===============================

“악성코드 분석” 최종 보고서

===============================

x32dbg를 이용한 어셈블리 디버깅 이론 및 실습

AI를 활용한 보안전문가 양성 과정

최형회

2025년 7월 21일

조효제 강사님

===============================

서론

이번 과제는 CrackMe 예제를 활용하여 어셈블리어 분석과 디버거 사용법을 학습하고, 이를 통해 악성코드 분석 역량을 기르는 것을 목표로 하였습니다. 실습은 VMWare에 제공된 윈도우 이미지를 활용하여 진행되었으며, CPU의 기본 구성과 PE(Portable Executable) 파일의 실행 과정, 그리고 어셈블리어의 구조와 Opcode(프로그래밍 언어에서 함수와 유사한 역할을 수행하는 명령어)에 대해 학습하였습니다. 또한, 간단한 실습용 파일을 x32dbg 디버거에 업로드하여 어셈블리어 코드의 흐름을 직접 추적하고 분석함으로써 디버깅 도구의 사용법을 익히는 실습도 함께 이루어졌습니다. 이를 통해 이론뿐만 아니라 실습을 통해 디버깅 및 분석 능력의 기반을 체계적으로 향상시킬 수 있었습니다.

본론

“.exe” 실행 파일을 x32dbg에 업로드하여 분석하는 과정은 프로그램의 내부 동작을 어셈블리 수준에서 이해하고, 그 흐름을 추적하기 위해 사용됩니다. 먼저 x32dbg를 실행한 후 분석 대상인 .exe 파일을 열면, 프로그램은 자동으로 해당 실행 파일의 엔트리 포인트(실행 시작 지점)부터 어셈블리 코드로 디스어셈블하여 보여줍니다. 이를 통해 사용자는 프로그램이 메모리에 어떻게 로드되고 어떤 명령어들이 순차적으로 실행되는지를 직접 확인할 수 있습니다.

분석 과정에서는 레지스터의 변화, 스택의 상태, 조건 분기, 함수 호출 등을 관찰하며 프로그램의 논리 흐름을 추적할 수 있습니다. 또한 브레이크포인트를 설정하여 특정 지점에서 실행을 일시 중지하고, 해당 시점의 메모리 상태나 변수 값을 확인할 수도 있습니다. 이러한 방식으로 프로그램의 인증 과정이나 암호화 루틴, 숨겨진 기능 등을 파악할 수 있으며, 특히 CrackMe와 같은 연습용 파일을 통해 리버스 엔지니어링 기초를 익히는 데 매우 효과적입니다. x32dbg는 직관적인 인터페이스와 강력한 분석 도구를 제공하기 때문에 악성코드 분석, 취약점 연구 등 보안 관련 분야에서도 널리 활용되고 있습니다.

