|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | |  | | KITRI 모의해킹 28기 | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  |  | | |  | |
|  | 루트킷 | | | | | | |  |
|  | | 시스템 해킹 | | | |  | | |
|  | |  | | | |  | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |
|  | | |  | |  | | | |

|  |
| --- |
| 1. 루트킷의 개요 |
| 본 자료는 안랩의 루트킷의 역사 시리즈 글을 참고하여 작성되었음. |
| 1.루트킷 |
| 특정 패스워드를 입력하면 별다른 인증 절차 없이 관리자(root) 권한으로 로그인하게 하는 서버 프로그램을 설치하거나 로그인 프로그램 자체를 자신이 만든 프로그램으로 교체하여 기능을 하는 프로그램들을 ’백도어’라고 불렀다.  또한 다른 시스템에 쉽게 침입하기 위해서 로컬 네트워크의 패킷들을 수집해서 아이디와 패스워드를 빼내거나 로그인 프로그램을 조작해서 다른 사용자들의 아이디와 패스워드를 훔쳐내는 기능을 하는 프로그램 등을 실행해두기도 하였다.  초기에 루트킷은 이런 백도어나 악의적인 기능을 가진 프로그램들을 통칭하는 용어로 사용됐다(root + (tool)kit). 그러나 점차 보안 프로그램 또는 운영체제로부터 악성코드를 보호하고 은닉하기 위해 특별히 제작된 프로그램을 지칭하는 용어로 의미가 변했다. |
| 2. 분류 |
| 1. 1세대 루트킷   1세대 루트킷은 운영체제의 커널이나 시스템 프로그램을 수정하기보다는 바꿔치기하거나 추가하는 기법을 사용한다. 앞의 설명처럼 유닉스 시스템의 로그인 백도어를 설치한다든지 특정 명령을 전송받는 네트워크 서버 프로그램을 실행해두는 방법이 가장 접하기 쉬운 형태의 1세대 루트킷들이다.  이런 형태의 루트킷은 시스템 설정 파일이나 원본 파일을 바꿔치기하거나 일부를 수정하기 때문에 원본 파일과 다른 파일이 시스템에 존재하게 된다. 따라서 원본 파일의 해시 값을 알고 있다면, 파일에 대한 무결성 검사만으로도 시스템에 침해가 발생했는지 여부를 확인할 수 있다.   1. 2세대 루트킷   1세대 루트킷처럼 파일 자체를 변경하는 것은 원본 파일의 해시 값을 이용한 무결성 검사 방식으로부터 자유로울 수 없었다. 따라서 해커들은 다른 방법을 생각하게 되었는데, 프로그램(실행 파일)이 실행되면 시스템의 가상 메모리에 로드되고, 메모리 공간에 로드된 코드가 실행된다는 점에 착안했다.  메모리 공간은 읽고 쓰기가 자유로운 공간이므로 메모리 영역에 로드된 프로그램의 코드를 수정하면 공격자가 원하는 기능을 구현할 수 있으며 원본 파일도 전혀 수정하지 않기 때문에 기존의 무결성 검증을 통한 루트킷 탐지 기술을 완벽하게 무력화할 수 있다.  이런 2세대 루트킷도 디스크상에 존재하는 원본 실행 파일이 메모리상에 로드된 상황을 시뮬레이션해서 현재 로드된 메모리 영역을 1:1 비교해보면 수정된 내역을 찾아낼 수 있다.   1. 3세대 루트킷   2004년 Jamie Butler는 블랙햇 발표에서 DKOM(Direct Kernel Object Manipulation)이라는 것에 대해서 발표했는데, 이후 대부분의 루트킷/안티루트킷 프로그램들이 이 기법에 대해 많은 연구를 진행하게 된 계기가 되었다.  DKOM은 메모리상에 생성되어 있는 커널 오브젝트를 직접 수정하는 기법이다. 원래 프로세스 리스트, 드라이버 모듈 리스트 등 동적으로 관리되는 메모리 영역 자체를 수정하여 프로세스를 숨기거나 커널 드라이버 목록을 숨기는 기능을 한다. |

|  |
| --- |
| 1세대 루트킷 |
| 1세대 루트킷은 대부분 유닉스 시스템에서 동작하는 백도어들이다.  백도어 때 진행했었던 /bin/ping파일의 변조(ping명령어로 동작하다가 특정 단어와 함께 입력 시 뒤에 오는 명령어를 실행)  pam\_unix.so 인증모듈의 변조와 같은 것들을 생각하면 된다. |
| 추가적으로는 소켓프로그래밍을 이용한 특정 포트에서 대기하도록 만들어 접속을 허용하는 코드이다(C의 sockaddr\_in 구조체를 사용한다, 소스는 별도 첨부) |
| … 중략 …  int main(int argc, char\* argv[])  {  int serv\_sock;  int clnt\_sock; //sockaddr\_in은 소켓 주소의 틀을 형셩해주는 구조체로 AF\_INET일 경우 사용    struct sockaddr\_in serv\_addr;  struct sockaddr\_in clnt\_addr; //accept함수에서 사용됨.  socklen\_t clnt\_addr\_size;  strcpy(argv[0],HIDE);  //TCP연결지향형이고 ipv4 도메인을 위한 소켓을 생성  serv\_sock=socket(PF\_INET, SOCK\_STREAM, 0);  if(serv\_sock == -1)  error\_handling("socket error");  //주소를 초기화한 후 IP주소와 포트 지정  memset(&serv\_addr, 0, sizeof(serv\_addr));  serv\_addr.sin\_family=AF\_INET; //타입: ipv4  serv\_addr.sin\_addr.s\_addr=htonl(INADDR\_ANY); //ip주소  // serv\_addr.sin\_port=htons(atoi(argv[1])); //port //소켓과 서버 주소를 바인딩  serv\_addr.sin\_port=htons(atoi(PORT));  … 중략 … |
| 위 소스처럼 특정 포트를 열고 특정 프로그램처럼 보이도록 위장을 한다. |
| |  | | --- | | telnet으로 로그인 시도 시 로그인이 되는 것을 확인 | |  | | 한계 및 결론 | | 1세대 루트킷은 프로그램을 바꿔치기하거나 새로운 프로그램을 백그라운드로 실행하는 정도였다. 디스크상에 존재하는 파일의 무결성이 훼손되기 때문에 1세대 루트킷은 시스템 파일을 변경하는 것이기 때문에 쉽게 탐지할 수 있다. | |

|  |
| --- |
| 2. 2세대 루트킷 |
| 2세대 루트킷은 프로그램이 실행되는 시점, 즉 디스크에 있는 파일이 메모리에 로드되어 실행되는 시점에 변경 을 가하는 방식으로 발전했다. 이렇게 실행되는 프로그램의 메모리 영역을 변경하면 디스크에 존재하는 파일의 무결성이 훼손되지 않기 때문에 기존의 방식으로는 탐지할 수 없게 된다.  즉, 2세대 루트킷은 변경될 이유가 없는 프로그램의 코드 영역 또는 초기화된 데이터 영역을 실행 시점에 변조 해서 공격자의 의도대로 프로그램의 로직을 변경하는 루트킷을 의미한다. |
| 1. PE 구조 |
| 윈도우는 프로그램의 실행 시점에 프로그램의 코드 영역을 변경하기(런타임 코드 패칭 : Runtime Code Patching) 위해서는 기본적으로 실행 파일의 구조에 대한 사전 지식이 필요하다.  윈도우 운영체제에서 사용하는 실행 파일은 PE(Portable Executable) 구조를 사용한다.  실행 파일은 크게 코드 영역과 데이터 영역으로 구분할 수 있다. |
