실행 제어 명령어의 목적 주소 검증을 통한 소프트웨어 공격 탐지 기법

최 명 렬[†] · 박 상 서[†] · 박 종 욱[††] · 이 균 하[†††]

요 약

소프트웨어 공격이 성공하기 위해서는 공격 코드가 프로그램의 주소 공간에 주입된 후 프로그램의 제어 흐름이 공격 코드 위치로 변경되어야 한다. 프로그램의 주소 공간 중 코드 영역은 실행 중에 변경이 불가능하므로 공격 코드가 주입될 수 있는 곳은 데이터 영역 밖에 없다. 따라서 데이터 영역으로 프로그램의 제어가 변경될 경우 주입된 공격 코드로 제어가 옮겨 가는 공격이 발생한 것으로 판단할 수 있다.

따라서 본 논문에서는 프로그램의 제어 흐름과 관련된 CALL, JMP, RET 명령어의 목적 주소를 검사하여 제어가 옮겨간 목적 주소가 프로그램의 실행 코드가 저장된 텍스트 영역이 아닌 데이터 영역일 경우 소프트웨어 공격이 발생한 것으로 간주하는 소프트웨어 공격 탐지 기법을 제안하였다. 제안된 방법을 이용하면 함수의 복귀주소뿐만 아니라 함수포인터, longjmp() 버퍼 등 프로그램 제어 흐름과 관련된 모든 데이터가 변경되었는지 점검할 수 있었기 때문에 기존 기법들보다 더 많은 종류의 공격을 탐지할 수 있었다.

키워드 : 소프트웨어 공격 탐지, 실행 제어 명령어, 목적 주소 검증

Software Attack Detection Method by Validation of Flow Control Instruction's Target Address

Myeongryeol Choi[†] · Sangseo Park[†] · Jong-wook Park[††] · Kyoon-Ha Lee[†††]

ABSTRACT

Successful software attacks require both injecting malicious code into a program's address space and altering the program's flow control to the injected code. Code section can not be changed at program's runtime, so malicious code must be injected into data section. Detoured flow control into data section is a signal of software attack.

We propose a new software attack detection method which verify the target address of CALL, JMP, RET instructions, which alter program's flow control, and detect a software attack when the address is not in code section. Proposed method can detect all change of flow control related data, not only program's return address but also function pointer, buffer of longjmp() function and old base pointer, so it can detect the more attacks.

Key Words : Software Attack Detection, Flow Control Instruction, Target Address Validation

1. 서 론

C 언어는 프로그램의 실행속도를 빠르게 하기 위해서 속도 저하를 초래할 수 있는 배열 첨자에 대한 범위 검사를 수행하지 않고 필요시 프로그래머가 구현도록 하고 있다. 따라서 프로그래머가 배열 첨자에 대한 범위 검사를 수행하지 않을 경우 할당된 버퍼 크기를 초과하여 인접한 메모리의 데이터를 임의의 값으로 변경하는 버퍼오버플로우가 발생할 수 있다. 즉, 인접한 메모리 내용 중 함수의 복귀주소(return address), 함수 포인터(function pointer) 등 프로그램의 실행 흐름과 관련된 데이터가 있을 경우 공격자는 이 값을 변경함으로써 프로그램의 실행 경로를 고의적으로 변경하여 정보를 유출, 삭제, 변조하는 공격코드를 실행시킬 수 있다. 또한 C 언어는 시스템 프로그래밍 언어로 개발되었기 때문에 메모리와 하드웨어를 프로그래머가 쉽게 조작할 수 있어 포인터 조작이 가능할 뿐 아니라 malloc(), printf() 등 보안상 문제점을 유발할 수 있는 함수들을 제공한다. 이러한 특성 때문에 C 언어로 작성된 프로그램에 대해 버퍼오버플로우 공격[1], 힙오버플로우 공격[2], 포맷스트링 공격[3], return-to-libc 공격[4] 등 다양한 공격이 가능하다.

CERT에 의하면 최근까지 보고된 소프트웨어 공격 기법의 50% 이상이 버퍼오버플로우 공격이었다[5]. 버퍼오버플

† 정 회 원 : 국가보안기술연구소 선임연구원
†† 정 회 원 : 국가보안기술연구소 책임연구원
††† 종신회원 : 인하대학교 컴퓨터공학부 교수
논문접수 : 2005년 12월 30일, 심사완료 : 2006년 6월 7일

로우 공격이 발생하면 함수의 복귀주소가 변경된다. 따라서 지금까지 제안된 기법들은 버퍼오버플로우가 발생하더라도 함수의 복귀주소가 변경되지 않도록 하거나 함수의 복귀주소가 변경된 경우 이를 탐지하여 프로그램의 실행을 종료시킴으로써 공격코드가 실행되지 못하도록 하였다. 이러한 방법은 버퍼오버플로우를 이용하여 함수의 복귀주소를 변경하는 공격에 효과적으로 대응할 수 있다. 하지만 프로그램의 실행 흐름과 관련된 데이터에는 함수의 복귀주소이외에도 함수포인터, longjmp() 함수 버퍼 등이 있으며 이들 데이터는 포맷스트링 공격, return-to-libc 공격 등의 다른 기법을 이용하여 변경할 수 있기 때문에 기존 방법들만 이용해서 모든 공격에 대응하기는 어렵다.

본 논문은 Linux 운영체제에서 프로그램의 실행 흐름을 비정상적으로 변경시키는 것을 탐지함으로써 버퍼오버플로우 공격뿐만 아니라 새로운 공격 기법에 대해서도 대응이 가능한 새로운 소프트웨어 공격 탐지 기법을 제안한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 소프트웨어 공격 기법의 공격 대상들을 살펴보고, 3장에서는 소프트웨어 공격 탐지 관련 연구 동향을 기술한다. 4장에서는 프로그램 실행 제어 명령어의 목적 주소 검증을 통한 공격 대응 기법을 제안하고, 5장에서는 제안한 방법의 성능을 평가한다. 6장에서 결론과 향후 연구 방향에 대해서 기술한다.

