번에는 조건문(JNZ)에 사용되는 AL 레지스터 값을 세팅하는 0x00465F6E를 패치하는 **"intermediate" 패치**를 보여줄 것이다.

왜 이렇게 복잡한 패치를 진행하는지 궁금하다면, 만약 "JNZ" 조건문을 패치해 버리면 그냥 메인 루틴 부분만 패치되는 것이다. 따라서 다른 루틴에서 해당 함수를 호출해 버리면 무용 지물이 된다. 그러나 "[EAX+C]" 함수 내부에서 AL 레지스터 값을 패치해 버리면 다른 루틴에서 해당 함수를 호출해도 자동적으로 패치가 된다.

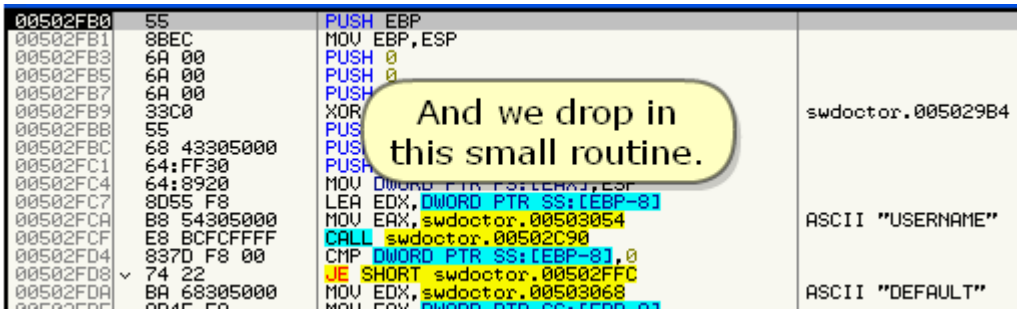
다시 값들을 입력 후 'Regiser Now'를 눌러준 후 0x00465F6E위치까지 와준다.



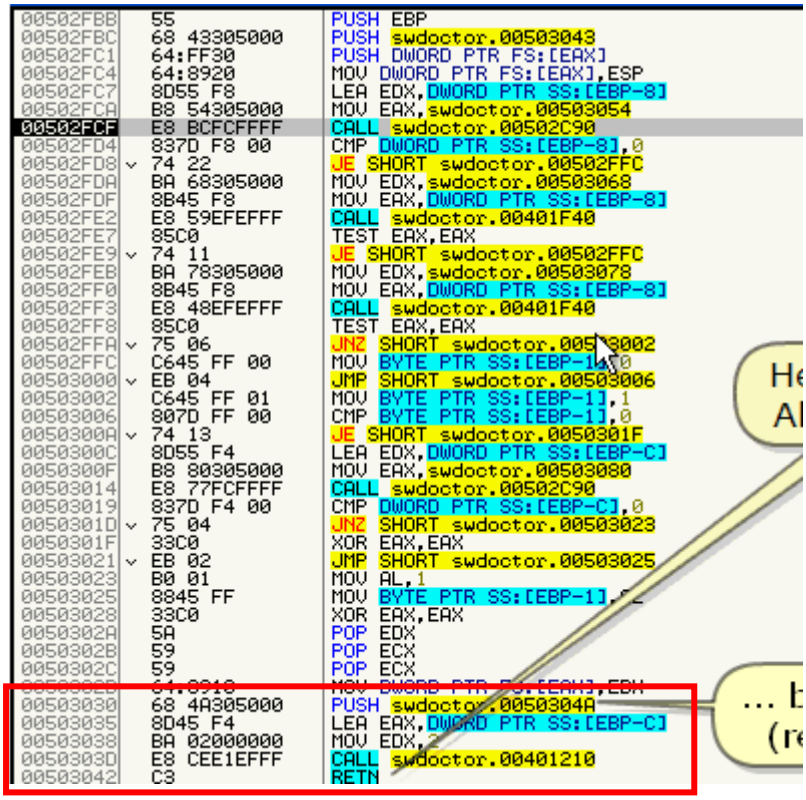
해당 부분을 'F7'을 눌러 들어가준다.



0x0050296D에 도달하고 'F8'을 이용해 하나 내려가 주면 JMP분기문이 실행되는 걸 볼 수 있을 것이다.



0x0050296D에 HW BP를 설치하고 프로그램을 재실행해보니 해당 부분이 총 23번 호출되었다. 23번 호출되었다는 의미는 등록 상태를 검사하는 코드가 23번 사용되었다는 것이다.



