이진 코드 변환을 이용한 효과적인 버퍼 오버플로우 방지기법

김 윤 삼[†] · 조 은 선[††]

요 약

버퍼 오버플로우 공격은 가장 흔하고 위협적인 취약점 중의 하나이다. 최근 이러한 버퍼 오버플로우 공격을 막기 위하여 많은 연구가 이루어지고 있으나 실행시 발생하는 오버헤드 때문에 이를 적용하는 문제가 있다.

본 논문은 이진코드 형태의 파일에서 사용자 정의 함수를 변환하여 리턴 주소의 복사본을 스택의 특정 구역에 저장하고 공격 위험이 있는 문자열 함수를 재작성하고, 재작성된 함수 종료시 리턴 주소와 복사된 리턴 주소의 비교와 ebp 레지스터 값의 비교를 통해 오버플로우 공격을 탐지하는 방법을 제안한다.

키워드 : 버퍼 오버플로우, 이진 코드, 재작성

Efficient Buffer-Overflow Prevention Technique Using Binary Rewriting

Yun-Sam Kim[†] · Eun-Sun Cho[††]

ABSTRACT

Buffer overflow is one of the most prevalent and critical internet security vulnerabilities. Recently, various methods to prevent buffer overflow attacks have been investigated, but they are still difficult to apply to real applications due to their run-time overhead.

This paper suggests an efficient rewrite method to prevent buffer-overflow attacks only with lower costs by generating a redundant copy of the return address in stack frame and comparing return address to copied return address. Not to be overwritten by the attack data, the new copy will have the lower address number than local buffers have. In addition, for a safer execution environment, every vulnerable function call is transformed during the rewriting procedure.

Key Words : Buffer Overflow, Binary Code, Rewriting

1. 서 론

공격 후 루트 권한의 획득이라는 장점과 프로그램 자체의 취약성을 이용한다는 특징에 의하여 스택 기반 버퍼 오버플로우는 최초 보고가 된지 20년이 되고 있지만, 아직까지 해킹에 이용되고 있다. 미국의 CERT[1]에 의하면 전체 보고된 보안 침해 사례 중 버퍼 오버플로우 공격이 차지하는 비중이 50% 정도에 해당된다. 또한 OWASP가 발표한 10대 웹 어플리케이션 보안 리스트[2]에 포함될 정도로 버퍼 오버플로우를 이용한 공격은 아직까지 심각한 상황이다.

이러한 버퍼 오버플로우 공격을 효과적으로 방어를 하기 위하여 많은 연구가 진행되고 있다. 그 중 몇 가지 방법은 Visual Studio등 실제 프로그램 개발 도구에서 이용이 되고 있다. 하지만 이러한 방법들은 특수한 경우에 대하여서는 버퍼 오버플로우 공격을 막을 수 없거나 프로그램의 속도가 느려져 이에 대한 보완이 필요한 상황이다.

본 논문은 이러한 연구들의 단점을 보완하기 위하여 스택에 리턴 주소를 저장함으로써 버퍼 오버플로우를 탐지하는 효과적인 기법을 제안한다. 이를 위하여 레지스터 저장공간과 지역변수 사이에 리턴 주소의 복사본을 안전하게 위치시키고, strcpy함수를 재작성 함으로써 버퍼오버플로우를 탐지한다. 2장에서는 버퍼 오버플로우가 무엇인지 설명하며, 3장에서는 이러한 버퍼 오버플로우 공격을 막기 위해 지금까지 연구되어왔던 방법들을 설명한다. 4장에서는 버퍼오버플로우 공격을 막기 위한 1차적인 방법과 제안된 방법이 지니고 있는 한계점, 그리고 이를 보완하기 위한 추가적인 방법을 소개한다. 5장에서는 제안한 방법들에 의하여 발생하는 오버헤드가 얼마나 크게 작용하는지에 대한 실험 결과에 대하여 기술한다.

※ 본 연구는 학술진흥재단 과제(2003-002-D00312)의 지원에 의하여 수행되었음.
† 준 회 원 : 충북대학교 전자계산학과
†† 정 회 원 : 충북대학교 전기전자컴퓨터공학부 조교수
논문접수 : 2004년 8월 25일, 심사완료 : 2005년 3월 14일

2. 버퍼 오버플로우

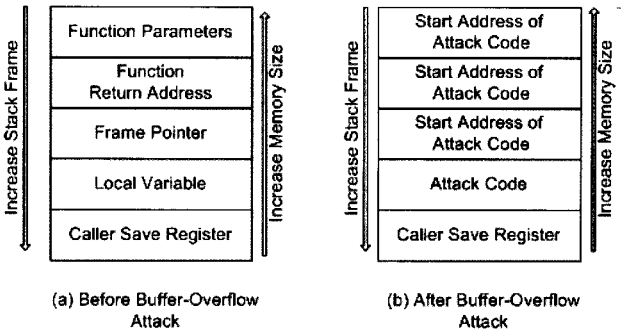
버퍼 오버플로우는 크게 프로그램 내에 악성 코드를 삽입하는 것과 리턴 주소를 조작하여 악성 코드를 실행시키도록 하는 두 가지 목표로 나누어진다. 이러한 버퍼 오버플로우 공격은 오버플로우가 발생하는 위치에 따라 스택 버퍼 오버플로우와 힙 버퍼 오버플로우로 나누어 진다. 힙 버퍼 오버플로우는 악성 코드를 실행시키기 위한 PC값 수정이 힘들다. 하지만 스택 버퍼 오버플로우의 경우에는 악성 코드의 삽입과 PC값의 조작이라는 두 가지 문제를 한번에 해결할 수 있기 때문에 대부분의 공격은 이를 이용한다.

