리눅스 운영체제에서 커널 스택의 복구를 통한 커널 하드닝

장 승 주[†]

요 약

리눅스 운영체제에서 커널 개발자의 오류로 인하여 발생되는 시스템 정지 현상을 줄이기 위해서 커널 하드닝이 필요하다. 그러나 기존의 커널 하드닝 방식은 고장 감내 기능을 통하여 제공되고 있기 때문에 구현이 어려울 뿐만 아니라 많은 비용이 소요된다. 본 논문에서 제안하는 커널 하드닝 방식은 패닉이 발생한 커널 주소에 대한 커널 스택 내의 값들을 정상적인 값으로 복구하기 위해서 커널 내의 panic() 함수 등 커널의 일부분을 수정하므로, 적은 비용으로 시스템 가용성을 높일 수 있다. 제안한 방식의 실험을 위하여 네트워크 모듈에 강제적인 패닉 현상 유발시키고, 잘못된 스택 값을 정상적인 값으로 복구하는 동작을 확인하였다.

키워드 : 리눅스 운영체제, 커널 하드닝, 스택 복구

Kernel Hardening by Recovering Kernel Stack Frame in Linux Operating System

Seung-Ju Jang[†]

ABSTRACT

The kernel hardening function is necessary in terms of kernel stability to reduce the system error or panic due to the kernel code error that is made by program developer. But, the traditional kernel hardening method is difficult to implement and consuming high cost. The suggested kernel hardening function that makes high availability system by changing the panic() function of inside kernel code guarantees normal system operation by recovering the incorrect address of the kernel stack frame. We experimented the kernel hardening function at the network module of the Linux by forcing panic code and confirmed the proposed design mechanism of kernel hardening is working well by this experiment.

Key Words : Linux O.S, Kernel Hardening, Stack Recovery

1. 서 론

리눅스 운영체제는 open source인 리눅스 커널 소스를 수정하여 파일 시스템, 디바이스 드라이버 등을 커널에 추가할 수 있게 하므로, 임베디드 시스템 분야에서 널리 이용되고 있다. 기업에서도 웹서버, 파일 서버, DB 서버 등으로 활용되고 있다. 하지만 리눅스 운영체제는 많은 사람이 공개적으로 커널의 내용을 수정함으로써 커널 소스나[1~3] 커널 모듈 프로그램을 잘못 작성한 경우에는 시스템 동작 중에 비정상적으로 정지되는 현상이 발생하고, 심각한 경우 데이터가 파괴되기도 한다[13, 14]. 일반적으로 UNIX 운영체제는 하드웨어, 운영체제 커널, 응용 프로그램 환경의 계층적인 구조를 가지고 있으며. 각 계층마다 고유의 특성을 가진 다양한 형태의 고장이 발생되므로 각 계층에 대해서 별도의 독립적인 고장 관리 체계를 제공해야 한다[2, 5].

커널 하드닝은 임의의 커널 내 오류(fault)에 따른 panic에 적절히 대처할 수 있는 기법으로 커널 코드의 적절성 여부에 대한 시험을 통해서 에러 코드에 오류를 줄이는 과정이다[1, 3, 5, 6, 11, 12]. 이러한 기능은 운영체제 커널 내에서 잘못된 코드로 인하여 시스템이 정지되는 것을 정상적으로 동작되도록 하기 때문에 시스템 가용성을 높게 하여서, 안정된 시스템 사용을 보장할 수 있다. 기존의 커널 하드닝 방식은 고장감내 기능을 통하여 제공되고 있다. 고장감내 기능은 많은 자원들을 이중화시킴으로써 시스템 안정화를 도모하고 있다. 따라서 이것은 구현하기가 어려울 뿐만 아니라 많은 비용이 소요된다. 본 논문에서 제안하는 커널 하드닝 기능은 고장감내 시스템과 달리 구현이 간단하므로 많은 운영체제에서 구현될 수 있다.