'F8'을 이용해서 내려가다 보면 RETN이 있는 걸 볼 수 있다.

해당 빨간 박스를 보면 PUSH를 사용하고 RETN이 나오는 걸 보면 JMP를 의미하는 걸 알 수 있다. (PUSH + RETN == JMP)

※ 원래 PUSH -> CALL -> RETN != JMP 이다.

왜냐하면 CALL함수가 정상적으로 리턴하지 않으면 RETN에 도달하지 않기 때문이다.

1. PUSH addr + RET가 JMP addr인 이유

- PUSH <addr>: 스택에 <addr> 값을 넣음 (스택 포인터 SP가 감소하고 그 위치에 <addr> 저장).

- RET: 스택에서 값 하나를 팝해서(POP) 그 값을 리턴 주소로 사용해 점프함.
따라서 바로 이어서 PUSH target 다음에 RET이 실행되면, RET는 스택에서 target을 꺼내서 그 주소로 점프한다. 결과적으로 push target; ret는 jmp target과 동작이 동일하게 보인다.

2. 중간에 CALL이 있는 경우

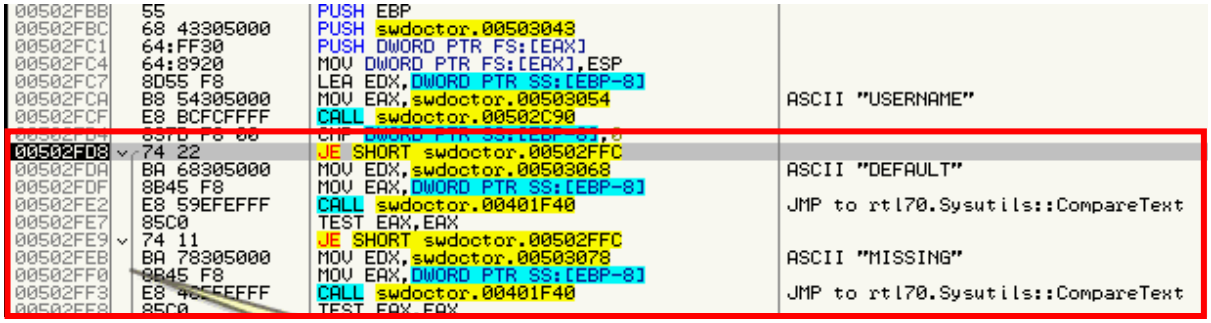
- CALL 명령은 현재 위치(리턴 주소)를 스택에 푸시하고 호출한 함수로 점프한다.

그래서 만약 실행 흐름상 PUSH target 이후 사이에서 다른 PUSH/CALL/POP 등이 실행된다면 스택의 최상단(RET가 pop할 값)이 바뀐다.
이러면 RET가 target이 아니라 그 사이에 쌓인 다른 값(예: CALL의 리턴 주소)을 팝해서 그쪽으로 점프하게 되므로 의도한 동작이 깨진다.

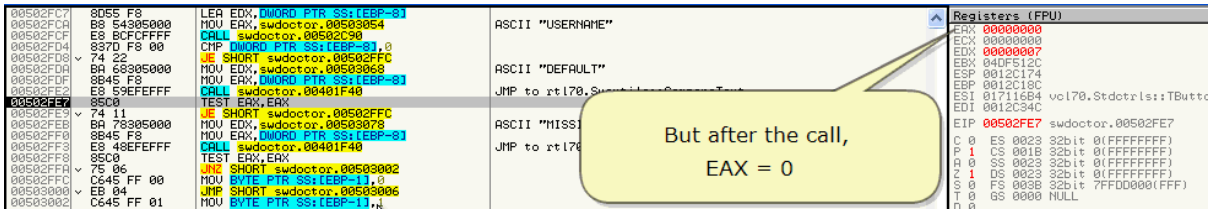
-> PUSH target — (CALL 어떤 함수) — RET을 **트램폴린(trampoline)** 형태라고 불린다.

