데이터 보호를 위한 암호화 파일시스템의 분석

An Analysis of Crypto-File System for Protecting Sensitive Data

임재덕(J.D. Lim) 보안운영체제연구팀 연구원
은성경(S.K. Un) 보안운영체제연구팀 선임연구원
김정녀(J.N. Kim) 보안운영체제연구팀 선임연구원, 팀장

본 논문은 지금까지 제안 및 개발되어 온 암호화 파일시스템에 대하여 살펴본다. 암호화 파일시스템은 사용자 개개인 혹은 조직 등에서 기밀을 유지하여야 하는 중요한 데이터에 대한 안전한 저장을 목적으로 개발되었다. 암호화 파일시스템의 기능으로는 침입자 혹은 원하지 않는 타인의 접근에 대해 데이터의 기밀성 및 안정성을 보장하고, 암호화 기능의 투명화를 통해 사용의 편리성을 제공하며, 암호화 기능의 수행으로 인해 시스템의 성능이 저하되는 것을 방지하는 것 등이 있다. 현재까지 개발되어 온 대표적인 암호화 파일시스템으로는 Cryptographic File System(CFS), Transparent Cryptographic File System(TCFS), Cryptfs, 그리고 Steganographic File System(StegFS) 등이 있다. 차후에는 분석된 암호화 파일시스템을 통해 좀 더 효율적인 보안성과 이식성을 제공하고, 사용자에 대해 편리성을 제공하는 파일시스템 구조의 설계 및 개발이 필요하다.

I. 서 론

최근 인터넷과 같은 네트워크 환경의 발전에 따른 시스템들간의 개방성 증가로 정보의 공유가 일반화되고, 사용자에게 편리함을 제공하는 반면, 개개인 혹은 조직의 중요한 기밀 정보들에 대한 접근 또한 용이해져 해킹과 같은 시스템 침입이 빠르게 증가하는 추세이다[1]. 따라서 시스템의 취약점을 보완하는 패치나 업그레이드를 통한 시스템 침입의 방지 혹은 시스템 수준에서의 접근 통제 기술 등에 의한 중요한 데이터로의 비정상적인 접근을 통제하고 있지만, 항상 위험성이 잠재하고 있다.

따라서, 사용자의 데이터를 암호화하여 저장하는 방법에 큰 관심이 모아지고 있으며 많은 연구가 진행중이다. 데이터를 암호화하여 저장하는 방법은 시스템의 물리적인 보안보다 더 안전하며, 디스크 자체의 도난에 대해서도 중요한 데이터에 대한 유출을 방지할 수 있다[2].

데이터를 암호화하는 방법으로 디스크 전체를 암호화하는 방법, 파일별로 암호화하는 방법 등이 제시되었으나 사용자 편리성, 사용 부주의에 의한 공격 가능성 등 많은 문제점으로 인하여 시스템 자체에서 데이터 암호화를 제공하는 기법 즉, 암호화 파일시스템에 대한 연구가 많이 이루어지고 있다.

대표적인 것으로는 Cryptographic File System(CFS), Transparent Cryptographic File System(TCFS), Cryptfs 그리고 Steganographic File System(StegFS) 등이 있다.

본 논문에서는 위에 소개한 몇 가지 대표적인 파일시스템을 분석하여, 앞으로 설계 및 개발되어야 할 암호화 파일시스템의 방향을 제시한다.

II. 데이터 암호화

1. 디스크 볼륨 단위의 암호화

디스크 암호화 시스템은 데이터가 디스크로 들어오고 나가고 할 때, 데이터를 암호화하기 위해 디바이스 드라이버 계층을 이용한다[3]. 볼륨 단위의 암호화는 전체 디스크 볼륨을 보호하는 데는 편리하지만, 디렉토리나 파일 같이 세분화된 객체에 대해서는 개별적으로 암호화와 복호화를 할 수가 없다.

2. 파일 단위의 암호화

파일 단위의 암호화는 사용자 데이터를 가장 손쉽게 보호하는 일반적인 방법이다[4],[5]. 암호화는 프로그램 내부와 외부에서 이루어질 수 있으며 각각 다음과 같은 문제점이 있다.

프로그램 내부에서 암호화 기능을 수행하기 위해서는 각각의 프로그램이 자체적으로 암호화 기능을 내장하여야 한다. 예를 들어, 문서 편집기의 경우 파일이 열릴 때 키를 요구하면, 그 이후로 파일의 데이터가 읽히고 쓰여질 때 자동적으로 복호화 및 암호화 된다. 물론, 같은 데이터를 조작하는 모든 프로그램들은 같은 암호화 엔진을 포함해야 한다. 암호화가 자동적으로 이루어지더라도 사용자는 파일이 처음 열렸을 때에는 각 프로그램들에게 키를 제공해야 한다. 암호화 기능을 포함하지 않는 프로그램들이 따로 암호화 프로그램을 사용하지 않는다면 데이터를 안전하게 사용할 수 없다. 게다가 암호화가 하나의 프로그램에 국한되기 보다는 여러 프로그램들 사이에 퍼져 있어, 각각의 프로그램이 다른 프로그램과 안전하고 올바르게 상호 작용하기 위해서 서로 신뢰되어야 한다. 암호화 알고리즘의 변경은 그 알고리즘을 사용하는 모든 프로그램의 수정을 요구하고, 수정 과정 중에 많은 오류를 포함시킬 가능성이 높다. 또한 사용자 수준의 암호화 코드에 대한 많은 복사본은 시스템의 성능 저하를 가져온다.

프로그램 외부에서 암호화를 하는 것은 복호화와 다시 암호화하는 사용자의 명확한 행동이 필요하게 된다. 이러한 문제는 만약 하나의 파일이 단지 한 사람의 사용자에 의해 액세스되는 것이라면 관리 차원의 문제이겠지만 다수의 사람이 공유할 필요가 있다면 훨씬 더 어려운 문제이다. 공유 파일에 대한 명시적 암호화를 하는 것은 암호키의 공유를 필요로 하고, 암호키를 아는 사람이 늘어날수록 암호키가 누출될 가능성이 커진다. 또한, 파일의 복호화는 암호화되지 않은 원래 내용이 디스크나 백업 매체에 남게 될 위험성을 안고 있다.

III. 암호화 파일시스템

위에서 살펴 본 바와 같이, 디스크 볼륨 전체를 암호화하는 것과 파일 단위의 사용자 수준에서의 암호화는 많은 문제점을 가지고 있다. 이에 대한 대안으로 시스템 수준에서 이루어지는 암호화 파일시스템이 제시되었다. 이는 사용자로 하여금 데이터 암호화를 위한 특별한 조작을 필요 없게 하는 사용상의 투명성을 제공하고, 시스템 침입자에 대해서 그들이 원하는 정보를 감춤으로써 데이터에 대한 보안성을 제공해 준다. 대표적인 암호화 파일시스템으로는 다음과 같은 것들이 있다.

