平成19年度 学士論文

Xenによるゲスト OSの監視に基づく パケットフィルタリング

東京工業大学 理学部 情報科学科
学籍番号 04-1292-6
安積 武志

指導教員
千葉 滋 助教授

平成20年2月29日

概要

仮想マシンが普及してきたことで、1 つのマシン上で複数の OS が動作するようになり、ホスティングを行う場合、データセンターが仮想マシン単位で利用者に貸し出しを行えるようになった。これまで仮想マシンのセキュリティ管理では、仮想マシン内の OS 管理者が OS レベルで行っていた。OS レベルで行えば OS 内のプロセス情報を利用でき、詳細な制御が行える。しかし、仮想マシン内の OS 管理者は管理者としての知識が不足している場合がありえ、知識があったとしてもフルタイムで管理することは難しい。そこで、データセンターの管理者が常時管理することが望ましい。攻撃を受けた場合、即座に対処できることで被害を最小限に抑えられるからである。特に、仮想マシン上のプロセスが外部に攻撃を仕掛けているときには、データセンターの管理者がアクセス制御したい。しかしデータセンターの管理者が制御すると、仮想マシンから特定のポート番号に対する外部へのアクセスを一律に禁止することしかできない。これでは攻撃者にポートスキャンを行われた場合、全ポートへのアクセスを禁止するしかなくなってしまう。これでは善良なユーザもアクセスできなくなってしまい、望ましくない。

本研究では、仮想マシンモニタからゲスト OS のパケットをプロセス単位、ユーザ単位でフィルタリング可能にする xFilter を提案する。xFilter では仮想マシンモニタのメモリにゲスト OS のメモリをマッピングし、ゲスト OS 内の情報を参照することにより、ゲスト OS のプロセス情報を得ている。プロセス情報を基に禁止したい通信を行っているプロセスを探し、その通信が使っているポート番号を得て、ポート番号を基に仮想マシンモニタが通信を禁止する。仮想マシンモニタは、ゲスト OS のパケットを中継するため、すべてのパケットについて制御を行うことができる。xFilter は、仮想マシンモニタからゲスト OS 内のプロセス情報を取得するシステムと実際にフィルタリングを行うシステムの 2 つからなっている。これには仮想マシンモニタ Xen が提供する、仮想マシンのメモリを操作する関数を利用した。フィルタリングを行うシステムは、Linux のファイアウォール機能である iptables を用いて実装した。iptables はポート番号を指定してフィルタリングを行うことができるので、指定したプロセス、またはユーザが開いているポートを遮断することができる。プロセス単

位、ユーザ単位のフィルタリングが可能になったことで、仮想マシンモニタの管理者がゲスト OS に対してもゲスト OS 内で行う場合と同じ粒度でパケットのフィルタリングが行えるようになった。ゲスト OS 内のプロセス情報を取得できれば、その情報を用いてフィルタリングの設定が行えるようになるからである。また、xFilter では、フィルタリングするユーザプロセスを一覧表示する機能があり、管理者によるフィルタリングを支援する。我々は xFilter を Xen3.1.0、Linux2.6.18 に実装し、実験を行った。その結果、オーバーヘッドは平均的に小さいという結果を得た。

謝辞

本研究を進めるにあたり、研究の方針や、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、滝沢裕二氏、田所秀和氏には、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

仮想マシンが普及してきたことで、1つのマシン上で複数のOSが動作するようになり、ホスティングを行う場合、仮想マシン単位で貸し出しを行えるようになった。しかし、仮想マシン管理者は管理者としての知識が不足している場合があるだろうし、知識があったとしてもフルタイムで管理することは難しい。そこで、仮想マシンモニタ管理者が常時管理することが望ましい。なぜなら攻撃を受けた場合、即座に対処できることで被害を最小限に抑えられるからである。特に、仮想マシン上のプロセスが悪さをしているときには、仮想マシンモニタ管理者が制御したい。しかし、現在のシステムには問題がある。既存のセキュリティ技術には、OSレベルで行う方法とネットワーク上で行う方法がある。OSレベルで行えばOS内のプロセス情報を利用でき、詳細な制御が行える。しかし仮想マシンモニタから制御使用とした場合、ゲストOSに対するアクセス権がなければこれは不可能で、ネットワーク上で行う方法を取るしかない。ネットワーク上で行う場合、ゲストOS内の情報は利用できず、仮想マシン単位でしか制御できない。これでは仮想マシン全体のサービスを停止させてしまい、望ましくない。