2. 소프트웨어 공격 기법 및 공격 대상

소프트웨어 취약점을 공격하기 위해서는 공격 대상 프로그램이 원래 의도하지 않은 명령을 실행시켜야 한다. 이때 실행될 명령어는 공격 대상 프로그램 주소 공간에 새로 주입되거나 공격 대상 프로그램에 미리 포함된 명령일 수 있다[6, 7]. [6, 7]의 내용을 요약하면 다음과 같다.

공격 대상 프로그램에 미리 포함되어 있지 않은 명령을 실행시키기 위해서는 실행시킬 명령을 공격 대상 프로그램에 주입시킨 후 주입된 명령의 시작 위치로 프로그램의 제어 흐름을 변경시켜야 한다. 이때, 실행시킬 명령은 공격 대상 컴퓨터에서 실행될 수 있는 어셈블리 코드 프로그램이어야 한다. 명령의 주입은 공격 대상 프로그램이 문자열을 입력받거나 네트워크를 통해서 데이터를 수신 하는 등 외부로부터 입력을 받아 데이터 공간에 저장할 때 일어난다. 프로그램의 제어를 주입된 명령의 시작 위치로 변경시키기 위해서는 버퍼오버플로우 등 프로그램의 실행 흐름 변경과 관련된 메모리를 변경할 수 있는 취약점을 이용해야 한다. 함수의 복귀주소, 호출한 함수의 프레임포인터(frame pointer 또는 base pointer), 함수 포인터, longjmp() 함수 버퍼 등이 이러한 목적으로 이용될 수 있다. 그 다음에는 호출된 함수에서 복귀하거나, 함수포인터를 이용하여 함수를 호출하거나, longjmp() 함수가 호출될 때 주입된 명령이 실행된다. 프로그램 실행 흐름과 관련된 데이터를 변경하는 방법으로는 버퍼오버플로우 공격, 힙오버플로우 공격, 포맷스트링 공격, 정수오버플로우 공격 등이 알려져 있다.

공격 대상 프로그램에 미리 포함된 명령을 실행시키는 방법은 return-to-libc 공격이 알려져 있다. 예를 들어 공격 대상 프로그램에 system() 함수를 호출하는 부분이 있을 경우 파라미터로 사용되는 문자열 변수의 내용을 임의의 값으로 변경함으로써 원래 의도되지 않은 다른 프로그램을 수행시킬 수 있다. 만약 문자열 변수의 내용을 "/bin/sh"로 변경하면 공격 대상 프로그램의 권한으로 명령 쉘을 수행시킬 수 있다. 이 경우에도 파라미터의 내용을 임의의 문자열로 변경한 후 system() 함수를 수행하도록 프로그램의 실행 흐름을 변경해야 한다. 즉 모든 소프트웨어 공격은 프로그램의 실행 흐름을 비정상적으로 변경시켜야 한다.

3. 관련 연구

소프트웨어 공격에 대응하기 위해서는 소프트웨어가 취약점을 가질 수 없도록 하거나 소프트웨어 공격을 탐지하여 공격을 중단시켜야 한다. 소프트웨어 공격 방지 기법은 정적인 방법과 동적인 방법으로 구분할 수 있다.

정적인 소프트웨어 공격 방지 기법은 프로그램의 소스코드를 분석하여 취약점을 찾아 제거하는 것이다. 이 방법은 발견된 취약점이 제거되면 프로그램을 실행할 때 오버헤드가 전혀 발생하지 않는 장점이 있지만 모든 취약점을 찾아내기가 어렵고(false negative) 취약점이 없는 코드를 취약점이 있다고 판단할 수 있기(false positive) 때문에 최종적으로는 프로그래머가 분석 결과를 보고 취약점 유무를 판단하여 수정해야 하므로 프로그래머에게 많은 부담이 발생한다.

동적 기법은 프로그램 작성 과정에서 모든 소프트웨어 취약점을 제거하는 것이 불가능하다는 가정을 바탕으로 한다. 이 기법에서는 프로그램이 실행되는 도중에 미리 알려진 소프트웨어 공격과 동일한 상태가 되면 공격으로 판단하여 프로그램의 실행을 종료하는 코드를 실행 프로그램에 포함시킨다.

대표적인 동적 기법으로는 StackGuard, StackShield, SSP 등이 있다. StackGuard[8]는 함수가 호출될 때 스택에 저장되는 함수의 복귀 주소가 변경 되었는지 조사하여 공격을 탐지하는 기법이다. 즉, 지역 변수와 함수의 복귀주소 사이에 카나리아(canary)라는 임의의 값을 저장하여 두고 지역 변수에서 버퍼오버플로우가 발생하여 함수의 복귀주소가 변경되면 카나리아값도 함께 변경되므로 함수에서 복귀할 때 카나리아값이 원래 값과 다른 값으로 변경되었으면 공격이 발생할 것으로 판단한다.

StackShield[9]는 함수가 호출될 때 함수의 복귀주소의 복사본을 별도로 유지되는 배열에 저장하고 있다가 함수에서 복귀할 때 배열에 저장된 복사본과 스택의 원본 복귀주소를 비교하여 일치하지 않으면 공격이 발생한 것으로 판단한다.

SSP[10]는 ProPolice라고도 불리는 방법으로 StackGuard, StackShield가 탐지하는 함수의 복귀주소 변경 공격 이외에도 로컬 변수와 함수 파라미터에 있는 함수포인터, longjmp()

버퍼 변경 공격 등을 탐지할 수 있다. 로컬 변수 영역의 문자열 변수가 저장되어 있는 주소보다 상위 주소에 함수포인터, longjmp() 버퍼 등이 로컬 변수나 함수 파라미터 형태로 있을 경우, 문자열 버퍼가 오퍼플로우되면 프로그램의 실행 흐름과 관련된 데이터가 변경될 수 있다. 따라서 SSP는 함수 파라미터와 로컬 변수에 있는 함수포인터, longjmp() 버퍼를 지역 문자열 변수보다 하위 주소로 이동시킴으로써 이들 데이터가 변경되는 것을 방지한다. 또한 함수의 복귀주소가 변경되는 것을 탐지하기 위해서 StackGuard의 카나리아값과 같은 개념인 가드값(guard value)을 사용한다.

