제어흐름주소 검증을 이용한 소프트웨어 취약점 공격 대응 기법

최명렬° 김기한 박상서
국가보안기술연구소
{mrchoi°, ghkim1, sangseo}@etri.re.kr

New Defense Method Against Software Vulnerability Attack by Control Flow Address Validation

Myeongryeol Choi° Gihan Kim, Sangseo Park
National Security Research Institute

요 약

높은 효율성과 시스템 자원을 세밀하게 제어할 수 있는 편리성을 제공하기 위해서 소프트웨어의 안전성에 대한 책임을 개발자가 지게하는 C 언어의 특성으로 인해서 버퍼 오버플로우, 포맷 스트링 기법등을 이용한 소프트웨어 공격이 계속 나타나고 있다. 지금까지 알려진 소프트웨어 공격 기법의 다수가 버퍼 오버프로우 기법을 이용한 것이어서 지금까지의 연구는 주로 버퍼 오버플로우 공격 방지 및 탐지에 집중되어 있어 다른 공격 기법에 적용하는 데는 한계가 있었다.

본 논문에서는 소프트웨어 공격의 궁극적인 목적이 제어흐름을 변경시키는 것이라는 것을 바탕으로 프로그램의 제어흐름이 정상적인 범위를 벗어날 경우 이를 공격으로 탐지하는 새로운 기법을 제안하고 기존 연구 결과들과 비교하였다.

1. 서론

C 언어는 빠른 실행 속도와 시스템 자원에 대한 세밀한 제어가 가능한 특징으로 인해서 많은 시스템 소프트웨어 작성에 사용되고 있다. 하지만 시스템 자원에 대한 세밀한 제어 특징은 부주의하게 사용될 경우 시스템에 대한 비인가된 접근, 변조 등과 같은 공격을 가능하게 하는 취약점을 내포시킬 수 있는 단점이 있다. CERT에 보고되고 있는 소프트웨어 취약점은 그림 1과 같이 급격하게 증가하고 있다[1].

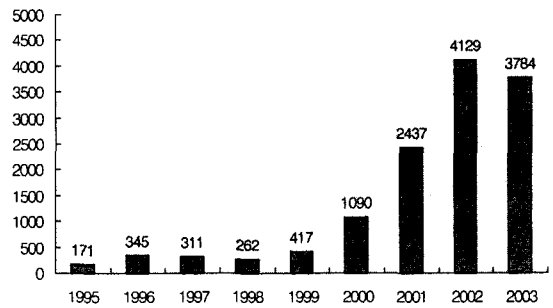

그림 1. 소프트웨어 취약점 발견 동향

현재 발견되고 있는 소프트웨어 취약점의 대부분은 버퍼 오버플로우 취약점으로 2003년의 경우 새로 발견된 소프트웨어 취약점의 50% 이상이 버퍼 오버플로우 취약점이었다[1]. 버퍼 오버플로우 취약점을 이용한 소프트웨어 공격 기술은 AlephOne[2]에 의해 공개되어 시스템 공격을 위한 익스플로잇 개발과 Slammer, CodeRed, Balster 등과 같은 웜의 전파 기법으로 사용되고 있다.

버퍼 오버플로우 취약점을 이용한 공격을 방지 또는 탐지하기 위한 기법은 크게 정적인 방법과 동적인 방법으로 나눌 수 있다.

정적인 방법은 소프트웨어 개발 과정에서 소스 코드를 분석하여 취약점이 있는 부분을 찾아 내는 방법으로 일반적으로 많은 false positive, false negative를 발생시키므로 실효성이 떨어지는 단점이 있다. 동적인 방법[3]은 소프트웨어 취약점을 이용한 공격이 탐지될 경우 프로그램 실행을 종료 시키는 코드를 소프트웨어 자체 또는 라이브러리에 삽입시키는 방법으로 이미 알려진 기법을 이용한 공격을 탐지할 수 있지만 새로운 공격 기법이 발견될 경우 새로운 공격을 탐지할 수 있는 코드를 추가하여야 하는 단점이 있다.

