# 리눅스 운영체제에서 주소값 오류시 스택 복구를 통한 커널 하드닝 기능 구현

장 승 주*

## The Implementation of Kernel Hardening Function by Recovering the Stack Frame of Malfunction Address on the Linux Operating System

Seung-Ju Jang*


### 요 약

본 논문은 리눅스 커널 운영체제에서 커널 개발자의 실수나 의도하지 않은 오류 및 시스템 오류로 인하여 발생되는 시스템 정지 현상을 줄이기 위해서 커널 스택의 복구를 통한 커널 하드닝 기능을 구현한다. 본 논문에서 제안하는 커널 하드닝 기능은 panic 이 발생한 커널 주소에 대해서 커널 스택 내의 값들을 정상적인 값으로 복구함으로써 정상적인 시스템 동작을 보장한다. 본 논문에서 제안한 커널 하드닝 기능을 리눅스 커널 중에서 많이 사용하는 네트워크 모듈에 적용하였다. 커널 스택 복구를 통한 커널 하드닝 기능의 실험을 위하여 네트워크 모듈에 강제적인 panic 현상을 유발시키고, 잘못된 스택 값의 복구를 통해서 정상 동작을 실험하였다.

### ABSTRACT

This paper designs the kernel hardening function by recovering the kernel stack frame to reduce the system error or panic due to the kernel code error. The suggested kernel hardening function guarantees normal system operation by recovering the incorrect address of the kernel stack frame. The suggesting kernel hardening mechanism is applied to the network module of Linux which is much using part. I experimented the kernel hardening function at the network module of the Linux by forcing panic code.



키워드

커널 하드닝(kernel hardening), 스택 복구(stack recovery), 리눅스 운영체제(Linux O.S), 시스템 신뢰성(system availability)


## Ⅰ. 서 론

최근에 리눅스 운영체제의 사용이 증가되고 있다. 리눅스 운영체제는 open source의 특징을 이용하여 임베디드 시스템 분야에서도 널리 이용되고 있으며 기업에서는 웹서버, 파일 서버, DB 서버 등으로 활용되고 있다. 리눅스 운영체제의 특징은 리눅스 커널 소스를 수정하여 파일 시스템, 디바이스 드라이버 등을 커널에 추가할 수 있다는 것이다. 리눅스 운영체제는 많은 사람이 공개적으로 커널의 내용을 변경, 수정으로 인하여 커널이 상업적인 운영체제에 비하여 일부 문제점을 가지고 있다.[1,2,3]리눅스 운영체제 커널 소스를 수정하거나 커널 모듈 프로그램을 잘못 작성한 경우에는 시스템 동작과 직결된다. 잘못된 코드가 커널에 존재할 경우에 시스템 동작

* 동의대학교

접수일자 : 2006. 7. 18

중에 비정상적인 동작으로 시스템이 정지되는 현상이 발생하고, 심각한 경우 데이터가 파괴되기도 한다.[1,12]

본 논문에서 제안하는 커널 하드닝 기능은 운영체제 커널 내에서 잘못된 코드로 인하여 시스템이 정지되는 것을 정상적으로 복구함으로써 시스템이 정상 동작되도록 보장한다. 본 논문에서 제안하는 커널 하드닝 기능은 시스템 가용성을 높이고, 안정된 시스템 사용을 보장할 수 있다. 커널 하드닝 기능은 기존의 고장 감내 시스템과는 달리 많은 비용을 들이지 않고 구현이 가능하다. 따라서 앞으로 많은 운영체제에서 커널 하드닝 기능 구현으로 커널의 안정성을 높이고 저렴한 비용으로 시스템 안정성을 높일 수 있을 것으로 기대된다.

본 논문은 리눅스 운영체제 커널에서 복구 가능한 오류에 대해서 복구가 될 수 있도록 커널 하드닝 기능을 설계한다. 본 논문에서 제안하는 커널 하드닝 기능은 "panic"이 발생될 경우에 "panic"이 발생한 커널 스택 프레임 내에 비정상적인 레지스터리 값들을 정상적인 값으로 복원함으로써 정상적인 동작이 가능하도록 하는 것이다.

커널 내에서 "panic"이 발생되면 주소 타입인 경우에 잘못된 주소 값이 cr2 레지스터에 저장이 된다. 이 주소 값을 복원할 수 있을 경우에 정상적인 동작이 가능하도록 한다. 본 논문에서 제안하는 커널 하드닝 기능은 리눅스 운영체제에서 많은 사용자들이 사용하는 네트워크 모듈에 적용한다. 네트워크 모듈에 적용하여 동작의 안정성을 실험한다.

