컴파일러 확장을 이용한 효율적인 버퍼오버플로우 공격 방지 기법

김종의[0] 이성욱 홍만표
아주대학교 정보통신전문대학원 정보통신 공학과
{paper97i[0], wobbler , mphong}@madang.ajou.ac.kr

Efficient Defense Method of Buffer Overflow Attack Using Extension of Compiler

Jong-Ewi Kim[0] Seong-Uck Lee Man-Pyo Hong
Graduate School of Information and Communication Technology, Ajou Universty

요약

최근 들어 버퍼오버플로우 취약성을 이용한 해킹 사례들이 늘어나고 있다. 버퍼오버플로우 공격을 탐지하는 방법은 크게 입력 데이터의 크기검사, 비정상적인 분기 금지, 비정상 행위 금지의 세가지 방식 중 하나를 취한다. 본 논문에서는 비정상적인 분기를 금지하는 방법을 살펴본 것이다. 기존의 방법은 부가적인 메모리를 필요로 하고, 컨트롤 플로우가 비정상적인 흐름을 찾기 위해 코드를 추가하고 실행함으로 써 프로그램 실행시간의 저하를 단점으로 이야기할 수 있다. 본 논문에서는 부가적인 메모리 사용을 최소한으로 줄임으로 메모리 낭비를 저하시키고 실행시간에 컨트롤 플로우가 비정상적으로 흐르는 것을 막기 위한 작업들을 최소화 함으로서 기존의 방법보다 더 효율적인 방법을 제안하고자 한다.

1.서론

*최근 들어 시스템의 취약성을 이용하여 시스템을 침입하는 해킹 사례들이 많아지고 있다. 그 중 가장 많은 비율을 차지하는 공격방법은 버퍼오버플로우를 이용한 공격 방법이다.

버퍼오버플로우를 이용한 공격이란 지정된 버퍼보다 더 많은 양의 데이터를 입력하여 시스템이 비정상적으로 동작하게 만드는 것을 의미한다. C 언어에서 지정된 버퍼보다 데이터가 크게 데이터가 들어오는 것을 체크하지 않는 것이 일반적인데 버퍼를 하나하나 체크하다 보면 프로그램의 수행 시간이 느려지기 때문에 대부분의 C 컴파일러 뿐만 아니라 코드를 작성하는 프로그래머들도 이러한 입력 데이터의 크기 체크는 하지 않았다. 이러한 문제들로 인하여 대부분의 프로그램들이 버퍼오버플로우의 취약성들을 가지고 현재도 작동하고 있으며 이러한 취약 성들은 쉽게 발견되지도 고쳐지지도 않고 있다.

이러한 취약성들이 옛날부터 알려져 왔지만 프로그램의 비정상적으로 종료 할뿐이어서 사람들의 관심을 끌지는 못했다. 하지만 버퍼오버플로우가 일어나는 순간에 사용자가 원하는 명령어를 실행시킬 수 있다는 가능성이 알려지면서부터 문제가 되기 시작했다[1].

버퍼오버플로우를 이용하면 임의의 명령어를 실행시킬 수 있는데 이때 쉘(Shell)을 이용하여 명령어를 실행시킬 수 있는 발판을 마련하게 된다. 만약 공격을 당하는 프로그램이 슈퍼유저 권한으로 실행되고 있다면 슈퍼유저 권한의 쉘을 이용할 수 있으므로 많은 문제를 일으킬 수 있다[2].

* 본 연구는 BK21 사업과 안철수 연구소의 컴퓨터 바이러스 분야 전문 인력 양성 및 핵심 기술 연구 과제로 추진된 연구 결과임

기존의 버퍼오버풀로우 공격을 막기위한 방법은 다음의 세가지로 나눌 수 있다. 첫번째는 프로그램내에서 정의한 버퍼크기보다 더 많은 크기의 데이터를 버퍼에 쓰지 못하게 입력 데이터의 길이를 항상 체크하는 것이다. 두 번째는 지정된 버퍼크기보다 입력 데이터의 크기가 크게 쓰여지는 것은 허용하되 프로그램의 컨트롤 플로우가 비정상적으로 흐르지 못하게 항상 체크하는 것이다. 세 번째는 위의 두 가지의 취약점들을 모두 허용하되 컨트롤 플로우가 비정상적으로 넘어와도 정상적인 행위이외의 비정상적인 행동들은 실행하지 못하게 하는 것이다[2].