그 밖에도, [11]에서는 함수의 복귀주소 값이 정상적인 실행 코드 영역 내부 주소인지 확인함으로써 버퍼오버플로우 공격을 탐지하는 방법을 제시한다. 이 방법은 실행 프로그램을 분석하여 실행 코드 영역의 주소를 획득하게 되는데 이 경우 동적으로 적재 또는 제거되는 동적링크라이브러리 영역의 주소는 얻을 수 없는 단점이 있다. [12]에서는 프로그램의 소스코드 없이 실행 프로그램을 분석하여 함수 단위로 실행 프로그램을 재작성하는 데 이때 함수의 복귀주소의 복사본을 별도 공간에 저장하고, 버퍼오버플로우를 유발할 수 있는 문자열 함수를 변경함으로써 버퍼오버플로우를 탐지하는 방법이다.

마이크로소프트(Microsoft)는 소프트웨어 공격을 탐지하기 위해서 다양한 방법을 적용하고 있다[13]. 프로그램을 /GS 옵션을 이용하여 컴파일 할 때 Visual C++ .Net 2003의 경우 StackGuard와 유사한 방법을 이용하여 버퍼오버플로우 공격을 탐지하고, Visual C++ .Net 2005에서는 SSP와 유사한 방법을 이용함으로써 버퍼오버플로우 공격 이외의 공격으로 탐지 범위를 확장하였다. 이외에도 Microsoft의 Windows XP SP2에는 실행보호(NX, not execute) 기법이 적용되었다[14]. 이 기법은 메모리 페이지 단위로 코드 페이지, 데이터 페이지 등 메모리 페이지의 특성을 기록하고 있으며, 데이터 페이지의 내용을 실행시키려고 할 때 예외를 발생시켜 공격을 탐지한다. 이 기법은 본 논문에서 제안하는 방법과 유사한 특성을 가지고 있지만 하드웨어의 특성을 이용하고 있어 AMD K8, Intel Itanium 등 일부 64비트 CPU에서만 적용 가능한 단점이 있다.

4. 제안하는 방법

2장에서 살펴본 것처럼 소프트웨어 공격이 성공하기 위해서는 프로그램이 의도하지 않았던 부분으로 프로그램의 실행 흐름이 변경되어야 한다. CPU의 명령어는 산술 및 논리 명령어, 자료 전송 명령어, 실행 제어 명령어 등으로 구분할 수 있다. 프로그램의 실행 흐름 변경은 실행 제어 명령어를 통해서 이루어지므로 실행 제어 명령어를 실행할 때 프로그램의 실행코드가 들어있는 주소범위 내부로 실행 흐름이 변경되는지 확인하면 소프트웨어 공격이 발생 했는지 확인할 수 있다.

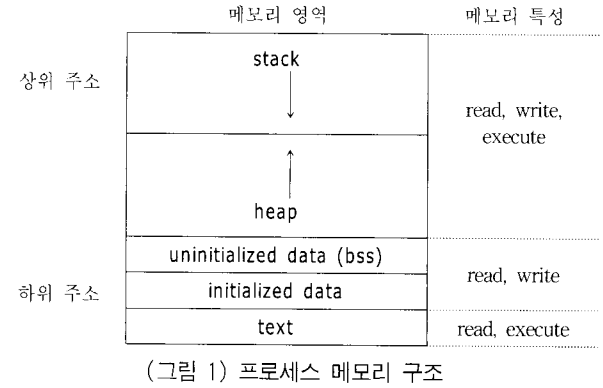
인텔 x86 CPU의 실행 제어 명령어에는 JC, JO, JS 등과 같은 조건부 분기 명령, 무조건 분기 명령 JMP, 함수 호출 명령 CALL, 함수 복귀 명령 RET가 있다[15](인터럽트 처리 루틴 호출과 관련된 INT, INTO 명령과 IRET 등은 논의에서 제외한다. 공격 코드를 인터럽트 처리 루틴으로 등록하기 위해서는 인터럽트 디스크립터 테이블(IDT, Interrupt Descriptor Table)의 내용을 변경해야 하지만 IDT는 운영체제에 의해서 보호되므로 공격 코드를 인터럽트 처리 루틴으로 등록하는 것은 매우 어렵다. 따라서 이들 명령과 관련된 공격이 일어날 가능성은 없다). 조건부 분기 명령의 경우 현재 IP(instruction pointer)를 기준으로 제어가 옮겨갈 목적주소가 상대 주소로 지정되고, JMP 명령은 분기할 목적주소가 직접 지정된다. CALL 명령의 경우 호출될 함수의 주소가 직접 지정되고, RET 명령은 함수에서 복귀하여 제어가 옮겨갈 목적주소가 명령에 직접 지정되는 대신 실행 중에 CALL 명령에 의해 스택에 저장된다. 따라서 인텔 x86 CPU에서는 이들 명령어를 대상으로 프로그램 실행 흐름이 정상적인 프로그램의 실행코드가 저장된 곳으로 변경되는지 확인함으로써 소프트웨어 공격을 탐지할 수 있다.

4.1 목적 주소 검증 대상 실행 제어 명령어

운영체제는 프로그램의 메모리 영역을 특성에 따라 실행가능(execute) 부분, 읽기가능(read) 부분, 쓰기가능(write) 부분으로 나누어 관리한다. 프로세스의 메모리 구조는 (그림 1)과 같다[11]. text 영역은 프로그램의 실행코드가 저장되는 부분으로 읽기가능, 실행가능 특성을 가지고 있다. 이 부분은 쓰기가능 특성이 없기 때문에 프로그램 실행 도중에는 변경이 불가능하다. 따라서 만약 프로그램에서 내용을 변경하려고 하면 메모리 오류가 발생되어 프로그램이 비정상적으로 종료된다.