본 논문의 구성은 2장에서 관련 연구를 살펴보고, 3장에서 커널 하드닝 구현, 4장에서는 실험 결과에 대해서 설명하고, 마지막으로 5장에서 결론으로 구성되어 있다.

## Ⅱ. 관련 연구

일반적으로 UNIX 운영체제는 계층적인 구조를 가지고 있다. 각 계층마다 고유의 특성을 가진 다양한 형태의 고장을 일으킨다. 그러므로 하드웨어, 운영체제 커널, 응용 프로그램 환경의 각각에 대해서 별도의 독립적인 고장 관리 체계를 제공해야 한다. 이들 각 모듈 간의 고장 복구(fault recovery) 전략은 서로 간에 연관성을 가지고 있다. 운영체제에서 고장 감내 기능 지원은 운영체제 커널과 시스템 관리 부분에 고장 관리 및 고장 복구 기능이 있어야 한다. 일반적으로 고장 감내성을 제공하기 위해서는 고장 감내 기능의 강도에 따라 L1 - L5의 5가지로 분류하고 있다.[2,5]

지금까지 상용화된 대표적인 고장 감내 시스템으로 Tandem, Fujitsu, Stratus, DEC 등을 꼽을 수 있다. Tandem은 OLTP(On-Line Transaction Processing) 시장을 겨냥한 비상 안전 컴퓨터 시스템을 개발했다. Tandem 시스템은 loosely coupled 형태를 취하고 있으며 비상 안전을 위하여 모든 시스템 요소들 사이에 이중 경로(dual path)를 제공하고 있다. 고장 탐지(fault detection)는 하드웨어적인 방법과 소프트웨어적인 방법을 사용한다. 고장 복구(fault recovery)를 위해서는 모든 프로세스가 두 대의 컴퓨터에서 수행되는 "process pair" 개념을 사용한다. "process pair" 개념은 동일한 기능을 하는 프로세스를 2개 이상 동작시키고, 이 프로세스를 primary process, secondary process로 동작시킨다. primary process에 이상이 발생하면 secondary process가 primary process의 기능을 대체하는 것이다. Stratus는 데이터 손실이나 성능 저하, 그리고 특별한 프로그램이 없는 지속적인 동작을 위한 failover 개념을 이용한 고장 감내 기능을 제공하고 있다. Fujitsu는 대형 컴퓨터나 일반적인 업무용 시스템에 적용할 수 있는 고장 감내 기능을 제공하고 있다[2,5,7]

현재까지 커널 하드닝 관련 연구는 많이 이루어지고 있지 않다. 최근의 리눅스 커널 하드닝 관련 연구 중에서 몬타비스타에서 연구가 활발히 이루어지고 있다. 몬타비스타는 이미 커널 하드닝 기능이 내장되어 있는 CGE(Carrier Grade Edition) 버전의 리눅스 운영체제를 상업적으로 판매되고 있다.[3,7,8,9] 몬타비스타의 CGE 버전은 커널 하드닝 기능을 3가지 영역으로 분류하고 있다. 일반적인 커널 하드닝 기능은 code review, panic removal, fault injection testing 등이 있다.[3] Code reviews는 커널 코드를 설계 및 구현하고 난후 지속적인 점검을 통해서 원천적으로 커널 코드의 오류를 방지하는 기능이다. Panic removal 기능은 운영체제 코드를 검사한 후, 시스템을 중지(panic) 시킬 것인지 아니면 프로세스를 kill시킬 것인지를 결정하는 기능이다. Fault injection testing 기능은 소프트웨어 오류인 경우 리눅스 커널이 복구할 수 있는 능력이 있는지 없는지에 대해서 검사하도록 한다.

위와 같이 몬타비스타의 커널 하드닝 기능은 Code

Reviews를 통해서 코드를 재검토한 후 특정 프로세스가 panic 루틴으로 들어왔을 때 panic 루틴으로 들어온 모든 프로세스를 kill하여 시스템이 정상적으로 수행 되도록 하는 것은 아니다. 현재 프로세스가 시스템에 영향을 주는 프로세스인 경우에 대해 panic() 함수를 수행하여 시스템이 정지 되도록 한다. 리눅스 운영체제 커널 코드가 프로그래머의 오류 등 사용자의 잘못에 의한 경우라면 시스템을 정지시키지 않고 현재 프로세스만 kill하여 시스템이 정상적으로 수행되도록 한다. 몬타비스타의 커널 하드닝 과정은 시스템의 적합성에 대한 판단으로 시스템 오류가 발생한 모든 조건의 kernel panic에 대한 검토를 포함하고 있다.[4,10] 몬타비스타의 커널 하드닝은 커널 내의 가용성 측면에서 기능이 약한 반면 주로 커널 문제에 대한 관리 측면이 강한 운영체제이다. 이와 같이 커널 하드닝 기능은 시스템 고 가용성(high availability)을 보장하고자 하는데 목적이 있다. 커널 하드닝 기능은 임의의 커널 내 오류(fault)에 따른 panic에 적절히 대처할 수 있는 기법으로 code path 시험을 통해서 에러 코드에 오류를 줄이는 과정이다. 이러한 개념을 fault injection이라고 한다.[1,3,5,6,11,12]

