슈퍼유저의 파일접근 권한으로부터 보호받을 수 있는 리눅스 파일시스템 구현

이 호 정, 이 기 훈, 김 화 종
강원대학교 전자공학과

Implementation of Linux File System to be Protected from Super-User's File Access

Ho-Jung Lee, Ki-Hoon Lee, Hwa-Jong Kim
Dept. of Electronics Engineering, Kangwon National University

Abstract - 최근 인터넷의 급속한 발전과 함께 인터넷에 접속되는 서버의 수가 기하 급수적으로 증가하고 있다. 이러한 인터넷의 보급과 함께 중요한 문제로 부각되는 것이 보안 문제이다. 최근 서버를 리눅스(Linux)로 구축하는 사례가 늘고 있다. 본 논문에서는 현재 널리 사용되고 있는 리눅스의 파일시스템을 보안적인 차원에서 문제점이 무엇인지 분석하고 이를 해결하는 방안으로 리눅스 커널 레벨에서 동작하는 새로운 리눅스 보안 파일시스템을 구현하였다.

1. 서 론

리눅스의 장점은 프로그램 소스코드가 공개돼 있어 프로그래머가 원하는 대로 특정기능을 추가할 수 있고 어느 플랫폼에도 포팅이 가능하다는 것이다.

여러 가지 장점을 가진 리눅스 시스템이지만 기존의 시스템은 슈퍼유저(root) 권한에 제한이 없었다. 기존의 파일시스템은 슈퍼유저의 권한만 있으면 일반유저의 모든 디렉토리와 파일들을 열람할 수 있다. 즉, 파일의 수정 및 삭제가 가능하고 볼 수 있다는 것이다.

이러한 문제점은 개인의 사생활이 담긴 파일(메일, 일기 등)이나, 중요한 기밀문서 또는 소스코드들을 슈퍼유저가 마음대로 처리할 수 있다는 것이다. 이러한 기존 파일 시스템의 문제점 때문에 슈퍼유저가 일반유저의 사적인 문서들을 접근 할 수 없으며 수정 및 삭제가 불가능한 새로운 시스템이 필요하게 되었다.

이러한 문제점을 해결하기 위해서는 파일을 암호화하는 방법이 일반적이다. 하지만 파일을 암호화했다고 해도 슈퍼유저가 파일을 볼 수만 없을 뿐 수정 및 삭제가 가능하기 때문에 위의 문제를 근본적으로 해결할 수가 없다. 또한 현재의 파일 암호화 시스템은 파일을 암호화하기 위하여 자신의 시스템에 암호화 어플리케이션을 설치해야하고, 보호하고 싶은 파일을 일일이 하나씩 암호화해야 한다는 단점이 있다. [표1]에서 현재의 파일 암호화 시스템과 본 논문에서 소개할 새로운 보안 파일시스템을 비교하였다.

본 논문에서는 이러한 문제점과 단점을 해결하기 위해 RedHat Kernel 2.4 버전을 기반으로 기존의 파일 시스템을 수정 보완하여 슈퍼유저의 파일 접근 권한을 기존의 시스템보다 축소하는 방법을 제안하였다. 1장의 서론에 이어 2장에서 기존 파일시스템의 동작원리 개념을 설명하고, 3장에서 새로운 보안 파일시스템에 대해서 알아보고, 새로운 보안 파일시스템의 실행 예를 보이며, 4장에서 결론을 맺는다.

