Anti Reversing Technic

< 목 차 >

1. Symbol 정보 제거

☑ Compiler 기반의 Program 이라고 하더라도 Program Import Table, Export Table 안에는 약간의 Symbol 정보가 들어갈 수 있다. Program 이 많은 DLL 을 포함하고 있고, DLL 들이 상당히 많은 함수들을 Export 하고 있다면 Export 함수들의 이름이 Reverser 에게 유용한 정보가 될 수 있다. 이를 위해 Export 함수의 이름을 서술적으로 바꾸는 것도 좋은 방법이다.

☑ Byte Code 기반의 언어인 경우에는 내부적으로 주소 대신 이름을 사용해서 참조하므로 Program 을 Compile 할 때 모든 내부적인 이름들이 유지된다. 이를 방지하기 위해서 단순히 문자열뿐 아니라 내부적인 참조 관계를 유지하기 위해 관련된 문자열들 까지 모두 교체 해야한다.

2. Program 난독화

☑ 정적 분석을 방지하기 위한 일반적인 방법은 Program 을 Compile 한 후에 Program 을 암호화 하고 Program 내부에 복호화를 수행하는 Code 를 삽입함으로써 이뤄 진다. 대부분의 경우 복호화를 수행하는데 필요한 Code 가 프로그램 안에 존재(복호화 Logic 뿐 아니라 복호화 Key 까지) 하므로 완벽한 방어 법은 아니다.

☑ 자동 Unpacking Program 의 경우 Program 에 구현된 특정 암호 Algorithm 과 복호화 Key 를 자동으로 찾아내 Program 을 복호화 해준다. 이를 방지 하기 위한 한가지 방법으로 Program 실행 시에 계산되는 Key 값을 이용하는 것으로 복호화 Key 를 숨기는 것이 있다.

* 이는 Program 의 다양한 부분에서 연속적으로 참조하고 변경하는 여러개의 전역변수를 이용해서 구현 가능하다. 즉, 복호화 Key 가 필요할 때 마다 전역변수를 복잡한 수학 공식의 일부분으로 사용하는 것이다.

3. Anti Debugging Code 삽입

1) Notice (Debugger 제어)

Software Breakpoint	Breakpoint 를 설정하면 Debugger 는 해당 명령을 int 3 으로 교체 한다. * 이는 Breakpoint interrupt 로써 Code 의 실행이 Breakpoint 에 도달했다고 Debugger 에게 통보해준다. * 통보를 받은 Debugger 는 int 3 명령을 원래의 명령으로 교체하고 Program 실행을 중지시켜 Program 의 상태를 조사 할 수 있게 만든다.
Hardware Breakpoint	Debugging 대상 Program 안의 어느 것도 변경하지 않고, 단순히 Processor 가 특정 Memory 주소에 대한 접근이 이뤄 질 때 Breakpoint 가 발생했다고 판단하는 것이다. * IA-32 Processor 에서는 이러한 Single Step 의 구현을 EFLAGS Register 의 TrapFlag(TF) 에서 구현하고 있다. * TF 가 ON 이 되어 있으면 Processor 는 각 명령이 실행된 이후에 Interrupt 를 발생(Num = 1)시킨다.

2) IsDebuggerPresent API

☑ IsDebuggerPresent API 는 현재 Process 의 PEB(Process Environment Block) 를 조사해서 User mode Debugger 가 동작중인지 판단한다. 이 방법은 단지 해당 API 를 제거함으로써 간단히 우회가 가능하므로, 이를 효과적으로 하기 위해 Program Code 안에 내부적으로 구현해야 한다.

```
1   mov eax,fs:[00000018]
2   mov eax,[eax+0x30]
3   cmp byte ptr [eax+0x2],0
4   je RunProgram
```

[그림 3 - 1] IsDebuggerPresent API Code

☑ 위 Code 는 TEB Data 구조체에서 Offset + 30 위치의 값을 가져오며, 그 값은 현재 Process 의 PEB 를 가리키는 Pointer 가 된다. 그 후, Offset + 2 위치에서 현재 Debugger 가 존재하는 지 여부를 나타내는 1 Byte 를 읽어 들인다.