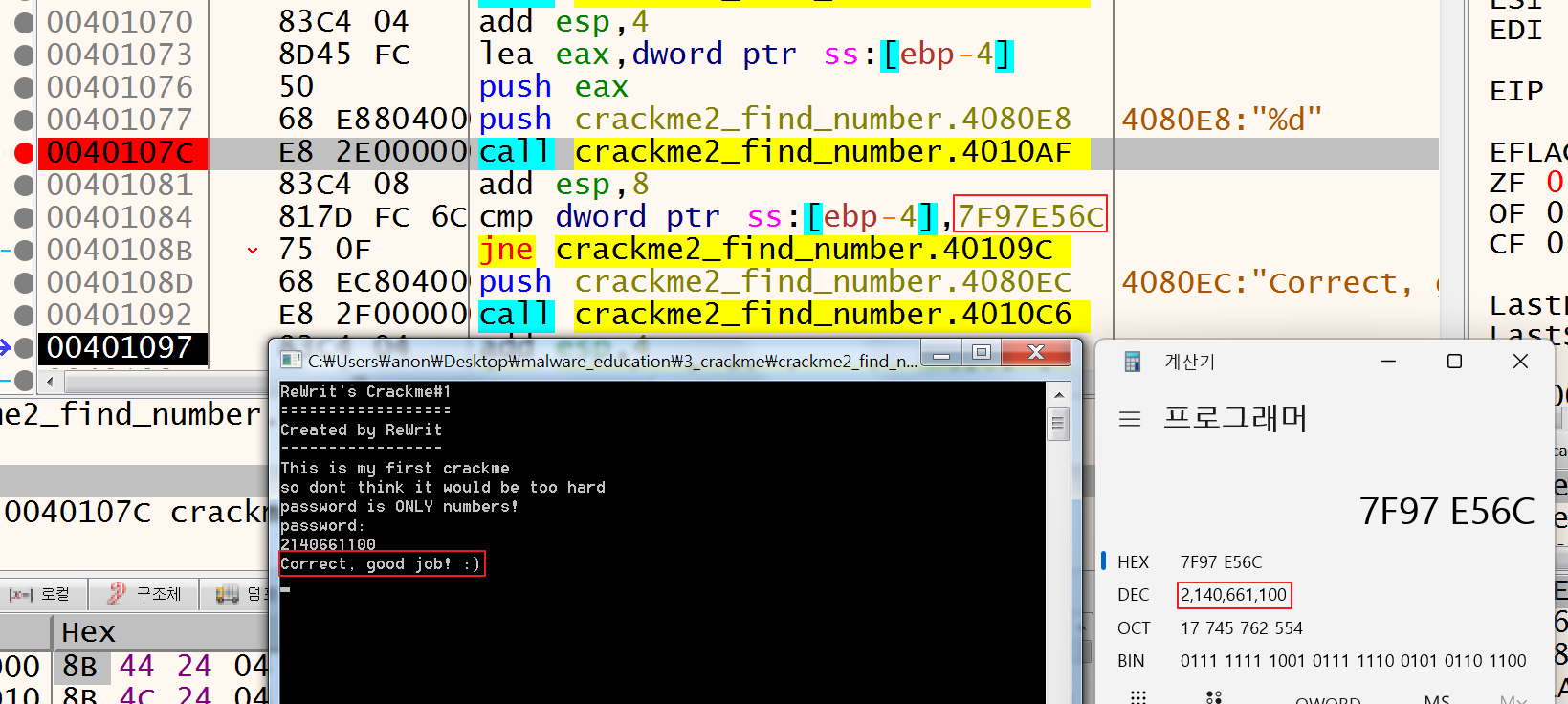
어셈블리어는 Opcode와 Operand가 결합된 명령 코드와 주소값, 그리고 변수에 저장된 값들로 구성되어 있습니다. 이 중 Opcode는 어셈블리어에서 핵심적인 역할을 수행하는 명령어로, 고급 프로그래밍 언어에서 함수 이름에 해당하는 기능을 합니다. Opcode는 CPU가 수행할 작업의 종류를 지정하며, Operand는 그 작업의 대상이 되는 데이터나 주소를 의미합니다. 어셈블리어에서 사용되는 Opcode는 데이터 이동, 산술 연산, 비트 연산, 논리 연산, 제어 이동 등 다섯 가지로 분류할 수 있습니다.

* 데이터 이동(Move): 메모리나 레지스터 간에 데이터를 복사하는 명령이다. 예를 들어 MOV 명령은 특정 레지스터에 값을 저장하거나 다른 위치로 옮기는 데 사용된다.
* 산술 연산(Arithmetic): 덧셈, 뺄셈, 곱셈, 나눗셈 등 수학적 연산을 수행하는 명령이다. 대표적인 명령어로는 ADD, SUB, MUL, DIV 등이 있다.
* 비트 연산(Bitwise): 데이터의 이진수 표현을 기준으로 각 비트를 대상으로 연산을 수행한다. AND, OR, XOR, NOT, SHL, SHR 등의 명령이 여기에 해당한다.
* 논리 연산(Logical): 조건 판단이나 플래그 설정을 위해 사용되며, 비교 결과에 따라 분기할 수 있는 조건들을 구성한다. 대표적으로 CMP 명령이 있으며, 이는 두 값을 비교한 뒤 플래그에 결과를 저장한다.
* 제어 이동(Control Transfer): 프로그램의 실행 흐름을 변경하는 명령어로, 조건문이나 반복문 구현에 사용된다. JMP, JE, JNE, CALL, RET 등의 명령이 있으며, 이를 통해 코드 실행 위치를 변경하거나 함수 호출 및 복귀를 제어할 수 있다.
* 이러한 Opcode의 조합을 통해 어셈블리어는 매우 낮은 수준에서 시스템 자원을 직접 제어할 수 있으며, 이를 분석함으로써 프로그램의 내부 동작을 정밀하게 이해할 수 있다.

사전 실습1

이번 실습은 “crackme2\_find\_number.exe”와 “crackme3\_find\_input\_key.exe” 두 가지 CrackMe 예제 프로그램을 대상으로 x32dbg 디버거를 활용하여 분석하고, 입력값을 추론하여 정답을 도출하는 과정을 통해 어셈블리어의 동작 원리와 디버깅 기술을 학습하는 데 목적이 있었습니다.

