ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

# Pin을 이용한 안티디버깅 우회 설계 및 구현☆

# The design and implementation of pin plugin tool to bypass anti-debugging techniques

홍 수 화[1] 박 용 수[1*]
Soohwa Hong Yongsu Park

요 약

Pin은 프로그램 동적 분석 도구를 생성할 수 있는 프레임워크로, 리눅스와 윈도우에서 사용자 영역의 프로그램 분석을 수행할 수 있게 한다. 역공학 방지 프로그램이나 악성코드는 프로그램 분석을 방해하는 안티디버깅이 적용되어 있기 때문에 Pin을 사용한 분석이 어렵다. 본 논문에서는 Pin을 이용해서 프로그램에 적용된 안티디버깅을 우회하여 동적 분석을 진행할 수 있는 Pin 플러그인 프로그램을 설계한 내용과 구현한 내용을 제안한다. Pin 탐지 안티디버깅을 우회할 수 있는 각각의 Pin 코드를 작성하고, Pin 코드를 하나로 합쳐 여러 안티디버깅을 우회할 수 있는 Pin 도구를 구현한다. 구현된 Pin 도구는 안티디버깅을 지원하는 프로텍터로 생성한 파일로 안티디버깅 우회 실험을 진행한다. 본 기법은 추후에 발견되는 안티디버깅 우회 코드 작성의 참고자료가 될 것이고 발견된 안티디버깅에 맞춰 수정 후 추가 적용이 가능할 것으로 예상된다.

☞ 주제어 : 안티디버깅, Pin, 동적 분석, 프로텍터, 역공학

ABSTRACT

Pin is a framework that creates dynamic program analysis tools and can be used to perform program analysis on user space in Linux and Windows. It is hard to analyze the program such as Anti-reversing program or malware using anti-debugging by Pin. In this paper, we will suggest the implementation of scheme bypassing anti-debugging with Pin. Each pin code is written to bypass anti-debugging detecting Pin. And Pin creates a pin tool combined with Pin codes that bypass anti-debugging methods. The pin tool are tested with files created by anti-debugging protector. The technique in the paper is expected to be a reference of code bypassing anti-debugging and be applied to bypass newly discovered anti-debugging through code modification in the future.

☞ keyword : Anti-debugging, Pin, Dynamic analysis, Protector, Reverse engineering

## 1. 서 론

프로텍터는 프로그램에 안티디버깅, 가상머신 탐지, 코드 난독화 등을 적용하여 역공학을 방지한다. 주로 소프트웨어 개발자가 자신들이 개발한 소프트웨어가 어떻게 동작하고 소프트웨어의 특별한 정보들을 보호하기 위해 프로텍터를 사용하여 프로그램을 보호한다. 문제는 악성코드 개발자들도 프로텍터를 이용해서 악성코드를 보호하는 것이다[1-3]. 악성코드가 어떤 악성행위를 하는지 분석하려 한다면 프로텍터가 추가한 안티디버깅 코드로 인해 분석기를 탐지하고 정상 실행 경로를 벗어나 프로그램을 종료시킨다. 결국, 악성코드를 분석하기 위해서는 분석기로 안티디버깅 기법을 우회하는 것이 필요하다.

Pin[4-5]은 프로그램 동적 분석 도구를 생성할 수 있는 프레임워크로, 리눅스와 윈도우에서 사용자 영역의 프로그램 분석을 수행할 수 있게 한다. 프로그램 분석은 Pin 내부 가상머신의 JIP(Just-in-time) 컴파일러를 이용하여 프로그램 코드 사이사이에 분석 코드를 삽입하고 프로그램 코드와 삽입된 분석 코드가 실행되면서 진행된다. 사용자는 분석 코드를 통해 동적으로 프로그램 정보를 얻을 수 있게 된다[6-7]. 프로그램 정보를 얻기 위해 삽입하는 분석 코드는 Pin이 제공하는 다양한 API와 C++ 언어를 사용하여 쉽게 만들 수 있다[8]. 이러한 특성을 이용하여 Pin을 탐지하는 안티디버깅 기법을 우회하는 코드를 삽입하여 안티디버깅 유무와 관계없이 프로그램을 분석하고자 한다.

[1] Department of Computer and Software, Hanyang University, Seoul, 133-791, Rep. of Korea.
* Corresponding author (yongsu@hanyang.ac.kr)
[Received 20 April 2016, Reviewed 3 May 2016, Accepted 25 August 2016]
☆ 이 논문은 2014년 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2012R1A1A2007263)

안티디버깅 기법이 탐지할 수 있는 분석기는 각각 다르다. 따라서 Pin을 탐지하는 안티디버깅 기법을 조사하고 탐지 방법을 연구한다. 연구 내용을 토대로 각각의 안티디버깅 기법을 우회할 수 있는 Pin 코드를 삽입하고 우회 실험을 진행한다. 문제는 하나의 프로그램에 여러 개의 안티디버깅 기법이 적용 될 수 있다. 그래서 각각의 안티디버깅 기법을 우회할 수 있는 코드를 하나로 통합하여 작성할 필요가 있다. 본 논문에서 각각의 안티디버깅 기법을 우회할 수 있는 코드들의 통합 알고리즘을 제시하고 안티디버깅 기법으로 프로그램을 보호하는 프로텍터를 대상으로 실험을 진행한다.

## 2. 관련 연구

새로운 악성코드들이 계속해서 생겨나면서 새로운 안티디버깅 기법들이 발견되고 있다. 따라서 새로운 안티디버깅 기법들에 대한 연구가 지속적으로 진행되고 있다. 현재의 안티디버깅 관련 연구 몇 가지를 살펴보면 다음과 같다.

