平成17年度 学士論文

仮想計算機を用いてOSを介さずに行う安全なファイルアクセス制御

東京工業大学 理学部 情報科学科

学籍番号 02-1474-1

滝澤　裕二

指導教員

千葉 滋 助教授

提出日 2月6日

概要

ネットワークの普及に伴い、電子メールや電子商取引など、日常の作業の多くをコンピュータで行うようになった現在、扱うデータの種類や情報量は大きく、データの持つ重要性が高まっている。しかし、ネットワークを利用するということは、いつ、いかなる時にもウィルスの侵入、クラッカーからの攻撃を受ける可能性があるということである。

このような攻撃を防ぐために、これまで OS で行う様々なアクセス制御機構が提案されてきた。しかし、これらのアクセス制御機構は OS 自体に脆弱性が存在する場合には、正常に機能しなくなる可能性がある。例えば、クラッカーがコンピュータに侵入した場合、クラッカーの取得した権限がスーパーユーザではない場合には、システムファイルへのアクセスや、アプリケーションのインストールを行うことはできない。しかし、アクセス制御を行う OS 自体に脆弱性がある場合には、クラッカーは OS の脆弱性を攻撃して管理者権限を奪い、すべてのファイルへのアクセスを許してしまう。

本研究では、仮想計算機を用いることにより OS に依存しないファイルアクセス制御を可能にする SecureAccess を提案する。SecureAccess は、２つの仮想計算機を利用して、それぞれでユーザが通常の作業を行う OS（作業 OS）とアクセス制御を行う OS（認証 OS）を動かす。作業 OS でファイルアクセスを行うときには、作業 OS が認証 OS と通信をして認証を行い、認証 OS が作業 OS の画面に表示したダイアログにパスワードを入力させる。パスワード入力は認証システムに直接つながれたキーボードをタイプすることでのみ可能であり、作業 OS を乗っ取っても入力することはできない。また、このダイアログに入力したパスワードは認証 OS に直接渡されるため、作業 OS では知ることができない。ダイアログには認証 OS で登録した文字列を表示することで、そのダイアログが認証 OS から出されたものであることを確認する。そのため、作業 OS に対する攻撃、例えばキーロガーの設置などでパスワードが漏洩することはない。

SecureAccess ではファイルアクセスに関して２つのポリシーを記述できる。１つは空間に関する記述であり、もうひとつは時間に関する記述である。空間に関するポリシーでは認証によってユーザーにアクセス許可されるファイルの範囲を限定することが可能である。時間に関するポリシー

はアクセスの許可される期間を認証時に指定することができる。空間的制限では、ファイルの範囲だけでなくそれぞれのファイル・ディレクトリに対して読み込み専用、追記のみ等のパーミッションが設定可能である。

本研究では、仮想計算機モニタである Xen を用いて、SecureAccess のプロトタイプの実装を行った。作業 OS、認証 OS として Fedora Core4 を使用した。ファイルアクセス命令の転送には、ptrace を用いてシステムコールに割り込みをかけ、ファイルアクセス命令を認証 OS に転送する方法を用いた。実験では、作業 OS と分離された認証 OS においてファイルアクセス制御が行えることを確認した。

謝辞

本研究を進めるにあたり、研究とは何か、プレゼンテーションの仕方や論文の書き方について助言をしていただいた指導教官の千葉先生に心より感謝いたします。

東京工業大学の光来先生にはシステムの設計・実装、プログラミング等研究全般に渡り指導していただきました。心より感謝いたします。

東京工業大学の柳澤佳里氏には、研究活動において多くの助言をいただきました。心より感謝致します。

最後に、ともに研究活動をおこなった研究室の皆様に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

ネットワーク利用者の増加に伴い、コンピュータが外部からの攻撃を受ける危険性も高まっている。いつ、いかなる時にウィルスの侵入、ハッカーからの攻撃を受ける可能性がある。

しかし、既存のシステムのセキュリティ機構では、十分な安全性を確保できていない。通常システムでは、バックアップ等の必要な管理タスクを行うために、スーパーユーザーはセキュリティ上なんでもできるようになっている。これは、セキュリティを犠牲とした一種の妥協であり、ハッカーの狙うものである。たとえば、アプリケーションにセキュリティホールが発見された場合、管理者はパッチが提供され、それが適用されるまでの間は、サービスを停止させるか、攻撃を受けないことを期待しながら、脆弱性を持った状態で運用するしかない。どちらにしても、早くパッチを適用することが必要である。しかし、パッチの適用には、それが稼動中の他のアプリケーション、サービスに及ぼす影響も綿密に検証しなければならない。管理者は検証のために十分な時間をとろうとすれば、パッチの適用は遅れ、パッチの適用を急ぐと検証が疎かになってしまう。これは管理者が膨大な権限を持っていることが原因であり、ハッカーの侵入を許し、管理者権限を奪われてしまった場合の被害が大きくなる。

既存技術であるセキュアOSでは、従来の管理者権限に依存した管理や単純なファイルオーナーによる権限制御を、より厳格なアクセス制御機構によって補完している。例えば、アクセス制御を強化して、絶対的な管理者権限をなくし、ハッカーの侵入を無効化する等である。セキュアOSは、強制アクセス制御によってセキュリティ管理者以外はセキュリティ設定が不可能である、ユーザー別のアクセス制御によって管理者権限を分割し特権の集中を防ぐ、プロセスごとのアクセス制御がが可能、といった機能がある。しかし、セキュアOSでもOS自体の脆弱性を攻撃された場合にはハッカーからの攻撃を防ぐことはできない。

これらは既存システムのセキュリティ機構の大部分がOSによって実現されているために起こる問題である。これらの問題を解決するために、本研究ではファイルアクセス制御機構の作業OSから分離したシステムであるSecureAccessを提案する。SecureAccessでは仮想計算機を用いて一台のマシン上で2つのOSを動作させる。このうち1つのOS（作業OS）で

ユーザーは作業を行うことになる。もう 1 つの OS（認証 OS）はファイルアクセス制御のために使用される。ファイルアクセスに対して、アクセス制御を行う OS は認証を求める。認証には、認証 OS が作業 OS の画面上にダイアログを表示して、パスワードを入力させる。パスワードは認証 OS に直接渡される。そのため、作業 OS に対する攻撃ではパスワードは漏洩しない、また作業 OS からの入力も不可能である。ダイアログには認証 OS に登録した文字列を表示させて、認証 OS から出されたものであることを確認する。

SecureAccess では 2 つのポリシーを記述できる。1 つは空間に関する記述であり、もうひとつは時間に関するである。空間に関するポリシーでは認証によってユーザーにアクセス許可されるファイルの範囲を限定することが可能である。時間に関するポリシーはアクセスの許可される期間を認証時に指定することができる。空間的制限では、ファイルの範囲だけでなくそれぞれのファイル・ディレクトリに対して読み込み専用、追記のみ等のパーミッションが設定可能である。

以下、2 章では既存のセキュリティ機構とその問題点について述べ、3 章では既存技術の問題を解決するために本研究で提案するシステムの特徴を述べる。4 章では、本システムの実装と使用例について、5 章では、通常のシステムと本システムを使用した場合の実行速度の比較を行い、6 章でまとめを述べる。

第2章　問題点と関連技術

2.1　従来システムの問題点

通常のシステムでは、スーパーユーザー（システム管理者権限、ルート権限）は必要な管理タスク（バックアップ等）を行うために、セキュリティ上なんでもできるようになっている。この何でもできるという一種の妥協がセキュリティホールに繋がり、攻撃者が狙うものである。通常のシステムでは一度、システム管理者権限を奪取されてしまうと、ログファイルの書き換えて侵入の痕跡を消去したり、システムの設定の変更、バックドアやキーロガーの設置などの行為を行われ、対応が困難である。この章では、セキュリティに関連する既存技術として、Storage-Based Intrusion Detection [2]、Self-Securing Storage[1]、セキュア OS、ファイルサーバーについて述べる。

2.2　Storage-Based Intrusion Detection

2.2.1　侵入検知システム

いままで多くの侵入検知システムが開発されてきた。これらの侵入検知システムは、大きくネットワーク型とホスト型に分類される。ネットワーク型はファイアウォールや sniffer 等に組み込まれ、ネットワーク上を流れるデータを検査し、攻撃と思われるシグネチャやトラフィックを検出する方法であり、、ホスト型は OS に組み込まれ、ログなどのローカルな情報を検査する方法である。

2.2.2　ホスト型侵入検知システム

ホスト型の侵入検知システムは公開サーバなど、監視対象となるホスト（コンピュータ）自身に導入する。多くのホスト型侵入検知システムではファイルの状態の監視が可能である。ファイルのサイズやパーミッションといったものが当てはまる。

例えば Web サーバであれば、監視したい項目として考えられるのがコンテンツの書き換え（ファイルの改ざん）である。ソフトウェアによって

違いはあるが、検知方法のひとつとして整合性のチェックによる監視がある。ファイルサイズの HASH 値をデータベースに保存して定期的にチェックすることで、内容の変化を検知する。変更したはずのないファイルが整合性チェックに引っかかったら、不正アクセスを許したと考えられる。

2.2.3　ネットワーク型侵入検知システム

ネットワーク型の侵入検知システムは、リアルタイムでネットワーク上を流れるデータを監視し、不審な通信を検知する。

例えば、攻撃者は攻撃の準備段階として、ポートスキャン[1]をする。ネットワーク型侵入検知システムはこれを検知できる。

ネットワーク型の検知として以下の 2 つを挙げておく。

- シグネチャと呼ばれる情報を持つもの。シグネチャとは、攻撃と推測されるパケットの特徴をパターン化したものである。シグネチャを持つものは、ネットワーク上を流れるパケットとシグネチャを比較する。シグネチャと一致するものが見つかれば、攻撃を受けている可能性があると判断できる。シグネチャとパケットを比較するやりかたは、新たな攻撃手法に対応するために、攻撃手法が発見されるごとにシグネチャを更新する必要がある。

- シグネチャを利用しないもの。シグネチャを利用しないものは、統計情報から分析を行い、以上状態を検知する。シグネチャを利用しないので更新の必要がない。

2.2.4　ストレージ型侵入検知システム

ホスト型、ネットワーク型とは異なる方法として、ストレージ型の侵入検知システムがある。ストレージ型侵入検知システムは、ストレージデバイスにリクエストの監視や、デバイスを使用するクライアントの振る舞いを監視させるシステムである。ストレージ型侵入検知システムはストレージデバイスのファームウェア[2]ファームウェアとは、ハードウェアの基本的な制御を行うために機器に組み込まれたソフトウェアのこと。通常ファームウェアは機器に内臓された ROM やフラッシュメモリなどに記憶されている。} として組み込まれる。

[1] ポートスキャンとは、TCP/UDP のポートがオープンしているかどうか調査する行為のことをいう。広義では、ポートに接続してそのポートの素性を調べたり、サーバの情報を集めたりする行為を含むこともある。ポートスキャンでわかることは、どういうサービスが稼動しているか、外部からアクセス可能か、といった情報を知ることができる。