현재 동적인 방법을 중심으로 소프트웨어 취약점 공격을 탐지 또는 방지하기 위한 기법들이 연구되고 있지만 대부분 버퍼 오버플로우 취약점을 대상으로 한 것으로서 상대적으로 새로 발견된 소프트웨어 공격 기법들인 포맷 스트링 공격, return-to-libc 공격, 힙 오버플로우 공격, 정수 오버플로우 공격 등을 탐지하는데는 한계가 있다.

따라서 본 논문에서는 소프트웨어 취약점 공격 기법의 궁극적인 목표인 비정상적인 소프트웨어의 제어흐름을 탐지함으로써 버퍼 오버플로우 공격뿐만 아니라 새로운 공격 기법에 대해서도 동일한 방법으로 대응할 수 있는 새로운 기법을 소개한다.

2장에서는 동적인 소프트웨어 취약점 공격 탐지 관련 연구들에 대해서 알아보고, 3장에서는 본 논문에서 제안하는 제어흐름주소 검증을 통한 공격 대응 기법에 대해서 기술한다. 4장에서는 본 논문에서 제안하는 방법을 기존 기법들과 비교하고 5장에서 결론과 개선 사항들에 대해서 기술한다.

2. 관련 연구[3]

대표적인 동적 소프트웨어 공격 탐지 기법들에는 StackGuard, StackShield, SSP 등이 있다.

StackGuard는 버퍼 오버플로우 공격을 탐지하는 대표적인 기법이다. 버퍼 오버플로우 공격이 성공하기 위해서는 현재 함수의 리턴 주소가 공격자가 입력한 임의의 실행 코드가 저장된 주소로 변경되어야 한다. 즉 스택상에 생성된 버퍼를 오버 플로우 시켜 리턴 주소를 변경시켜야 하는데 이때 버퍼와 리턴 주소 사이의 데이터들이 임의의 값으로 변경되는 특성을 이용하여 리턴 주소와 버퍼가 포함된 함수의 지역 변수 사이에 임의의 값(Canary 값)을 저장하여 두고 함수가 리턴되는 시점에서 Canary 값을 원래 값과 비교하여 변경되었을 경우 공격이 발생한 것으로 탐지하는 방법이다.

StackShield는 리턴 주소를 저장할 수 있는 별도의 배열을 이용하여 함수가 호출되었을 때 리턴 주소를 배열에 저장하고 함수가 리턴될 때 배열에 저장된 리턴 주소와 스택에 저장된 리턴 주소를 비교하여 값이 일치하지 않을 때 공격이 발생한 것으로 탐지한다.

SSP는 ProPolice라고도 불리는 방법으로 StackGuard의 Canary 값과 동일한 개념의 Guard 값을 사용하는 한편 모든 char 버퍼, 포인터 형태의 파라미터와 로컬 변수의 위치를 조정하여 char 버퍼가 오버 플로우될 때 Guard 값이 반드시 변경되도록 하고, 포인터 변수는 변경되지 않도록 함으로써 공격 방지 및 탐지한다.

3. 제어흐름주소 검증을 통한 공격 대응 기법 제안

2장에서 살펴본 기법들 중 StackGuard는 여러 가지 공격 기법 중에서 버퍼 오버플로우 공격 기법만 탐지할 수 있다. StackShield는 프로그램의 실행코드가 저장된 최상위 주소를 전역변수에 저장하여 함수 호출이나 리턴 시 호출 주소 및 리턴 주소가 이 전역변수의 값보다 크면 스택에 있는 코드 등으로 제어를 옮기는 것으로 판단하여 공격으로 탐지하는 방법을 제시하고 있으나 실행코드가 저장된 최상위 주소를 얻는 방법을 제시하지 못하고 있다. SSP는 함수의 파라미터와 로컬 변수의 위치를 재조정함으로써 버퍼 오버플로우이외의 기법을 이용한 공격을 탐지할 수 있지만 파라미터나 로컬 변수에 구조체의 구성요소로서 char 버퍼가 포함된 경우 재조정을 하지 않는 등의 단점이 있다.

