리눅스 네트워크 모듈에서 커널 하드닝 기능 설계

장승주*

Design of the Kernel Hardening Function in the Linux Network Module

Seung-ju Jang*

2003년도 한국전자통신연구원 연구비 지원에 의하여 연구가 이루어졌음
이 논문은 2003년도 Brain Busan 21사업에 의하여 지원되었음

요 약

본 논문은 리눅스 커널 네트워크 모듈에서 커널 개발자의 실수나 의도하지 않은 오류 등으로 인하여 발생되는 시스템 정지 현상 또는 시스템 패닉 현상을 줄이기 위한 커널 하드닝 설계 내용을 제안한다. 본 논문에서 제안하는 네트워크 모듈 내의 커널 하드닝 기능은 문제가 발생한 리눅스 네트워크 모듈을 수행 중인 프로세스에 있어서 문제가 발생한 코드의 주소 형태와 값 형태에 따라 복구를 하도록 하는 것이다. 값과 주소에 대한 잘못된 수행으로 시스템이 정지되는 현상을 줄임으로써 안정적인 네트워크 기능 동작을 보장한다. 오류가 발생한 커널 코드에 대해서 복구 가능한 경우에는 ASSERT() 함수에서 복구가 가능하도록 설계되었다.

Abstract

A panic state is often caused by careless computer control. It could be also caused by a kernel programmer's mistake. It can make a big problem in computer system when it happens a lot. When a panic occurrs, the process of the panic state has to be checked, then if it can be restored, operating system restores it, but if not, operating system runs the panic function to stop the system in the kernel hardening O.S. To decide recovery of the process, the type of the panic for the present process should be checked. The value type and the address type have to restore the process. If the system process is in a panic state, the system should be designed to shutdown hardening function in the Linux operating system. So it has to decide whether the process should be restored or not before going to the panic state.

I. 서 론

최근에 리눅스 운영체제의 사용이 증가되고 있다. 리눅스 운영체제는 open source의 특징을 이용하여 많은 사람들이 널리 이용하고 있으며 기업에서는 웹서버, 파일 서버, DB 서버 등으로 활용되고 있다. 리눅스 운영체제의 특징은 리눅스 커널 소스를 누구나 수정하여 파일 시스템, 디바이스 드라이버, 네트워크 기능 등을 커널에 추가할 수 있다는 것이다. 그러나 이러한 리눅스의 특징은 많은 사람이 공개적으로 커널의 내용을 변경, 수정함으로써 안정화되어 있지 않다 [1, 2, 3, 4, 5, 6]. 특히 리눅스 운영체제는 비 상업적인 목적으로 개발되었기 때문에 상업적인 목적의 운영체제보다 커널 코드

*동의대학교 컴퓨터공학과
접수일자 : 2004. 4. 29

가 안정적이지 못하다. 또한 리눅스 운영체제 커널 소스를 수정하거나 커널 모듈 프로그램을 잘못한 경우에는 시스템 동작에 치명적일 수 있다. 잘못된 코드가 커널에 있을 경우에 시스템이 정지되는 현상이 발생된다. 심각한 경우에는 데이터가 파괴되기도 한다 [2, 13, 14, 15, 16, 17].

본 논문은 리눅스 커널내 네트워크 모듈에서 복구 가능한 오류에 대해서 복구가 될 수 있도록 커널 하드닝 기능을 설계한다. 본 논문에서 설계한 리눅스 커널내 네트워크 모듈 하드닝 기능은 ASSERT() 함수 내에서 복구 가능한 메모리 공간에 대해서 복구하도록 하는 기능을 갖고 있다.

ASSERT 함수는 두가지 형태에 의하여 분류된다. 하나는 주소값에 의한 분류이고, 다른 하나는 값에 의한 분류이다. 주소값에 의한 분류는 "ASSERT(new != NULL, return -1;);"의 경우로써 new 변수가 "char *"타입이다.값에 의한 분류는 "ASSERT(self->magic== LAP_MAGIC, return;); "과 같이 self->magic의 타입이 정수형으로 선언된 경우이다. ASSERT() 함수를 이용하여 커널 내 네트워크 모듈에서 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 논문의 구성은 2장에서 관련 연구를 살펴보고, 3장에서 커널 하드닝 설계, 4장은 실험 결과에 대해서 설명하고, 마지막으로 5장은 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.