Rodrigo Rubira Branco[9]의 연구를 보면 수십만의 새로운 악성코드가 널리 퍼지고 있다. 그리고 악성코드 개발자는 악성코드 분석에 방해하기 위해 분석 회피 기법들을 사용한다. 분석 회피 기법에는 디스어셈블리 방해, 안티디버깅, 가상머신 탐지가 있다. 400만개의 악성코드 샘플의 분석 결과로 6.42%가 분석 회피 기법을 이용한다. 분석 회피 기법을 사용하는 샘플 중 81.40%가 가상머신 탐지, 68.95%가 난독화, 43.21%가 안티디버깅, 12.13%가 디스어셈블리 방해 기법을 사용한다. 이 악성코드 분석 결과는 보안 회사와 악성코드 분석가들이 빠르게 분석 회피 기법을 우회할 수 있게 할 것으로 예상한다.

Adam J. Smith의 연구[10]에서 난독화된 안티디버깅 기법을 정적으로 자동 탐지하는 중간형태의 룰 엔진 탐지(REDIR)을 제안한다. 이 REDIR은 역공학 분석의 성능 향상을 위해 디자인 된다. REDIR은 3가지 원리를 기초로 한다. 중간형태가 명령어 집합을 줄여서 바이너리 프로그램 분석력을 향상시킨다. 그리고 전문가 시스템의 룰 엔진이 중간형태를 검색해서 안티 디버깅 기법 탐지를 위한 과정을 초기화를 한다. 마지막으로 중간 형태 분석 과정이 안티디버깅 기법의 존재를 확인한다. 이 REDIR은 분석기의 플러그인으로 사용되어 분석가가 안티디버깅 기법이 프로그램 흐름을 변경하는지 결정하게 한다.

Kota Yoshizaki의 연구[11]에서 안티디버깅 함수로 악성코드 탐지 방법을 제안한다. 안티디버깅 함수는 악성코드 분석가가 프로그램을 분석하는 것을 막는 방법이다. 안티디버깅 함수 후킹으로 리턴 값을 변경하여 프로그램 분석중이 아닌 상태로 만든다. 그리고 악성 프로그램과 악성이 아닌 프로그램 사이의 행위적 차이점에 초점을 두고 두 개의 행동 패턴을 비교하여 악성코드를 탐지한다. 행동 패턴의 차이는 컴퓨터 시스템 기동동안 레지스트리로 입력의 추가하는 것이다. 이것으로 악성코드 혼자서 자동 실행될 수 있게 한다.

## 3. Pin을 이용한 안티디버깅 우회 실험

Pin은 프로그램 동적 분석 도구를 생성할 수 있는 프레임워크로 다양한 API와 C++ 언어로 사용할 수 있다. 본 논문에서 Pin 2.14 버전으로 80x86을 사용하는 Windows 7 환경으로 가정한다.

### 3.1 Pin을 탐지하는 안티디버깅 기법

안티디버깅 기법마다 탐지할 수 있는 분석기가 다르다. 따라서 잘 알려진 안티디버깅 기법[12-15]을 대상으로 Pin을 발견하는 것을 조사할 필요가 있다. 그 결과는 다음과 같다.

(표 1) Pin을 탐지하는 안티디버깅 실험 결과
(Table 1) Anti-debugging test result for detecting Pin

| 안티디버깅 기법 | 탐지 |
|---|---|
| IsDebuggerPresent | X |
| CheckRemoteDebuggerPresent | X |
| OutputDebugString | X |
| FindWindow | X |
| NtQueryInformationProcess (ProcessDebugPort) | X |
| NtSetInformationThread Debugger Detaching | X |
| OllyDbg OutputDebugString() Format String | X |
| SeDebugPrivilege OpenProcess | X |
| NtQueryInformationProcess (ProcessDebugFlags) | O |
| NtQueryInformationProcess (DebugObjectHandle) | X |

| 안티디버깅 기법 | 탐지 |
|---|---|
| Hardware Breakpoints | X |
| VMware LDT Register Detection | X |
| VMware STR Register Detection | X |
| RDTSC | O |
| NTQueryPerformanceCounter | O |
| GetTickCount | X |
| timeGetTime | X |
| INT 3 Exception (0XCC) | X |
| INT 2D (Kernel Debugger Interrupt) | X |
| ICE Breakpoint | X |
| Single Step Detection | O |
| Unhandled Exception Filter | X |
| CloseHandle | X |
| Control-C Vectored Exception | X |
| Prefix Handling | O |
| CMPXCHG8B and LOCK | X |
| Memory Breakpoint | O |
| VMware Magic Port | X |

Pin은 기존의 28가지 안티디버깅 기법 중에서 6가지 안티디버깅 기법에 탐지된다. 즉, Pin이 대부분의 안티디버깅 기법에 탐지되지 않는다. 하지만 Pin 탐지 안티디버깅 기법이 하나라도 존재하면 프로그램 분석이 진행되지 않기 때문에 Pin 탐지 안티디버깅 기법들을 모두 우회할 수 있어야 한다.

안티디버깅 기법을 우회하기 위해서 각각의 안티디버깅 기법이 Pin을 탐지하는 방법을 알아야 한다. 본 절에서 Pin을 탐지하는 안티디버깅 기법을 간단히 소개한다.

