리눅스 시스템의 버퍼 오버플로우 공격 대응 기법

김 정 녀[†] · 정 교 일[††] · 이 철 훈[†††]

요 약

본 논문에서는 시스템 해킹 시에 가장 많이 사용되는 버퍼 오버플로우 공격의 기법을 소개하고 운영체제 커널 수준에서 막을 수 있는 대응 기법을 소개한다. 보안 운영체제 시스템이란 리눅스 커널에 접근제어, 사용자 인증, 감사 추적, 해킹 대응 등과 같은 보안 기능을 추가 구현하여 시스템에서 발생 가능한 해킹을 방지하고 차단하는 시스템을 말한다. 본 논문에서는 보안 운영체제 시스템에서 제공하는 해킹 대응 기술 중의 하나로 버퍼 오버플로우 공격을 막을 수 있는 커널 내 구현 내용을 설명한다.

The Blocking of buffer overflow based attack for Linux kernel

Jeong-Nyeo Kim[†] · Kyo IL Chung[††] · Cheolhoon Lee[†††]

ABSTRACT

In this paper, we describe a blocking method of buffer overflow attack for secure operating system. Our team developed secure operating system using MAC and ACL access control added on Linux kernel. We describe secure operating system (SecuROS) and standardized Secure utility and library. A working prototype able to detect and block buffer overflow attack is available.

키워드 : 보안운영체제(Secure Operating System), 버퍼 오버플로우 공격(Buffer overflow attack), 해킹 대응(Anti-Hacking), 접근 제어(Access Control)

1. 서 론

최근 들어 성능이 우수하고, 안정성이 뛰어나며 비용이 절감되는 측면에서 공개 운영체제인 리눅스(Linux)나 Free-BSD 등이 범용화 되어 사용되고 있다. 이러한 개방성은 공개 운영체제들만의 중요한 특징이지만 커널의 원천코드 공개로 인하여 해커들에게 악용될 소지가 많아 웹서버, 메일서버 등과 같은 인터넷 서버로 사용하는 데에 많은 문제점들이 발생하고 있다. 이 문제점들을 해결하기 위한 방법으로 외부로부터의 접근을 차단하여 정보를 보호하는 방화벽, 침입탐지 시스템들과 커널 내부를 수정한 보안 운영체제[1] 시스템들이 제안되고 있다. 보안 운영체제란 운영체제 상에 내재된 보안상의 결함으로 인하여 발생 가능한 각종 해킹으로부터 시스템을 보호하기 위하여 기존의 운영체제 내에 보안 기능을 추가한 운영체제를 말한다. 보안 운영체제를 구성하는 요소 중의 하나로 버퍼 오버플로우나 서비스 거부(Denial of Service) 등과 같은 해킹 기법을 차단할 수 있는 해킹 대응 모듈을 들 수 있다.

CERTCC-KR의 발표에 따르면 2001년 상반기에 국내외에서 접수된 해킹 사고는 총 2,710건으로 2000년 상반기 월 평균 120건에서 2001년 상반기 월 평균 452건으로 약 3.8배 증가 하였다. 연도별 해킹사고 접수 처리 현황을 살펴보면, '97년 64건, '98년 158건, '99년 572건, '00년 1,943건, '01년 6월 현재 2,710건의 해킹사고가 접수되는 등 매년 2~3배씩 증가하고 있다. 또한 2001년 상반기에 나타난 피해시스템의 운영체제 현황을 살펴보면, 개인 PC를 제외한 서버급에서는 리눅스 (36.3%), 윈도우 NT/2000 서버 (5.7%), 솔라이스 (4.6%) 순으로 해킹 사고가 접수되었다. 이렇듯 현재 많은 리눅스 시스템들이 공개 운영체제라는 이유로 가장 먼저 해킹의 대상에 오르고 있으며, 그 중 버퍼 오버플로우 해킹 기법은 시스템의 루트 권한을 획득하는 가장 유용한 기법으로 쓰여지고 있다. 본 논문에서는 근래 유행하는 해킹의 형태를 살펴보고 그 중 버퍼오버플로의 해킹 형태 유형을 파악하며 버퍼오버플로우를 방지할 수 있는 커널 수준에서의 대응 기법을 제시해보고자 한다.

