Journal of the Korea Academia-Industrial cooperation Society
Vol. 15, No. 5 pp. 3121-3131, 2014

http://dx.doi.org/10.5762/KAIS.2014.15.5.3121
ISSN 1975-4701 / eISSN 2288-4688

스택 영역에서의 코드 재사용 공격 탐지 메커니즘

김주혁[1], 오수현[2*]
[1]한국지역정보개발원, [2]호서대학교 정보보호학과

Detection Mechanism against Code Re-use Attack in Stack region

Ju-Hyuk Kim[1] and Soo-Hyun Oh[2*]
[1]**Korea Local Information Research & Development Institute**
[2]**Division of Information Security, Hoseo University**

요 약 메모리 관련 취약점은 컴퓨터 시스템 상에서의 가장 위협적인 공격이며 메모리 취약점을 이용한 실제 공격의 또한 증가하고 있다. 따라서 다양한 메모리 보호 메커니즘이 연구되고 운영체제 상에서 구현되었지만, 보호 시스템들을 우회하기 위한 새로운 공격 기법들이 함께 발전하고 있다. 특히, 메모리 관련 공격 기법 중 버퍼 오버플로우 공격은 코드 재사용 공격이라 불리는 Return-Oriented Programming(ROP), Jump-Oriented Programming(JOP) 등으로 발전하여 운영체제가 포함하는 메모리 보호 메커니즘을 우회하고 있다. 본 논문에서는 코드 재사용 공격 기법의 특징을 분석하고, 분석된 결과를 이용하여 바이너리 수준에서의 코드 재사용 공격을 탐지할 수 있는 메커니즘을 제안하며, 실험을 통해 제안하는 메커니즘이 코드 재사용 공격을 효율적으로 탐지할 수 있음을 증명한다.

Abstract Vulnerabilities related to memory have been known as major threats to the security of a computer system. Actually, the number of attacks using memory vulnerability has been increased. Accordingly, various memory protection mechanisms have been studied and implemented on operating system while new attack techniques bypassing the protection systems have been developed. Especially, buffer overflow attacks have been developed as Return-Oriented Programing(ROP) and Jump-Oriented Programming(JOP) called Code Re-used attack to bypass the memory protection mechanism. Thus, in this paper, I analyzed code re-use attack techniques emerged recently among attacks related to memory, as well as analyzed various detection mechanisms proposed previously. Based on the results of the analyses, a mechanism that could detect various code re-use attacks on a binary level was proposed. In addition, it was verified through experiments that the proposed mechanism could detect code re-use attacks effectively.

Key Words : Return-Oriented Programming, Jump-Oriented Programming, Code Re-use attack

1. 서론

최근 들어 컴퓨터 시스템 상에는 시스템 또는 애플리케이션 개발 중 내포되는 취약점을 악용하여 시스템에 대한 직접적인 해킹을 도모하는 공격이 증가하고 있다. 따라서 미국의 국토안보부 산하 컴퓨터 긴급 대응팀 US-CERT에서는 다양한 취약점을 조사하여 이를 공개하고 있으며, 이러한 취약점들 중에서 상당수가 메모리 관련 취약점으로 나타나고 있다. 메모리 관련 취약점은 공격자가 반환 주소(return address)와 같은 민감한 메모리 영역을 침범하는 공격 기법인 버퍼 오버플로우(buffer overflow) 공격 이나 포맷 스트링(format string)과 같이 메모리 공격을 통하여 공격자가 설치한 임의의 프로그램을 실행할 수 있다는 점에서 매우 위험한 취약점이다 [1].

"이 논문은 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임" (No. 2012R1A1A3007176)
*Corresponding Author : Soo-Hyun Oh(Hoseo Univ.)
Tel: +82-41-540-5716 email: shoh@hoseo.edu
Received January 13, 2014 Revised (1st February 27, 2014, 2nd March 24, 2014) Accepted May 8, 2014

따라서 이러한 취약점들이 공격자에 의해 악용되는 경우, 악성코드 및 루트킷과 같은 프로그램을 실행할 수 있고, 이를 통해 공격자는 사용자 개인 정보 및 시스템 제어권 등을 탈취할 수 있다.

운영체제는 이러한 취약점들로부터 사용자를 보호하기 위하여 메모리 주소 공간 재배치(ASLR, Address Space Layout Randomization) 및 데이터 실행방지(DEP, Data Execution Prevention) 등의 보호 메커니즘을 포함하고 있지만, 메커니즘의 발전과 더불어 공격자들 역시 메모리 관련 공격 기법을 더욱 발전시키고 있다. 이러한 공격 기법의 발전은 각각의 메모리 영역에서 나타난다. 특히, 스택 영역에서의 발전된 공격 기법에는 DEP 메커니즘을 우회하기 위한 방법으로, 시스템 내의 실행 권한이 있는 코드를 재사용하는 ROP(Return-Oriented Programming) 공격 기법이 있다. 또한, 최근에는 ROP 공격과 유사한 형태인 JOP(Jump- Oriented Programming) 공격 기법이 등장하여 기존 ROP 공격에 대한 대응 방안을 회피하고 있다. 최근에 제안된 공격 기법들은 시스템 내의 코드를 재사용하여 공격 코드를 프로그래밍하기 때문에 코드 재사용 공격이라 일컫고 있다. 코드 재사용 공격에 관한 기존의 연구들은 ROP 및 JOP 공격의 특징을 이용하여 각각의 공격만을 탐지할 수 있는 대응 방안을 제안하고 있으므로, 다양한 공격 기법에 적용할 수 있는 대응 방안이 요구된다.