[2] {

ホスト型、ネットワーク型に比べて有利な点

ストレージ型侵入検知システムでは、ハードウェアへの物理的なアクセスはできない侵入者に対して効果がある。つまり、侵入者がホストシステム上の任意のソフトウェアを修正、実行、終了、インストールできるが、ストレージデバイスと、管理者のコンソールは操作できない状況である。このような脅威に対して、ホスト型のシステム、ネットワーク型のシステムでは侵入者にシステムが攻撃されていることを検知することは難しい。何故ならば、ホスト型侵入見地システムでは侵入者により検知システムが無効化されてしまう、ネットワーク型は侵入の試みや侵入前の攻撃を調べることが主な仕事であるためである。一方、ストレージ型な方法は以下の点で有利である。

- ホスト OS に依存していない。

  OS カーネルを含むクライアント上で動作するソフトウェアでは無効化できない。

- 侵入者が行うほとんどの操作はストレージデバイスを使う命令である。

  例えば、ログファイルの修正、バックドアの設置などの行為に対し、ストレージデバイスによる監視が有効である。

detection rule

ストレージ型侵入検知システムでは、進入の証拠と思われるリクエストをキャッチしたら、管理者に警告をだす。監視できる侵入者の行動の分類は以下のようになっている。

- データとメタデータの修正

  普段書き換えることのないファイル（例えばシステムの実行ファイル、スクリプト、設定ファイル、システムのヘッダーファイル、ライブラリなど）への修正は侵入のサインと考えられる。

  ホスト型侵入検知システムではデータベースとの比較によってこの変更を検知する仕組みがあるが、そのチェックが行われる保障がない、一度修正してチェックされる前に元に戻すなどの短期間の変更は検知できない等の欠点がある。ストレージ型は、データベースなどとの比較ではないのでこのような問題はない。

- アップデートのパターンファイルの中には、一定のパターンに従って更新されるものがある。そのようなファイルのパターンから外れた更新は侵入者によるものと考えられる。例えば、

  - ログファイルへの変更
    ログファイルへの変更は普通 append しか行わない。それ以外の変更は侵入者による変更だと思われる。
  - タイムスタンプが前の時間に変更される場合
  - DoS と思われるもの
    大量のデータの書き込みがあった場合は侵入者による DoS 攻撃かもしれない。しかしこれは、正規ユーザーによる悪意のない行動（例えば、サイズの大きいメディアファイルのダウンロード等）と区別できない。

- ファイルの中身の一貫性

  ファイルの中には、特定のフォーマットを持ったものがある。例えば、UNIX の”/etc/passwd”は次のような規則を持っている。

  - passwd の中身は改行で区切られたレコードの集まりである。
  - それぞれのレコードはセミコロンで区切られた 7 つのフィールドからなる。

  ストレージ型の侵入検知システムでは、ファイルの中身がフォーマット通り正しく書かれているかをチェックする。”/etc/passwd”の例ではパスワードフィールドは空白でないか、shell のフィールドは正しく設定されているか等をチェックする。

- 疑わしいコンテンツ

  いわゆるウィルスチェックであり、シグネチャとファイルの比較を行う。

ストレージ型侵入検知システムの限界

ストレージデバイスで監視を行うという特性ゆえに、明らかな限界と欠点がある。まず、ストレージデバイスに関係しない振る舞いは監視できないこと。それとパフォーマンスの低下である。監視する項目や規則が増えるほどパフォーマンスが低下する。

侵入と思われるリクエストを検知した場合

侵入と思われる行動を検知したあとの対応の仕方には以下のような方法がある。

- 管理者に警告を出す
- 警告後、検知に引っかかった処理を遅らせる。
  例えば、管理者が警告を確認するまで遅らせる。
- ファイルバージョニングを開始する

2.3　Self-Securing Storage

Self-Securing Storage は攻撃者によるデータの永久的な消去、検知不能なファイルアクセスを防ぐストレージシステムである。これらの攻撃を防ぐために、Self-Securing Storage ではストレージデバイスでリクエストの監視や、ファイルのバージョニングを行なっている。Self-Securing Storage は侵入を検知するシステムではなく、侵入後の管理者の診断と回復の作業をアシストするシステムである。Self-Securing Storage の機能であるファイルのバージョニング、リクエストの監視ログによって、管理者の侵入検知やシステムの回復といったタスクを手助けする。

2.3.1　診断と回復

攻撃者がコンピュータへ侵入した後、管理者には 2 つの仕事がある。システムの診断と回復である。システムの診断は以下の 3 つの局面に分けられる。

- 侵入の検知：システムが攻撃されたかどうか調べる。
- 攻撃された弱点の発見：攻撃された弱点がわからないと、システムの回復をしても再び侵入を許してしまう。
- 被害の特定：どのファイルが侵入者によりアクセス又は、修正されたか。バックドアなどが仕掛けられていないか等。

一般に診断の作業は難しい。侵入者はログファイルやタイムスタンプの書き換えを行うからである。

一般に診断ができないので、回復には OS の再インストール、バックアップからのデータの復元という作業をすることになる。しかし、この方法では時間が掛かる、最後にバックアップされてからのファイルの変更は復元不可能である。Self-Securing Storage を使うとこの問題が解決される。

2.3.2　Self-Securing Storage の機能

Self-Securing Storage の主な機能は、ファイルのバージョニングと、ファイルアクセス（リクエスト）の監視である。ファイルのバージョニングはファイルに修正があるごとに新しいファイルのバージョンを作成する。バージョン化されたファイルは管理者によって定められた detection window と呼ばれる期間の間保持され、detection window を過ぎると破棄される。バージョン化されたファイルはファイルへのアクセスログと一緒に history pool と呼ばれる領域に保存される。detection window を長く設定するほど、バージョン化されたファイルのための領域（history pool）が大きくなる。

Self-Securing Storage はファイルのバージョニング及び、ファイルアクセスの監視を行い、侵入の痕跡が残ることを保証することで、前述の診断と回復に対する問題を解決している。

- 診断に対する問題の解決
  侵入者がログファイルを修正した場合であっても、history pool にある侵入者により修正される直前の状態のログファイルを復元し調べることで、侵入後に生成されたシステムのメッセージをみることができるので、侵入の検知、弱点の特定が容易になる。Self-Securing Storage アクセス（リクエスト）ログを調べれば、侵入者が直接アクセスしたファイルを特定することができる。

- 回復に対する問題の解決
  Self-Securing Storage により診断の作業ができるので、システムの回復も容易になる。診断の作業で侵入者により直接アクセスされたファイルの特定ができるため、修正されたファイルを個々に復元することで、OS の再インストールが不要になる。また、システムファイルを他のコンピュータに容易しておいたデータベースと照合し、修正があったものをデータベースから複製するという方法があるが、これはスタティックなファイルにしか適用できない。一方、Self-Securing Storage では任意のファイルに対して復元が可能である。また、バックアップからのファイルのコピーするのではなく、history pool からのファイルの復元により、ファイルを侵入者により修正される直前の状態に復元することが可能である。

これらの解決は、Self-Securing Storage の「侵入者がファイルのバージョニングシステムを回避できない」という性質に基づいており、これは Self-Securing Storage の一番の利点である。

2.3.3　Self-Securing Storage の問題点

ここでは、Self-Securing Storage の問題点２つについて説明をする。まず適切な detection window の設定が難しいことである。detection window を長く設定すると、それだけ記憶装置の容量が必要になる。detection window を短く設定すると、必要な容量は小さくなるがすぐに古いバージョンが破棄されてしまう。そのため、コンピュータに侵入されてから、detection window の期間内に侵入を発見できなかった場合は、history pool に保持されているファイルは、侵入者に修正された後のバージョンのファイルしか保持されていないことになり、侵入者に修正されたファイルの復元ができなくなってしまう。

残りの１つは、侵入者により history pool がオーバーフローさせられる危険があることである。この問題には良い解決方法が見つかっていない。history pool のオーバーフローは単純に history pool の領域を増やすだけでは解決できない。以下に３つの欠点のある解決方法を示す。

1. history pool が一杯になったら、一番昔に作成されたファイルのバージョンを削除する。

   欠点：侵入者により history pool の中身をすべて入れ替えられてしまうと、侵入検知に必要なアクセスログなどの情報が読めなくなり、診断・回復ができなくなる。

2. history pool が一杯になったら、ファイルのバージョニングを止める。

   欠点：古いバージョンのファイルは復元できるが、history pool が一杯になったあとに行われたファイルアクセスについては何の情報も得られなくなり、診断ができなくなる。例えば、侵入者が history pool が一杯になるまで悪意のない行為をしてから、バックドアを仕掛けた場合には、history pool からはバックドアを仕掛けられたという情報は得られない。

3. history pool が一杯になったら、history pool に空きができるまで Self-Securing Storage に対する新しいファイルのバージョンを作成するようなすべてのリクエストを拒否する。

   欠点：結果的に全てのユーザーに対して DoS をすることになる。

Self-Securing Storage では history pool が一杯になったときどうするかではなく、history pool が一杯にならないようにするようなポリシーにしている。そのために、上記の方法を組み合わせた方法が取られている。

- 方法：

侵入者がやりそうな悪用を検知して、マシンから Self-Securing Storage へのアクセスを制御する。history pool をオーバーフローさせようとしていると思われる振る舞いに対して、選択的に帯域幅を減らしたり、待ち時間を増やすことで、history pool がオーバーフローするのを防ぐ。

- 利点：

  オーバーフローと思われる行為に対してのみを帯域幅や待ち時間の制御をするので、Self-Securing Storage がオーバーフロー攻撃を受けている最中でもユーザーが作業を続けることができる。

- 欠点：

  悪意のある行為とそうでない行為の区別ができない。例えば、ユーザーがサイズの大きいメディアファイルをダウンロードするなどの行為が、オーバーフロー攻撃とみなされる可能性がある。

2.3.4　Self-Securing Storage Server の実装

Self-Securing Storage Server（S4）は Self-Securing Storage の機能を持ったバージョニングシステムのプロトタイプである。S4 は NFS を拡張して作られている。この節では S4 について述べる。

S4 はネットワークで接続されたオブジェクトベースのストレージデバイスになっている。

表 2.1 は、S4 の提供する RPC コマンドである。read-only コマンド（read,getarrt,getacl,plist,pmount）はオプションとして時間のパラメータを受け取る。S4 が時間のパラメータを受け取ると、S4 は受け取った時間に最も近いバージョンのファイルに対して、そのコマンドを実行する。

S4 の実際の処理の大まかな流れは以下のようになる。

1. S4 を利用するクライアントがファイルアクセスのリクエストを出す。
2. リクエストを S4 の提供する RPC コマンドに変換する。
3. 変換したリクエストに従って、S4 の RPC を呼び出す。
4. S4 で実際に処理。
5. クライアントに結果が返ってくる。

図 2.1,2.2 は S4 システムの 2 種類の実装の概観である。図 2.1 は前述の処理の 2 番「リクエストを S4 の RPC コマンドに変換する」をクライアント側で行い、図 2.2 はサーバー側で行うようにな仕様になっている。