본 논문의 구성은 2장에서 현재까지 최신 해킹 기법과 리눅스 시스템의 버퍼오버플로우 취약성을 살펴보고, 3장에서 버퍼오버플로우의 원리를 알아보고, 4장에서 개발된 버퍼

† 정 회 원 : 한국전자통신연구원 보안운영체제 연구팀장(선임연구원)
†† 정 회 원 : 한국전자통신연구원 정보보호기술연구본부 정보보호기반연구부장(책임연구원)
††† 정 회 원 : 충남대학교 컴퓨터공학과 교수

논문접수 : 2001년 10월 4일, 심사완료 : 2001년 12월 22일

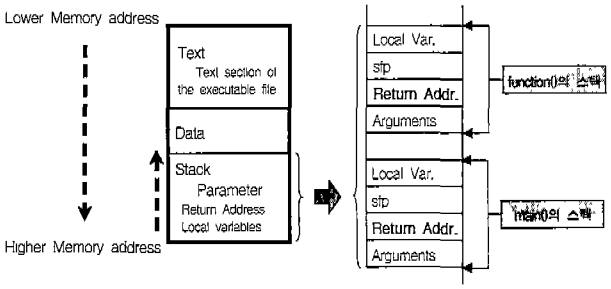
오버플로우 대응 기법의 전체적인 모습도를 그려보고 5장에서는 결론과 향후 연구 방향에 대해 논의한다.

2. 최신 버퍼 오버플로우 기술

버퍼 오버플로우는 시스템 해킹 기법 중에 가장 널리 알려진 기법이다[3]. 이러한 버퍼 오버플로우도 가장 손쉽게 사용할 수 있는 스택 오버플로우 기법부터 스택 오버플로우의 일종이지만 조금 쉽지 않은 Frame Pointer Overwrite나 Format String Attack 등이 있다.

2.1 버퍼 오버플로우 원리

버퍼 오버플로우 기법의 원리를 알려면 먼저 프로세스의 메모리 구조를 알아야 한다. 다음 메모리 상의 구조를 나타내는 (그림 1)이다.



(그림 1) 프로세스의 메모리 구조

프로세스가 사용하는 메모리 영역은 한정된 메모리의 한계를 극복하기 위하여 가상메모리(Virtual Address) 방식을 사용한다. 서로 다른 프로세스들이 분리된 주소공간을 가질 수 있도록 하며, 이는 운영체제가 가지고 있는 정보에 의해 물리적 주소로 변환되는 형태이다. 그림에서의 메모리 영역은 텍스트, 데이터 그리고 스택으로 구분된다. 텍스트는 프로그램의 인스트럭션 코드와 읽기 전용 모드(read-only) 데이터를 가지고 있으며, 데이터는 읽기, 쓰기가 가능한 지경으로 전역 변수(global variable)나 정적 변수(static variable) 등으로 선언되는 변수들을 나타낸다. 데이터 영역에는 비초기 영역(uninitialized region)으로 불리는 heap 영역도 속한다.

스택은 프로그램의 수행중 함수 호출이 있었을때 호출된 함수에서 사용되는 지역 변수(local variable)와 파라메터 변수(parameter variable), 복귀될 때의 인스트럭션이 있는 주소인 반환 주소(return address) 등을 push 하고 호출한 함수 코드를 실행한다. 함수를 실행하고 난 후에는 반환 주소를 pop 한 후에 해당 주소로 제어를 옮긴다.