| 코드 영역은 프로그램의 기능을 구현하는 로직, 즉 CPU의 명령어들이 저장된 영역이므로 프로그램 실행 시 메모리에 로드된 상황에서 변경될 이유가 전혀 없다.  데이터 영역 중 초기화된 데이터 영역 역시 코드 영역과 마찬가지로 프로그램 실행 도중 변경될 필요가 없다. |
| PE = PE 헤더+PE 바디  DOS header ~ Section header : PE 헤더 (헤더 영역)  그 아래 Section ~ : PE 바디(데이터 영역)  \*PE헤더에는 프로그램이 실행되기 위한 모든 정보가 적혀있다,  어떻게 메모리에 적재되고 어디서부터 시작하고 실행에 필요한 DLL은 무엇이 있고 필요한 stack/heap 메모리의 크기를 얼마나로 할지 등등.. |
| \*PE 종류는 실행계열과 라이브러리계열, 그 외 드라이버계열과 오브젝트 파일 계열이 있다. |
| |  |  |  |  | | --- | --- | --- | --- | | 종류 | 주요 확장자 | 종 류 | 주요 확장자 | | 실행 계열 | EXE, SCR | 드라이버 계열 | SYS, VXD | | 라이브러리 계열 | DLL, OCX, CPL, DRV | 오브젝트 파일 | OBJ | |
| 먼저 실행파일 메모장(notepad.exe)의 PE구조를 보면 |
|  |
| IMPORT 섹션과 EXPORT 섹션 |
| 실행 파일(.exe) PE는 디스크상에서 여러 종류의 헤더와 여러 개의 섹션(Section)으로 구분되어 있다. 위 그림의 PE 구조에서 런타임 코드 패칭과 밀접한 연관을 맺고 있는 것이 섹션 부분이다. 섹션은 성격이 비슷한 데이터, 코드들을 모아서 보관하는 영역이다.  대표적인 섹션으로는 .idata라는 이름의 Import 섹션과 .edata라는 이름의 Export 섹션, 그리고 ’.text’라는 이름을 가진 Text 섹션이 있다.  Import 섹션은 프로그램이 참조하는 외부 모듈(예: kernel32.dll, ntdll.dll 등)에 대한 정보를 보관하고 있고, Export 섹션에는 프로그램이 외부로 노출시키는 함수, 데이터들에 대한 참조 주소가 기록되어 있다. Text 섹션은 프로그램의 로직을 담고 있는 코드가 저장된 공간이다.  파일에서와 파일에서 메모리에 적재될때 가지는 표현이 다른데  파일에서는 Offset으로 메모리에서는 VA(Virtual Address, 절대주소)로 위치를 표현한다  파일이 메모리에 실제로 올라가면 그 모양이 변한다는 것만 기억하자  \*pe헤더(도스부터 섹션헤더까지)의 끝과 각 section의 끝에는 NULL PADDING이라고 부르는 영역이 있으며 이부분이 뻥튀기 되어 파일마다 모양이 달라지는 것)  \* NULL PADDING - 컴퓨터에서 파일, 메모리, 네트워크 패킷 등을 처리할 때 효율을 늘이기 위해 최소 기본 단위 개념을 사용하는 것. (생성되는 크기를 보면 규칙이 존재)  \* VA는 프로세스 가상 메모리의 절대주소를 말한다.  PE 헤더 내의 정보는 RVA(상대주소) 형태로 된 것이 많다, 그 이유는 PE 파일이 프로세스 가상 메모리의 특정 위치에 로딩될때 다른 파일이 (IMPORT) 되어있을 수 있기 떄문에 절대주소로 하면 당연히 중복나서 에러가 날 수 있기 때문이다, 그렇기 때문에 상대주소로 해놓고 위치를 유동적으로 함으로써 파일에 IMPORT를 여러 개 할 수 있게 된다. |

|  |
| --- |
| 2-1. IAT 후킹 |
| IAT란 (IMPORT ADDRESS Table) 이라는것으로 프로그램이 어떤 라이브러리의 도움을 받아서 그 라이브러리의 어떤 "함수"를 사용하는지를 기술하고 있는 테이블이다. |
| DLL (동적 연결 라이브러리) |
| 예전 16비트 DOS 시절 때에 DLL 개념이 없었을 시절, 그냥 Library만 존재할 때, 예를들어 C언어에서 printf() 함수를 사용할 때 컴파일러는 C라이브러리에서 해당 함수의 코드를 그대로 삽입해서 사용했으나 이는 매우 비효율적이다.  그래서 동일라이브러리가 실행되지 않기 위해 설계된 것이 바로 DLL, '프로그램에 라이브러리를 포함하는 대신에 별도의 파일로 구성하여 필요시 쓰겠다'라는 것이다.  일단 한번 로딩된 DLL의 코드는 리소스 Memory Mapping 기술로 여러곳에서 공유해 쓸 수 있다,  DLL 로딩 방식은 2가지이다.  프로그램이 사용되는 순간에 로딩하고 사용이 끝나면 메모리에서 해제되는 (Explicit Linking) 와 프로그램 시작시 로딩되어 종료할 때 메모리에서 해제되는 방법인(Implicit Linking) 방식이 존재한다.  우리가 다룰 것은 <프로그램 시작시 로딩되어 종료할 때 메모리에서 해제되는 방법인> IAT이다. |
| 메모장 프로그램의 Import 정보 |
| dependency walker 프로그램으로 메모장 프로그램의 Import 정보를 확인한 그림이다. 상당히 많은 외부 DLL들을 참고하고 있고, 각 DLL로부터 참조하고 있는 함수의 이름 또는 Ordinal 값이 보여지며 참조한 함수의 가상 메모리 주소 값도 확인할 수 있다. |
| IMAGE\_IMPORT\_DESCRIPTOR 구조체와 IMAGE\_IMPORT\_BY\_NAME 구조체 (출처:winnt.h) |
|  |
| 실행 파일의 Import 테이블의 구조 |
| 실행 파일의 Import 테이블의 구조를 좀더 자세히 표현한 것이다. 실행 파일의 Import 섹션은 IMAGE\_IMPORT\_DESCRIPTOR라는 구조체의 배열로 구성되어 있는데 이 구조체에는 Import한 DLL의 이름(예: ntdll.dll, kernel32.dll 등) 정보와 해당 DLL에서 Import한 함수들의 이름 또는 순서 정보를 담고 있는 IMAGE\_IMPORT\_BY\_NAME 구조체 배열의 시작 주소가 들어있다. Import 섹션이 가리키는 최종 목적지인 IMAGE\_IMPORT\_BY\_NAME 구조체 배열을 INT(Import Name Table)라 한다.  프로그램을 실행하면 윈도우의 실행 파일 로더는 실행 파일을 메모리상에 각 섹션별로 매핑한다. 이때 프로그램의 Import 섹션 정보를 읽어서 필요한 DLL을 함께 메모리에 로드한다. DLL이 정상적으로 메모리에 로드되면 Import Name Table에 명시된 함수들의 주소를 계산해서 테이블을 생성하고, IMAGE\_IMPORT\_BY\_NAME 구조체의 FirstThunk가 가리키던 값을 Import Name Table에서 생성한 테이블로 변경하는데 이 과정을 DLL 바인딩이라고 한다. |
| Import Address Table(IAT) |
| 이 DLL 바인딩 과정이 완료되면 FirstThunk 값은 Import Name Table이 아닌 Import Name Table에 명시된 함수들의 메모리상의 실제 주소 값을 가진 배열을 가리키게 된다. 참조된 함수들의 메모리상의 실제 주소를 가리키는 테이블을 Import Address Table(IAT)라고 한다.  \*FirstThunk 멤버 변수는 IAT(Import Address Table)의 RVA 값을 가지는 멤버 변수입니다. IAT도 INT와 마찬가지로 IMAGE\_THUNK\_DATA32 구조체배열로 구성되어 있다, 정확하게 말하자면, 바인딩(Binding)이라는 과정을 거치기 전에는 INT와 IAT는 완전하게 동일한 IMAGE\_THUNK\_DATA32 구조체 배열로 구성되어 있지만 바인딩 과정을 거치게 되면서 기존의 IAT의 IMAGE\_THUNK\_DATA32 구조체의 값들이 실제적인 함수의 주소로 바뀌게 된다. |