스택 오버플로우 공격은 C언어의 문자열 관련 함수 중 일부가 오버플로우에 대한 취약성을 가지는 점을 이용한다. 이러한 함수들은 입력 문자열을 버퍼에 저장을 할 때 경계조건 검사를 하지 않는다. 또한 경계조건을 검사하는 함수들도 프로그래머가 입력한 값을 경계조건으로 이용하기 때문에 프로그래머가 입력 값을 버퍼의 크기보다 큰 값으로 설정할 경우에도 버퍼 오버플로우가 발생하게 된다. (그림 1)과 같이 함수의 입력이 저장을 위하여 할당된 버퍼의 크기보다 큰 경우 입력으로 들어온 문자들은 저장을 위하여 할당된 버퍼 이외의 공간까지 이용을 한다. 만약 공격자가 충분히 큰 문자열을 입력할 경우 문자열 함수는 지역 변수 저장 공간 뿐만 아니라 호출 함수로 돌아가기 위하여 저장된 리턴 주소까지 변경시킨다. 이러한 경우 프로그램이 제대로 실행되지 않으며 보통 프로그램이 강제로 종료된다. 이러한 문자열 함수들의 문제점은 버퍼 오버플로우 공격에 그대로 이용되고 있다.

```
#include <string.h>
int main(void)
{
        char buf1[30]="This is sample program"
        char buf2[10];
        strcpy (buf2,buf1);    //buffer overflow is occurred
        return 0;
}
```

(그림 1) 버퍼의 크기보다 입력의 크기가 큰 경우

스택 버퍼 오버플로우를 발생시키는 가장 흔한 방법은 (그림 2)의 (b)와 같은 형태로 스택 프레임을 조작하는 것이다. 취약성이 존재하는 문자열 함수를 사용할 때 입력 문자열에 악성 코드를 삽입함으로써, 스택에 악성 코드를 위치시킴과 동시에 버퍼 오버플로우를 이용하여 리턴 주소를 변경시킨다. 이 경우 변경되는 값을 악성 코드의 시작위치로 설정하게 되면 정상적으로 호출 함수로 제어권이 넘어가지 않고 삽입된 악성 코드가 실행되게 된다. 악성 코드는 보통 exec ("/bin/sh")과 같은 간단한 쉘 실행 명령어로 구성된다. 버퍼 오버플로우가 발생하여 쉘이 실행될 경우 쉘은 공격당한 프로그램이 가진 권한과 같은 권한을 가지게 된다. 즉, 버퍼 오버플로우 공격을 당한 프로그램이 root권한에서 수행되었다면 버퍼 오버플로우에 의하여 실행되는 쉘의 권한도 root가 되어 시스템에 큰 피해를 입힐 수도 있다. 또한 Slammer Worm과 같이 프로그램 자체의 버그와 버퍼 오버플로우를 이용하여 불특정 다수에게 웜을 전파할 수도 있다[3].



(그림 2) 버퍼 오버플로우에 의한 스택의 변화[6]

3. 관련 연구

3.1 소스코드가 주어진 경우의 대응

버퍼 오버플로우 공격을 막는 방법은 크게 소스 코드가 존재하는 경우와 존재하지 않는 경우가 있다. 소스 코드가 존재하는 경우는 크게 소스 코드를 스캔하여 취약점을 찾아내는 방법과 커널 및 라이브러리 파일을 수정하여 버퍼 오버플로우 공격을 막는 방법 등으로 나눌 수 있다.

이 중 소스를 스캔하는 방법 중 대표적인 것으로 ITS4[4]를 들 수 있다. ITS4는 소스 코드를 스캔하여 버퍼 오버플로우의 발생 가능성을 탐지한다. ITS4는 스캔을 통하여 얻어진 정보를 바탕으로 버퍼 오버플로우가 발생할 수 있는 함수들을 분류한다. 그 후 각 함수들의 위험성을 단계별로 나누어 프로그래머에게 취약성을 지닌 부분과 그 취약성의 위험성을 알려준다. 이러한 방법은 소스 코드를 통하여 취약성을 검사하여 보고된 부분들을 수정여부를 프로그래머에게 전적으로 위임하기 때문에 소스 스캔에 의한 실행파일의 실행시간에서의 오버헤드는 발생하지 않는다. 하지만, 이러한 방법의 경우에는 취약성이 존재하는 함수에 대하여 위험성을 분류할 때 취약성이 없는 함수가 취약성이 있다고 판별하거나(false alarm) 취약성이 있는 함수를 취약성이 없다고 판별하는 경우가 발생할 수 있다. 때문에 이러한 취약성 보고에 대하여 프로그래머가 일일이 조사를 하여 프로그램에 적용할 것인지에 대하여 결정해야 하기 때문에 프로그램 개발에 대한 오버헤드가 발생한다. 이러한 소스 스캔 방법은 동적 분석 방법들과 결합하여 인터넷 패킷이 버퍼 오버플로우 공격을 일으킬 위험성이 있는지에 대한 연구에도 이용되고 있다[5].