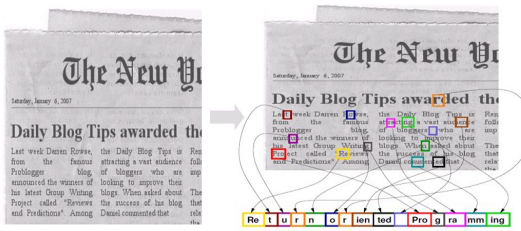
따라서 본 논문에서는 코드 재사용 공격에 대하여 지금까지 제안된 모든 공격을 탐지할 수 있는 대응 방안을 제안하고자 한다. 본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 공격의 대상이 되는 프로세스 메모리의 구조를 살펴보고, 프로세스 메모리 내에서의 취약점을 이용한 버퍼 오버플로우 공격에 대해 기술한다. 3장에서는 스택영역에서의 코드 재사용 공격인 ROP와 JOP에 대해 분석하고, 이에 대한 기존의 대응 방안들을 살펴본다. 4장에서는 스택 영역에서의 다양한 코드 재사용 공격을 탐지할 수 있는 메커니즘을 제안하고, 이에 대한 성능 분석을 수행하며, 마지막으로 5장에서는 논문의 결론을 맺는다.

2. 관련 연구

2.1 Return-Oriented Programming

스택, 힙 영역등과 같이 실행 권한이 있는 메모리 영역에서의 데이터 실행을 방지하기 위한 메커니즘인 W^X(Write XOR Execute), DEP 등이 운영체제에 탑재되면서 버퍼 오버플로우 공격을 이용한 메모리 영역에서의 쉘 코드 수행이 방지되었다[2]. 하지만 이를 우회하기 위한 공격 기법으로 RTL(Return to Library)이 등장하였다[3]. RTL공격 기법은 데이터 실행 방지 메커니즘으로 인해 실행 권한이 없는 스택 영역 대신 실행 권한이 있는 시스템 라이브러리 영역의 함수를 호출하여 쉘 들을 실행하는 공격기법이다. 하지만 RTL 공격 기법은 시스템 라이브러리로 등록된 함수만을 이용할 수 있기 때문에 공격자가 원하는 공격을 수행하는데 제약을 가진다. 따라서 이러한 제약을 해결하기 위한 공격 기법으로 RTL 공격을 응용한 ROP 공격 기법이 제안되었다[4].

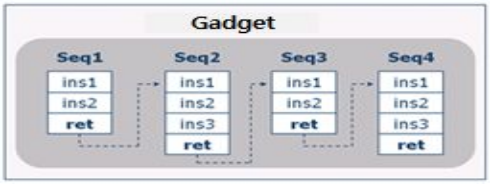
ROP 공격은 공격자가 메모리 상에 임의의 코드를 삽입하지 않고 시스템 라이브러리를 이용하여 공격자가 원하는 연산을 수행하여 최종적으로 공격 코드를 실행시킬 수 있도록 프로그래밍 하는 공격 기법으로 메모리 내의 코드를 재사용한다고 해서 코드 재사용 공격으로 불린다. 공격자가 시스템 내의 코드를 재사용하여 공격 코드를 프로그래밍하고 수행할 수 있다는 점에서 ROP 공격 기법은 튜링 기계로 해결할 수 있는 문제를 프로그래밍 언어나 추상기계를 통해 해결할 수 있음을 나타내는 튜링 완전성(Turing Completeness)을 가진다고 할 수 있다. Fig. 1은 ROP 공격의 개념을 나타낸다.



[Fig. 1] ROP attack

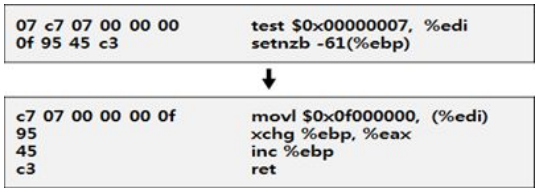
ROP 공격을 수행하기 위해서는 먼저 임의의 연산을 수행하는 가젯(gadget)을 구성하여야 하며, 이를 위해서는 시스템 상에 존재하는 기계어 코드 섹션에 대한 검색이 요구된다. ROP 공격을 구성하는 기계어 코드 간의 흐름 이동은 SP(Stack Pointer)를 통해 이루어지기 때문에 가젯 구성에 요구되는 각 기계어 코드는 스택 포인터를 증가시키는 RET 명령어로 끝나는 명령어 시퀀스로 존재

해야 한다. 이렇게 구성된 시퀀스들을 이용하여 ROP 공격을 수행하기 위해서는 먼저 운영체제 및 애플리케이션의 취약점을 이용하여 ROP를 구성하는 명령어 시퀀스로 실행 흐름을 변경해야 한다. 이후, 변경된 실행 흐름을 통해 각 시퀀스 내의 명령어가 수행되면서 마지막 RET 명령어로 인해 스택 포인터를 변경하기 때문에 공격자의 의도에 따라 시퀀스들을 이용하여 구성한 가젯들이 수행될 수 있고, 이를 통해 최종적으로 공격자가 실행하고자 하는 쉘 코드를 수행할 수 있게 된다. Fig. 2는 ROP 공격을 위한 가젯의 구성과 제어의 흐름을 나타낸다.



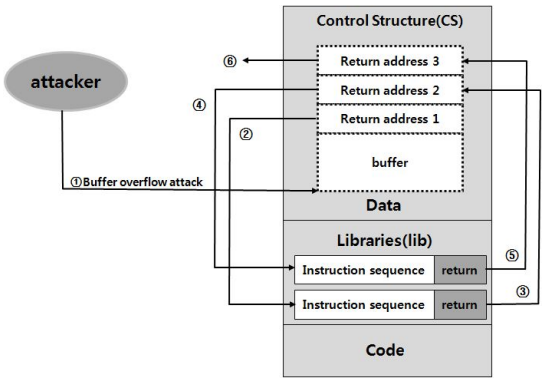
[Fig. 2] ROP sequence flow

ROP 공격을 수행하기 위해 사용되는 RET 시퀀스는 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 Fig. 3과 같이 기존의 명령어에 대한 바이트 오프셋 변경을 통해서 RET 시퀀스를 구성하여 사용할 수 있다.