RPC TYPE	Allows Time-based Time-based Access	Description
Create	no	Create an object
Delete	no	Delte an object
Read	yes	Read data from an object
Write	no	Write data to an object
Append	no	Append data to the end of an object
Truncate	no	Truncate an object to a specified length
GetAttr	yes	Get the attributes of an object (S4-specific and opaque)
SetAttr	no	Set the opaque attributes of an object
GetACLByUser	yes	Get an ACL entry for an object given a specific UserID
GetACLByIndex	yes	Get an ACL entry for an object by its index in the object's ACL table
SetACL	no	Set an ACL entry for an object
PCreate	no	Create a partition (associate a name with an ObjectID)
PDelete	no	Delete a partition (remove a name/ObjectID association)
PList	yes	List the partitions
PMount	yes	Retrieve the ObjectID given its name
Sync	not applicable	Sync the entire cache to disk
Flush	not applicable	Removes all versions of all objects between two times
Flush0	not applicable	Removes all versions of an object between two times
SetWindow	not applicable	Adjusts the guaranteed detection window of the S4 device

表 2.1: RPC コマンドリスト

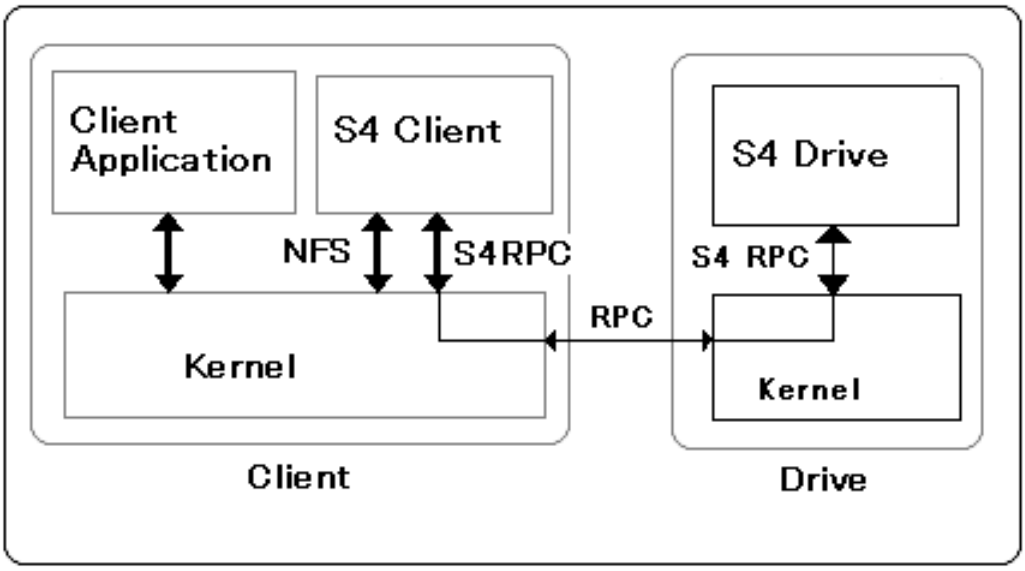


図 2.1: S4 の概観：NFS のリクエストから S4RPC への変換をクライアント側で行う

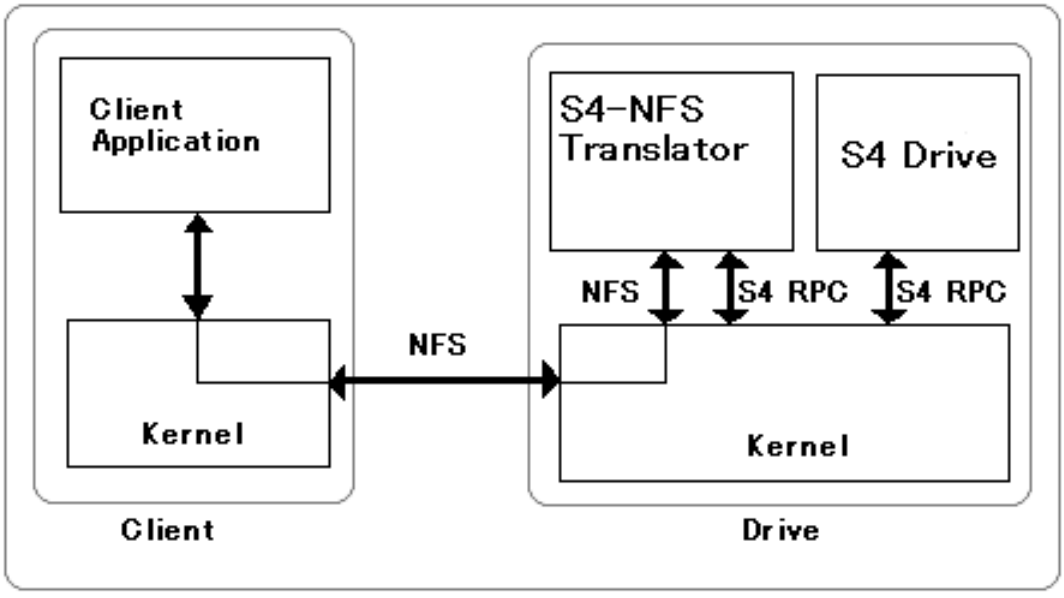


図 2.2: S4 の概観：NFS のリクエストから S4RPC への変換をサーバー側で行う

2.4　セキュア OS -SELinux-

2.4.1　セキュア OS とは

いままでのＯＳでは十分に行われていなかったアクセス制御を強化し、より強いセキュリティを持つものがセキュア OS である。

セキュア OS と従来 OS の違いは、従来 OS のセキュリティ技術が「どのように侵入を阻止するか」に注目していたのに対し、セキュア OS では「コンピュータに侵入された後のこと」に着目している。セキュア OS ではアクセス制御を強化して、侵入を実質的に無力化する。

これまでの OS が抱えている問題には以下のようなものがある。

- 任意アクセス制御と呼ばれる、ファイルの所有者がアクセス権を勝手に変更できる仕組みがある。
- root アカウントは、アクセス制御を無視して全てのファイルにアクセスできる。
- プロセスに特権を与える際に余計な特権までも与えてしまう。プロセスが乗っ取られた場合、システムに大きな被害を及ぼす。

セキュア OS では、上記の問題を解決するために「強制アクセス制御（MAC:Mandatory Access Control）」と呼ばれるアクセス機構を備えている。これは、ファイルの所有者が勝手にアクセス権を変更できないようにし、システム全体を管理する者の意図通りのアクセス権を強制するものである。

MAC は root アカウントにも強制力を持つ。root アカウントがファイルなどのアクセス権を変更するのには、特定の手順を用いてシステムの状態を変更してからアクセス権を変更する必要がある。

もうセキュア OS の一つの特徴は「最小特権」である。セキュア OS ではプロセスに特権を与える際に、全ての特権を一度に与えるのではなく、細かく分割された特権を少しずつ与えることができる。これにより、プロセスに対して余計な特権を与えてしまう可能性が少なくなり、仮にプロセスが乗っ取られたとしてもシステムに及ぼす影響を最小限にできる。

セキュア OS とは、MAC と最小権限という 2 つの機能を満たしているものとされている。Linux 用の代表的なセキュア OS モジュールには、

- SELinux
- LIDS
- RSBAC

がある。

2.4.2　LSM

Linux カーネル 2.6 には、新機能として「LSM（Linux Security Module）」機能が追加された。LSM とは、カーネル j 内のセキュリティチェック機構へのフック関数群を定義するフレームワークを提供する機能（運用環境固有のセキュリティカーネルを実装するための機能）である。LSM を用いることで、カーネルのセキュリティチェック機能をユーザーが独自に拡張することが可能である。

LSM を有効にすると、I/O ポートアクセスの許可などのセキュリティチェックポイントで、ユーザーが登録した LSM のコールバック関数が呼び出され、操作の正当性チェックが行われる (2.32.4)。

LSM で定義可能なセキュリティ・チェックポイントは、150 項目以上にも及ぶ。主な項目として、以下のような操作に対するセキュリティ機構が実装可能である。

- I/O ポートへのアクセスの可否
- ホスト/ドメイン名の設定の可否
- システムのシャットダウンの可否
- プロセスの生成/終了の可否
- 各種シグナル操作の可否
- 各種ファイルシステム操作の可否
- 各種ソケット操作の可否

2.4.3　SELinux

SELinux(Security-Enhanced Linux) とは、LSM に対応した Linux カーネルのセキュリティ拡張モジュールである。SELinux は Linux カーネルに「セキュア OS」の機能を付加する。

アプリケーションにセキュリティホールが発見されてからパッチが供給され、そのパッチが適用されるまでの間は、サービスを停止するか、攻撃を受けないことを期待しながらセキュリティホールのある状態で運用を継続するしかない。どちらにしても、いち早くパッチを適用することが最優先となる。しかし、パッチの適用には、それが稼働中の他のアプリケーションに及ぼす影響も検証しなければならない。検証のために十分な時間を取ろうとすると、パッチの適用がおくれ、クラッカーから攻撃を受けてしまうかもしれない。これは、Linux が root アカウントという絶対的な