표 1. 보안 파일시스템 비교

	현재의 파일 암호화 시스템	새로운 보안 파일시스템
슈퍼유저의 읽기 가능여부	불가능	불가능
슈퍼유저의 쓰기 가능여부	불가능	불가능
슈퍼유저의 삭제 가능여부	가능	불가능
어플리케이션 설치여부	필요	불필요

2. 기존 파일시스템 동작원리

기존 파일시스템에서 프로세스가 파일에 접근하기 위해서는 우선 파일을 열어야 한다. 파일을 열면 커널 내부에서는 파일 오픈 시스템 콜을 호출하게 된다. [그림1]에서 이러한 파일 오픈 시스템 콜의 제어 흐름 과정을 볼 수 있다. 처음으로 sys_open()을 호출하게 된다. sys_open()은 fs/open.c 파일에 구현되어 있다. 이 함수는 우선 filp_open() 이라는 커널 내부 함수를 호출한다. filp_opne() 함수는 우선 open_namei() 이라는 커널 내부 함수를 호출한다. open_namei() 함수는 인자로 전달된 파일 이름과 디렉토리 구조를 이용해 그 파일에 대응되는 inode를 파일시스템에서 찾아 리턴한다. 그럼 filp_open() 함수는 파일 구조를 위한 메모리 공간을 할당하고 이 구조의 초기화를 수행한다[3].

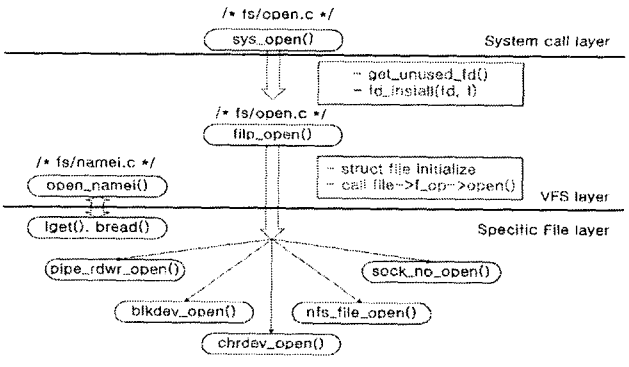


그림 1. 파일 시스템의 시스템 콜 제어 흐름도

2.1 task_struct

프로세스란 실행중인 프로그램이다. 즉 프로그램의 메모리 내 image에 해당한다. 모든 프로그램은 실행되기 위해선 메모리로 loading 되어야 하며, 운영체제는 이러한 프로세스를 관리하기 위한 정보를 내부에 유지한다. 리눅스에서는 task_struct라는 구조체를 이용해서 이러한 프로세스를 관리하며, 프로세스가 실행되기 위한 모든 정보를 다 담고 있어야 한다.

task_struct 구조체를 살펴보면 다음과 같다.

```
struct task_struct {
        volatile long state;
        unsigned long flags;
        int sigpending;
        mm_segment_t addr_limit;

        ...
```

가장 처음에 나오는 것이 프로그램 상태(-1: unrunnable, 0 : runnable, >0 : stopped)를 나타내는 state이다. 그리고 flag 값은 시스템 상태를 나타낼 수 있는데 "PF_SUPPERPRIV" (0x00000100)을 나타내면 super user 권한으로 사용되고 있다는 것을 나타낸다.[1]

```
        ...
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;

        ...
```

프로세스 인증을 나타내는 값으로 uid, euid, suid, fsuid, gid_egid, sgid, fsgid가 있다. 이 값은 프로세스의 증명서 역할을 하는 것으로 접근 제어에 사용된다. [표2]에 각 값들에 대한 기능을 나타내고 있다.

표 2. 프로세스 인증

process credential	설 명
uid	사용자 ID 값
euid	유효 사용자 값
suid	보관된 사용자 ID 값
fsuid	파이 시스템 사용자 ID 값
gid	그룹 ID 값
egid	유효 그룹 ID 값
sgid	보관된 그룹 ID 값
fsgid	파일 시스템 그룹 ID 값

프로세스들은 여러개가 묶여서 프로세스 그룹을 이룬다. 프로세스는 고유한 ID로 pid르 운영체제로부터 할당 받게 되며, 속한 그룹의 값으로는 pgrp를 가진다. 또한 프로세스는 session에 속하게 되며, session에는 leader가 되는 프로세스가 존재하게 된다.

```
        ...
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;
        int leader;
        ...
```

2.2 dentry_struct

dentry 구조체는 해당하는 파일에 대한 디렉토리 정보를 가지고 있는 object이다. 각각의 파일 시스템은 자신의 파일에 묶어서 전체적인 구조를 저장하기 위한 구조를 가지고 있으며, 이 정보는 디스크의 일정 구역에 저장된다. Unix나 Windows NT 및 Linux에서는 이러한 정보를 디렉토리라구 한다.디렉토리의 내용은 각각의 파일을 나타내는 엔트리이다. 이 구조체를 이용하여 보안 디렉토리 파일을 검사할 수 있게 하였다.