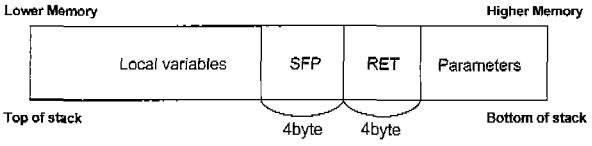
이런 메모리 구조에서 버퍼 오버플로우는 크게 스택 오버플로우와 heap 오버플로우 두 가지 방식으로 나누어진다. 스택 오버플로우는 스택에서 push된 값이 다시 pop되어 복귀되면서 복귀하는 주소를 변경함으로써 본래의 복귀 주소가 아닌 문제의 코드를 실행시키는 주소로 옮겨가면서 일어나게 된다. heap 오버플로우는 데이터 영역에서 프로그램 내에 동적으로 할당된 메모리 영역에서 포인터 영역으로 바꿔서 원하는 부분으로 변경함으로써 공격이 가능하게 된다. heap 오버플로우의 경우는 공격하고자 하는 프로그램마다 포인터나 버퍼의 크기를 추정하기가 힘들므로 스택 오버플로우 보다 공격이 많지는 않다.

2.2 Frame Pointer Overwrite

Frame Pointer Overwrite는 스택 오버플로우의 한 방법으로 Frame Pointer Overwrite는 프로그램 상의 아주 작은 실수도 공격이 대상이 될수 있다는 것을 보여준다. Frame Pointer Overwrite에서는 1바이트의 오버플로우 라는 정교한 공격방법을 다룬다. 배열을 이용해서 프로그램을 작성할 때 실수하는 부분이 배열의 index가 0부터 시작한다는 것을 잊어버린다는 것이다. 이 프로그램의 문제점은 배열의 크기를 계산할 때 실수를 하여 1바이트를 더 계산함으로써 1바이트 메모리의 내용이 Overwrite될수 있도록 하였다. 이것이 어떠한 의미를 가지는 것인지는 아래의 내용을 살핀 후에 자세하게 설명하도록 한다.

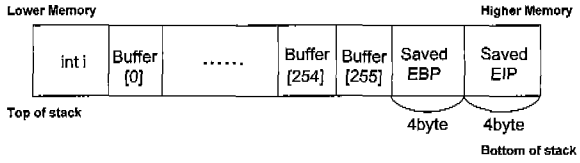
```
#include <stdio.h>

func(char *sm)
{
    char buffer[256];
    int i;
    for(i = 0; i <= 256; i++)
    buffer[i] = sm[i];
}

main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("missing args\n");
        exit(-1);
    }
    func(argv[1]);
}
```

C에서 함수가 호출되는 경우는 Assembly 언어에서는 CALL이라는 명령어가 호출되는 것과 마찬가지이다. 아래의 그림은 C에서 함수가 호출되었을 때 스택의 모습을 나타낸 것이다.



(그림 2) 함수 호출 시 스택 모습도

다음 그림은 vul.c에서 func()가 호출되었을 때 스택의 모습이다.



(그림 3) func() 호출시의 스택 모습도

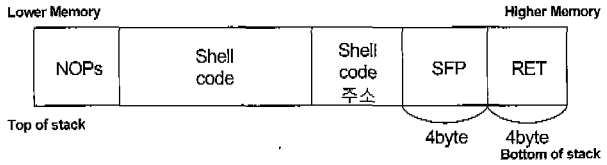
함수가 호출될 때 Return 주소에 해당하는 EIP를 스택에 push한다.

다음으로 아래의 명령어를 실행하여 local frame을 활성화시킨다.

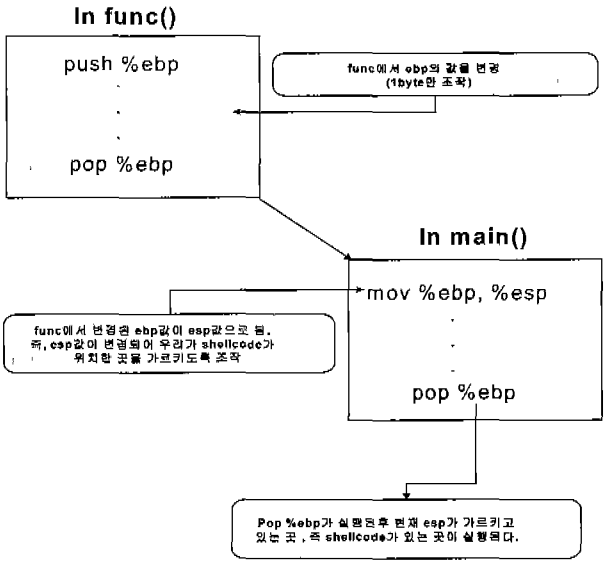
Frame Pointer Overwrite는 기존의 stack-based overflow와는 다르게 EIP즉 RET를 직접 수정 할 수는 없다. 그러나 Frame Pointer Overwrite를 이용하여 프로세서가 EIP의 위치를 착각하도록 하여 우리가 원하는 곳으로 프로그램의 흐름을 바꿀 수 있다.