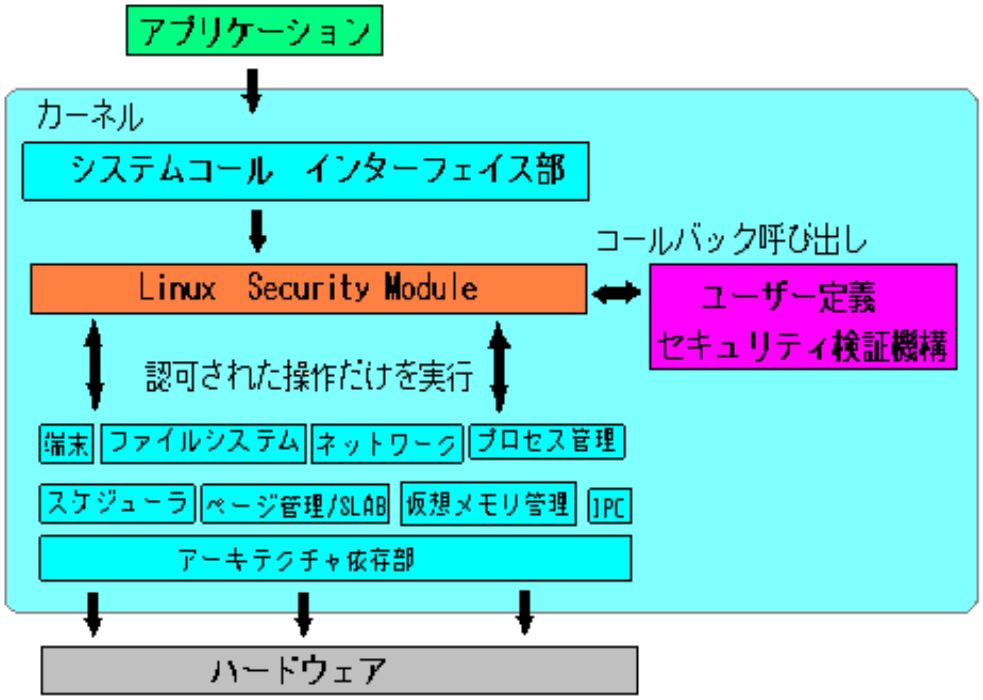


図 2.3: LSM の構造

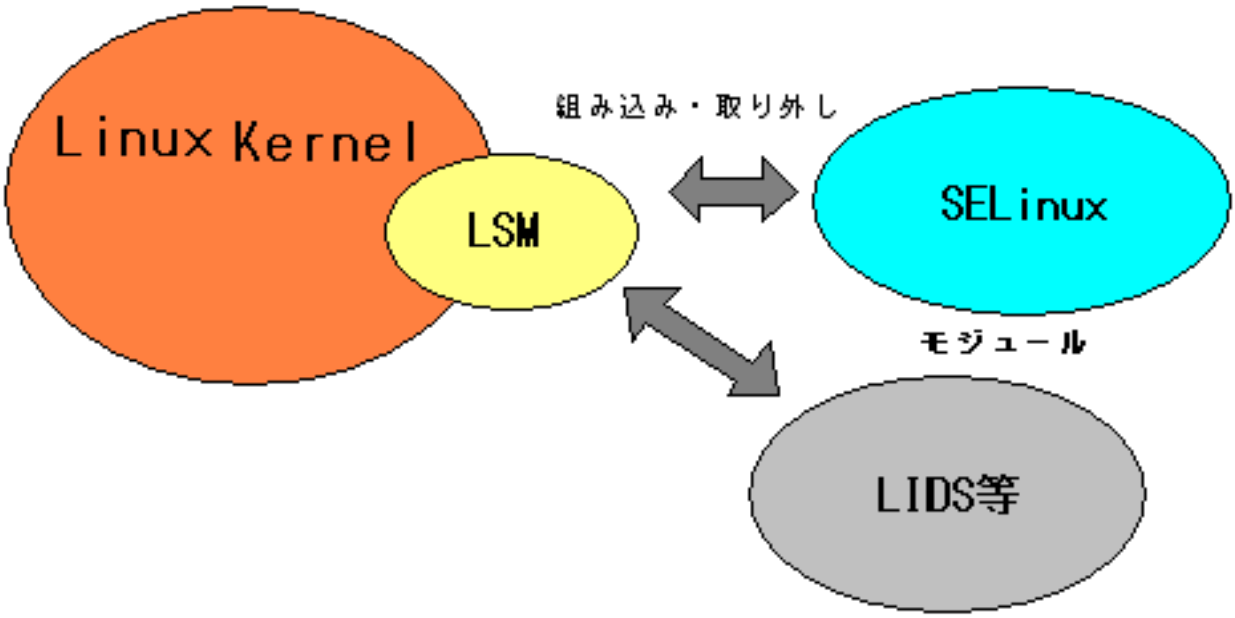


図 2.4: LSM の構造 2

管理アカウントに依存していることが問題である。root に全権限を与えてしまっているために、攻撃者に侵入を許し、root 権限を奪われてしまったときの被害が大きくなってしまうからだ。

SELinux を導入することで、この問題はある程度解決できる。SELinux では、従来の root に依存した管理や、単純なファイルオーナーによる権限制御、より厳格な 3 種類のアクセス制御機構によって補完する。

3 種類のアクセス制御機構とは以下のものである。

- **TE**
  従来の Linux では、動作するプロセスはそれぞれを実行するユーザー権限でファイルなどのリソースにアクセスする。つまり、各プロセスに対して、オーナー、グループとパーミッションに基づいた権限の制御しかできなかった。SELinux では「TE（Type Enforcement）」という機構が用意されている。TE ではプロセスに対し「ドメイン」、ファイル等のリソースに対しては「タイプ」というラベルが付加される。また、各リソースは、ファイルやディレクトリ、ソケットなどの「オブジェクトタイプ」ごとに「アクセスベクタ」が割り当てられる。アクセスベクタとは、リソースに対して行える操作の種類である[3]。例えば、ファイルに対するオブジェクトタイプ「file」には read,wirte,lock、append など約 20 種類のアクセスベクタが存在する。SELinux ではプロセス（ドメイン）毎に、タイプに対して許可されるアクセスベクタを設定できる。つまり、あるプロセスはファイルの読み書きができるが、別のプロセスはファイルを読むことしかできないといったように、プロセス毎に固有のアクセス制御を詳細に行うことができる。図 2.5 の例は、httpd プロセスが/var/www/ディレクトリ以下のファイルに read のアクセスのみできるという設定である。httpd プロセスに「httpd_t」というラベルを、/var/www/ディレクトリ以下に「httpd_sys」というラベルがついて、プロセスとリソースの間にはアクセスベクタ「read」が設定されている。httpd プロセスが書き込みを行うならば、「write」というアクセスベクタを追加して設定する必要がある。

- **RBAC**
  プロセスからリソースへのアクセス権限を制御する TE に対し、RBAC(Role Based Access Control) はユーザのリソースへのアクセス制御機構である。ユーザにはそれぞれ管理者、一般ユーザといったロールを割り当て、各ロールには最低限必要な操作が可能なようにポリシを設定し、リソースへのアクセスを許可する。RBAC

[3] 正確には、「read,write」等のバイナリ表現のことを「アクセスベクタ」と呼ぶそうである。

図 2.5: TE の例

を使うと root アカウントであっても他のユーザと同じように限定された権限しか許可されなくなる。これにより、攻撃者が root 権限を奪取したとしても、割り当てられたロールの範囲でしか攻撃を加えられないため、大幅にリスクを減らすことができる。

- **MAC**
  Linux をはじめ、一般的名 UNIX 系 OS の権限制御（ファイルオーナー、グループとパーミッションによる管理）は、DAC（Discretionary Access Control）-任意アクセス制御-と呼ばれる。DAC では、自分の所有するファイルであれば、そのファイルに対する各種許可を自由に設定できる。これは、各ユーザーにとって利便性が高い反面、人為的なセキュリティホールの原因になる。

  これに対し、SELinux は MAC（Mandatory Access Control）-強制アクセス制御機能を提供している。MAC では、ポリシファイルに記述された設定に基づいて、TE,RBAC 等のアクセス制御を全てのユーザとプロセスに例外なく適用する。ポリシ・ファイルの設定を行えるのは、限定されたシステムの管理者のみである。

また、SElinux にはプロセスのアクセス制御に関連してドメイン遷移という機能がある。この機能は、親プロセスが子プロセスを生成する場合に、子プロセスに親プロセスとは異なるドメインを割り当てるというものである。この機能によって、子プロセスを生成を繰り返す複雑なプロセスに対しても、細かい権限制御が可能である。

SELinux の欠点

SELinux にも対処できない問題がある。それは、DoS 攻撃などの侵入以外の攻撃や、限定された権限内での行為は対応できない。また、OS そのものにセキュリティホールがある場合にも攻撃を防ぐことはできない。

2.5　ファイルサーバー

ファイルサーバーは記憶装置をネットワーク上のほかのコンピュータと共有し、外部から利用できるようにするコンピュータである。ファイルサーバー上にあるファイルは許可をされている人であれば誰でも他のコンピュータから読み込んだり書き込んだりできる。

ファイルサーバーはクライアントがファイルアクセスをする際に認証を行い、ファイルを保護している。認証の方法は、IP アドレスによる認証やパスワード入力を求めるなどである。これらの認証方法では、ファイル保護は十分に行えない。IP アドレスによる認証の場合は IP アドレスを偽称されてしまうとファイルアクセスを許してしまう。パスワードによる認証では、クライアントシステムにキーロガーを仕掛けられてしまうとパスワードが漏れ、攻撃者が認証に成功してしまう。また、ファイルサーバー自体が攻撃される危険もある。

SELinux やファイルサーバーは OS に脆弱性がない場合には有効かもしれないが、プログラムにバグや、脆弱性がないことを保証するのは難しい。SELinux での権限の分割によるファイル保護も、OS カーネルへの攻撃によって、無効化される可能性がある。従って、OS カーネルに依存しないようなアクセス制御が必要である。作業している OS カーネルから分離されたアクセス制御では、ユーザーが作業しているシステムに障害がでた場合にも、正常に機能する。作業している OS にキーロガーが仕掛けられてもパスワードが漏れないなどの利点がある。

第3章　SecureAccess

既存システムでは、OS そのものに脆弱性がある場合や、キーロガーなどによるパスワード漏洩などの問題に十分対応はできていない。本研究では、これらの問題を解決しファイルアクセス制御を行うシステムとして、SecureAccess の設計と実装を行った。以下で、SecureAccess について述べる。

3.1　特徴

SecureAccess は、ファイルアクセス制御を操作中の OS から分離することで、操作中の OS に障害が起きた場合であっても、設定したポリシー通りにファイルアクセス制御を行うシステムである。SecureAccess の特徴は以下である。

- ファイル単位、ディレクトリ単位での細かいアクセス制御を行い、管理者権限を持ったユーザが無条件に全てのファイルへアクセスできてしまう状況を回避している。

  ファイル、フォルダをまとめて、グループを作成し、グループ単位でファイルアクセス制御を行う。グループ内のそれぞれのファイルに対し、Write-only,Read-only,Write and Read,APPEND などのパーミッションを設定可能である。

  ```
  〈 group1 〉
  w:
  /home/username/thesis/example1.txt
  /home/username/thesis/example2.txt
  r:
  /home/username/thesis/read-only/
  rw:
  /home/username/thesis/exampleRW.txt
  append:
  /home/username/thesis/access.log
  ```

例えば、上記のように設定ファイルを書いた場合、group1 はそれ以降に書かれたファイル、ディレクトリから構成される、という意味になる。認証が成功すると、グループ内のファイル・ディレクトリに対してアクセスが許可される。上記の例では、/home/username/thesis/ ディレクトリの example1.txt と example2.txt は Write 操作のみ、read-only ディレクトリにあるファイルは Read 操作のみ、exampleRW.txt には Read と Write 操作の両方を許可し、access.log ファイルは append 操作のみ許される。

SecureAccess では認証によって、ファイルアクセスできるようになる期間が設定可能であり、その期間は認証時にユーザーからの入力により決定される。

- ファイルアクセスの際の認証を操作 OS とは別のシステムで実施することで、操作 OS と独立してファイルアクセス制御を行う。認証に利用されるシステムはリモートからはアクセス不可能である。認証は、認証用のシステムに物理的に接続されたキーボードからのパスワード入力のみにより行われる。これにより、操作中の OS にキーロガーを仕掛けられた場合に、パスワードの漏洩が防げる。ユーザが操作するシステムと SecureAccess でファイルアクセス制御を行うシステムが分離しているため、操作中の OS に障害が発生した場合であっても、SecureAccess のファイルアクセス制御は設定されたポリシーに従って正常に動作する。

- ユーザーがアクセスするファイルは認証を行うシステムで保持し、ファイル操作も認証を行うシステムが行う。従って、ユーザは管理者権限を持っていてもファイルアクセスに対して、認証システムを回避できない。侵入者が操作 OS の管理者権限を奪った場合にも、ファイルアクセス制御を強制できる。

3.2　システムの構成

SecureAccess のシステムは大きく 2 つの部分に分けられる。(ユーザ) 作業用のシステムと、認証・ファイル操作を行うシステムの 2 つである。この 2 つのシステムはバーチャルマシンを用いて、1 台のマシン上で動作させる。

- 作業用システム

  ユーザーは作業用のシステムで作業を行う。ファイルにアクセスする操作があった場合には、そのリクエストを認証・ファイル操作用

のシステムに転送し、認証・ファイル操作用のシステムからの返ってくる結果のみを利用する。

- 認証・ファイル操作用システム

  作業用のシステムからリクエストを受け取ったら、認証を実施する。認証は、認証システムに直接繋がったキーボードからのパスワード入力により行う。認証に成功したら、要求されたファイル操作を行い、結果を作業用システムに返す。