本研究では、仮想マシンモニタからゲストOSのパケットをプロセス単位、ユーザ単位でフィルタリング可能にするxFilterを提案する。xFilterは、仮想マシンモニタからゲストOS内のプロセス情報を取得するシステムと実際にフィルタリングを行うシステムの2つからなっている。仮想マシンモニタからゲストOSのプロセス情報を取得するシステムは、仮想マシンモニタのメモリにゲストOSのメモリをマッピングし、ゲストOS内の情報を参照する。これには仮想マシンモニタXenが提供する、仮想マシンのメモリを操作する関数を利用した。フィルタリングを行うシステムは、Linuxのファイアウォール機能であるiptablesを用いて実装した。iptablesはポート番号を指定してフィルタリングを行うことができるので、指定したプロセス、またはユーザが開いているポートを遮断することができる。プロセス単位、ユーザ単位のフィルタリングが可能になったことで、仮想マシンモニタの管理者がゲストOSに対してもゲストOS内で行う場合と同じ粒度でパケットのフィルタリングが行えるようになった。ゲストOS内のプロセス情報を取得できれば、その情報を用いてフィルタリ

ングの設定が行えるようになるからである。また、xFilter では、フィルタリングするユーザプロセスを一覧表示する機能があり、管理者によるフィルタリングを支援する。我々は xFilter を Xen3.1.0、Linux2.6.18 に実装し、実験を行った。その結果、オーバーヘッドは平均的に小さいという結果を得た。

以下 2 章では、既存のセキュリティシステムとその問題点を述べ、3 章では本研究の実装の詳細と実装する上で必要な知識について述べる。また、4 章では実験を行い、xFilter を使用することによるオーバーヘッドがどの程度であるかを測定した。5 章では、今回できなかったさらにセキュリティ性の高い実装方法を紹介し、現在の実装との差異を議論し、6 章では本研究についてまとめる。

第2章　問題点と関連技術

2.1　クラックの危険性と対処

インターネットにつながったコンピュータはさまざまな攻撃にさらされており、大まかに侵入攻撃と DoS 攻撃の 2 つに分類することができる。侵入攻撃とは、攻撃者がセキュリティホールを利用してコンピュータの制御を乗っ取り、ホームページやシステムを破壊したり、機密データを盗んだり、さらには踏み台にして他のサイトに攻撃するようなものまである。DoS 攻撃とは、大量のパケットを送るなどしてサーバが提供しているサービスを停止させる攻撃である。侵入攻撃により侵入を許してしまうと、攻撃者のやりたい放題になってしまうため、特に被害が大きくなる。従来クラッカーなどの悪意のある人間が侵入攻撃を行ってきたが、近年はワームにより無差別攻撃が行われるようになっている。

侵入攻撃の方法

Linux に対する侵入攻撃の手順は図のようになる。まず、下見としてポートスキャンなどを行い、攻撃対象のセキュリティホールを見つけ、後の侵入を確実なものとする。ワームの場合は無差別に攻撃を行うので下見はスキップする。次に、下見で見つけたセキュリティホールを利用して、攻撃対象に侵入し、root 権限を奪取する。Linux では、root 権限はすべての権限をもっているので、あとは攻撃者のやりたい放題になってしまい、ジ・エンドである。侵入攻撃のメインである、侵入・root 権限奪取の部分をもう少し詳しく見ていく。この部分は、直接侵入攻撃と間接侵入攻撃に分けることができる。

- 直接侵入攻撃

  Linux では、root 権限で動いているデーモンプロセスが存在する。このようなプロセスにセキュリティホールがあると、攻撃者はセキュリティホールを悪用してプロセスの制御を乗っ取り、root 権限でやりたい放題できてしまう。プロセスの制御をのっとる方法としては、バッファオーバーフロー攻撃が最も一般的である。