첫 번째 실습에서는 crackme2\_find\_number.exe 파일을 x32dbg에 업로드하여 문제를 해결하였습니다. 문자열 참조 기능을 활용하여 “%d” 형식을 찾아 4010AF 주소가 scanf 함수라는 점을 파악하였고, 이후 코드의 흐름을 따라가며 입력된 값이 특정 레지스터에 저장된 뒤 cmp 명령을 통해 0x7E97E56C 값과 비교된다는 사실을 확인하였습니다. 이 값을 10진수로 변환하여 입력한 결과, “Correct, good job! 😊”라는 출력 메시지를 확인함으로써 문제를 해결할 수 있었습니다. 이 과정에서는 불필요한 분석을 줄이기 위해 문자열을 통한 접근 방식을 선택하였으며, 디버깅 중 push, call, add, cmp, jne 등 다양한 Opcode의 동작과 역할도 함께 학습할 수 있었습니다. 또한 디버거 내 각 창의 역할과 단축키 활용 방법에 대해서도 익힐 수 있었습니다.



본 실습

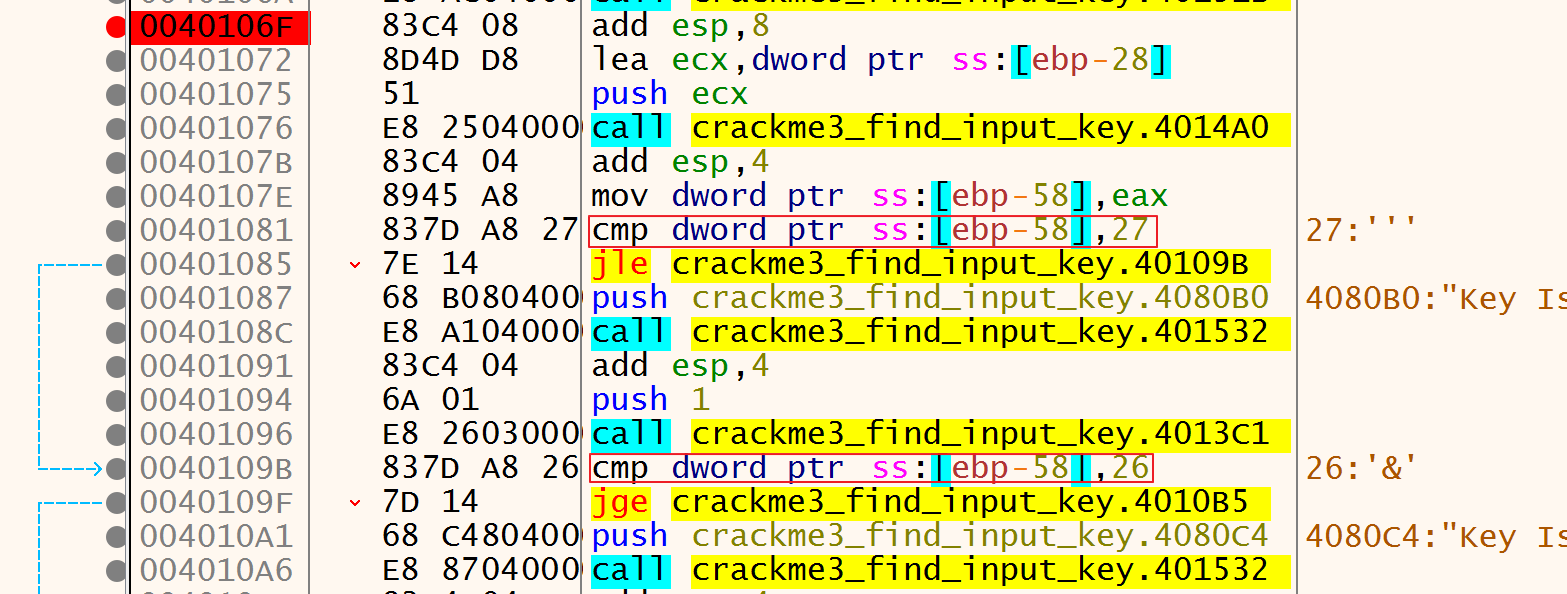
본 실습에서는 crackme3\_find\_input\_key.exe 파일을 대상으로 보다 복잡한 구조의 코드 분석을 수행하였습니다. 전체적인 코드 흐름을 파악한 결과, 올바른 입력값을 판별하기 위해 다음과 같은 네 가지 주요 과정을 거친다는 사실을 확인하였습니다.