[Fig. 3] ROP sequence search

이렇게 검색된 시퀀스들은 스택 상에 시퀀스 주소를 삽입함으로써 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 이용하여 제어 흐름을 변경하는 것이다. 각 명령어 시퀀스들은 명령어가 수행되면서 마지막에 존재하는 RET 명령어를 통해 스택 상의 다음 시퀀스를 수행하도록 SP를 변경하기 때문에 공격자의 의도에 따라 시퀀스들을 이용하여 구성된 가젯들이 수행될 수 있고, 이러한 가젯들을 통해 최종적으로 공격자가 실행하고자 하는 쉘 코드를 수행할 수 있게 된다. Fig. 4는 ROP 공격에 대한 전체적인 제어 흐름을 나타낸다.



[Fig. 4] The control flow of ROP attack

이러한 ROP 공격은 데이터 실행 방지 메커니즘인 DEP를 우회할 수 있으나 각 메모리 영역을 랜덤화하는 Full ASLR 메커니즘[5]에 대해서는 우회가 불가능하다. 하지만 Full ASLR이 적용된 시스템 환경에서도 부분적으로 ASLR이 적용되지 않는 취약점을 악용하여 공격이 성공할 수 있기 때문에, 이를 이용하여 ROP 공격이 수행될 수 있다. 대표적인 사례로서, Mac OSX 운영체제에서는 시스템 라이브러리에 대한 코드 재사용 공격을 막기 위해 라이브러리 영역에 대한 ASLR 메커니즘을 탑재하고 있지만, 동적 링커인 dyld에서 고정적인 주소의 시스템 라이브러리를 사용함으로써 ASLR 메커니즘이 무력화되는 현상이 발생되었다[6]. 이와 같이 보호 메커니즘들에 대한 회피 가능성이 존재하기 때문에 ROP 공격은 강력한 시스템 공격 수단이라 할 수 있다.

최근 들어, ROP 기법에 대해 컴파일러 수준과 바이너리 수준에서의 대응 방안들이 제안되고 있다. 컴파일러 수준에서의 대응 방안은 프로그램 소스 코드 분석을 통해 수행되며, ROP 공격에 사용될 수 있는 연산들을 탐지하거나 이를 방지하는 방법이다[7]. 반면 바이너리 수준에서의 대응방안은 프로그램이 실시간 수행 중에 ROP 공격이 발생하는 경우에 이를 탐지하는 방법으로, 바이너리 환경에서의 제안된 대표적인 대응 방안들을 살펴보

면 다음과 같다.

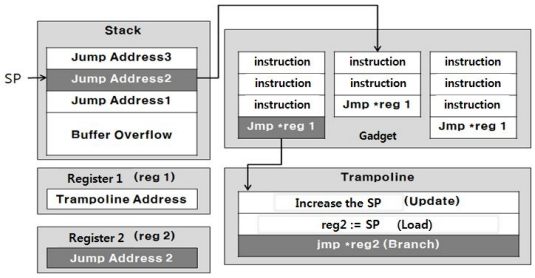
먼저, Chen 등에 의해 제안된 최초 ROP 대응 방안인 DROP은 ROP 공격의 특성상 다수의 RET 명령어가 발생한다는 사실에 근거하여 공격 여부를 탐지한다. 따라서 현재 수행되는 RET 명령어와 이후 수행될 RET 명령어 사이의 명령어들의 수를 계산하고, 사전에 정의해놓은 수 이하의 명령어가 실행되는 가젯을 탐지하는 경우에 이전 탐지된 가젯의 최대 수를 이용하여 ROP 공격을 판단하여 탐지하는 기법이다[8].

DROP은 RET 명령어만을 탐지하기 때문에 중간에 JMP 명령어를 통한 분기가 발생하는 경우에는 공격에 대한 탐지가 불가능한 상황이 발생할 수 있다. 또한, Davi 등은 ROP defender이라고 하는 대응 방안을 제안하였다[9]. 제안된 방법은 시스템 내에서 함수 호출이 일어날 경우에 반환 주소가 스택 상에 저장되는데, ROP defender는 이를 이용하여 Shadow 스택이라는 가상의 스택을 구성하고, CALL 명령어 발생 이후 Shadow 스택에 반환 주소를 이중 저장한다. 이후 프로그램이 RET 명령어 실행 시에 RET 주소와 Shadow 스택에 저장된 RET 주소를 비교하여 정상적인 호출에 의한 반환인지를 확인하는 방법으로 ROP 공격을 탐지하고 있다. 그러므로 ROP defender는 CALL-RET 간의 정확한 match가 ROP 공격 탐지를 위한 가장 중요한 요소가 된다.

따라서 탐지 메커니즘 적용 중에 CALL-RET 간의 mismatch가 발생하는 경우, 정상적인 실행을 공격에 의한 실행으로 판단할 수 있는 가능성이 존재한다.

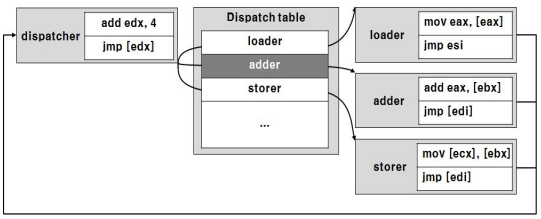
2.2 Jump-Oriented Programming

ROP 공격의 방어 대책과 RET 시퀀스에 대한 탐지의 어려움을 극복하고자 새로운 공격 기법이 등장하였다. 새로운 공격 기법은 시스템 상에서 RET 시퀀스 보다 그 수가 많은 JMP 시퀀스를 사용하여 시퀀스 간의 제어 흐름을 변경하고 이를 이용하여 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 Stephen 등에 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[10]. 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. [Fig. 5]는 트램폴린을 이용하는 JOP 공격의 전체적인 제어 흐름을 나타낸다.