본 논문은 리눅스 운영체제 커널에서 복구 가능한 오류에 대해서 복구가 될 수 있도록 하는 커널 하드닝 기능을 보인다. 커널 내에서 "panic"이 발생되면 주 기억장치 주소가 잘못

---

† 정 회 원 : 동의대학교 컴퓨터공학과 부교수
논문접수 : 2004년 9월 9일, 심사완료 : 2006년 5월 16일

된 경우에 잘못된 주소 값이 레지스터에 저장된다. 제안하는 기능은 "panic"이 발생될 경우에 커널 스택 프레임 내에 저장된 비정상적인 레지스터 값들을 정상적인 값으로 복원하는 것이다. 이 주소 값을 복원할 수 있을 경우에 정상적인 동작이 가능하도록 하고, 주소 값을 복원할 수 없는 경우에는 기존의 방식과 같은 과정을 거치도록 한다. 이 방식은 커널 내의 panic() 함수 등 커널의 일부분을 수정하므로, 적은 비용으로 시스템 가용성을 높일 수 있다. 제안한 방식의 실험을 위하여 네트워크 모듈에 강제적인 패닉 현상 유발시키고, 잘못된 스택 값을 정상적인 값으로 복구하는 동작을 확인하였다.

2장에서 커널 하드닝과 관련한 연구를 살펴보고, 3장에서 제안한 커널 하드닝 기능을 소개한다. 4장에서는 네트워크 모듈에 제안된 커널 하드닝 기능을 구현하여 실험한 결과를 설명한다. 마지막으로 5장에서 결론을 내린다.

2. 관련 연구

몬타비스타(MontaVista)는 커널 하드닝 기능이 내장되어 있는 CGE(Carrier Grade Edition) 버전의 리눅스 운영체제를 상업적으로 판매하고 있다[9]. 몬타비스타의 CGE 버전은 커널 하드닝 기능을 <표 1>과 같이 3가지 영역으로 분류하고 있다.

〈표 1〉 CGE 버전의 커널 하드닝 기능

1) Code reviews 코드 재검토 기법을 통해서 오류가 발생하는 부분을 찾음 2) Panic Removal StandarAd Linux Code를 검사한 후 시스템을 중지 시킬 것인지 아니면 process를 kill시킬 것인지를 결정 3) Fault Injection Testing Software fault인 경우 error에 대해서 kernel이 복구할 수 있는 능력이 있는지 없는가에 대해서 검사

몬타비스타의 커널 하드닝 기능은 코드를 재검토한 후에 특정 프로세스가 panic 루틴으로 들어왔을 때 panic 루틴으로 들어오면 시스템이 정상적으로 수행 되도록 하는 것은 아니다. 현재 프로세스가 시스템에 영향을 주는 프로세스인 경우이면 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. 그러므로, 몬타비스타의 커널은 시스템의 적합성에 대한 판단으로 시스템 오류가 발생한 모든 조건의 kernel panic에 대한 검토를 포함하고 있으므로[4, 10], 커널의 가용성 기능이 약한 반면 커널 문제에 대한 관리 측면이 강한 운영체제이다[2, 5, 14, 15].

3. 커널 하드닝 개발

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 커널이 동작가능 하도록 하는 것이다. 커널 내에서 "panic"이 발생되면 주소 타입인 경우에 잘못된 주소 값이 cr2 레지스터에 저장이 된다. 이 주소값을 복원해 줌으로써 정상적인 커널 동작이 가능하도록 한다. 이와 같이 잘못된 연산에 대해서 커널 스택 프레임을 복구함으로써 정상적인 동작이 되도록 개발한다. 리눅스 운영체제에서 구현한 커널 하드닝 기능을 네트워크 모듈에 구현하였다. 최근 리눅스 네트워크 기능의 이용한 안정적인 커널 동작을 필요로 하고 있다. 따라서 리눅스 커널내의 네트워크 기능의 안정적인 커널 서비스를 보장하기 위하여 본 논문에서 제안한 커널 하드닝 기능을 ip_input.c 커널 소스 코드의 ip_rcv() 함수에 구현하였다.