実際の処理の流れは、ファイル操作の度に作業用システムから認証・ファイル操作システムへ移り、ファイル操作が終わると作業用システムに戻ってくるというようになる (図 3.1)。

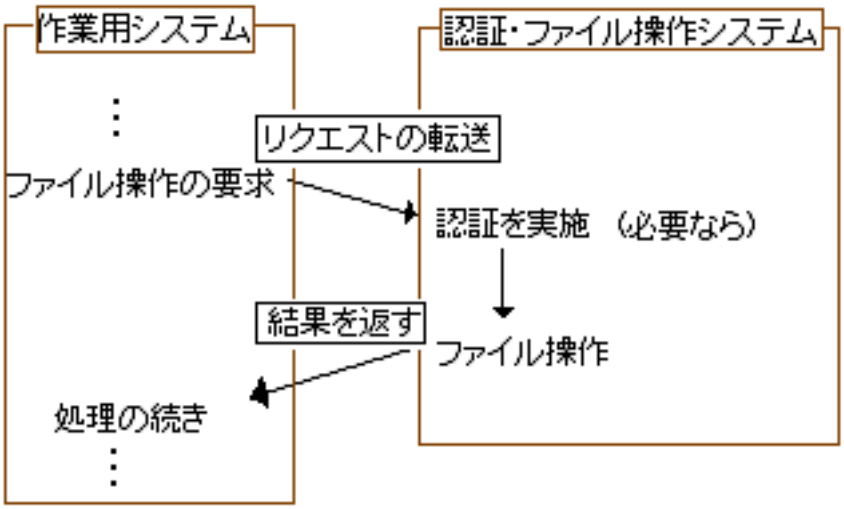


図 3.1: 処理の流れ

3.3　SecureAccess の提供する機能

3.3.1　安全性の確保

想定する脅威

SecureAccess は、リモートから OS カーネルが乗っ取られた場合の攻撃者による任意のファイルへのアクセスを防ぐ。例えば、OS カーネルの脆弱性を攻撃されて、管理者権限を奪取された場合がこれにあたる。

認証

従来システムにおける、ファイルアクセス制御は、ユーザーが作業しているOSが行っていたが、SecureAccessにおけるファイルアクセス制御では、ユーザーが作業しているOSでは行わず、リモートから隔離された別システムにおいて実施する。ファイルアクセス制御における認証は、ファイルopen命令についてのみ行われる。

確保される安全性

SecureAccessを使用すると、侵入者による任意のファイルへのアクセスを制御することができる。これは、たとえ侵入者が管理者権限を手にいれた状況でも同様である。侵入者が奪取できる権限は、作業用のOSに関するものだけであり、認証・ファイル操作を行うOSの権限を入手することができないためである。この理由は、認証・ファイルアクセスのシステムが、リモートからアクセスできないようになっているためである。また、侵入者により仕掛けられたキーロガーによるパスワードの漏洩もない。何故ならば、認証に使われるOSがキーロガーの仕掛けられたOSとは別のOSであるからである。

3.3.2　SecureAccessの制限

SecureAccessではファイルアクセス制御に関して、従来のシステムで行っている制御の他に、２つのアクセス制御の機能を持っている。時間的制限と、空間的制限の２つである。

時間的制限

時間的制限とはファイルアクセスの際の認証により、ファイルアクセスを許可する期間を制限する機能である。ファイルアクセス毎に認証を行い、パスワード入力を求めるのでは、利便性が失われてしまう。そこで、SecureAccessでは一度認証を成功させると、ある一定の期間の間はファイルアクセスに対して認証を行わなくてもファイルアクセスができるようにしてある。このファイルアクセスを許可する期間は、認証時にユーザーが設定することができ、ユーザーがこれから行おうとしている作業の時間だけファイルアクセスを許可（認証をスキップ）するというようなことが可能である。

空間的制限

SecureAccess では、ファイルをグループにまとめて、グループ毎にアクセス制御を行っている。グループの定義はユーザーが SecureAccess の設定ファイルに記述しておく必要がある。また、グループに定義されるそれぞれのファイル・ディレクトリに対して、パーミッションを指定できる。指定できるパーミッションには、読み出しのみ許可、書き込みのみ許可、読み書き両方許可、追記のみ許可がある。所属するグループの定義されていないファイルは、それらをまとめて１つグループとして扱われる。ユーザーに定義されていないファイルに対しては、読み出しのみ許可などのパーミッションの設定はできない。

SecureAccess はファイルオープンのリクエストを受け取ると、オープンするファイルがどのグループに属するか調べる。ファイルの所属するグループが見つかると、次にファイルオープンのモード（読み出しのみや追記等）をチェックを行い、許可されていないモードであれば、エラーを返す。許可されているモードであれば、認証を行う。認証に成功すると、そのファイルが属するグループに対してファイルアクセスが許可される。

一度認証を行った後、再びファイルオープンを行う場合、オープンするファイルの属するグループを調べ、所属するグループが以前に認証を行っていて、ファイルアクセス許可されている期間内であれば、ファイルオープンのモードのみチェックを行い認証はスキップされる。ファイルの属するグループがファイルアクセスを許可されていない場合は、認証を行う。

第4章　実装

この章では、システムで用いられている既存の技術についての説明を交えながら、SecureAccess の実装について述べる。SecureAccess の本実装は、SecureAccess のアイデアを試すためのプロトタイプであり、実験による性能測定では大きなオーバーヘッドが生じているが、実際に OS に組み込み場合には後述のシステムコールのトラップや、通信のコストが無くなる又は大幅に小さくなる。

4.1　Xen

前章で述べたように SecureAccess では、作業用システムと認証・ファイル操作用システムの２つのシステムを利用する。システムの利便性を考慮してこれらの２つシステムを、２つのヴァーチャルマシンを用いて一台のマシン上で稼動させる。２つのシステムを一台で稼動させるために仮想マシンモニタである Xen[3] を使用する。

Xen の提供する機能を簡単にいうと、以下の２つになる。

- 仮想マシンの事項環境の準備
- 仮想マシン実行時の各種管理機能の提供

Xen は仮想マシンモニタであり、仮想マシンの実行単位を「ドメイン」として管理する。ドメイン上で動作するものは「OS」である。

Xen は完全な PC/AC 互換機の環境をエミュレートするのではなく、物理ハードウェアの上に Xen のマイクロカーネルがあり、その上で複数のゲスト OS が動作する。物理ハードウェアへのアクセスは Xen のマイクロカーネルが制御する。そのため、それぞれのゲスト OS のカーネルは Xen マイクロカーネルのサービスを利用するように修正する必要がある。しかし、その上で動くアプリケーションに関しては修正は不要である。

Xen 自体の制御は、ドメイン０と呼ばれる特別なゲスト OS から行う。ドメイン０では、Xen を制御するユーザーインターフェイスだけでなく、物理ハードウェアのデバイスドライバを含むドメイン０カーネルが動いている。

その他のゲスト OS が動く仮想マシンをドメイン U と呼ぶ。ドメイン U で動く OS カーネルは物理デバイスへのアクセスをドメイン 0 に依頼するドライバが動いている (図 4.1)。

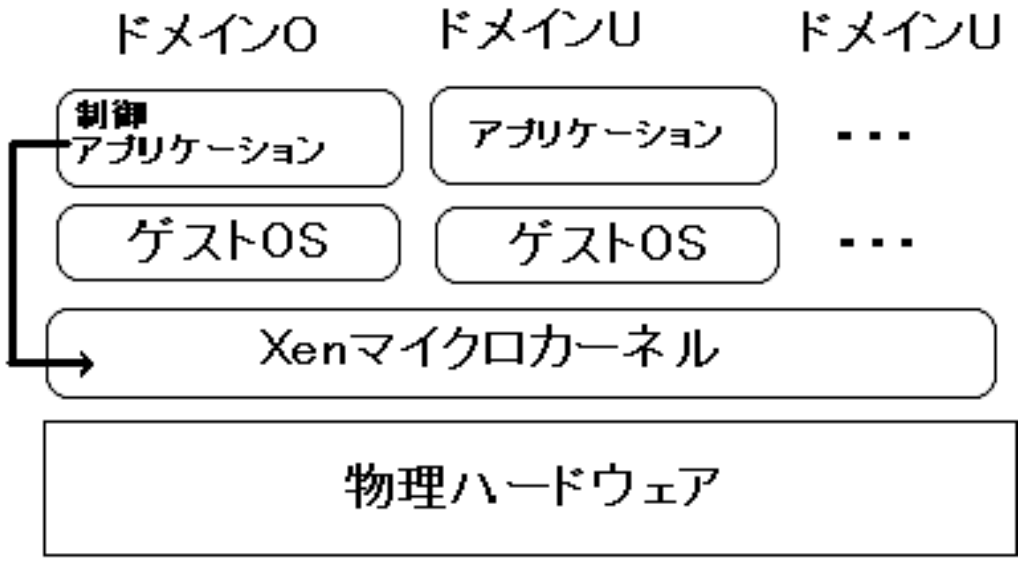


図 4.1: Xen

4.2　SecureAccess の構造

SecureAccess の構造は図 4.2 のようになる。作業用 OS 上では SACore が、認証・ファイルアクセス制御用 OS 上では SAserver が動いている。SAcore は SACatcher と SACommunicator から構成される。SACatcher はアプリケーションからのファイルアクセスのリクエストをトラップし、ファイルアクセスに必要な情報を取得し、SACommunicater に受け渡す役割を果たす。SACommunicator は SAServer との通信の役割を担う。SAsever では、ユーザーの認証とリクエストされた操作の実行を行う。

4.3　SACatcher

SecureAccess のプロトタイプを実装するにあたって、作業 OS によるのファイル操作を認証・ファイル操作 OS に転送する部分の実装（図 4.2 における SACather 部分）に対して 2 つの実装方法を試した。1 つは、C 言

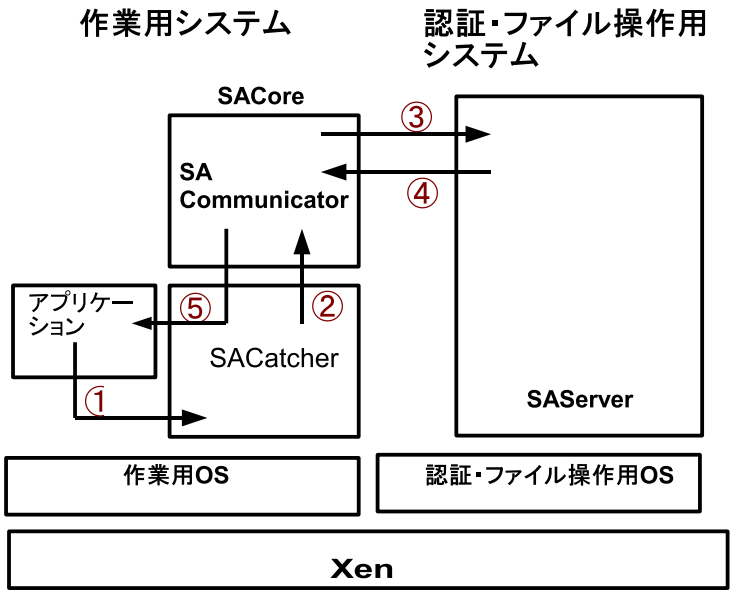


図 4.2: SecureAccess の構造

語の標準ライブラリを置き換える実装方法であり、残りの 1 つは ptrace によりシステムコールをトラップする方法である。SecureAccess のプロトタイプでは、適用範囲の広さから、後者の方法を採用した。