## Ⅲ. 커널 하드닝 구현

### 3.1. 커널 하드닝 기능

리눅스 커널 내에서 "panic"이 발생하게 되면 기존의 리눅스 커널은 시스템이 더 이상 정상적인 동작을 할 수 없는 상태가 된다. 본 논문에서 제안하는 커널 하드닝 기능은 "panic"이 발생될 경우에 "panic"이 발생된 커널 스택 프레임 내에 비정상적인 레지스터리 값들을 정상적인 값으로 복원함으로써 정상적으로 동작이 가능하도록 하는 것이다. 리눅스 커널에서 시스템 이상이 발생하게 되면 "panic" 함수를 수행하게 된다. 따라서 "panic" 함수에서 이상이 발생된 주소 값에 대한 복구가 가능하면 일반적인 시스템 이상 상황에도 정상적인 시스템 동작을 보장할 수 있다. 본 논문에서 제안하는 커널 하드닝 기능은 임베디드 시스템 환경에서 네트워크 모듈 동작의 안정화를 목표로 연구를 진행하였다. 따라서 본 논문에서 제안하는 커널 하드닝 기능을 리눅스 네트워크 모듈에 적용한다. 커널 내에서 "panic"이 발생되면 주소 타입인 경우에 잘못된 주소 값이 cr2 레지스터에 저장이 된다. 이 주소 값을 복원해 줌으로써 정상적인 커널 동작이 가능하도록 한다. 본 논문에서 제안하는 커널 하드닝 기능은 비 정상적인 커널의 동작으로 인한 손실을 줄일 수 있다.

### 3.2. 커널 하드닝 구현

리눅스 커널 내 네트워크 기능의 안정적인 커널 서비스를 보장하기 위하여 본 논문에서 제안한 커널 하드닝 기능을 ip_input.c 커널 소스 코드의 ip_rcv() 함수에 구현하였다. 이 함수는 사용자가 "ping" 명령어 등을 수행하는 경우에 동작이 되는 함수이다. 그리고 인위적인 실험 환경의 구축을 위하여 네트워크 접속이 일어나면 "panic"이 발생되도록 "linux/net/ipv4/ip_input.c"에 다음 그림 1과 같은 코드를 삽입하였다.

```
ip_rcv()
{
    //panic이 되기 전의 상황을 확인하기 위한 임의의 static 변수값
    static int harden_count;
    .....
    harden_count++;
    printk("ip_rcv() -> %d₩n", harden_count);
    if(harden_count > 150) {
    aa = 10 + 10;
    printk("aa=%d", *aa); // 잘못된 페이지 영역에 대한 참조로
                                    page fault가 강제적으
                                    로 발생하도록 하여
                                    panic을 유발시킨다.
    } else {
    printk("harden_count=%d₩n", harden_count);
    }
    .....
}
```

그림 1. 강제 panic 발생을 위한 ip_rcv() 내의 구현 코드
Fig. 1 Implemented Code for Panic Event in ip_rcv()

그림 1과 같이 네트워크 기능을 이용할 경우에 ip_rcv()함수 내에 panic이 강제적으로 발생되도록 하고, panic이 발생된 과정을 역순으로 찾아내기 위해 /linux/kernel/panic.c소스 코드를 중심으로 스택 정보를 출력한다.

함수내의 레벨 good_area, bad_area, no_context, do_sigbus에 대한 printk문을 이용하여 잘못된 주소 정보를 확인한다. 우선 good_area, bad_area, no_context, do_sigbus 레벨에 대하여 살펴보면, "good_area :"부분은 프로세스 주소 영역 안에서 발생한 폴트 주소를 다룬

다."bad_area :" 부분은 프로세스 주소 밖에서 발생한 폴트 주소를 다룬다. "no_context :" 부분은 인터럽트이거나 또는 커널 쓰레드가 실행 중일때 발생하는 예외(exception)를 처리하는 부분이다. "do_sigbus :" 부분은 획득한 세마포어를 놓고, 프로세스의 쓰레드 구조체에 있는 cr2 필드에 오류를 일으킨 주소를 넣는다.

