情報保護學會論文誌
第 22 卷 第 5 號, 2012. 10

바이너리 수준에서의 Jump-Oriented Programming에 대한 탐지 메커니즘

김 주 혁,† 이 요 람, 오 수 현‡
호서대학교 정보보호학과

A detection mechanism for Jump-Oriented Programming at binary level

Ju-Hyuk Kim,† Yo-Ram Lee, Soo-Hyun Oh‡
Dept. of Information Security, Hoseo University

요 약

컴퓨터 시스템의 안전성을 위협하는 주요 취약점으로 메모리 관련 취약점이 알려져 있으며, 최근 들어 이러한 메모리 취약점을 이용한 시스템 상에서의 실제 공격 또한 증가하고 있다. 이에 따라 시스템을 보호하기 위해서 다양한 메모리 보호 메커니즘들이 연구되고 운영체제를 통해 구현되어 왔지만, 더불어 이를 우회할 수 있는 공격 기법들 또한 발전하고 있다. 특히 버퍼 오버플로우 공격은 Return to Library, Return-Oriented Programming 등의 공격 기법으로 발전되어왔으며, 최근에는 Return-Oriented Programming 공격 기법에 대한 보호 방법 등의 연구로 인해 이를 우회하는 Jump-Oriented Programming 공격 기법이 등장하였다. 따라서 본 논문에서는 메모리 관련 공격 기법 중 최근 등장한 Jump-Oriented Programming 공격 기법에 대해 살펴보고, 이에 대한 특징을 분석한다. 또한, 분석된 특징을 통한 바이너리 수준에서의 탐지 메커니즘을 제안하고, 실험을 통해 제안하는 방법이 Jump-Oriented Programming 공격에 대한 탐지가 가능함을 검증한다.

ABSTRACT

It is known that memory has been frequently a target threatening the computer system's security while attacks on the system utilizing the memory's weakness are actually increasing. Accordingly, various memory protection mechanisms have been studied on OS while new attack techniques bypassing the protection systems have been developed. Especially, buffer overflow attacks have been developed as attacks of Return to Library or Return-Oriented Programing and recently, a technique bypassing the countermeasure against Return-Oriented Programming proposed. Therefore, this paper is intended to suggest a detection mechanism at binary level by analyzing the procedure and features of Jump-Oriented Programming. In addition, we have implemented the proposed detection mechanism and experimented it may efficiently detect Jump-Oriented Programming attack.

Keywords: Memory Corruption, Stack Ovcerflow, Jump-Oriented Programming, Return-Oriented Programming

I. 서 론

오늘날 대부분의 컴퓨터 환경에서의 위협은 주로 시스템 상에 존재하는 취약점을 이용하여 시스템을 해킹하는 공격 형태로 나타나고 있다. 이러한 공격의 대다수는 메모리 관련 취약점을 통해 발생하고 있으며,

접수일(2012년 4월 20일), 수정일(2012년 6월 27일), 게재확정일(2012년 8월 15일)
† 주저자, dreple@naver.com
‡ 교신저자, shoh@hoseo.edu

그 공격 방법도 다양하게 나타나고 있다. 특히, Stack smashing[1]이라고 불리는 최초의 버퍼 오버플로우 공격이 알려지면서 메모리 관련 공격은 시스템을 해킹하는 대표적인 공격 방법이 되었다. 이에 따라 현재 대부분의 운영체제들은 이러한 메모리 관련 공격을 방지하여 공격으로부터 시스템을 보호하기 위한 다양한 보호 메커니즘을 포함하고 있고, 이러한 메커니즘들을 통해 안전한 시스템 환경을 제공하고자 하고 있다[2][3][4][5]. 하지만 보호 메커니즘의 발전에도 불구하고 시스템의 안전성을 위협하는 공격 방법 또한 이를 우회할 수 있는 형태로 발전하여 시스템 상에 존재하는 보호 메커니즘들을 무력화시키고 있다.

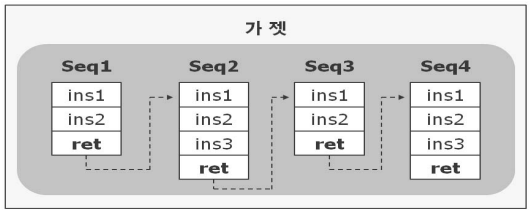
이러한 보호 메커니즘을 우회하는 공격기법들은 주로 기존 버퍼 오버플로우 공격의 발전된 형태이며, 공격자가 원하는 악의적인 코드를 실행하는데 목적을 두고 있다. 그 중 ROP(Return-Oriented Programming) 공격 기법은 시스템 라이브러리를 이용한 기존 RTL(Return to Library) 공격이 보호 메커니즘에 의해 더 이상 공격을 수행할 수 없는 한계점을 극복하기 위해 제안된 공격 기법이다. 이러한 ROP 공격 기법은 메모리 페이지 내의 실행을 방지함으로써 악의적인 공격을 방어하는 DEP(Data Execution Prevention) 보호 메커니즘[3]을 우회하여 공격자가 원하는 임의의 코드를 실행하기 위해, 기존 공격 기법들과 달리 공격자가 공격을 위한 연산 코드를 메모리 내의 위치한 명령어 시퀀스를 이용하여 직접 프로그래밍 할 수 있다는 장점이 있다. 또한 연산을 위한 코드는 다른 연산에서 재사용될 수 있다는 점에서 코드 재사용 공격이라고도 불린다. 이러한 ROP 공격의 등장으로 인해 공격에 대한 탐지 및 공격으로부터 시스템의 안전성을 유지하기 위해서 보호 대책에 대한 연구가 다양하게 진행되었다. 이러한 보호 대책에 관한 연구들은 또 다시 이를 우회할 수 있는 형태로 메모리 관련 공격 기법을 발전시켰으며, 최근 기존 ROP에서의 RET 시퀀스 대신 JMP 시퀀스를 사용하는 JOP(Jump-Oriented Programming) 라는 공격 기법으로 발전되었다. JOP 공격 기법은 기존 ROP 공격에서 사용되는 RET 시퀀스에 대한 검색의 한계점 및 ROP 보호 대책을 회피하기 위해 제안되었다. 따라서 이러한 새로운 공격 기법의 등장으로 인해 시스템 상의 안전성을 유지하기 위한 공격 탐지 방법이 요구된다. 본 논문에서는 JOP 공격의 특징을 토대로 공격 탐지 방법을 제안하기 위해 2장에서 관련연구로 이전 메모리 공격 기법인 ROP (Return-Oriented Programming) 및 그에 대한 보호 방법들을 소개하고, 3장에서는 ROP의 발전된 형태인 Jump-Oriented Programming 공격 기법에 대해 설명한다. 4장에서는 JOP 공격기법의 특징을 이용한 탐지 방법을 제안하고, 5장에서 탐지 방법에 대한 구현 및 실제 시스템 상에서의 성능 평가를 수행하며 마지막으로 6장에서 결론을 맺는다.