1. CFS

CFS는 1993년 AT&T Bell 연구소의 Matt Blaze가 NFS 파일시스템 내에 암호화 기능을 추가한 것으로, 암호화된 파일에 대해 표준 유닉스 파일시스템 인터페이스를 적용함으로써 시스템 수준에서의 보안 저장장치(secure storage)를 제공한다[4]. 사용자는 보호하고자 하는 디렉토리를 암호화 키와 연관시켜 각 디렉토리마다 키를 명시한다. 이러한 디렉토리 내의 파일들은 더 이상의 사용자 개입 없이 명시된 키를 이용하여 암호화 및 복호화 된다. 평문 형태의 데이터는 절대로 디스크 상에 저장되지 않으며, 원격 파일 서버로의 전달도 이루어지지 않는다. 또한 CFS는 NFS와 같은 원격 파일시스템을 포함하여 사용 가능한 파일시스템을 수정하지 않고 이용할

수 있다. CFS의 기본 개념은 시스템 내에서 신뢰되는 부분(trusted components)은 신뢰되지 않은 부분(untrusted components)으로 데이터를 전송하기 전에 무조건 암호화해야 한다는 것이다.

가. 기능

CFS의 가장 큰 목적은 암호화된 파일이 다소 '특별하다'라는 개념 없이, 그리고 하나의 세션에서 같은 키를 여러 번 입력할 필요 없이 통합된 방법(seamless manner)으로 동작하는 보안 파일 서비스를 사용자에게 제공하는 것이다. CFS에서 발생하는 대부분의 동작은 표준 시스템 콜을 통해 이루어지므로 CFS를 통해 발생되는 파일과 그렇지 않은 파일들 간의 두드러진 차이점은 없다.

CFS는 사용자가 제공하는 키를 이용해서 자동적으로 암호화되는 디렉토리 계층 구조에 대한 투명한 유닉스 파일시스템 인터페이스를 제공한다. 사용자가 암호화 키를 디렉토리에 부착(attach)하기 위해 간단한 명령어를 실행하면 보통의 시스템 콜과 툴을 이용하여 그 디렉토리를 사용자가 이용할 수 있지만, 데이터들이 읽혀지고 쓰여질 때 자동적으로 복호화되고 암호화된다.

각각의 디렉토리는 암호화 키의 집합으로 보호된다. 이런 키는 키보드를 통한 사용자의 입력에 의해 제공될 수 있고, 하드웨어가 이용 가능하면 클라이언트 시스템에 연결된 스마트 카드를 통해 제공될 수 있다. 키보드를 통해 입력된 키는 임의의 길이를 가지는 문장인 'passphrase'의 형태를 갖는다.

Passphrase는 CFS의 암호화 루틴에 의해 사용되는 내부 암호화 키 집합을 생성하기 위해 사용되고, 몇 개의 독립된 키를 생성할 수 있도록 충분한 길이를 가져야 한다.

나. 파일 암호

CFS에서는 데이터를 암호화하기 위해 DES 알고리즘을 사용한다. DES 암호화에 사용되는 모드는 여러 가지가 있으나 각각 단점을 가지고 있다. 우선 ECB(Electronic Code Block, 이하 ECB) 모드는 정해진 길이의 블록에 대해서만 암호화가 이루어지므로 같은 내용의 데이터 블록은 항상 같은 암호문으로 암호화 되어 구조적 분석(structural analysis)이 가능해지고, CBC(Cipher-Block Chaining, 이하 CBC) 모드에서는 구조적 분석을 피할 수 있으나 임의의 위치에 있는 블록을 복호화할 때 그 블록을 포함하는 파일의 처음부터 복호화가 이루어져야 하는 단점이 있다. 이는 임의의 위치에 있는 블록을 액세스하기 위해서는 항상 그 블록을 포함하는 파일의 처음 부분부터 액세스가 이루어져야 함을 의미한다. 게다가 액세스 하고자 하는 블록에 대한 액세스 시간이 처음 위치보다 멀어질수록 더 많은 시간을 소비하게 되어, 결과적으로 시스템의 성능을 저하시키며, 임의의 액세스마저 불가능하게 한다.

CFS에서는 구조적 분석의 가능성을 줄이고, 임의의 액세스를 가능하도록 두 가지 방법의 암호화를 제공한다. 하나는 긴 길이의 passphrase를 이용하는 것이고, 하나는 초기벡터(initial vector)를 이용하는 것이다.

1) passphrase를 이용한 암호화

Passphrase는 일상적인 단어들로 이루어진 잘 사용되지 않는 의미 없는 문장으로서, 사용자들이 기억하기는 쉬우나 암호보다 길이가 길기 때문에 공격자들이 공격하기는 어렵다. 예를 들면 'if you have nothing 2 hide you Have nothing too fear!' 같은 문장은 passphrase가 될 수 있다.

디렉토리 부착 시에 제공되는 passphrase는 두 개의 분리된 56bits DES 키로 나뉘어져 파일 블록 암호화에 사용되고, 임의의 액세스도 제공해 줄 수 있다. 파일 블록의 쓰기는 다음과 같은 과정으로 이루어진다.

- 입력된 passphrase를 두 부분으로 나눈다. (각각 56bits DES 키) 첫번째 키를 K1, 두번째 키를 K2라 하자.
- K1으로 512kbytes 정도의 pseodu-random

bit mask를 계산한다.

- 파일 블록의 오프셋을 마스크 길이로 모듈로 연산한 곳의 마스크 블록과 파일 블록을 XOR 한다.
- XOR 된 블록을 두번째 키를 이용하여 ECB 모드로 암호화한다.

파일 블록의 읽기는 쓰기 과정과 반대 과정으로 행해진다. 이런 방법으로 임의의 액세스가 가능해지고, 암호화 과정이 파일의 위치에 기반하기 때문에 파일 내에 같은 내용의 블록이 암호화 되더라도 다른 암호문이 된다.

2) 초기벡터를 이용한 암호화

긴 passphrase를 이용하여 위와 같은 방법을 사용하면 임의의 액세스를 제공하면서 구조적인 분석을 어느 정도 막을 수 있다. 하지만 다른 파일에서 동일 위치의 동일 내용의 블록이라면 역시 같은 내용으로 암호화가 이루어지므로, 구조적 분석의 여지를 남길 수 있다. 이를 막기 위해 파일 블록 암호화에 초기벡터를 이용한다.