본 논문은 세 분류의 방지 방법 중에서 비정상적인 분기를 금지하는 방법에 속한 것이며 이 방법은 컴파일러를 확장한 해결책이다. 기존의 컴파일러를 수정하여 사용자가 작성한 프로그램내의 함수의 도입부와 말단부에 버퍼오버플로우 공격 탐지와 관련된 코드를 삽입함으로써 실행시에 더 많은 일을 하게한다. 또한 실행시에 부가적인 메모리를 사용하게 되므로 평소보다 더 많은 메모리를 사용하게 된다.

본 논문에서는 기존의 방법보다 부가적인 메모리 사용을 최소로 하고 실행시간의 버퍼오버플로우 탐지 속도를 더욱 향상시키기 위하여 또 다른 해결책을 제시하고자 한다.

2.관련 연구

참고문헌 [2]에는 버퍼오버플로우 공격을 막기위한 세가지 방법 중 비정상적인 분기를 금지하는 방법을 소개하고 있다.

확장된 컴파일러는 다음과 같은 일을 하도록 확장되었다. 첫번째는 함수 호출이 일어날 때 복귀 주소의 복사본을 컴파일러가 지정한 데이터 영역에 저장한다. 컴파일러는 이 복귀

주소의 복사본을 저장할 공간을 미리 확보하게 된다. 두 번째는 함수 도입부와 말단부에 새로운 코드를 삽입하도록 컴파일러를 수정하였다. 함수의 도입부에는 복귀주소를 첫번째 언급했던 데이터 영역에 저장하는 기능을 하는 코드를 삽입한다. 함수의 말단부에는 데이터 영역에 있는 복귀주소와 함수 호출이 일어날 때 스택에 저장했던 복귀주소를 비교하는 코드를 삽입하게 된다.

실행시에 함수 호출이 일어날 때 마다 데이터 영역에 복귀주소의 복사본을 저장하고 함수가 복귀할 때는 이 복사본과 스택 영역에 저장하고 있는 주소를 비교하게 된다. 정상적으로 프로그램이 진행될 경우에는 정상적으로 함수의 호출 및 복귀가 이루어 지지만 버퍼오버플로우 공격을 당하거나 프로그램 실행시에 메모리에 공격을 당했다면 함수 복귀시 복귀주소가 변경 되었기 때문에 프로세스를 종료하거나 시스템 관리자에게 e-mail을 통하여 실시간으로 버퍼오버플로우 공격을 탐지하게 된다.

침입자가 복귀 주소의 복사본을 공격할 경우를 대비하여 함수 도입부에 복귀 주소를 컴파일러가 생성해 놓은 데이터 영역에 저장하는 일 이외의 다음과 같은 일을 추가 함으로 더욱 안전하게 프로세스를 버퍼오버플로우 공격에서 보호할 수 있다. 복귀 주소를 데이터 영역에 저장할 때 데이터 메모리 영역의 속성을 복귀 주소를 저장할 때만 쓰기 가능으로 속성을 바꾸어 주고 그 이외의 경우에는 항상 읽기 가능으로만 속성을 유지해 준다. 메모리 영역을 읽기만 가능으로 유지 하기 때문에 프로세스를 버퍼오버플로우 공격에서 안전하게 보호할 수 있는 것이다.

이 방법의 단점은 두 가지 이다. 첫 번째는 프로세스가 실행될 때 방법을 적용하기 전보다 더 많은 일을 하기 때문에 시간적으로 오버해드가 수반되는 것이다. 두 번째는 함수의 호출이 일어날 때 마다 컴파일러가 생성하는 메모리 영역에 함수의 복귀 주소를 저장하므로 메모리의 낭비를 들 수 있다.

이러한 단점을 보완하기 위하여 컴파일/링크 시간 후와 실행 시간 전에 처리를 더 함으로서 실행시간에 해야 할 일을 줄일 수 있다. 다음 절에서 새로 제안하는 방법은 참고문헌 [2]와 같이 컴파일러를 확장한 방법이며 컴파일/링크 시간과 실행 시간 사이에 부가 적인 처리 과정을 거치므로 실행시간의 오버해드를 줄일 수 있을 것이다.

