리눅스 운영체제 안정화를 위한 커널 하드닝 기능 설계

장승주*
*동의대학교 컴퓨터공학과
e-mail:sjjang@deu.ac.kr

Design of the Kernel Hardening Function for Stability 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 위한 커널 하드닝 기능을 설계한다. 본 논문에서 제안하는 커널 하드닝 기능은 문제가 발생한 커널 부분을 수행 중인 프로세스에 대한 동작을 정지시키는 기능과 오류가 발생한 코드에 대한 변수 값이나 주소 값이 가진 특정한 값을 복구시키는 기능을 가진다. 커널 하드닝 기능에서 문제가 있는 모든 프로세스를 무조건 복구하는 것이 아니라 복구 가능성을 판별하여, 복구 가능한 프로세스에 대해서만 복구 될 수 있도록 한다. 또한 오류가 발생한 커널 코드에 대해서 복구 가능한 경우에는 ASSERT() 함수에서 복구가 가능하도록 설계하였다.

Ⅰ. 서론

최근에 리눅스 운영체제의 사용이 증가되고 있다. 리눅스 운영체제는 공개 소스(open source)의 특징을 이용하여 임베디드 시스템 분야에서도 널리 이용하고 있으며 기업에서는 웹 서버, 파일 서버, DB 서버 등으로 활용되고 있다. 리눅스 운영체제의 특징은 리눅스 커널 소스를 수정하여 파일 시스템, 디바이스 드라이버 등을 커널에 추가할 수 있다. 리눅스 운영체제는 많은 사람이 공개적으로 커널의 내용을 변경, 수정함으로써 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 있을 경우에 시스템이 정지되는 현상이 발생한다. 심각한 경우 데이터가 파괴되기도 한다 [13].

커널 하드닝 기능은 운영체제 커널 내에서 잘못된 코드로 인하여 시스템이 정지되는 것을 정상적으로 복구함으로써 시스템이 정상 동작되도록 하는 것이다. 본 논문은 리눅스 운영체제 커널에서 복구 가능한 오류에 대해서 복구가 될 수 있도록 커널 하드닝 기능을 설계한다. 본 논문에서 제안하는 리눅스 커널 하드닝 기능으로 현재 프로세스를 kill하여 시스템이 정상적으로 동작할 수 있도록 하는 기능, ASSERT() 함수 내에서 복구 가능한 메모리 공간에 대해서 복구하도록 하는 기능을 갖고 있다.

본 논문의 구성은 2장에서 관련 연구를 살펴보고, 3장에서 커널 하드닝 설계, 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 [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 판매하고 있다 [3,7,8,9]. 몬타비스타의 CGE 버전은 커널 하드닝 기능을 3가지 영역으로 분류하고 있다. [표 1]은 몬타비스타에서 발표한 CGE 버전의 커널 하드닝 기능을 나타낸다 [3].

[표 1] 몬타비스타에서 배포한 CGE 버전의 커널 하드닝 기능
[Table 1] Hardening Function on CGE Version of Montavista

커널 하드닝 기능	세부 설명
Code Reviews	커널 코드를 설계 및 구현하고 후, 지속적인 점검을 통해서 원천적으로 커널 코드의 오류를 방지 (코드 재검토)

Panic Removal	리눅스 운영체체제에서 코드를 검사한 후 시스템을 중지(panic) 시킬 것인지 아니면 프로세스를 kill시킬 것인지를 결정
Fault Injection Testing	소프트웨어 오류인 경우 리눅스 커널이 복구할 수 있는 능력이 있는지 없는지에 대해서 검사

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.

이와 같이 커널 하드닝은 시스템 고가용성(high availability)을 보장하고자 하는데 목적이 있다. 커널 하드닝 기능은 임의의 커널 내의 오류(fault)에 따른 panic에 적절히 대처할 수 있는 기법으로 code path 시험을 통한해서 에러 코드에 오류를 줄이는 과정이다. 이러한 개념을 fault injection이라고 한다. 이러한 실험적 방법을 통하여 구현하게 되는 커널 리소스를 통해 보다 안정된 운영체제 커널 코드를 생성할 수 있다 [1, 3, 6].

커널 하드닝과 관련한 기타 관련 연구로 시쿼이아에서 유닉스 운영체제에 커널 하드닝 기능을 설계한 적이 있다 [5]. 이 경우는 커널 하드닝 관점보다도 시스템 가용성 측면에서 운영체제를 설계한 것이다.