II. 관련 연구

2.1 Return-Oriented Programming

기존의 버퍼 오버플로우 공격이 메모리 보호 메커니즘의 발전에 따라 더 이상 수행되지 않기 때문에, 이를 개선한 공격 기법이 RTL(Return To Library) 공격 기법이다[6]. RTL 공격 기법은 시스템 라이브러리를 이용하여 원하는 쉘코드를 수행하고자 하는 공격 기법이다. 하지만 운영체제 에서 제공되는 메모리 보호 메커니즘에 의해 이러한 공격이 무력화됨으로써 RTL 공격을 응용한 ROP(Return-Oriented Programming) 공격이 제안되었다[7]. ROP 공격은 데이터 실행 방지 메커니즘인 DEP(Data Execution Prevention)를 우회할 수 있으나 각 메모리 영역을 랜덤화하는 full ASLR (Adress Space Layout Randomization) 메커니즘에 대한 우회는 불가능하다. 하지만 full ASLR이 적용된 시스템 환경에서도 부분적으로 ASLR이 적용되지 않는 취약점을 통해 공격 성공 기회를 제공하고 있기 때문에, 이를 이용하여 ROP 공격이 수행될 수 있다.

이러한 ROP 공격의 가장 큰 특징은 공격자가 메모리 상에 임의의 코드 삽입 없이도 시스템 라이브러리를 이용하여 공격자가 원하는 연산을 수행하여 최종적으로 쉘코드를 실행시킬 수 있도록 프로그래밍 할 수 있다는 점이다. 따라서 ROP 공격을 수행하기 위해서는 먼저 임의의 연산을 수행하는 가젯(gadget)을 구성하여야 하며, 이를 위해서는 시스템 상에 존재하는 기계어 코드 섹션에 대한 검색이 요구된다. ROP 공격을 구성하는 기계어 코드 섹션의 흐름 이동은 SP(Stack Pointer)를 통해 이루어지기 때문에 가젯 구성에 요구되는 각 기계어 코드 섹션은 스택 포인터를 증가시키는 RET 명령어로 끝나는 명령어 시퀀스로 존재하여야 한다. 이렇게 구성된 시퀀스들을 통해 ROP 공격을 수행하기 위해서는 먼저 운영체제 및 애플리케이션의 취약점을 이용하여 ROP를 구성하는

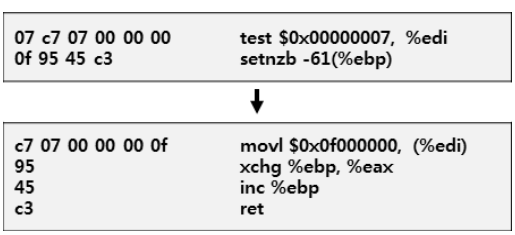


〔그림 1〕 ROP 공격을 위한 가젯 구성

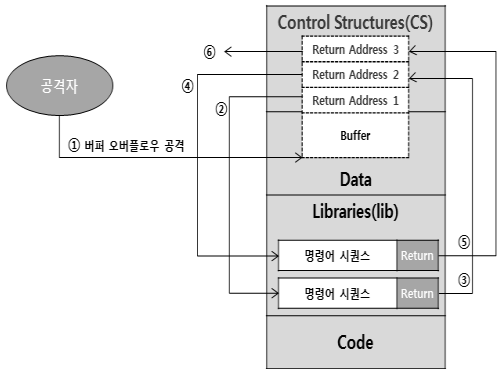
명령어 시퀀스로 실행 흐름을 변경하여 하며, 변경된 실행 흐름을 통해 각 시퀀스 내의 명령어가 수행되면서 마지막 RET 명령어로 인해 스택 포인터를 변경하기 때문에 공격자의 의도에 따라 시퀀스들을 통해 구성한 가젯들이 수행될 수 있고, 이를 통해 최종적으로 공격자가 실행하고자 하는 쉘 코드를 수행할 수 있다. 〔그림 1〕은 ROP 공격을 위한 가젯의 구성과 제어의 흐름을 나타낸다.