3.제안된 방법

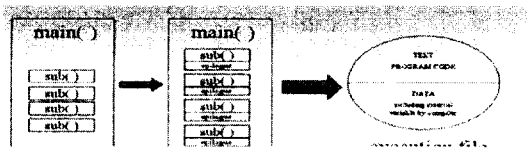
기존의 방법은 컴파일/링크 시간에 부가적으로 추가되는 코드를 실행하는 2단계로 이루어 진다. 제안된 방법에서는 컴파일/링킹 후에 만들어진 실행파일 내에서 코드영역의 메모리 주소와 크기를 얻어내어 실행파일 내에 컴파일러가 만들어 놓은 데이터영역에 메모리 주소와 크기를 삽입한다. 삽입한 메모리 주소와 크기를 실행시간에 함수가 복귀될 때마다 복귀주소가 실행파일내의 코드영역에 속하는지 체크하여 컨트롤 플로우가 코드영역 내에 속하는지 체크하는 것이다.제안된 방법은 3단계로 이루어 진다.

- ●Compile/Link Time
- ●Pre-execution Time
- ●Execution Time

3.1 Compile/Link Time

기존의 리눅스 (커널 버전 2.2.16)에서 작동하는 gcc 컴파일러 (버전 2.95.3)을 수정하였다. 수정된 컴파일러는 두 가지 기능을 첨부하게 된다. 첫번째는 컴파일/링킹후에 얻어진 실행파일에서 코드영역의 주소와 크기를 저장할 전역변수를 선언하는 것이다. 두 번째는 함수가 복귀하는 부분 즉 함수의 말단부분에 현재 함수의 복귀시에 복귀주소가 코드영역의 주소에 속하는가 체크하는 코드를 함수의 복귀부분에 항상 삽입하는 기능을 첨가하고 있다.

[그림1]은 컴파일/링킹 시간에 일어나는 작업을 보여주고 [그림2]는 함수의 말단부에 새로 삽입되는 버퍼오버플로우 탐지 코드를 보여준다.

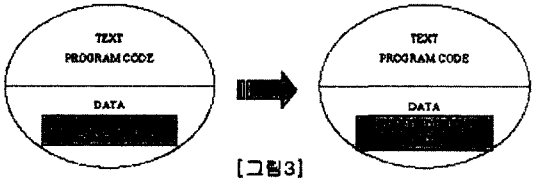



```
function epilogue pseudo code
{
 if (return address – start address of code segment) < size of code segment
   then normal function return
 else
      Buffer Overflow Attack Detected
      Process Terminate
}
```

[그림2]

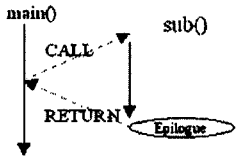
3.2 Pre-execution Time

컴파일후에 별도의 프로그램을 작성하여 처리하게 된다. 컴파일/링킹 시간 후에 얻어진 실행파일 내에서 코드영역의 시작주소와 크기를 추출하여 실행파일내의전역변수에 저장하게 된다. 컴파일/링킹 시간 후에 얻어진 실행파일 내에서 전역변수에 코드영역의 시작주소와 크기를 저장하려면 전역변수를 선언할 때 특정한 스트링으로 이루어진 시그네이처를 저장하고 나서 실행파일내부에 시그네이처를 스캔하여 전역변수의 위치를 파악할 수 있다. 별도의 프로그램은 시그네이처를 스캔하고 그 뒤에 코드영역의 시작주소와 크기를 저장하게 된다.[그림3]은 실행 전에 처리하는 작업을 보여준다.



[그림3]

3.3 Execution Time



[그림4]

위의 두 단계에서 실행파일은 함수 복귀부분에 새로운 코드를 삽입하였고 실행파일 내의 전역변수에 코드영역의 주소와 크기를 가지고 있게 된다. 프로그램이 실행될 때 함수가 호출되고 나면 프로세스의 스택 영역에 복귀 주소가 삽입되게 되고 함수가 복귀될 때 이 주소가 코드영역에 있는 지를 체크하게 된다. 정상적인 경우에는 함수의 복귀주소가 코드영역에 속하게 되므로 정상적으로 복귀하게 되지만 버퍼오버플로우 공격이 일어났거나 비정상적인 경우에는 복귀주소가 코드영역이 아닌 다른 영역이 됨을 탐지하고 프로세스를 종료하게 된다.[그림4]는 실행시간에 이루어 지는 일들을 보여준다.