일반적인 데이터는 (un)initizliaed data 영역과 heap, stack 영역에 저장된다. (un)initialized data 영역은 읽기가능, 쓰기가능 특성을 가지고 있으나 실행은 불가능하다. 이 부분의 내용을 실행하려고 할 경우 메모리 오류가 발생한다. heap, stack 영역의 내용은 읽기가능, 쓰기가능, 실행가능 특성을 가지고 있다. 이 영역에 실행코드를 저장하고 실행시키면 정상적으로 실행된다.

소프트웨어 공격을 위해 공격 대상 프로그램에 주입되는 공격코드는 프로그램의 실행 도중에도 변경이 가능한(un)



(그림 1) 프로세스 메모리 구조

initialized data, heap 또는 stack 영역에 저장된다. 따라서 프로그램 실행 중에 프로그램의 제어가 이 부분으로 변경된다면 소프트웨어 공격이 발생한 것으로 판단할 수 있다. 프로그램의 실행코드는 text 영역에 저장되고 이 영역은 읽기가능 특성을 가지고 있으나 쓰기가능 특성이 없으므로 소프트웨어 공격 도중에 내용이 변경될 수 없다. 따라서 실행 제어 명령어들 중에서 실행 흐름이 변경될 곳의 주소 정보가 명령어의 일부로 text 영역에 고정되는 명령어들은 소프트웨어 공격에 의해 프로그램의 제어가 비정상적인 부분으로 변경될 수 없다. 그러나 실행 제어 명령어에 의해 실행 흐름이 변경될 주소 정보가 (un)initialized data 영역이나 heap, stack 영역에 저장되는 경우에는 실행 흐름 목적 주소가 정상적인 부분인지(text 영역인지) 확인해야 한다.

실행 제어 명령어들 중 조건부 분기 명령의 경우, 분기 대상이 현재 명령어로부터의 오프셋(offset)으로 주어지고 그 값은 명령어의 일부로 text 영역에 저장된다. 무조건 분기 명령의 경우, 분기 대상이 컴파일할 때 결정될 수도 있고(직접주소지정 오퍼랜드, direct operand), (un)initialized data, heap 또는 stack 영역에 있는 데이터가 분기 대상이 될 수도 있다(간접주소지정 오퍼랜드, indirect operand). 간접주소지정 오퍼랜드의 경우 프로그램이 실행되는 동안 그 내용이 변경될 수 있으므로 간접주소지정 오퍼랜드를 사용하는 무조건 분기 명령인 JMP를 실행할 때는 분기 대상 주소가 정상적인 영역에 속하는지 확인해야 한다. 함수 호출 명령의 경우도 간접주소지정 오퍼랜드를 이용한다면 함수 주소를 점검해야 한다. 함수 복귀 명령 RET의 경우 실행 흐름은 함수 호출 명령인 CALL을 실행하는 동안 stack 영역에 저장된 복귀주소로 변경된다. 따라서 RET의 경우는 복귀주소가 변경될 수 있으므로 복귀주소를 항상 정상 범위인지 확인해야 한다.

4.2 longjmp()

C 언어는 4.1 절에서 살펴 본 실행 제어 명령 이외에 프로그램 전역에서 제어를 변경할 수 있는 longjmp 메커니즘을 제공한다. longjmp 메커니즘은 setjmp()와 longjmp() 함수에 의해서 구현된다. setjmp() 함수는 setjmp(env); 형태로 호출되는데 setjmp()가 호출되는 시점의 스택 상태와 레지스트리 값들을 env 변수에 저장한다. longjmp()는 longjmp(env, val); 형태로 호출된다. longjmp()가 호출될 때 레지스트리 값들은 env에 저장된 값으로 복원되고, 프로그램의 실행 흐름은 간접주소지정 JMP 명령을 이용하여 env에 저장되어 있는 setjmp() 함수를 호출한 위치로 변경된다. env 변수는 (un)initialized data, heap 또는 stack 영역에 저장되기 때문에, env 중에서 IP 값이 변경되면 프로그램의 실행 흐름이 변경되므로 longjmp() 함수가 호출되기 전에도 IP 값이 정상적인 범위에 속하는지 검증해야 한다.

4.3 프로그램의 실행코드 영역의 주소 범위 획득

제어 흐름 명령어의 목적 주소가 정상적인 범위에 속하는지 확인하기 위해서는 프로세스의 메모리 영역 중 프로그램 실행코드가 저장된 text 영역의 주소 범위를 획득할 수 있어야 한다. Linux 운영체제의 경우 프로세스의 메모리 맵을 획득할 수 있는 간단한 방법을 제공한다. (그림 2)는 Linux 운영체제의 /sbin/init 프로세스가 점유하고 있는 메모리 맵을 /proc 파일시스템의 /proc/1/maps 파일에서 획득한 것이다(1은 /sbin/init 프로세스의 프로세스 id이다)[16]. 이 그림에서 각 메모리 영역별로 주소 범위, 메모리 특성(허용 동작)과 함께 메모리 영역에 어떤 파일이 맵핑되었는지 나타내고 있다. 또한, /sbin/init 프로그램은 /lib/ld-2.2.93.so와 /lib/i686/libc-2.2.93.so 라이브러리를 이용하고 있음을 알 수 있다.

```
   Address range    Perms  Mapped file
  08048000-0804e000 r-xp   /sbin/init
  0804e000-0804f000 rw-p   /sbin/init
  0804f000-08052000 rwxp
  40000000-40012000 r-xp   /lib/ld-2.2.93.so
  40012000-40013000 rw-p   /lib/ld-2.2.93.so
  4002f000-40030000 rw-p
  42000000-42126000                        r-xp
/lib/i686/libc-2.2.93.so
  42126000-4212b000                        rw-p
/lib/i686/libc-2.2.93.so
  4212b000-4212f000 rw-p
  bffff000-c0000000 rwxp
```

(그림 2) /sbin/init 프로세스의 메모리 영역

/sbin/init 프로그램이 실행되는 동안 소프트웨어 공격이 발생하지 않는다면 r-xp 특성을 가지는 부분으로만 제어 흐름이 일어나야 한다. 따라서 4.1절에서 살펴본 목적 주소 검증 대상 명령어들에 대해서 제어 흐름의 목적 주소가 r-xp 특성을 가지는 메모리 영역 안에 있는지 점검해야 한다. 이 영역들은 /proc/<process id>/maps 파일을 읽어 메모리 영역의 특성이 r-xp인 메모리 주소 범위를 유지함으로써 얻을 수 있다.