프로세서가 프로시저로부터 돌아올 때, 스택에 있는 첫 번째 워드를 단순히 꺼낸다.

아래의 그림은 우리가 frame을 overwrite해서 우리의 shellcode가 실행되도록 하기 위해 우리가 만들어야 할 stack의 layout이다.



(그림 4) shellcode가 삽입된 stack layout



(그림 5) ebp의 변경을 통한 shellcode의 실행 과정 흐름도

처음 "NOP + shellocode + shellcode의 주소 + overwrite 할 1바이트"를 담고 있는 버퍼를 func의 버퍼에 써넣는다. 그러면 <shellcode가 삽입된 stack layout>과 같이 구성된다.

한 바이트가 overwrite되는데 1바이트가 overwrite되고 난 이후에 ebp의 값은 shellcode의 주소가 들어 가 있는 공간의 주소보다 4가 작은 값이 들어가 있다. 이유는 func가 실행을 main에게 넘긴 후 main에서는 mov %ebp, %esp로 ebp의 값을 esp에 넘기고 pop된다.

ebp가 pop되면 esp는 4만큼 증가시키므로 사전에 func에서의 ebp의 값을 4작게 설정하게 되면 main에서는 4가 저절로 증가되게 되어 esp는 shellcode의 주소를 가진 곳의 주소를 가르키게 된다.

2.3 Format String Attack

Format String Attack은 작년, 2000년에 소개되어 현재 많은 프로그램에서 FSA에 대한 취약점이 발견되고 있다. Format string attack은 Tim Newsham이 쓴 "Format String Attacks"에서 처음 소개되었다. FSA가 발표되기 이전에는 BoF에 취약한 프로그램들에 대한 공격이 주를 이루었으나 FSA 발표이후 많은 exploit code가 발표되고 있다. FSA는 BoF에 비하여 난이도가 높지만 이미 많은 취약점이 발견되고 있다. printf()라는 함수는 C 프로그램에서 흔히 볼 수 있는 함수이다. (그림 6)에는 printf()를 이용하여 문자열을 출력하는 예를 보였다.

```
char *buff = "Hello~~~~^_ _____^\n" ;
printf("%s", buff) ;  /* 1 */
printf(buff) ;        /* 2 */
printf("hello world\n", 1) ; /* 3 */
```

(그림 6) printf()의 사용 예

(그림 6)에서 볼 수 있듯이 문자열을 출력하는 방법은 두 가지가 있다. printf()에 format string을 표시하여 출력하는 방법 /* 1 */과 다른 하나인 출력할 메모리의 주소를 직접 쓰는 방법/* 2 */이 있다. (그림 6)에 있는 printf()들이 사용한 format string을 살펴보면, /*1*/의 경우 "%s", /* 2 */의 경우 직접 buffer를 사용, /* 3 */의 경우 "hello, world\n"가 된다.