그림 2는 do_page_fault()함수 내의 "no_context :"부분에서 panic 처리의 마지막 단계를 나타내는 코드이다.

```
if (address < PAGE_SIZE)
                printk(KERN_ALERT "Unable to handle kernel NULL
            pointer dereference");
        else
                printk(KERN_ALERT "Unable to handle kernel
    paging request");
        printk(" at virtual address %08lx₩n",address);
        __asm__("movl %%cr3,%0" : "=r" (page));
        printk(KERN_ALERT "current->tss.cr3 = %08lx, %%cr3
  = %08lx₩n",
                tsk->tss.cr3, page);
        page = ((unsigned long *) __va(page))[address >> 22];
        printk(KERN_ALERT "*pde = %08lx₩n", page);
        if (page & 1) {
                page &= PAGE_MASK;
                address &= 0x003ff000;
                page = ((unsigned long *) __va(page))[address
      >> PAGE_SHIFT];
                printk(KERN_ALERT "*pte = %08lx₩n", page);
        }
        die("Oops", regs, error_code);
    do_exit(SIGKILL);
```

그림 2. panic()내의 최종 시스템 처리 단계
Fig. 2 Final System Procedure in panic()

그림 2 과정을 거치게 되면 panic이 발생한 커널 내의 부분과 관련한 정보를 출력한다. 관련 정보로는 가상 주소 값, 커널에서 사용하는 레지스터 값들이 해당된다. 전반적인 리눅스 네트워크 모듈에서 커널 하드닝 기능의 구현 내용은 3.2.1 절에서 설명한다.

가. 네트워크 모듈에 커널 하드닝 구현

본 논문은 네트워크 모듈에 본 논문에서 제안하는 커널 하드닝 기능을 설계한다. 리눅스 네트워크 모듈은 사용 빈도가 높고 커널 내에서 문제가 발생하면 시스템 사용에 치명적인 결과를 가지고 올 수 있다. 따라서 본 논문에서 제안하는 리눅스 커널 하드닝 개념을 네트워크 모듈에 구현하고 실험을 한다. 그림 3은 본 논문에서 구현하고자 하는 네트워크 모듈의 수행 과정을 나타낸다.

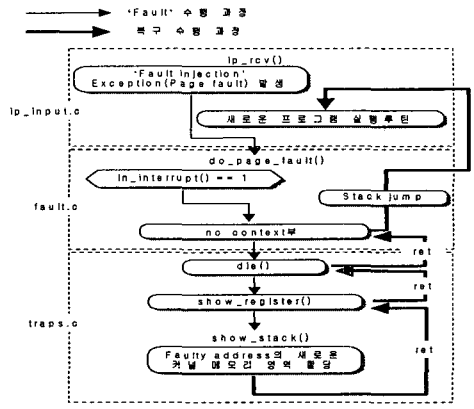


그림 3. 네트워크 모듈 중 ip_rcv() 함수에서 커널 스택 복구 과정
Fig. 3 Stack Recovery Procedure of Network Module in ip_rcv()

커널 하드닝 기능 시험을 위하여 인위적인 커널 동작의 오류가 생기도록 한다. 이러한 인위적인 "커널 오류 코드는 ip_input.c 파일 내의 ip_rcv() 함수 내에서 이루어진다. ip_rcv() 함수 내에서 "panic"이 발생하면 ./arch/i386/mm/fault.c의 do_page_fault() 함수를 거쳐서 패닉 과정을 진행하게 된다.

다음으로 do_page_fault() 함수의 no_context 레벨 코드 부분이다. 이 코드에서 in_interrupt() 인터럽트 처리 루틴을 거치게 된다. 그리고 do_page_fault() 함수에서 다음과 같은 어셈블리 명령을 실행하도록 한다.

**__asm__("movl %%cr2,%0":"=r" (address)); (1)**

이 어셈블리 명령에서 주소가 바로 본 논문에서 설정해 놓은 잘못된 주소 값을 가지고 있다. 이 주소 값은 cr2 레지스터와 관련이 있다. 또한, 패닉이 발생했을 때 스택 값들을 확인하는 과정이 필요하다. 이는 프로그램이 실행됨에 있어서 스택에 쌓이는 값들을 통해 어떤 루틴을 실행하는지에 대해 확인하는 과정이다. 본 논문에서는 패닉이 발생했을 때 스택 값을 추적해 본다. 이는 "nm -n"이라는 유닉스 명령어인 심볼 보기 옵션을 통해서 패닉 발생 경로를 알 수 있다.