### 3.1.1 Memory BreakPoint

```
lea ecx, [,dwpOldProtect]
push ecx
push 120
push 10
mov edx, [,dwMemResion]
push edx
call [VirtualProtect]
jmp [,dwMemResion]
```

(그림 1) Memory Breakpoint 예제
(Figure 1) Memory Breakpoint example

Memory Breakpoint는 Pin이 메모리의 PAGE_GUARD 설정을 무시하는 특징을 이용하여 실행 흐름을 변경하는 안티디버깅 기법이다. 특정 메모리 구역을 리턴 명령어로 채운다. VirtualProtect API를 이용하여 메모리 구역에 PAGE_GUARD 설정을 추가한다. 정상적인 실행에서 해당 메모리 구역을 호출하게 되면 PAGE_GUARD 설정으로 인해 예외가 발생한다. 발생한 예외를 처리하기 위해 SEH가 호출되고 예외가 처리된 후 다음 명령어가 실행된다. 하지만 Pin을 이용한 분석 중인 경우에는 해당 메모리 구역을 호출하게 되면 PAGE_GUARD 설정이 무시되어 메모리에 채워져 있는 리턴 명령어를 수행하게 된다. 리턴 명령어로 돌아온 후 예외 처리 없이 다음 명령어가 실행된다. 이러한 차이를 이용해서 SEH를 수행하지 않으면 프로그램이 종료되게 한다.

### 3.1.2 Prefix Handling

```
Prefix rep:
int1
```

(그림 2) Prefix Handling 예제
(Figure 2) Prefix Handling example

Prefix Handling은 분석기가 rep와 같은 prefix 다음의 바이트를 넘어가면서 명령어가 수행되지 않는 특징을 이용한 안티디버깅 기법이다. rep 명령어 다음에 인터럽트를 일으키는 명령어가 존재하지만 인터럽트가 발생하지 않게 된다. 따라서 SEH도 호출되지 않는다. 이러한 점을 이용해서 SEH를 수행하지 않으면 프로그램이 종료되게 설정하여 안티디버깅을 한다.

### 3.1.3 QueryInformationProcess(0x1f)

```
lea  eax, [.dwReturnLen]
push eax
push 0x4
lea  ebx, [.dwDebugProt]
push ebx
push 0x1f
push 0xffffffff
call [QueryInformationProcess]
cmp  dword [.dwDebugProt], 0
jne  .debugger_found
```

(그림 3) QueryInformationProcess(0x1f) 예제
(Figure 3) QueryInformationProcess(0x1f) example

QueryInformationProcess는 ntdll.dll의 API로 안티디버깅에 많이 이용된다. 이 API는 5 개의 파라미터를 가지고 있는데, 2번째 파라미터가 타겟 프로그램의 알고 싶은 데이터 타입을 나타내는 상수이다. 안티디버깅 기법에 사용되는 2번째 파라미터 상수는 ProecessDebugPort(0x07), ProcessDebugObjectHandle(0x1e), ProcessDebugFlags(0x1f)가 있다. 이 중에서 Pin을 탐지하는 안티디버깅 기법에 사용되는 2번째 파라미터 상수는 ProcessDebugFlags(0x1f)이다. QueryInformationProcess의 2번째 파라미터로 ProcessDebugFlags(0x1f)를 준 경우 3번째 파리미터로 지정한 메모리에 EPROCESS의 NoDebugInherit 값의 역수를 저장한다. 따라서 3번째 파라미터로 지정한 메모리에 0이 저장되어 있으면 Pin을 발견한 것을 나타낸다.

### 3.1.4 QueryPerformanceCounter

```
lea   eax, [.dwReturn1]
push eax
call [QueryPerformanceCounter]
lea   ebx, [.dwReturn2]
push ebx
call [QueryPerformanceCounter]
mov   eax, [.dwReturn2]
sub   eax, [.dwReturn1]
cmp   eax, 0xff
jnb   .debugger_found
```

(그림 4) QueryPerformanceCounter 예제
(Figure 4) QueryPerformanceCounter example

시간관련 안티디버깅 기법은 다양한 방법이 있다. 그 중에서 pin을 탐지하는 안티디버깅 기법의 예제이다. 그림 4의 안티디버깅 기법은 프로그램에 Pin과 같은 분석기가 붙으면 분석으로 인한 프로그램 수행에 추가 시간이 발생하는 점을 이용해서 분석기의 존재 유무를 확인한다. QueryPerformanceCounter는 kernel32.dll의 시간 관련 API이다. QueryPerformanceCounter는 하드웨어 성능 계수기의 현재 데이터 값을 파라미터에서 지정한 메모리에 저장한다. 안티디버깅은 QueryPerformaceCounter를 2번 호출한 후 두 개의 데이터 값의 차이가 일정 이상이 되면 프로그램이 분석되고 있는 것으로 판단한다.

### 3.1.5 RDTSC

RDTSC는 마지막 리셋부터 클락 사이클의 횟수를 기록하는 프로세서 타임 스탬프 값을 리턴하는 명령어이다.

```
RDTSC
mov    ebx, eax
RDTSC
sub    eax, ebx
cmp    eax, 0xff
jnb    .debugger_found
```

(그림 5) RDTSC 예제
(Figure 5) RDTSC example

QueryPerformanceCounter와 유사하게 시간 기반 안티디버깅에 많이 사용된다. 사용 방법은 QueryPerformanceCounter와 동일하게 RDTSC를 2번 수행 후 두 개의 데이터 값의 차이가 일정이상이 되면 프로그램이 분석되고 있는 것으로 판단한다. QueryPerformanceCounter와의 차이점은 RDTSC는 API가 아닌 명령어라는 점으로 데이터 값이 파라미터에서 지정한 메모리가 아닌 EDX:EAX 레지스터에 저장된다.

