버퍼오버플로우 공격 방지를 위한 컴파일러 기법

김 종 의[†] · 이 성 욱[††] · 홍 만 표[†††]

요 약

최근 들어 버퍼오버플로우 취약성을 이용한 해킹 사례들이 늘어나고 있다. 버퍼오버플로우 공격을 탐지하는 방법은 크게 입력 데이터의 크기 검사, 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.

Improving Compiler to Prevent Buffer Overflow Attack

JongEwi Kim[†] · Seong-Uck Lee[††] · ManPyo Hong[†††]

ABSTRACT

Recently, the number of hacking, that use buffer overflow vulnerabilities, are increasing. Although the buffer overflow problem has been known for a long time, for the following reasons, it continues to present a serious security threat. There are three defense method of buffer overflow attack. First, allow overwrite but do not allow unauthorized change of control flow. Second, do not allow overwriting at all. Third, allow change of control flow, but prevents execution of injected code. This paper is for allowing overwrites but do not allow unauthorized change of control flow which is the solution of extending compiler. The previous defense method has two defects. First, a program company with overhead because it do much thing before than applying for the method in execution of process. Second, each time function returns, it store return address in reserved memory created by compiler. This cause waste of memory too much. The new proposed method is to extend compiler, by processing after compiling and linking time. To complement these defects, we can reduce things to do in execution time. By processing additional steps after compile/linking time and before execution time. We can reduce overhead.

키워드 : 정보보안(Information Security), 버퍼넘침(buffer overflow), 배열경계(array bounds)

1. 서 론

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].

기존의 버퍼오버플로우 공격을 막기위한 방법은 다음의 세가지로 나눌 수 있다. 첫 번째는 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

※ 본 연구는 한국전자통신연구원 2002년 위탁연구에 의해 지원되었음.
† 준 회 원 : 아주대학교 대학원 정보통신전문대학원
†† 준 회 원 : 아주대학교 대학원 컴퓨터 공학과
††† 종신회원 : 아주대학교 정보통신전문대학원 교수
논문접수 : 2002년 3월 7일, 심사완료 : 2002년 6월 25일

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 실행 시에 더 많은 일을 하게 한다. 또한 실행 시에 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.

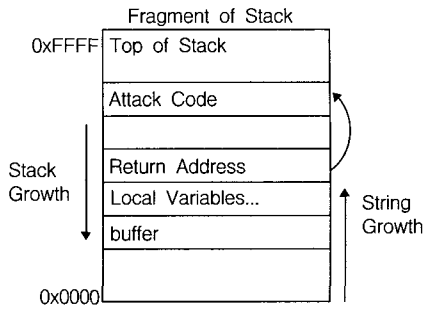
2장에서는 버퍼오버플로우 공격이 동작하는 원리에 관하여 살펴볼 것이며 3장에서는 비정상적인 분기금지 방법에 속하는 버퍼오버플로우 공격방지기법에 관하여 살펴볼 것이다. 4장과 5장에서는 3장에서 소개된 기존의 방법보다 실행시간에 부담을 줄인 새로운 버퍼오버플로우 공격방지기법을 제안할 것이다. 6장과 7장에서는 제안된 방법의 성능 테스트와 결론을 맺을 것이다.

2. 버퍼오버플로우 공격

버퍼오버플로우 공격은 프로그램이 사용자가 입력될 데이터의 크기와 저장될 공간의 크기를 검사하지 않기 때문에 발생한다. 만약 프로그램이 입력될 데이타가 저장될 버퍼 배열의 공간보다 크게 입력된다면 프로그램 내부의 입력 버퍼 배열의 주변에 있는 다른 데이터를 겹쳐 쓰게 된다. 프로그램이 실행하는 동안 버퍼의 바운더리(boundary)가 넘치는 것을 검사하지 않는 것이 버퍼오버플로우 공격을 일으키는 취약성이 된다.