소프트웨어 취약점을 이용한 공격 기법의 궁극적인 목적은 공격자가 프로그램 데이터 영역(스택, 힙 등)에 입력한 실행 코드(쉘코드 등) 또는 프로그램에 이미 존재하는 의도하지 않았던 권한을 공격자에게 줄 수 있는 실행 코드(/bin/sh를 실행시키는 코드 등)를 실행시키기 위해서 프로그램의 실행 순서를 바꾸는 것이다[5, 4]. 즉 소프트웨어 공격이 성공하기 위해서는 원래 프로그램이 의도하지 않았던 부분으로 프로그램의 제어흐름이 변경되어야 한다.

프로그램의 제어흐름이 변경되는 경우는 CALL, JMP, RET 명령이 실행될 때이다.

버퍼 오버플로우 공격 기법은 함수의 리턴 주소를 변경함으로써 RET 명령이 실행될 때 제어가 옮겨갈 위치를 변경하는 기법이다. 리턴 주소는 메모리 상에서 공격자가 내용을 변경할 수 있는 스택 영역에 저장되므로 버퍼 오버플로우에 의해서 임의의 값으로 변경이 가능하다. CALL 또는 JMP 명령은 컴파일러에 의해서 명령이 생성될 경우 오퍼랜드가 정적으로 고정되는 경우와 동적으로 변경 가능한 경우로 나눌 수 있다. 정적인 경우는 프로그램 내부의 함수를 CALL하거나 함수 내부로 JMP하는 경우이다. 동적인 경우는 함수 포인터 변수에 저장된 주소로 제어를 옮기기 위해서 레지스터에 주소를 옮기고 레지스터 값의 주소로 CALL 또는 JMP를 수행하는 경우이다. C 언어는 이외에도 전역적인 범위에서 제어를 변경할 수 있는 longjmp 메커니즘을 제공한다. longjmp 메커니즘도 궁극적으로는 레지스터 오퍼랜드를 이용한 JMP 명령으로 구현된다.

따라서 실제적으로 제어흐름이 변경되는 레지스터 오퍼랜드를 이용한 CALL과 JMP 명령, 그리고 RET 명령을 수행하는 시점에서 레지스터 변수의 값과 리턴 주소의 값이 프로그램의 실행 코드가 들어있는 주소 범위내로 옮겨가는지 검증할 수 있으면 공격이 일어났는지를 확인할 수 있다. 즉 제어흐름이 공격자가 변경 가능한 데이터 부분으로 옮겨지지 않아야 한다.

운영체제는 프로그램의 메모리 영역을 특성에 따라 실행가능 부분, 읽기가능 부분, 쓰기가능 부분으로 나누어 관리한다. 그림 2는 Linux 운영체제의 init 프로세스가 점유하고 있는 메모리 맵을 /proc 파일시스템에서 보여주는 내용이다[6]. 각 메모리 영역별로 주소 범위, 메모리 특성(허용 동작)과 함께 어떤 파일이 맵핑되었는지 나타내고 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. init 프로세스의 메모리 영역

메모리 특성 중 r-xp 부분은 프로그램의 텍스트 부분으로서 이 부분의 내용은 변경이 불가능함을 보여주고 있다. rw-p는 프로그램의 데이터 부분으로서 읽기와 쓰기가 가능하지만 실행이 불가능함을 나타낸다. rwxp 부분은 스택 영역을 나타내고 있다. rw-p 부분으로 제어흐름이 변경될 경우에는 실행 불가능한 영역을 실행하려

고 하므로 MMU에 의해서 오류가 발생하게된다. rwxp 부분은 실행가능한 특성을 가지고 있지만 공격자에 의해서 임의의 실행 코드가 주입될 수 있으므로 정상적인 경우 이 부분으로 제어흐름이 옮겨질 가능성이 거의 없으므로 공격일 가능성이 높다. 따라서 제어흐름은 r-xp 부분으로만 일어나야 한다.