지금까지 상용화된 대표적인 고장 감내 시스템으로 Tandem, Fujitsu, Stratus, DEC 등을 꼽을 수 있다. Tandem 은 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. 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.

Tandem NonStop 시스템은 무정지 연산 시스템의 구현을 기본 목표로 하고 있다. 따라서 모든 시스템 구성 요소에 대해서 이중 구조를 가지고 있다. 파일 시스템도 역시 마찬가지이다. 이러한 시스템 구조는 UI HAWG(UNIX International Hardware Working Group)에서 분류한 가용성에 따르면 완벽한 고장 감내 시스템이라고 할 수 있다. Tandem NonStop 운영체제에서는 시스템의 구조가 동일한 디스크(화일 시스템)에 대해서 이중의 접근 경로를 제공하기 때문에 이러한 기능을 제공하기 위한 고장 감내 기능이 있어야 한다. Tandem NonStop 운영체제는 primary process 와 backup process 가 있다. Primary process는 주기적으로 "I'm alive"라는 메시지를 발생시켜서 고장이 없음을 알린다. Backup process는 primary process에 고장이 발생했을 때 primary process를 대신하여 동작한다. 결함 복구 기능은 check point 기능을 사용한다. Backup process는 primary process에 이상이 발생했을 경우에 마지막 checkpoint 에 정의된 상태에서 다시 수행한다. 파일 시스템의 결함 복구 기능도 마찬가지이다.

Chorus 운영체제는 개방, 분산, 가변의 특성을 지향하는 운영체제이다. 이러한 특성을 만족하기 위하여 마이크로 커널 개념을 채택하고 있다. 마이크로 커널 운영체제는 기존의 운영체제에서 제공하는 여러 가지 기능을 독립적인 서버에서 제공할 수 있도록 하고 있다. Chorus는 이런 특성을 만족하기 위하여 프로세스 간 비동기 메시지 교환과 RPC(Remote Procedure Call)를 지원한다. Chorus 운영체제에서 파일 시스템 고장 감내 기능은 takeover 기능을 이용한 고장 감내 기능이다. Takeover 기능은 동일한 기능을 하는 2개의 프로세스(process)가 동작한다. 2개의 프로세스는 주 프로세스와 보조 프로세스의 역할을 담당한다. 보조

프로세스는 주 프로세스에 이상이 발생하면 기능을 대체한다. 이 프로세스는 각 서버에 이상이 발생하면 기능을 대체한다. 이 프로세스는 각 서버에 대해서 적용된다. Chorus 운영체제에서 파일 시스템 기능을 하는 서버는 FM(File Manager) 서버이다. FM 서버가 2개 동작하면서 고장 감내 기능을 제공한다 [2].

현재까지 커널 하드닝 관련 연구는 많이 이루어지고 있지 않다. 최근의 리눅스 커널 하드닝 관련 연구 중 몬타비스타의 연구가 대표적이다. 몬타비스타는 이미 커널 하드닝 기능이 내장되어 있는 CGE(Carrier Grade Edition) 버전의 리눅스 운영체제를 상업적으로 판매하고 있다 [3,6,7,8,9]. 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 운영체체 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.

몬타비스타의 커널 하드닝 기능은 Code Reviews를 통해서 코드를 재검토한 후 특정 프로세스가 panic 루틴으로 들어왔을 때 Standard Linux Code의 검사를 통해 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.

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 통한, 즉 에러 코드에 대한 모순되지 않은 과정을 피해보고자 하는 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].

커널 하드닝과 관련한 기타 관련 연구로 시퀘이아에서 유닉스 운영체제에 커널 하드닝 기능을 설계한 적이 있다 [4,10,11,12]. 이 경우는 커널 하드닝 관점보다도 시스템 가용성 측면에서 운영체제를 설계한 것이다.