### 3.1.6 Single Step Detection

```
pushfd
or byte ptr [esp+0x1], 0x1
popfd
```

(그림 6) Single Step Detection 예제
(Figure 6) Single Step Detection example

Single Step Detection은 분석기가 single step 예외를 무시하는 특징을 이용하여 실행 흐름을 변경하는 안티디버깅 기법이다. pushfd 명령어로 스텍에 EFLAGS 레지스터의 값을 저장한다. 스텍에 저장된 EFLAGS 중 trap flag를 OR 명령어를 사용해서 1로 셋팅한다. 마지막으로 popfd 명령어로 스텍에서 변경된 EFLAGS 값으로 레지스트 값이 변경된다. 이 명령어가 수행된 후 single step 예외가 발생한다. 정상적인 실행에서는 single step 예외가 발생하면 SEH가 호출되고 예외가 처리된다. 하지만 분석기를 이용한 분석 중인 경우에는 예외를 무시하고 다음 명령어가 실행된다. 이러한 차이를 이용해서 SEH를 수행하지 않으면 프로그램이 종료되게 한다.

## 3.2 Pin을 탐지하는 안티디버깅 기법 우회 방법

본 절에서는 Pin을 탐지하는 안티디버깅 기법들을 우회하는 방법을 설명한다. 우회 방법은 Pin이 명령어 단위 분석을 하는 경우로 한정한다.

### 3.2.1 Memory BreakPoint

Memory breakpoint를 이용한 안티디버깅 기법은 pin에서 PAGE_GUARD가 설정된 메모리를 호출하면 예외를 발생시키도록 한다. 우선, 브랜치나 호출이 발생하는지 확인하기 위해 타겟 주소를 확인한다. 타겟 주소가 존재하면 PIN_CheckReadAccess(targetaddr)로 PAGE_GUARD가 적용된 주소인지 확인하고 예외를 발생시킨다. 예외 발생으로 SEH가 호출되면 memory breakpoint 기법을 우회되어 프로그램이 정상 실행과 동일한 루트로 진행한다.

```
VOID bypass(CONTEXT *ctxt,
 ADDRINT *instaddr, ADDRINT *targetaddr,
 THREADID tid){
  if(instaddr < 0x10000000 && targetaddr != 0){
    if(!PIN_CheckReadAccess(targetaddr)){
      EXCEPTION_INFO exceptInfo;
      PIN_InitExceptionInfo(&exceptInfo,
       EXCEPTCODE_ACCESS_INVALID_PAGE,
       (ADDRINT)instaddr);
      PIN_RaiseException(ctxt, tid, &exceptInfo);
    }
  }
}
```

(그림 7) Memory Breakpoint 우회 Pin 코드
(Figure 7) Pin code bypassing Memory Breakpoint

### 3.2.2 Prefix Handling

```
VOID bypass(CONTEXT * ctxt,
 ADDRINT *instaddr, USIZE* size,
 char *instr){
  if(strncmp(instr, "int", 3) == 0 &&
   (UINT32)size == 3){
    PIN_SetContextReg(ctxt,  REG_INST_PTR,
     (UINT32)instaddr+2);
    PIN_ExecuteAt(ctxt);
  }
}
```

(그림 8) Prefix Handling 우회 Pin 코드
(Figure 8) Pin code bypassing Prefix Handling

Pin이 prefix handling 기법을 수행하게 되면 오류가 발생하고 Pin이 중단된다. Pin은 prefix handling의 rep 부분을 인식하지 못하고 뒷부분의 int1 명령어만을 인식한다. 하지만 명령어의 길이는 rep 부분까지 포함해서 3으로 인식한다. Pin에서 int1 명령어 길이는 1이지만 prefix handling 부분으로 인해 3으로 인식되어 명령어 길이로 인한 오류가 발생한다. 따라서 Pin이 인식하지 못하는 prefix handling 부분을 넘어가서 int1 부분부터 실행되도록 한다. 현재 명령어가 int로 시작하지만 명령어 길이가 3인 경우 prefix handling이 적용된 것으로 판단하고 현재 명령어 포인터에 2를 증가시킨 지점에서 다시 실행되도록 변경한다. 결국 prefix handling 부분을 넘어가서 정상적으로 int1 명령어가 수행된다.

### 3.2.3 QueryInformationProcess

```
VOID bypass(VOID *instaddr, char *api,
 ADDRINT *eax, ADDRINT *esp){
  nowaddr = (ADDRINT)instaddr;
  if(nowaddr > 0x10000000 &&
   preaddr < 0x10000000)
    if(strcmp(api, qif) == 0)
      if(*(((UINT32*)esp)+2) == 0x1f){
        arg1 = ((ADDRINT*)*((UINT32*)(esp+3)));
        temp = 1;
      }
  if(temp == 1 && nowaddr < 0x10000000)
    if(eax == 0){
      (UINT32)*arg1 = 1;
      temp = 0;
    }
  preaddr = (ADDRINT)instaddr;
}
```

(그림 9) QueryInformationProcess 우회 Pin 코드
(Figure 9) Pin code bypassing QueryInformationProcess