III. 커널 하드닝 설계

3.1 커널 하드닝 기능

본 논문에서는 커널 하드닝 기능을 설계하였다. 본 논문에서 설계한 커널 하드닝 기능은 시스템에 문제가 발생하여 시스템이 정지되어야 하는 경우에 현재 프로세스를 검사하여 복구가 가능한 프로세스인지를 판단한다. 복구가 가능하다고 판단된 프로세스의 경우는 현재 프로세스를 kill하여 시스템이 정상적으로 동작할 수 있도록 하며, 그렇지 않는 경우라면 panic() 함수를 수행하여 시스템이 더 이상 동작되지 않도록 한다. 본 논문에서 제안하는 리눅스 커널 하드닝 기능의 설계를 통해서 보다 안정적인 커널 동작을 보장하고 안정적인 시스템 운용을 보장한다.

3.2 커널 하드닝 설계

본 논문에서 커널 하드닝을 구현하기 위하여 ASSERT() 매크로 함수를 이용한다. 본 논문에서 제안하는 ASSERT() 매크로 함수는 수식 (1)과 같다.

$$ASSERT() = \begin{cases} expr2 = expr3, \text{if } expr4 = 1 \; And \\ \quad \text{EXP}R(expr1) = TRUE \\ panic(), \text{if } expr1 \neq 1 \text{ or} \\ \quad \text{if EXP}R(expr1) = FALSE \end{cases} \quad \text{...수식(1)}$$

수식 (1)에 대한 동작 흐름은 [알고리즘 1]에서 보여준다. 본 논문에서 제안하는 리눅스 커널 하드닝 복구는 ASSERT() 매크로 함수를 통해서 이루어진다. panic() 함수의 수행 여부는 ASSERT() 매크로 함수에서 결정하므로 만약 ASSERT() 매크로 함수에서 현재 프로세스를 복구할 수 있을 경우에 복구하고 정상적으로 동작하게 한다면 시스템은 아무런 이상 없이 정상적으로 동작할 수 있을 것이다. 아래는 커널 하드닝에서 복구 가능 여부를 판단하는 경우를 나타낸 것이다.

① ASSERT() 함수에서 expression이 값 형태(value type)인 경우는 모두 복구가 가능하다.

② 주소 형태(address type)인 경우에 대해서는 현재 프로세스가 사용자 프로세스인 경우에는 복구하도록 하고 시스템 프로세스인 경우에 대해서는 panic() 함수를 수행하도록 한다.

본 논문에서는 기존 리눅스의 ASSERT() 매크로 함수의 인자를 1개에서 4개로 변경했다. 기존의 ASSERT() 매크로 함수의 인자는 ASSERT(expr)으로 되어 있다. 하지만 인자 값이 하나인 경우에는 ASSERT() 함수에서 복구를 할 수 있는 정보를 알 수 없기 때문에 ASSERT() 매크로 함수의 인자를 추가해 주었다. 즉, 본 논문에서는 기존의 ASSERT(expr1)을 ASSERT (expr1, expr2, expr3, expr4)로 변경한다. ASSERT() 함수의 첫 번째 인자 expr1은 ASSERT() 함수에서 정상적인 값과 잘못된 값의 판단 여부에 사용된다. expr3은 panic이 발생하지 않은 정상적인 상태에서의 조건값을 나타낸다. expr2가 잘못된 값을 가질 경우에 expr3값을 expr2에 대입함으로써 정상적인 복구가 가능하도록 한다. 마지막으로 expr4는 현재 ASSERT() 함수가 값 형태인지 주소 형태인지를 구분하는데 사용한다. 만약 expr4가 1이면 값 형태로 정의된 경우이고, expr4가 2이면 주소 형태로 정의된 경우이다.

[알고리즘 1]은 주소 형태(address type)에서의 커널 하드닝 기능에서 복구 과정을 나타낸다.

```
Algorithm Recovery-Address-Type
Input : The set of the E = {expr1, expr2, expr3, expr4} and
        process id(Pid)
Output : kill process or execute panic()

Step 1. if 주소 형태 then Step 2.
        else goto Step 7.
        end
Step 2. if expr1이 FALSE(실제 잘못된 주소값을 가질
        경우) then
            goto step 6.
        end
Step 3. expr2와 expr3이 정상적인 주 기억장치 주소의 주소
        값인지를 판단
        if expr2 와 expr3이 정상적인 주소값일 경우
            then goto Step 4.
        end
Step 4. if Pid = user process then Step 5.
        end
Step 5. force_sig_kill(Pid)
Step 6. Panic()
Step 7. Stop
```