III. 네트워크 모듈에서 커널 하드닝 설계

3.1 커널 하드닝 설계

본 논문에서 제안하는 커널 하드닝의 기본 개념은 시스템에 문제가 발생하여 panic() 함수로 들어가기 전에 잘못된 커널 코드를 복구할 수 있는지를 판단하여 복구할 수 있는 오류인 경우라면 이를 복구하여 시스템이 정상적으로 동작하도록 하는 것이다. 이렇게 함으로써 불안정한 시스템을 보다 안정된 시스템으로 만들 수 있다. 그러나 일반적으로 커널 오류가 비정상적인 경우는 복구를 하지 않고 panic() 함수로 유도를 하는 것이 시스템에 치명적인 결과를 초래하지 않는다. 이런 경우는 기존의 커널과 같이 panic이 발생되도록 유도를 한다. 본 논문에서는 커널 하드닝 기능을 구현하기 위하여 ASSERT() 매크로 함수를 이용한다. 본 논문에서 제안하는 커널 하드닝 기능을 위한 ASSERT() 매크로 함수의 수행 과정은 다음 [그림 1]과 같다.

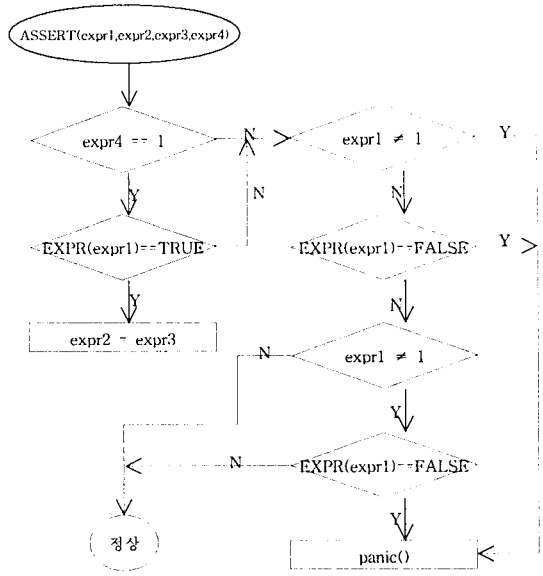

그림 1. 커널 하드닝 기능을 위한 ASSERT() 함수 수행 과정

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.

표 1. 커널 하드닝 기능으로 복구 가능한 형태

커널 하드닝으로 복구 가능한 형태	ASSERT() 함수에서 expression이 값 형태(value type)인 경우는 모두 복구가 가능
	주소 형태(address type)인 경우에 대해서는 접근 가능한 주소인지를 판단하여 접근 가능한 주소인 경우에 복구 가능

커널 하드닝을 통해서 복구가 가능한 경우에 대한 과정은 다음과 같다. ASSERT() 매크로 함수를 통해서 들어오는 수식(expression)은 값 형태와 주소 형태의 두 가지 종류를 가질 수 있다. 값 형태인 경우에는 현재 커널을 수행하는데 필요한 데이터가 있지만 이 값은 정상적인 값과 일치하지 않기 때문에 ASSERT() 매크로 함수에서 잘못된 데이터 값만 정상적으로 변경 한다면 panic() 함수를 수행하지 않고 정상적으로 복구가 가능할 것이다.

ASSERT() 내 인자값인 expr1의 표현식이 FALSE 일 경우는 주소 형태인지 값 형태인지에 따라서 연산을 수행한다. expr1은 "expr2==exp r3" 의 결과에 의하여 결정이 되는 값이다. expr1이 FALSE 인 경우에 복구과정에서 값 형태인 경우는 expr2에 잘못 설정된 값을 expr3의 정상적인 값으로 설정하여 이루어진다. 값 형태가 아니고 주소 형태인 경우는 인자값(expr2,expr3)이 커널 내 사용 가능한 메모리 영역인지에 대한 검증을 access_ok()함수를 통해 검증한 후 복구 가능한 여부를 판단하게 한다. expr2와 expr3가 링크를 연결하는 올바른 주소는 아니지만 정상적 메모리 영역의 주소임을 확인하게 되면, 이는 panic으로 처리하는 것이 올바르다. 왜냐하면, 두 인자(expr2,expr3) 중 어느 값이 올바른 주소 값인지 찾는 것이 힘들다.