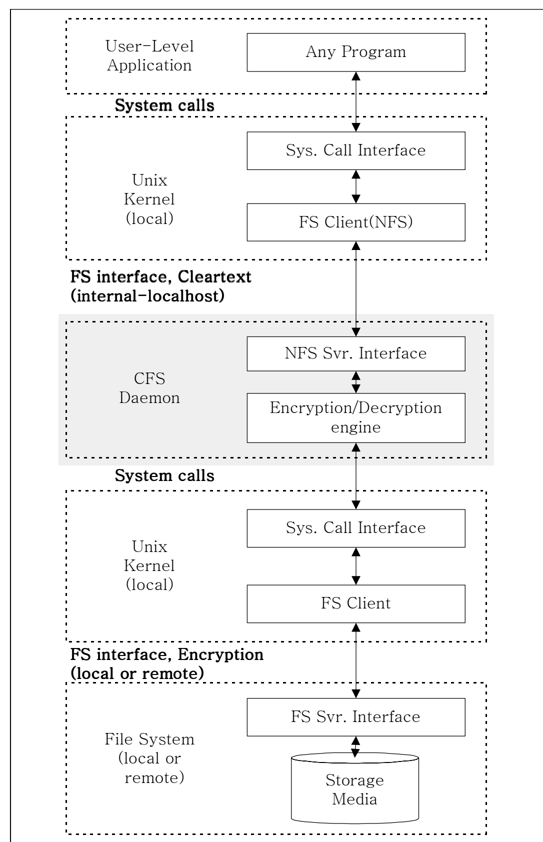
초기벡터는 파일이 존재하는 동안 파일시스템에서 각 파일이 유일하게 유지하고 있는 inode 번호로부터 발생된다. 따라서 각 파일마다 서로 다른 초기벡터를 가질 수 있고 암호화에 이용한다면 긴 passphrase를 사용할 때보다 구조적 분석에 대해 좀더 안정성을 제공할 수가 있다. 하지만 초기벡터의 저장은 발생된 inode에 하는 것이 가장 자연스러우며, inode 필드에 대한 직접적인 액세스를 제공하는 시스템 콜이 존재하지 않아 새로운 파일 속성을 추가할 수 없으므로, 존재하는 어느 한 필드를 대체해야 하는 문제점이 있다. 이 방법은 존재하는 inode 필드가 모두 목적이 있기 때문에 시스템의 불안정성을 유발할 수 있다. 따라서 CFS는 암호화된 디렉토리를 생성할 때 선택할 수 있는 두 가지 모드를 가진다.

- Standard mode: 초기벡터를 사용하지 않는다. 동일한 블록에 대한 구조적 분석이 가능하다.
- High security mode: 초기벡터가 파일 inode의 group id(gid) 필드에 저장된다. 암호화된 디렉토리 내의 각 파일들의 그룹은 파일들이 속한 디렉토리의 그룹과 같아지므로 같은 디렉토리 내에 서로 다른 gid를 가진 파일들은 존재하지 않는다. 하지만 CFS 외부에서는 파일 그룹에 대한 변경이 가능하므로 변경될 경우 제대로 복호화가 되지 않을 위험도 있다.

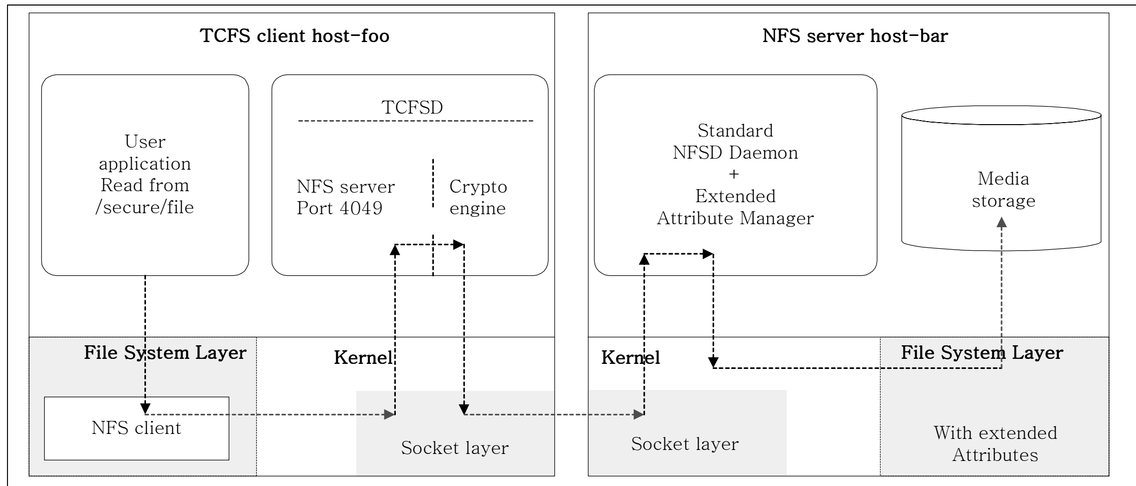
다. 구조

CFS는 전체적으로 NFS 인터페이스를 통해 유닉스 커널과 통신하는 사용자 수준에서 구현되었다. (그림 1)은 CFS의 구조 및 데이터의 흐름을 나타낸다.

CFS 클라이언트 호스트는 로컬 호스트 인터페이스 상에서 CFS 파일시스템의 요청을 번역하는 특별한 NFS 서버인 'cfsd(CFS daemon)'을 동작시킨다. 시스템은 부팅될 때 'cfsd'를 동작시키고, CFS를 시작하기 위해 CFS 디렉토리 상에 로컬호스트 인터페



(그림 1) CFS 구조 및 데이터 이동



(그림 2) 사용자 수준으로 구현된 TCFS 구조

이스의 NFS ‘mount’를 발생시킨다. 그리고 클라이언트가 정상적인 NFS 서버로써 동작하도록 하기 위해 CFS는 표준 NFS와 다른 접속 포트 상에서 동작한다.

NFS 클라이언트는 파일시스템의 성능을 개선하기 위해 캐시를 유지하며, 모든 통신은 클라이언트에 의해 초기화된다. 서버는 각각의 RPC가 도착했을 때 처리할 수 있으며, 처리 후 다음 RPC를 대기한다. 구현의 복잡성은 대부분 클라이언트 쪽에 있다.

‘cfsd’는 CFS 클라이언트 호스트에서 동작하고, CFS 파일시스템의 요청을 번역하는 특별한 NFS 서버이자 RPC 서버로서 구현된 CFS 데몬이다. 부착된 디렉토리 내의 파일시스템 연산은 표준 NFS 클라이언트 인터페이스를 통하여 ‘cfsd’로 보내지는 정규 NFS RPC 호출처럼 보이게 된다.

라. 구조적 문제

‘cfsd’는 전체 파일시스템의 구조에 비해서 매우 간단하다. 특히 ‘cfsd’는 실제 파일의 저장에 대해서 알 필요가 없다. 이런 간단성은 시스템 성능을 떨어뜨리는 요인이 된다. 또한 데이터 저장을 위해 시스템 콜을 사용하는 사용자 수준에서 동작하고, RPC 인터페이스를 통해 클라이언트와 통신하므로 각각의 클라이언트 요청에 대해 부가적인 데이터 복사가 발생하고, 이런 데이터 복사는 시스템에 과부하로 작용하게 된다. 또한 DES 암호화 코드의 실행 역시 시스템 과부하의 주된 요인이 된다.