버퍼 오버플로우를 막기 위하여 많은 방법들은 컴파일러를 수정하고 있다. 이 중 가장 대표적인 방법으로는 StackGuard를 들 수 있다[6]. StackGuard는 컴파일러를 수정하여 스택프

레임의 리턴 주소 저장 공간과 스택 프레임 사이에 캐너리 워드를 삽입한다. 이 방법은 스택 버퍼 오버플로우가 발생할 경우 문자열 함수에 의하여 변경이 되는 부분이 지역변수 저장 공간과 리턴 주소 저장 공간 뿐 만 아니라 그 사이에 위치하는 모든 값들 또한 변한다는 점에 착안한다. 만약 중간에 삽입된 캐너리 워드가 변경될 경우 버퍼 오버플로우가 발생하였다고 판단하여 프로그램을 종료시킨다. 이 때 삽입되는 캐너리 워드는 일정한 값이 삽입될 경우 디버깅 모드를 이용하면 쉽게 유추가 가능해지므로 캐너리 워드는 임의의 값으로 생성된다. StackGuard는 대부분의 버퍼 오버플로우를 막을 수 있지만 두 가지 문제점을 가지고 있다. 첫째, 공격자가 캐너리 워드 값을 알아내는 경우이다. 공격자가 몇 번의 버퍼 오버플로우 공격을 통하여 캐너리 워드 값을 알아내어 캐너리 워드가 삽입된 곳에 같은 값을 입력하는 경우 버퍼 오버플로우 공격을 막을 수 없다. 또한 아예 캐너리 워드 값을 건너 띄는 경우도 캐너리 워드는 변하지 않았지만 실제적인 리턴 주소는 변하기 때문에 버퍼 오버플로우 공격을 탐지할 수 없다.

StackGuard와 유사한 방법으로 Visual Studio .Net에서는 컴파일 시 /GS옵션을 이용한다[7]. 이 옵션을 이용할 경우 쿠키가 StackGuard의 캐너리 워드와 같은 위치에 삽입된다. 캐너리 워드는 임의의 값으로 스택 프레임에 삽입되지만 쿠키는 리턴 주소들을 xor한 값이 된다. 이러한 방법은 StackGuard에 비하여 값을 생성하고 비교하는데 걸리는 오버헤드가 적게 걸린다. 하지만 모든 리턴 주소를 알아낼 경우 쿠키 값까지 같이 알아낼 수 있다는 단점이 있다.

또한 국내에서 리턴 주소값의 위치를 이용하여 버퍼 오버플로우를 탐지하는 기법도 연구되었다[8]. 이 기법은 버퍼 오버플로우 공격에 사용되는 악성 코드가 대부분 버퍼에 존재한다는 것에 착안하여 실행 주소가 스택일 경우 버퍼 오버플로우가 발생하였다고 판단한다. 이 방법은 단지 리턴 주소가 나타내는 곳이 스택인지 아닌지 만을 조사하여 버퍼 오버플로우 공격을 탐지하기 때문에 프로그램 자체의 오버헤드가 적게 걸린다. 스택의 제한적 사용은 유닉스나 리눅스에서도 스택에서의 실행을 금지하는 옵션을 둠으로써 유사하게 구현을 하고 있다. 하지만, Lisp나 Object C의 Nested Function, 예외 핸들러의 경우에는 스택에서 실행해야 하지만 스택에서의 실행을 금지할 경우 이러한 정상적인 동작조차 버퍼 오버플로우 공격이라고 판단하게 된다. 때문에 단순히 리턴 주소를 코드 부분으로 한정시키는 것은 문제가 있다[9].

이외에 소스 코드가 존재하는 경우 버퍼 오버플로우를 막기 위해 StackGhost[10], Wagnel *et al.*의 기법[11] 등 다양하게 연구들이 진행되고 있다.