[Fig. 5] The control flow of JOP attack

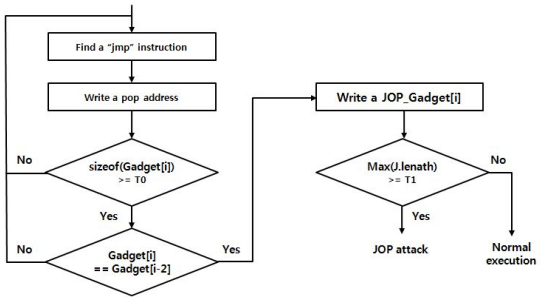
반면 Bletsch 등이 제안한 방법은 스택 포인터를 사용하지 않고 제어 흐름을 변경한다[11]. 따라서 스택 포인터에 대한 연산이 요구되지 않는 특성을 가진다. 이를 위해서 디스패처 테이블(Dispatch table)을 구축하고 공격에 요구되는 연산들을 구성한다. 공격의 제어 흐름은 디스패처(Dispatcher)를 통해 이동되기 때문에 각 연산에 사용되는 시퀀스들은 연산 수행 후 디스패처로의 제어 흐름 이동이 요구된다. 이 공격 기법은 스택 포인터를 사용하지 않기 때문에 기존 공격들에서 발생하였던 스택 포인터에 대한 변경이 발생하지 않는다. 따라서 스택 포인터의 변경에 대한 탐지를 통해 공격 유무를 판단하는 방법은 유효하지 않게 된다. Fig. 6은 디스패처를 이용하는 공격의 제어 흐름을 나타내고 있다.



[Fig. 6] The attack using dispatcher

JOP 공격에 대한 탐지 메커니즘은 Kim 등에 의해 제안된 방법으로 JOP 공격 기법이 가지는 특징을 이용하여 공격 특징을 탐지한다[12]. 이 방법에서는 먼저 JMP 시퀀스를 탐지하고, JOP 공격을 위해 사용되는 가젯들이 항상 디스패처(또는 트램폴린) 가젯에 의해 실행 흐름이 제어되는 특징을 탐지함으로써 JOP 공격 여부를 판단하는 방법이다. Fig. 7은 JOP 공격탐지 메커니즘의 실행 흐름을 나타낸다.

이러한 JOP 탐지 메커니즘은 JOP 공격에 국한되어 탐지를 수행하기 때문에 JOP 공격 중 RET 시퀀스가 발생하는 경우, 공격에 대한 탐지가 불가능할 수 있다.



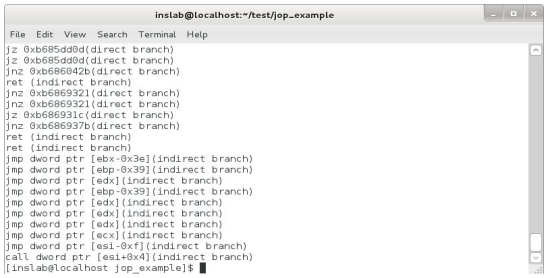
[Fig. 7] JOP attack detection mechanism

3. 제안하는 코드 재사용 공격 탐지메커니즘

기존의 코드 재사용 공격에 대한 대응 방안들은 ROP 및 JOP 공격 각각에 국한되어 설계되었기 때문에 각 공격에서 사용하는 시퀀스가 혼합된 공격 및 서로 다른 공격에 대해서는 탐지가 불가능하다는 취약점을 갖는다. 따라서 두 가지 공격기법들을 모두 탐지 할 수 있는 새로운 대응 방안이 요구된다.

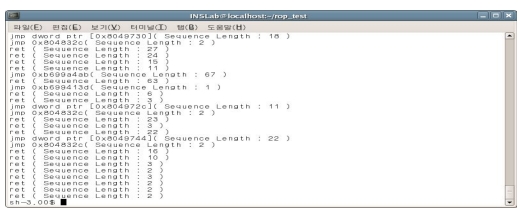
본 논문에서 제안하는 코드 재사용 공격에 대한 대응 방안은 ROP 및 JOP 공격 각각에 국한되지 않고, 각 공격들의 공통적인 특징을 통해 공격에 대한 수행 여부를 탐지한다. 따라서 제안하는 탐지 메커니즘은 ROP 및 JOP 공격과 같은 코드 재사용 공격 기법의 특징 중 다음과 같은 공통의 특징을 이용한다.

(1) 코드 재사용 공격 기법은 코드 간의 실행 흐름을 변경하기 위해 RET 및 JMP 명령어를 이용한다. 이때 JMP 명령어는 절대 주소를 통한 직접 분기 명령어로 나타날 수 없기 때문에 레지스터에 의한 간접 분기인지를 확인한다. Fig. 8은 JOP 공격 시 분기 형태에 대한 실험 화면을 나타낸다.

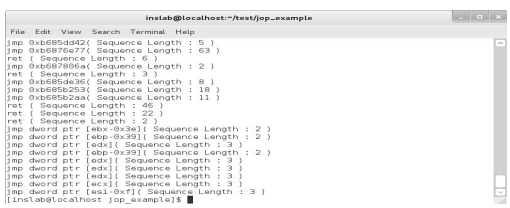


[Fig. 8] Branch pattern in JOP attack

(2) 공격에 사용되는 각 공격 시퀀스는 공격자가 원하는 대로 프로그래밍 할 수 있어야 하기 때문에 짧은 명령어의 수를 포함하는 시퀀스로 나타난다. 따라서 하나의 시퀀스가 포함하는 최대 명령어의 수를 MAX_INS로 정의하고, 시퀀스 간의 명령어의 수가 MAX_INS 이하인지를 확인한다. Fig. 9는 JOP 공격 시에 시퀀스간의 명령어 수에 대한 실험 화면을 나타낸다.