본 논문에서 제안하는 방법은 이러한 특성을 이용하여 레지스터를 오퍼랜드로 하는 CALL, JMP 명령과 RET 명령 수행 부분의 코드를 변경하여 해당 주소가 rwxp 부분에 해당하는 지 검증함으로써 소프트웨어 공격이 발생하였을 경우 탐지하는 방법을 이용한다. 공격 탐지 대상 C 소스 프로그램은 gcc의 -S 옵션을 이용하여 어셈블리 언어로 변환한 후 변환된 프로그램에서 레지스터를 오퍼랜드로 사용하는 CALL, JMP 명령과 RET 명령을 식별하여 해당 명령 바로 앞에 제어흐름 주소가 r-xp 부분에 해당하는지를 점검하여 다른 부분으로 제어가 옮겨갈 경우 프로그램 실행을 중단시키는 코드를 추가한다. 현재 프로그램의 r-xp 메모리 영역의 리스트는 Linux의 /proc 파일시스템의 maps 파일의 내용을 읽음으로서 획득한다.

4. 제안 기법 평가

본 논문에서 제안한 방법의 효과를 검증하기 위해서 동적인 소프트웨어 공격 탐지 기법의 효과를 시험하는 테스트 베드[3]를 사용하였다. [3]에서 제안한 테스트베드는 소프트웨어 공격 기법들이 공격 가능한 20가지 경우에 대해서 각 대응 기법이 공격을 탐지할 수 있는지 검증할 수 있게 해준다.

표 1은 기존 소프트웨어 공격 대응 기법들에 대한 실험 결과[3]와 본 논문에서 제안한 방법의 실험 결과를 비교한 것이다. 표에서 볼 수 있듯이 제안된 방법은 기존 방법들 중에서 가장 많은 형태의 공격 기법을 탐지할 수 있는 SSP보다 더 다양한 형태의 공격을 탐지할 수 있었다.

5. 결론 및 향후 연구 계획

본 논문에서는 소프트웨어의 제어흐름을 변경시키는 방법을 이용하여 버퍼 오버플로우 취약점뿐만 아니라 포맷 스트링 취약점, 정수 오버플로우 취약점 등을 포함하는 소프트웨어 공격을 탐지하는 기법을 제안하고 제안한 방법을 기존 동적인 방법들과 비교 분석하였다. 제안된 방법은 기존 방법들 중에서 가장 많은 공격 기법을 탐지할 수 있는 방법인 SSP보다 더 많은 범위의 공격을 탐지할 수 있었다.

현재 제안된 방법은 C 언어 프로그램을 gcc를 이용하여 생성된 어셈블리 코드를 후처리(post-processing)하여 공격 탐지 코드를 추가하도록 구현되어 있으므로 사용이 다소 번거롭고, 프로그램 사이즈가 커지면 효율성에서 문제가 발생하는 단점이 있다. 향후 어셈블리 코드를 생성하지 않고 직접 실행코드를 하도록 개선하는 한편 제안된 기법을 다양한 실재 환경에 적용하여 효용성을 검증하는 과정이 필요할 것으로 판단된다.

표 1. 소프트웨어 공격 탐지 기법 비교

	①	②	③	④
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	-	-	-	+

① StackGuard, ② StackShield, ③ SSP, ④ 제안하는 방법

[참고 문헌]

[1] CERT/CC. http://www.cert.org/

[2] AlephOne. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), 1996.11.

[3] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, *Network and Distributed System Security Symposium (NDSS) '03*, pp.149-162, 2003.2.

[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 Buffer-Overflow Attacks, *7th USENIX Security Symposium*, pp.63-78, 1998.1.

[5] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *DARPA Information Survivability Conference and Exposition (DISCEX) '00*, pp.119-129, 2000.1.

[6] D. Bovet and M. Cesati. *Understanding the LINUX Kernel: From I/O Ports to Process Management*, 2002.12.