3.1 panic() 과정에서 개발

리눅스 운영체제에서 구현한 커널 하드닝 기능을 네트워크 모듈에 개발하였다. 최근 리눅스 네트워크 기능의 안정적인 커널 동작을 필요로 하고 있다. 따라서 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. 사용자가 ping 명령어 등을 수행하는 경우에 동작이 되는 함수이다.

telnet 접속시 접근하게 되는 경로를 찾고자 먼저 Loopback을 이용하여 커널에 panic 발생되도록 linux/net/ipv4/ip_input.c에 다음 (그림 1)과 같은 기능을 삽입하였다.

```
static int harden_count;  //panic이 되기 전의 상황을 확인하기 위해
                            임의의 static 변수값
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 강제적으로 발생하도록
                             하여 panic을 유발시킨다.
} else {
    printk("harden_count=%d\n", harden_count);
}
```

(그림 1) 강제 panic 발생을 위한 ip_rcv() 내의 기능

(그림 1)과 같이 ip_rcv()함수 내에 panic이 강제적으로 발생되도록 하고, panic이 발생된 과정을 역순으로 찾아내기 위해 /linux/kernel/panic.c소스 코드를 중심으로 스택 정보를 출력한다. fault.c내의 함수에는 good_area, bad_area, no_context, do_sigbus의 각 부분에 대한 printk문을 찍어 확인해 보았다. 우선 good_area, bad_area, no_context, do_sigbus에 대하여 살펴보면,

(good_area :) 부분은 프로세스 주소 영역 안에서 발생한

```
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 정리 단계

faulty address를 다룬다.

(bad_area :) 부분은 프로세스 주소 밖에서 발생한 faulty address를 다룬다.

(no_context :) 부분은 인터럽트이거나 또는 커널 쓰레가 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 2)는 do_page_fault()함수 내의 (no_context :) 부분에서 panic 처리의 마지막 단계이다.

(그림 2)는 no_context 부분에서 panic 처리의 마지막 단계를 나타낸다. 이 부분이 panic 처리의 최종 처리단계이다. (그림 3)은 실제 no_context를 호출하는 부분이다.

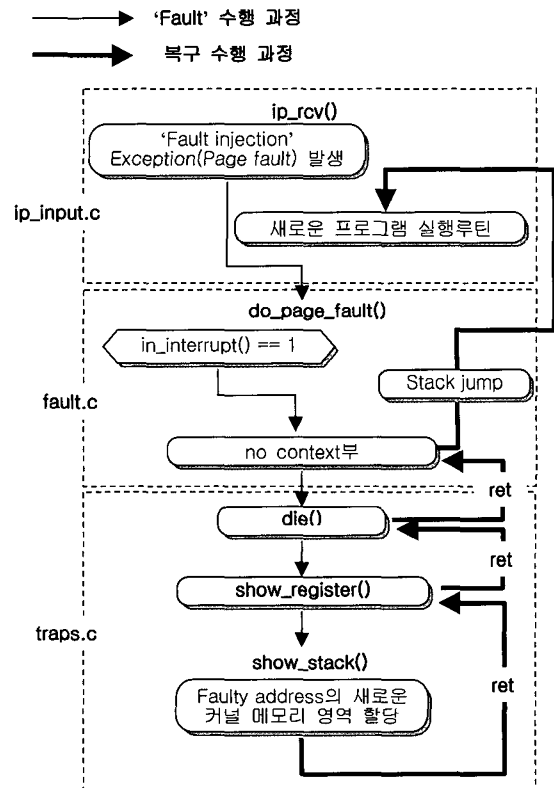
```
if (in_interrupt() || mm == &init_mm)
      //printk("in_interrupt()=%d",in_interrupt())
      //printk("address = %x \m", address);
      goto no_context;
```

(그림 3) panic()에서 no__context 부분의 동작

(그림 3) 부분이 현재 만들어 놓은 panic상황에서 나타나게 되어 no_context부분으로 이동하게 된다. 위의 if문을 만족하는 조건은 in_interruput()의 값으로 이 부분이 no_context부분으로 가는 조건임을 확인할 수 있다. char형 포인터 aa에 우리가 10+10, 즉 20이라는 decimal address값을 주었으므로 이를 확인하기 위해 no_context로 가게 하는 if문 사이에 address값을 출력해 보면 16진수 0x14가 출력됨을 확인할 수 있다. 이 값이 포인터에 설정해 논 address값임을 확인할 수 있다.

3.3 네트워크 모듈에서 커널 하드닝 개발

본 논문은 네트워크 모듈에 커널 스택의 복구 기능을 이용해서 커널 하드닝 기능을 개발한다. (그림 4)중에서 본 논문에서 구현하고자 하는 네트워크 모듈인 ip_input.c에서의 호출 과정을 나타낸다.



(그림 4) 네트워크 모듈 중 ip__rcv() 함수에서 커널 스택 복구 과정

우리가 임의로 만들어 놓은 panic은 kernel 영역에서 처리하는 영역 이외의 메모리 영역에 대한 참조로 일어나게 되는 현상이다. 그 과정이 ./arch/i386/mm/fault.c의 do_page_fault() 함수를 거쳐서 panic이 거치게 되는 경로를 찾으려고

최초 작업을 진행하게 되었다. 이는 예외적 상황의 발생으로 주소값에 대한 영역을 판단함으로써 처리하게 되는 Page Fault Exception Handler를 거치게 된다.

do_page_fault() 함수의 no_context레벨이 panic을 일으키게 하는 부분에서 거치게 되는 함수이다. 이 함수에서 in_interrupt() 함수의 인터럽트 처리 루틴을 거치게 된다. do_page_fault() 함수에서 다음과 같은 어셈블리 명령을 실행하도록 한다.

___asm___("movl %%cr2,%0)":"=r" (address));

이 어셈블리 명령에서 address가 바로 우리가 설정해 놓은 잘못된 주소값을 가지고 있다. 이 주소값은 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. 우선은 패닉이 일어났을 때 스택 값을 추적해 본다. 이는 "nm -n"이라는 유닉스 명령어인 심볼 보기 옵션을 통해서 시스템 패닉(system panic)시에 있어났던 스택 값를 통해 panic이 발생한 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_context부분에서 die() 함수를 호출한다. die() 함수에서는 show_register()함수를 호출한다. show_register() 함수는 show_stack() 함수를 호출한다. 이 show_stack() 함수에서 문제의 cr2레지스터 값을 확인할 수 있는데, 우리는 이 부분에 스택의 주소 값이 address와 같게 되는 부분을 확인할 수 있다. 이 주소 값을 정상적인 값으로 복구하고, 스택 내의 잘못된 경로 호출 과정을 정상적인 과정으로 복구해 주면 정상적인 시스템 동작이 가능하다.

다음 (그림 5)는 show_stack() 함수의 소스 코드를 보여준다.

(그림 5)는 리눅스 커널에서 커널 스택의 복구를 통한 하드닝 기능을 구현하기 위한 show_stack() 함수를 보여준다. (그림 5)에서 cr2 레지스터에 들어가 있는 주소 정보를 주소(address) 변수에 저장한다(SS4 문장). 주소 변수는 패닉이 발생된 값을 담고 있는 변수이다. 스택 내에 들어있는 값중에서 주소 변수 값과 일치하는 값이 있을 경우에 이 값을 정상적인 값으로 변경을 해 주면 리눅스 커널의 동작이 정상적으로 이루어진다. 이 과정이 (그림 5)의 SS10 문장에서 SS15 문장까지의 과정이다.

4. 실 험

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

3장에서 개발한 show_stack() 함수의 내용을 실제 이용