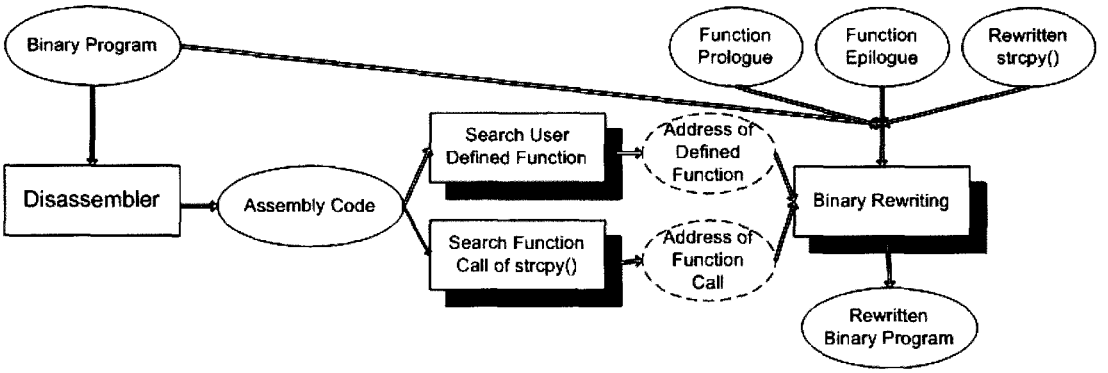
3.2 소스코드의 부재시 적용되는 기법

윈도우용 실행 프로그램이나 유닉스나 리눅스 등의 운영체제에 패키지의 형태로 설치되는 프로그램의 경우에는 소스 코드를 제공하지 않는 경우가 많다. RAD[12]는 소스 코드가 없는 경우 이진파일에서 얻어낸 역어셈블 정보를 바탕으로 하여 이진 파일을 수정한다. 즉, 함수 시작 시 리턴 주소를 .RAR이라 불리는 임의의 힙 메모리에 저장하여 함수가 끝난후의 리턴 주소와 일치하는지 조사하여 버퍼 오버플로우를 탐지한다. 만약 .RAR에 저장되어 있는 값들 중 리턴 주소와 일치하는 값이 없는 경우 버퍼 오버플로우가 발생하였다고 판단을 하여 프로그램을 종료한다. 이 .RAR 공간은 힙 메모리에 위치하기 때문에 원칙적으로 모든 프로그램에서 접근이 가능하다. 만약 다른 프로그램에서 .RAR에 저장되어 있는 값을 임의로 변경하게 되면 함수의 리턴 주소가 정상적임에도 불구하고 버퍼 오버플로우가 발생하였다고 판단할 수가 있기 때문에, .RAR의 처음과 마지막 공간을 VirtualProtect()라는 Win32 API함수를 사용하여 제한된 프로그램이 접근하는 것을 방지한다. 하지만 이 VirtualProtect()나 MemProtect()와 같이 메모리의 상태를 제어하는 함수는 함수 자체에서의 여러 가지 복잡한 과정이 필요하며 메모리 접근이 발생할 때마다 권한에 대한 설정을 해줘야 하기 때문에 막대한 오버헤드가 발생한다. 또한 프로그램 내에 .RAR 공간에 값을 저장하고 .RAR과 리턴 주소가 맞는 것이 있는지 조사하는 루틴도 같이 추가시켜줘야 한기 때문에 오버헤드의 경우 문제가 될 수 있다.

4. 제안 프로그램

4.1 기본 개념

이러한 소소코드가 없는 이진파일을 수정하여 버퍼 오버플



(그림 3) 재작성 단계

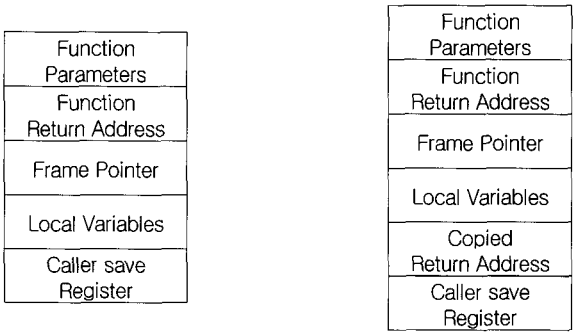
로우 공격을 잡는 기법들은 아직 무시할 수 없는 오버헤드를 가진다.

본 논문에서는 커널 및 컴파일러 그리고 소스들의 코드가 공개되지 않는 상황에서 버퍼 오버플로우 공격을 막는 것을 목적으로 하는 효율적인 기법을 제안한다. 이진 코드는 (그림 3)과 같은 과정을 통하여 버퍼 오버플로우 공격에 대해 안전하게 재작성된다. 먼저 이진 코드에서 재작성에 필요한 정보를 얻어내기 위해 역어셈블러를 이용하여 역어셈블 소스를 생성시킨다. 이렇게 얻어진 어셈블 소스를 이용하여 사용자 정의 함수의 시작과 끝의 위치와 그리고, strcpy함수 콜의 위치를 찾는다. 찾아진 함수는 프로그램에 삽입된 에필로그와 프롤로그 및 재작성된 strcpy함수의 호출로 바꿈으로써 이진 코드 재작성이 이루어진다. 함수 프롤로그와 재작성된 strcpy은 버퍼 오버플로우가 발생했는지 탐지한다. <표 1>은 버퍼 오버플로우를 막기 위한 이진 파일 재작성의 절차를 간단히 나타낸 것이다.

〈표 1〉 전체 프로그램의 재작성 절차

1. 역어셈블[14]
2. 함수의 시작 및 끝을 조사
3. 함수의 시작과 끝을 프롤로그와 에필로그로 이동하는 코드로 재작성
4. strcpy()의 재작성
5. strcpy()가 사용되는 경우 버퍼가 ebp값보다 큰지 조사
6. 버퍼오버플로우 발생 여부 조사 및 발생시 프로그램 종료

제안된 기법은 RAD와 유사하게 버퍼 오버플로우를 방지하기 위해 이진 실행 파일을 수정한다. 하지만 제안 기법의 특성에 의하여 RAD에 비해 리턴 주소 사본을 관리하는 오버헤드를 줄일 수 있다. 즉, RAD가 힙 메모리에 .RAR라는 별도의 저장 공간을 생성하여 리턴 주소의 복사본을 관리하는 반면 제안 기법은 (그림 4)와 같이 스택 프레임을 사용한다. 복사된 리턴 주소는 안정성을 위하여 지역 변수와 레지스터 저장 공간에 위치한다. 버퍼 오버플로우가 발생할 경우 문자열은 메모리 주소가 큰 쪽으로 기록하기 때문에 복사된 리턴 주소는 변경되지 않는다.