4.3.1　C 言語の標準ライブラリを置き換える方法

C 言語に用意されている標準ライブラリ（関数）の呼び出しを自分で定義した関数の呼び出しで置き換える方法である。UNIX 系のシステムでは、環境変数 LD_PRELOAD を標準ライブラリ関数を置き換える関数のあるフォルダ（ファイル）に設定すると、標準ライブラリ関数がユーザー定義の同名の関数に置き換えられる。この方法の長所と短所は以下の通りである。

- 長所：実行速度が速い。
- 短所：C 言語の標準ライブラリを使用しているアプリケーションにしか適用できない。

4.3.2　システムコールをトラップする方法

ptrace

システムコールをトラップするのに ptrace というシステムコールを用いた。ptrace() システムコールは、親プロセスが別のプロセスの実行を監視したり、制御したり、コアイメージやレジスタを調べたり変更したりする手段を提供する。ptrace による実装は、処理速度は遅いがシステムコールのトラップにより全てのアプリケーションに対して適用できるという特徴がある。SecureAccess のプロトタイプでは、汎用性という観点から ptrace による実装を採用した。

ptrace を使うと、実行中のプロセスに対して、レジスタの書き換えやメモリ上のデータの書き換えといった様々な操作を行うことができる。ptrace の書式は以下である。

```
#include <sys/ptrace>
long ptrace(enum _ptrace_request request,pid_t pid,void *addr,void *data);
```

_ptrace_request の値が ptrace() システムコールの動作を決定する。_ptrace_request の値として以下の定数が用意されている。

- PTRACE_TRACEME

  プロセスが親プロセスによってトレースを開始される。このトレースされるプロセスに（SIGKILL を除く）シグナルが配送されると、プロセスは停止し、親プロセスに wait() を通じて通知される。以後このプロセスが exec() を呼び出すと、SIGTRAP が送信される。これにより、親プロセスは新しいプログラムが実行を開始する前に制御できる。

- PTRACE_PEEKTEXT,PTRACE_PEEKDATA

  子プロセスのメモリの addr の位置から 1 ワード読み出す。読み出した値は ptrace() の返り値になる。

- PTRACE_PEEKUSR

  子プロセスの USER 領域[1]のオフセット addr の位置から 1 ワード読み込む。読み出した値は ptrace() の返り値になる。

- PTRACE_POKETEXT,PTRACE_POKEDATA

  ワード data を子プロセスのメモリの addr の位置にコピーする。

[1]USER 領域にはそのプロセスのレジスタ (registers) などの情報が保持されている。

- PTRACE_POKEUSR

  ワード data を子プロセスの USER 領域のオフセット addr の位置にコピーする。

- PTRACE_GETREGS,PTRACE_GETFREGS

  PTRACE_GETREGS は子プロセスの汎用レジスタ、PTRACE_GETFREGS は浮動小数点レジスタを親プロセスの data の位置にコピーする。

- PTRACE_SETREGS,PTRACE_SETFREGS

  PTRACE_SETREGS は子プロセスの汎用レジスタ、PTRACE_SETFREGS は浮動小数点レジスタに親プロセスの data の位置からコピーする。

- PTRACE_CONT

  停止した子プロセスの実行を再開させる。

- PTRACE_SYSCALL,PTRACE_SINGLESTEP

  PTRACE_CONT と同様に停止した子プロセスを再開させるが、PTRACE_SYSCALL の場合は子プロセスが次にシステムコールに入るかシステムコールを抜けるかする時に、PTRACE_SINGLESTEP の場合は 1 命令実行したあとに停止するようにする。

- PTRACE_KILL

  子プロセスに SIGKILL を送り終了させる。

- PTRACE_ATTACH

  pid で指定されたプロセスに接続し、その指定したプロセスを子プロセスとしてトレースできるようにする。子プロセスは PTRACE_ATTACHME したかのように振舞う。

- PTRACE_DETACH

  プロセスのトレースを解除し、停止した子プロセスを再開する。

ptrace でのプロセス監視開始

プロセスの監視は PTRACE_ATTACH か PTRACE_TRACEME によって開始される。PTRACE_ATTACH は親プロセスがトレースと行うことを宣言し、PTRACE_TRACEME は子プロセスがトレースされることを宣言する。

図 4.3 は 2 つのプロセスの監視をはじめる方法である。PTRACE_ATTACH は SIGSTOP を生成して子プロセスをとめてからアタッチを行うが、SIGSTOP

```
Fig.1
  pid=fork();
  if(pid==0){execl(...);}
  ptrace(PTRACE_ATTACH,pid,NULL,NULL);
Fig.2
  pid=fork();
  if(pid==0){
    ptrace(PTRACE_TRACEME,0,NULL,NULL);
    execl(...);
  }
```

図 4.3: ptrace の初回プロセスアタッチ

は発行されたその瞬間にあるわけではないので、Fig.1 では子プロセスの初めのいくつかのシステムコールを制御できない可能性がある。

システムコールの制御

プロセスのアタッチが終わったら、PTRACE_SYSCALL でシステムコールを制御する。具体的には、以下のようになる。

- waitpid() で子プロセスがブレークするのを待つ
- waitpid() から返ってきた status を見て WIFEXITED と WSIGNAL でシグナルの種類を判別する。
- SIGTRAP だったら、制御処理。
- PTRACE_SYSCALL で次のシステムコールを待つ。

waitpid(pid_t pid,int * status,int options) は指定した子プロセスの状態変化を待つ関数である。状態変化とは次のいずれかである：子プロセスの終了、シグナルによる子プロセスの停止、シグナルによる子プロセスの再開。

status が NULL でなければ、waitpid() は status で指す int に状態情報を格納する。この整数を検査するためのいくつかのマクロが用意されている。SecureAccess で使ったマクロは以下である。

- WIFEXITED(status)

  子プロセスが正常に終了した場合に真を返す。

- WIFSIGNALED(status)
  子プロセスがシグナルにより終了した場合に真を返す。
- WIFSTOPPED(status)
  子プロセスがシグナルの配送により停止した場合に真を返す。
- WSTOPSIG(status)
  子プロセスを停止させたシグナル番号を返す。

4.3.3　SACatcher の実装

SACatcher の仕事は、アプリケーションからの open,write,read 等のファイルアクセスに関するシステムコールを ptrace を用いてトラップし、トラップしたシステムコールの引数の情報を取得し、SACommunicator に受け渡す。システムコールの引数は、子プロセスのシステムコール呼び出し時のレジスタの値を調べればわかる。子プロセスのレジスタの値は、_ptrace_request の値を PTRACE_GETREGS に指定すると data に示されるアドレスに書き込まれる。レジスタの値は、以下で定義される構造体として取得される。

```
struct user_regs_struct
{
long int ebx;
long int ecx;
long int edx;
long int esi;
long int edi;
long int ebp;
long int eax;
long int xds;
long int xes;
long int xfs;
long int xgs;
long int orig_eax;
long int eip;
long int xcs;
long int eflags;
long int esp;
long int xss;
}
```

大抵の user_regs_struct のメンバの役割は以下のようになっている。

- eax:システムコールの戻り値が入る
- ebx,ecx,edx,esi,edi:もしあれば、この順にシステムコールの引数が格納されている。
- esp:スタックポインタ。システムコールの引数が６つ以上ある場合には６つ目から先がスタックに格納されている。
- orig_eax:システムコールの番号が入っている。

SACatcher は子プロセスのシステムコールをトラップすると、子プロセスのレジスタの値から、システムコールの番号と、引数を取得し、ファイル操作に関係するシステムコールの場合は SACommunicator にそれらの情報を渡す。ファイル操作に関係しないシステムコールの場合は何もせずに子プロセスの処理を再開させる。SACommunicator は認証・ファイル操作システム上で動いている SAServer にシステムコール命令を転送し、SAServer 側で実行されたシステムコールの返り値を SACatcher に返す。返り値を受け取った SACatcher は、子プロセスのレジスタ eax に返り値を代入し、システムコールを実行させないために、システムコール番号（orig_eax の値）を-1 に修正した後子プロセスの処理を再開させる。SACatcher の処理をまとめると以下のようになる。

1. 子プロセスのシステムコール呼び出しをトラップする。
2. システムコールの番号を調べ、ファイルアクセス関係（open,read,write...）のシステムコールならば、システムコールの番号、引数を SACommunicator へ渡す。それ以外の場合は４へ。
3. SACommunicator から受け取った返り値とシステムコール番号-1 を子プロセスのレジスタにセットする。
4. 子プロセスの処理を再開させ１に戻る。

システムコールの引数の取得

システムコールを SAServer に転送するためには、子プロセスからファイル名などの文字列を取得する必要がある場合がでてくる。子プロセスのレジスタには、３２ bit 以下の整数（ポインタを含む）が格納されているだけある。そこで、文字列を得るために図 4.4 のような関数を定義し使った。この関数は、子プロセス pid のアドレス child_addr から、最大 length バイトの文字列を取得し、str に格納する。

```
int ptrace_getstring(int pid,char* str,int child_addr,int length){
      long data;
      char* tmp;
      int counter;
      length-=4;
      str[length-1]=0;
      for(counter=0;counter¡length;counter+=4){
        data=ptracec(PTRACE_PEEKTEXT,pid,(void*)(child_addr+counter),NULL);
        tmp=(char*)&data;
        str[counter+0]=tmp[0];if(tmp[0]==0)return 0;
        str[counter+1]=tmp[1];if(tmp[1]==0)return 0;
        str[counter+2]=tmp[2];if(tmp[2]==0)return 0;
        str[counter+3]=tmp[3];if(tmp[3]==0)return 0;
       }
       str[counter]=0;
      return 1;
    }
```

図 4.4: 子プロセスから文字列を取得する関数

4.4 SACommunicator

SACommunicator は SACatcher から受け取ったシステムコール命令の SAServer への転送と、SAServer からシステムコール命令の返り値や read 命令の場合には読み取ったデータを受け取る。受け取った返り値やデータは SACatcher により、子プロセスのレジスタや、メモリに書き込まれる。