다음으로 두 인자(expr2,expr3) 중 하나라도 NULL값을 가지게 될 시에 처리 방법이다. 이는 커널에서는 access_ok()함수를 통해 NULL값을 가진 인자, 즉 접근할 수 없는 비 정상적 메모리 영역에 대한 값을 가지고 있을 시에 이 루틴을 실행하게 된다. 비교되는 두 개의 인자값이 모두 NULL값을 형성하게 될 시에는 force_sig()을 이용하여 링크체크 루틴을 무시하고 그 프로세스를 kill 함으로서 시스템에 아무런 일이 일어나지 않았던 것처럼 처리를 한다. 단순히 panic으로 처리를 하는 것보다 force_sig()를 사용하여 panic을 유발시킬 수 있는 프로세스를 kill하고 그 이외의 작업은 정상적으로 진행할 수 있도록 한다. pid값을 통하여 유저 프로세스인지에 대한 값을 정확히 구하는 것은 쉽지가 않다. 유저 프로세스로 식별되는 프로세스는 force_sig()함수를 적용하여 처리한다. 시스템 프로세스(또는 daemon process)에 대해서 force_sig()를 이용하여 프로세스를 종료시킬 경우는 시스템이 비 정상적인 종료를 시키는 것과 같다. 따라서 시스템프로세스인 경우는 프로세스를 종료시키는 경우보다 "panic" 처리를 하는 것이 바람직하다.

[알고리즘 1] RAT_KH_NM 알고리즘으로 주소 형태(address type)에서의 네트워크 모듈에서 커널 하드닝 기능 복구 과정을 보여준다.

Algorithm RAT_KH_NM(Recovery-Address-Type for Kernel Hardening in Network Module)
Input : The set of the Expr = {expr1, expr2, expr3, expr4} and process id(pid)
Output : recovery variable, kill process or execute panic()

RAT1 : if address type then **RAT2**
else **RAT7**
end
RAT2 : expr1이 FALSE(실제 잘못된 주소값을 가질 경우)
RAT3 : expr2와 expr3이 정상적인 주 기억장치 주소의 주소 값인지를 판다.
if expr2 와 expr3이 정상적인 주소값일 경우
then **RAT4**
else
goto **RAT6**
end
RAT4 : if pid = user process then **RAT5**
else
goto **RAT6**
end
RAT5 : force_sig_kill(pid)
RAT6 : Panic()
RAT7 : Stop

알고리즘 1. 주소 형태에서 커널 하드닝 복구 과정

[알고리즘 1]은 주소 형태(address type)의 네트워크 모듈에서 커널 하드닝 복구 과정을 나타낸다. [알고리즘 1]에서 왼쪽에 표시된 것(RAT)은 알고리즘 내에서 문장 번호를 나타내는 것이다. ASSERT() 함수에서 expr4가 2로써, address type인 경우에는 먼저 expr1이 TRUE인지 FALSE인지를 검사한다. 이 값이 TRUE인 경우에는 조건식이 정상적이므로 ASSERT

() 함수에서 어떤 동작도 취할 필요가 없이 기존의 ASSERT() 함수 기능과 같이 정상적인 과정을 수행하면 된다. 만약 expr1 조건식이 FALSE인 경우라면 현재 expr2와 expr3의 메모리가 정상적으로 존재하는 메모리인지를 검사하도록 한다. 메모리가 올바른지를 검사하는 함수는 커널에서 제공하는 access_ok() 함수로써 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 경우는 정상적으로 사용 가능한 주소 값인 경우이고, 만약 1이 리턴되는 경우라면 비정상적인 메모리 주소 값이다. expr2와 expr3중 하나라도 리턴 값이 1이라면 비정상적인 메모리 주소이므로 panic() 함수를 수행 하도록 해주고, 리턴 값이 0인 경우라면 정상적인 메모리 주소로 처리되도록 한다.

3.2 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.