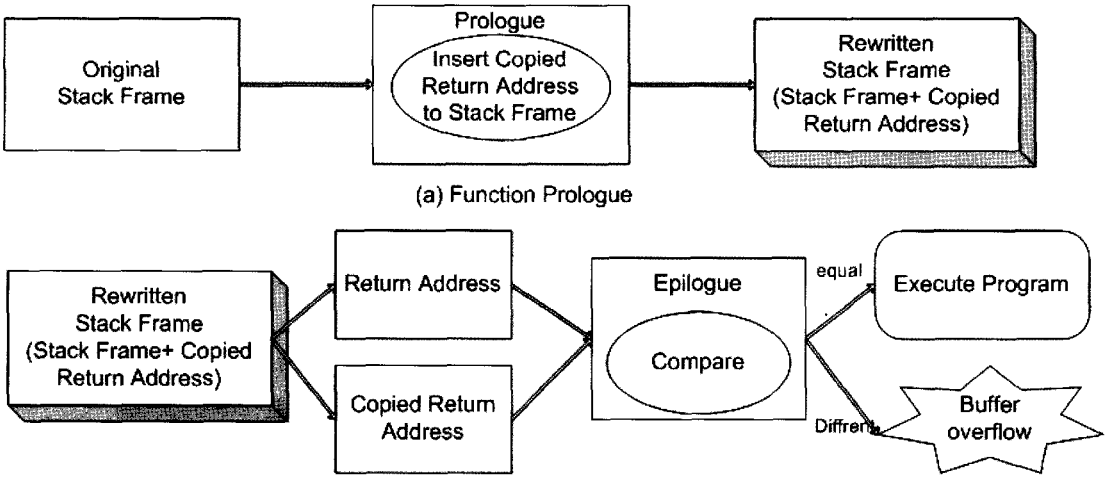
(a) 기존 스택 프레임의 구성 (b) 제안하는 스택 프레임

(그림 4) 제안하는 스택 프레임

4.2 사용자 정의 함수의 재작성

버퍼 오버플로우는 스택의 리턴 주소를 변경시키기 때문에 이를 막기 위해서는 사용자 정의 함수를 재작성해야 한다. 이때 사용자 정의 함수 내에 버퍼 오버플로우를 탐지하는 부분을 삽입하여야 한다. 하지만, 각각의 사용자 정의 함수는 10byte 이내의 작은 공간으로 구분이 되기 때문에 이진 파일의 경우 사용자 정의 함수 내에 탐지 기법을 삽입할 수 없다. 때문에 사용자 정의 함수는 (그림 5)와 같이 리턴 주소의 복사본을 삽입하는 프롤로그와 리턴 주소를 비교하는 에필로그를 호출하는 방식을 사용한다.

사용자 정의 함수에 삽입되는 프롤로그는 (그림 6)처럼 JMP 명령어에 의하여 덮어 쓰인 기존 함수의 명령어와 리턴 주소를 리턴 복사 공간으로 복사하는 부분 그리고, 기존의 함수로 돌아가는 함수로 이루어진다. 또한 에필로그는 (그림 7)



(그림 5) 사용자 정의 함수의 재작성 및 버퍼 오버플로우 탐지

과 같이 리턴 주소와 리턴 복사 공간의 값을 비교하여 같을 경우 전체 스택 크기를 감소시킨 후 호출 함수로 이동을 시킨다. 만약 두 값이 다를 경우 프로그램은 종료된다.

```
push esp
mov ebp, esp
Sub esp, 044                  //스택 프레임 크기를 4만큼 더 증가
                              //스택 복사 공간을 할당하기 위해
Mov eax, dword[esp-0C]        //리턴 주소를 레지스터에 저장
Mov dword[esp-04], eax        //레지스터에 저장된 리턴
                              //주소를 스택 복사 공간에 저장
Call 004010A6
```

(그림 6) 삽입되는 함수 프롤로그

```
move  eax, dword[esp-0c]  //리턴주소 공간의 값을 저장
cmp  eax, dword[esp-04]   //리턴주소와 리턴주소 복사 공간값
                            비교
jne   09                  //다를 경우 프로그램 종료
mov  esp, ebp             //스택 감소 과정
pop  ebp
ret
```

(그림 7) 삽입되는 함수 에필로그

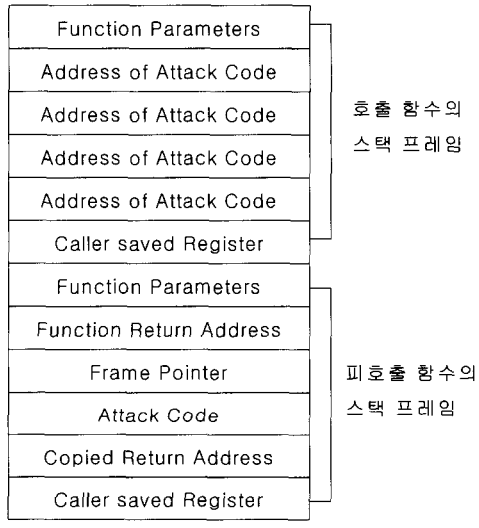
함수 프롤로그와 에필로그를 삽입하게 위하여 기존에 존재하는 사용자 정의 함수는 (그림 7)과 같이 이진 코드에서 존재하는 6바이트 크기의 코드를 프롤로그와 에필로그로 이동하는 명령어로 각각 대체된다. (그림 8)의 (1)은 스택의 크기를 증가 시키는 부분이 프롤로그 부분으로 이동하는 부분으로 대체하는 것을 나타낸다. 또한 (그림 8)의 (2)는 스택 크기를 감소시키고 호출 함수로 되돌아가는 부분이 에필로그로 이동하게 된다.