Format string attack의 공격대상이 되는 것은 /* 2 */에 나타나 있는 것과 같이 사용했을 때이다. 이것은 실수라고 말하기보다는 프로그래머의 프로그래밍 스타일에서 오는 문제로 FSA와 같은 공격 방법이 발표되었기 때문에 /* 2 */와 같은 프로그래밍 스타일은 가급적 피하는 것이 좋다. printf의 문제점을 보여주기 위해 아래의 프로그램을 살펴보도록 하자.

```
Main()
{
    char name[30] ;
    scanf("%s", name) ;
    printf(name) ;
    printf("\n") ;
}
```

(그림 7) 취약 프로그램

```
[sccureos1 : bret : /home/bret 71 ]
# Go Go Go ^^ 71 # a.out
AAAA%x%x%x%x%x%x%x
AAAAff23b564ff2339380ff233938ff1a01c00ffbefb70 /* 1 */
[secureos1 : bret : /home/bret 72 ]
# Go Go Go ^^ 72 # a.out
AAAA%x%x%x%x%x%x%x%x%x
AAAAff23b564ff2339380ff233938ff1a01c00ffbefb701414141412578 /* 2 */
[secureos1 : bret : /home/bret 73 ]
# Go Go Go ^^ 73 #
```

(그림 8) (그림 7)의 실행 결과

scanf의 입력으로 "AAAA"로 시작하는 문자열과 "%x"라는 문자열을 여러 개 주었을때 프로그램의 실행 결과가 (그림 8)에 나타나 있다. 결과를 보면 꽤 많은 문자열이 출력되어 있는 것을 볼 수 있다. 이것은 "%x"라는 것이 printf() 내에서 16진수를 나타내는 format string으로 인식되어 메모리에 있는 내용을 출력하는 것이다.

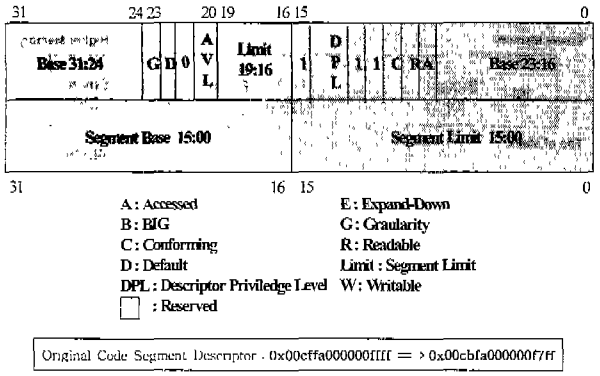
3. 버퍼 오버플로우 공격 대응 기법

이 논문에서 제시하고자 하는 기법은 위의 모든 버퍼 오버 플로우 기법에서 보여지는 것처럼 사용자 스택 영역에서 악의 있는 사용자의 코드가 실행되는 것을 알 수 있으며, 인스트럭션이 실행될 때 탐지를 하여야 하므로 하드웨어적인 요소가 아니고서는 근본적인 해결이 되지 않는다는 것을 알 수가 있다. 즉 버퍼 오버플로우가 발생되는 것은 스택 영역에 기록해 놓은 임의의 코드 엔트리 포인트를 복귀 주소(return address)가 있는 스택 영역에 overwrite하여 스택 영역에서 코드를 실행하도록 하는 것인데, 이를 막기 위해 스택 영역을 실행 불가능 영역으로 하여 버퍼 오버플로우를 어렵게 하는 것이다.

3.1 코드 세그먼트의 주소 limit 수정

스택 영역에서 실행하지 못하도록 하는 것은 사용자의 코드 세그먼트를 스택 영역의 최하위 주소로 한다. 이렇게 하면 코드 세그먼트의 제한된 영역 밖인 스택 영역에서 실행을 하려고 하면 예외 상황(exception)이 발생한다.

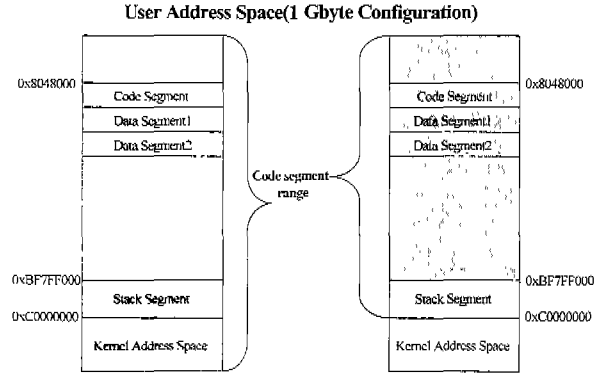
주소 limit를 수정하려면 먼저 코드 세그먼트 디스크립터의 필드 내용을 알아야 한다. 코드 세그먼트가 나타내는 각 필드의 내용은 (그림 9)와 같다.