(a) ROP attack

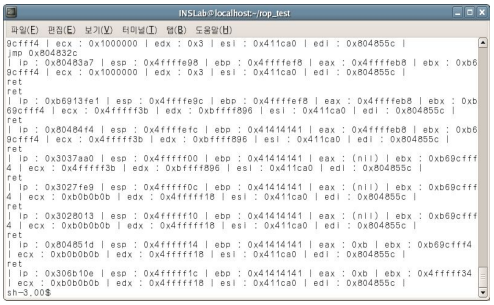


(b) JOP attack

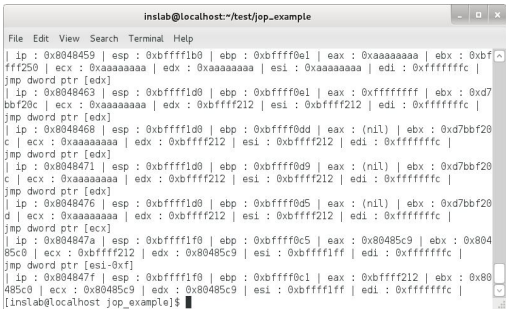
[Fig. 9] The number of instruction in sequences

(3) 코드 간의 실행 흐름 변경으로 인해 ESP를 포함하는 GP 레지스터의 작은 변화가 발생한다. 보통은 공격

코드 간 흐름 변경을 위해 0x4 정도의 차이를 일정하게 발생한다. 하지만 POP 및 POPAD와 같은 명령어를 통해 0x4 이상의 차이가 발생할 수도 있다. 따라서 이러한 점을 고려하여 최대 레지스터 변경 값을 MAX_REG로서 정의하고, 정의된 값 내에서의 값 변경이 발생하는 지를 확인한다. Fig. 10은 코드 재사용 공격 시에 발생되는 레지스터 값 변경에 대한 실험 화면을 나타낸다.



(a) ROP attack



(b) JOP attack

[Fig. 10] The experiment of register change

이러한 공통의 특징을 기반으로 본 논문에서는 Fig. 11과 같은 형태의 큐를 이용한 메커니즘을 제안한다. 이러한 큐의 길이는 탐지하고자 하는 코드 재사용 공격에 대해 탐지고하자 하는 시퀀스의 길이에 따라 전체 n의 길이를 가지며, 프로그램 수행 중에 RET 및 JMP 명령어를 포함하는 분기 명령어 발생 시에 element를 삽입한다. 이때 삽입되는 element는 다음과 같다.

- Instruction : RET 및 JMP 명령어를 포함하는 분기 명령어
- REG[i] : 분기 명령이 레지스터에 의한 분기인지 확인한다.(단, RET 명령의 경우는 레지스터에 의한 분기로 표기)
- SEQ_Length[i] : 이전 element 사이의 명령어의 수가 사전에 정의된 MAX_INS 보다 이하인지 확인한다.
- REG_Modify[i] : ESP를 포함하는 GP 레지스터의 값의 변동 폭이 사전에 정의된 MAX_REG 이하인지 확인한다.
- S[i] = REG[i] & REG_Modify[i] & SEQ_Length[i]

Instruction	JMP	CALL	RET	RET	JMP
REG	%eax	...	Null	Null	%ebx
SEQ_Length = number of sequences < MAX_INS	1	0	1	0	1
REG_Modify = number of changed registers< MAX_REG	1	0	0	1	1
$S = REG\&SEQ\_Length\ \&\ REG\_Modify$	1	0	0	0	1

[Fig. 11] Queue in the proposed mechanism

이와 같이 큐에 element가 삽입된 이후에 코드 재사용 공격 탐지를 수행한다. 공격 여부에 대한 탐지는 큐의 모든 element들이 위에서 정의한 코드 재사용 공격의 특징을 만족하는지를 확인함으로써 코드 재사용 공격에 대한 탐지를 수행한다. 따라서 전체 S에 대해 비트열로 나열하여 S = 2n – 1인 경우에 코드 재사용 공격으로 간주한다.

4. 구현 및 성능 분석

바이너리 환경에서 코드 재사용 공격을 탐지하는 메커니즘을 구현하기 위해서 Dynamic Binary Instrumentation(DBI) 툴을 이용하는 방법이 요구된다. 따라서 본 논문에서 제안하는 탐지 메커니즘을 구현하기 위해 Intel에서 개발한 JIT 기반의 DBI 툴인 Pintool을 이용하여 구현하였으며[13], Pintool 상에서의 바이너리 측정을 통해 수행되는 명령어를 확인하고, 이에 따라 제안하는 메커니즘을 적용하여 스택 영역에서의 코드 재사용 공격에 대한 탐지를 수행하였다. 본 논문에서의 구현은 Linux 계열의 운영체제에서 수행되었지만 제안하는 방법은 이에 국한되지 않고 인텔 x86 명령어 셋을 사용하는 어떠한 운영체제에서도 구현될 수 있다. [Fig. 13]은 Pin 프레임 워크 내에서의 제안하는 탐지 메커니즘의 아키텍처를 나타내고 있다[12].

4.1 성능 분석

4.1.1 ROP 공격 탐지 실험

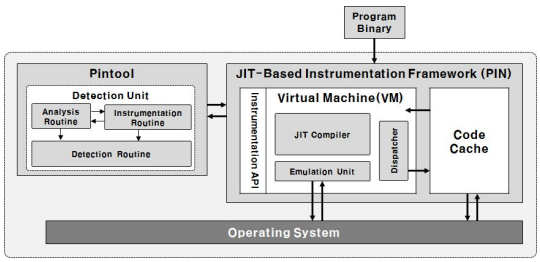
구현된 탐지 매커니즘을 이용하여 실제 코드 재사용 공격이 탐지 가능함을 검증하기 위해서, Table 1과 같은

ROP 공격에 취약한 샘플 프로그램을 이용하여 검증을 수행하였다. 검증을 수행한 환경은 다음과 같다.