✱IsDebuggerPresent API 를 사용하는 방법은 Program Code 안에 Assembly 언어 Code 를 삽입할 수 있는 능력을 필요로 한다.

3) SystemKernelDebuggerInformation

☑ NtQuerySystemInformation Native API 를 사용하면 Kernel Debugger 가 존재하는지 판단할 수 있다. 이 함수의 정보 요청 중 SystemKernelDebuggerInformation 요청 Code 를 사용하면 현재 Kernel Debugger 가 동작중인지 여부를 알아낼 수 있다.

```
1   ZwQuerySystemInformation(SystemKernelDebuggerInformation,
2       (PVOID) &DebuggerInfo, sizeof(DebuggerInfo), &ulReturnedLength);
```

[그림 3 - 2] SystemKernelDebuggerInformation 요청 Code

```
1  typedef struct _SYSTEM_KERNEL_DEBUGGER_INFOMATION {
2      BOOLEAN DebuggerEnabled;
3      BOOLEAN DebuggerNotPresent;
4  } SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
```

[그림 3 - 3] SystemKernelDebuggerInformation 응답 Data 구조체 정의

☑ System 에 Kernel Debugger 가 동작중이면 DebuggerEnabled 값이 TRUE 가 된다.

4) SingleStep Interrupt 를 이용한 SoftICE 탐지

☑ SoftICE 의 경우 SingleStep 분석을 위해서 int 1 명령을 사용하며 이에 대한 Handler 를 IDT(Interrupt Descriptor Table) 에 설정한다는 사실을 이용하는 것이다.

✱ 예외 Handler 를 설치하고 int 1 을 호출하여 예외 Code 가 전통적인 STATUS_ACCESS_VIOLATION 이 아니면 SoftICE 가 실행 중이라고 판단 할 수 있다.

```
__try
{
    _asm int 1;
}

__except(TestSingleStepException(GetExceptionInformation()))
{
}

int TestSingleStepException(LPEXCEPTION_POINTERS pExceptionInfo)
{
    DWORD ExceptionCode = pExceptionInfo -> ExceptionRecord -> ExceptionCode;
    if (ExceptionCode != STATUS_ACCESS_VIOLATION)
        printf ("SoftICE is present!");

    return EXCEPTION_EXCUTE_HANDLER;
}
```

[그림 3 - 4] C 언어로 작성한 탐지 Code

5) TrapFlag

☑ 현재 Process 의 TrapFlag(TF) 를 설정해서 예외 발생 여부로 Debugger 의 존재 여부를 판단한다.

✱ “예외를 Debugger 가 “삼켜” 서 발생하지 않는 다는 것”에 기반을 두고 있다. User mode 와 Kernel mode 모두 TF 를 사용하기 때문에 모든 Debugger 를 탐지할 수 있다는 장점이 있다.

```
BOOL bExceptionHit = FALSE;

__try
{
    _asm
    {
    pushfd
    or dwrd ptr [exp], 0x100 //Trap Flag 설정
    popfd //EFLAGS Register 로 값을 Load
    nop
    }
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    bExceptionHit = TRUE; //예외가 발생하면, Debugger 가 존재 하지 않음
}
if( bExcetionHit == FALSE )
    printf( "A Debugger is present!\n");
```

[그림 3 - 5] C 언어로 작성한 탐지 Code

6) Code Checksum

☑ Debugger 나 Breakpoint 를 설정하려면 Code 를 변경해야 하므로 실행 중 Code 의 일부분이나 전체에 대한 Checksum 을 계산하는 것은 강력한 Anti Debugging 기술이 될 수 있다.

* Program 에 대한 함수들의 Checksum 을 미리 계산한 다음에 임의로 함수를 선택해서 해당 함수가 변경됐는지 확인한다. 이 방법은 AntiDebugging 뿐만 아니라 CodePatching 을 방지할 때에도 효과적이지만 느린게 단점이다.