4.4.1 UNIX のプロセス間通信

SAServer とのデータのやり取りをするには、SAServer と SACommunicator（SACore）間でプロセス間通信をする必要がある。

プロセス間通信を行うためには、通信をするプロセス同士が通信路で結ばれていなければならない。プロセスと通信路の接続点はソケットである。

ソケットは大きく分けると 2 種類にわけることができる。一つが UNIX ドメインのソケット、もう一つが INET ドメインのソケットと呼ばれる。

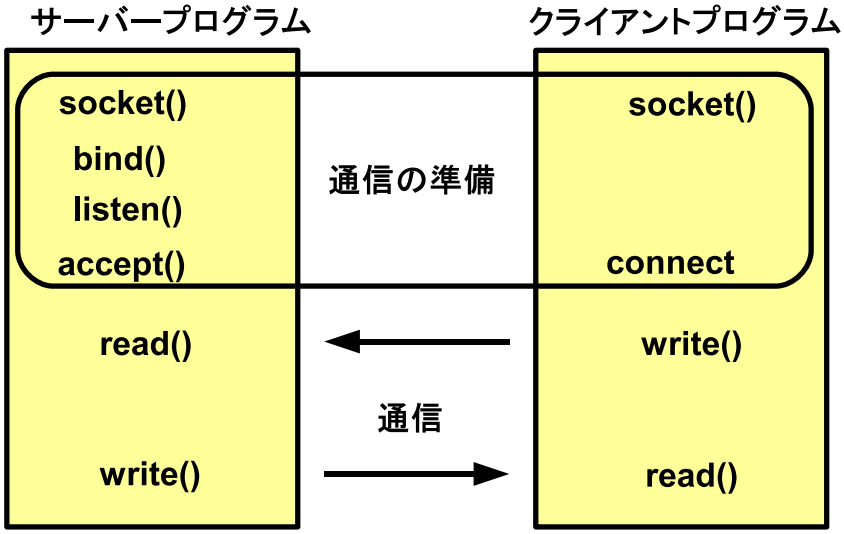
- **UNIX ドメインのソケット**

同じマシン内で動作しているプロセス同士で通信を行うためのソケット。他のマシン上のプロセスとは通信できない。

- **INET のドメインのソケット**

  ネットワークで繋がっていれば、離れたマシン上のプロセス同士でも通信を行うことができる。

ソケットを使って通信を行うには、ソケットを作ったり、ソケットを使ってメッセージを送信したりするための関数が必要である。C 言語には、ソケットを用いて通信を行うためのライブラリが用意されている。通信を行うための基本的な関数をまとめると図 4.4.1 のようになる。



それぞれの関数の説明は次。

- ソケットの生成 (socket)

  ソケット通信を行うためのソケットを用意する。

  int socket(int domain,int type,int protocol)

  domain には sys/socket.h で定義されている値を使う。UNIX ドメインの場合は PF_UNIX、INET ドメインを使う場合は、PF_INET,PF_INET6 等を指定する。

  type は同じく sys/socket.h で定義されている値を使う。代表的な値として SOCK_STREAM と SOCK_DGRAM があげられる。

  protocol にはソケットによって使用される固有のプロトコルと指定する。通常それぞれのソケットは、与えられたプロトコル・ファミリの種類毎に一つのプロトコルをサポートする。その場合は protocol に 0 を指定できる。

- ソケットアドレスの指定 (bind)

  socket() で作成したソケットに対して名前を割り当てる。ここでつけられた名前がクライアントが接続要求する際に指定する名前になる。

  int bind(int s,struct sockaddr *name,int namelen)

  s には soket() の返り値を渡す。name には sys/socket.h で定義されている sockaddr 構造体へのポインタである。socket() の引数の domain によって、内容が異なる。socket() の引数を PF_UNIX にしたときは、sockadd_un 構造体を使い、PF_INET の場合には sockaddr_in を使う。

```
struct sockaddr_un {
u_char sun_len;         /* 構造体の長さ*/
u_char sun_family;     /* AF_UNIX   */
char sun_path[104];    /*  ソケットのパス名*/
}

struct sockaddr_in {
u_char sin_len;       /*構造体の長さ*/
u_char sin_family;     /*PF_INET*/
u_short sin_port;      /*ポート番号*/
struct in_addr sin_addr;/*アドレス*/
char sin_zero[8];      /**/
}
```

- 接続準備 (listen)

  サーバーは listen により接続のための準備を行う。

  int listen(int s,int backlog)

  s はソケットの返り値である。backlog は接続要求を受け付ける数を指定する。

- 接続要求（connect）

  int connect(int s,struct sockaddr *name,int namelen)

  引数は bind() のものと同じである。

- 接続 (accept)

  サーバー側では、クライアントの connect 要求を accept によって受け入れる。

int accept(int s,struct sockaddr *addr,int *addrlen)

s は、socket() の返り値である。name には空の構造体ポインタを渡す。accept() が成功すると、システムが自動的に name に書き込む。接続要求が完了すると accept() はクライアントと接続された新しいソケットディスクプリタを返す。サーバはクライアントとの通信にこの新しいディスクプリタを使用する。

- 通信（read,write）

  接続が完了した後は、write(),read() で通信を行うことができる。

  ssize_t write(int d,const void *buf,size_t nbytes)

  s はソケットディスクプリタ、buf は送信したいデータである。。

  ssize_t read(int d,void *buf,size_t nbytes)

  s はソケットディスクプリタ、buf は受信したデータを置くバッファである。。

SecureAccess のプロトタイプでは、別システム間の通信をおこなうため、INET のドメインソケットを使用している。

4.5　SAServer

SAServer では、ユーザーの認証と実際のファイル操作を行う。

SAServer はまず、ユーザー定義のファイルアクセス制御のポリシーファイルの読み込みを行う。ファイルアクセス制御は、ファイル単位、ディレクトリ単位での指定が可能であり、それぞれに対して、ファイル操作の種類に対するパーミッションを定めることができる。指定できるパーミッションは、read-only,write-only,readAndwite,append である。図 4.5 に、ポリシーファイルの記述例を示す。

図 4.5 の記述には、グループが 2 つ定義されている。一つは thesis、もう一つは bibliography という名前のグループである。グループ thesis には、/home/takizawa/thesis/ディレクトリの thesis.pdf,thesis.tex,chapter1.tex,thesis.log ファイルが含まれ、グループ bibliography は/home/takizawa/biblilography ディレクトリ内のファイル全てが含まれる。さらにグループ thesis の各ファイルに対して、thesis.pdf は読み込みのみ、thesis.tex,chapter1.tex は読み書き、thesis.log は追記のみ許可し、グループ bibliography では /home/takizawa/bibliography ディレクトリ内のファイルは全て読み込みのみを許可するということを表している。認証はポリシーで定義されたグループ毎に行う。つまり、グループ内のいづれかのファイルアクセスに

```
⟨thesis ⟩
r:
/home/takizawa/thesis/thesis.pdf
rw:
/home/takizawa/thesis/thesis.tex
/home/takizawa/thesis/chapter1.tex
append:
/home/takizawa/thesis/thesis.log

⟨bibliography ⟩
r:
/home/takizawa/bibliography/
```

図 4.5: ポリシーの記述例

対して認証が成功した場合には、そのグループ内のファイル全てに対してファイルアクセスが許可される。

SAServer はまず、SACommunicator からシステムコール番号を受け取る。次に受け取ったシステムコールの種類を判別し、それぞれのシステムコールに対する処理を行う。

open システムコール

SAServer が受け取ったシステムコール番号が open のシステムコール番号であった場合、SAServer は open システムコールの引数 (ファイルパス、フラグ、モードの 3 つ) を SACommunicator から受け取る。SAServer はファイルパスからそのファイルの属するグループを調べ、認証を行う。認証が要求されると画面にパスワードとファイルを公開する（アクセスを許可する）期間を入力するダイアログが表示される。パスワードと時間を入力し、認証に成功すると SAServer は実際に open 命令を実行し、その返り値（ファイルディスクプリター）を SACommunicator に送る。取得されたファイルディスクプリターは記録され、read,write,close システムコールの呼び出し時の検査に利用される。

read,write,close システムコール

SAServer に送られたリクエストが read,write,close システムコールの場合には、認証は行わない。システムコールの引数を検査するだけである。SAServer はシステムコールの引数を受け取ると、それぞれの第一引数で

あるファイルディスクプリターが、正当なものであるかチェックする。正当であるとは、SAServer の open システムコールで取得されたものであるかということである。受け取ったファイルディスクプリターは open システムコールの時に記録したファイルディスクプリターのリストと比較される。リストに存在しなければエラーを返す。そうでない場合はそれぞれのシステムコール命令を実行する。

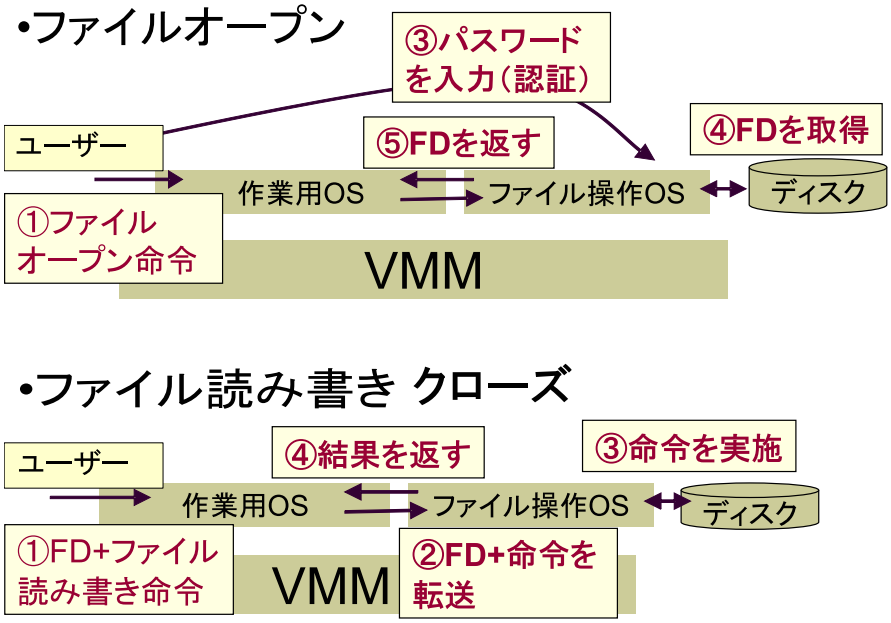


図 4.6: SecureAccess におけるファイルアクセスの手順

認証ダイアログ

SecureAccess では、認証のためにダイアログを出しパスワードを入力する。認証のためのダイアログは、作業用 OS とは別システムである認証・ファイル操作 OS によって作られる。別システムで認証を行うことでキーロガーが回避される。しかし、このダイアログは攻撃者によって仕掛けられたソフトウェア等によって出された偽のダイアログかもしれない。そこで、SecureAccess では認証用のダイアログが確実に認証用の OS から出されていることを確認する仕組みを持っている。

SecureAccess では、認証・ファイル操作用のシステムを起動したときにキーワードとなる文字列を標準入力から受け取る。この入力されたキーワードはキーロガーでは盗めない。何故ならば、認証用のシステムはリモートからはアクセス不能であるからである。キーワードは認証時に作成

されるダイアログのタイトルに表示される。よって、ダイアログのタイトルを見れば、そのダイアログが本当に SecureAccess の認証用システムから出されたものか判断できる。タイトルがキーワードと一致していれば、SecureAccess から出されたダイアログであり、それ以外は別のソフトウェアが出したダイアログである。図 4.7 はキーワードとして「HOGEHOGE」を打った例である。

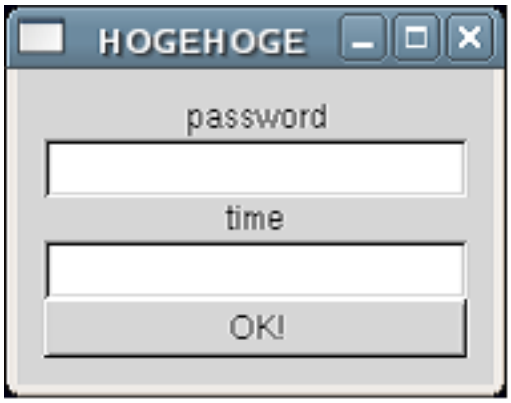


図 4.7: ダイアログ

4.6　使用例

この節では、SecureAccess のプロトタイプの使用例を順を追って説明する。ドメイン 0 で SAserver を動作させ、ドメイン U では SACore と実行したいプログラムを動作させる。Xen 上で、ドメイン U を GUI で操作するために VNC を使用する。