- CPU : Intel Core i5 2.8GHz
- OS : Fedora 4
- Kernel : Linux 2.6.11-1
- System Library : Libc 2.3.5

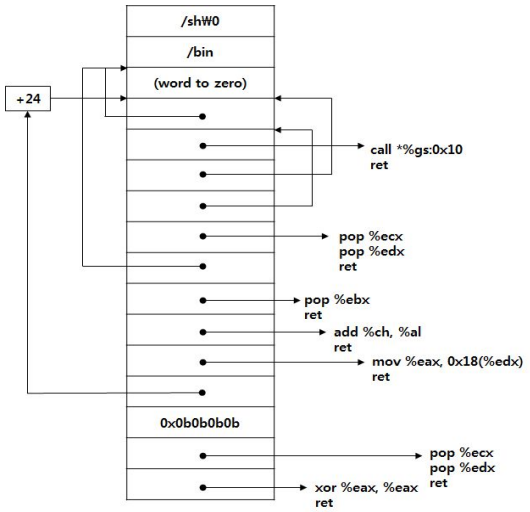
[Table 1] The sample code of ROP attack

```
 1 #include <sys/types.h>
 2 #include <sys/stat.h>
 3 #include <sys/mman.h>
 4 #include <fcntl.h>
 5
 6 void do_map_libc(void) {
 7  int fd;
 8  struct stat sb;
 9  fd = open("libc-2.3.5.so", O_RDONLY, 0);
10  fstat(fd, &sb);
11  mmap((void *)0x03000000, sb.st_size,
12  PROT_READ|PROT_EXEC, MAP_FIXED|MAP_SHARED, fd, 0);
13 }
14 void do_map_stack(void) {
15  int fd;
16  fd = open("/dev/zero", O_RDONLY, 0)
17 mmap(0x04f000000, 0x001000000, PROT_READ|PROT_WRITE,
18  MAP_FIXED|MAP_PRIVATE, fd, 0);
19 }
20 void overflow(char *arg) {
21  char buf[64];
22  strcpy(buf, arg);
23 }
24 void move_stack(char *arg) {
25  __asm("mov $0x04fffff00, %esp\n");
26  overflow(arg);
27  _exit(0);
28 }
29 int main(int argc, char *argv[]) {
30  do_map_libc(); do_map_stack();
31  move_stack(argv[1]);
32 }
```



[Fig. 12] The detection mechanism architecture in PIN binary framework

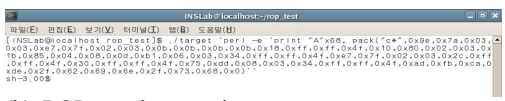
이러한 ROP 공격에 취약함 샘플 코드를 이용하여 Fig. 13과 같은 ROP 시퀀스를 구성하여 ROP 공격 탐지 메커니즘의 성능을 분석하였다. ROP 공격 샘플 프로그램은 버퍼 오버플로우를 통해 최초 ROP 시퀀스로 제어 흐름을 변경한 후 최종적으로 /bin/sh를 수행하는 ROP 공격이다.



[Fig. 13] Sequences of ROP attack

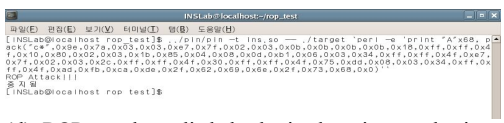
ROP 공격 탐지 메커니즘에 대한 실험으로 공격상황과 정상적인 프로그램에서의 적용 등에 대한 결과는 Fig. 14와 같다.

(a) normal program execution

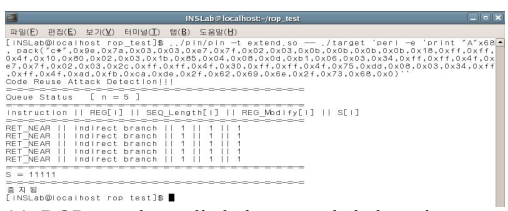


(b) ROP attack execution

(c) normal program applied the detection mechanism



(d) ROP attack applied the basic detection mechanism



(e) ROP attack applied the extended detection mechanism

[Fig. 14] The result of ROP attack detection experiments

4.1.2 JOP 공격 탐지 실험

구현된 탐지 메커니즘을 이용하여 실제 코드 재사용 공격이 탐지 가능함을 검증하기 위해서 Table 2와 같은 JOP 공격에 취약한 샘플 프로그램을 이용하여 검증을 수행하였다[14]. 검증을 수행한 환경은 다음과 같다.

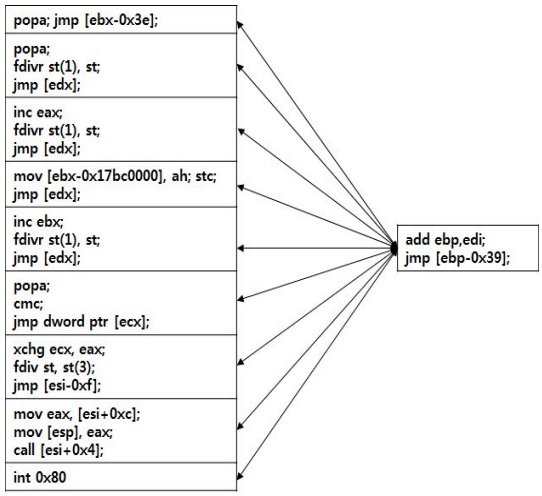
- CPU : Intel Core i5 2.8GHz
- OS : Fedora 15
- Kernel : Linux 2.6.38.6-26

이러한 JOP 공격에 취약한 샘플 코드를 바탕으로 Fig. 15와 같은 JOP 시퀀스를 구성하여 JOP 공격 탐지 메커니즘의 성능을 분석하였다. JOP 공격 샘플은 버퍼 오버플로우를 통해 최초 JOP 시퀀스로 제어 흐름을 변경한 후 최종적으로 쉘을 수행하는 공격이다.