| IAT 훅이 설치된 이후의 메모리 상태를 나타낸다. |
|  |
| IAT 후킹 후 |
| 프로그램에서 advapi32.dll의 CryptEncrypt( ) 함수를 이용해서 중요한 정보를 암호화한다고 가정해보자. 응용 프로그램은 Import Address Table의 CryptEncrypt 함수를 호출하겠지만 실제로는 Import Address Table에 저장된 CryptEncrypt 함수의 주소가 HACKER.dll의 Hack\_CryptEncrypt 함수 주소로 변경된 상태이기 때문에 암호화되지 않은 데이터가 그대로 노출된다. |
| IAT 후킹의 한계 |
| IAT 후킹은 프로그램의 실행 시에 메모리에 동적으로 만들어진 Import Address Table(IAT)의 4바이트 주소 값만을 변경해서 프로그램의 로직을 변경할 수 있다. 또한 디스크상의 파일을 건드리지도 않기 때문에 매우 효과적인 방법이라 할 수 있다. 하지만 IAT 후킹에는 한계점이 존재하는데 참조 함수에 대한 명시적 호출을 수행하는 경우 IAT 훅이 동작하지 않는다.    위 코드처럼 외부 참조 함수를 명시적으로 불러다 쓰는 경우 프로그램의 Import Address를 참조하지 않기 때문에 프로그램의 로직을 변경할 수 없는 상황이 발생할 수 있다. |
| 2-2. EAT 후킹 |
| PE에는 Import 섹션 외에 ’.edata’라는 이름을 가진 Export 섹션이 존재한다고 앞에서 설명하였다.  DLL 바인딩 과정에서 윈도우 실행 파일 로더는 DLL을 로드하고, 로드한 DLL의 Export 섹션 정보를 분석한다. 또한 Import 섹션에서 명시한 DLL의 이름과 API의 이름 또는 서수 값(Ordinal)과 매칭되는 정보를 Export 섹션에서 찾아서 Import Address Table을 구성한다. |
| Export table의 구조 |
| kernel32.dll의 Export 섹션 정보를 보여준다. DLL의 이름, 서수 값(Ordinal), 함수의 이름, 함수의 메모리 주소 값이 출력됨을 확인할 수 있다.    Export table의 구조를 도식화한 것이다. Export table은 IMAGE\_EXPORT\_DIRECTORY라는 구조체로 정의되어 있는데 구조체의 가장 중요한 부분은 그림에 표시된 Export Address Table, Export Name Table, Export Ordinal Table이다.  Export Address Table은 DLL이 외부에 노출한 API들의 메모리상의 주소를 배열한 테이블이고, Export Name Table은 노출한 API들의 이름 문자열 배열이다. Export Ordinal Table은 노출한 API가 이름으로 노출되지 않고, 순서 값으로 노출된 경우 해당 API의 순서 값 배열이다. |
| Export Address Table의 메모리상의 배치 |
| 예를 들어, 디스크상의 DLL 파일에는 ActivateActCtx 함수의 주소 값이 0x000005B7로 기록되어 있고, 이 DLL이 0x7DD70000에 로드되었다면 메모리상에 만들어진 Export Address Table에는 0x7DD70587이라는 값이 설정된다. |
| EAT 후킹 |
| EAT 후킹은 메모리상에 로드된 Export Address Table의 주소 값을 바꿔치기하여 프로그램의 로직을 변경하는 것이다.  프로그램의 EAT가 변조되어 있기 때문에 프로그램에서 해당 DLL을 명시적으로 참조한다고 하더라도 변조된 로직이 정상 작동하기 때문에 IAT 후킹의 단점을 극복할 수 있다. |
| EAT 후킹의 한계 |
| 만일 변경하고자 하는 함수가 IAT에도 없고, EAT에도 없다면 어떻게 처리할 수 있을까? 혹은 함수의 중간쯤에서 변수 하나의 값만 필요에 따라 변경해야 한다면 IAT 후킹이나 EAT 후킹의 경우 PE 구조에 정의된 함수 포인터 배열의 값만 바꿔치기하는 방식이기 때문에 해당 함수가 테이블에 존재하지 않거나 특정 코드의 중간에서 로직을 변경하는 것은 불가능하다. |

|  |
| --- |
| 2-3. Inline 후킹 |
| EAT 후킹은 Export table에 있는 함수 주소의 값을 바꿔치기해서 다른 DLL 쪽으로 제어권을 넘기는 방식을 이용했다. Inline 후킹은 Export table에 있는 함수의 주소를 변경하는 것이 아니라 함수의 주소가 가리키는 실제 프로그램 코드를 변경한 것이다. |
| 기본 원리 |
| Export table의 구조를 좀더 자세히 표현한 것이다. Export Address Table이 가리키는 위치는 프로그램의 구현 코드가 저장된 메모리 공간의 주소이다. 여기에는 CPU 명령어들이 저장되어 있는데 이 영역에 직접 CPU 명령어를 덮어써 원하는 기능을 구현하는 것이 Inline 후킹의 기본 원리이다. |
| 세부 과정 |
| Inline 후킹을 하려면 먼저 루트킷 코드로 이동(Jump)하기 위한 CPU 명령어를 덮어쓸 위치를 찾아야 한다. 보통 함수의 시작 지점에 명령어를 덮어쓰는 경우가 많다. 함수의 중간에서 코드를 덮어쓰면 코드 실행 도중 상태 정보를 가지고 있는 변수나 스택의 상태 등을 원상 복원하는 데 문제가 발생할 수 있기 때문이다. 하지만 탐지를 회피하기 위해서 함수의 중간 부분에서 후킹을 하는 경우도 많다.    Inline 훅이 설치된 후 외부에서 함수를 호출하면 삽입된 jmp 코드로 인해 detour function으로 이동하게 된다. Detour function은 jmp 코드가 덮어쓴 원본 코드의 백업과 덮어쓴 이후의 위치로 jmp하는 코드를 실행한다.  결국 경로가 변경되었을 뿐 원본 함수의 실행에는 아무런 문제가 없다. 원본 코드, 즉 Target function이 실행되면 다시 detour function으로 리턴하고 detour function에서는 호출자에게 리턴하면 모든 호출 흐름이 종료된다.  Detour function은 이름처럼 함수 호출의 흐름을 변경하는 함수이며 호출자가 넘겨준 파라미터에 대한 조작이 가능하다. 또한 원본 함수가 실행되고 리턴된 값에 대한 조작도 가능하기 때문에 필요한 모든 기능은 detour function에서 구현하는 것이 일반적이다.  Inline 후킹은 거의 모든 함수에, 함수의 전체 길이가 jmp 코드를 삽입할 수 있을 정도의 크기만 된다면 적용이 가능하기 때문에 많은 악성코드나 보안 제품들이 주로 사용하는 후킹 기법이다. |

|  |
| --- |
| 2-4. DLL 인젝션 |