4. Anti Disassembler

1) Notice (Disassembler 에 대한 일반적인 사항 2가지)

☑ Linear sweep : 단순히 Module 전체의 Code 를 순차적으로 Disassemble 하는 방법

☑ Recursive traversal : Code 의 흐름을 따라가면서 명령을 분석하는 방법

Debugger / Disassembler 이름	**Disassembly** 방법
OllyDbg, IDA, PEBrowse Professional	재귀 순회 (Recursive Traversal)
SoftICE, WinDbg	선형 스윕 (Linear Sweep)

[표 4 - 1] Reversing Tool 에서 사용되는 Disassembly 방법

2) Linear Sweep Disassembler

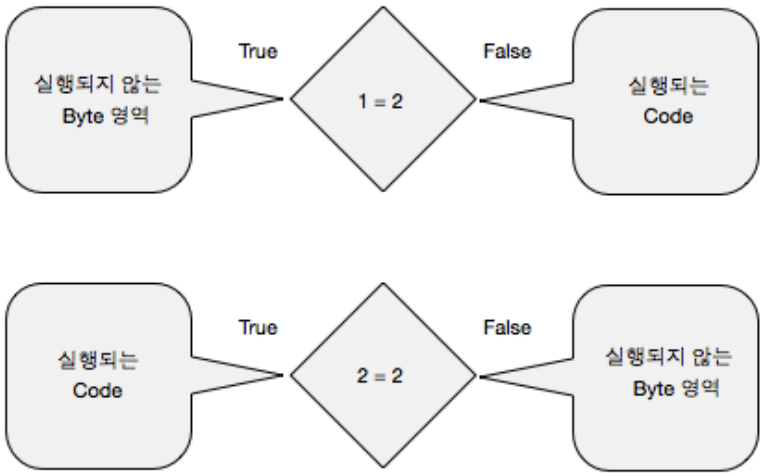
```
1   _asm
2   {
3       Some code ...
4       jmp After
5       _emit 0x0f
6
7       After:
8           mov eax,[SomeVariable]
9           push eax
10          call AFunction
11  }
```

[그림 4 - 1] Test Code

☑ Ollydbg 로 해당 Code 를 Load 하면 동일하게 보이지만, SoftICE 와 같은 Linear sweep Disassembler 에서는 실행될 일이 없는 0x0f 때문에 제대로된 Disassemble 을 하지 못한다.

3) Recursive Traversal Disassembler

☑ 재귀 순회 Disassembler 를 혼동 시키려면 opaque predicate 가 필요하다. 이는 FALSE 값을 가지는 분기 명령으로, 겉으로 보기에는 마치 조건 명령처럼 보이지만 실제로는 그렇지 않은 것 이다.. 분기 명령에서 Code 가 두갈래로 갈리게 되는데, 한쪽에는 실제 Code 가 있고, 다른 한쪽에는 아무런 의미가 없는 Data 가 존재한다.



[그림 4 - 2] 조건문의 결과가 False, True 가 되는 opaque predicate

4) Apply

```
1  #define paste(a,b) a##b
2  #define pastesymbols(a,b) paste(a,b)
3
4  #define OBFUSCATE()\
5  _asm {mov   eax, __LINE__ *0x635186f1          };\
6  _asm {cmp   eax, __LINE__ *0x9cb16d48          };\
7  _asm {je    pastesymbols(Junk, __LINE__)       };\
8  _asm {mov   eax, pastesymbols(After, __LIINE__) };\
9  _asm {jmp   eax                                };\
10 _asm {pastesymbols(Junk, __LINE__):            };\
11 _asm {_emit (0xd8 + __LINE__ % 8 )             };\
12 _asm { pastesymbols(After, __LINE__):          };
```

[그림 4 - 3] C Program 에 간단한 Assembly 언어 Code 를 추가하는 Macro

☑ 위의 Code 에서 mov 와 cmp 의 경우 현재 Code Line 번호를 이용해서 계산한 난수 값이며, 위의 Junk Byte 의 범위는 0xd8 ~ 0xdf 이다. 위 Macro 는 __LINE__ Macro 를 사용하여 Macro 가 호출될 때마다 매번 다른 Symbol 이름이 정의 된다.