따라서, ROP 공격을 위해서는 RET 시퀀스에 대한 검색이 요구되며 이를 위해서 기존의 시스템 라이브러리에서 원하는 시퀀스를 검색하여 사용할 수 있으나, RET 명령어는 시스템 상에서도 많은 수가 포함되어 있지 않기 때문에 원하는 시퀀스에 대한 검색의 어려움이 발생할 수 있다. 따라서 〔그림 2〕와 같이 기존의 명령어에 대한 바이트 오프셋 변경을 통해서 RET 시퀀스를 구성하여 사용할 수 있다.

이렇게 검색된 시퀀스들은 스택 상에 시퀀스 주소를 삽입함으로써 이후 연산이 수행될 수 있도록 구성한다. 이후, ROP 공격을 수행하기 위해서는 먼저 운영체제 및 애플리케이션의 취약점을 이용하여 ROP를 구성하는 명령어 시퀀스로 제어 흐름을 변경하여야 한다. 이때 주로 사용되는 방법은 버퍼 오버플로우 취약점을 통해 제어 흐름을 변경한다. 각 명령어 시퀀스들은 명령어가 수행되면서 마지막에 존재하는 RET 명령어를 통해 스택 상의 다음 시퀀스를 수행하도록 SP를 변경하기 때문에 공격자의 의도에 따라 시퀀스들을



〔그림 2〕 바이트 오프셋 변경을 통한 RET 시퀀스 구성 방법



〔그림 3〕 Return-Oriented Programming

이용하여 구성된 가젯들이 수행될 수 있고, 이러한 가젯들을 통해 최종적으로 공격자가 실행하고자 하는 쉘 코드를 수행할 수 있다. 〔그림 3〕은 ROP 공격에 대한 전체적인 제어 흐름을 나타낸다.

2.2 Return-Oriented Programming에 대한 보호 기법

ROP 공격 기법에 대한 대응방법은 다음과 같은 두 가지 수준으로 구분할 수 있다.

- 컴파일러 수준의 보호 대책 : ROP 대응책이 컴파일러 또는 링커에 의해 적용되며, 프로그램의 소스 코드가 요구된다. 보호 대책에 대한 적용은 코드의 재작성 등을 통해 수행된다.
- 바이너리 수준의 보호 대책 : 소스 코드를 요구하지 않으며, 프로그램 수행 중에 보호 대책을 적용하여 ROP 공격에 대한 탐지 시에 프로그램의 수행을 중단한다.

(1) 컴파일러 수준의 보호 대책

컴파일러 수준에서의 ROP 공격을 방어하는 방법은 프로그램 소스 코드를 요구하며, 소스 코드 분석을 통해 ROP 공격에 사용될 수 있는 연산들을 탐지하고 소스코드를 재작성하는 방법을 통해 이루어진다. 소스 코드 내에서의 ROP 공격은 RET opcode를 탐지하여 수행되며, 이때의 RET opcode는 0xc2, 0xc3, 0xca, 0xcb가 된다. 이를 통해 다음과 같은 방법으로 ROP 공격을 방어할 수 있다.〔8〕

- opcode에서 RET opcode를 발견할 경우, 인터럽트(INT3)를 발생시키거나 ROP 공격에

영향을 미치지 않는 명령어를 발생시킴으로써 원래의 프로그램 흐름을 변경한다.

- immediate 상수 값에서 RET opcode를 발견할 경우, 다음과 같이 명령어를 나누어서 사용한다.

ADD REG, IMM32 -〉 (ADD REG, IMM16), (ADD REG IMM16)

- ModR/M 바이트에서 〔그림 4〕와 같이 RET opcode가 발견될 경우, 다음과 같이 명령어를 교체함으로써 ROP 공격을 방어한다.

MOV EAX, EBX (0x8b 0xc3) -〉 PUSH EBX; POP EAX

ModR/M	Operand 1	Operand 2
0xc2	%eax, %al, %al	%edx, %dx, %dl
0xc3	%eax, %al, %al	%ebx, %bx, %bl
0xca	%ecx, %cx, %cl	%edx, %dx, %dl
0xcb	%eax, %cx, %cl	%ebx, %bx, %dl

SIB	Base	Scaled Index
0xc2	%edx	%eax*8
0xc3	%ebx	%eax*8
0xca	%edx	%ecx*8
0xcb	%ebx	%ecx*8

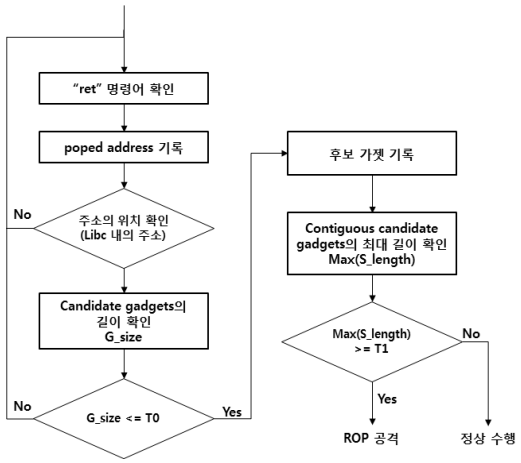
〔그림 4〕 ModR/M, SIB를 통해 추출 가능한 RET opcode

(2) 바이너리 수준의 보호 대책

바이너리 수준에서의 ROP 보호 대책으로 ROP 공격에 대한 탐지 메커니즘은 Chen등에 의해 DROP라는 이름으로 최초로 제안되었다〔10〕. DROP은 ROP 공격기법의 특징을 통해 ROP 공격을 탐지하며, 이는 ROP 공격의 특성상 다수 RET 명령어가 발생한다는 사실에 근거하여 공격 여부를 탐지하고자 하는 것이다. 이를 위해서 먼저 다음과 같은 값들을 정의 한다.