| 근대의 범용 운영체제는 한정된 물리적 메모리 자원을 효율적으로 사용하고, 응용 프로그램들이 독립적으로 자신만의 메모리 공간을 사용할 수 있게 하는 가상 메모리 시스템을 지원한다.  [프로세스 A]와 [프로세스 B]는 서로 독립적인 메모리 공간을 사용하고 있으며 상대방의 메모리 영역에 직접적으로 접근할 수 없다.    IAT 훅, EAT 훅, Inline 훅 등은 모두 실행 코드의 로직을 변경하는 방법이며 타깃 프로세스가 사용중인 메모리 영역을 수정해야 하는 기법들이다. 그런데 프로세스들은 서로 독립적인 메모리 공간을 사용하고 있기 때문에 정상적인 방법으로는 후킹이 불가능하다.    타깃 프로세스 공간에 실행 가능한 코드를 주입(injection)하는 방법이 필요하다. 실행 가능한 코드 블록을 주입하는 것보다는 다양한 기능을 쉽게 구현하기 위해서 DLL을 만들어 주입하는 방법을 이용한다. 이를 DLL 인젝션(Injection)이라고 한다.  일단 DLL이 타깃 프로세스 공간에 로드되면 타깃 프로세스와 동일한 권한을 가지고 코드가 실행되므로 타깃 프로세스의 메모리를 조작하는 데 아무런 문제가 없다. DLL 인젝션 기법은 여러 가지가 있지만 가장 대표적인 두 가지 방법에 대해서만 설명하겠다. |
| 1)SetWindowsHookEx( ) API를 이용한 DLL 인젝션 |
| 윈도우 운영체제는 이벤트라는 메시지를 통해서 응용 프로그램의 GUI와 운영체제 간의 통신을 처리한다. 예를 들어, 키보드가 눌리면 운영체제는 WM\_KEYDOWN이라는 메시지를 만들어 운영체제의 메시지 큐에 등록한다. 운영체제가 메시지 큐로부터 메시지를 꺼내어 현재 활성화된 응용 프로그램의 메시지 큐에 등록하면 응용 프로그램은 해당 윈도우 메시지를 처리하는 방식이다.  이 메시지 큐는 윈도우 메시지 훅 체인이라는 것을 가지고 있는데, 운영체제의 메시지 큐에서 애플리케이션 메시지 큐로 메시지가 전달될 때 미리 지정한 유형의 윈도우 메시지라면 애플리케이션의 메시지 큐에 등록하기 전에 등록된 훅 프로시저에게 먼저 처리할 수 있도록 우선권을 제공한다.    마이크로소프트가 만든 Spy++이라는 프로그램으로 윈도우 운영체제에서 발생하는 모든 윈도우 메시지를 볼 수 있다. 윈도우 메시지 훅 체인 메커니즘은 운영체제의 변경 없이도 응용 프로그램 개발자들이 GUI나 기타 애플리케이션의 기능을 자유롭게 확장할 수 있도록 만들어둔 장치로 보인다.  SetWindowsHookEx( ) 함수를 이용하면 간단히 윈도우 메시지 훅 프로시저를 등록할 수 있다.    코드는 'inject.dll'이라는 DLL의 'KeyboardHookProcedure'라는 함수를 WH\_KEYBOARD 윈도우 메시지의 핸들러로 등록하는 코드인데, SetWindowsHookEx( ) 함수의 마지막 파라미터가 0인 경우 운영체제의 모든 스레드에 메시지 훅 체인이 적용된다는 의미이다. WH\_KEYBOARD 메시지에 대한 훅을 설치했기 때문에 키보드 이벤트가 발생할 때마다 활성화된 윈도우를 가진 애플리케이션의 스레드에서 inject.dll의 HookProcedure 함수가 호출된다. Inject.dll의 HookProcedure 함수에서는 IAT 훅, EAT 훅, Inline 훅 등을 이용해서 다른 프로세스의 로직을 자유롭게 변경할 수 있다. |
| SetWindowsHookEx( ) API의 한계 |
| SetWindowsHookEx( ) 함수를 이용한 DLL 인젝션의 경우 기능 자체가 운영체제에서 제공하는 정식 기능이기 때문에 가장 안정적인 방법이며, 윈도우 9x부터 최신의 운영체제 버전까지 사용할 수 있다.  그러나 SetWindowsHookEx( ) 함수를 이용한 DLL 인젝션에는 큰 약점이 있다. 기본적으로 운영체제로부터 윈도우 메시지를 받기 위해서는 윈도우 핸들이 필요한데 이 윈도우 핸들은 GUI를 가지는 애플리케이션에만 있다. 따라서 콘솔 모드 응용 프로그램 또는 서비스 프로그램들은 DLL 인젝션이 불가능하다. |
| 2)CreateRemoteThread( ) API를 이용한 DLL 인젝션 |
| CreateRemoteThread( ) API는 다른 프로세스에 새로운 스레드를 생성하는 API로 여러 파라미터를 전달받지만 DLL 인젝션에 중요한 파라미터는 아래와 같다.    CreateRemoteThread( ) API를 통한 DLL 인젝션의 키 포인트는 4번째 파라미터인 LPTHREAD\_START\_ROUTINE lpStartAddress 파라미터이다. LPTHREAD\_START\_ROUTINE은 아래와 같이 정의되어 있다.        DWORD 값을 리턴하고, stdcall 호출 방식을 사용하며, 포인터 유형의 파라미터를 받는 함수이다. 그런데 윈도우 API 중에서 LoadLibrary( ) 함수가 있는데 이 함수는 DLL을 호출한 프로세스 공간에 로드하는 기능을 제공한다.        이 함수도 마찬가지로 HMODULE(DWORD와 차지하는 메모리의 크기가 같음)을 리턴하며 stdcall 호출 방식을 사용하고, 포인터 변수를 파라미터로 받아들인다. 결국 LoadLibrary( ) 함수는 PTHREAD\_START\_ROUTINE과 일치하기 때문에 CreateRemoteThread( ) API의 파라미터로 전달이 가능하다.    CreateRemoteThread( ) API를 이용한 DLL 인젝션은 다음과 같은 순서로 진행된다.    STEP #1 OpenProcess( ) API를 통해서 타깃 프로세스의 핸들을 오픈한다.    STEP #2 GetProcessAddress( ) API를 통해서 'LoadLibrary()' API의 주소를 구한다. DLL은 메모리 영역을 수정하지 않는 한 모든 프로세스가 공유하므로 injector 프로세스에서 구한 API의 주소와 타깃 프로세스의 주소가 동일하다.    STEP #3 CreateRemoteThread( )를 호출할 때 주입할 DLL의 경로를 파라미터로 전달한다.    STEP #4 타깃 프로세스에서 생성된 새로운 스레드는 CreateRemoteThread( ) API의 파라미터로 전달받은 LoadLibrary( ) 함수를 실행할 것이다. 또한 CreateRemoteThread( )의 다섯번째 파라미터로 주입할 DLL의 경로를 전달했으므로 결국 최종적으로는 다음의 코드가 실행된다.        이 모든 일련의 작업이 완료되면 타깃 프로세스 공간에서 IAT 훅, EAT 훅, Inline 훅 등을 이용해서 다른 프로세스의 로직을 자유롭게 변경할 수 있다. |
| CreateRemoteThread( ) API의 한계 |
| CreateRemoteThread( ) API는 SetWindowsHookEx( ) API와는 달리 윈도우 핸들을 가지지 않는 콘솔 프로그램 또는 서비스 프로그램에도 동일하게 DLL을 주입할 수 있기 때문에 가장 많이 사용된다. 하지만 윈도우 9x 계열 운영체제에서는 해당 API를 지원하지 않기 때문에 이 기법을 사용할 수 없다. |

|  |
| --- |
| 2-5. 런타임 코드 패칭 기법의 활용 |
| 리눅스는 LKM(Loadable Kernel Module)을 이용해서 동적으로 외부 모듈을 커널에 적재할 수 있는 기능이 있다. |
| 1) 리눅스 커널 루트킷 |
| 리눅스는 LKM(Loadable Kernel Module)을 이용해서 동적으로 외부 모듈을 커널에 적재할 수 있는 기능이 있다.  다음 예제는 루트킷이 LKM을 이용해서 운영체제에서 제공하는 시스템 콜 테이블을 변경하여 특정 조건이 일치하는 경우 관리자 권한(root)을 할당하는 예이다. 이후 /dev/ttyR0에 특정 단어(magic[ ] 에 정의된 단어)를 echo로 입력하면 해당 사용자를 루트(root)로 탈바꿈시킨다. Import Address Table, Export Address Table을 후킹하는 방식과 동일하다. |