QueryInformationProcess를 이용한 안티디버깅 기법을 우회하기 위해서는 3번째 파라미터로 지정한 메모리에 저장된 값을 1로 변경한다. QueryInformationProcess 안티디버깅 우회는 API 호출 확인 단계와 우회 설정 단계로 나누어진다. API 호출 확인 단계는 이전 주소가 사용자 공간 주소이고 현재 주소가 커널 공간 주소인지 확인한다. 사용자 공간 주소에서 커널 공간 주소로 점프하는 것을 API 호출로 판단한다. 호출된 Api는 RTN_FindNameByAddress (INS_Address(ins))의 값과 QueryInformationProcess 스트링이 저장된 qif와 비교하여 QueryInformationProcess가 호출된 것을 확인한다. 마지막으로 2번째 파라미터 값을 확인하기 위해 esp+2에 저장된 값이 ProcessDebugFlags (0x1f)인지 비교한다. 모든 조건을 만족시키면 안티디버깅 기법으로 판단하고 3번째 파라미터로 지정한 메모리

주소를 저장한다. 우회 설정 단계에서는 QueryInformation Process가 끝이나 사용자 공간으로 리턴하면 실행된다. eax값이 0인지 비교하여 QueryInformationProcess가 정상 수행된 것을 확인하고 저장된 메모리 주소를 이용해 메모리 값을 1로 변경한다. 결국, 변경된 값으로 인해 QueryInformacionProcess로는 Pin을 탐지하지 못한 것으로 나타나게 만든다.

### 3.2.4 QueryPerformanceCounter

```
VOID bypass(VOID *instaddr, char *api,
 ADDRINT *esp){
  nowaddr = (ADDRINT)instaddr;
  if(nowaddr > 0x10000000 &&
   preaddr < 0x10000000)
    if(strcmp(api, qpc) == 0){
      arg1 = ((ADDRINT*)*(esp+1));
      temp = 2;
    }
  if(temp == 2 && nowaddr < 0x10000000){
   LARGE_INTEGER *arg1 = timecount;
    timecount++;
    temp = 0;
  }
  preaddr = (ADDRINT)instaddr;
}
```

(그림 10) QueryPerformanceCounter 우회 Pin 코드
(Figure 10) Pin code bypassing QueryPerformanceCounter

QueryPerformanceCounter를 이용한 안티디버깅 기법을 우회하기 위해서는 2번의 호출 후 데이터 값의 차이가 작아야 한다. 그러기 위해서 QueryPerformanceCounter의 파라미터로 지정한 메모리에 저장된 값을 변경해야 한다. QueryPerformanceCounter도 API 호출 확인 단계와 우회 설정 단계로 나누어진다. API 호출 확인 단계의 처음으로 이전 주소가 사용자 공간 주소이고 현재 주소가 커널 공간 주소인지 확인하고 api와 qpc를 비교하여 QueryPerformance Counter가 호출된 것을 확인한다. 모든 조건을 만족시키면 안티디버깅 기법으로 판단하고 파라미터로 지정한 메모리 주소를 저장한다. 우회 설정 단계에서는 QueryPerformance Counter가 끝이나 사용자 공간으로 리턴하면 실행된다. 저장된 메모리 주소를 이용해 메모리 값을 timecount로 변경하고 timecount를 1증가시킨다. 첫 번째 QueryPerformance Counter의 값은 timecount고 두 번째 QueryPerformance Counter의 값은 timecount+1이 되어 QueryPerformance Counter로는 Pin을 탐지하지 못한다.

### 3.2.5 RDTSC

```
VOID bypass(char * inststr, ADDRINT *eax,
 ADDRINT *edx){
  if(strcmp(inststr, "rdtsc") == 0)
    temp = 3;
  else if(temp == 3){
    *edx = 0;
    *eax = timecount;
    timecount++;
    temp = 0;
  }
}
```

(그림 11) RDTSC 우회 Pin 코드
(Figure 11) Pin code bypassing RDTSC

RDTSC를 이용한 안티디버깅 기법을 우회하기 위해서는 2번의 호출 후 리턴 값의 차이가 작아야 한다. 그러기 위해서 RDTSC의 리턴값이 저장되는 edx와 eax의 값을 변경해야 한다. RDTSC 명령어 확인은 명령어가 RDTSC인지 확인한다. 현재 명령어가 RDTSC이면 명령어가 수행된 후 다음 명령어가 수행되기 전에 edx값을 0으로 수정하고 eax값을 timecount로 변경한다. 그리고 timecount를 1 증가시킨다. 첫 번째 RDTSC의 리턴값은 timecount이고 두 번째 RDTSC의 리턴값은 timecount+1이 되어 RDTSC로는 Pin을 탐지하지 못한다.

### 3.2.6 Single Step Detection

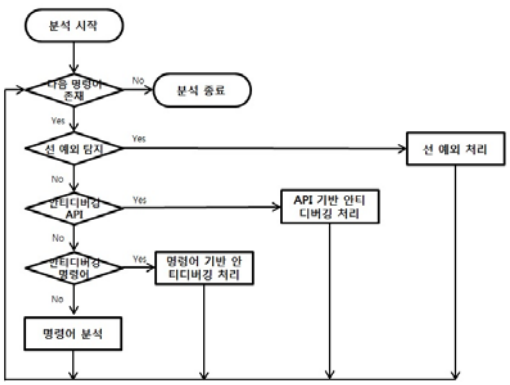
```
VOID bypass(CONTEXT *ctxt,
 ADDRINT *instaddr, char * inststr,
 THREADID tid, ADDRINT *esp){
  if(strcmp(inststr, "popfd") == 0 &&
   instaddr < 0x10000000){
    flags = (UINT32)*esp & 0x00000100;
    if(flags == 0x00000100){
      EXCEPTION_INFO exceptInfo;
      PIN_InitExceptionInfo(&exceptInfo,
       EXCEPTCODE_DBG_SINGLE_STEP_TRAP,
       (ADDRINT)instaddr+1);
      PIN_SetContextReg(ctxt, REG_ESP,
       (ADDRINT)(esp+1));
      PIN_RaiseException(ctxt, tid, &exceptInfo);
    }
  }
}
```