- 트램폴린(trampoline)

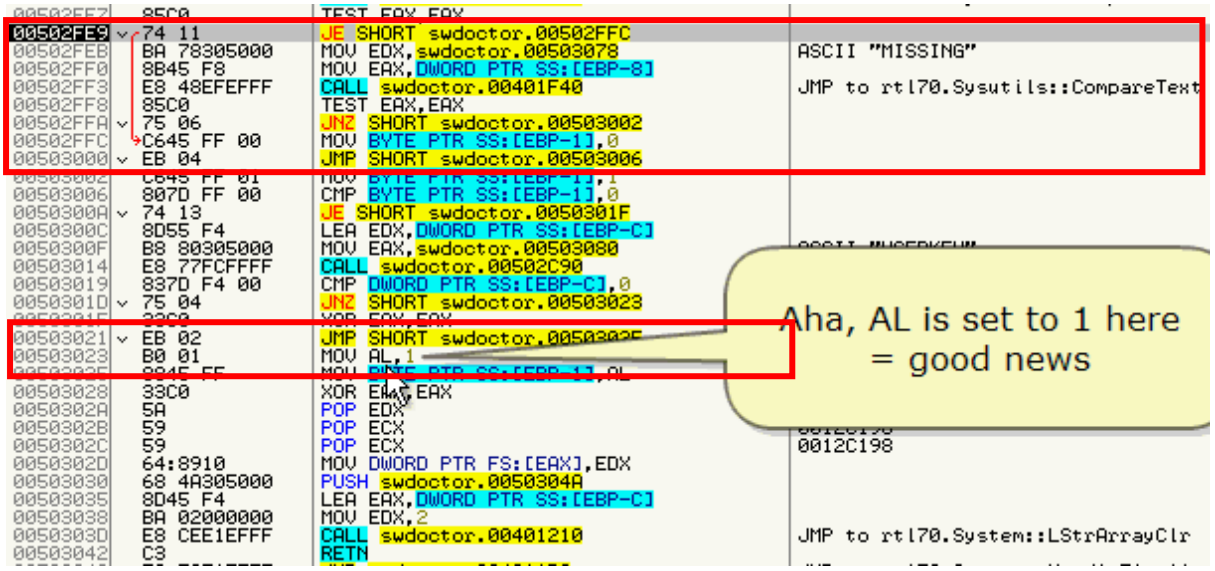
트램폴린(trampoline) 은 어셈블리나 시스템 프로그래밍, 함수 호출 구조에서
"제어 흐름을 다른 코드로 '튀겨' 보내는 중간 다리 역할"을 하는 짧은 점프용 코드 조각이다.



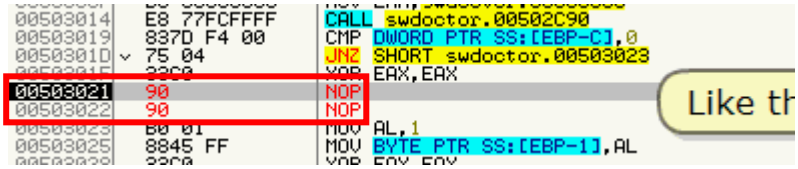
JE분기문에서 점프 안하는 걸 볼 수 있다. 근데 해당 부분은 CompareText가 있는 부분이다. 그럼 이 부분에서 결정되는 걸 볼 수 있다.



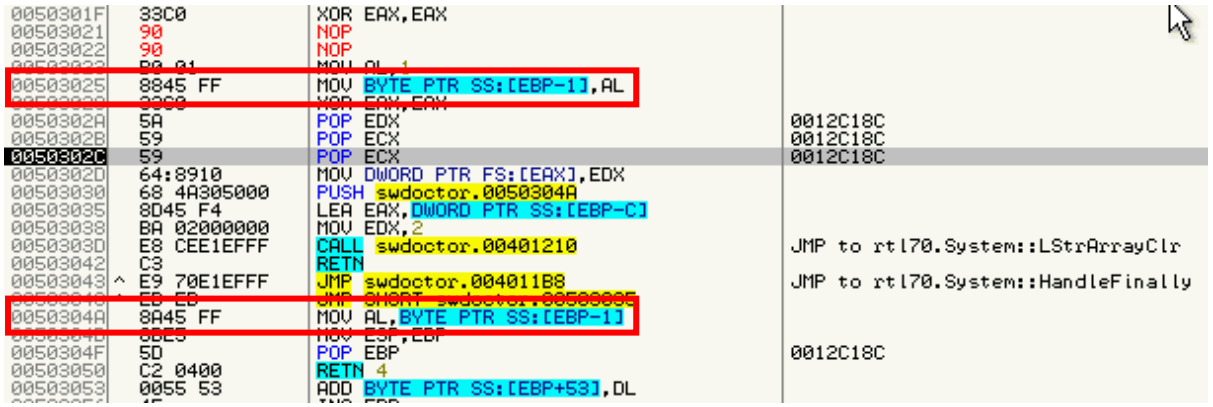
EAX == 0이 나오는 걸 볼 수 있다.