(그림 9) 코드 세그먼트 구성

위의 (그림 9)에서 처럼 많은 필드가 있는데, 그 중 중요한 것 몇 가지만 설명을 하겠다. 먼저 세그먼트 베이스의 경우는 시작 주소를 말하면 이는 0x00000000로 수정이 되지 않는다. 세그먼트 리미트의 경우는 세그먼트의 마지막 주소를 나타내는데, 이는 granularity와 상관이 있다. 만약 granularity가 1로 세트가 되어 있으면 페이지(0x1000) 단위가 되는 것이고, 0이면 1바이트 단위가 되는 것이다. 즉 리미트는 5개의 16진수로 되어 있기 때문에 4Gbyte를 나타내려면 페이지 단위가 되어야 나타낼 수 있다. 그 이외는 펜티엄 매뉴얼에 자세히 나와 있다.

여기에서 수정이 되어야할 값은 세그먼트 리미트 값이다. 본래의 사용자 코드 세그먼트값을 0x00cffa00000000ffff 에서 0x00cbfa000000f7ff로 수정한다. 그렇게 함으로써 사용자 주소 공간은 다음 (그림 10)과 같이 변경된다. 즉 사용자 코드 세그먼트 영역이 0x00000000에서부터 0xBF7F7ff00까지로 세팅된다.



(그림 10) 변경된 사용자 주소 공간

3.2 트랩 처리

이에 따라 물론 잘못된 경우가 되겠지만, 스택 영역에서 코드가 실행되려고 하면 예외 번호 13번의 인터럽트가 발

생한다. 이는 general protection 오류로 세그먼트의 리미트를 넘은 경우뿐만이 아니라 다른 경우에도 발생하므로 이 경우에 스택 오버플로우 발생인 경우를 확인하여 오류값을 반환하여야 할 것이다.

Linux에서는 13번이 발생되면 trap gate에서 general protection 처리를 하게 된다. 여기에서 예외적으로 신호 처리에서 커널 내에서 시그널 처리를 한 후에 사용자 공간에서 복귀하기 위해 시스템 호출을 부르는데, 이때 스택 영역에서 실행을 하므로 이에 대한 처리를 하여야 한다. 이는 물론 스택 프레임을 셋업하는 코드를 수정할 때 하는 동작과 일치하여야 한다.

- 시그널 처리인 경우에는 먼저 시그널 처리의 경우인지 확인하는 코드가 실행된다. 먼저 코드 세그먼트가 사용자 코드 세그먼트인가 확인하고, 실행 인스트럭션이 ret인가 확인한다. 그리고는 사용자 스택에서 얻은 값이 시그널 처리 루틴에서 저장한 값인가를 확인 하고 그런 경우에는 해당하는 시스템 호출을 실행하고 복귀한다.
- 이번에는 스택 오버플로우가 발생되는 경우를 확인한다. 이는 먼저 코드 세그먼트가 사용자 코드 세그먼트인가 확인하고, 실행 인스트럭션이 ret인가 를 확인한 후에 해당 주소를 확인하여 스택영역 내의 주소이면 세그먼트 폴트를 내도록 한다.
- 그 이외에도 a. out 파일이나 elf 파일의 처리에서 스택에서 실행이 가능하다고 하는 플래그(PF_STACKEXEC) 를 모드 클리어 시킴으로써 스택 영역에서는 실행이 불가능함을 나타내도록 세트한다.