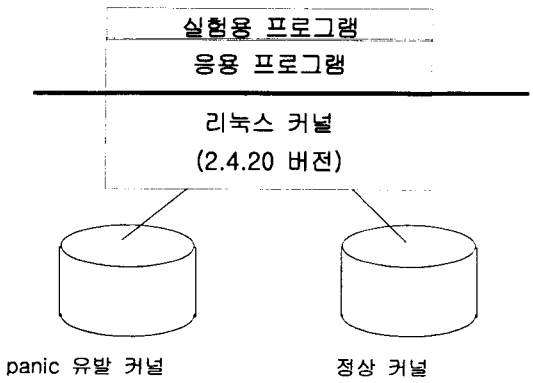

그림 2. 개발 시스템 환경

3.3 리눅스 네트워크 모듈내 하드닝 설계

리눅스 네트워크 모듈 내에서 기존의 ASSERT() 매크로 함수의 형태를 분류하면 네가지로 나눌 수 있다. 이 네가지 유형은 다음 [표 2]와 같다.

표 2. 네트워크 모듈 내에서 ASSERT 유형

	route.c ip_rt_init()	tcp_ipv4.c tcp_v4_init()	tcp.c tcp_init()	tcp_diag.c tcpdiag_init()	icmp.c icmp_init()
ASSERT_TCP_CACHE	O		O		
ASSERT_TCP_PAGE	O		O		
ASSERT_NETLINK				O	
ASSERT_CREATE		O			O

[표 2]는 네트워크 모듈에 구현한 커널 하드닝 기능을 ASSERT() 함수별로 분류하여 구현한 내용이다. ASSERT_TCP_CACHE()와 ASSERT_TCP_PAGE() 매크로 함수는 네트워크 기능에서 TCP 관련 기능을 하는 부분에서 커널 하드닝을 구현한 함수이다. 이 매크로 함수는 route.c와 tcp.c 소스 코드에서 구현되었다. ASSERT_TCP_ CACHE() 매크로 함수는 기존의 리눅스 커널에서 kmem_cache_create() 함수를 사용하는 네트워크 모듈에 적용된다. ASSERT_TCP_PAGE() 매크로 함수는 __get_free_pages() 함수를 사용하는 네트워크 모듈에 적용된다. ASSERT_NETLINK() 매크로 함수는 TCP 프로토콜에서 네트워크 기능 진단에 사용하는 함수에 적용된다. ASSERT_CREATE() 매크로 함수는 TCP 프로토콜과 ICMP 프로토콜에 커널 하드닝 기능을 적용하는 함수로 사용된다.

Linux/net/ipv4 경로 내에 존재하는 소스들 중 panic을 유발하는 부분이 있다. 이중에서 tcp.c 소스 파일내의 tcp_init()함수에서 panic()을 유발하는 부분이 있다. tcp_init()함수에서 특정 변수 값이 NULL일 경우 시스템은 panic()을 유발하게 된다. 이들 각각의 변수들은__get_free_pages() 또는 kmem_cache_create() 함수를 수행한 후 반환되는 값에 의해서 결정된다. route.c 소스 파일에도 이와 같은 형태의 panic() 유발 코드가 있다. 이러한 유형의 변수들의 형태를 살펴보면 다음 [그림 3]과 같다.

```
tcp_openreq_cachep =
  kmem_cache_create("tcp_open_request",sizeof(struct
                                    open_request),0,
                  SLAB_HWCACHE_ALIGN,NULL, NULL);
if(!tcp_openreq_cachep)
  panic("tcp_init: Cannot alloc open_request cache."
-------------------------------------------------------
tcp_bucket_cachep =
  kmem_cache_create("tcp_bind_bucket",sizeof(struct
                                    tcp_bind_bucket),0,
                  SLAB_HWCACHE_ALIGN, NULL, NULL);
                                    if(!tcp_bucket_cachep)
panic("tcp_init: Cannot alloc tcp_bind_bucket cache.");
-------------------------------------------------------
tcp_timewait_cachep =
  kmem_cache_create("tcp_tw_bucket",sizeof(struct
                                    tcp_tw_bucket),0,
                  SLAB_HWCACHE_ALIGN,NULL, NULL);
if(!tcp_timewait_cachep)
  panic("tcp_init: Cannot alloc tcp_tw_bucket cache.");
-------------------------------------------------------
tcp_ehash = (struct tcp_ehash_bucket *)
  __get_free_pages(GFP_ATOMIC, order);
if(!tcp_ehash)
  panic("Failed to allocate TCP established hash
  table₩n");
-------------------------------------------------------
tcp_bhash = (struct tcp_bind_hashbucket *)
  __get_free_pages(GFP_ATOMIC, order);
if(!tcp_bhash)
  panic("Failed to allocate TCP bind hash table₩n");
```