Linux 운영체제에서 프로세스의 메모리 영역은 프로그램 실행 도중에 변경될 수 있다. heap 영역의 경우, 프로그램에서 동적으로 메모리를 할당한다면 운영체제가 사전에 heap 영역으로 할당해 놓은 영역이 모자라게 되면 heap 영역이 확장될 수 있다. stack의 경우도 운영체제가 stack 영역으로 할당해 놓은 영역이 부족하다면 확장된다. 이와 같은 heap과 stack 영역의 확장은 프로그램의 실행코드 영역이 확장되는 것이 아니므로 프로그램의 실행코드 범위를 추적할 때 고려할 필요가 없다. Linux 운영체제는 프로그램 실행 중에 프로그램이 사용하는 라이브러리를 동적으로 적재 또는 제거하는 동적 로딩을 제공한다. dlopen(), dlclose()을 이용하면 프로그램 실행 중에 heap과 stack 영역 사이에 프로그램에서 필요로 하는 라이브러리를 동적으로 로딩할 수 있다. 따라서 main() 함수 시작 후에 프로세스의 실행코드 영역에 대한 최초 범위를 획득하였더라도 dlopen()과 dlclose()의 호출 전후에는 프로세스의 메모리 맵이 변경되므로 실행코드 영역의 범위를 다시 획득해야 한다.

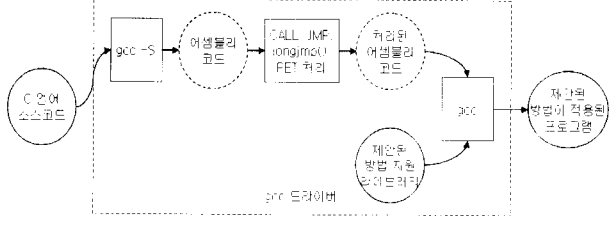
4.4 구현

StackGuard, SSP의 경우 해당 공격 탐지 기법을 적용하기 위해서 gcc 컴파일러를 수정하기 위해서 소스코드를 패치하였으며[8, 10] StackShield의 경우 gcc 컴파일러를 이용하여 어셈블리 코드를 임시로 생성한 후 공격 탐지 기법을 적용하고 다시 gcc 컴파일러를 호출하여 컴파일하는 gcc 드라이버로 구현되었다[9]. 컴파일러를 직접 수정할 경우 컴파일러 버전이 변경될 경우 새로운 패치를 생성해야 하는 단점이 있으므로 본 논문에서는 공격 탐지 기법을 StackShield와 같이 gcc 드라이버 형태로 구현하였다. 구현된 gcc 드라이버가 C 언어 소스코드를 이용하여 제안된 공격 탐지 기법이 적용된 실행파일을 생성하는 과정을 (그림 3)에 나타내었다.

첫 번째로 C 언어 소스코드를 -S 옵션으로 컴파일하여 어셈블리 코드를 임시 파일로 생성하였다.

그 다음, 생성된 어셈블리 코드에서 프로그램 제어 흐름과 관련된 CALL, JMP, RET 명령어와 longjmp()함수 호출을 찾아서 정상적인 실행코드 영역으로 제어 변경이 일어나는지 확인하는 코드를 추가하고, 프로그램의 실행코드 영역이 변경되는 main(), dlopen(), dlclose() 함수 호출을 찾아서 실행코드 영역의 범위를 획득하는 코드를 추가하였다.

gcc는 어셈블리 코드를 생성할 때 CALL, JMP 명령의 오퍼랜드가 간접주소지정 오퍼랜드일 경우 오퍼랜드 앞에 '*' 문자를 추가한다. 따라서 CALL, JMP 명령의 오퍼랜드가 '*'로 시작될 경우 해당 명령 앞에 오퍼랜드의 값이 실행코드 영역에 속하는 지 점검하는 코드를 추가하였다. (그림 4)는 간접주소지정 CALL 명령에 대해서 현재 단계까지 처리된 결과의 예이다. 이 중에서 __IsValidCodeAddr, _AddrError 등은 (그림 3)의 '제안된 방법 지원 라이브러리'에 포함되어 있는 코드이다.



(그림 3) 제안된 방법의 컴파일 과정

```
movl   -8(%ebp), %eax
call   *%eax
```

```
movl   -8(%ebp), %eax
pushl  %eax
call   __IsValidCodeAddr
cmpl   $0, %eax
popl   %eax
jne    .LC0
pushl  %eax
call   __AddrError
.LC0:
```

(a) 간접주소지정 CALL (b) 제안된 방법이 적용된 간접 주소지정 CALL

(그림 4) 제안된 방법 적용 예

RET 명령에 대해서는, 스택에 저장된 복귀주소가 text 영역에 속하는지 점검하는 코드를 추가하였다. longjmp() 함수 호출의 경우에는 함수의 struct jmp_buf 타입 파라미터인 첫 번째 파라미터에서 IP가 저장된 곳의 값을 점검하는 코드를 추가하였다. 또한 프로세스가 실행되는 동안 프로세스 메모리 영역 중 text 영역의 주소 범위를 구하기 위해서 main() 함수의 시작될 때, 그리고 dlopen()과 dlclose() 함수가 각각 호출된 직후에 프로세스 id를 이용하여 /proc 파일시스템의 프로세스 메모리 맵으로부터 text 영역의 범위를 획득하는 코드를 호출하는 코드를 추가하였다.

