Windows 64bit조작체계에서 C언어를 리용한 아쎔블리코드후킹실현의 한가지 방법

리선남, 전철용

위대한 령도자 **김정일**동지께서는 다음과 같이 교시하시였다.

《프로그람을 개발하는데서 기본은 우리 식의 프로그람을 개발하는것입니다. 우리는 우리 식의 프로그람을 개발하는 방향으로 나가야 합니다.》(《**김정일**선집》 증보판 제21권 42페지)

오늘날 콤퓨터는 인민경제의 주체화, 현대화, 정보화, 과학화를 실현하는데서 없어서는 안될 수단으로 되고있으며 나날이 응용범위가 확대되고있다.

후킹(Hooking)[1, 3, 4]은 어떤 공정흐름의 도중에 추가적인 조작을 진행하기 위하여 공정에서 잠간 리탈하여 필요한 조작을 진행한 후 다시 원래의 공정을 계속 수행하는 과정이다.

류사한 개념인 려과(Filtering)[2]에서는 공정흐름과정에서 필요한 결과만을 통과시키며 공정을 통과하는 모든 입력들을 다 검사하지만 입력값에 따라 원래공정으로 귀환할수도 있고 그렇게 하지 않을수도 있다.

그러나 후킹은 공정의 도중에 추가적인 조작을 수행하고 반드시 원래의 공정으로 귀환한다.

선행연구[1]에서 제안한 IAT후킹방법은 IAT(Import Address Table)를 리용하여 후킹을 진행하는 방법인데 Windows실행화일의 PE구조체에서 Windows API들의 주소목록을 보관하는 IAT에 Windows API의 주소대신에 후킹처리부분의 주소를 써넣는 방법이다. 이 방법에서는 실행화일에서 Windows API를 호출하려고 IAT로부터 주소를 얻을 때 후킹처리부분의 주소가 대신 얻어지며 실행화일은 후킹처리부분을 호출한다.

이 방법의 우점은 간단하고 조작체계판본에 상관없이 안전하게 동작할수 있는 반면에 다음과 같은 여러가지 문제점들이 있다.

우선 IAT라는 한가지 자원에 의거하는것으로 하여 이미 후킹한 Windows API에 다시 후킹을 걸면 충돌이 일어나게 된다.

다음 Windows API에만 후킹을 걸수 있으며 광범한 실행령역에 후킹을 할수 없다는 부족점들이 있다.

이러한 부족점들을 극복하기 위하여 론문에서는 Windows 64bit에서 광범한 실행코드령역에 대한 아쎔블리(Assembly)코드후킹을 실현하는 방법을 제안하였다.

Windows 32bit에서 아쎔블리코드후킹은 실행코드령역에 0xE8이나 0xE9로 시작되는 call/jmp명령을 덧쓰기하고 후킹처리부분에서는 등록기값을 탄창에 넣었다가 후킹처리를 다 진행한 다음 탄창에서 등록기값들을 읽어서 회복한 다음 원래의 공정을 수행하는 방법으로 후킹을 실현한다.

Windows 64bit에서 이러한 후킹을 실현하자면 다음의 두가지 문제점들이 제기된다.

첫째로, 후킹하려는 모듈과 후킹후처리부분이 들어있는 모듈들이 다르고 두 모듈이 위치한 주소들의 편위가 4B한계를 넘는 경우 선행연구방법을 리용할수 없다.

Windows 64bit에서는 CPU가 처리하는 모든 등록기들의 크기가 64bit(8B)이며 모든 주소와 지적자들도 8B로 표현되고 또한 일반적으로 후킹처리부분은 후킹하려는 주소와 서로 다른 모듈에 존재할수 있으므로 두 모듈사이 편위차값이 4B한계를 넘어나는 경우도 있을수 있다.

그러므로 0xE8이나 0xE9로 시작되는 jmp나 call명령들과 같이 4B편위값에 의하여 목적주소가 결정되는 명령들은 다른 모듈로 이행하는데는 쓸수 없다.

둘째로, 64bit에서는 inline아쎔블러명령을 쓸수 없으며 declspec(naked)와 같은 예약어를 지원하지 않는것으로 하여 등록기값들을 저장하는 공정이 간단하게 진행될수 없다.

론문에서는 Windows 64bit에서 이러한 문제점들을 극복하기 위하여 아쎔블리코드후킹을 다음과 같이 실현하였다.

① 후킹하려는 주소로부터 편위가 4B이하로 표현되는 령역에 기억구역을 할당한다.

② 등록기값들과 기발변수값을 보관하는 아쎔블러명령에 해당한 16진수자료를 기억구역에 써넣는다.

③ 기억구역으로 이행하도록 한다.

이 방법은 후킹주소로부터 최대한 가까운 기억구역에 후킹코드를 자료로 보관하는 방법이라고 말할수 있다.

후킹실현을 실례를 들어 고찰하자.

처음으로 후킹하려는 주소와 최대한 가까운 주소에 후킹을 위한 아쎔블러명령들을 보관할 기억구역을 할당한다.