따라서 시스템의 동작을 커널 수준에서 이루어지도록 하여야 하고, 시스템에 과부하를 적게 줌과 동시에 보안성이 우수한 암호화 코드의 사용이 이루어져야 한다.

2. TCFS

이탈리아 Salero 대학에서 개발된 TCFS는 암호화 서비스와 파일시스템 사이에 더 깊은 통합을 제공함으로써 Matt Blaze의 CFS를 개선한 것이다 [6],[7]. TCFS는 NFS를 기반으로 구현되었으며, 실제 많은 부분이 리눅스에서 발견되는 표준 NFS 구현으로부터 파생되었다. TCFS는 사용자가 그룹 내의 사용자들과 파일들을 안전하게 공유할 수 있도록 해 주는 좋은 특성이 있지만, 커널 패치의 필요 및 최소한의 키 관리 제공 등의 문제점이 있다.

가. 동작 방법

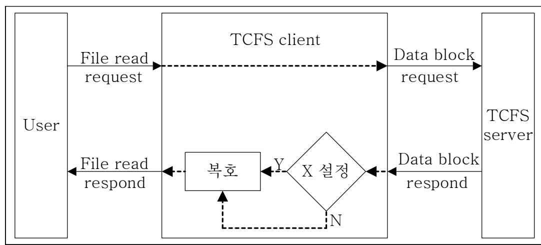
TCFS는 NFS와 암호화 기능의 결합으로, 클라이언트와 서버로 구성되고, 클라이언트와 서버 모두가 리눅스 운영시스템 상에서 동작한다.

(그림 2)는 초기버전(커널 1.2.13)에서 동작하는 TCFS의 구조를 보여준다. 초기 버전에서는 사용자

프로세스로써 구현되어 CFS가 사용자수준의 구조로 인해 가지는 문제점을 안고 있다.

하지만 이후버전(커널 버전 2.0.x)에서는 성능상의 문제로 인해 커널 수준에서 구현되었다.

TCFS 클라이언트는 NFS 클라이언트가 하는 것과 같이 TCFS의 서버에게 파일의 데이터 블록을 요청한다. 만약 확장된 속성 X(extended attribute X)가 그 파일에 설정되어 있다면, TCFS 클라이언트는 읽기를 시도했던 사용자 응용에게 데이터 블록을 전달하기 전에 그 블록을 복호화한다. 만약 확장된 속성 X가 설정되어 있지 않다면 요구될 때 바로 전달된다. 사용자 응용(cat, more 혹은 vi 등)은 액세스할 데이터들이 암호화 되었는지 알 필요가 없다. (그림 3)은 TCFS의 동작 방법을 나타낸다.



(그림 3) TCFS의 동작 방법

나. 키 관리 및 암호화 알고리즘

TCFS는 동적인 암호화 모듈 특성을 가지고, 사용자가 TCFS에 의해 사용될 암호화 엔진을 명시할 수 있도록 해준다. 암호화 엔진은 리눅스 모듈 형태로 주어져야 하고, 매우 간단한 TCFS API를 따라야 한다. 현재 사용 가능한 모듈은 Triple DES, RC5 그리고 Blowfish 알고리즘 모듈이 있다.

모든 파일들은 같은 암호화 알고리즘으로 암호화되고, 사용자 키는 로그인 암호(password)를 기본으로 하며, 사용자 키는 '/etc/tcfspasswd' 파일에 저장되므로 보안성을 감소시킨다.

사용자가 TCFS를 사용할 수 있도록, BKMS (Basic Key Management System)이라는 표준 키 관리 시스템을 제공하고, 사용자에게 사용자 로그인 암호 이외의 어떠한 암호도 기억할 필요가 없도록 한다. BKMS는 사용자의 로그인 암호를 사용하여 암호화된 키 DB 내에 TCFS 암호화 키를 저장한다. 'tcfsadduser' 유틸리티는 사용자에 대한 키 DB 내에 빈 엔트리를 생성한다. 'tcfsgenkey' 유틸리티는 암호화 키를 생성하고, 키로써 사용자 로그인 암호를 이용하여 키를 암호화하고, 암호화된 키를 키 DB 내에 추가한다. 'tcfsputkey' 유틸리티는 사용자 로그인 암호를 요청하고, 키 DB로부터 사용자 암호화 키를 검색하고, 복호화하며 TCFS 클라이언트에게 전달한다. 'tcfsrmkey'는 TCFS에게 전달된 키를 제거한다.

다. XATTRD

XATTRD는 서버 시스템에서 동작하는 데몬으로 TCFS 파일시스템의 파일에 대한 확장된 속성에 관해 클라이언트에게 알리는 역할을 한다. TCFS는 요청된 파일에 대해 이 데몬이 알려주는 속성으로써 파일을 복호화할 것인지를 결정한다. TCFS는 다음과 같은 확장된 속성을 가지고 있다.

- Secure/Unsecure flag: 파일의 암호화 여부를 나타낸다.
- Shared/Unshared flag: 그룹 멤버들 사이에서의 파일 공유 여부를 나타낸다.
- Spure byte flag: 파일 길이가 8bytes의 배수가 아닐 경우 파일을 채우고, 채운 bytes를 부호화하는 3bits이다(암호화 알고리즘에서 입력으로 8bytes 블록을 요구한다).

라. CFS와 TCFS의 차이점

CFS를 사용할 때 사용자는 모든 파일들이 암호화되어 있는 보안 디렉토리를 갖는다. 암호화된 파일을 액세스하기 전에 사용자는 사용자의 디렉토리를 특별한 마운트 디렉토리에 부착할 필요가 있고, 각각의 부착된 디렉토리에 대한 키를 공급할 필요가 있다.

반면, TCFS는 사용자에게 완전히 투명성을 제공

하고, 마치 NFS 처럼 사용되어 질 수 있다. 즉, 암호화된 파일과 복호화된 파일 모두가 같은 방법으로 액세스되고, 사용자는 자신의 파일이 암호화되어 있는지를 알 필요가 없다.

CFS는 암호화 디렉토리에 대해서 암호화가 가능하지만, TCFS는 각각의 파일과 디렉토리에 대해서 암호화가 가능하고, 특별한 플래그인 'X'를 적용함으로써 암호화 및 암호화를 해제할 수 있다.

좀 더 기술적인 측면에서 보면, CFS는 사용자 영역에서 동작하는 반면, TCFS는 커널 영역에서 동작하므로 개선된 성능과 보안성을 제공한다.

3. Cryptfs

Cryptfs는 1998년 콜롬비아 대학의 Erez Zadok에 의해 Stackable Vnode Layer loadable kernel module로써 설계 및 구현된 파일시스템이다[1],[8]. 그리고 사용자에게 클라이언트 파일시스템을 '캡슐화' 함으로써 투명한 암호화 기능을 수행하며 커널 수준에서 동작한다.