[알고리즘 1] 주소 형태에서 커널 하드닝 복구 과정
[Algorithm 1] Kernel Hardening Recovery Procedure in Address Type

[알고리즘 1]은 주소 형태(address type)에서의 커널 하드닝 복구 과정을 나타낸다. ASSERT() 함수에서 expr4가 2인 경우가 주소 인 경우이다. address type인 경우에는 먼저 expr1이 TRUE인지 FALSE인지를 검사한다. 이 값이 TRUE인 경우에는 조건식이 정상적인 경우로써 ASSERT() 함수 내에서 어떤 동작도 필요가 없다. 만약 expr1 조건식이 FALSE인 경우라면 현재 expr2와 expr3의 메모리가 정상적으로 존재하는 메모리인지를 검사하도록 한다. 메모리가 올바른지를 검사하는 함수는 커널에서 제공하는 access_ok() 함수를 사용한다.

access_ok() 함수의 인자는 3개를 가지게 된다. 첫 번째 인자로 read 메모리인지 혹은 write 메모리인지에 관한 정보를 나타낸다. 두 번째 인자로 메모리 주소이고, 마지막 인자로 메모리 주소의 사이즈 값을 주게 된다.

access_ok() 함수에서 리턴 값이 0인 경우 정상적으로 사용가능한 주소 값인 경우이고, 만약 1이 리턴되는 경우라면 비정상적인 메모리 주소 값이다. 만약 expr2와 expr3중 하나라도

리턴 값이 1이라면 비정상적인 메모리 주소이므로 panic() 함수를 수행 하도록 해주고, 리턴 값이 0인 경우라면 정상적인 메모리 주소로 처리되도록 한다.

[알고리즘 1]에서 정상적인 메모리 주소인 경우 현재 프로세스가 시스템 프로세스인지 사용자 프로세스인지를 판단하게 된다. 만약 시스템 프로세스인 경우라면 이 프로세스를 kill 할 경우 시스템에 더 큰 피해를 끼치므로 기존의 방식대로 panic() 함수를 수행 하도록 한다. 사용자 프로세스인 경우에는 현재 프로세스를 kill하여 시스템이 정상적으로 동작할 수 있도록 해준다.

3.3 개발 환경

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.

3.4 ASSERT() 함수의 수정

커널 하드닝 기능을 구현하기 위해서 리눅스 커널 ASSERT() 매크로 함수를 본 논문에서 제안한 설계 내용으로 수정하였다. 수정된 ASSERT() 함수는 복구가 가능하다고 판단된 경우 시스템이 정상적으로 동작할 수 있도록 구현되었다. 리눅스 운영체제에서 ASSERT() 매크로 함수의 인자 값(expr)을 받아서 ASSERT() 함수 내에서 최종 시스템 복구 여부를 판단하게 된다. 특히, expr로 들어오는 값이 TRUE라면 정상적인 경우이므로 ASSERT() 함수를 빠져나오게 하고, FALSE인 경우라면 비정상적인 경우이므로 panic() 함수를 수행하도록 한다. 본 논문에서는 expr이 FALSE인 경우에 정상적인 복구 여부를 최종 판단한 후 복구여부를 결정하게 된다. [그림 1]은 일반적으로 리눅스 운영체제에서 사용하는 ASSERT() 함수 소스 코드를 나타낸다.

```
#define ASSERT(expr) do {\
        if(expr) {\
            printk("\n case of true ASSERT \n");\
        }else{\
            printk("\n ASSERT check false routine\n");\
            printk("Assertion[%s]failed %s:%s(line=%d)\n"\
            #expr1,__FILE__,__FUNCTION__,__LINE__);\
 }}while(0)
```