| 실습 |
| # yum install kernel-headers-$(uname -r) kernel-devel-$(uname -r)  # vim root.o (소스는 별도 첨부)  # vim Makefile (소스는 별도 첨부)  # make  모니터링할 터미널을 별로도 연다  # tail -f /var/log/kern.log  # insmod ./root.ko  기본적으로 장치는 루트만 액세스할 수 있으므로 모듈을 로드한 후 사용 권한을 변경해야 한다.  # sudo chmod 0666 /dev/ttyR0  이후 일반 계정으로 로그인하여 확인  [user0@CS7-166 ~]$ id  uid=1000(user0) gid=1000(user0) groups=1000(user0)  [user0@CS7-166 ~]$ echo "g0tR0ot" > /dev/ttyR0  $ id  uid=0(root) gid=0(root) groups=0(root) |
| 2)윈도우 SSDT후킹 |
|  |
| SSDT ? |
| 윈도우 운영체제는 KeServiceDescriptorTable이라는 특별한 테이블을 관리한다. 이 테이블은 윈도우 커널의 일부로 운영체제가 응용 프로그램에게 제공하는 서비스(예: 파일 관리, 프로세스/스레드 관리, 메모리 관리 등)를 위한 함수의 배열이다.  이 테이블을 SSDT(System Service Descriptor Table)라고 하며 Import Address Table이나 Export Address Table처럼 함수 포인터의 연속된 배열이다.  이 테이블에 있는 함수의 주소를 다른 주소로 바꿔치기한다면 얼마든지 원하는 기능을 구현할 수 있다.  특히 SSDT는 응용 프로그램과는 달리 운영체제 커널의 일부이기 때문에 시스템에서 실행 중인 모든 프로세스에 적용된다.  따라서 악성코드에게는 최적의 공격 하거나 보안 프로그램에게도 SSDT가 악성코드 행위를 차단하기 위한 최적의 포인트가 된다. |
| 정상적인 과정 |
| 1. INT 0x2E나 SYSENTER 같은 시스템 콜이 호출된다. (XP 이후는 SYSENTER를 사용)  2. 시스템 콜이 호출되면 커널에서 KiSystemService(시스템 서비스 디스패처)를 호출한다.  - EAX에서 시스템 콜 번호를 읽어 SSDT의 인덱스로 사용  - EDX는 인자를 유저모드 스택에서 커털 모드 스택으로 복사한 곳을 가리킴  3. EAX에서 받은 시스템 콜 번호에 맞게 KeServiceDescriptorTable(SSDT)를 참조하여 Native API를 호출한다.  - [SSDT+EAX\*4]와 같이 참조  4. 시스템 콜 종료하고 유저 모드로 복귀한다. (XP 이후는 SYSENTER를 사용) |
| 시스템 프로세스 레코드의 구조 및 SSDT후킹의 원리 |
| 숨기고자하는 프로세스의 이전 프로세스의 포인터를 변조하여 숨긴다.    - SSDT(System Service Dispatch Table) : 시스템 콜 번호 순서대로 해당 함수의 메모리 주소를 갖는다.  - SSPT(System Service Parameter Table) : SSDT의 각 함수들에 대한 인자 크기를 갖는다.  - KeServiceDescriptorTable : SSDT와 SSPT의 BASE 주소를 갖는다. SSDT 안의 시스템 콜 개수를 갖고 있기 에 SSDT와 SSPT에 접근할 때 이용된다.  즉 이를 다시 해석하자면 SSDT(System Service Descriptor Table)는 윈도우즈의 모든 시스템 서비스의 주소를 가지고 있고 시스템 콜을 하면 커널은 KiSystemService 함수가 호출된다, 이 함수는 SSDT에 있는 원하는 시스템 서비스를 실행하게 되는데, KeServiceDescriptorTable를 통해서 KiServiceTable을 참고한다.  즉, 이 테이블에 있는 서비스의 원래 주소에 우리가 원하는 주소로 바꿔 놓는다면 후킹이 가능하다. |
| 디바이스 드라이버 |
| 디바이스 드라이버에 들어가기 전에 Ring에 대해서 알아보고 넘어가자. Windows에서 RING은 권한을 나타낸다. 앞에서 말했던 커널모드는 Ring0, 유저모드는 Ring3이다.    그림에서 볼 수 있듯 Ring0의 권한이 가장 높고 Ring3의 권한이 가장 낮다. 그리고 권한이 낮은 Ring은 권한이 높은 Ring의 메모리에 접근자체가 불가능하다. 참고로 Windows는 Ring0과 Ring3만을 사용한다. 디바이스 드라이버는 유저모드의 어플리케이션이 커널모드로 들어가기 위해서 걸쳐야 했던 과정을 거치지 않고 직접적으로 커널모드에 접근이 가능하다. |
| 메모리 쓰기 보호 |
| SSDT Hooking 하고자 하는 함수의 주소를 가지고 있는 테이블의 값을 디바이스 드라이버를 작성하여 내가 원하는 주소로 바꿔치기하면 된다. 하지만 여기에서 한 가지 문제가 생긴다. SSDT는 물리메모리에 존재하며 SSDT의 기본 속성이 Read Only이기 때문에 억지로 값을 바꾸면 블루스크린을 띄워버린다. 이는 Windows가 메모리를 보호하기 위해 Write Protection이라는 기법을 이용하여 메모리의 쓰기 권한을 삭제하였기 때문이다. SSDT Hooking을 하기 위해서는 이 Write Protection(이하 WP) 기법을 무력화 시켜야 한다. 무력화 시키는 방법은 CR0 레지스터를 이용하는 방법과 MDL(Memory Descriptor List)를 이용하는 방법 등이 있다.  CR0의 세부정보는 이렇다.    총 5개의 CR이 존재하고 그 중 CR0은 32bit로 이루어져 있는데 SSDT후킹을 위해 0번째부터 시작해서 16번째 있는 비트 값(WP)을 수정시켜야한다.  0 -> Write 가능  1 -> Read-Only  따라서 SSDT후킹을 위해서 0으로 클리어 해줘야한다. |
| CR 값 변경 |
| /\*CR0 레지스터의 WP비트 값을 0으로 셋팅(메모리 write)\*/  void MemoryProtect\_Off()  {  \_\_asm  {  cli //IF Register Disable  push eax  mov eax,CR0  and eax,0xfffeffff  mov CR0,eax  pop eax  }  }    /\*CR0 레지스터의 WP비트 값을 1으로 셋팅(메모리 read-only)\*/  void MemoryProtect\_On()  {  \_\_asm  {  push eax  mov eax,CR0  or eax, NOT 0xfffeffff  mov CR0,eax  pop eax  sti //IF Register enable  }  }  이처럼 CR0의 WP를 해제한 후  SSDT후킹을 통해 파일의 삭제방지, 폴더숨기기, 프로세스 숨기기가 가능해지게된다.  파일의 삭제 방지에는 커널에서 DeleteFile() API가 사용되는데 이는 ZwQueryInformationFile, ZwSetInformationFile, ZwClose 순으로 호출된다.  프로세스정보가 호출되는 과정에는 ZwQuerySystemInformation 함수가 이용된다.  따라서 프로세스 은닉에는 ZwQuerySystemInformation함수를 똑같이 생성하여 내가원하는 동작을 하도록 설정을하면 된다. 각 함수 및 구조체를 선언할 때에는 MSDN을 참조하여 원본 함수 및 구조체를 참고하여 생성하여 아래 그림처럼 루트킷으로 새로운 함수를 동작하여 원래 포인터가 가르키던 다음프로세스 레코드의 메모리 주소를 바꿔치기하여 숨기는 것이다. |