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;
        struct inode *d_inode;
        struct dentry *d_parent;
        struct list_head d_vfsmnt;
        struct list_head d_hash;
        struct list_head d_lru;
        struct list_head d_child;
        struct list_head d_subdirs;
        struct list_head d_alias;
        struct qstr d_name;
        ...
}
```

[표3]에 dentry 구조체 각 필드의 기능을 나타내었다.

표 3. dentry 필드 설명

Field	Description
atomic_t d_count	denty object의 사용계수
unsigned int d_flags	denty의 flag
struct inode *d_inode	파이 이름과 관련된 inode에 대한 포인터
struct dentry *d_parent	Parent director에 대한 denty 포인터
struct list_head d_vfsmnt	VFS mount에 대한 연결 구조체
struct list_head d_hash	denty를 찾기 위한 hash 연결 구조체
struct list_head d_lru	사용되지 않는 dentry에 대한 연결 구조체
struct list_head d_child	Parent directory에 포함된 dentry object들에 대한 연결 구조체
struct list_head d_subdirs	하위 directory에 이는 dentry에 대한 연결 구조체
struct list_head d_alias	관련된 inode들에 대한 연결 구조체
struct qstr d_name	파일의 이름

3. 새로운 보안 파일시스템

이 장에서는 앞장에서 설명한 기존의 리눅스 파일 시스템을 기반으로 새로운 리눅스 보안 파일시스템을 구현할 것이다. [그림2]에서 새로운 보안 파일시스템의 기본적인 개념을 나타내고 있다.

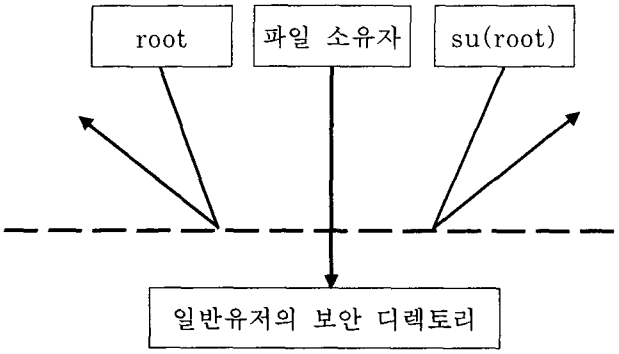


그림 2. 동작원리

[그림2]에서 볼 수 있듯이 보안 디렉토리에 접근 가능한 자는 보안 파일의 소유주뿐이다. 슈퍼유저(root)의 접근과 슈퍼유저(root)의 'su' 명령어를 통한 접근을 보안 디렉토리는 거부한다. 슈퍼유저(root)의 접근은 유저아이디(uid)의 값을 비교하여 판별할 수 있다. 하지만 'su' 명령어를 통한 슈퍼유저(root)의 접근은 유저아이디(uid)의 값만을 비교해서 판별할 수 없다. 왜냐하면, 슈퍼유저(root)가 'su' 명령어를 통해 파일 소유주의 아이디로 접속하기 때문에 유저아이디(uid)의 값이 같아지게 되기 때문이다. 이러한 문제를 해결하기 위해 session을 사용한다.