처리된 어셈블리 코드는 실행 프로그램을 생성하기 위해서 라이브러리 코드와 함께 gcc를 이용하여 컴파일된다. 라이브러리에는 앞에서 살펴본 __IsValidCodeAddr, __AddrError 함수 등의 지원함수, text 영역의 주소 범위를 저장하기 위한 배열, 그리고 프로세스의 메모리 맵으로부터 text 영역의 주소 범위를 구하는 함수가 포함되어 있다. (그림 5)는 본 논문에서 구현한 text 영역 주소 범위를 관리하는 자료구조와 함수를 가상 코드로 표현한 것이다.

```
int __VAC; // Valid Address Count
struct { int start; int end } __VAL[64];  // Valid Address List

void __InitVAL(void) {
    __VAC;
    /proc/<process id>/maps file을 읽음;
    for each 메모리 영역 {
        영역의 시작 주소, 끝 주소, 메모리 특성을 분석;
        if (메모리 특성 == "r-xp") {
            시작 주소, 끝 주소를 __VAL[__VAC]에 등록;
            ++__VAC;
        }
    }
}
```

(그림 5) text 영역 주소 범위 관리 코드

5. 성능 평가

제안된 기법의 성능을 평가하기 위해서 정성적 평가와 정량적 평가를 수행하였다. 정성적 평가에서는 제안된 기법을 이용하여 발생 가능한 모든 소프트웨어 공격을 탐지할 수 있는지 확인하고, 정량적 평가에서는 제안된 기법이 어느 정도의 실행 성능 감소를 초래하는지 확인하였다.

5.1 정성적 평가

정성적인 평가를 위해서 공개된 공격코드들에 제안된 기법을 직접 적용해보는 대신 John Willander[7]가 제안한 동적 소프트웨어 공격 탐지 기법의 효과를 시험하는 테스트베드를 이용하였다. [7]에서 제안한 테스트베드는 소프트웨어 공격의 공격 대상(함수포인터, 함수 복귀주소, 호출 함수의 프레임에 대한 포인터, longjmp() 버퍼)과 공격 대상이 존재할 수 있는 위치(stack, heap, (un)initialized data) 그리고 공격 대상을 변경하는 기법(버퍼오버플로우처럼 주위 데이

터를 함께 변경시키는 방법, 공격 대상 데이터만 변경하는 방법)에 따라 조합 가능한 20가지 공격 유형에 대해서 동적인 소프트웨어 공격 탐지 기법의 효과를 검증할 수 있는 도구이다(제시된 공격 유형 중 중복되는 것이 있어 18가지 공격 유형에 대해서 공격 탐지 기법의 효과를 검증할 수 있다). 이 테스트베드는 이론적으로 발생가능한 모든 소프트웨어 공격에 대해서 공격 탐지 기법이 탐지가 가능한 지를 간단하게 확인할 수 있게 해준다.

이 테스트베드를 이용하여 본 논문에서 제안한 방법과 대표적 동적 소프트웨어 공격 탐지 기법인 StackGuard, StackShield, SSP의 공격 탐지 범위를 비교하면 <표 1>과 같다(StackGuard, ShtackShield, SSP에 대한 평가는 [17]의 결과를 활용하였다). 결과에서 제시된 것처럼 본 논문에서 제안한 방법은 가장 많은 형태의 공격을 탐지할 수 있었다.

〈표 2〉 소프트웨어 공격 탐지 기법의 탐지 범위 비교

	Stack-Guard	Stack-Shield	SSP	제안된 방법
Stack overflow to target				
• Parameter function pointer	×	×	○	○
• Parameter longjmp buffer	×	×	×	○
• Return address	○	○	○	○
• Old base pointer	○	○	○	○
• Function pointer	×	×	○	○
• Longjmp buffer	×	×	○	○
Heap/BSS overflow to target				
• Function pointer	×	×	×	○
• Longjmp buffer	×	×	×	○
Pointer on stack				
• Parameter function pointer	×	×	○	○
• Parameter longjmp buffer	×	×	○	○
• Return address	×	○	○	○
• Old base pointer	○	○	○	○
• Function pointer	×	×	○	○
• Longjmp buffer	×	×	○	○
Pointer on heap/BSS				
• Return address	×	○	×	○
• Old base pointer	○	○	○	○
• Function pointer	×	×	×	○
• Longjmp buffer	×	×	×	○

그러나 제안된 방법은 제어 흐름 명령의 목적 주소를 검증하여 공격을 탐지하기 때문에 공격을 탐지하지 못하거나(false negative) 공격이 아닌 경우에도 공격으로 탐지(false positive)하는 단점이 있다. <표 2>와 같이 return-to-libc 공격이 발생하면 실행 제어 명령어가 실행될 때 프로그램이 원래 의도하지 않았던 비정상적인 제어 흐름이 발생하지만 실행 제어 명령어의 목적 주소가 프로그램의 코드 영역에 속해 있으므로 본 논문에서 제안한 방법으로는 탐지가 불가능하다. 또한 Lisp나 Object C 등과 같은 언어는 프로그램을 수행하는 중에 스택 영역의 일부에 실행 코드를 동적으로 생성하여 실행시키는 트램폴린(trampoline) 함수 기능을 지원하는 데, 제안된 기법이 적용된 프로그램에서 트램폴린 함수를 호출할 경우 정상적인 제어 흐름이지만 실행 제어 명령어의 목적 주소가 프로그램의 데이터 영역에 속해 있으므로 공격으로 탐지한다. 이러한 탐지 오류는 윈도우 XP SP2의 실행보호(NX) 기법에서도 동일하게 발생한다[14]. 윈도우 XP SP2에서는 이 두 가지 오류 중 트램폴린 함수와 관련된 문제를 해결하는 방법으로 프로그램이 동적으로 코드를 생성하여 실행시키고자 할 때는 코드를 저장할 공간을 동적으로 할당받은 후 할당된 메모리 영역에 실행가능 특성을 추가하여 사용하라고 권고하고 있다. 본 논문에서 제안한 방법도 이러한 방법을 이용하는 프로그램에 대해서는 메모리 영역의 특성을 변경하는 함수 다음에 정상적인 실행코드 영역의 범위를 다시 계산하는 코드를 삽입함으로써 정상적인 호출을 공격으로 탐지하는 오류를 제거할 수 있다. 하지만 return-to-libc 공격을 탐지하기 위해서는 새로운 방법을 강구해야 한다.