[Table 2] The sample code of JOP attack

```
 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   asm("add ebp,edi; jmp [ebp-0x39];"); //dispatcher
16   asm("popa; jmp [ebx-0x3e];"); //initializer
17   asm("popa; fdivr st(1), st; jmp [edx];"); //g00
18   asm("inc eax; fdivr st(1), st; jmp [edx];"); //g01
19   asm("mov [ebx-0x17bc0000], ah; stc; jmp [edx];");
//g02
20   asm("inc ebx; fdivr st(1), st; jmp [edx];"); //g03
21   asm("popa; cmc; jmp dword ptr [ecx];"); //g07
22   asm("xchg ecx, eax; fdiv st, st(3); jmp [esi-0xf];");
//g08
23     asm("mov eax, [esi+0xc]; mov [esp], eax; call
[esi+0x4];"); //g09
24   asm("int 0x80"); //g0a
25   asm(".att_syntax noprefix");
26 }
27
28 void overflow() {
29    char buf[256];
30    fscanf(fd,"%[^\n]",buf);
31    return;
32 }
33
34 int main(int argc, char** argv) {
35    if(argc>1) filename = argv[1];
36    fd=fopen(filename, "r");
37    overflow();
38 }
```

탐지 메커니즘 구현을 위해 사용되는 Pintool은 JIT 기반의 최적화를 수행하기 때문에 수행이 반복되는 명령어에 대하여 측정되지 않는 경우가 발생한다. 따라서, 기본 메커니즘을 이용하여 JOP 공격을 탐지하기 위해서는 Fig. 16과 같이 명령어 포인터(IP, Instruction Pointer)를 기반으로 디스패처 가젯을 탐지하고, 탐지된 디스패처 가젯을 토대로 JOP 공격 탐지를 수행하여야 한다.



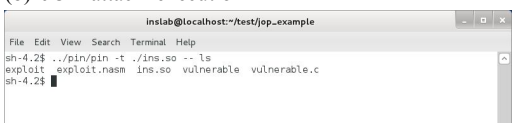
[Fig. 15] Sequences of JOP attack

[Fig. 16] Instruction pointer in JOP attack

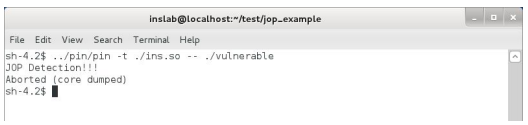
JOP 공격 탐지 메커니즘에 대한 실험으로 공격상황과 정상적인 프로그램에서의 적용 등에 대한 결과는 Fig. 17과 같다.

(a) normal program execution

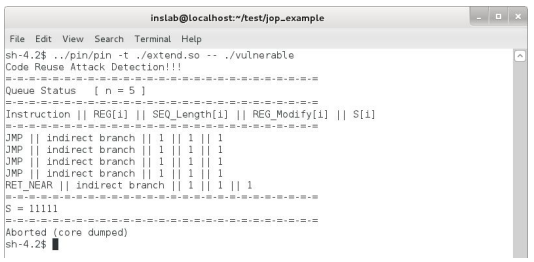
(b) JOP attack execution



(c) normal program applied the detection mechanism



(d) JOP attack applied the basic detection mechanism



(e) JOP attack applied the extended detection mechanism

[Fig. 17] The result of JOP attack detection experiments

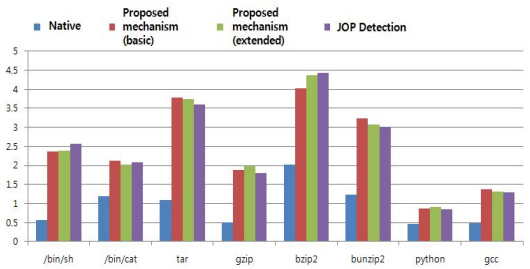
4.1.3 비교 분석

제안하는 코드 재사용 공격 탐지 메커니즘은 기존의 바이너리 환경에서의 대응 방안들이 DBI 툴을 사용하여 구현하고, 탐지를 위한 별도의 메모리 공간을 요구한다는 점에서 저장 공간상으로 유사한 오버헤드를 가진다. 그러나 기존의 대응 방안들은 ROP 및 JOP 각 공격에 국한되어 있기 때문에 이를 혼용한 공격은 탐지할 수 없다는 한계점이 있다. 하지만 제안하는 메커니즘은 ROP 및 JOP를 포함하는 코드 재사용 공격의 공통적인 특징을 이용한 대응 방안이기 때문에, 기존의 대응 방안들과는 달리 지금까지 제안된 모든 코드 재사용 공격을 탐지할 수 있는 대응 방안이다.

제안하는 탐지 메커니즘과 기존 대응 방안들을 비교한 결과는 Table 3과 같다. 구현된 탐지 메커니즘의 성능을 평가하기 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 DBI 툴로 인해 발생하는 오버헤드 및 탐지 루틴의 적용으로 인해 기본 메커니즘의 경우에는 기존의 실행시간 대비 평균 약 2.78배, 확장 메커니즘의 경우에는 약 2.8배의 오버헤드를 발생시켰다. 이러한 측정 결과는 Fig. 18과 같다.