```
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 5) show__stack() 함수 소스 코드

하여 실험을 수행하였다. 이 실험은 본 논문에서 개발한 내용이 정상적으로 동작되는지를 확인하기 위한 실험이다. show_stack() 함수를 ip_rcv() 함수에서 (그림 6)과 같이 호출되도록 한다.

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

(그림 6) show__stack() 함수를 이용한 실험

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

본 논문에서는 esp레지스터의 값을 이용하여 커널 하드닝을 구현한다. esp레지스터를 show_stack()함수를 통하여 이용하게 되면 특정 로컬 변수가 가지는 주소값이 스택에 차례대로 들어가는 것을 알 수 있다.

ip_rcv()에서의 로컬변수의 주소를 알아보고, 페이지 폴트를 발생시킨 aa 라는 로컬 변수가 가지는 스택의 위치를 찾아본다. 실제 실험을 위하여 커널 내에 강제적인 패닉을 유발하도록 하기 위하여 (그림 1)과 같이 커널을 수행한다.

aa가 가지는 주소값으로 인하여 시스템은 패닉을 유발되게 된다. 이런 상황에서 do_page_fault() 함수를 수행하게 되는걸 알 수 있었는데, traps.c에서 새로운 aa변수에 대해서 정상적인 주소값(메모리)을 할당해 줌으로써 fault.c까지 정상적으로 다시 귀환하는 것을 알 수 있다.