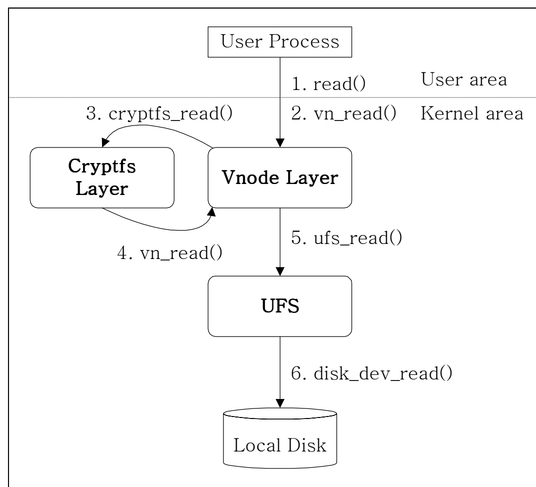
커널 수준에서 동작한다는 것은 암호화 기능이 파일시스템의 일부가 되어 파일을 액세스 하는 모든 응용에 대해 일관된 암호화를 자동적으로 제공하고, 문맥 교환의 횟수를 줄일 수 있으며, 사용자 수준이나 NFS에 기반한 파일시스템 - CFS, TCFS 등 - 보다 효과적인 보안성 및 성능을 제공할 수 있음을 의미한다. 그리고, Cryptfs의 키는 사용자 ID 뿐만 아니라 프로세스 세션 ID에 기반한다는 점과, 커널 메모리의 액세스는 사용자 메모리보다 더 어렵다는 사실로써 좀 더 강한 보안성을 제공한다.

가. Stackable Vnode Interface

Virtual node 혹은 vnode란 파일, 디렉토리, 디바이스 혹은 파일시스템 이름공간에서 보여질 수 있는 엔트리(예, 소켓) 등을 표현하기 위해 유닉스 기반의 운영체제 내에서 사용되는 데이터 구조체이다. Vnode는 구현되어 있는 물리적인 파일시스템의 형태를 보여주지 않으며, vnode 인터페이스는 고수준의 운영체제 모듈이 vnode 상에서 일정한 형태로 연산을 수행할 수 있도록 해준다.

'Vnode Stacking'이란 vnode 개념을 개선시킨 것으로 한 vnode 인터페이스에서 다른 vnode 인터페이스를 호출함으로써 파일시스템의 기능을 모듈화 시키는 기술이다. 'Stacking' 개념이 존재하기 전에는 단일 vnode 인터페이스만이 존재하여, 좀 더 높은 수준의 운영체제 코드가 vnode 인터페이스를 호출하면 vnode 인터페이스는 특정한 파일시스템의 코드를 호출할 수 밖에 없었다. Vnode Stacking의 개념을 사용하면 여러 개의 vnode 인터페이스들이 존재할 수 있어서 서로 서로의 인터페이스를 차례로 호출할 수 있게 된다.

(그림 4)는 간단한 단일 수준의 stackable 암호화 파일시스템의 구조이다. 시스템 콜은 vnode 수준의 콜로 변환되고, Cryptfs 수준의 동일한 콜을 유발시킨다. Cryptfs는 다시 암호화 기능을 수행한 후 일반적인 vnode 연산을 유발시키고, 유발된 vnode 연산은 UFS 처럼 각각의 '저수준' 파일시스템에 특정한 연산을 호출한다.



(그림 4) Vnode Stackable File System의 구조

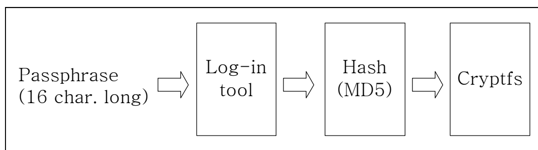
나. 키 관리

Cryptfs에서 사용하고 있는 암호화 키는 사용자

의 UID와 세션 ID의 조합으로 생성되고, 메모리 내의 데이터 구조체에 유지된다. 따라서 공격자는 사용자의 키를 획득하기 위해서는 사용자 계정을 획득해야 함은 물론, 사용자의 passphrase를 받아 들인 프로세스의 세션 ID 까지도 획득해야 한다. 이는 공격자의 공격을 매우 어렵게 만든다.

(그림 5)는 사용자의 passphrase가 키로 형성되어 Cryptfs로 전달되는 과정이다.

새로이 구현된 로그인 툴을 통해 사용자의 passphrase를 입력 받아, MD5 알고리즘을 이용하여 입력 받은 passphrase를 해시시킨다. 그리고 그 결과는 특별한 'ioctl'을 통하여 Cryptfs로 전달된다.



(그림 5) Cryptfs의 키 전달 과정

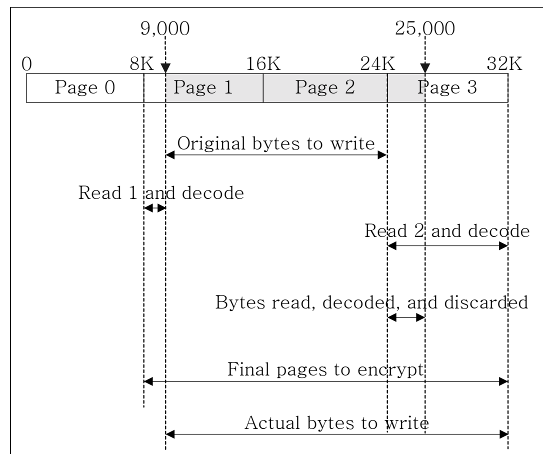
다. 암호화 모드 및 알고리즘

충분히 강한 암호화를 제공하기 위해 CBC 모드로 암호화한다. 하지만, 어느 한 부분을 액세스하기 위해 전체 부분을 복호화하는 것은 시스템의 성능 저하를 가져오므로, 운영체제가 사용하는 블록 크기 - 4kbytes 혹은 8kbytes - 에 대해서 CBC 모드를 적용한다. 암호화 알고리즘으로는 빠르고, 간단한 64-bit 블록 암호인 Blowfish를 사용한다. Blowfish는 자동적인 복호기와 같은 키가 자주 변하지 않는 응용에 적합하고, 448bits 정도의 다양한 길이의 키를 사용할 수 있다. 기본적으로 128bits 길이의 키를 사용한다.

라. 파일의 암호화 및 복호화

Cryptfs에서의 암호화는 기본 페이지 크기인 8kbytes 단위로 이루어진다. 즉, 암호화되는 블록의 크기는 8kbytes이며, 해당 블록 내의 데이터는 Blowfish 알고리즘과 CBC 모드로 암호화된다. 파일 암호화 및 복호화의 설명에 사용될 V는 상위 레벨의 vnode를, V'는 하위 레벨의 vnode를 가리킨다.

파일의 암호화는 파일이 쓰여질 때 이루어지고, (그림 6)은 파일의 임의의 구간에 데이터가 쓰여지는 방법을 나타낸다.