3.3 시그널 처리

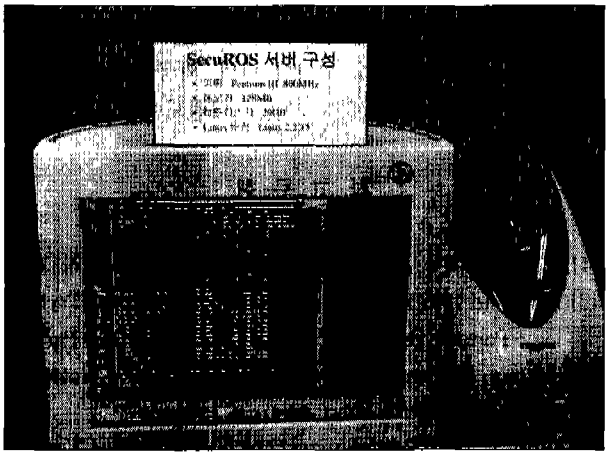
리눅스에서는 시그널을 처리한 후에 사용자 공간으로 복귀하기 위해 스택 영역에 해당 인수를 push 한 후에 해당되는 시스템 호출을 부른다. 이렇게 하다 보면 스택 프레임에 인수를 셋업하는데에서 셋업을 하게 되면 스택에서 실행을 하지 못하는 경우에 시그널 처리가 되지 않는다. 이에 문제점을 해결하기 위해 스택 프레임을 셋업하는 코드 내에 시스템 호출을 콜 하는 대신 트랩 처리에서 알아 볼 수 있는 값을 사용자 스택 프레임에 push 하고 스택에서 실행하도록 한다.

이 경우에도 return하는 코드는 두 가지 경우로 일반 시그널 처리에 대한 복귀인 sys_sigreturn 시스템 호출 처리, 실시간 처리의 경우에 return 하도록 하는 sys_rt_sigreturn 시스템 호출 처리이다. 이 두 가지 경우를 구분하여 각 플래그를 스택 프레임에 넣는다.

불론 이 플래그에 따라 트랩 처리 루틴에서 경우에 따라 sys_sigreturn이나 sys_rt_sigreturn 시스템 호출을 실행하고 반환하게 된다. 이에 대한 전체적인 처리 그림은 다음과 같다.

4. 해킹 대응 프로토타입

해킹 대응 기술은 펜티엄 Ⅲ 800Mhz 프로세서에 메모리 128Mbyte, 하드 디스크 20Gbyte로 구성된 서버 시스템에서는 테스트되었다. 이 기능은 개발된 시스템을 쉽게 테스트할 수 있도록, 버퍼 오버플로우를 일으키는 프로그램을 만들어서 실행시키는 것으로 이루어졌다.



(그림 11) 테스트 프로토타입

먼저 해킹 대응 기능이 추가되지 않은 기존 시스템에서 실행시키고, 루트 권한이 획득되는 것을 확인하고, 다음은 개발한 시스템에서 루트 권한이 획득되지 않고 "Segmentation fault"가 일어나는 것으로 확인하였다. 운영체제 환경은 알짜 리눅스 6.1 시스템에 리눅스 2.2.15 커널을 팻치하여 구성한 본래의 시스템과, 알짜 리눅스 6.1 시스템에 리눅스 2.2.15 커널을 팻치한 원천코드에 해킹 대응 기능을 구현하여 구성한 시스템으로 나타낸다. 시험한 테스트 프로토타입은 위의 (그림 11)과 같다.

5. 결 론

본 논문에서는 시스템 해킹 중 루트 권한을 획득하기 위해 가장 많이 사용되는 버퍼 오버플로우를 해결하는 리눅스 기반의 보안 운영체제를 소개하였다. 또한 리눅스 기반의 보안 운영체제는 다단계 접근제어를 제공하고 그 위에 IEEE에서 정의한 ACL과 MAC에 관련된 모든 보안 명령어 및 라이브러리를 사용자 인터페이스로 제공한다. 개발된 보안 운영체제는 루트 권한의 획득을 감소함으로써 시스템 해킹을 줄이며, 운영체제를 통한 많은 위험 요소를 줄일 수 있었다. 이러한 버퍼 오버플로우 해결에도 불구하고 다른 시스템 해킹 위험 요소들이 있어 루트 권한을 획득하게 되

나, 이는 커널 수준의 접근 제어나 시스템 관리자인 루트의 권한을 분산시킴으로써 시스템의 주요 파일이나 장치들을 보호할 수가 있다. 요즈음은 버퍼 오버플로우 뿐만 아니라 분산 서비스 거부 공격 등과 같은 해킹 기법들에 의해 시스템을 halt 상태로 만들기도 한다. 이를 위하여 시스템 차원의 감사 추적 기능이나 시스템 모니터링 기능으로 탐지하고 차단할 수 있도록 연구하여야 할 것이다.