"panic" 이 발생하고 난 이후의 과정은 다음과 같다. 잘못된 주소 값에 대해서 리눅스 커널 내에서 인터럽트가 발생한다. 인터럽트가 발생하게 되면 do_page_fault() 함수의 no_context 레벨에서 die() 함수를 호출한다. die()

함수에서는 show_register()함수를 호출한다. show_register() 함수는 show_stack() 함수를 호출한다. 이 show_stack() 함수에서 문제의 cr2레지스터 값이 스택 내에 있음을 확인할 수 있다. 이 주소 값이 잘못되어서 시스템 패닉이 발생한 것이다. 이 주소 값을 정상적인 값으로 복구하고, 스택 내의 잘못된 경로 호출 과정을 정상적인 과정으로 복구해 주면 정상적인 시스템 동작이 가능하다. 그림 4는 잘못된 주소 값을 복구하는 show_stack() 함수의 소스 코드를 보여준다.

```
int show_stack(unsigned long * esp)
{
        unsigned long *stack;
        int i;
        unsigned long address;

SS1:    if(esp==NULL)
SS2:            esp=(unsigned long*)&esp;
SS3:    stack = esp;
SS4:  __asm__("movl %%cr2,%0":"=r" (address));

SS5:  for(i=0; i < kstack_depth_to_print; i++) {
SS6:            if (((long) stack & (THREAD_SIZE-1)) == 0)
SS7:                    break;
SS8:            if (i && ((i % 8) == 0))
SS9:                    printk("₩n      ");
SS10:       if(*stack == address) {
SS11:               *stack = kmalloc(8, GFP_KERNEL);
SS12:               printk("Change Value of Stack addr =
SS13:                       0x%x !!!!!!!!!!!!!!!₩n", *stack);
SS14:           return 1;
SS15:       }
SS16:           printk("addr %08lx : ", stack);
SS17:           printk("%08lx ", *stack++);
SS18:   }
SS19:   printk("₩n");
SS20: return 0;
}
```

그림 4. 잘못된 레지스터 값을 복구하는 show_stack() 함수 소스 코드

Fig. 4 Source Code of show_stack() Function for Wrong Register Value

그림 4는 리눅스 커널에서 커널 스택의 복구를 통한 하드닝 기능을 구현하기 위한 과정을 보여준다. 그림 4에서 cr2 레지스터에 들어가 있는 주소 정보를 address 변수에 저장한다(SS4 문장). address 변수는 패닉이 발생된 값을 담고 있는 변수이다. 스택 내에 들어있는 값 중에서 address 변수 값과 일치하는 값이 있을 경우에 이 값을 정상적인 값으로 변경을 해주면 리눅스 커널의 동작이 정상적으로 이루어진다. 이 과정이 그림 4의 SS10 문장에서 SS15 문장까지의 과정이다.

### 3.3. 구현 환경

리눅스 운영체제에 커널 하드닝 기능을 구현하기 위한 시스템 환경은 Intel CPU 450MHz 프로세서와 RAM은 128Mbyte을 사용하였으며, 메인보드 캐시는 256KByte이고 리눅스 RedHat 9.0 기반의 운영체제를 사용하였다. 리눅스 커널 버전은 2.4.20을 사용하였다. 또한 프로그램 개발을 위해서 GNU 툴을 사용하였다.

### 3.4. 네트워크 모듈에서 패닉 형태

리눅스 커널의 네트워크 부분에서 커널 패닉을 일으키는 유형을 살펴본다. 이들 유형은 kmem_cache_create(), __get_free_pages(), netlink_kernel_create(), create() 함수들에 의하여 결정된다. 이 함수들에 결정되어지는 주소 값을 통해 패닉의 유무를 결정하고 있다. 패닉을 결정하게 되는 네트워크 기능에서 각 변수 타입을 살펴보면 다음 그림 5와 같다.

```
route.c
    ip_rt_acct = (struct ip_rt_acct *)
                        __get_free_pages(GFP_KERNEL, order)
    ipv4_dst_ops.kmem_cachep = kmem_cache_create(
                        "ip_dst_cache", sizeof(struct rtable), 0,
                        "SLAB_HWCACHE_ALIGN, NULL, NULL)
    rt_hash_table = struct rt_hash_bucket *)
                        __get_free_pages(GFP_ATOMIC, order)

tcp_ipv4.c
    err=ops->create(tcp_socket, IPPROTO_TCP)

tcp.c
    tcp_openreq_cachep = kmem_cache_create("tcp_open_request",
                        sizeof(struct open_request), 0,
                        SLAB_HWCACHE_ALIGN,  NULL, NULL);
    tcp_bucket_cachep = kmem_cache_create("tcp_bind_bucket",
                        sizeof(struct tcp_bind_bucket),
                        0, SLAB_HWCACHE_ALIGN, NULL, NULL);
    tcp_timewait_cachep = kmem_cache_create("tcp_tw_bucket",
                        sizeof(struct tcp_tw_bucket),
                        0, SLAB_HWCACHE_ALIGN, NULL, NULL);
    tcp_ehash = (struct tcp_ehash_bucket *)__get_free_pages(GFP_ATOMIC, order);
    tcp_bhash = (struct tcp_bind_hashbucket *)
                        __get_free_pages(GFP_ATOMIC, order);
```