C 프로그램은 실행시에 데이터영역, 스택영역, 코드영역의 세 부분으로 나뉘어 지게 된다. 스택영역에서는 지역변수, 지역 버퍼 배열, 함수의 인수, 함수의 복귀주소가 동적으로 저장된다. 공격자는 스택영역을 버퍼오버플로우 공격의 대상으로 삼게 된다. 버퍼 배열이 넘치게 되는지 검사하지 않는 다는 것을 이용하여 함수의 복귀주소를 공격자가 심어놓은 임의의 코드가 있는 곳으로 바꿔놓는다. (그림 1)은 버퍼오버플로우 공격이 일어난후의 스택의 모양을 나타낸 것이다.

버퍼오버플로우 공격의 대상이 되는 프로그램들은 대부분 슈퍼유저의 권한으로 실행되는 데몬 프로세스들이다. 공격자는 공격의 대상이 되는 프로그램 내부에 임의의 코드를 심어 놓게 되는데 대부분 쉘(Shell)을 생성시켜주는 코드를 심어 놓는다. 슈퍼유저권한을 가진 프로그램이 버퍼오버플로우 공격을 통하여 쉘이 실행되게 되면 슈퍼유저권한을 가지는 쉘이 생성되게 된다. 이때부터 공격자는 슈퍼유저의 권한을 가지게 되는 것이다.



(그림 1)

버퍼오버플로우 공격은 대부분 슈퍼유저의 권한을 얻기 위한 방법으로 사용되며 공격의 대상들은 슈퍼유저권한을 가지는 데몬 프로그램들이 된다.

3. 관련 연구

확장된 컴파일러는 다음과 같은 일을 하도록 확장되었다. 첫 번째는 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 첫 번째 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[2].

실행 시에 함수 호출이 일어날 때마다 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 이루어지지만 버퍼오버플로우 공격을 당하거나 프로그램 실행 시에 메모리에 공격을 당했다면 함수 복귀 시 복귀 주소가 변경되었기 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 유지하기 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 오버헤드가 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 3장에서 소개한 컴파일러를 확장한 방법이며 컴파일/링크 시간과 실행시간 사이에 부가적인 처리과정을 거치므로 실행시간의 오버헤드를 줄일 수 있을 것이다.

4. 제안된 방법

버퍼오버플로우 공격이 일어나기 위해서는 먼저 버퍼오버플로우 취약성을 이용하여 스택영역에 함수의 복귀주소를 변경시켜야 한다. 기존의 방법은 컴파일/링크 시간에 함수 호출 시 복귀주소의 복사본을 저장하는 함수의 도입 코드와 함수 복귀 시 스택에 저장되어 있는 복귀주소와 복귀주소의 복사본을 비교하는 함수의 말단 코드를 삽입하게 된다. 프로그램이 실행될 때 마다 컴파일/링크 시간에 부가적으로 추가되는 코드를 실행하는 2단계로 이루어진다[2, 3].

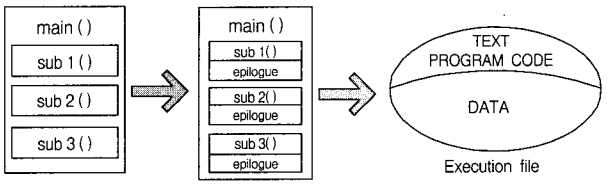
새로 제안된 방법은 기존의 방법에서 문제가 되었던 실행시간의 오버헤드를 줄일 수 있도록 설계되었다. 제안된 방법은 사용자가 작성한 프로그램 내부의 모든 사용자 정의 함수들의 복귀주소는 항상 실행 프로그램의 코드영역이라는 가정을 기반으로 하고 있다. 버퍼오버플로우 공격이 일어났을 경우에는 실행 시에 복귀주소를 코드영역 이외의 다른 영역으로 복귀시킴을 이용하여 공격자가 지시한 다른 곳으로 복귀하게 된다. 이러한 가정에 기반 하여 다음과 같은 일을 하게 된다.

컴파일/링크 시간 후에 만들어진 실행파일 내에서 코드영역의 메모리 주소와 크기를 얻어낸다. 그 후, 실행파일 내에 컴파일러가 만들어 놓은 데이터 영역에 메모리 주소와 크기를 삽입한다. 삽입된 메모리 주소와 크기는 실행시간에 함수가 복귀될 때마다 복귀주소가 실행파일내의 코드영역에 속하는지 체크하여 컨트롤 플로우가 코드영역 내에 속하는지 체크하는 것이다. 제안된 방법은 Compile/Link, Pre-execution, Execution의 3 단계로 이루어져 있다.