프로그램은 재작성 기법을 이용하여 함수가 끝난 후 리턴 주소와 리턴 주소 복사 공간의 값이 서로 다를 경우 버퍼 오버플로우가 발생하였다고 판단을 한다. 버퍼 오버플로우가 발생할 경우 프로그램을 공격이 일어나기 전 상황으로 돌리는 것을 실제적으로 거의 불가능 하므로 프로그램은 강제 종료된다.

제안된 기법을 사용할 경우 리턴주소의 사본을 관리하기 위하여 스택에 리턴 주소 복사 공간을 생성하여 버퍼 오버플로우 발생을 탐지하기 때문에 힙 메모리에 특별한 공간을 만들 필요가 없다. 또한 스택은 악의적으로 메모리를 조작하지 않는 경우 다른 프로그램이나 함수에서 기존 함수의 스택 부분을 수정할 수 없기 때문에 VirtualProtect()나 MemProtect()와 같은 오버헤드가 큰 함수를 이용하지 않는다. 따라서 RAD등에서 발생하는 복사된 리턴주소의 관리에 소요되는 비용을 최소화 시킬 수 있다. 또한 저장된 리턴 주소 값을 찾을 때 모든 주소 값을 비교 하지 않고 자신의 스택프레임에 복사된 리턴주소만을 이용하여 주소 값을 비교하기 때문에 리턴 주소 탐색에 의해 소모되는 시간 또한 줄일 수 있다. 리턴 주소 복사 공간으로 할당되는 공간은 스택 프레임이 할당될 때 추가로 4바이트를 할당되기 때문에 메모리를 일일이 할당하거나 반환할 필요가 없다.

4.3 strcpy() 재작성 기법

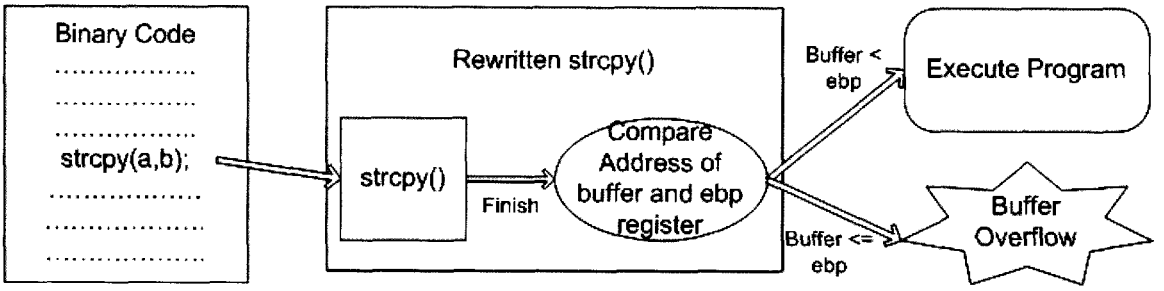
리턴주소 복사 공간을 이용하는 경우 피호출 함수의 스택 프레임과 문자열 함수의 특성을 이용하기 때문에 대부분의 버퍼 오버플로우 공격을 효과적으로 막을 수 있다. 하지만, 이렇게 리턴 주소를 위하여 스택 프레임을 이용하는 대부분의 경우에는 한 가지 문제점을 가진다[13]. 공격자가 스택프레임 값을 알아내어 리턴 주소를 건드리지 않고 호출 함수의 스택프레임만 변화시키는 경우 버퍼 오버플로우를 감지할 수 없다. (그림 9)와 같이 상위 스택 프레임이 변화되지 않은 상태에서 하위 스택 프레임의 리턴 주소 값과 복사된 리턴 주소 공간 값이 동일하게 변화되는 경우, 피호출 함수에서는 리턴 주소와 리턴 주소 복사 공간이 같기 때문에 버퍼 오버플로우를 탐지할 수 없다.



(그림 8) 리턴 주소 복사 공간 이용 기법의 취약성

이러한 문제점 때문에 단순히 사용자 정의 함수를 조작하는 것만으로는 함수의 안정성을 완벽하게 보장할 수 없다. 함수의 안정성을 보장하기 위해서는 호출 함수의 안정성을 증명하거나 피호출 함수에 의하여 호출 함수의 리턴 주소가 변경되지 않는 방법을 선택하여야 한다. 본 논문에서는 안정성 문제를 해결하기 위하여 문자열 함수를 재작성하여 호출 함수의 리턴 주소가 변경되지 않는 방법을 사용한다.