첫 번째 빨간 박스를 보면 두 번째 CompareText가 실행안되는 걸 볼 수 있고 밑에 코드를 좀 더 보면 MOV AL, 1을 볼 수 있다. 그럼 앞에는 코드들은 다 상관없는 걸 볼 수 있다.

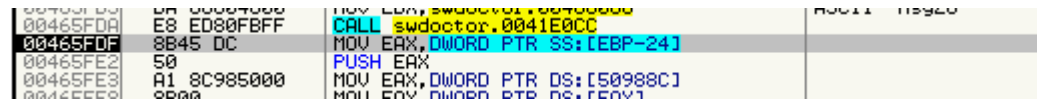


그럼 여기서 JMP분기문이 실행이 안되어야하므로 NOP으로 바꿔주면 된다.



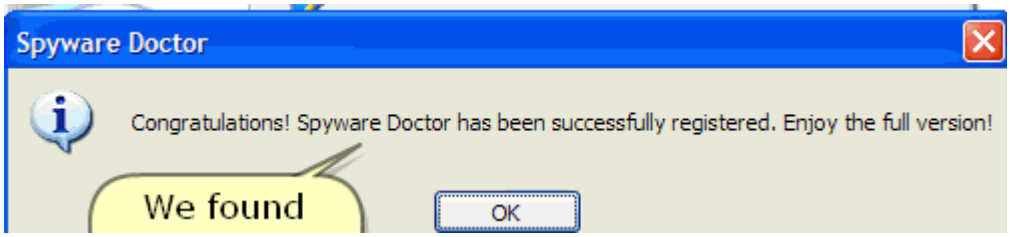
AL이 [EBP-1]에 저장되는 걸 볼 수 있고 RETN이 끝나고 다시 AL이 [EBP-1]에 저장되어 있는 값을 가져 오는 걸 볼 수 있다.

RETN 4까지 끝나야 내부 함수에서 나오는 걸 볼 수 있다.



```
Stack SS:[0012C1AC]=01A0A778, (ASCII "Congratulations! %s has been successfully registered. Enjoy the full version!")
EAX=0012C184
```

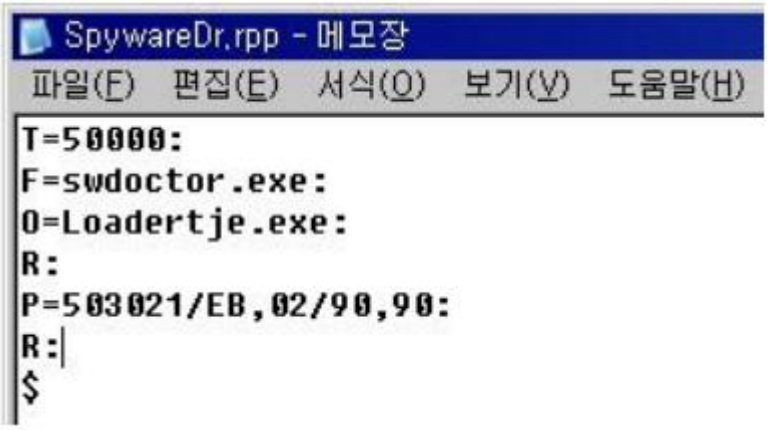
'F8'을 이용해서 실행해보면 0x00465FDF에서 성공 메시지가 나오는 걸 볼 수 있다.



해당 창이 나오고 'F9'를 눌러서 실행해준다.

"Buy Now"가 없어진 창이 뜨는 걸 볼 수 있다.

이제 해야 할 일은 "0x00503021" 주소 명령어를 패치(opcode 패치, "EB 02 => 90 90" == "JMP" => "NOP") 하는 로더를 만드는 것이다. 이전 레벨과 마찬가지로 "R!SC Process Pathcer"를 이용할 것이다.



위와 같이 스크립트 파일을 만들고 "R!SC Process Pathcer"를 이용하여 로더를 제작하면 이제 프로그램이 실행 될 때마다 임의의 시리얼 값으로 등록이 될 것이다.