1. まず Xen を使い、ドメイン 0 とドメイン U の OS を起動する (図 4.8)。
2. ドメイン U において VNCserver を起動する。必要ならここでパスワードの設定を行う。
3. ドメイン 0 から VNCviewer でドメイン U に接続する。ダイアログがでるのでパスワードを入力する (図 4.9)。接続に成功すると、図 4.10 のような画面が出るので、ドメイン U での操作は VNCviewer を使って GUI 操作ができる。

図 4.8: Xen 上で 2 つの OS を起動

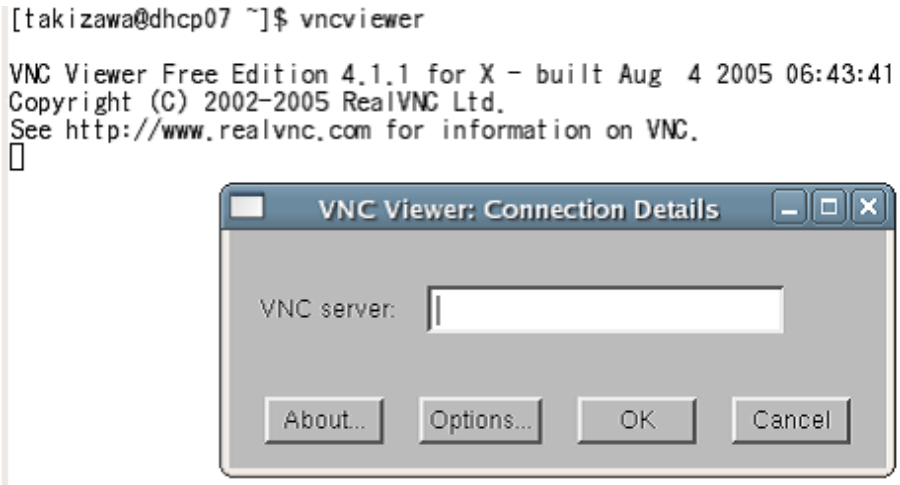


図 4.9: VNCviewer で接続

4. ドメイン 0 で SAserver を起動する。起動時に認証用ダイアログのタイトルに表示されるメッセージの入力を求められるので、好きな文字列を入力する。ここでは server test と入力した。（図 4.11）

5. ドメイン U で、動作させたいプログラムのパスを引数として、SACore を起動する (図 4.12)。すると、SACore によりプログラムの監視が始まる。例では「write_test」というプログラムを起動している。

6. プログラムがファイルアクセスをしようとするとダイアログが表示される (図**??**)。ダイアログのタイトルには、SAserver を起動したとき入力した文字列が表示されている。

7. ダイアログのパスワードの欄にあらかじめ定めておいたパスワードを入力し、ファイルアクセスを許可する期間を time の欄に入力し

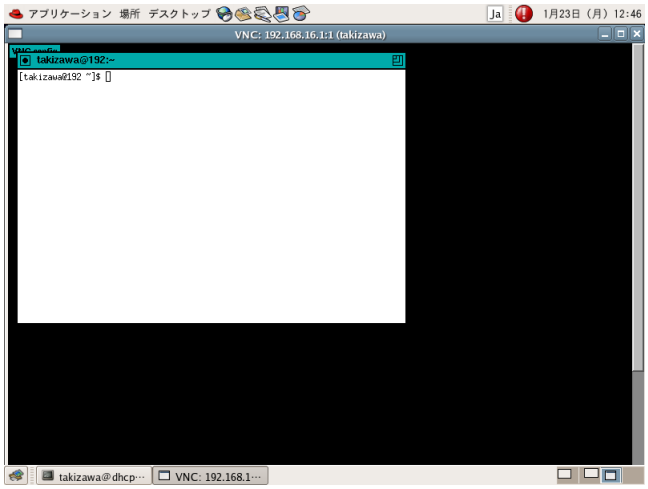


図 4.10: VNCviewer で接続 2

図 4.11: SAserver の起動

て、「OK」ボタンを押す。認証に成功するとファイルアクセスが許可されて、処理が続行される。失敗するとエラーが返る。

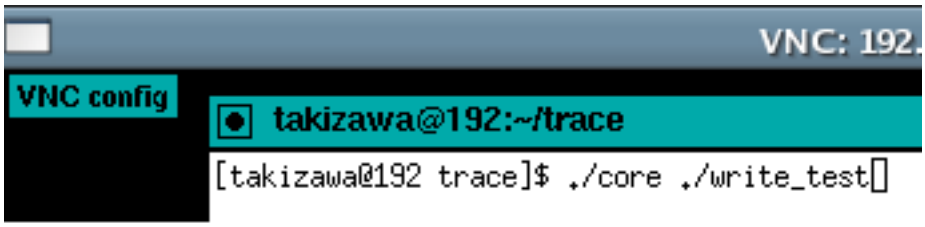


図 4.12: プログラムの起動

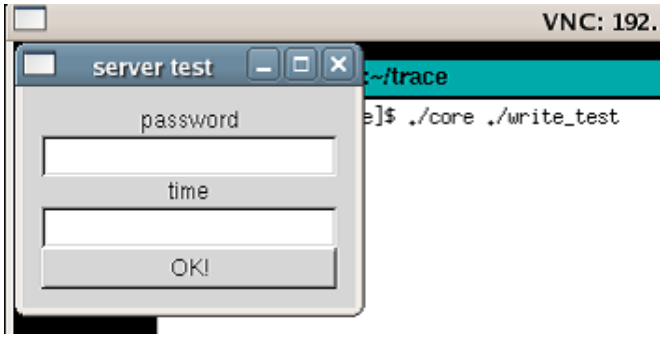


図 4.13: 認証用のダイアログ

第5章　実験

SecureAccess のファイルアクセス速度を測定するために実験を行った。内容としては、ファイルアクセス速度を測定するためにいくつかの単純なプログラムを作成し、実行速度を測定した。

5.1　実験環境

- CPU：PentiumD 3.00GHz
- MEMORY:1GB
- Xen2.0.7
- 作業用 OS:Fedora Core4 Linux
- 認証・ファイル操作用 OS:Fedora Core4 Linux

実験用に用意したプログラムは図 5.1 図 5.2 である。

```
int fd,i;
struct timeb t1,t2;

ftime(&t1);
fd =open("write_test_file",O_WRONLY | O_CREATS | O_TRUNC,...);
for(i= 0;i 〉 10000;i++){
write(fd,buf,128);
}
close(fd);
ftime(&t2);
```

図 5.1: write_test.c の抜粋

これらのプログラムの実行時間を、普通に実行した場合 (noSA)、SecureAccess を使用した場合 (useSA)、システムコールをトラップして子プロセスのメモリへのデータの読み出しや書き込みを行うがリクエストは転

```
int fd,i;
struct timeb t1,t2;

ftime(&t1);
fd =open("write_test_file",O_RDONLY);
while(read(fd,buf,128) !=0);

close(fd);
ftime(&t2);
```

図 5.2: read_test.c の抜粋

送しない場合 (only-ptrace) の 3 つの状況で比較した。実験の数値は 100 回試行を繰り返してその平均値をとった。結果は表 5.1 の通りである。

表 5.1: 実験結果 (ミリ秒)

	write 実行時間	read 実行時間
noSA	20	6
useSA	2309	1856
only-ptrace	1629	536

only-ptrace の実行時間と useSA の実行時間の差が SecureAccess を OS カーネルに組み込んだ場合の時間であるが、実際にはバーチャルマシン間の通信の時間も短縮されるので、この数値よりも小さなオーバーヘッドで済むと考えられる。

第6章　まとめ

本研究では、ファイルアクセス制御を作業用 OS から分離して行うシステムである SecureAccess の提案と実装を行った。本システムを利用することで、作業中の OS に依存しないファイルアクセス制御が行える。

作業用 OS からのファイルアクセス制御の分離には、ファイル操作に関係するシステムコールを制御する ptrace を用いた。さらに仮想計算機モニタである Xen を利用することで、ファイルアクセス制御を分離しながらも一台のマシン上で作業をすることが可能であり、ユーザーはファイルアクセス制御の分離を意識する必要はない。

本システムは、作業用の OS とは別システムによるファイルアクセス制御であるために、作業用 OS に障害が起こった場合にもファイルアクセス制御は正常に動作する。また、認証のためのパスワード入力には認証用のシステムを介して行われるために、キーロガーによるパスワードの漏洩も回避できる。

ファイルアクセス制御には、空間的制限と時間的制限を設けた。空間的制限とは、ファイルアクセスに対する認証において、認証成功時に作業用 OS に公開するファイルの範囲の制限のことであり、時間的制限とはファイルを公開する期間の制限のことである。空間的制限は、ファイル単位、ディレクトリ単位で指定可能である。また、それぞれのファイル、ディレクトリに対して読み込みのみ許可、書き込みのみ許可、追記のみ許可などの細かいパーミッションが設定可能である。

実験では、本システムを用いた場合と、通常のシステムを用いた場合でのファイルアクセスの性能の比較を行った。

今後の課題

今後の課題としては以下の事柄が挙げられる。

- **仮想計算機モニタの改造による実装**

  本論文では、提案したシステムである SecureAcess のプロトタイプの実装を行った。しかし、実装には ptrace によるシステムコールのトラップ、INET ドメインによるプロセス間通信を行っているため

に、オーバーヘッドが大きくなっている。ptrace、プロセス間通信で行っていることを仮想計算機モニタを改造して行うことで、オーバーヘッドが大幅に改善されると考えている。

参考文献

[1] Self-Securing Storage:Protecting Data in Compromised Systems John D.Strunk,Garth R.Goodson,Michael L.Scheinholtz,Craig A.N.Soules Gregory R.Ganger Carnegie Mellon University in Proc 4th on OSDI'00

[2] Storage-based Intrusion Detection:Watching storage activity for suspicous behavior Adam G.Pennington,John D.Strunk,John Linwood Griffin, Craig A.N.Soules,Garth R.Goodson,Gregory R.Ganger Carnegie Mellon University Proc 12th USENIX'03

[3] Xen and the Art of Virtualization:Paul Barham, Boris Dragovic,Keir Fraser,Steven Hand,Tim Harris,Alex Ho,Rolf Neugebauer,lan Pratt,Andrew Warfield, University of Cambrigde Computer Laboratory

[4] GTK+ 2.0 Tutorial
http://www.gtk.org/tutorial/

[5] NSA National Security Agency
http://www.nsa.gov/selinux/

[6] NAI Labs SELinux Brief(PDF)
http://www.mcafee.com/common/media/nai/pdf/nailabs/nai-labs-se-linux-2-26-02.pdf

[7] An Introduction to the NSA's Security-Enhanced Linux:SELinux
http://www.sans.org/rr/whitepapers/linux/232.php