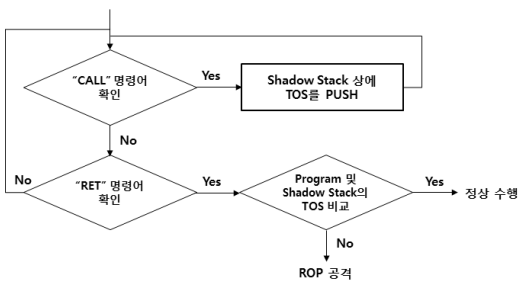
- G〔1,…,n〕 = 'ret' 명령어로 끝나는 명령어 시퀀스의 집합
- G〔i〕 = 집합 G 내의 i번째 가젯
- G_size = sizeof(G〔i〕) If Min_Addr 〈= G〔i〕.Addr 〈= Max_Addr&&G〔i〕∈G;
- S = {S〔i〕|S〔i〕.G_size, S〔i+1〕.G_size 〈= T0&&S〔i〕, S〔i+1〕∈G};
- S_length = {length|length = sizeof(S)};
- ROP = Assert(Max(S_length) 〉= T1);

이와 같이 정의된 값들을 통해 현재 수행되는 RET 명령어와 이후 수행될 RET 명령어 사이의 명령어들의 수를 계산하고, T0로 사전에 정의해 놓은 수 이하의 명령어가 나타날 경우와 T1으로 정의한 인접한 가젯에 대한 최대 수를 통해 ROP 공격을 판단하여 이를 탐지하는 기법이다. 이러한 기법은 〔그림 5〕와 같은 흐름을 통해 처리된다.



〔그림 5〕 DROP 실행 흐름

또한, Davi등은 ROPdefender라고 하는 ROP에 대한 방어 대책을 제안하였다〔11〕. 제안된 방법은 시스템 내에서 함수 호출이 일어날 경우에 반환 주소가 스택 상에 저장되는데, ROPdefender는 이를 이용하여 Shadow 스택이라는 가상의 스택을 구성하고, CALL 명령어 발생 이후 Shadow 스택에 반환

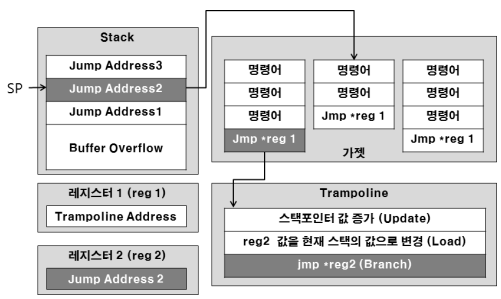


〔그림 6〕 ROPdefender 실행 흐름

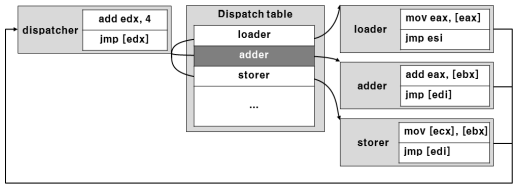
주소를 이중 저장하기 위해 TOS(Top of Stack)를 저장한다. 이후 프로그램이 RET 명령어 실행 시에 스택과 Shadow 스택의 TOS를 비교함으로써, 정상적인 호출에 의한 반환인지를 확인하는 방법으로 ROP 공격을 탐지하고 있다. 〔그림 6〕은 ROP-defender의 실행 흐름을 나타낸다.

III. Jump-Oriented Programming

ROP 공격의 방어 대책과 RET 시퀀스에 대한 탐지의 어려움을 극복하고자 새로운 공격 기법이 등장하였다. 새로운 공격 기법은 기존의 ROP 공격을 위해 구성되는 RET 시퀀스에 대한 탐지의 어려움을 해결하기 위해서 시스템 상에서 RET 시퀀스 보다 그 수가 많은 JMP 시퀀스를 사용하여 시퀀스 간의 제어 흐름을 변경하고 공격 연산을 수행한다. 따라서 RET 시퀀스가 가지는 특징을 이용한 기존의 ROP 보호 대책 또한 회피할 수 있게 된다. 새로운 공격 기법은 두 가지 형태로 각각 제안되었다. 먼저 Stephen등에 의해서 제안된 방법은 ROP 공격과 마찬가지로 스택 상에 시퀀스 주소를 삽입하고 SP를 이용하여 각 시퀀스에 대한 제어 흐름을 변경한다. 이 때 제어흐름을 변경하기 위해 기존의 RET 명령어 대신 JMP 명령어를 사용하여 명령어 시퀀스의 흐름을 이동하는 기법이다〔12〕. 흐름 이동의 방법은 Update-Load-Branch 기능을 수행하는 트램폴린(trampoline)을 통해서 이루어진다. 트램폴린은 Update 기능을 통해 SP를 증가하고, Load 기능으로 특정 레지스터에 SP가 참조하는 메모리 값을 저장한다. 마지막으로 Branch 기능을 통해서 특정 레지스터로 흐름을 변경하여 명령어 시퀀스들로 구성된 가젯을 실행하여 공격에 사용되는 연산들을 수행하고 있다. 〔그림 7〕은 트램폴린을 이용하는 JOP 공격의 전체적인 제어 흐름을 나타낸다.



〔그림 7〕 Update-Load-Branch를 이용한 JOP 공격 흐름



〔그림 8〕 dispatcher를 이용한 JOP 공격 흐름

반면 Bletsch등이 제안한 방법은 스택을 사용하지 않고 제어 흐름을 변경한다〔13〕. 따라서 스택 포인터에 대한 연산이 요구되지 않는 특성을 가진다.