〈표 3〉 제안된 방법의 탐지 범위

제어 흐름의 정당성 / 제어 흐름 목적지	정상적인 제어 흐름	비정상적인 제어 흐름
코드 영역	정상적상적인 수행	false negative (예: return-to-libc 공격)
데이터 영역	false positive (예: trampoline 함수)	공격 탐지

5.2 정량적 평가

본 논문에서 제안한 방법을 적용했을 때 발생하는 최대 오버헤드를 계산하기 위해서 RET, 간접주소지정 CALL, 간접주소지정 JMP 명령 위주로 작성된 프로그램의 실행시간을 제안된 방법이 적용되지 않았을 때의 실행시간과 비교하였다. 또한, 일반적인 경우의 오버헤드를 계산하기 위해서 함수 호출이 많은 경우인 재귀 함수를 이용해 구현된 Hanoi Tower 프로그램과 디스크 I/O 및 연산이 많은 경우인 /bin/tar 프로그램의 실행시간을 비교하였다. 실행시간 계산은 Intel Pentium III 1GHz CPU, 128MB RAM 상에서 Red Hat Linux 8.0(kernel 2.4.18)이 설치된 컴퓨터에서 수행되었다.

먼저, RET 명령어의 오버헤드를 계산하기 위해서 아무런 작업을 수행하지 않는 함수를 500,000,000번 호출할 때 소요된 시간을 비교하였다. 그 결과 <표 3>에서 보인 것과 같이 113.7%의 오버헤드가 발생하였다. 이 오버헤드는 RET 명령을 수행할 때 복귀주소가 정상적인 실행코드 범위에 있는지 확인하기 때문에 발생하는 것이다.

두 번째로, 첫 번째 사용된 함수를 함수포인터를 이용하여 간접주소지정 CALL 명령이 실행될 때 발생하는 오버헤드를 측정하였다. 이 경우 161.2%의 오버헤드가 발생하는데, 그 이유는 간접주소지정 CALL에서 함수포인터의 값이 정상적인 실행코드 범위에 있는지 확인하는 부분이 추가되었기 때문이다.

세 번째로, 두 번째 사용된 프로그램에서 호출되는 함수에 간접주소지정 JMP 명령이 포함되도록 수정한 후 오버헤드를 계산하였다. 이 경우 180.5%의 오버헤드가 발생하였는데, 이는 간접주소지정 JMP에서 목적 주소가 실행코드 범

위에 있는지 확인하는 부분이 추가되었기 때문이다. 이 경우 목적 주소 점검 대상인 간접주소지정 CALL, 간접주소지정 JMP, RET 명령어를 모두 포함하고 있는 최소한의 프로그램이므로 이때 발생한 오버헤드는 제안된 방법으로 인해 발생할 수 있는 최대 오버헤드로 간주할 수 있다. 즉, 제안된 방법은 최대 180.5%의 오버헤드가 발생한다.

〈표 4〉 정량적 평가 결과

평가 방법	실행시간(초)	제안된 방법의 실행시간(초)	제안된 방법의 오버헤드(%)
RET	5.05	10.79	113.7
간접주소지정 CALL & RET	6.05	15.80	161.2
간접주소지정 JMP & 간접주소지정 CALL & RET	8.66	24.29	180.5
Hanoi Tower	86.88	127.56	46.8
/bin/tar	78.41	79.33	1.2

네 번째로, 32 개의 디스크로 이루어진 Hanoi Tower 문제의 경우 46.8%의 오버헤드가 발생했다. Hanoi Tower 문제를 재귀 함수를 이용하여 구현하면 거의 대부분의 실행 시간이 함수를 호출하고 복귀하는데 소요되므로 이 오버헤드를 함수 호출이 많은 프로그램의 오버헤드로 간주할 수 있다.

다섯 번째로, /bin/tar 프로그램으로 gcc 컴파일러의 3.3.6 버전 소스코드를 -z 옵션을 이용하여 압축된 tar 파일로 만들 때 발생한 오버헤드는 1.2%였다. /bin/tar 프로그램의 경우 실행 시간의 대부분을 디스크 I/O와 압축을 위한 연산에 소모하므로 이 오버헤드를 I/O와 연산을 주로 하는 프로그램의 오버헤드로 간주할 수 있다.

이 결과들로부터 제안된 방법이 일반적인 프로그램에 적용되었을 경우 프로그램의 종류에 따라 발생하는 오버헤드에 편차가 있지만 50% 이하인 것으로 추정할 수 있다.

6. 결론 및 향후 연구 과제

소프트웨어 공격이 성공하기 위해서는 프로세스의 데이터 영역에 주입된 공격코드를 실행하기 위해서 프로그램의 제어 흐름 변경과 관련된 데이터 영역을 변경시켜야만 한다. 본 논문에서는 이러한 특성을 이용하여 프로그램의 제어 흐름을 변경시키는 명령어들의 목적 주소가 프로그램 코드 영역인지 공격코드가 주입되어 있을 수 있는 데이터 영역인지 확인하는 소프트웨어 공격 탐지 기법을 제안하고 제안된 기법의 성능을 분석하였다.