(그림 12) Single Step Detection 우회 Pin 코드
(Figure 12) Pin code bypassing Single Step Detection

Pin이 single step detection 안티디버깅 기법을 실행하게 되면 프로그램이 중단된다. Pin의 기본 설정이 single step 예외를 처리하지 못하기 때문이다. 따라서 single step 예외 발생을 미리 확인하고 Pin이 처리할 수 있는 예외를 발생시킨다. 우선, popfd 명령어를 만나게 되면 명령어를 수행하기 전에 스택에 저장된 EFLAGS값을 확인한다. EFLAGS에서 single step 예외를 발생시키는 8번째 비트 trap flag가 1인지 확인한다. Trap flag가 1로 single step 예외가 발생할 것을 확인하면 popfd 명령어가 수행된 것으로 하기위해 eip와 esp를 1 증가시키고 예외를 발생시킨다. Pin은 예외 발생으로 SEH를 호출하고 프로그램은 정상적인 루트로 실행하게 된다.

# 4. Pin을 이용한 안티디버깅 우회 알고리즘 실험

## 4.1 Pin을 이용한 안티디버깅 우회 알고리즘




(그림 13) Pin을 이용한 안티디버깅 우회 알고리즘
(Figure 13) Anti-debugging bypassing algorithm with Pin

프로그램의 분석을 방해하기 위해 여러 개의 안티디버깅 기법들이 적용된다. 따라서 3.2절에서 제시한 각각의 안티디버깅 기법 우회 방법들을 하나로 합쳐 사용할 필요가 있다. 그림 13은 안티디버깅을 우회하기 위한 Pin 코드 알고리즘이다. 우회 순서는 선 예외 탐지, 안티디버깅 API 확인, 안티디버깅 명령어 확인이다. 선 예외 탐지는 Pin이 인식하지 못하는 예외나 오류를 이용한 안티디버깅 기법을 우회하는 부분이다. Memory breakpoint와 prefix handling 기법이 여기에 속한다. 안티디버깅 API 확인은 현재 호출된 API가 안티디버깅에 이용되는 API이고 파라미터를 확인하여 안티디버깅을 위한 API 호출인지 결정한다. 안티디버깅 위한 API 호출로 확인되면 API가 끝이 난 후 리턴값을 수정한다. 안티디버깅 API 확인에는 QueryInformaionProcess와 QueryPerformaceCounter 기법이 속한다. 안티디버깅 명령어 확인은 명령어 패턴 비교를 통해 안티디버깅 기법에 쓰이는 명령어인지 확인한다. 안티디버깅 명령어 확인에는 RDTSC와 single step detection 기법이 있다. 본 절에서 제시한 알고리즘을 Pin이 명령어 단위로 반복 수행하면서 안티디버깅 기법을 발견하고 안티디버깅에 따른 각각의 우회 방법을 적용하면서 프로그램 분석을 진행한다.

## 4.2 안티디버깅이 적용된 helloworld 파일 실험

우회 알고리즘이 재대로 동작하는지 확인하기 위해, 3장에서 설명한 Pin을 탐지하는 6가지 안티디버깅 기법들을 모두 적용한 helloworld 출력 파일을 대상으로 실험을 진행한다.

helloworld 출력 파일에는 Memory breakpoint, QueryPerformace Counter, Single Step, Prifix Handling, RDTSC, Queryinformation Process 순으로 안티디버깅이 적용되어 있다. 우회 알고리즘이 포함된 Pin 코드를 실행하면, 프로그램 분석을 진행하는 도중에 선 예외 탐지에서 메모리 읽기 권한이 없는 주소로의 접근을 발견하고 예외를 발생시켜 Memory breakpoint를 우회한다. 다음으로 QueryPoerformanceCounter API가 호출이 되면, 안티디버깅 API임을 확인하고, API가 종료되어 리턴된 후 리턴된 값이 저장된 메모리를 수정하여 우회를 한다. 프로그램이 popfd 명령어 사용 시 우회 알고리즘의 안티디버깅 명령어 확인에서 스택의 탑 값의 8번째 비트가 1임을 확인하여 Single Step으로 판단하고 예외를 발생시킨다. 선 예외 탐지를 통해 명령어가 int로 시작하고 명령어의 사이즈가 3인 경우 Prefix handling 오류를 탐지하고 수정하여 실행한다. 안티디버깅 명령어인 RDTSC 호출 시 eax와 edx의 값을 각각 0과 timecount로 수정하여 안티디버깅을 우회한다. 마지막으로 QueryInformationProcess API 호출 시, 안티디버깅 API임을 확인하기 위해 2번째 인자값이 0x1f임을 확인하여 안티디버깅 API로 판단하고 API가 끝이 난 후 리턴값이 저장된 메모리를 1로 수정하여 우회한다. 그 결과, Pin을 탐지하는 6가지 안티디버깅 기법들을 모두 우회하고 정상적으로 helloworld가 출력됨을 확인하였다.