그림 3. 네트워크 모듈에서 panic() 을 결정하게 되는 변수 유형

If 조건에 따라 panic()을 수행할 수도 있고 정상적인 동작을 할 수도 있다. 만약 할당 받은 변수 값의 조건을 만족하여 panic()을 수행할 경우에 커널 하드닝 기능을 설계한다. 즉, 새로 설계한 ASSERT() 매크로 함수를 적용한다. ASSERT() 매크로 함수 내의 사용하게 될 인자값에 이들 각각의 조건에 판단되는 변수들을 어떻게 적용하여 새로운 ASSERT() 매크로 함수를 정의한다. 새롭게 정의되는 ASSERT() 매크로 함수는 다음 [그림 4], [그림 5]와 같다.

```
#define    ASSERT_TCP_CACHE(expr1,expr2,expr3,
                            expr4,expr5,expr6,
                            expr7,expr8) \
        if(!expr1){\
                expr1  =  kmem_cache_create(expr2,
                          expr3,expr4,expr5,expr6,ex
                          pr7);\
                if(expr1 == NULL){\
                        expr8=1;\
                }else{\
                        expr8=0;\
                }
        }
```

그림 4. kmem_cache_create 함수를 사용하는 경우에 새롭게 정의된 ASSERT 함수

```
#define      ASSERT_TCP_PAGE(expr1,expr2,expr3,
                             expr4) \
        if(!expr1){\
                expr1  =  __get_free_pages(expr2,
                                           expr3);\
                if(expr1 == NULL){\
                        expr4=1;\
                }else{\
                        expr4=0;\
                }\
        }
```

그림 5. __get_free_pages 함수를 사용하는 경우에 새롭게 정의된 ASSERT 함수

ASSERT() macro 함수 내에서 판단되는 조건을 살펴보면, ASSERT(a != NULL, a, NULL, 2)일 경우를 생각해 보자. ASSERT() macro 함수 내에서 복구 가능한 주소 값에 대한 판단을 할 수 없을 경우는 단순히 "panic"처리를 하는 것보다, "force_sig(SIGKILL, p_pid)"를 사용하여 "panic"을 유발할 수 있는 프로세스를 종료시키는 것이 더 효율적이다. force_sig() 함수를 사용하여 프로세스를 종료시키는 경우는 사용자 프로세스에 대해서만 적용이 되도록 한다. 시스템 프로세스(또는 daemon process)에 대해서 force_sig() 함수를 이용하여 프로세스를 종료시킬 경우는 시스템을 비정상적으로 종료시키는 것과 같다. 따라서 시스템 프로세스인 경우는 프로세스를 종료시키는 경우보다 "panic" 처리를 하는 것이 안전한 방법이다. 사용자 프로세스와 시스템 프로세스를 구분하는 방법은 proc table의 p_pid값을 이용하여 각각의 프로세스를 구분할 수가 있다.

위 [그림 5]를 기준으로 ASSERT() 구문의 사용 형태를 정리하면 다음 [표 2]와 같다.