4.1 Compile/Link Time

복귀주소가 실행 시에 코드영역 이외의 곳으로 변경되는 것을 막기 위해 컴파일러를 다음과 같은 기능을 첨부하도록 수정하였다. 첫 번째는 컴파일/링크 시간 후에 얻어진 실행파일에서 코드영역의 주소와 크기를 저장할 전역변수를 선언하는 것이다. 두 번째는 함수가 복귀하는 부분 즉 함수의 말단 부분에 현재 함수의 복귀 시에 복귀주소가 코드영역의 주소에 속하는가 체크하는 코드를 함수의 복귀부분에 항상 삽입하는 기능을 첨가하고 있다. 기존의 리눅스 (커널 버전 2.2.16)에서 작동하는 gcc 컴파일러 (버전 2.95.3)을 수정하였다.

(그림 2)는 컴파일/링크 시간에 일어나는 작업을 보여주고 (그림 3)은 함수의 말단부에 삽입되는 버퍼오버플로우 탐지코드를 보여준다.



(그림 2)

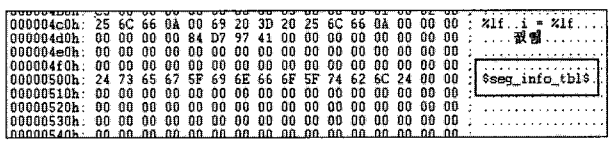
```
Function_epilogue_pseudo_code
{
  if ((return address-start address of code segment) < size of
       code segent)
    then normal function return
  else
       Buffer Overflow Attack Detected!!!
       Process Terminate
}
```

(그림 3)

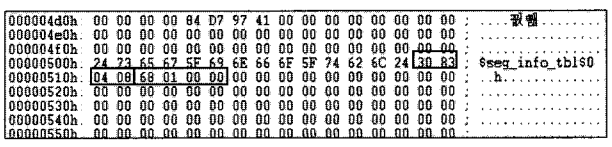
4.2 Pre-execution Time

이번단계는 기존의 방법에서의 실행 시간 오버헤드를 줄이기 위해서 추가된 단계이다.컴파일후에 별도의 프로그램을 작성하여 처리하게 된다. 컴파일/링킹 시간 후에 얻어진 실행파일 내에서 코드영역의 시작주소와 크기를 추출하여 실행파일내의전역변수에 저장하게 된다. 컴파일/링크 시간 후에 얻어진 실행파일 내에서 전역변수에 코드영역의 시작주소와 크기를 저장하려면 전역변수를 선언할 때 특정한 스트링으로 이루어진 시그네이처를 저장하고 나서 실행파일내부에 시그네이쳐를 스캔하여 전역변수의 위치를 파악할 수 있다. 별도의 프로그램은 시그네이처를 스캔하고 그 뒤에 코드영역의 시작주소와 크기를 저장하게 된다.

(그림 5)는 실행파일의 이진 형태를 보여주고 있다. 코드영역의 시작주소와 크기를 저장하기위해서 특정한 스트링으로 이루어진 시그네이처를 보여준다. (그림 4)은 별도의 프로그램이 시그네이처를 찾아서 그 뒤에 코드영역의 시작주소와 크기를 저장한 것을 보여준다.



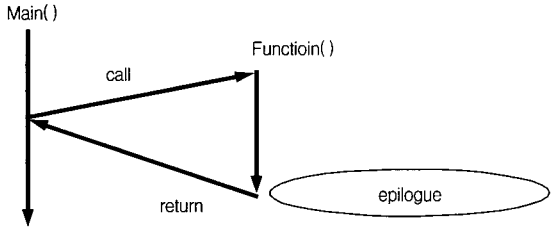
(그림 4)



(그림 5)

4.3 Execution Time

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. (그림 6)는 실행시간에 이루어지는 일들을 보여준다.