(그림 8)은 esp 레지스터 값을 sp라는 변수로 읽어낸다. 그리고 show_stack(sp) 함수를 이용하여 스택 복구를 시도한다. 그 다음으로 정상적인 스택 정보를 배열한다. (그림 8)

```
__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 주소값을 정상적으로 복구

과 같이 함으로써 비정상적인 스택의 값을 정상적인 스택 값으로 복구한다.

본 실험은 본 논문에서 제안하는 커널 스택에 대한 복구를 통한 커널 하드닝 기능 개발을 확인하는 과정이다. 본 논문에서 제안하는 커널 하드닝 기능은 간단히 고장 감내 시스템을 개발할 수 있도록 해 준다.

5. 결 론

리눅스 운영체제 커널에서 잘못된 연산이나 잘못된 프로그램으로 인하여 시스템이 정지되는 현상이 발생한다. 이러한 현상을 시스템 패닉이라고 하는데, 리눅스 커널에서 패닉이 발생할 경우에 잘못된 주소로 인한 패닉에 대해서 정상적인 주소 값을 복원함으로써 정상적인 시스템 동작이 되도록 하는 것은 매우 중요하다. 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 주소값을 복원할 수 있을 경우에 복원해 줌으로써 정상적인 동작이 가능하도록 한다. 리눅스 운영체제에서 구현한 커널 하드닝 기능을 네트워크 모듈에 구현하였다. 네트워크 모듈에서 강제적인 패닉 유발 상황을 만들어 놓고, 패닉이 발생했을 경우 복구를 할 수 있도록 한다. 본 논문은 리눅스 운영체제에서 복구 가능한 오류에 대해서는 복구가 될 수 있도록 커널 하드닝 기능을 설계한다. 많은 사용자들이 리눅스 운영체제를 사용하고 있는데 본 논문에서 제안하는 기능은 보다더 리눅스 커널을 안정화시킬 수 있을 것이다.

앞으로 연구 과제는 리눅스 커널에 본 논문에서 제안하는 기능을 구현하여 모듈화하는 작업이 필요하다. 커널 하드닝 기능의 모듈화를 통하여 사용자가 원하는 시점 쉽게 사용이 가능하도록 해야 한다.

참 고 문 헌

[1] 권수호, Linux programming bible, pp.20-35, 글로벌, 2002.

[2] 장승주, 김해진, 김길용, “마이크로 커널 기반 운영체제에서 고장 감내 연구”, pp.408-411, 한국정보처리학회 추계학술발표논문집 제3권 제2호, 1996.

[3] Jeffery Oldham & Alex Samuel, Advanced Linux Programming, pp.45-55, Mark Mitchell, 2001.

[4] John Mehaffey, “Montavista Linux Carrier GradeEdition [WHITE PAPER]”, Montavista Software Inc., April 8, 2002.

[5] Tim Udalll, "kernel Hardening Guidelines", Sequoia, 1994.

[6] SILBERSCHATZ&GALVIN&GAGNE, Operating System Concepts(6th), JOHNWILEY&SONGS INC. 2002.

[7] Ivan Buetler, “Compass Security Hardening Solaris”, SunMicro System, 2001.3.

[8] John Mehaffey, "MontaVista Linux Carrier Grade Edition Version 2.1 : The Foundation for Next-Generation Carrier Grade Applications," Montavista Software Inc., 2002.
[9] ichael Beck,Mirko Dziadzka,Ulrich Kunitz and Harald Bohme, Linux Kernel Internals, Addison-Wesley,1997.
[10] The Linux Online, http://www.linux.org
[11] Gary Nutt, Kernel Projects for Linux, Addison Wesley Longman, 2001.
[12] A.Rubini&J.Corbet, Linux Device Driver(2nd), O'Relly, 2001.
[13] BOVET & CESATI, OREILLY, Understanding the Linux Kernel, pp.216-222, 2001.
[14] G.B.Adams III, and H.J.Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems, pp.443-454. IEEE Trans. on Comput. Vol. C-31, No.5 May 1982.
[15] http://hpc.postech.ac.kr/~dolphin/research/ds/mighty/design/designfault.html, 2000.
[16] Beck, Linux Kernel Programming, pp.2-5, ADDISON WESLEY,2002.

장 승 주

e-mail : sjjang@deu.ac.kr
1985년 부산대학교 계산통계학과 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, 보안 정형 기법