### 4.3 안티디버깅이 적용된 프로텍터 대상 실험

프로텍터는 프로그램에 안티디버깅, 가상화 탐지, 난독화, 암호 입력, 기간 설정 등을 적용하여 프로그램 분석으로부터 보호한다. 그중에서 프로그램의 역공학을 방지하기 위해 프로텍터는 안티디버깅 기법을 적용한다. 이 안티디버깅 기법을 지원하는 프로텍터를 사용하여 프로그램을 보호하고, Pin과 Ollydbg를 이용한 분석이 가능한지 확인한다. Pin은 우회 알고리즘이 포함된 경우와 포함하지 않는 경우에 대해 실험한다. Obsidium의 경우 가상머신에 사용되는 vboxguest를 이용해서 Pin을 탐지하기 때문에, 우회 알고리즘 이외의 가상머신 탐지를 처리해주는 부분이 추가적으로 필요하다. 따라서 본 실험에서는 프로텍터가 가상머신 탐지를 통해 Pin을 탐지한 경우를 제외한 안티디버깅 기법을 통해 Pin을 탐지하는 경우로 한정한 실험 결과이다.

(표 2) 프로텍터의 Pin과 Ollydbg 탐지 및 우회 결과
(Table 2) Pin & Ollydbg detection and bypassing results for each protector

| | Ollydbg | Pin 우회 알고리즘 X | Pin 우회 알고리즘 O |
|---|---|---|---|
| ASProtect | X | O | O |
| Enigma Protector | X | O | O |
| Themida | X | O | O |
| VMProtector | X | X | O |
| Obsidium | X | X | O |

표 2는 안티디버깅을 지원하는 5개의 프로텍터로 생성한 프로그램들을 Ollydbg, 우회 알고리즘이 적용되지 않은 Pin, 우회 알고리즘이 적용된 Pin으로 분석을 시도한 결과이다. 그 결과, 모든 프로텍터들이 안티디버깅 기법을 통해 Ollydbg를 탐지하고 분석을 중단시킨다. 우회 알고리즘이 적용되지 않은 Pin의 경우는 ASProtect, Enigma Protect, Themida로 생성된 프로그램들은 분석이 중단되지 않았지만, VMProtector, Obsidium으로 생성된 프로그램들은 분석이 중단되었다. 따라서 2가지 프로텍터의 경우에는 Pin을 탐지하는 안티디버깅 기법들이 적용되어 있음을 알 수 있다. 마지막으로 우회 알고리즘이 적용된 Pin의 경우는 5개의 프로텍터로 생성된 프로그램들을 분석이 중단되지 않고 실행되는 것을 확인하였다. 결국, Pin을 탐지하는 안티디버깅 기법들이 우회 알고리즘을 통해 우회되어 분석이 진행된 것을 알 수 있다.

Ollydbg의 경우 모든 프로텍터의 안티디버깅 기법에 탐지되어 분석이 중단된다. 모든 프로텍터들이 IsDebugger Present, CheckRemoteDebuggerPresent 등 여러 개의 안티디버깅 기법들을 프로그램에 적용하여 Ollydbg를 탐지하기 때문에, Ollydbg와 같은 기존 분석기를 이용한 분석이 어렵다.

우회 알고리즘을 적용하지 않은 Pin의 경우는 3가지 프로텍터 ASProtect, Enigma Protector, Themida가 적용하는 안티디버깅 기법 중 Pin을 탐지하는 안티디버깅 기법들이 존재하지 않기 때문에 정상적으로 분석이 진행된다. Enigma Protector에서 QueryPerformaceCounter API를 연속해서 호출하지만 리턴값을 Pin 탐지를 위한 안티디버깅에 사용하지 않는다. Themida는 RDTSC 명령어를 호출하지만 다음 명령어가 pop edx, pop eax이여서 RDTSC의 리턴값이 사용할 수 없게 된다. 따라서 RDTSC를 단순히 코드 난독화를 위해 호출한 것이다. 나머지 VMProtector과 Obsidium은 Pin을 탐지하는 안티디버깅이 적용되어 있어 분석이 중단된다. 따라서 Pin에 우회 알고리즘이 필요하다.

우회 알고리즘을 적용한 Pin의 경우는 모든 프로텍터의 안티디버깅 기법을 우회하여 분석이 진행된 것을 확인하였다. VMprotector는 RDTSC로는 Pin을 탐지하지 못하지만 프로그램 수행 중 popfd 명령어와 8번째 비트가 1인 스택의 탑 값을 이용하여 single step을 발생시켜 Pin을 중단시킨다. 하지만 우회 알고리즘의 안티디버깅 명령어 탐지에서 popfd 명령어일 경우 스택의 탑 값의 8번째 비트 확인을 통해 single step detection을 미리 탐지하고 예외를 발생시켜 single step detection을 우회하여 분석이 계속 진행되도록 한다. Obisidium은 single step detection과 QueryInformationProcess(0x1f)를 통해 2번 Pin을 탐지한다. 먼저, QueryInforamtionProcess API를 호출하여 안티디버깅을 시도하지만 우회 알고리즘의 안티디버깅 API에서 안티디버깅에 사용되는 API이고 2번째 인자가 0x1f임을 확인하여 Pin 탐지 안티디버깅 기법임을 확인한다. 그래서 QueryInformationProcess API가 종료된 후 리턴값이 저장된 메모리를 1로 수정하여 Pin이 탐지되지 않은 것으로 하여 분석이 진행되도록 한다. 그 후, single step detection은 VMProtector와 동일하게, 우회 알고리즘의 안티디버깅 명령어 탐지에서 popfd명령어일 경우 스택의 탑 값의 8번째 비트 확인을 통해 single step detection을 미리 탐지하고 예외를 발생시켜 분석이 계속 진행되도록 한다.