(그림 6) 데이터 쓰기에서의 암호화 과정

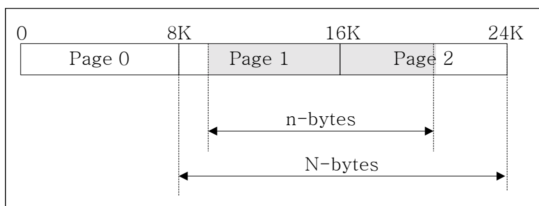
암호화된 파일의 일부를 수정하거나 새로이 저장할 때 다음과 같은 연산이 이루어진다. 여기서는 9,000bytes에서 25,000bytes까지 데이터를 저장한다고 하자.

데이터에 대한 쓰기 연산이 9,000~25,000bytes 범위로 V상에서 호출되고, 확장된 페이지의 범위(8,192~32,767bytes, 3개의 페이지)를 계산한다. V에서의 콜을 V'로 전달하고, 3개의 빈 페이지를 할당한다. V'로부터 8,192~8,999bytes(페이지 1)를 읽고, 복호화 한 후 할당한 첫번째 페이지에 위치시킨다.

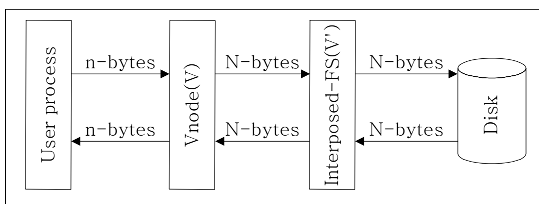
9,000bytes부터 복호화된 데이터는 수정될 데이터로 덮어 쓰여질 것이기 때문에 필요하지 않다. 페이지 2는 수정된 데이터로 모두 덮어 쓰여질 것이므로 무시한다. V'로부터 24,576~32,767bytes(페이지 3)을 읽고, 복호화한 후 할당한 세번째 페이지에 위치시킨다. 이번에는 마지막 쪽의 데이터를 보존해야 하므로 전체 페이지를 복호화해야 한다. 기록되기

위한 데이터를 할당된 3개의 페이지에 기록한다. 이때 처음부분의 오프셋은 9,000-8,192=808이 된다. 이제 할당한 3개의 페이지에는 원하는 데이터가 기록되었지만, 암호화된 상태가 아니므로 암호화한 후 V'상에서 쓰기 연산을 호출하여 디스크에 저장한다.

파일의 복호화는 파일을 읽을 경우 이루어지고, 파일을 쓸 경우보다 훨씬 간단하게 이루어진다. (그림 7), (그림 8)은 파일의 임의의 구간을 읽는 경우를 나타낸다.



(그림 7) Cryptfs에서의 데이터 읽기



(그림 8) 각 층에 전달되는 데이터의 크기

읽을 데이터의 범위가 n-bytes에 속하는 데이터라면 사용자는 V에 n-bytes 데이터를 요구하고, V는 n-bytes가 포함된 전체 페이지 N-bytes를 V'에 요청한다. V'는 N-bytes를 읽고 복호화하여 V에게 전달하고, V는 복호화된 n-bytes를 사용자에게 전달한다.

위와 같이 수정되는 데이터의 페이지에 속한 수정되지 않는 부분의 복호화 및 암호화가 필요한 이유는 페이지 단위로 CBC 모드에 의해 암호화 및 복호화가 이루어지기 때문이다.

4. StegFS

StegFS 역시 데이터를 암호화하여 사용자 데이터를 보호한다는 목적을 가지고 있고, 여기에 steganography(정보 은닉) 특성까지 추가한 파일시스템이다[9].

Steganography란 통신의 존재를 감추는 방법으로 통신하는 기술을 말하며, 주로 메시지를 잡음에 섞어 전송할 때 많이 사용된다[10].

암호화 방식은 공격자가 암호화된 메시지를 찾아내어 원래 내용을 볼 순 없지만, 수정 및 삭제가 가능하다. 하지만, steganography의 목적은 손상되지 않은 메시지 내에 보호하려는 메시지를 감추는 것이므로, 공격자는 감추어진 메시지를 찾아낼 수가 없다.

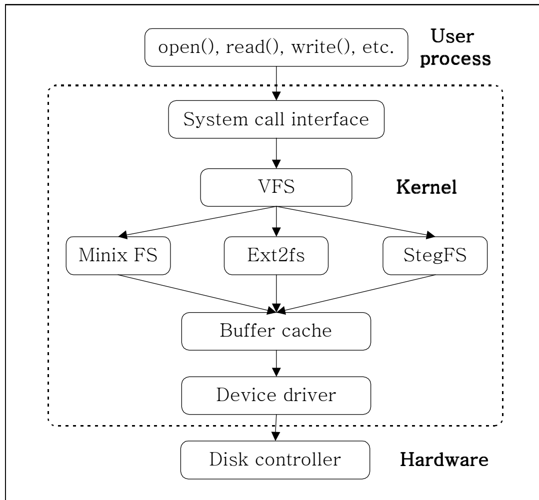
StegFS는 파일 블록들을 임의의 위치에 암호화된 블록들을 기록함으로써 임의의 데이터들 사이에 숨겨질 수 있고, 파일시스템에 기록되는 블록들이 많아질수록 원래 데이터 블록에 대한 overwrite 발생빈도가 높아진다. 따라서, 각 블록을 복구하기 위해 여러 개의 복사본이 필요하고, overwrite 되었을 경우 그렇지 않는 데이터 블록과 구분할 수 있는 방법들이 필요하게 된다.

StegFS semantic은 모든 표준 유닉스 파일시스템 semantic과 매우 유사하므로 하위 디렉토리, inodes, symbolic & hard links 등을 가지고, 시스템에서의 숨겨진 파일 손실 등과 같은 에러 상태가 일반 파일시스템에서의 매우 유사한 상황에서 조치되는 오류 코드를 통해 전달되어, 표준 소프트웨어는 자동적으로 적절한 조치가 가능하다.

가. 구조

StegFS는 VFS 인터페이스와 버퍼 캐시 사이에서 Ext2fs, Minix 등과 같은 수준에 위치된다. (그림 9)는 시스템 내에서의 stegFS의 위치를 보여준다.

StegFS 파티션은 Ext2fs 파티션과 호환 가능하다. 즉, StegFS 드라이버는 Ext2fs 파티션 상에서 동작할 수 있고, 반대로도 동작이 가능하다. 이는 StegFS 드라이버가 시스템에서 완전히 제거된 후에도 StegFS 파티션에 있는 숨겨지지 않은 파일이 표준 Ext2fs 드라이버에 의해 액세스가 가능함을 의미한다.



(그림 9) 시스템 내에서 StegFS의 위치

나. 보안 등급

보안 등급(security level)이란 파일의 은닉 혹은 보호 정도를 나타내는 것으로, 모두 15단계의 보안 등급이 있으며, 각 파일은 그 중 하나의 보안 등급을 가진다. 사용자가 n 등급의 보안 등급을 오픈 하였다면, n 등급의 숨겨진 파일들에 대해서 읽기와 쓰기 등이 가능하다. 즉, 숨겨진 파일에 대한 액세스는 해당 파일에 대한 보안 등급의 오픈에 의해서만 가능하다.