(그림 6)

5. 긍정적 결함(False Positive)

현재의 방법으로는 콜백(Call back)함수와 같이 사용자가 작성한 프로그램 내로 함수가 복귀하는 것이 아니고 사용자가 사용한 공유 라이브러리로 함수의 복귀가 이루어 지는 경우도 버퍼오버플로우 공격으로 탐지하게 된다. 이것은 콜백 함수의 경우 함수의 복귀주소가 코드영역의 주소가 아니라 공유 라이브러리이기 때문이다. 콜백 함수 문제를 해결하기 위해서 컴파일 이전에 고급언어로 작성된 프로그램 코드 수준에서 콜 그래프 작성, 콜백함수 시그네이처 작성의 2단계의 과정을 거쳐서 해결한다.

5.1 콜 그래프 작성

콜백함수 문제를 해결하기 위한 첫 번째 단계로 콜백함수로 사용되는 사용자 정의 함수를 찾아야 한다. (그림 7)은 C 라이브러리 함수 중 이진 탐색 기능을 하는 라이브러리인 bsearch 함수가 콜백 함수를 이용하여 작성된 프로그램을 보여준다.

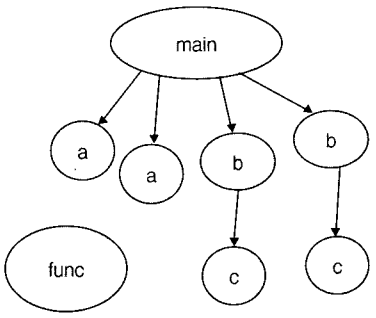
bsearch 함수가 호출 될 때 bsearch 함수는 사용자가 작성한 비교함수를 호출하게 된다. 비교함수가 복귀할 때 전역 변수에 명시되어 있는 프로그램의 코드영역으로 복귀하는 것이 아니라 bsearch함수가 있는 메모리의 라이브러리 영역으로 복귀하게 된다.

이것을 해결하기 위해서 (그림 8)과 같이 콜 그래프를 작성하게 된다. 콜 그래프를 작성하게 되면 어떠한 사용자 정의 함수에서도 호출하지 않은 함수를 발견하게 된다. 이 함수는 bsearch 함수의 인자로만 등록되어 있으므로 bsearch라는 공유 라이브러리가 호출하는 콜백함수이다.

컴파일 이전에 고급언어로 작성된 프로그램 코드 수준에서 콜 그래프를 도출함으로써 콜백함수를 찾아낼 수 있다.

```
.
.
static int func (void *, void *) ;
void a ( ) ; void b ( ) ;
int c ( ) ;
{
        .
        .
        .
        a ( ) ; a ( ) ; b ( ) ; b ( ) ;
        result = vsearch (&key, numbers, NUM,
                sizeof(numbers[0]), (void * )func) ;
        .
        .
        .
}
static int func (void * a, void * b)
{
        printf ("%d%d \n", * (int * )a, * (int * )b) ;
        if ( * (int * )a == * (int * )b) return(0) ;
        if ( * (int * )a < * (int * )b) return(-1) ;
        return 1 ;
}
void a ( ) { printf ("This is dummy function!\n" ) ; }
void b ( )
{       int dummy ;
        printf ("This is dummy function!n") ;
        dummy = c ( ) ;
}
int c( ) { printf ("This is dummy function!/n") ;  return 0 ; }
```

(그림 7)



(그림 8)

5.2 콜백함수 시그네이처 작성

별도의 프로그램을 이용하여 프로그램의 사용자 함수에 시그네이처를 삽입하여 프로그램이 실행될 때 함수의 말단부에서 콜백 함수 여부를 판단하여 버퍼오버플로우 공격을 탐지하게 된다.