현재 보안 운영체제 시스템은 범용화 되어 사용되지 않고 있기 때문에 그 안전성이나 성능을 평가하는 것 또한 쉽지 않다. 국내에는 아직 보안 운영체제 시스템을 평가하는 기준이 마련되지 않은 상태이다. 국제 평가 기준을 따르면서도 국내 상황에 알맞은 기준이 마련되어 시스템의 안전성과 성능을 판별할 수 있어야 하며, 외국 제품이나 기술과 차별화 되는 기술 보유의 기틀이 마련되어야 하겠다.

참 고 문 헌

[1] 홍승필, 고제욱, "정보보안 기술과 구현", pp.293, 파북, 1998.

[2] IEEE Std 1003.1e-Draft standard for Information Technology-Portable Operating System Interface(POSIX) Part 1 : System Application Program Interface(API)- Protection, Audit and Control Interfaces.

[3] IEEE Std 1003.2c-Draft standard for Information Technology-Portable Operating System Interface(POSIX) Part 2 : Shell and Utilities : Protection and Control Interfaces.

[4] "DoD Trusted Computer System Evaluation Criteria," http : //147.51.219.9/otd/c2protect/isso/DOD/520028std/5200_28std1.htm#1.0.

[5] "Evaluated Product List by Vendor," http : //www.radium.ncsc.mil/tpep/epl/epl-by-vendor.html.

[6] "Rule Set Based Access Control," http : //www.rsbac.org/.

[7] "Medusa," http : //www.medusa.formax.sk.

[8] "Security-Enhanced Linux," http : //www.nsa.gov/selinux/.

[9] "Trusted Solaris 8," http : //www.sun.com/trusted-solaris/.

[10] Jong-Gook Ko, So-young Doo, Sung-Kyung, and Jeong-Nyeo Kim, "Design and Implementation for Secure OS based on Linux," WISA2000, Vol.1, No.1, pp.175-181.

김 정 녀

e-mail : jnkim@etri.re.kr

1987년 전남대학교 전산통계학과(이학사)

1995년~1996년 Open Software Foundation Research Institute 공동 연구 파견(미국)

1998년~2000년 충남대학교 대학원 컴퓨터공학과(공학석사)

2000년~현재 충남대학교 대학원 컴퓨터공학과(박사과정)

1988년~현재 한국전자통신연구원, 보안운영체제 연구팀장(선임연구원)

관심분야 : OS, Secure OS, Distributed Processing, Fault Tolerant System,

정 교 일

e-mail : kyoil@etri.re.kr

1981년 한양대학교 전자공학과(공학사)

1983년 한양대학교 산업대학원 전자계산학과(공학석사)

1997년 한양대학교 대학원 전자공학과(공학박사)

1981년~현재 한국전자통신연구원 정보보호기술연구본부 정보보호기반연구 부장(책임연구원)

관심분야 : IC Card, Security, Biometrics, 국가기반보호, 신호처리

이 철 훈

e-mail : chlee@ce.cnu.ac.kr

1983년 서울대학교 전자공학과(공학사)

1983년~1986년 삼성전자 컴퓨터개발실 연구원

1988년 한국과학기술원 전기및전자공학과(공학석사)

1992년 한국과학기술원 전기및전자공학과(공학박사)

1992년~1994년 삼성전자 컴퓨터사업부 선임연구원

1994년~1995년 Univ. of Michigan 객원연구원

1995년~현재 충남대학교 컴퓨터공학과 부교수

관심분야 : 운영체제, 병렬처리, 결함 허용 및 실시간 시스템 등.