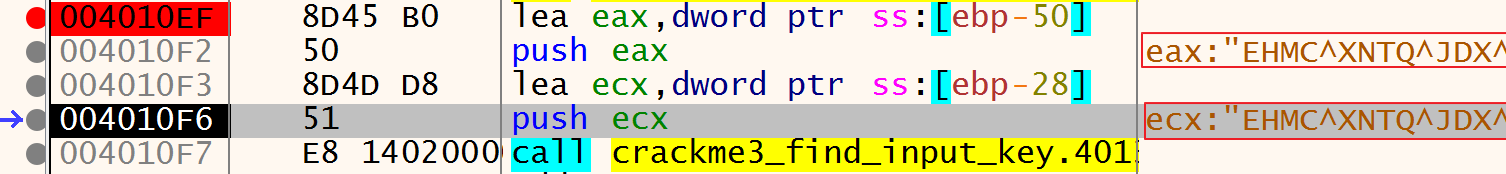
* 입력된 문자열의 글자 수 확인
* 반복문을 통해 입력 문자열을 암호화
* 암호화된 문자열을 기준 문자열과 비교하고 결과를 특정 레지스터에 저장하는 함수 호출
* 레지스터에 저장된 비교 결과를 기반으로 최종 출력 여부를 결정

이러한 구조 속에서 실습자는 한 줄씩 코드를 실행시키며 레지스터와 스택에 저장된 값을 확인하였고, 이를 바탕으로 입력값을 하나씩 추론해나가는 과정을 거쳤습니다.

이처럼 복잡한 흐름을 효율적으로 분석하기 위해서는 단순히 코드를 따라가는 것이 아니라, 프로그램 전체 구조를 먼저 파악하고 어떤 부분이 핵심적인 로직인지 판단하는 능력이 중요하다는 점을 체감할 수 있었습니다.

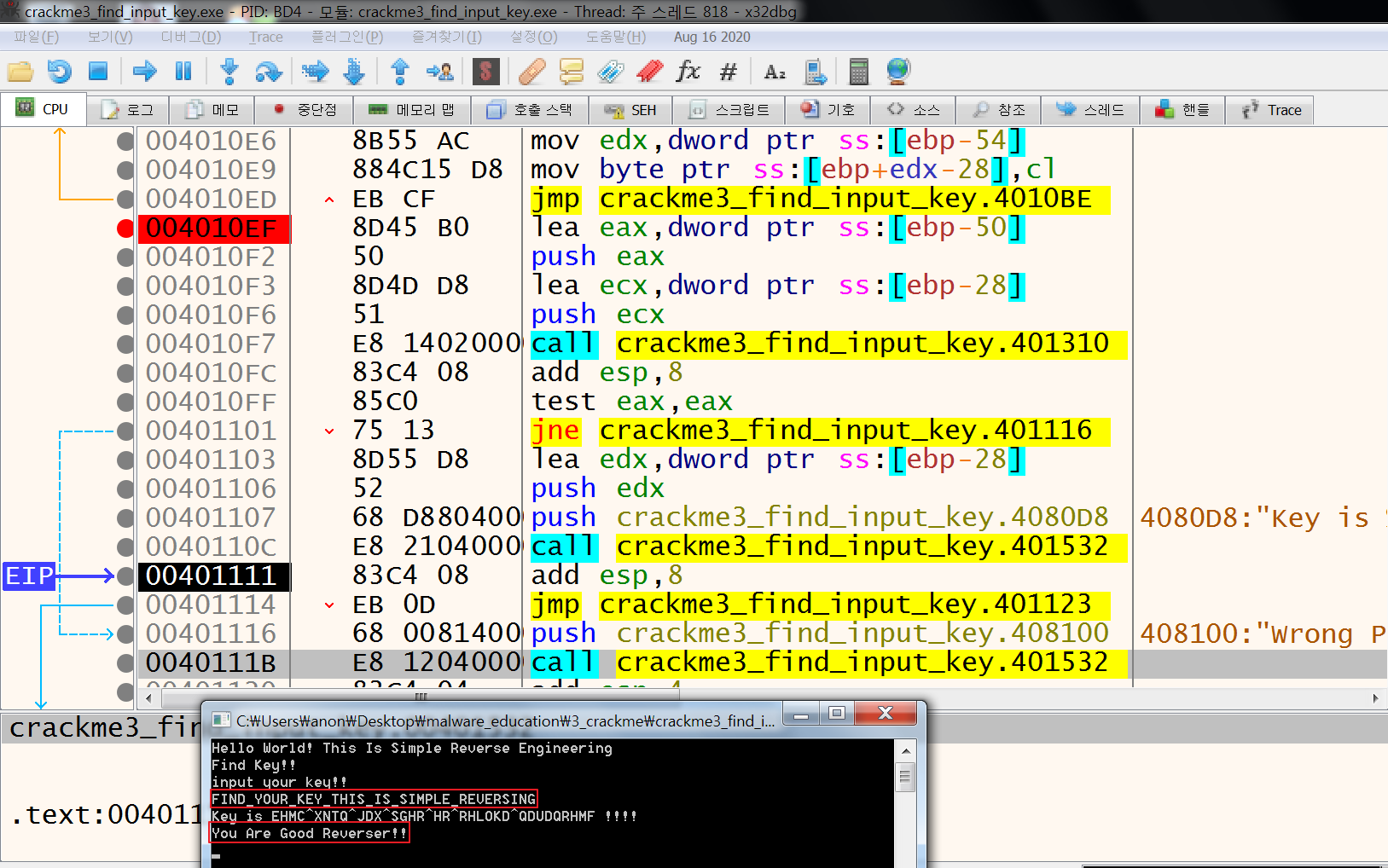
첫 번째 과정에서는 비교 조건에 등장하는 숫자들이 16진수로 표현되어 있다는 점이 초보자에게 혼란을 줄 수 있었습니다. 예를 들어 0x26, 0x27은 각각 38, 39 글자를 의미하지만, 처음에는 이를 단순히 26자, 27자로 오해하여 문제를 해결하는 데 시간이 소요되었습니다. 하지만 실제로는 0x26과 0x27이었고, 이에 해당하는 글자 수를 찾아내어 해당 조건을 넘어갈 수 있었습니다.