리눅스에서의 모든 프로세스는 fork()에 의해서 생성되며, 부모(parent) 프로세스를 가진다. 즉, 새로이 생성된 프로세스는 자식(child) 프로세스가 되며, 부모를 가리키는 pointer를 task_struct 에 가진다.

또한, 프로세스들은 여러 개가 묶여서 프로세스 그룹을 이룬다. 이렇게 프로세스 그룹을 만드는 이유는 커널에서 그룹에 속한 프로세스들에게 어떤 작용(action)을 가하기 위한 것이다. 프로세스는 고유한 ID로 pid를 운영체제로부터 할당 받게 되며, 속한 그룹의 값으로는 pgrp를 가진다. 일반적으로 프로세스들은 pgrp값을 부모로부터 상속하며, 각각의 그룹에는 로그인 터미널을 하나씩 가지고 있고 /dev/tty file에 관련짓는다. 리눅스에서는 이러한 여러 개의 프로세스 그룹들을 묶어서 세션(session)을 만들며, 모든 세션은 리더(leader)를 가지게 되는데 바로 이 리더가 세션아이디(sid)가 되는 것이다. [그림3]에서 세션과 프로세스의 관계를 볼 수 있다[1].

보안파일 소유주가 자신의 보안파일에 접근할 경우 세션아이디(sid) 값과 보안파일에 접근하는 프로세스 아이디(pid)와 값이 같다. 하지만, 슈퍼유저(root)가 'su' 명령어를 통해 보안파일 소유주로 로그인 해서 보안파일에 접근하는 프로세스 아이디(pid) 값과 세션아이디(sid) 값이 다르다. 이러한 점을 이용하여 슈퍼유저(root)의 'su' 명령어를 통한 보안파일 접근을 차단할 수 있다.

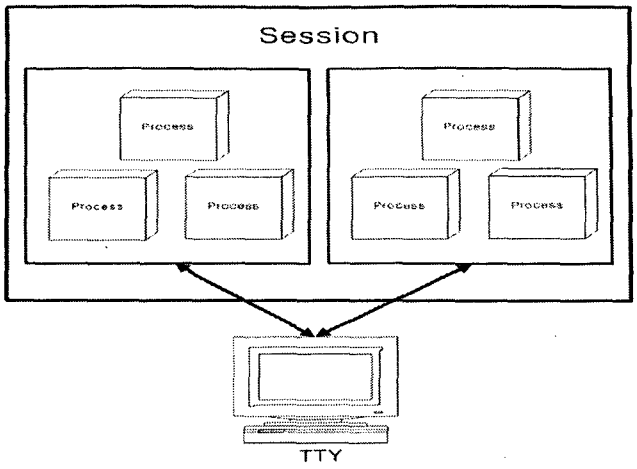
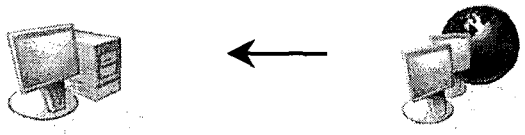


그림 3. 세션과 프로세스

다음은 본 논문에서 구현한 보안 파일시스템의 실행 예이다.



[Server] [Client]

그림 4. 환경

* Server (리눅스 - Redhat Kernel 2.4)
: 새로운 보안 파일시스템이 포함되어있는 리눅스 커널

* Client (OS에 상관없음)
: 터미널을 통해서 서버에 접근

그림 5. 슈퍼유저(root)가 보안 디렉토리에 접근하였을 경우

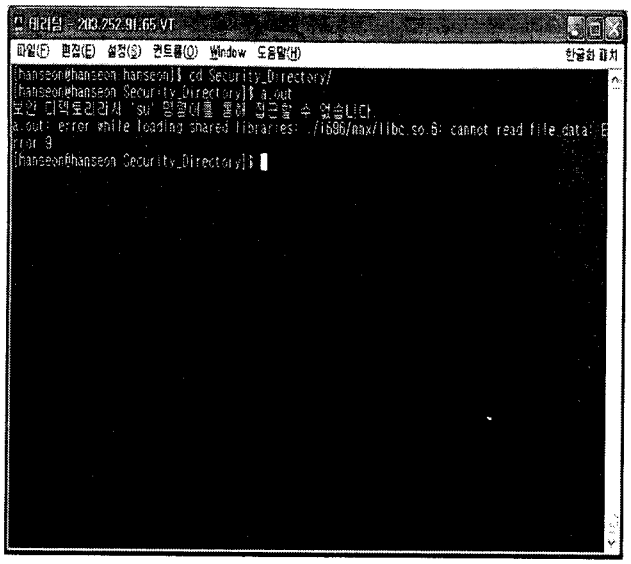


그림 6. 'su'를 통한 일반유저로 접속한 슈퍼유저가 보안 디렉토리에 접근한 경우

〔그림5〕는 슈퍼유저(root)가 보안 디렉토리에 접근하였을 경우이다. 그림에서 알 수 있듯이 "보안 디렉토리라서 root는 접근할 수 없습니다." 라는 메시지를 출력하면서 파일에 접근할 수 없다는 것을 보여주고 있다.