보안 등급이 오픈되지 않았을 경우에 다루어지는 파일은 숨겨지지 않은 파일로써 파일시스템은 Ext2fs로 동작한다. 단, 파일이 삭제될 때 해당 블록이 임의의 데이터로 채워지고, 새로 생성된 파일들의 일부분은 정상적인 경우보다 좀 더 임의의 지점에 위치된다. 이 때는 숨겨지지 않은 파일 블록이 이전의 숨겨진 파일 블록들을 overwrite 할 수 있다.

보안 등급이 오픈되었을 경우 다루어지는 파일은 숨겨진 파일로써 해당 보안 등급의 숨겨진 파일을 볼 수가 있다. 보안 등급이 오픈되어 있는 동안, 볼 수 있는 숨겨진 파일에 의해서 사용된 블록들은 Ext2fs 영역에 의해 overwrite 되지 않는다.

숨겨진 파일들은 보안 등급이 오픈되지 않았을 경우에 숨겨지지 않은 파일들에 의해 overwrite 될 가능성이 많다. 따라서 보안 등급이 오픈되었을 경우 복구를 위해서는 숨겨진 파일에 대한 inode와 데이터 블록들이 파티션을 통해 복사되어야 한다.

다. 키 관리 및 파일 암호

사용자들에게 숨겨진 파일의 융통성을 제공하기 위해 보안 문맥(security context)이라는 것을 추가로 관리한다. 각각의 보안 문맥은 모든 보안 등급의 부분집합에 대한 액세스를 준다. 즉, 보안 문맥 C는 보안 등급 1, 2, …, C 에 대한 액세스를 가능하게 한다. 사용자는 보안 문맥 C에 대한 passphrase만 입력하면 된다.

키가 생성되어 파일 블록이 암호화되는 과정은 다음과 같다.

1) $HP_C = h(PP_C)$
   PP_C는 보안 문맥 C에 해당하는 passphrase이고, h는 보안 해시 함수이다.
2) 블록 테이블의 끝에 15×15 보안 매트릭스 M을 추가한다. M의 각 엔트리는 128bit 워드이다.
3) $M_{C,L} = \{SK_L\}_{HP_C}$
   C가 L에 대한 액서스를 제공한다면, M은 HP_C로 암호화된 SK_L을 포함한다.
4) 다른 모든 매트릭스 엔트리 $M_{C,L}$은 단지 임의의 데이터를 갖는다.
5) 디스크 블록 i에 대한 키($BK_{L,i}$)를 생성한다.
   $BK_{L,i} = SK_L \oplus i$
6) 디스크 블록 i가 보안 등급 L로 숨겨진 파일에 속해 있다면, 디스크 블록 i와 그 블록과 관계된 블록 테이블 엔트리는 위에서 생성된 $BK_{L,i}$로 암호화된다.

각각의 블록은 CBC 모드로 암호화되고, 알고리즘으로는 Serpent 혹은 RC6 알고리즘이 사용된다.

라. 블록 테이블

블록 테이블(block table)은 Ext2fs에서의 블록 할당 비트맵과 동일한 것으로, overwrite 된 블록을 감지할 수 있도록 각 블록에 대해 암호화된 체크섬을 저장하는 것과 inode를 포함하는 블록에 대한

inode 번호를 저장하는 것이 목적이다.

각 블록 테이블 엔트리는 해당되는 디스크 블록의 데이터와 같은 키로 암호화된 128bits 길이를 가지고, 사용되지 않는 블록들에 해당하는 블록 테이블 엔트리들은 임의의 데이터로 채워진다.

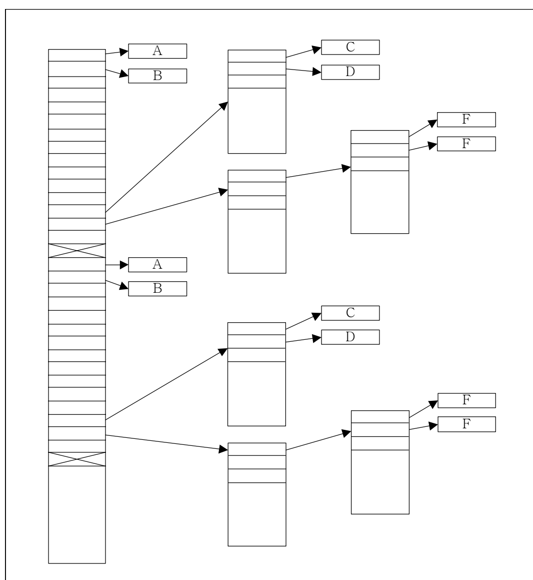
마. 숨겨진 StegFS Inode 구조

StegFS의 숨겨진 inode 구조는 Ext2fs의 inode 구조와 동일하지만, inode와 데이터 블록으로 구성된 복사본의 개수를 포함하여 해당하는 모든 복사본에 대한 참조를 가능하게 한다. (그림 10)은 중복되어 숨겨지는 inode의 구조를 보여준다.

각각의 숨겨진 inode에 대한 복사본의 개수는 디스크의 저장 공간에 중복성을 제공하고, 시스템의 성능에 많은 영향을 미친다.

숨겨지지 않은 파일과 숨겨진 파일과의 구분은 inode 번호의 처음 2bits으로 가능하고, 숨겨진 파일의 경우 inode 번호 내에 4bits의 보안 등급을 포함한다.

StegFS에서의 숨겨진 파일의 복구는 복호화된 블록 테이블에서 inode를 찾아 해당 블록의 체크섬으로 overwrite 되지 않은 블록을 찾아나간다.



(그림 10) 숨겨진 StegFS inode 구조

바. 파일의 읽기와 쓰기

파일 블록의 암호화 및 복호화되는 과정은 키관리 부분에서 설명하였으므로, 이 단원에서는 복사본을 가지고 있는 파일 블록들에 대한 읽기 및 쓰기에 대해서 설명한다.

파일을 읽을 경우 대개는 해당하는 블록의 첫번째 복사본만이 요구되는 경우가 대부분이다. 이 블록의 체크섬이 올바르면 해당 블록을 읽을 수 있도록 복호화한다.