| 파일 숨기기 코드부분 |
| while(curr)  {  if (curr-&gt;FileName != NULL)  {  //파일명이 "Hook"으로 시작하거나 "TestFile.txt"인 경우  if((0 == memcmp(curr-&gt;FileName, L"Hook", 8)) || (0 ==memcmp(curr-&gt;FileName, L"HIDENPROCESS.sys", 32)))  {  //전 FileInformation이 존재하는 경우  if(prev)  {  //전 FileInformation의 NetxtEntryOffset에 다음 FileInformation의 위치를 저장한다.  if(curr-&gt;NextEntryOffset)  prev-&gt;NextEntryOffset +=curr-&gt;NextEntryOffset;  //마지막 프로세서일 경우  else  prev-&gt;NextEntryOffset = 0;  }  //전 FileInformation이 존재하지 않는 경우  else  {  //다음 FileInformation이 존재하는 경우  if(curr-&gt;NextEntryOffset)  {  //다음 FileInformation을 맨 앞의FileInformation으로 만든다.  (char \*)FileInformation +=curr-&gt;NextEntryOffset;  }  //마지막 프로세서일 경우  else  FileInformation = NULL;  }  }  }  //현재 FileInformation을 저장  prev = curr;  //다음 FileInformation으로 넘어감  if(curr-&gt;NextEntryOffset ) ((char \*)curr += curr-&gt;NextEntryOffset);  else curr = NULL;  } |
| 프로세스 숨기기 코드부분 |
| struct \_SYSTEM\_PROCESSES\* curr = (struct \_SYSTEM\_PROCESSES\*)SystemInformation;  struct \_SYSTEM\_PROCESSES\* prev = NULL;  while (curr)  {  if (curr->ProcessName.Buffer != NULL)  {  //프로세스명이 "cmd"인 경우  if (0 == memcmp(curr->ProcessName.Buffer, L"cmd", 6))  {  //전 SystemInformation이 존재하는 경우  if (prev)  {  //전 SystemInformation의 NetxtEntryDelta에 다음 SystemInformation의 위치를 저장한다.  if (curr->NextEntryDelta)  prev->NextEntryDelta += curr->NextEntryDelta;  //마지막 프로세서일 경우  else  prev->NextEntryDelta = 0;  }  //전 SystemInformation이 존재하지 않는 경우  else  {  //다음 SystemInformation을 맨 앞의 SystemInformation으로 만든다.  if (curr->NextEntryDelta)  {  (char\*)SystemInformation += curr->NextEntryDelta;  }  //마지막 프로세서일 경우  else  SystemInformation = NULL;  }  }  }  //현재 SystemInformation을 저장  prev = curr;  //다음 SystemInformation으로 넘어감  if (curr->NextEntryDelta) ((char\*)curr += curr->NextEntryDelta);  //다음 SystemInformation이 존재하지 않는다면  else curr = NULL;  } |
| 전체 코드 |
| 자세한 코드는 따로 첨부하였다. ssdt코드를 참조하기바란다. 해당 코드는 Visual Studio를 사용하며 빌드하려고 시도 시 에러로 인해 빌드되지 않을 것이다. 아마 Visual Studio 버전 및 wdk버전이 업데이트 되면서 해당 코드의 빌드를 허용하지 않는 것으로 보인다.  이러한 SSDT 후킹을 막기 위해 MS는 Patch Guard를 만들어내었고 현재는 또 이를 우회하는 방법이 있다. |
| 결론 |
| 2세대 루트킷 구현 기술은 한마디로 정리하자면 런타임 코드 패칭 기술로 운영체제 또는 소프트웨어의 설계•구현 수준에서 생각해 볼 때 변경되도록 의도되지 않았던 영역에 대한 변경인 셈이다. |
| 2세대 루트킷의 탐지 |
| 2세대 루트킷은 변경을 의도하지 않은 영역을 변경하는 특징을 지닌다고 했다.  다시 말하면, 프로그램의 코드 영역을 메모리상에서 변조하기 때문에 원래의 프로그램 코드와 실행시점에 메모리에 적재된 코드를 비교해 보면 쉽게 차이점을 발견할 수 있다.    실제로도 대부분의 안티-루트킷 소프트웨어들이 원본 코드와 메모리상의 코드를 비교하여 루트킷을 탐지한다.      루트킷 언후커(Rootkit Unhooker)라는 프로그램으로 코드패치 내역을 검사한 화면이다. 여러 개의 훅이 설치된 것을 확인할 수 있으며, ‘Type of Hook’ 부분을 보면 ‘Inline – Relative Jump’라고 출력된 부분을 확인할 수 있다. 디스크상의 원본 프로그램과 현재 메모리를 비교하여 훅의 설치 여부를 검증할 수 있으며, 당연히 후킹된 코드를 원래의 코드로 복원하는 것도 가능하다. |

|  |
| --- |
| 3. 3세대 루트킷 |
| 3세대 루트킷은 2세대 루트킷과 달리 데이터 영역, 즉 원래 변경되도록 의도된 영역을 변경시킨다. 따라서 원본 코드와의 비교를 통한 탐지가 불가능하며 프로그램 실행 중에 계속 변경되는 영역이 변경된 것이기 때문에 탐지가 매우 어렵다. 하지만 프로그램의 흐름을 직접적으로 제어하는 방식이 아니므로 원하는 기능을 구현하는 데 많은 제약사항이 따른다. |
| DKOM(Direct Kernel Object Manipulation) |
| 3세대 루트킷 기술은 일반적으로 DKOM(Direct Kernel Object Manipulation)이라는 이름으로 불린다.  DKOM에 대해서는 FU라는 아주 유명한 루트킷의 소스 코드를 살펴보는 것이 가장 확실한 방법이다. 참고로 FU 루트킷의 제작자가 바로 제임스 버틀러이다. FU 루트킷의 소스 코드는 예전에는 루트킷닷컴(rootkit.com)에서 다운로드할 수 있었으나 현재는 해당 사이트가 폐쇄되어 접속이 불가능하다. 하지만 검색 엔진에서 검색해보면 많은 링크에서 해당 소스 코드를 구할 수는 있다.  FU 루트킷은 프로세스 숨기기, 드라이버 숨기기, 권한 설정 기능 등을 구현한다. 이 모든 기능들은 코드에 수정을 가하지 않고, 운영체제가 서비스를 위해 특별히 관리하는 데이터를 직접 수정해서 구현한다. |
| 프로세스 숨기기  EPROCESS 구조체 |
| 윈도우 커널이 실행 중인 프로세스 관리를 위해 사용하는 EPROCESS 구조체의 일부이다. 프로세스는 사용자의 요청에 따라 생성되었다가 종료되기를 반복한다. 이렇게 동적으로 생성되었다가 종료되기를 반복하는 프로세스를 관리하기 위한 자료 구조로는 리스트 형태가 가장 알맞을 것이다. |