그림 5. 커널 함수들을 통해 panic을 결정하는 각 소스 내의 변수들

Fig. 5 Variables of Each Source to decide Panic

그림 5에서 보는 바와 같이 이들 각각의 변수들이 특정 커널 함수에 의해 값이 결정되어지고 모든 변수들은 조건문에 의해 NULL로 판단하게 될 시에 kernel panic을 결정하게 되는 형태이다.

## Ⅳ. 실 험

3장에서 제안한 커널 하드닝 개념을 실제 네트워크 모듈에 적용한 경우에 대해서 실험한다. 실제 리눅스 커

널의 네트워크 기능 모듈에 구현하여 네트워크 동작 중에 이상 상황의 발생에 대해서 정상적인 동작이 가능한지를 실험하였다. 3장에서 설계한 show_stack() 함수의 내용을 실제 이용하여 실험을 수행하였다. show_stack() 함수를 ip_rcv() 함수에서 그림 6과 같이 호출되도록 한다.

```
ip_rcv()
{
.....
show_stack(sp);
/* aa값 확인 */
.....
}
```

그림 6. show_stack() 함수를 이용한 실험
Fig. 6 Experiment of show_stack() Function

시스템 호출 동안에 스택에 저장되어지는 레지스터의 구조체는 asm-i386/ptrace.h 헤더 파일에 다음과 같이 정의되어 있다.

```
struct pt_regs{
        long ebx,ecx,edx,esi,edi,ebp,eax;
        int xds, xes,orig_eax,eip,xds,eflags,esp,xss;
}
```

그림 7. 스택에 저장되는 정보 자료 구조
Fig. 7 Data Structure of Stored Stack Information

본 논문에서는 esp레지스터의 값을 이용하여 커널 하드닝을 구현한다. esp레지스터를 show_stack()함수를 통하여 이용하게 되면 특정 로컬 변수가 가지는 주소 값이 스택에 차례대로 들어가는 것을 알 수 있다. ip_rcv()에서의 로컬 변수의 주소를 알아보고, 페이지 폴트를 발생시킨 "aa" 라는 로컬 변수가 가지는 스택의 위치를 찾는다. 실제 실험을 위하여 커널 내에 강제적인 panic을 유발하도록 하기 위하여 그림 1과 같이 커널을 수행한다.

"aa"가 가지는 잘못된 주소 값으로 인하여 시스템은 panic을 유발되게 된다. panic이 발생되면 do_page_fault() 함수를 수행하게 되고, traps.c에서 잘못된 "aa"변수에 대해서 정상적인 주소 값(메모리)을 할당해 줌으로써 panic 발생이 아닌 정상적인 동작이 되도록 한다. 이 과정은 그림 3에서 ret라고 되어있는 과정을 나타낸다.

```
__asm__("movl %%esp,%0":"=r" (sp));
printk("---------- iph=0x%x, &bb=0x%x, aa=0x%x,
        sp=0x%x₩n", iph, &bb, aa, sp);
show_stack(sp);
__asm__("popl %%eax₩n₩t": : );
__asm__("movl %%eax,%0":"=r" (check));
__asm__("movl %%esp,%0":"=r" (sp));
printk("---------- check=0x%x, sp=0x%x₩n", check, sp);
__asm__("pushl %%eax₩n₩t": : );
```

그림 8. 비정상적인 주소 값을 정상적으로 복구
Fig. 8 Normal Recovery for Abnormal Address Value

그림 8은 비정상적인 주소 값을 정상적인 주소 값으로 복구하는 과정을 나타낸다. 여기서 esp 레지스터 값을 sp 변수로 읽어낸다. 그리고 show_stack(sp) 함수를 이용하여 스택 복구를 시도한다. 그 다음으로 정상적인 스택 정보를 저장한다. 그림 8과 같이 함으로써 비정상적인 스택의 값을 정상적인 스택 값으로 복구가능하다.