정성적인 평가 결과로부터 제안된 방법을 적용하면 소프트웨어 공격의 대상이 될 수 있는 프로그램의 제어 흐름과 관련된 18가지 종류의 데이터를 변경하는 공격 기법을 탐지할 수 있어 기존의 방법들보다 더 많은 범위의 공격을 탐지할 수 있음을 알 수 있었다. 특히 기존 방법들이 해결할 수 없었던 longjmp()와 관련된 공격을 탐지할 수 있었다. 하지만 제안된 방법은 실행 제어 명령어의 목적 주소만으로 공격을 탐지하기 때문에 트램폴린 함수를 호출하는 프로그램에 대해서는 정상적인 제어 흐름이지만 공격으로 탐지하고, return-to-libc 공격 기법의 경우 탐지하지 못함을 확인할 수 있었다. 트램폴린 함수와 관련된 문제점은 트램폴린 함수를 사용하는 프로그램을 재작성함으로써 해결할 수 있으나 return-to-libc 공격의 경우 공격을 탐지할 수 있는 다른 방법을 강구하여야 한다. 정량적인 평가 결과, 제안된 방법은 일반적인 프로그램의 경우 최대 50%의 오버헤드가 발생할 수 있다는 것을 알 수 있었다. 이러한 오버헤드는 간접주소지정 JMP, 간접주소지정 CALL 및 RET 명령이 실행될 때 제어가 옮겨가는 목적지 주소가 코드 영역인지 확인하기 때문에 발생한다. 본 논문의 구현에서는 각 프로세스의 코드 영역의 주소 리스트를 유지하고 목적지 주소가 주소 리스트에 포함되어 있는지 확인하므로 많은 비교 연산으로 인해 오버헤드가 발생하였다. 특정 메모리 주소가 주어졌을 때 해당 주소가 데이터 영역인지 또는 코드 영역인지 효율적으로 판단할 수 있는 방법을 찾으면 이러한 오버헤드를 감소시킬 수 있을 것으로 판단된다.

참 고 문 헌

[1] AlephOne, Smashing the Stack for Fun and Profit, Phrack, Volume 7, Issue 49, http://www.phrack.org/phrack/49/P49-14, 1996.11.

[2] Matt Conover and w00w00 Security Team, w00w00 on Heap Overflows, http://www.w00w00.org/files/articles/heaptut.txt, 1999.1.

[3] Tim Newsham, Format String Attacks - White Paper, http://www.lava.net/~newsham/format-string-attacks.pdf, Setp., 2000.

[4] Wikipedia, Return-to-libc attack, http://en.wikipedia.org/wiki/Return-to-libc.

[5] CERT Condination Center, http://www.cert.org/advisories

[6] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," DARPA Information Survivability Conference and Exposition (DISCEX 2000), Jan., 2000.

[7] John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," Network and Distributed System Security Symposium (NDSS) '03, Feb., 2003.

[8] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang and Heather Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 7th USENIX Security Symposium, Jan., 1998.

[9] Vendicator, StackShield: A Stack Smashing Technique Protection Tool for Linux, http://www.angelfire.com/sk/

stackshield.
[10] Hiroaki Etoh and Kunikazu Yoda, Protecting from Stacksmashing Attacks, http://www.trl.ibm.com/projects/security/ssp/main.html, Jun., 2000.
[11] 김종의, 이성욱, 홍만표, "버퍼오버플로우 공격 방지를 위한 컴파일러 기법," 정보처리학회논문지C, 제9-C권 제4호, pp.453-458, 2002.
[12] 김윤삼, 조은선, "이진 코드 변환을 이용한 효과적인 버퍼 오버플로우 방지기법," 정보처리학회논문지C, 제12-C권 제3호, pp.323-330, 2005.
[13] Microsoft, Microsoft Security Developer Center, http://msdn.microsoft.com/security/.
[14] Microsoft, Changes to Functionality in Microsoft Windows XP Service Pack 2 - Part 3: Memory Protection Technologies, http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx.
[15] Intel, Intel Architecture Software Developer's Manual Vol. 2: Instruction Set Reference
[16] Daniel Bovet and Marco Cesati, Understanding the LINUX Kernel: From I/O Ports to Process Management, 2nd Ed., pp.670-672, Dec., 2002.
[17] Peter Silberman and Richard Johnson, "A Comparison of Buffer Overflow Prevention Implementations and Their Weaknesses," Black Hat USA 2004 Briefings and Training, Jun., 2004.

박 상 서

e-mail : sangseo@etri.re.kr
1991년 중앙대학교 전자계산학과(공학사)
1993년 중앙대학교 전자계산학과(공학석사)
1996년 중앙대학교 전자계산학과(공학박사)
1996년~1998년 국방정보체계연구소 선임연구원
1999년~2000년 국방과학연구소 선임연구원
2000년~현재 국가보안기술연구소 선임연구원
관심분야 : 정보전

박 종 욱

e-mail : khspjw@etri.re.kr
1986년 경북대학교 전자공학과(공학사)
1988년 경북대학교 전자공학과(공학석사)
2004년 경북대학교 전자공학과(공학박사)
1988년~2000년 국방과학연구소 선임연구원
2000년~현재 국가보안기술연구소 책임연구원
관심분야 : 정보보호, 네트워크 통신

최 명 렬

e-mail : rchoi@etri.re.kr
1991년 인하대학교 전자계산공학과(공학사)
1993년 인하대학교 전자계산학과(공학석사)
1996년~1998년 국방정보체계연구소 선임연구원
1999년~2000년 국방과학연구소 선임연구원
2001년~현재 인하대학교 컴퓨터 · 정보공학과 박사과정
2000년~현재 국가보안기술연구소 선임연구원
관심분야 : 정보보호, 정보보증

이 균 하

e-mail : khlee@inha.ac.kr
1970년 인하대학교 전기공학과(공학사)
1976년 인하대학교 전자공학과(공학석사)
1981년 인하대학교 전자공학과(공학박사)
1977년~1981년 광운대학교 교수
1981년~현재 인하대학교 컴퓨터 · 정보공학과 교수
관심분야 : 지능통신망, 패턴인식