이를 위해서 디스패처 테이블(dispatch table)을 구축하고 공격에 요구되는 연산들을 구성한다. 공격의 제어 흐름은 디스패처 (dispatcher)를 통해 이동되기 때문에 각 연산에 사용되는 시퀀스들은 연산 수행 후 디스패처로의 제어 흐름 이동이 요구된다. 제안된 공격 기법은 스택 포인터에 대한 사용이 요구되지 않기 때문에 기존 공격들에서 발생되었던 스택 포인터에 대한 변경이 발생하지 않는다. 따라서 스택 포인터의 변경에 대한 탐지를 통해 공격 유무를 판단하는 방법은 유효하지 않게 된다. 〔그림 8〕은 dispatcher를 이용하는 공격의 제어 흐름을 나타내고 있다.

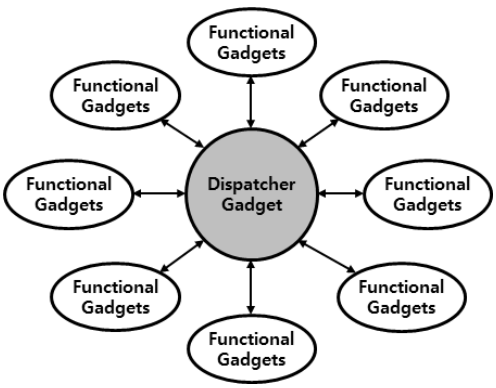
IV. 제안하는 바이너리 수준의 JOP 공격 탐지 메커니즘

JOP 공격에 대한 탐지 방법은 기존 ROP 공격에 탐지 방법과 동일하게 컴파일러 수준과 바이너리 수준에서의 탐지 방법이 적용될 수 있으나, 본 논문에서 제안하는 탐지 방법은 바이너리 수준에서의 탐지를 수행한다. 또한, 이전의 연구로서 ROP 공격에 대한 탐지 메커니즘들은 ROP 공격에서 발생하는 특징을 이용하는 탐지 방법이기 때문에 JOP 공격 기법에 대한 탐지 방법으로 적용될 수 없다. 따라서 JOP 공격 기법에 대한 탐지방법을 구현하기 위해서는 먼저 JOP 공격 기법이 가지는 특징을 분석하고 이에 대한 탐지 방법을 고려해야 한다. 제안하는 JOP 공격에 대한 탐지 방법을 구현하기 위해 다음과 같이 JOP 공격 기법의 특징을 분석하고, 이를 이용하여 탐지를 수행하는 탐지 메커니즘을 제안한다.

- 공격에 사용되는 연산이 JMP 시퀀스를 통해 구성된다.

- 각 시퀀스는 최대 4~5개의 명령어의 수를 포함한다.
- 연산 수행에 대한 제어 흐름은 디스패처(또는 트램폴린)를 이용하여 변경한다.
- 디스패처(또는 트램폴린)로의 제어 흐름 변경은 레지스터를 이용하여 간접적으로 이루어진다.

먼저, JOP 공격에 사용되는 명령어 시퀀스는 명령어 시퀀스 끝에 JMP 명령어를 포함하는 시퀀스로서 구성된다. 이러한 시퀀스들이 포함하는 각 명령어의 수는 공격자가 JMP 시퀀스를 결합하여 요구하는 연산을 수행하기 위해 최대 4~5개의 명령어를 포함하는 특징을 가진다. 또한, JOP 공격의 제어 흐름을 살펴보면 공격을 위한 연산에 사용되는 가젯들이 항상 디스패처(또는 트램폴린) 가젯에 의해 실행 흐름이 제어되는 특징을 가지고 있다. 따라서 제어 흐름을 단순히 그래프로 나타내면 〔그림 9〕와 같다.



〔그림 9〕 JOP 공격의 Control Flow Graph

마지막으로, JOP 공격은 디스패처(또는 트램폴린) 가젯으로의 제어 흐름 변경이 직접적인 주소 값이 아닌 레지스터를 이용하여 간접적으로 이루어진다는 특징을 가진다. 따라서 이러한 특성을 이용해서 제안하는 JOP 공격에 대한 탐지 방법은 먼저 다음과 같은 값들을 정의 한다.

- T0 : JOP 공격에서 사용되는 명령어 시퀀스에서의 최대 명령어 개수 정의
- T1 : JOP 공격에서 사용되는 명령어 시퀀스의 최소 개수 정의
- Gadget : 탐색된 가젯들의 배열
- J_count : JOP 가젯의 수를 측정하기 위한 변수

제안하는 탐지 방법은 다음과 같이 총 5단계를 걸쳐 JOP 공격에 대한 탐지를 수행한다.

① JOP 공격에 사용되는 JMP 시퀀스를 탐색을 위해서 각 명령어에 대해 JMP 명령어를 인식하고, 이때 JOP 공격이 레지스터를 이용하여 제어흐름 변경이 이루어진다는 특징을 이용하여 JMP 시퀀스가 레지스터를 이용한 간접 분기를 수행하는지 확인한다.

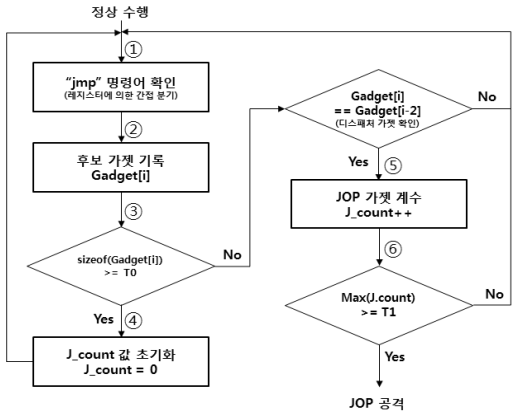
② 탐색된 명령어 시퀀스가 JOP 공격에 사용되는 디스패처 가젯인지를 판단하기 위해 일단 후보 가젯으로 정의하고 일련의 배열 내에 저장한다.