실험을 위하여 리눅스 커널 내부에 10개의 이중 연결 리스트를 구현하여 링크의 변경을 통하여 시스템에 오류가 발생되도록 한다. 다음 그림 9는 정상적으로 링크가 연결되어 있을 때에 커널 내에서 발생하는 정보를 보여준다.

```
*******  harden_count = 15 *******
15
*******  harden_count = 16 *******
15      16
*******  harden_count = 17 *******
15      16      17
*******  harden_count = 18 *******
15      16      17      18
*******  harden_count = 19 *******
15      16      17      18      19
*******  harden_count = 20 *******
15      16      17      18      19      20
*******  harden_count = 21 *******
15      16      17      18      19      20      21
*******  harden_count = 22 *******
15      16      17      18      19      20      21      22
*******  harden_count = 23 *******
15      16      17      18      19      20      21      22
*******  harden_count = 24 *******
15      16      17      18      19      20      21      22
```

그림 9. 10개의 링크드 리스트가 생성되고 난 후 콘솔에 출력되는 정보
Fig. 9 Display Information of Console after 10 Linked List Creation

그림 10은 ioctl시스템 콜을 사용하여 ip_input.c에 구현한 이중 연결 리스트의 링크를 변경한다. 이중 연결 리스트 링크 정보를 변경함으로써 커널 내에서 인위적인 "panic"이 발생되도록 한다.

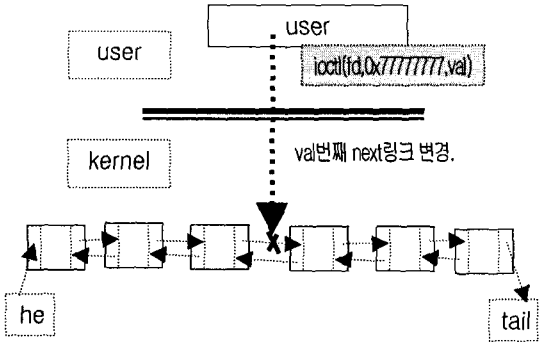


그림 10. ioctl() 시스템 콜을 사용한 커널 내 링크 변경
Fig. 10 Changing Link of Kernel by ioc시() System Call

그림 10은 사용자 모드에서 ioctl()시스템 콜을 사용하여, 정상적으로 생성한 링크의 n번째 포인터의 값을 삭제 또는 변경한 것을 보여준다. 링크를 변경하는 함수는 find_dnode_unlink()함수로 ioctl() 시스템 콜에서 "0x77777777"명령어를 이용할 수 있도록 한다.

```
[root@harden linux]# ./ast
-------------- current->pid =  1845
죽었음
[root@harden linux]#
```

그림 11. 링크 삭제를 하였을 때 출력 화면
Fig. 11 Display after Deleting Link

위 그림 11은 사용자 프로그램(ast.c-3번째 링크의 next 포인터를 NULL로 가지게 함)을 통해 리눅스 커널 내의 링크 정보를 변경한 후 프로세스의 동작 결과를 보여주는 화면이다. 그림 12는 콘솔에 나타난 결과를 보여준다.

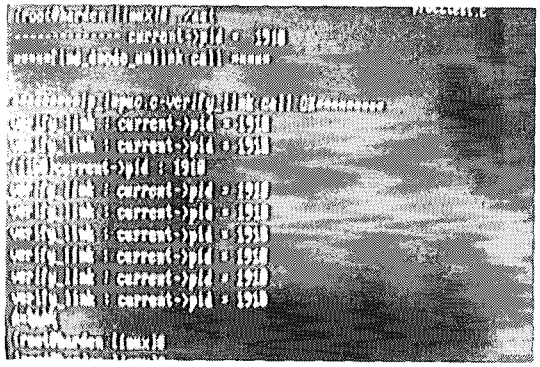

그림 12. 콘솔에 나타난 결과 화면
Fig. 12 Display of Console

그림 12는 verify_link()함수의 주요 부분으로 사용자 프로그램에서 ioctl() 시스템 호출 함수를 사용하여 커널 내에 정의된 ioctl()에 함수를 통해서 find_dnode_unlink() 함수를 호출하고, find_dnode_unlink()함수에서는 링크를 강제적으로 변경한다. 그러고 나서 그림 12의 verify_link()함수를 호출하게 된다.

```
01:     for( 0 to (전체 링크 사이즈)-2  increase by 1)
02:     {
03:             for( 2 to (전체 링크 사이즈)  increase by 1)
04:             {       q=q->prev;
05:             }
06:             /*
07:              *   ASSERT 판단 구문 삽입
08:              */
09:         printk("verify_link : current->pid = %d\n",current->pid);
10:             q=q->tail;
11:             p=p->next;
12:     }
```

그림 13. verify_link() 함수 주요 소스부
Fig. 13 Source of verify_link()

그림 13의 ASSERT 판단 구문을 통하여 생성한 이중 연결 리스트의 연결 관계를 검증한다. ASSERT 판단 구문에 사용되는 값으로는 예를 들면, b 노드의 next 포인터와 c 노드의 포인터, c 노드의 prev 포인터와 b 노드와의 포인터가 같은지에 대한 판단을 통하여 복구과정이 여부를 밝히어 실행한다. 이 실험 결과는 커널 내에서 링크가 잘못된 경우에 이것을 복구하여 정상적인 동작이 됨을 보여준다.