표 2. ASSERT구문에 사용되는 인자 값

인자	기능
expr1	expr2 == expr3의 상태 정보를 가짐
expr2, expr3	특정값을 가진 변수또는 주소값
expr4	address인지 value인지를 결정

[그림 5]는 커널 하드닝 기능을 네트워크 기능이 수행되는 함수에 실제 적용한 소스 코드를 보여준다. 커널 하드닝이 적용된 함수는 "ping" 명령어를 실행하면 수행이 되는 "tcp_init()"에 적용하였다.

```
. . . . .
if (!tcp_ehash){
      ASSERT_TCP_PAGE(tcp_ehash,GFP_
                        ATOMIC,order,p_flag);
      if(p_flag==1)
        panic("Failed to allocate TCP
                established hash table\n");
}
for (i = 0; i < (tcp_ehash_size<<1); i++) {
      tcp_ehash[i].lock =
                        RW_LOCK_UNLOCKED;
      tcp_ehash[i].chain = NULL;
}
do {
      tcp_bhash_size = (1UL << order) *
      PAGE_SIZE /sizeof(struct tcp_bind_
                                hashbucket);
      if ((tcp_bhash_size > (64 * 1024)) &&
                                order > 0)
      continue;
      tcp_bhash = (struct tcp_bind_hashbucket *)
          __get_free_pages(GFP_ATOMIC,
            order);
} while (tcp_bhash == NULL && --order >=
  0);
. . . . .
```

그림 6. 네트워크 함수에 커널 하드닝 기능을 적용한 예제 코드

[그림 4]와 [그림 5]에서 ASSERT_TCP_PAGE() 함수가 커널 하드닝 기능 구현을 위한 코드 부분이다. 이 함수에서 tcp_ehash 변수는 _get_free_pages(GFP_ATOMIC, order)을 수행한 후에 리턴 받는 값이 할당된다. p_flag는 tcp_ehash가 할당이 될 경우는 0으로 설정이 되고 할당이 되지 못하고 NULL이 될 경우는 1로 설정된다.

[그림 6]은 기존의 네트워크 관련 함수에서 커널 하드닝 기능을 위하여 ASSERT_TCP_PAGE() 함수를 추가하였다. ASSERT_TCP_PAGE() 함수에서 tcp_ehash, order 변수는 기존의 루틴에서 사용하던 변수이다. GFP_ATOMIC는 메모리 할당을 받기 위해서 기존에 정의된 값이다. p_flag는 새롭게 추가된 변수로써 복구가 불가능할 경우에 이 변수에 1이 반환된다. 이 변수값이 1일 경우는 기존의 방식대로 panic() 함수를 수행한다. 그렇지 않을 경우는 이미 ASSERT_TCP_PA

GE() 매크로 함수에서 커널의 복구가 이루어진 상태이다.

IV. 실 험

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. [그림 6]의 double linked list에서 중간의 노드가 끊어져서 다음 노드로 진행하지 못할 경우에 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 수행이 가능하다.

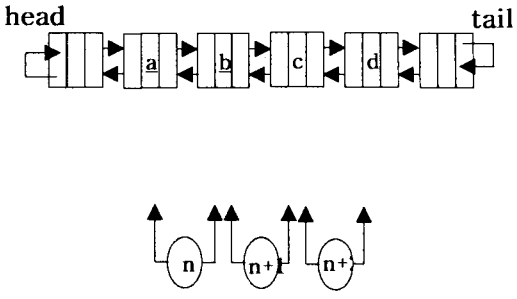

그림 7. 실험을 위한 Double Linked List의 설정 구조

double linked list가 동작이 되고 난후에 ip_rcv() 함수를 수행하게 되면 콘솔에 나타나는 메시지는 [그림 8]과 같다. [그림 8]은 리눅스 커널에 double linked list 노드를 추가했을 경우 콘솔

에 출력되는 정보다.