| 윈도우 커널의 프로세스 관리방식 |
| 윈도우 커널의 실행 중인 프로세스 관리 방식을 도식화한 것이다.  EPROCESS 구조체는 ActiveProcessLinks라는 2중 연결 리스트를 가지고 있는데 이 연결 리스트는 다른 EPROCESS 구조체의 ActiveProcessLinks 변수의 주소를 가리킨다. 새롭게 생성된 프로세스는 실행 중인 다른 프로세스의 ActiveProcessLinks와 연결되고, 프로세스가 종료되면 EPROCESS 구조체의 ActiveProcessLinks 리스트를 다른 EPROCESS와의 연결로부터 제거한다.  또한 윈도우 커널은 자체적인 관리 목적으로 PsActiveProcessHead라는 포인터를 외부에 노출하고 있는데 이 포인터는 그림과 같이 비어있는 2중 연결 리스트의 시작 주소를 가리킨다.  실행 중인 프로세스 전체의 목록을 구하기 위해서는 PsActiveProcessHead가 가리키는 2중 연결 리스트를 끝까지 순회하면 된다.  따라서 숨기고자 하는 프로세스의 EPROCESS 구조체의 ActiveProcessLinks의 백 링크(Back Link)를 자신의 ActiveProcessLinks의 포워드 링크(Forward Link)로 변경하면 된다. |
| 실습 |
| <https://docs.microsoft.com/ko-kr/windows-hardware/drivers/download-the-wdk> 접속    [Download Visual Studio Community 2019](https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=16) 를 다운하여 설치     데스크톱 모바일 에서 .NET과 C++ 2개를 체크하여 설치한다.    SDK 다운하여 설치  주의 사항은 설치경로가 Program Files아래 Windows Kits의 10 인지 확인을 해야한다.    WDK 다운하여 설치(경로확인)    설치 후 Empty WDM Driver가 보이는지 확인한다.    그 다음 첨부한 KernelMemoryIO를 열어 .sln 파일을 Visual Studio로 열어준다.    .exe 파일이 생성되었는지 확인한다.    이후 해당 KernelMemoryIO-master폴더를 가상 OS(윈도우 10으로 진행한다)로 옮긴다.    KernelMemoryIO.sys 파일의 경로를 확인 후 복사한다.  cmd창을 관리자권한으로 열어 해당 경로로 이동한 뒤  sc create "KernelMemoryIO" binpath= "<경로명>\KernelMemoryIO.sys" type= kernel start= demand  명령어를 실행  [SC] CreteService  성공 메시지를 확인  이 후 sc start kernelmemoryio를 사용하여    서비스를 구동 시킨다.  이 후 MemoryIOLib.exe 파일이 있는 경로로 이동하여    MemoryIOLib.exe를 실행하여 작업관리자에서 HideProcess라는 프로세스가 생성되는지 확인한다.        이 후 엔터를 눌렀을 때 해당 프로세스가 사라졌다가 나타나는 것을 확인한다.  해당 소스를 조금 변경하여 원하는 PID값을 가진 프로세스를 숨기는 것도 가능해진다.  이 후  운영체제보다 먼저 실행되어 운영체제를 변경하고, 실행 환경을 조작하는 루트킷인 부트 킷,  인텔 CPU의 VT-x기능을 활용한 HVM 루트킷,  특정 하드웨어의 드라이버로 위장하여 펌웨어에 숨어있는 펌웨어 루트킷,  MS의 PatchGuard를 우회할 수 있는 Infinityhook 등 여러가지가 나왔고 현재도 끊임없이 나오고있다. |

|  |
| --- |
| 4. 루트킷 탐지 |
| 간단 정리  **루트킷**  유저 모드 및 커널 모드 루트킷 기술의 이해   * 루트킷 기술은 Anti-Virus, 가상 드라이브 프로그램 등에서 활용 * 악성코드는 분석 방해, 은닉 등의 목적을 위해서 사용   유저 모드 루트킷   * 응용프로그램의 권한(Ring3) 만을 이용해서 동작하는 루트킷 * 커널 모드 루트킷 기술에 비해서 탐지하기가 쉬움   커널 모드 루트킷   * 커널의 권한(Ring0)을 이용하여 동작하므로 행위의 제한 없음 * 유저 모드 루트킷에 비해서 상대적으로 탐지하기가 어려움 |
| 윈도우의 백도어 탐지 |
| * 프로세스 확인 - Process Explorer / 비정상적으로 포트를 열고있는 서비스 확인(Kill Process tree로 제거)   비정상 프로세스는 Parent가 <non-existent Parent>   * 레지스트리 확인 - 윈도우 시스템은 시스템의 운영과 관련해 재부팅되도 기본 설정 값이 변하지 않도록 레지스트리에 여러 값 기록, 백도어도 주로 레지스트리 이용, 백도어 삭제 시에는 레지스트리도 내용 확인해야 함, 레지스트리 편집기에서 [파일]-[내보내기] 메뉴 선택, 레지스트리를 txt 파일로 내보냄, 메모장에서 열어 백도어파일명을 찾으면 검색 결과 없음(ex. 백도어 파일명이 system.exe일 때 system.exe의 아스키 값은 73,79,73,74,65,6d,2e,65,78,65 레지스트리에서는 유니코드 형태(1글자당 2바이트)로 저장, 중간에 00을 추가하여, 73,00,79,00,73,00,74,00,65,00, 6d,00,2e,00,65,00,78,00,65,00로 검색, 레지스트리 파일에서 해당 내용 확인) 발견이 된다면 레지스트리 편집기 이용 해당경로로 이동하여 삭제 필요 * 백도어 제거 - 백도어 프로세스의 중지, 백도어 파일의 삭제, 레지스트리 삭제 * 시스템 무결성 검사 - 윈도우에서는 SFC(System File Checker) 기본 제공(스캔은 ‘sfc /SCANNOW’명령 통해 수행) |
| 실행 중인 프로세스 목록 확인 |
| Process Enumeration  실행 중인 프로세스를 식별하고 그 목록을 나열하는 행위 |
| 은닉된 프로세스 확인 |
| Process Enumeration within the context of malware forensics   * 악성코드 중상 당수는 “쉬운 탐지”를 피하기 위해 프로세스를 은닉 * 어떠한 process enumeration 및 추적이 은닉된 프로세스 탐지에 사용 * 이 기법은 악성코드 탐지에 활용 |
| **Active Process Link Traversing vs Hooking** |
| System Service Call Path & Hooking Points    SSDT Hooking을이용한프로세스은닉   * 공격자는 SSDT의 NtQuerySystemInformation함수의 주소를 교체해서 자신의 함수가 먼저 호출 * 공격자의 함수 내에서 원래의 NtQuerySystemInformation함수를 호출해서 얻은 결괏값을 변경     Active Process LinkActive Process Link Traversing vs Hooking   * 윈도우는 EPROCESS 오브젝트들을 EPROCESS.ActiveProcessLinks를 이용하여 링크드 리스트 형태로연결하여 관리 * 커널 변수인 PsActiveProcessHead는 해당 링크드 리스트의 헤더로 사용       변경된 NtQuerySystemInformation()    Active Process Link Traversing vs Hooking   * ActiveProcessLinkTraversing은 윈도우 API를 사용하지 않고, 직접 커널의 자료구조를 확인하는 방법 * Hooking으로 인해 윈도우 API가 훼손된 경우에도 안정적으로 프로세스 목록은 획득 |