(그림 9)는 콜백함수 시그네이처를 각 사용자 정의 함수에 삽입한 것을 보여준다. (그림 10)은 콜백함수 문제를 해결하기 위하여 새로 작성된 함수 말단부의 코드이다.

```
.
.
static int func (void *, void *) ;
void a ( ) ; void b( ) ;
int c ( ) ;
main( )
{
        .
        .
        .
        a ( ) ; a ( ) ; b ( ) ; b ( ) ;
        result = bsearch (&key, numbers, NUM,
                sizeof(numbers[0]), (void * )func) ;
        .
        .
        .
}
static int func (void * a, void * b)
{int callback = 1 ;
        printf("%d %d \n", * (int * )a, * (int * )b) ;
        if ( * (int * )a == * (int * )b ) return(0) ;
        if ( * (int * )a < * (int * )b ) return(-1) ;
        void a( ){int callback = 0 ;
…}
void b ( ){int callback = 0 ;
…}
void c ( ){int callback = 0 ;
…}
```

(그림 9)

```
Function_Epilgue_Pseudo_Code
{ if (callback == 0)
 { /* function is not callback function */
   if ((return address - start address of code segment) < size
        of code segment)
     then normal return
   else
   {
     Buffer Overflow Detected ;
     Process Terminate ;
   }
   } /* end of it */
  else { /* function is callback function */
        normal return
      }
}
```

(그림 10)

6. 실험 결과

버퍼오버플로우 공격을 탐지하기 위한 방법은 실행시간에 많은 오버헤드를 지닌다. 버퍼오버플로우 공격 탐지 코드를 수반하기 전보다 함수의 도입부와 말단부에서 더 많은 일을 하기 때문이다. <표 1>은 3장에서 소개한 기존의 방법의 성능을 평가한 것이다. 이 실험은 실행시간의 오버헤드를 알아보기 위해서 3가지 복귀타입을 가지는 함수를 500,000,000번 호출하여 탐지기능을 첨부하기 전보다 얼마나 많은 오버헤드를 지니는지 알아보았다. 식 (1)은 실험에 사용된 오버헤드를 도출하기 위한 식을 보여준다[2, 3].

탐지코드를 추가하지 않은 경우보다 약 116.5%의 오버헤드를 지닌다. 이것은 프로그램이 실행하는 동안 함수의 호출,복귀가 일어날 때마다 함수의 도입부와 말단부에서 탐지기능을 첨부하기 전보다 더 많은 일을 하기 때문이다.

$$\text{weight} = \frac{\text{a function's additional performance cost associated with Defense Method}}{\text{the function's original performance overhead}} \quad (1)$$

〈표 1〉

함수의 리턴타입	오리지널 시간	3장의 제시한 방법	오버헤드
Void inc()	12,814,378	30,897,126	1.41
Void inc(int *)	17,334,197	35,418,721	1.043
Int inc(int)	18,089,581	36,924,768	1.041

〈표 2〉

함수의 리턴타입	오리지널 시간	새로 제안된 방법	오버헤드
Void inc()	211,400	223,200	0.056
Void inc(int *)	261,000	273,000	0.046
Int inc(int)	296,700	301,400	0.016

기존의 방법과 본 논문에서 새로 제시한 방법을 비교하기 위해서 기존의 방법과 같은 방법의 실험을 진행하였다. <표 2>는 실험결과를 보여준다. 새로 제안된 방법의 경우 약 3.9%의 오버헤드를 지닌다. 기존의 방법보다 많은 성능향상을 보이는데 이것은 함수가 호출이 일어날 때 기존의 방법의 경우 함수의 도입부분에서 함수의 복귀주소의 복사본을 저장하기 위해서 메모리 영역을 Read-Only에서 Writable으로 만들어 주고 나서 저장을 하기 때문에 많은 오버헤드를 지녔었다. 하지만 새로 제안된 방법의 경우 함수의 도입부에서는 탐지하기 전과 같은 일을 하기 때문에 많은 오버헤드를 지니지 않는다. 실험결과를 통하여 새로 제안된 방법의 장점을 알 수 있었다.

첫 번째는 기존의 방법보다 오버헤드가 줄었다는 것이다. 기존의 방법에서는 함수의 도입부분에서 많은 오버헤드를 지녔지만 새로 제안된 방법에서는 이 부분을 사용하지 않고 함수의 말단부에서 지니는 오버헤드만을 수반하고 있으므로 기존의 방법보다 많은 성능향상이 이루어 졌다.