이진 실행파일은 문자열 버퍼의 경계를 나타내는 정보를 가지고 있지 않기 때문에 피호출 함수의 리턴 주소의 변경을



(그림 9) 재작성된 strcpy에 의한 버퍼 오버플로우 검사

막는 것은 쉽지 않다. 그러나 레지스터 정보를 이용하면 호출 함수의 리턴 주소와 지역변수가 변경되는 것을 막을 수 있다. 컴퓨터의 많은 레지스터 중 두개의 레지스터는 스택 프레임의 주소를 가진다. 이중 esp 레지스터는 가장 최근에 적재된 스택 프레임의 주소를 저장한다. 그리고 ebp 레지스터는 바로 전의 스택 프레임의 주소를 저장한다. 이 중 esp레지스터는 함수의 크기에 따라서 스택 프레임의 크기도 변하기 때문에 피 호출 함수의 리턴 주소의 변경을 체크하기가 힘들다. 그러나 ebp 레지스터를 이용할 경우 호출 함수의 레지스터 저장 공간이 변경되었는지는 체크 할 수 있다.

ebp 레지스터 값을 문자열 함수에 이용할 경우, 재작성된 문자열 함수는 호출 함수의 스택 프레임까지 버퍼 오버플로우 공격에 의하여 변경되는 것은 막을 수 있다. 그러나 ebp 레지스터를 이용한 문자열 함수의 재작성만으로는 피호출 함수에서의 버퍼 오버플로우를 알기 어렵기 때문에 4.1에서 제안된 기법과 같이 사용되어야 버퍼 오버플로우 공격을 막을 수 있다.

```
:004107B6 668917          mov word[edi], dx
:004107B9 8B442408        mov eax, dword[esp+08]
:004107BD 39FC            cmp ebp, edi
:004107BF 7D30            jge 004107F1
:004107C1 5F              pop edi
:004107C2 C3              ret
---------
:004107C3 668917          mov word[edi], dx
:004107C6 8B442408        mov eax, dword[esp+08]
:004107CA 39FC            cmp ebp, edi
:004107CC 7D30            jge 004107FE
:004107CE 5F              pop edi
:004107CF C3              ret
```

(그림 10) 재작성된 strcpy 함수의 일부분

strcpy()의 재작성 절차는 (그림 10)과 같다. 기존의 strcpy()는 널 문자의 위치에 따라서 널 문자 앞에 위치한 문자열을 삽입하고 strcpy()를 호출한 함수로 되돌아간다. 그러나 재작성된 strcpy()는 호출한 함수로 돌아가기 전 삽입된 버퍼의 위치와 ebp레지스터 값을 비교한다. 만약 버퍼의 주소 값이 ebp 레지스터의 값보다 큰 경우 버퍼 오버플로우라 판단한다. 이는 기존 함수에서 함수 종료 이전에 레지스터 비교의 간단한 명령어만이 추가되어 경계 조건 검사에 의한 오버헤드는 거의 발생하지 않는다. (그림 11)은 재작성된 함수의 일부분을 보여준다.

5. 실 험

5.1 사용자 정의 함수 재작성 기법의 오버헤드

제안된 기법들의 성능을 실험하기 위해 AMD 2400+, RAM 512MB, 윈도우즈 2003 환경에서 테스트를 하였다. 실행 프로그램은 리턴 주소 복사 공간 기법에 의한 오버헤드의 발생을 알아내기 위하여 1부터 50,000,000까지 카운트 하는 서로 다른 4개의 실행 파일을 생성하였다[6]. 첫 번째 실행 파일은 별다른 함수를 사용하지 않고 메인 함수에서 증가시켰다. 나머지 실행 파일들은 각각 전역 변수를 선언하고 증가 함수에서는 별다른 인자를 넘겨 주지 않는 방법(void increment(void))과 포인터로 선언된 인자를 이용하여 증가시키는 함수(void increment(int*) 그리고, 입력 인자와 출력 인자를 이용한 함수(int increment(int)를 사용하여 구현되었다. 또한 이러한 함수들의 오버헤드를 알아내기 위하여 다음과 같은 식을 사용하였다.

$$overhead = \frac{function's\ additional\ cost}{original\ function\ overhead} X100$$

(식 1) 오버헤드 계산 식[6]

리턴 주소 복사 공간을 이용한 기법의 오버헤드를 판단하기 위하여 3장에서 언급한 StackGuard ,RAD 및 "버퍼오버플로우 공격 방지를 위한 컴파일러 기법"[8]과 성능 비교를 하였으며, 그 결과는 <표 2>와 같다. RAD는 VirtualProtect의 사용과 .RAR에 의한 오버헤드가 매우 크게 나타났으며, 본 기법은 StackGuard보다 최대 1/10정도 수준인 15%정도의 오버플로우가 발생하였다. 또한 전체적 인 오버헤드는 12% 정도가 발생하였다. "버퍼오버플로우 공격 방지를 위한 컴파일러 기법"은 0.1%정도의 오버헤드만 발생하였지만 3.1에서 서술한 바와 같이 기법의 적용에 의한 기존 기술의 사용 불가

〈표 2〉 사용자 정의 함수 재작성 실험 결과의 비교

함수 종류	실행 시간 (ms)	제안 기법 실행 시간 (ms)	제안 기법의 오버 헤드 (%)	Stack Guard의 오버헤드 (%)	RAD의 오버 헤드 (%)	컴파일러 기법[8]의 오버헤드
i++	172	172	0	0	0	0
void increment (void)	1485	1723	16.02	125	141	0.056
void increment (int*)	1508	1691	12.14	69	104.3	0.046
int increment(int)	1547	1721	11.25	80	104.1	0.016

〈표 3〉 Strcpy()함수 재작성 실험 결과

프레임의 길이 (Byte)	실행시간 (ms)	제안된 기법에 의하여 걸린 시간 (ms)	오버헤드 (%)
40	632	641	1.424
100	1034	1046	1.160
200	1874	1883	0.480

는 오버헤드 이상의 문제를 발생시킬 수 있다. 본 논문에서 제안된 기법은 버퍼 오버플로우 공격이 발생했는지에 대한 정보를 리턴 주소를 직접 이용하기 때문에 난수를 생성하는 데서 오는 오버헤드가 발생하지 않았다. 또한 최초 스택 생성 시와 제거시 리턴 주소 복사 주소 공간이 동시에 생성되고 제거되기 때문에 주소 복사 공간의 유지를 위해 발생하는 오버헤드가 최소화되었다.