[그림 1] 일반적인 ASSERT() 함수 소스 코드
[Figure 1] Source Code of ASSERT() Function

커널 하드닝을 구현하기 위해서 [그림 1]의 일반적인 ASSERT() 함수 본 논문에서 제안하는 기능을 추가한다. ASSERT() 함수의 인자로 사용하는 expr의 type에는 address type과 value type이 존재하게 된다. [그림 1]에 대해서 커널 하드닝 기능을 구현한 소스 코드가 [그림 2]이다. [그림 2]에서 expr1의 조건이 FALSE인 경우는 address type인 경우, 주소 값이 존재하지 않는 경우이므로 기존의 커널 방식에서는 panic() 함수를 수행하여 시스템이 정지되게 될 것이다. 그리고 value type인 경우라면 메모리 주소는 올바르나 메모리 주소의 데이터가 잘못된 경우이므로, 이 데이터 값만 변경한다면 시스템은 정상적으로 동작할 수 있을 것이다.

```
#define ASSERT(expr1,expr2, expr3,expr4) do {₩
  if(Address Type인 경우) {₩
      if(expr1 조건이 거짓인 경우) {₩
          printk("ASSERT check --> false routine₩n");₩
          if(올바르지 않은 메모리인 경우){₩
            printk("current->pid : %d₩n",current->pid);₩
            if(현재 프로세스가 사용자 프로세인 경우){₩
               이 프로세스만 kill₩
             }else{₩
                panic() 함수 수행₩
             }₩
          }₩
          panic() 함수 수행₩
      }₩
  }₩
  else if(Value Type인 경우) {₩
      printk("ASSERT Funcion --- Value Type₩n");₩
      if(expr1) {₩
           printk("ASSERT Check Value Check TRUE₩n");₩
      } else {₩
           expr2 = expr3;₩
      }₩
  }₩
}while(0)
```

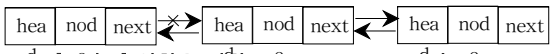
[그림 2] 커널 하드닝을 적용한 ASSERT() 매크로 함수의 소스 코드
[Figure 2] Source Code of ASSERT() Function with Kernel Hardening

[그림 2]는 본 논문에서 제안한 커널 하드닝 기능을 ASSERT() 함수를 이용하여 설계한 내용이다. 커널 내에서 잘못된 주소값을 가진 경우에 이 주소 값이 사용 가능한 주소 영역 범위 내에 존재할 경우이면서 사용자 프로세스인 경우는 시스템 복구가 되도록 한다. 사용자 프로세스가 아니고 시스템 프로세스인 경우는 커널 panic()을 유발한다. 또한 메모리 주소 값이 정상적인 주소 범위내의 값이 아닐 경우는 시스템 panic()을 유발한다. ASSERT()를 수행하는 함수가 주소값 형태가 아니고 값 형태인 경우에 해당 변수가 잘못된 값을 가진 경우는 정상적인 값으로 강제적으로 복구를 한 후에 정상적인 커널 수행이 되도록 한다.

3.5 ip_input.c 수정

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. 사용자가 ping명령어 등을 수행하는 경우에 동작이 되는 함수이다. 본 논문에서 설계한 커널 하드닝 기능을 설계 및 실험하기 위하여 ip_input.c 소스 코드에 double linked list를 구현하였다. 이 링크 자료 구조를 통해서 강제적인 시스템 오류 상황을 만든다. 강제적인 오류 상황 유발은 double linked list의 특정 노드의 next 필드의 값을 다음 노드의 주소가 아닌 다른 주소나 NULL 값을 가질 경우 커널 하드닝 기능이 없는 리눅스 운영체제의 경우 시스템이 비정상적으로 동작하여 시스템이 정지되게 될 것이다.

[그림 3]의 double linked list에서 중간의 노드가 끊어져서 다음 노드로 진행하지 못할 경우 다음 주소를 찾을 수 없게 되므로 ASSERT() 매크로 함수를 통해서 fault.c의 do_page_fault() 함수를 수행한다. 이때 do_page_fault() 함수는 die() 함수를 수행하여 시스템은 panic() 상태가 된다. 강제적인 시스템 구축 환경인 double linked list의 경우 현재 노드에서 다음 노드로 가는 주소는 끊어졌지만 다음 노드에서 현재 노드로 오는 previous 주소가 올바른 경우라면 현재 노드의 주소와 다음 노드의 next 필드와 일치시켜 준다면 정상적인 커널 동작을 보장할 수 있을 것이다. 이 경우는 panic 루틴을 수행하지 않고 시스템을 정상적으로 수행할 수 있다.



[그림 3] 실험을 위한 Double Linked List의 구조
[Figure 3] Double Linked List Structure for Experiment