두 번째는 기본의 방법보다 오버헤드가 줄었다. 3장에서 소개한 기존의 방법에서는 함수가 복귀하기 전까지 데이터 영역의 특정한 공간에 복귀주소의 복사본을 계속해서 스택의 위상과 같이 저장하게 된다. 함수의 호출이 복귀 없이 계속해서 일어나는 경우 부가적인 메모리 사용이 필요하게 된다. 하지만 새로 제안된 방법에서는 Pre-execution Time에 실행시간에 사용된 코드영역의 주소와 크기를 실행파일에서 추출하여 저장하게 되고 이 저장공간의 경우 실행시간에는 정적이기 때문에 부가적인 메모리 공간을 필요로 하지 않게 된다.

기존의 방법과 새로 제안된 방법의 실행시간을 비교하여 볼 때 기존의 방법의 실행시간이 약 60여배 느린 것은 실험을 한 환경의 CPU 속도와 메모리 차이 때문이다. 새로 제안된 방법의 경우 인텔 펜티엄 3500Mhz의 속도를 가지는 CPU와 256M의 메모리를 가지는 리눅스 운영체제를 가지는 컴퓨터를 가지고 실험하였고 기존의 방법은 일반 펜티엄 150Mhz의 속도를 가지는 CPU와 64M의 메모리를 가지는 컴퓨터를 이용하여 실험하였다. 하지만 다른 실험 환경을 가지고 실험하였어도 각각의 방법이 가지는 오버헤드는 정확하게 측정하였다. 이러한 실험결과를 통하여 우리는 새로 제안된 방법이 기존의 방법보다 적은 오버헤드를 지니는 것을 알 수 있었고 새로 제안된 방법은 탐지 기능을 첨부하기 전과 거의 차이 없는 성능을 지니는 것을 알 수 있었다.

7. 결론 및 향후 과제

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[2, 3].

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 실행 시에 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 실행 시에 부가적인 메모리 사용이 줄어들게 된다. 실행 속도와 부가적인 메모리 사용에 있어서 더욱 효율적으로 버퍼오버플로우 공격을 탐지할 수 있게 된다.

참 고 문 헌

[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, p.409, 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, San Antonio, TX, pp.63-78, January, 1998.

[4] Aleph One, "Smashing the Stack for fun and profit," Phrack Magazine, 49(14), 1998.

[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.

[6] 김종의, 이성욱, 홍만표, "컴파일러 확장을 이용한 효율적인 버퍼오버플로우 공격 방지 기법", 정보과학회, 2001년도 가을 학술발표, 2001.

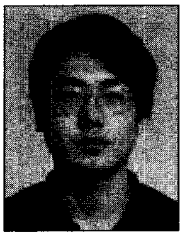
김 종 의

e-mail : paper97i@ajou.ac.kr

2001년 아주대학교 정보및컴퓨터 공학부 졸업(학사)

2001년~현재 아주대학교 정보통신전문대학원 석사과정

관심분야 : 정보보호



이 성 욱

e-mail : wobbler@ajou.ac.kr

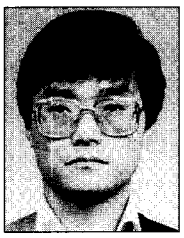
1994년 아주대학교 컴퓨터 공학과 학사

1996년 아주대학교 교통공학과 석사

1996년~1997년 기아정보시스템 지능형 교통시스템팀

1997년~현재 아주대학교 컴퓨터 공학과 박사과정

관심분야 : 병렬처리, 정보보호



홍 만 표

e-mail : mphong@ajou.ac.kr

1981년 서울대학교 계산통계학과 졸업 (학사)

1983년 서울대학교 계산통계학과 졸업 (석사)

1991년 서울대학교 계산통계학과 졸업 (이학 박사)

1983년~1985년 울산공과대학 전자계산학과 전임강사

1985년~현재 아주대학교 정보및컴퓨터공학부 교수

1993년~1994년 미네소타대학 교환 교수

1999년~현재 아주대학교 정보통신전문대학원 교수

2000년~2001년 조지와싱턴 대학교 컴퓨터과학과 교환교수

관심분야 : 병렬처리, 상호 연결망, 컴퓨터 보안