두 번째 과정에서는 입력된 문자열이 반복문을 통해 암호화되는 과정을 분석하였습니다. 반복문을 거친 후 새로운 문자열이 생성되는 것을 확인할 수 있었고, 두 문자열이 일치할 때 정답이라는 메시지가 출력된다는 사실도 파악하였습니다. 반복문 내의 연산을 분석한 결과, -1씩 문자값을 이동시키는 시저 암호화 방식이 적용되어 있었으며, 이를 바탕으로 생성형 AI를 활용하여 비교 문자열을 +1 시프트 방식으로 복호화하였습니다.



그 결과 “FIND\_XOUR\_KEY\_THIS\_IS\_SIMPLECREVERSING”이라는 문자열을 얻을 수 있었습니다. 그러나 이 문자열을 그대로 입력하였을 때는 오답으로 처리되었고, 코드 내부에서 추가적인 문자열 변형이 이루어진다고 판단하였습니다. 이에 따라 의미상 어색한 부분 두 군데를 수정하여 최종적으로 올바른 입력 문자열을 도출할 수 있었습니다.



이번 실습을 통해 디버깅 도구인 x32dbg를 활용하여 어셈블리어 수준에서 프로그램의 동작을 추적하고 분석하는 방법을 익힐 수 있었습니다.

결론

이번 실습을 통해 단순히 입력 키를 찾아내는 것을 넘어서, 프로그램의 흐름을 파악하고 불필요한 분석을 줄이며 핵심적인 지점을 효율적으로 찾아가는 디버깅 전략의 중요성을 배울 수 있었습니다. 사전 실습인 CrackMe2에서도 확인할 수 있듯이, 입력값 비교 이전에 호출되는 특정 함수는 겉보기에는 무해해 보일 수 있지만, 악성코드 분석 관점에서는 입력값을 활용할 가능성이 있기 때문에 주의 깊게 살펴봐야 할 대상입니다. 이러한 점에서, 전체 코드를 일일이 분석하기보다는 디버깅 도구에 대한 이해를 바탕으로 핵심 흐름을 빠르게 파악하고, 의심되는 지점을 선별해 추적해나가는 ‘눈 디버깅’ 능력이 중요하다고 느꼈습니다.

이번 실습은 지난주에 진행된 악성코드 이론 강의, 그리고 오늘 오전의 디버깅 실습과 유기적으로 연결되어 있었고, 그 과정에서 어셈블리어 분석과 디버거 활용에 대한 기초적인 지식과 감각을 익힐 수 있는 좋은 기회였습니다. 어쩌면 다소 어렵고 지루할 수 있었던 내용을 강사님께서 코드에 대한 애정(?)을 담아 흥미롭게 풀어내 주셔서 더욱 즐겁게 학습할 수 있었습니다. 내일은 또 다른 문제를 통해 한 단계 더 나아간 분석을 경험하게 될 것으로 기대되며, 동시에 긴장도 됩니다. 감사합니다.