이러한 결과들은 테스트 파일들이 함수 내에서 1의 증가 작업을 하였을 때 나온 오버헤드이다. 따라서 함수의 코드 수행 시간이 충분히 길어지면 오버헤드 또한 줄어들 것으로 보인다.

5.2 strcpy() 재작성 기법의 오버헤드

문자열 함수의 변환 기법은 지역변수가 사용되지 않는 함수에 의하여 생성되는 스택 프레임의 크기가 40바이트이기 때문에 호출 함수의 스택 프레임의 값을 변화시키기 위해서는 최소한 40바이트의 크기가 필요하다. 또한 단순 문자열 복사를 위한 스택 프레임이 200바이트를 넘지 않는다고 가정하였다. 이에 따라 40바이트와 100바이트, 200바이트로 스택 프레임을 늘리면서 수정된 strcpy함수의 오버헤드를 조사하였다. 그 결과는 <표 3>과 같으며, 최대 약 1.4%정도의 오버헤드만 발생되었다. 수정된 기법의 오버헤드는 레지스터끼리의 비교연산에서만 발생하므로 실제적인 오버헤드는 거의 발생하지 않음을 볼 수 있다.

6. 결론 및 향후 연구 과제

본 논문은 버퍼 오버플로우를 막기 위해 이진 코드 내의 사용자 정의 함수를 수정하여 리턴 주소의 사본을 스택에 안전하게 삽입하고 함수 종료 후 리턴 주소 값과 비교함과 동시에 취약점을 가지는 함수를 재정의하는 기법을 제안하였다. 제안된 기법은 20바이트 이내의 간단한 명령어들을 삽입함으로써 버퍼 오버플로우를 효과적으로 막을 수 있었다. 이 때 발생하는 오버헤드 또한 다른 기법에 비하여 적기 때문에 프로그램에 적용할 수 있을 것으로 기대된다.

제안된 기법은 현재 대단위 프로그램에 적용하여 제안 기법에서 발생 할 수 있는 문제점 및 여타 공격에 의한 취약점에 대한 연구를 추진 중에 있다. 또한 취약성이 존재하는 함수의 재작성을 strcpy 이외의 다른 함수로의 확장 또한 고려 중이다.

참 고 문 헌

[1] Cert coordination center, http://www.cert.org/advisories

[2] OWASP, "The Ten Most Critical Web Application Security Vulnerabilities", http://www.owasp.org/documentation/topten.html

[3] PSS Security Response Team Alert-New Worm: W32.Slammer, http://www.microsoft.com/technet/security/alerts/slammer.mspx

[4] J. Viega, J. Bloch, T. Kohno and G. McGraw, "ITS4: A static vulnerability scanner for c and c++ code", In proceeding of the 16th Annual Computer Security Applications Conference, Dec., 2000.

[5] Eric Gaugh, Matt Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities", In proceedings of the 2003 Symposium on Networked and Distributed System Security, Feb., 2003.

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

[7] Microsoft MDSN, "Compiler Security Checks In Depth", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp

[8] 김종의, 이성욱, 홍만표, "버퍼오버플로우 공격 방지를 위한 컴파일러 기법", 정보처리학회논문지C 제9-C권 제4호, pp. 453-458, 2002.

[9] A. Baratloo, N. Singh and T. Tsai, "Transparent run-time defense against stack smashing attacks", In proceedings

of USENIX Annual Technical Conference, June, 2000.

[10] Make Frantzen, Mike Shuey, "StackGhost: Hardware facilitated stack protection", In 10th USENIX Security Symposium, Aug., 2001.

[11] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities", In symposium on Network and Distributed System Security, pages 3-17, Feb., 2000.

[12] M. Prasad and T. Chiueh, "A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks", In proceedings of the IEEE Symposium on Security and Pricvacy, May, 1996.

[13] Tzi-Cker Chiueh and Fu-Hau Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks", In proceedings of International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April, 2001.

[14] 조 상, Windows disassembler, http://www.geocities.com/mysimpc/

김 윤 삼

e-mail : bijak1@hotmail.com

2004년 충북대학교 컴퓨터전공(학사)

현 재 충북대학교 전자계산학과 재학

관심분야 : 정보보호, Reverse Engineering 등

조 은 선

e-mail : eschough@cbnu.ac.kr

1991년 서울대학교 계산통계학과 계산학 (학사)

1993년 서울대학교 계산통계학과 전산학 (석사)

1998년 서울대학교 전산과학과 (박사)

1999년~2000년 한국과학기술원 연구원

2000년~2002년 아주대학교 정보통신전문대학원 조교수 대우

2002년~현재 충북대학교 전기전자컴퓨터공학부 조교수

관심분야 : 보안, 프로그램 분석 등