[Table 3] The result of Analysis

detection / counter measure	ROP	JOP	ROP + JOP
DROP	O	X	X
ROP Defender	O	X	X
JOP Detection	X	O	X
Proposed detection mechanism	O	O	O



[Fig. 18] The result of performance evaluation

제안하는 메커니즘은 JOP 공격을 실시간으로 탐지하는 것을 목표로 하여 동적 바이너리 측정 툴을 이용한 바이너리 수준에서의 탐지를 수행하기 때문에 동적 바이너리 측정에 대한 오버헤드는 불가피하다. 또한, 이러한 탐지 메커니즘의 대부분의 오버헤드가 DBI로 인해 발생하기 때문에 DBI 툴의 선택에 따라 발생되는 오버헤드가 달라질 수 있다. 특히, 기존 연구 중 DROP의 경우는 DBI 툴로 Valgrind를 이용하여 5.2배의 오버헤드가 발생하였다. 하지만 DROP의 경우도 다른 DBI 툴로 구현되는 경우에는 오버헤드가 낮아질 수 있다. Fig. 18과 같이 동일한 DBI 툴을 사용하는 JOP 공격 탐지 메커니즘과의 오

버헤드는 유사하게 나타난다. 따라서 제안하는 탐지 메커니즘은 기존의 탐지 메커니즘과 유사한 오버헤드를 가지지만 ROP, JOP 및 ROP와 JOP의 혼용 공격을 모두 탐지 할 수 있다는 장점이 있다.

5. 결 론

시스템 상에서의 메모리 관련 공격은 공격자가 쉘 코드라 일컫는 임의의 코드를 실행시킬 수 있다는 점에서 대단히 위협적이다. 따라서 운영체제는 이러한 공격에 대응할 수 있는 보호 메커니즘을 탑재하고 있지만, 그에 더불어 공격 기법 또한 나날이 발전하고 있다. 특히, 코드 재사용 공격 기법들은 공격자가 공격에 요구되는 연산을 직접 프로그래밍하여 최종적으로 쉘 코드를 수행하기 위한 공격 연산을 수행시킬 수 있다는 점에서 매우 효과적인 공격 기법이다. 하지만 이러한 공격 기법을 탐지하기 위해 기존의 대응 방안들은 코드 재사용 공격으로서 나타나는 ROP 공격 또는 JOP 공격 각각에 대해서만 탐지를 수행하기 때문에 서로 간의 공격을 허용하는 한계점을 가진다. 따라서 본 논문에서는 이러한 한계를 극복하기 위해 스택 영역에서의 코드 재사용 공격 기법의 공통적 특징을 이용하는 탐지 메커니즘을 제안하였다. 제안된 탐지 메커니즘은 ROP 및 JOP 공격 모두를 탐지할 수 있으며, 이후 발생할 수 있는 ROP와 JOP 간의 혼합 공격도 탐지 가능하다. 또한 탐지 메커니즘을 구현하기 위해서 DBI 툴을 사용하였으며, DBI 툴과 탐지 메커니즘의 적용으로 인해 성능은 기존의 정상적인 수행에 비해 약 2.8배의 오버헤드가 발생하였다. 따라서 탐지 메커니즘에 대한 성능 오버헤드를 줄일 수 있는 방안에 대한 연구가 더욱 필요할 것으로 생각된다.

References

[1] Aleph. One. "Smashing The Stack For Fun And Profit", Phrack49, 1996

[2] Microsoft TechNet, "Data Execution Prevention", http://technet.microsoft.com/ko-kr/library/cc738483(WS.10).aspxc0ntex, "Bypassing non-executable stack during exploitation using return-to-libc", http://www.infosecwriters.com/text/resources/pdf/return-to-libc.pdf

[3] H. Shacham. “The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)”, the 14th ACM Conference on Computer and Communications Security, 2007
DOI: http://dx.doi.org/10.1145/1315245.1315313

[4] Pax Project, "address space layout randomization", http://pax.grsecurity.net/docs/aslr.txt, 2003

[5] Ju-Hyuk Kim, Jin-Ho Choi, Yo-Ram Lee, Soo-Hyun Oh, “Study on Return-Oriented Programming in Mac OS X”, CISC-W 2011, pp. 146-149, 2011

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns", CCS 2010, 2010

[7] T. Bletsch, X. Jiang, V. Freeh, "Jump-Oriented Programming: A New Class of Code-Reuse Attack", In CSC Technical Report TR-2010-8, NCSU, 2010

[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. In ACSAC'10, Annual Computer Security Applications Conference, 2010.

[10] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In Lecture Notes in Computer Science, 2009.

[11] Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy, "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks", Technical Report HGI-TR-2010-001, 2010.

[12] Ju-Hyuk Kim, Yo-Ram Lee, Soo-Hyun Oh, "A detection mechanism for Jump-Oriented Programming at binary level”, Journal of The Korea Institute of Information Securoty & Cryptography, vol. 22 No. 5, pp. 1069-1078, 2012.

[13] 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. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, volume 40, pages 190～200, New York, NY, USA, 2005

[14] Mehmet Kayaalp, “Example Jump-Oriented Programming Attack", http://cs.binghamton.edu/~mkayaalp/jop.html, 2012

김 주 혁(Ju-Hyuk Kim) **[정회원]**

- 2011년 2월 : 호서대학교 정보보호학과 졸업(공학사)
- 2013년 2월 : 호서대학교 대학원 정보보호학과 졸업(공학석사)
- 2013년 3월 ~ 현재 : 한국지역정보개발원

<관심분야>
메모리 보안, 모바일 보안

오 수 현(Soo-Hyun Oh) **[정회원]**

- 2000년 2월 : 성균관대학교 전기전자 및 컴퓨터공학부 대학원 졸업(공학석사)
- 2003년 8월 : 성균관대학교 전기전자 및 컴퓨터공학부 대학원 졸업(공학박사)
- 2004년 3월 ~ 현재 : 호서대학교 정보보호학과 교수

<관심분야>
암호학, 네트워크 보안