| Active Process Link Traversing vs DKOM |
| Direct Kernel Object Manipulation을 이용한 프로세스 은닉   * 프로세스 목록과 관련된 커널 오브젝트를 직접 조작하는 방식으로 프로세스를 은닉 * 주로 링크드 리스트나 프로세스 관련 정보를 가진 테이블을 공격     **EPROCESS Scanning**  Process Signature (Example)            **EPROCESS Scan을 이용한 Scan을 이용한 Process Enumeration**  Direct Kernel Object Manipulation을 이용한 프로세스 은닉   * 프로세스 은닉은 프로세스의 “존재” 사실을 감춤 * 프로세스는 non-paged memory pool 영역에 \_EPROCESS 형태로 존재 * 프로세스 목록을 가지고 있는 별도의 자료구조를 이용 * “Signature”를 활용, non-paged memory pool에 존재하는\_EPROCESS 타입의 오브젝트를 찾는다면 * API 또는 자료구조가 조작되었다 하더라도 프로세스 목록을 안정적으로 구할 수 있음 * “signature”를 비켜가는 방법으로 우회할 가능성은 남아있음     **PspCidTableEnumeration**   * Handle table for process and thread client ID * 모든 프로세스는 PspCidTable에 하나의 엔트리를 가짐 * 이 테이블을 조사하여 실행 중인 프로세스 목록을 구할 수 있음 * Hooking 또는 DKOM으로 은닉된 프로세스를 확인 * PspCidTable과 관련된 non-exported & undocumented 함수들이 존재 * 이용하여 PspCidTable 내 특정 프로세스 관련 정보를 지우는 방법으로 우회가 가능   은닉된 프로세스 탐지  psxview를 이용한 은닉 프로세스 탐지      1. CSRSS Handle Table에는 SystemTable에는 System, smss.exe, csrss.exe는 포함되지 않기 때문에 정상  2. pslist, psscan, thrdproc, pspcdid, csrss 모두 hooking을 이용하여 은닉된 프로세스를 탐지  4 .psscan의 결과만 True인 경우 은닉된 프로세스일 수도 있으나 대체로 종료된 프로세스일 가능성이 높음  5. 정확한 것은 EPROCESS  6. DKOM 기법을 이용하여 은닉된 프로세스 탐지 |
| Code Injection |
| Code Injection Overview     * 타겟 프로세스에 새로운가 상 주소 공 간을 할당 * 실행시킬 코드를 복사한 후 EIP 값을 주입된 코드의 주소로 변경 * 별도의 프로세스를 생성하는 대신 정상적인 프로세스 안에 코드를 주입 하 여실 행시 키는 방법을 이용 * 은닉 효과를 누리고자 하는 목적 * 프로세스 기반의 firewalling을 우회하고자 하는 목적   where is the code?   * Resource Section에 저장한 코드를 타깃 프로세스에 주입 * 프로세스 이미지 전체를 읽어 타겟 프로세스에 그대로 주입 * 코드 섹션, 데이터 섹션 등에 준비해둔 코드를 타겟 프로세스에 주입   **Virtual Address Descriptor**   * 프로세스가 할당받아 사용하고 잇는 가상 주소 공간에 대한 정보를 가지고 있는 구조체 * 시작 주소를 기준으로 한이 진 트리 형태로 구성 * EPROCESS.VadRoot는 VAD Tree의 루트 노드를 가리킴     **VAD 분석**  vadinfo를 이용한 은닉 프로세스 탐지       1. EXECUTE 속성이지 정 되어있고 속성이 지정되어있고, 맵핑된 파일이 없음 맵핑된 파일이 없음. 코드 인젝션을 의심 2. EXECUTE 속성이 지정되어있지 않음 속성이 지정되어있지 않음, 코드가 실행될 수 없으므로 코드 인젝션 가능성 낮음 3. EXECUTE 속성이지 정 되어있으나 속성이 지정되어있으나, 관련된 파일이 맵핑되어있음 관련된 파일이 맵핑되어있음.   **VAD 분석**  Dumping Process’s Virtual Address Space  vaddump - VAD Tree를 트래버싱하여 프로세스가 사용하는 가상 주소 공간을 확인하고 이를 덤프      **부모 프로세스 확인**  volatility를 이용한 프로세스 상관관계 분석    step 1. 프로세스 목록 추출 및 프로세스 목록 추출 및 dot 파일 생성  CoreSecurity:zesrever$ vol.py-f malware-‐1.mem psscan-‐output=dot -‐output-file malware-‐1.dot    step 2. Graph 생성 및 분석    **커맨드 라인 파라미터 확인**  volatility를 이용한 커맨드 라인 파라미터 확인 가능    vol.py volshell -f 메모리 덤프 이미지 파일명  >>> ps()  #분석하고자 하는 프로세스의 ID와#분석하고자 하는 프로세스의 ID와 EPROCESS 구조체 시작 주소 확인    # PEB의 시작 주소 확인  >>> dt(‘\_EPROCESS’, eprocess구조체 주소)    # 프로세스 콘텍스트 변경  >>> cc(pid=프로세스 ID) pid=프로세스 ID)    # Process Parameter 값 확인  >>> dt(‘\_PEB’, peb구조체 주소)    # 실행파일 실행파일 full path 확인  >>> dt(‘\_RTL\_USER\_PROCESS\_PARAMETERS’, UserProcess Parameter주소)    **Code Injection 사용하는 API**  Step 1. CreateProcess(...CREATE\_SUSPENDED...)  CreateProcess()를 이용하여 악성코드 container 역할을 할 타겟 프로세스를 suspend 모드로 실행    Step 2. VirtualAlloEx()  VirtualAllocEx()를 이용하여 타깃 프로세스에 새로운 가상 메모리 영역 할당    Step 3. VirtualProtectEx()  VirtualProtectEx()를 이용하여 생성된 가상 메모리 공간의 Protection 속성 지정 (READ, WRITE, EXECUTE)    Step 4. ReadProcessMemory(), WriteProcessMemory()  자신의 프로세스 이미지를 읽은 후 이를 생성된 가상 메모리 공간에 복사    Step 5. GetThreadContext(), SetThreadContext()  메인 스레드의 EIP    Step 6. ResumeThread()  스레드 실행    **커맨드 라인 파라미터 및 환경변수 추출**  Process Parameter Block |