4.결론 및 향후 과제

참고문헌[1]에 제시하고 있는 기존의 컴파일러 확장을 이용한 버퍼오버플로우 공격 방지기법은 실행 시에 함수의 호출이 일어날 때 도입부와 말단부에 추가된 코드의 실행에 많은 시간이 소요된다. 그 중 함수 도입부의 복귀 주소의 복사본을 저장하는 부분에서 메모리 속성을 바꾸는 데에 많은 시간적 오버헤드가 수반되는 것이 사실이다.

본 논문에서 제시한 컴파일러 확장을 통한 효율적인 버퍼오버플로우 방지 기법은 컴파일/링킹 시간 과 실행 시간 사이에 프로그램의 코드 영역의 주소를 컴파일러가 프로그램에게 정해준 데이터 영역에 저장하고 실행 시간에 해야 할 일들을 과거의 방법보다 더욱 줄임으로서 실행시에 많은 처리시간을 단축시킬 수 있을 것이라 기대한다.

부가적인 메모리 사용에 있어서도 과거의 방법은 함수 호출에 따라서 메모리 영역이 증가하게 되는 메모리 사용의 비효율성이 있었다. 본 논문에서 제시한 방법에서는 프로그램의 코드 영역을 명시하는 주소와 크기만을 저장하게 되므로 실행시에 부가적인 메모리 사용이 줄어들게 된다. 실행 속도와 부가적인 메모리 사용에 있어서 더욱 효율적으로 버퍼오버플로우 공격을 탐지할 수 있게 된다.

향후 과제로는 콜백(call back)함수와 같이 사용자가 작성한 프로그램내로 함수가 복귀하는 것이 아니고 사용자가 사용한 공유 라이브러리로 함수의 복귀가 이루어 지는 경우를 고려해야 한다.

C 라이브러리 함수 중 이진 탐색 기능을 하는 라이브러리 함수 bsearch 함수를 예를 들 수가 있다. 이 함수는 사용자가 작성한 비교 함수를 함수 포인터를 이용하여 인자로 가져야만 한다. 사용자의 프로그램 내부에서 bsearch 함수가 호출될 때 bsearch 함수는 사용자가 작성한 비교 함수를 호출하게 된다. 비교 함수가 복귀할 때 전역 변수에 명시되어 있는 프로그램의 코드영역으로 복귀하는 것이 아니라 bsearch 함수가 있는 메모리의 라이브러리 영역으로 복귀하게 된다.

논문에서 제시하고 있는 해결책에서는 사용자가 작성한 프로그램내의 함수들은 비정상 적인 경우를 제외하고 모두 프로그램의 코드영역으로 복귀한다는 가정 하에 해결책을 제시한 것이었다.

이 문제는 컴파일 이전에 고급 언어로 작성된 프로그램 코드를 파싱하여 콜 그래프를 도출해 낸 후 콜 그래프에 나타나지 않는 함수들은 주(main)함수가 호출하지 않은 것이므로 콜백 함수들이라 명한다. 이 콜백함수들에게 플래그를 첨부하여 컴파일러의 말단부를 콜백함수 라는 플래그가 있다면 정상적으로 복귀하고 그렇지 않다면 제안된 방법으로 처리를 하게 하는 방법으로 해결할 것이다.

논문에서 제시한 해결책의 운영 환경은 리눅스 커널버전 2.2.16과 gcc-2.95.3 컴파일러를 이용하여 테스트를 진행 하고 있는 중이다.

● 참고문헌

[1] PLUS(포항공대 유닉스 보안 연구회), " Security PLUS for UNIX" , 영진출판사, 2000.

[2] Tzi-cker Chiueh,Fu-Hau Hsu,"RAD : A Compile-Time Solution to Buffer Overflow Attack," Proceedings of The 21st IEEE International Conference on DISTRIBUTED COMPUTING SYSTEM 16-19 April 2001, p409, 2001.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang,"Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.", In Proceedings of the 7th USENIX Security Symposium, pages 63--78, San Antonio, TX, January 1998.

[4] 황승호, 이대현, 이종열, "컴파일러의 개발-GNU C컴파일러 포팅을 중심으로-", 반도체 설계 교육 센터.

[5] Graham Glass, King Ables ,"UNIX System V Release 4, Programmers Guide : ANSI C and Programming Support Tools, Executable and Linkable Format (ELF), Tools Interface Standards (TIS), Portable Formats Specication, Version 1.1. ", Prentice Hall, 1992.