본 논문에서 제시된 알고리즘을 적용한 Pin 도구를 사용한 경우 프로텍터의 안티디버깅 기법을 성공적으로 발견하고 우회하여 분석이 막힘없이 진행하는 것을 확인하였다.

## 5. 결론 및 향후 연구

본 논문은 안티디버깅 기법을 포함한 프로그램을 대상으로 Pin을 탐지하는 안티디버깅 기법을 발견, 우회하여 프로그램 분석을 진행할 수 있는 알고리즘을 제시하였다. Pin을 탐지하는 6가지 안티디버깅 기법을 발견하였고 각각의 안티디버깅 기법을 우회하였다. 또한, 우회 방법을 통합한 알고리즘을 구현하여 안티디버깅을 제공하는 상용 프로텍터 5개를 대상으로 실험하여, Pin이 프로텍터에서 사용한 안티디버깅 기법들을 우회함을 확인했다.

이러한 연구 결과를 바탕으로 추후에 발견되는 안티디버깅 기법에 대한 Pin 우회 코드 작성의 참고자료가 될 것이고 발견된 새로운 안티디버깅 기법 우회 방법을 알고리즘에 추가 적용이 가능할 것으로 예상된다. 따라서 안티디버깅 기법을 적용하는 프로텍터로 보호되는 악성코드를 분석하는데 많은 분석 시간을 줄일 것으로 예상된다.

## 참 고 문 헌 (Reference)


[1] W. Yan, Z. Zhang, N. Ansari, “Revealing Packed Malware”, IEEE Security & Privacy, Vol.6, Issue 5, pp. 65-69, 2008.
http://dx.doi.org/10.1109/MSP.2008.126

[2] Dhruwajita Devi, Sukumar Nandi, “Detection of packed malware”, SecurIT ’12 Proceedings of the First International Conference on Security of Internet of Things, pp. 22-26, NY, USA, August, 2012.
http://dx.doi.org/10.1145/2490428.2490431

[3] Gabriel Negreira Barbosa, Rodrigo Rubira Branco, “Prevalent Characteristics in Modern Malware”, black hat USA 2014, Las Vegas, USA, August, 2014.
https://www.blackhat.com/docs/us-14/materials/us-14-Branco-Prevalent-Characteristics-In-Modern-Malware.pdf

[4] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Vijay Janapa Reddi, and Hazelwood, “K. Pin: building customized program analysis tools with dynamic instrumentation”, In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, June, 2005.
http://dx.doi.org/10.1145/1065010.1065034

[5] Steven Wallace, Kim Hazelwood, “SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance”, International Symposium on Code Generation and Optimization, San Jose, CA, March 2007.
http://dx.doi.org/10.1109/CGO.2007.37

[6] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo, ″Analysis of Computer Intrusions Using Sequences of Function Calls″, IEEE Transactions on Dependable and Secure Computing (TDSC), Vol 4, Issue 2, pp. 137-150, April, 2007.
http://dx.doi.org/10.1109/TDSC.2007.1003

[7] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. ″Dynamic Program Analysis of Microsoft Windows Applications″, International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.
http://dx.doi.org/10.1109/ISPASS.2010.5452079

[8] Pin 2.14 User Guide - https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/

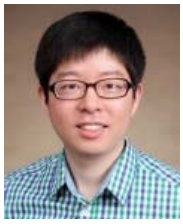
[9] RR Branco, GN Barbosa, PD Neto, “Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies”, black hat USA 2012, Las Vegas, USA, July, 2012.
https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_Slides.pdf

[10] A. J. Smith, R. F. Mills, A. R. Bryant, G. L. Peterson, M. R. Grimaila, “REDIR: Automated Static Detection of Obfuscated Anti-Debugging Techniques”, Collaboration Technologies and Systems (CTS), 2014 International Conference, Minneapolis, MN, USA, May, 2014.
http://dx.doi.org/10.1109/CTS.2014.6867561

[11] K. Yoshizaki, T. Yamauchi, “Malware Detection Method Focusing on Anti-debugging Functions”, Computing and Networking (CANDAR), 2014 Second International Symposium, Shizuoka, Japan, Dec, 2014.
http://dx.doi.org/10.1109/CANDAR.2014.36

[12] Tyler Shields. Anti-Debugging – A Developers View. Whitepaper, Veracode Inc, 2009.
[13] Peter Ferrie. The “Ultimate” Anti-Debugging Reference, May, 2011 - http://www.anti-reversing.com/the-ultimate-anti-debugging-reference/
[14] An Anti-Reverse Engineering Guide – http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide
[15] Introduction Into Windows Anti-Debugging – http://www.codeproject.com/Articles/29469/Introduction-Into-Windows-Anti-Debugging

## ◐ 저 자 소 개 ◑



**홍 수 화 (Soo-hwa Hong)**
2014년 한양대학교 컴퓨터공학부 졸업(학사)
2015~현재 한양대학교 대학원 컴퓨터소프트웨어학과 석사과정
관심분야 : 바이너리 코드 분석, 악성코드 분석.
E-mail : ghdtnghk@naver.com

**박 용 수 (Yong-su Park)**
1996년 KAIST 전산학과 졸업(학사)
1998년 서울대학교 대학원 컴퓨터공학과 졸업(석사)
2003년 서울대학교 대학원 전기컴퓨터공학부 졸업(박사)
2003년~2004년 서울대학교 자동제어특화연구센터 연수연구원
2005년~현재 한양대학교 소프트웨어전공 정교수
관심분야 : 컴퓨터 보안, 인터넷 보안.
E-mail : yongsu@hanyang.ac.kr