③ 또한 JOP 가젯 여부의 판단 전에 JMP 시퀀스들이 최대 4~5개의 명령어들로 이루어진다는 특징에 기반하여 가젯의 사이즈를 사전에 정의해 놓은 값(T0)과 비교함으로써 JOP 가젯의 후보군을 줄인다.

④ 만약 가젯의 사이즈가 T0보다 큰 경우, JMP 공격에 사용되는 가젯이 아니라고 판단한다. 이 때는 JOP 공격의 흐름이 아니기 때문에 JOP 가젯 수를 측정하는 J_count 값을 초기화하고, 정상적인 수행을 계속한다.

⑤ 현재의 후보 가젯이 실제 JOP에 사용되는 디스패처 가젯 인지를 확인한다. 따라서 현재 확인되는 후보 가젯(Gadget[i])과 이전에 기록된 후보 가젯(Gadget[i-2])이 동일한 디스패처 가젯인지를 확인하고, 아닐 경우는 정상 수행을 계속한다.

⑥ 디스패처 가젯으로 확인된 가젯은 완전한 JOP 가젯으로 판단하여 JOP 가젯의 수를 측정하는



〔그림 10〕 제안하는 탐지 기법의 실행 흐름

J_count 값을 증가시킨다.

⑦ J_count 값이 사전에 정의해 놓은 값과 비교하여 높을 경우 최종적으로 JOP 공격으로 판단하여 공격을 탐지하게 된다.

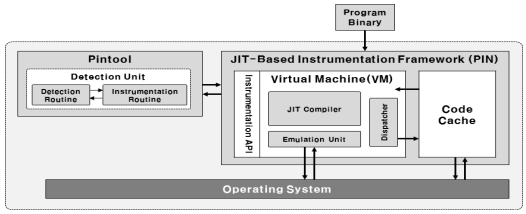
〔그림 10〕은 제안하는 JOP 공격에 대해 각 단계를 통한 탐지 방법의 전체적인 실행 흐름도이다.

V. 제안하는 JOP 공격 탐지 메커니즘의 구현 및 성능 평가

JOP 탐지 방법을 구현하기 위해서 Dynamic Binary Instrumentation(DBI) 툴을 이용하는 방법이 요구된다. 따라서 본 논문에서 제안하는 JOP 탐지 방법은 Intel에서 개발한 JIT 기반의 DBI 툴인 Pintool을 이용하여 구현하였으며〔14〕, 다음과 같은 환경에서 구현 및 성능평가를 수행하였다.

- CPU : Intel Core i5 2.8Ghz
- OS : Fedora 15
- Kernel : Linux 2.6.38.6-26

제안하는 방법의 실제 구현은 Pintool 상에서의 바이너리 측정을 통해 수행되는 명령어를 확인하고, 이에 따라 제안하는 방법을 적용하여 JOP 공격에 대한 탐지를 수행한다. 본 논문에서의 구현은 Linux 계



〔그림 11〕 PIN 바이너리 측정 프레임 워크를 통한 탐지 아키텍처

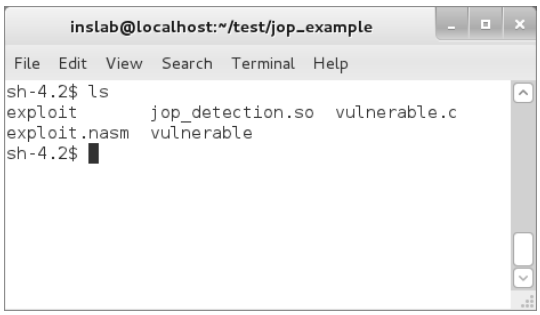
열에서 수행되었지만 이에 국한되지 않고 인텔 x86 명령어 셋을 사용하는 어떠한 운영체제에서 구현될 수 있다. 〔그림 11〕은 Pin 프레임워크 내에서의 제안한 탐지 아키텍처를 나타내고 있다.

구현된 탐지 방법을 통해 실제 JOP 공격에 대한 탐지 가능함을 검증하기 위해서 〔표 1〕과 같이 Buffer Overflow에 취약한 샘플 프로그램〔15〕을 통해 검증을 수행하였다. 이를 위해서 Bletsch 등의 논문에서 포함된 쉘 코드〔13〕를 이용하여 다음과 같이 작성된 샘플 프로그램을 통한 JOP 공격을 수행하고, 이에 대해 실제 탐지가 가능한지를 실험하였다. 공격 실험 시 이용된 JOP 가젯들은 샘플 프로그램 내에 삽입하여 공격을 수행하였으나, 실제 공격에서는 시스템 라이브러리 등에 존재하는 시퀀스를 이용하여 JOP 공격 코드를 작성할 수 있다. 실험결과는 〔그림 12〕와 같으며, 실제 JOP 공격에 대한 탐지가 이루어짐을 볼