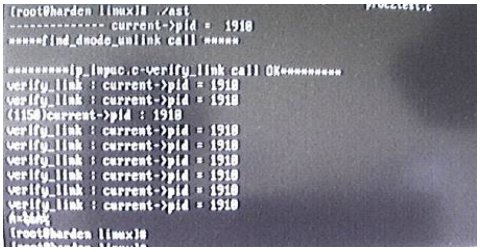
IV. 실험

4.1 실험 방법

본 논문에서 제안하는 커널 하드닝 설계 내용을 리눅스 운영체제에서 구현하여 실험하였다. 본 논문에서 설계한 커널 하드닝 기능을 추가한 커널을 컴파일한 후 부팅할 경우 커널이 부팅되면서 처음으로 ip_rcv() 함수를 수행 할 경우 double linked list를 초기화한다. double linked list가 동작이 되고 난 후에 ip_rcv() 함수를 수행하게 되면 콘솔에 나타나는 메시지는 [그림 4]와 같다. [그림 4]는 리눅스 커널에 double linked list에 노드를 추가했을 경우 콘솔에 출력되는 정보다.

```
******* harden_count = 16 *************
15   16   64 bytes from localhost.localdomain (127.0.0.1): icmp_seq
= 4
ttl=255 time=0.050 ms
******* harden_count = 17 *************
15   16   17   verify_link : current->pid = 1196
******* harden_count = 18 *************
15   16   17   18   verify_link : current->pid = 1196
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq = 4
ttl=255
time=0.087 ms
******* harden_count = 19 *************
15   16   17   18   19   verify_link : current->pid = 1196
******* harden_count = 20 *************
15   16   17   18   19   20   verify_link : current->pid = 1196
******* harden_count = 21 *************
15   16   17   18   19   20   21   verify_link : current->pid =
1196
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq = 4
ttl=255
time=0.085 ms
```

[그림 4] 리눅스 커널에서 ip_rcv()함수를 수행했을 때 콘솔에 나타나는 정보
[Figure 4] Console Print Information during ip_rcv() Function is Execution



[그림 5] ast 프로그램 실행 결과
[Figure 5] Execution Result Screen of ast Program

[그림 5]는 ast 사용자 프로그램을 수행한 후 ping 명령어를 실행했을 때 콘솔에 출력되는 정보를 보여준다. ast 사용자 프로그램은 리눅스 커널에 구현해 놓은 double linked list에서 특정 링크의 연결을 끊고자 할 경우에 사용하는 프로그램이다. 특정 노드의 next 필드에 NULL을 대입하여 다음 노드와 연결을 차단한다.

ast 사용자 프로그램을 수행시 double linked list에서 끊고자 하는 노드 번호를 입력하면 리눅스 커널의 ip_rcv() 함수에 정의된 double linked list의 지정된 노드의 next 필드의 값을 NULL로 한다. 본 논문에서는 3번째 노드의 링크 정보를 인위적으로 파괴시킨다. 이렇게 설정을 하게 되면 double linked list는 이 후에 정상적으로 동작될 수가 없다. ast 사용자 프로그램을 수행한 후 ping 명령어를 수행할 경우 ip_rcv() 함수의 double linked list를 출력하지만 [그림 5]에서와 같이 2번째 노드까지만 출력하고 3번째 노드에서 노드의 next 필드 값이 NULL이므로 잘못된 주소로 판단된다.

이때 커널 하드닝 기능이 구현되어 있지 않은 리눅스 운영체제인 경우 panic이 발생하여 시스템이 더 이상 동작이 되지 않는다. 커널 하드닝 기능이 구현되어 있는 커널의 경우는 실험에서와 같은 복구 가능한 환경에 대해서는 이를 복구하게 된다. 복구가 된 커널은 정상적인 동작을 하게된다.

V. 결론

ASSERT() 함수를 이용하여 커널 내에서 사용하는 변수의 유형에 따라 값 유형(value type)인 경우에 잘못된 값을 가진 변수에 대해서 정확한 값으로 강제 설정을 하는 경우와 주소 유형(address type)인 경우에 복구 가능한 주소 인지를 판단하여 복구 가능한 주소인 경우에 복구를 수행하고 그렇지 않을 경우는 정상적으로 시스템 panic()을 유발한다. 본 논문에서 제안하는 리눅스 커널 하드닝을 리눅스 운영체제에서 실험하였다. 실험 결과 커널 하드닝 설계한 부분이 인위적인 커널 동작의 오류에도 불구하고 정상적인 동작을 함을 확인할 수 있었다.

참고 문헌

[1] 권수호, Linux programming bible,pp20-35, 글로벌, 2002.
[2] 장승주, 김해진, 김길용, “마이크로 커널 기반 운영체제에서 고장 감내 연구”, pp.408-411, 한국정보터리학회 추계학술발표 논문집 제3권 제2호, 1996.
[3] Jeffery Oldham & Alex Samuel, Advanced Linux Programming, pp45-55, Mark Mitchell, 2001.
[4] John Mehaffey, Montavista Linux Carrier GradeEdition [WHITE PA PER],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] Software Fault Tolerant, http://user.chollian.net/~hsn3/korea/study_k2.html, 2000.
[8] http://www.mvista.com/cge/index.html, 2002.
[9] Michael 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 L-ongman, 2001.