〔그림6〕은 슈퍼유저가 'su' 명령어를 통해 일반유저로 접속한 경우이다. 이것 또한 그림에서 알 수 있듯이 "보안 디렉토리라서 'su' 명령어를 통해 접근할 수 없습니다." 라는 메시지를 출력하면서 파일접근을 거부하고 있다.

4. 결 론

본 논문은 기존의 리눅스 파일시스템에서 슈퍼유저(root)의 절대적인 파일 접근 권한 문제를 해결하기 위하여 새로운 리눅스 보안 파일시스템을 구현하였다. 근본적으로 슈퍼유저의 파일 접근 권한을 축소하기 위해서, 어플리케이션 기반이 아닌 커널 레벨에서 구현하였다. 즉, 새로운 커널 이미지를 만든 것이다.

어플리케이션 기반이 아닌 커널 레벨에서의 구현은 여러 가지 장점을 가진다. 첫째로, 일반유저가 사용하기에 편리하다는 것이다. 일반유저가 어플리케션을 설치하고 셋팅 과정이 필요 없고, 단지 보안 디렉토리에 보호하고 싶은 파일을 옮겨놓기만 하면 되기 때문이다. 둘째로, 일반유저에 거부감이 없다는 것이다. 기존의 파일시스템의 기능과 특징을 모두 가지고 있고, 보안 디렉토리라는 기능만 추가된 것이기 때문이다. 셋째로, 슈퍼유저가 임의로 보안 디렉토리 기능을 삭제할 수 없다는 것이다. 즉, 일반 어플리케이션은 임의로 삭제할 수 있지만, 커널 레벨에 프로그래밍 되어 있는 보안 파일시스템을 커널을 잘 알고 있지 않는 이상 삭제할 수 없다는 것이다.

우리는 장래성 있는 운영체제인 리눅스의 파일시스템 특징과 문제점을 분석하고 이를 해결하기 위해 새로운 파일시스템을 구현하였다. 새로운 보안 파일시스템은 일반유저의 정보를 슈퍼유저로부터 지켜줄 것이다. 앞으로 리눅스의 다른 문제점과 특징들을 분석하여 더욱 안전성 있고, 기능이 향상된 운영체제가 되도록 연구가 필요하다.

〔참 고 문 헌〕

〔1〕 권수호, "Linux Programming Bible", 2002
〔2〕 Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel", 2002
〔3〕 조유근, 최종무, 홍지만, "커널 프로그래밍 : 리눅스 매니아를 위한", 2002
〔4〕 Michael Beck, Harold Bohme, Mirko Dziadzka, "Linux Kernel Programming", 2002
〔5〕 David A. Curry, "Unix System Programming SVR4", 2001
〔6〕 http://bbs.kldp.org
〔7〕 http://kldp.org
〔8〕 http://lachesis.pe.kr/documents
〔9〕 http://tunelinux.pe.kr/sysadmin/kernel/html/tlk.html
〔10〕 http://kernel.pe.kr/home.php
〔11〕 http://man.kldp.org
〔12〕 http://linuxkernel.net
〔13〕 http://joinc.co.kr
〔14〕 http://kltp.kldp.org
〔15〕 http://www.linux.co.kr