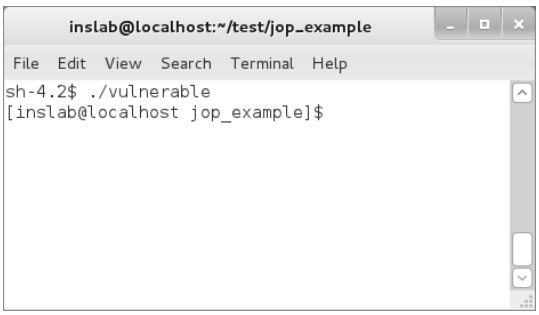
〔표 1〕 JOP 공격에 대한 샘플 프로그램

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <fcntl.h>
 5 #include <sys/stat.h>
 6 #include <sys/mman.h>
 7 #include <unistd.h>
 8
 9 char* executable="/bin//sh";
10 char* null="\0\0\0";
11 FILE * fd;
12
13 void attack_payload () {
14   asm(".intel_syntax noprefix");
15   //dispatcher
16   asm("add ebp,edi; jmp [ebp-0x39];");
17   //initializer
18   asm("popa; jmp [ebx-0x3e];");
19   //g00
20   asm("popa; fdivr st(1), st;
21        jmp [edx];");
22   //g01
23   asm("inc eax; fdivr st(1), st;
24        jmp [edx];");
25   //g02
26   asm("mov [ebx-0x17bc0000], ah; stc;
27        jmp [edx];");
28   //g03
29   asm("inc ebx; fdivr st(1), st;
30        jmp [edx];");
31   //g07
32   asm("popa; cmc;
33        jmp dword ptr [ecx];");
34   //g08
35   asm("xchg ecx, eax; fdiv st, st(3);
36        jmp [esi-0xf];");
37   //g09
38   asm("mov eax, [esi+0xc];
39        mov [esp], eax; call [esi+0x4];");
40   //g0a
41   asm("int 0x80");
42
43   asm(".att_syntax noprefix");
44 }
45
46 void overflow() {
47    char buf[256];
48    fscanf(fd,"%[^\n]",buf);
49    return;
50 }
51
52 int main(int argc, char** argv) {
53    if(argc>1) filename = argv[1];
54    fd=fopen(filename, "r");
55    overflow();
56 }
```

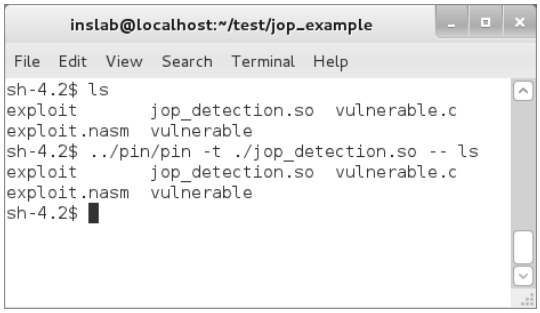
(a) 정상적인 프로그램 수행



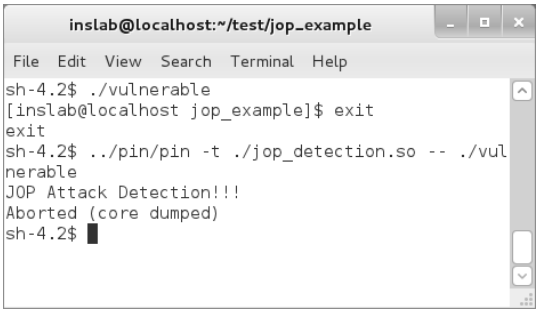
(b) JOP 공격 수행



(c) 정상적인 프로그램에 대한 탐지 메커니즘 적용



(d) JOP 공격에 대한 탐지 매커니즘 적용

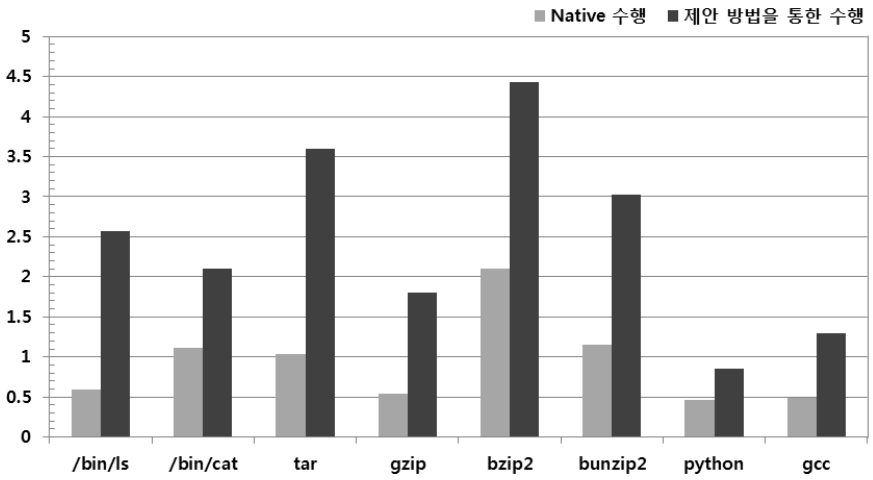


〔그림 12〕 JOP 공격 탐지에 대한 실험 결과

수 있다. 또한 정상적인 프로그램에 대해서는 정상적인 수행이 이루어짐을 함께 볼 수 있다.

구현된 탐지 방법의 성능 평가를 위해서 Linux 시스템 상에서 많이 사용되는 8개의 애플리케이션을 임의로 선정하고, 각각 애플리케이션에 대해 기존의 바이너리 수행시간과 탐지가 적용된 환경에서의 수행 시간을 측정하였다. 측정 결과, 애플리케이션 수행 시에 DBI로 인해 발생하는 오버헤드 및 탐지 루틴의 적용으로 인해 기존의 실행시간 대비 평균 약 2.7배의 오버헤드를 발생시켰다. 이러한 측정 결과는 〔그림 13〕과 같다.

제안하는 방법은 JOP 공격에 대한 실시간 탐지를



〔그림 13〕 제안하는 방법에 대한 성능 평가

목표로 하여 동적 바이너리 측정 툴을 이용한 바이너리 수준에서의 탐지를 수행하기 때문에 동적 바이너리 측정에 대한 오버헤드는 불가피하다. 또한, 이러한 오버헤드에 대해서 현재까지 JOP 공격에 대한 바이너리 수준에서의 탐지방법에 관한 연구가 없기 때문에 제안하는 탐지 방법에 대한 성능을 비교 분석하기에는 어려움이 따른다.

VI. 결 론

시스템 상에서의 메모리 관련 공격은 공격자가 쉘코드라 일컫는 임의의 코드를 실행시킬 수 있다는 점에서 대단히 위협적이다. 따라서 운영체제는 이러한 공격에 대응하는 보호 메커니즘을 탑재하고 있지만, 그에 더불어 공격 기법 또한 나날이 발전하고 있다. 특히 Return-Oriented Programming 공격 기법은 공격자가 공격에 요구되는 연산을 직접 프로그래밍하여 최종적으로 쉘코드를 수행하는 공격 연산을 수행할 수 있다는 점에서 매우 효과적인 공격 기법이다. 하지만 RET 시퀀스에 대한 탐지의 어려움과 ROP 공격이 가지는 특징을 이용한 다양한 보호 대책들의 제안으로 인해 RET 시퀀스 대신 JMP 시퀀스를 사용하는 Jump-Oriented Programming 공격 기법이 제안되었다. JOP 공격 기법은 ROP 공격의 특성을 이용하는 기존의 ROP 탐지 메커니즘들을 무력화시킬 수 있기 때문에 JOP 공격에 새로운 탐지 메커니즘이 요구된다. 따라서 본 논문에서는 메모리 관련 공격 기법을 이용한 시스템 위협 중 메모리 관련 공격 기법 중 최근 소개된 Jump-Oriented Programming에 대해 살펴보고, 공격에 대한 특징을 이용한 공격 탐지 메커니즘을 제안하였다. 또한 탐지 메커니즘에 대한 실제 구현 및 성능 평가를 수행하였다. 제안하는 방법이 바이너리 수준에서의 공격 탐지이기 때문에 기존 프로그램 수행 시간 대비 평균 2.7배의 오버헤드가 발생하였다. 따라서 이러한 오버헤드를 줄이면서도 메모리 관련 공격에 대해 포괄적인 보호 방안에 대한 연구가 더욱 필요할 것으로 보인다.