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_   LPVOIDlpAddress,
  _In_       SIZE_T dwSize,
  _In_       DWORD flAllocationType,
  _In_       DWORD flProtect
);
```

우의 함수를 리용하여 후킹하려는 주소부터 시작하여 성공할 때까지 페지단위만큼씩 증가하면서 기억구역을 할당한다.

```
Int i;
    void * page;
    MEMORY_BASIC_INFORMATIONmbi;
    SecureZeroMemory(&mbi, sizeof(mbi));
    /* 목적주소의 모듈시작주소를 얻는다. */
    VirtualQuery((LPCVOID)targetAddress, &mbi, sizeof(mbi));
    /* 모듈의 시작과 끝으로부터 증가하면서 기억구역할당이 성공할 때까지 진행한
    다. */
    for (int i = 1; I < MAXINT32; i *= 2) {
        page = VirtualAlloc((LPVOID)((size_t)mbi.AllocationBase −
                (i * OS_PAGE_SIZE)), OS_PAGE_SIZE, MEM_COMMIT |
                        MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

```
        if (NULL != page) {
            break;              }
    page = VirtualAlloc((LPVOID)((size_t)mbi.AllocationBase +
            mbi.RegionSize + (i * OS_PAGE_SIZE)), OS_PAGE_SIZE,
            MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (NULL != page) {
            break;      }
    }
```

다음으로 할당된 기억구역에 등록기값과 기발변수들을 저장하기 위한 아쎔블러명령들의 16진수코드배렬을 저장한다.(표 1)

표 1. 아쎔블러명령에 해당한 16진수자료

주 소	16진수자료
0x000007F706B08600	50 51 52 53 54 55 56 57 41 50 41 51 41 52 41 53
0x000007F706B08610	41 54 41 55 41 56 41 57 9C 48 B8 10 32 54 76 87
0x000007F706B08620	BA DC Fe 50 48 B8 F0 DE BC 9A 78 56 34 12 50 C3

표 2에 표 1에 해당하는 아쎔블러명령을 보여주었다.

표 2. 표 1에 해당하는 아쎔블러명령

주 소	아쎔블러명령
0x000007F706B08600	push rax
0x000007F706B08601	push rcx
0x000007F706B08602	push rdx
0x000007F706B08603	push rbx
0x000007F706B08604	push rsp
0x000007F706B08605	push rbp
0x000007F706B08606	push rsi
0x000007F706B08607	push rdi
0x000007F706B08608	push r8
0x000007F706B0860A	push r9
0x000007F706B0860C	push r10
0x000007F706B0860E	push r11
0x000007F706B08610	push r12
0x000007F706B08612	push r13
0x000007F706B08614	push r14
0x000007F706B08616	push r15
0x000007F706B08618	pushfq
0x000007F706B08619	mov rax, 0FEDCBA9876543210h
0x000007F706B08623	push rax
0x000007F706B08624	mov rax, 123456789ABCDEF0h
0x000007F706B0862E	push rax
0x000007F706B0862F	ret

표 2의 16진수코드렬은 등록기들과 기발변수를 탄창에 보관하고 0x123456789ABCDEF0(후킹처리부분)으로 이동하는 아쎔블리코드이다.

특히 0x7F706B08619부터 0x7F706B0862F까지의 아쎔블러명령은 0x123456789ABCDEF0에 위치한 함수를 호출하고 0xFEDCBA9876543210으로 귀환하는 과정이다.

다음의 코드를 실행함으로써 명령지적자는 0x7F706B08600에 위치한 아쎔블러명령을 실행한다.

```
DWORDdwOldProtect;
BYTEbzJumpCode[5];
bzJumpCode[0]=0xE8;
*(DWORD*)&bzJumpCode[1]=(DWORD)0x7F706B08600－(DWORD)<후킹주소> －5;
VirtualProtect((PVOID) <후킹주소>, 5, PAGE_EXECUTE_READWRITE, &dwOldProtect);
memcpy((PVOID) <후킹주소>, bzJumpCode, 5);
```

그다음 우와 같은 방법으로 등록기값들과 기발변수값을 탄창에서 읽어서 회복하고 원래의 위치로 돌아가 후킹처리를 결속한다.

맺 는 말

Windows 64bit에서 아쎔블리코드를 수정하여 후킹을 진행하는 한가지 방법을 제안하여 일반적으로 리용되는 Windows API후킹의 제한점과 부족점들을 극복하였다.

참 고 문 헌

[1] Greg Hoglund, James Butler; Rootkits: Subverting the Windows Kernel, Addison Wesley Professional, 250～352, 2005.

[2] Peter Gregory; Computer Viruses for Dummies, Wiley Publishing, 120～170, 2004.

[3] Wenhao Fan et al.; Computers & Security, 70, 224, 2018.

[4] Bill Blunden; The Rootkit Arsenal, Wordware Publishing, 450～560, 2009.

주체108(2019)년 8월 5일 원고접수

A Method of Assembly Code Hooking Implementation Using C Language in Windows 64bit Operating System

Ri Son Nam, *Jon Chol Yong*

In this paper, we have revealed some mistakes of Windows hooking API, which was commonly used in the past, and have presented a new method of Assembly code hooking which can work in global executable memory area.

Key words: Windows 64bit, C Language, code hooking