✱ 위와 같은 Macro 의 가장 큰 문제점은 Code 에서 반복적으로 호출되므로 자동화 Tool 을 이용한 Macro 의 검색 및 제거가 가능하다.

5. 제어 흐름 변환

☑ Code 의 가독성을 떨어뜨리는 방향으로 Program 의 흐름과 순서를 변경하는 것이다.

계산 변환	Program 에서 제어 흐름 정보를 제거하거나, 새로운 제어 흐름을 추가
집합 변환	Programmer 가 작성중에 만든 High level 구조를 파괴하여 의미 없게 만듬
순서 변환	Program 의 연산 순서를 최대한 임의적으로 바꿔서 가독성을 떨어뜨림

1) Opaque Predicates

☑ 동시성 기반 Opaque predicate 를 예로 들면, 여러개의 Thread 를 생성하여 Thread 가 지속적으로 새로운 임의의 값을 생성해서 그것을 전역 Data 구조체에 저장하게 만든다. 그리고 전역 Data 구조체에 저장되는 값에는 항상 간단한 규칙이 적용된다. 실제 Program Code 를 포함하는 Thread 는 전역 Data 구조체에 저장된 값이 특정 범위 안에 있는지 확인하면 역 난독기 Tool 을 상당히 무력화 시킬 수 있다.

2) Anti Decompiler

☑ Byte Code 기반의 언어에는 실행 파일에 너무 많은 자세한 정보가 포함된다. 이를 방지하기 위한 방법으로 Program Binary 를 변경해서 Byte Code 가 원래의 High level 언어로 변환되지 못하게 하는것이 있다.

3) Table 해석

☑ Code 를 여러개의 작은 덩어리로 나누고, Code 가 Loop 를 돌면서 특정 시점에 어느 Code 를 실행 시킬 지 조건에 따라서 판단하면, Code 의 구조를 완전히 숨기므로 가독성을 떨어 뜨릴 수 있다.

✱ 추가적으로 JMP 문에 대한 Index 를 직접 이용하지 않고 실행 시에 채워지는 Table 을 추가적으로 이용하면 더 강력하게 구현이 가능하다.

☑ 또 다른 방법으로는 Pointer Aliases 를 이용하는 방법이 있다.

✱ 이는 어느 한 Pointer 를 이용해서 Memory 내용을 변경한 것이 또 다른 Pointer 를 이용해서 동일한 Memory 위치의 Data에 접근했을 때 어떤 영향을 미치는지 판단해야 하므로 Data 흐름 분석 과정을 상당히 복잡하게 만든다.

4) Inlining & Outlining & Interleaving Code

Inlining	하나의 함수 Code 를 여러 곳에서 호출하는 것이 아니라 호출 부분은 Code 자체로 대체 한다.
Outlining	함수 안에 있는 특정 Code 를 새로운 함수로 만들어 호출한다.
Interleaving Code	둘 이상의 함수 구현 내용을 서로 끼워 넣어서 분석하기 힘들게 만든다.

5) 순서 변환

☑ 함수 내에서 Code 에 의존적이지 않은 연산을 찾아서 순서를 변경하는 것

6. Data 변환

1) 변수 Encoding

☑ 변수 값의 의미를 직관적으로 파악하기 어렵게 만들어 버리는 것으로 Encoding 작업은 단순히 오른쪽이나 왼쪽으로 1bit Shift 연산 시키는 것으로도 충분하다.

```
1    for ( int i = 1; i < 100; i ++ )
2    for ( int i - 2; i < 200; i+=2 )
```

[그림 6 -1] 변수 Encoding 의 예

☑ 위의 Code 는 기능은 같으나 표현이 다르다. Encoding 을 좀더 복잡하게 수행하면 변수의 의미나 목적을 알아내기 힘들게 할 수 있다.

* 하지만 이 방법은 Compiler 가 Program 최적화 시에 이런 요소를 제거 또는 변경 하기 때문에 Binary level 에서만 적용되어야 한다.

2) 배열 재구성

☑ 여러개의 배열을 하나의 큰 배열로 병합(각 요소를 큰 배열에 삽입 또는 두 배열을 연속적으로 붙임) 함으로써 Reverser 를 혼동 시킨다.