참고문헌

〔1〕 Aleph. One, "Smashing The Stack For Fun And Profit," Phrack49, 1996.

〔2〕 Pax Project, "address space layout randomization" http://pax.grsecurity.net/docs/aslr.txt, 2003.

〔3〕 Microsoft TechNet, "데이터 실행 방지, "http://technet.microsoft.com/ko-kr/library/cc738483(WS.10).aspx.

〔4〕 C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. "StackGuard:Automatic adaptive detection and prevention of bufferoverflow attacks," Proceedings of the 7th USENIX Security Conference, pp. 63-78, Jan 1998.

〔5〕 Arjan van de Ven, ""New security Enhancements in Red Hat Enterprise Linux v.3, update 3"," Red Hat, 2004.

〔6〕 Nergal, "The advanced return-into-lib(c) exploits (Pax case study), "http://www.phrack.org/issues.html?issue=58&id=4&mode=txt, Dec 2001.

〔7〕 H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)," Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552-561, Oct 2007.

〔8〕 Piotr Bania, "Security Mitigations for Return-Oriented Programming Attacks," http://piotrbania.com/all/articles/pbania_rop_mitigations2010.pdf, 2010.

〔9〕 Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda, "G-Free : defeating return-oriented programming through gadget-less binaries," Proceedings of the ACSAC'10, Annual Computer Security Applications Conference, pp. 49-58, Dec 2010.

〔10〕 P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," Proceedings of the 5th International Conference on Information Systems Security, LNCS 5905, pp. 163-177, 2009.

〔11〕 Lucas Davi, Ahmad-Reza Sadeghi and Marcel Winandy, "ROPdefender: A Detection Tool to Defend Against

Return-Oriented Programming Attacks," HGI-TR-2010-001, Ruhr University Bochum, 2010.
〔12〕 S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," Proceedings of the ACM Conference on Computer and Communications Security, pp. 559-572, 2010.
〔13〕 T. Bletsch, X. Jiang and V. Freeh, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," TR-2010-8, North Carolina State University, 2010.
〔14〕 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, GeoLowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190-120, 2005.
〔15〕 Mehmet Kayaalp, "Example Jump-Oriented Programming Attack," http://cs.binghamton.edu/~mkayaalp/jop.html, 2012.

〈著者紹介〉

김 주 혁 (Ju-Hyuk Kim) 학생회원
2011년 2월: 호서대학교 정보보호학과 졸업
2011년 3월~현재: 호서대학교 정보보호학과 석사과정
〈관심분야〉 정보보호, 메모리 보안, 모바일 보안 등

이 요 람 (Yo-Ram Lee) 학생회원
2011년 2월: 호서대학교 정보보호학과 졸업
2011년 3월~현재: 호서대학교 정보보호학과 석사과정
〈관심분야〉 정보보호, NFC 보안, 네트워크 프로토콜 등

오 수 현 (Soo-Hyun Oh) 종신회원
1998년 2월: 성균관대학교 정보공학과 졸업
2000년 2월: 성균관대학교 전기전자 및 컴퓨터공학과 석사(공학석사)
2003년 8월: 성균관대학교 전기전자 및 컴퓨터공학과 박사(공학박사)
2004년 3월~현재: 호서대학교 정보보호학과 교수
〈관심분야〉 암호 프로토콜, 네트워크 보안, 정보보호제품 평가 및 인증 등