## Ⅳ. 결 론

리눅스 운영체제 커널에서 잘못된 연산이나 잘못된 프로그램으로 인하여 시스템이 정지되는 현상이 발생한다. 이러한 현상을 시스템 panic 이라고 하는데, 시스템 panic 으로 인하여 시스템이 정지된다면 큰 문제가 발생될 수 있다. 본 논문은 리눅스 커널에서 panic이 발생할 경우에 잘못된 주소로 인한 panic 에 대해서 정상적인 주소 값을 복원함으로써 정상적인 시스템 동작이 되도록 한다.

본 논문에서 제안하는 커널 하드닝 기능은 "panic"이 발생될 경우에 "panic"이 발생한 커널 스택 프레임 내에 비정상적인 레지스터리 값들을 정상적인 값으로 복원

함으로써 정상적인 동작이 가능하도록 하는 것이다. 커널 내에서 "panic"이 발생되면 주소 타입인 경우에 잘못된 주소 값이 cr2 레지스터에 저장이 된다. 이 주소 값을 복원할 수 있을 경우에 복원해 줌으로써 정상적인 동작이 가능하도록 한다.

리눅스 운영체제에서 구현한 커널 하드닝 기능을 네트워크 모듈에 구현하였다. 최근 리눅스 네트워크 기능의 많은 이용은 안정적인 커널 동작을 필요로 하고 있다. 따라서 리눅스 커널내 네트워크 기능의 안정적인 커널 서비스를 보장하기 위하여 본 논문에서 제안한 커널 하드닝 기능을 ip_input.c 커널 소스 코드의 ip_rcv() 함수에 구현하였다. 사용자가 ping 명령어 등을 수행하는 경우에 동작이 되는 함수이다. 네트워크 모듈에서 강제적인 panic 유발 상황을 만들어 놓고, panic이 발생했을 경우 정상적인 동작을 실험하였다. 본 논문에서 제안한 리눅스 커널 하드닝을 네트워크 모듈에 한정하여 적용하였다. 앞으로 리눅스 커널 전반적인 안정화를 위하여 네트워크 모듈뿐만 아니라 다른 모듈에도 확장하여 적용하는 연구를 진행하고자 한다.

## 참고문헌


[1] 권수호, *Linux Programming Bible*, 글로벌, 2002

[2] 장승주, 김해진, 김길용, "*마이크로 커널 기반 운영체제에서 고장 감내 연구*", *한국정보처리학회 추계학술발표 논문집 제3권 제2호*, pp.408-411, 1996

[3] Jeffery Oldham & Alex Samuel, *Advanced Linux Programming*, Mark Mitchell, 2001

[4] John Mehaffey, *Montavista Linux Carrier Grade Edition[White Paper],Montavista Software* Inc., April 8, 2002

[5] Tim Udalll, *"Kernel Hardening Guidelines"*, Sequoia, 1994

[6] Silberschatz & Gavin & Gagne, Operating System Concepts(6th), John Neilley & Songs Inc. 2002.

[7] Software Fault Tolerant, http://user.chollian.net/~hsn3/korea/study_k2.html, 2000

[8] http://www.mvista.com/cge/index.html, 2002

[9] The Linux Online, http://www.linux.org

[10] Gary Nutt, *Kernel Projects for Linux, Addison Wesley Longman*, 2001

[11] A.Rubini & J.Corbet, *Linux Device Driver(2nd)*, O'Relly, 2001

[12] Bovet & Cesati, Orelilly, *Understanding the Linux Kernel*, 2001


## 저자소개


**장 승 주(Jang, Seung Ju)**

1985년 부산대학교 계산통계학(전산학) 학사
1991년 부산대학교 계산통계학과(전산학) 석사

1996년 부산대학교 컴퓨터공학과 박사
1987년~1996년 한국전자통신연구원 시스템 S/W연구실
1993년~1996년 부산대학교 시간강사
2000년~2002년 University of Missouri at Kansas City, visiting professor
1996년~현재 동의대학교 컴퓨터공학과 부교수
※관심분야 : 운영체제, 임베디드 운영체제, 분산시스템, 시스템 보안