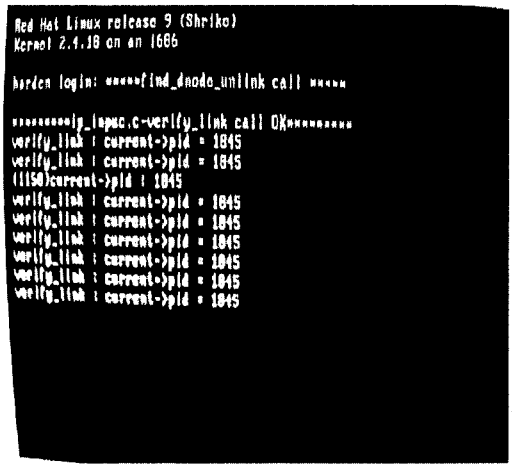

그림 8. 리눅스 커널에서 ip_rcv()함수를 수행했을 때 콘솔에 나타나는 정보

커널에 double linked list를 설정한 후 실험을 위해 작성한 ast 사용자 프로그램을 사용하여 double linked list를 사용자가 원하는 시점에 지정된 링크를 삭제할 수 있다. ast 사용자 프로그램을 이용하여 리눅스 커널의 ip_rcv() 함수에 double linked list의 특정 노드의 next 필드 값에 NULL값을 대입하여 double linked list가 정상적으로 동작이 되지 않도록 한다.

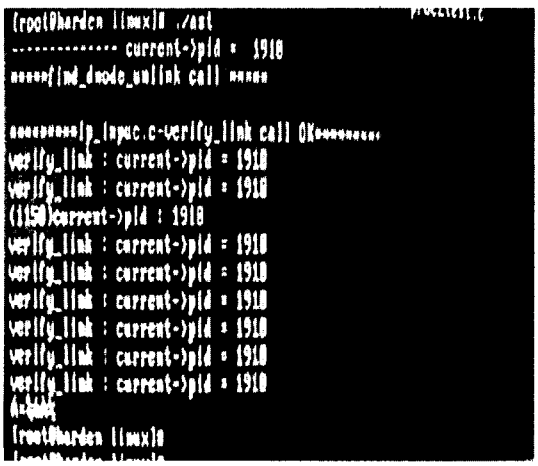

그림 9. ast 프로그램 실행 결과

[그림 9]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.

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 하게 된다.

V. 결 론

운영체제 커널에서 잘못된 연산이나 잘못된 프로그램으로 인하여 문제가 발생하면 시스템이 정지되는 현상이 발생한다. 커널의 문제로 인하여 시스템이 정지된다면 시스템에 큰 문제가 발생할 수 있다. 커널의 문제로 인하여 시스템이 정지되지 않고 정상적으로 동작할 수 있도록 한다면 시스템을 보다 안정적으로 사용할 수 있을 것이다. 본 논문은 리눅스 운영체제에서 복구 가능한 오류에 대해서는 복구가 될 수 있도록 기능을 설계한다. 리눅스 커널에서 복구 가능한 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 논문에서는 리눅스 커널의 ASSERT() 함수를 이용하여 ASSERT() 함수내의 조건식이 FALSE가 발생한 경우에 현재 프로세스의 복구 가능성을 판별하여 복구 가능한 경우는 현재 프로세스를 kill하여 시스템이 정상적으로 동작할 수 있도록 하였다. 그렇지 않고 복구가 불가능하다고 판단이 되는 경우는 panic() 함수를 수행하여 시스템이 정지될 수 있도록 설계 하였다. 본 논문에서 제안하는 리눅스 커널 하드닝을 리눅스 운영체제에서 실험하였다. 실험 결과 커널 하드닝 설계한 부분이 인위적인 커널 동작의 오류에도 불구하고 정상적인 동작을 함

을 확인할 수 있었다.

참고문헌

[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] ichael Beck,Mirko Dziadzka,Ulrich Kunitz and Harald Bohme, Linux Kernel Internals, Add ison-Wesley,1997.

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

[11] Gary Nutt, Kernel Projects for Linux,Addison Wesley L-ongman, 2001.

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

[13] BOVET & CESATI, OREILLY, Understanding the Linux Kernel, p216-p222, 2001.

[14] http://nodevice.com/sections/ManIndex/man 055.html, 2002.

[15] 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.

[16] http://hpc.postech.ac.kr/~dolphin/research/ds/mighty/design/designfault.html,2000.

[17] Beck, Linux Kernel Programming, pp2~5, ADDISON WESLEY,2002.

저자소개

장승주(Seung-ju Jang)

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, 분산시스템, Active Network, 시스템 보안