반면 파일을 수정하거나 생성하기 위해 데이터 블록을 쓸 경우라면 과정이 좀 복잡하다. 수정의 경우에 한해서 우선 해당되는 위치의 블록을 읽어서 복호화한다. 복호화된 블록을 변경한 후 해당 초기벡터를 수정하고, 블록을 암호화하여 디스크에 저장한다. 블록 할당 테이블 엔트리는 새로운 초기벡터와 체크섬으로 갱신된다. 이 블록의 복사본 리스트를 통해 그동안 overwrite 되었는지 혹은 아직 유효한지 검사한다. 아직 유효하다면 수정한 데이터를 기록할 곳의 블록과 같은 위치의 블록이므로 새로운 데이터를 암호화하여 디스크에 저장한다. Overwrite 되었다면, 그 위치의 블록은 전혀 새로운 데이터 블록이다. 따라서, 숨겨지지 않은 파일 혹은 Ext2fs에 의해 사용중일 경우에는 새로운 블록을 할당 받아 데이터를 암호화하여 저장하고, inode와 블록 테이블도 수정한다. 하지만 사용중인 경우가 아니라면 그 자리에 암호화하여 저장한다.

사. Trade-off

StegFS는 파일의 암호화 기능에 정보 은닉 기능까지 구현한 파일시스템이다. 위에서 살펴본 바와 같이 암호화 파일시스템의 성능은 암호화 루틴의 수행으로 인해 일반 파일시스템의 성능보다 떨어짐을 보았다. StegFS는 암호화 기능에 정보 은닉 기능까지 추가되었으며, 정보 은닉 기능은 시스템 성능에 상당한 비용을 요구한다.

따라서 StegFS의 성능은 일반 파일시스템에서 뿐만 아니라 다른 암호화 파일시스템 보다도 훨씬 떨

어진다. 정보 은닉의 필요성이 굳이 요구되지 않는다면 이전에 설명된 것과 같은 암호화 기능만을 가진 파일시스템의 사용이 성능 면에서 유리할 수도 있다.

Ⅳ. 암호화 파일시스템의 활용

암호화 파일시스템은 사용자에게 암호화를 사용하기 위한 조작을 최소로 하고, 시스템 전체에 대해서 데이터를 암호화 하므로 보안성을 강화시킨다. 다음은 암호화 파일시스템으로 인해 보안성이 강화되는 구체적인 예이다[5].

첫째, 원래 암호화 파일시스템을 지원하지 않는 프로그램을 암호화된 파일시스템 위에서 동작하도록 하여 강한 암호화 기능을 추가할 수 있다. 이는 프로그램의 작성을 좀 더 수월하게 해 주는 장점도 준다.

둘째, CD-ROM과 같은 매체 기반의 백업이 더욱 안전해 질 수 있다. 백업 데이터 자체가 암호화되어 있으므로 백업 매체에 대한 분실 및 도난에 대해서도 데이터를 안전하게 보호할 수 있다.

셋째, 시스템 자체의 보안을 개선시킬 수 있다. Tripwire와 같은 프로그램은 키 파일의 암호 서명을 기록할 수 있는데 이 프로그램은 고전적으로 침입자들이 참조 정보를 수정하지 못하게 하기 위해 읽기 전용 매체를 필요로 한다. 이는 여전히 내부인이 디스크의 중요한 정보를 바꿀 수 있다는 점을 충분히 생각할 수 있으며, 이런 중요한 정보를 암호화된 형태로 저장함으로써 변질된 내용의 디스크 생성을 어렵게 한다.

넷째, 원래 암호화를 지원하지 않는 전체 시스템에 강력한 암호화 도구를 제공할 수 있다. 암호화된 파일시스템은 NFS나 SMB를 통해서 익스포트(export)될 수 있어야 한다. 패킷 스니핑의 위험이 있다는 것은 여전히 문제로 남지만 디스크는 보호될 수 있다.

Ⅴ. 결론 및 향후 연구방향

중요한 데이터를 보호하는 방법 중의 하나인 암호화 파일시스템에 대해서 알아 보았다. 크게 사용자 수준에서 동작하는 파일시스템과 커널수준에서 동작하는 파일시스템으로 나눌 수가 있다.

사용자 수준에서 동작하는 암호화 파일시스템은 CFS와 StegFS 등이 있으며, 구현 및 개발이 쉽다는 장점이 있다. 하지만 사용자 프로세스로 동작하기 때문에 사용자 메모리 영역의 이용과 커널과의 문맥 교환의 횟수 증가로 보안성과 성능이 떨어지는 단점이 있다.

커널 수준에서 동작하는 암호화 파일시스템으로는 TCFS와 Cryptfs 등이 있다. 이 파일시스템은 사용자 영역의 사용이 없으므로 문맥 교환의 횟수가 적어 성능이 개선될 수 있고, 커널 영역은 사용자 영역보다 접근이 어렵다는 특성 등으로 사용자 영역보다 더 강한 보안성을 제공한다.

파일을 암호화하여 저장하는 방법은 어떤 데이터 보호장치보다 더 나은 보안성을 제공하지만 완벽한 것은 아니다. 위에서 살펴본 몇 가지 시스템의 구현에서 보면 대부분이 디스크 캐시에 복호화된 데이터를 그대로 가지고 있지만, 이는 캐시가 안고 있는 문제이다. 만약 이 페이지가 스왑 디스크에 쓰여지는 것이라면 중요한 문제가 된다.

따라서 앞으로의 연구는 위에 언급한 문제를 해결하는 방향으로 이루어져야 한다. 또한 사용자에 대한 인증을 강화하는 부가적인 장치 즉, 스마트 카드 같은 장치의 사용으로 시스템의 접근 통제를 강화하여야 한다.

참 고 문 헌

[1] E. Zadok *et al.*, "Cryptfs: A Stackable Vnode Level Encryption File System," *Technical Report CUCS-021-98. Computer Science Department*, Columbia University, 28 July 1998, Available at http://www.cs.columbia.edu/~library/

[2] Encrypting Your Disks with Linux, http://drt.ailis.de/crypto/linux-disk.html

[3] James Hughes *et al.*, "A Universal Access, Smart-card-Based, Secure File System," David Corcoran, Purdue University, 10 Feb. 2000.

[4] M. Blaze, "A Cryptographic File System for Unix," *Proceedings of the first ACM Conference on*

Computer and Communications Security(Fairfax, VA). ACM, Nov. 1993.

[5] 리눅스 저널: 암호화 파일시스템, 프로그램세계, 7월호, 1998, pp. 226 - 231.

[6] G. Cattaneo *et al.*, "Design and Implementation of a Transparent Cryptographic File System for Unix," *Unpublished Technical Report. Dip. Informatica ed Appl*, Universita di Salerno, 8 July 1997. Available via ftp in ftp://edu-gw.dia.unisa.it/pub/ tcfs/docs/tcfs.ps.gz.

[7] Transparent Cryptographic File System, http://tcfs.dia.unisa.it/

[8] Erez Zadok, "Stackable File Systems as a Security Tool," *Technical Report CUCS-036-99. Computer Science Department*, Columbia University, Dec. 1999, Available at http://www.cs.Columbia.edu/~ezk/research/fist/

[9] D. Andrew *et al.*, "StegFS: A Steganographic File System for Linux," IH'99, LNCS 1768, 2000, pp. 463 – 477.

[10] Steganography, http://www.jjtc.com/stegdoc/sec201.html