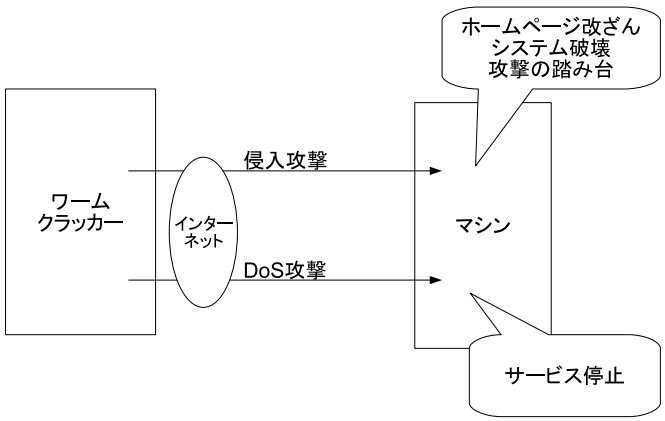

図 2.1: 攻撃手法

- 間接侵入攻撃

  最近の Linux では、ほとんどのデーモンプロセスは一般ユーザ権限で動くようになり、直接侵入攻撃を受けることは少なくなった。しかし、一般ユーザ権限で動いているからといって安心はできない。ローカル侵入攻撃を受けるからである。ローカル侵入攻撃の手順について説明する。まず、攻撃者はデーモンのセキュリティホールをついてデーモンを乗っ取ったり、一般ユーザのパスワードを推測してログインしたりして、一般ユーザ権限を得る。次に、攻撃者は SUID ビットが root になっているプログラム[1]を探す。例えば、passwd コマンドを使うと root しかアクセスできないパスワードファイルを書き換えることができる。それは、passwd コマンドの SUID ビットが root になっているからである。このような、SUID ビットが root になっているプログラムにセキュリティホールがあると、攻撃者はバッファオーバーフロー攻撃などでプログラムの制御を乗っ取って、root 権限を奪うことができる。

2.1.1　通信制御の必要性

実際に攻撃された例をあげると、研究室のサーバがクラッカーからの攻撃を受けたことがある。ssh による侵入攻撃で、おそらくパスワードを辞

[1]SUID ビットが root になっているプログラムとは、一般ユーザ権限でも使えるが、一時的に root 権限で動作するプログラムのことである。

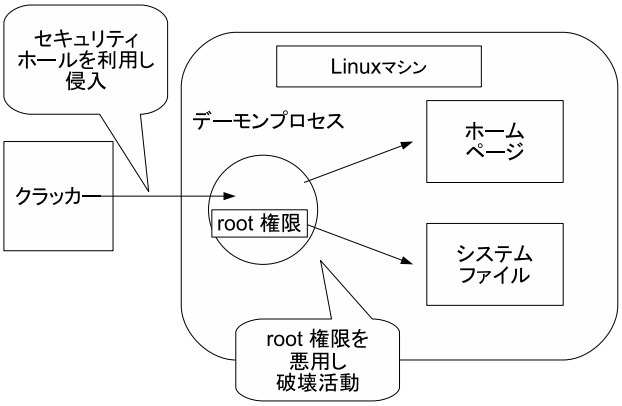

図 2.2: 直接侵入攻撃

書攻撃によって破り、侵入したものと思われる。直接狙われていたわけではなく、ほかのコンピュータへの攻撃の踏み台にする目的での侵入であった。パスワードを破ってログインし、攻撃対象のコンピュータから次の攻撃対象を探すために、ssh を外部に異常なほど投げ続けた。結局、ネットワーク上から問題のホストを切り離すことで応急処置をおこない、再インストールするまでネットワークに接続できないという対応を取った。しかし、これが企業などだった場合、ホストをネットワークから切り離してしまうことは難しい。そこで、OS の外部からでもプロセス単位、ユーザ単位で通信を制御できることが求められる。しかし、既存の技術ではそれはできていない。セキュリティ対策には 2 種類あり、OS レベルで行う方法と、ネットワークから行う方法がある。以下の節では、既存のセキュリティ技術を紹介する。

2.2　OS レベルでのセキュリティ対策

2.2.1　特徴

監視対象のサーバなどにインストールして使う。インストールされたホスト自身に対する攻撃以外は防げない。保護したいコンピュータが複数あれば、それぞれにインストールする必要がある。

ホストのイベントを監視するので、警告が発せられた場合、そのホストが影響を受けている場合が多い。つまり、「検知したイベントの信頼性が

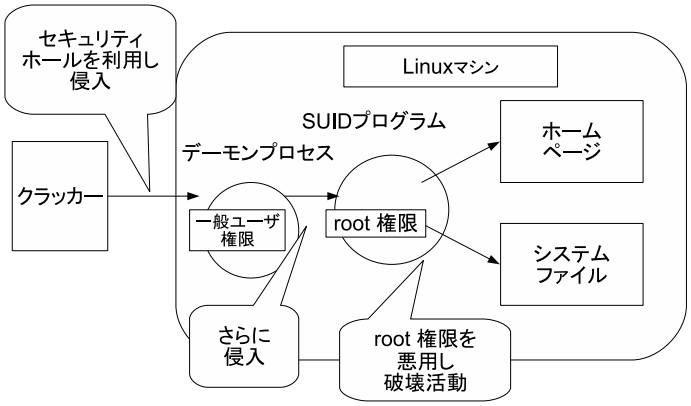

図 2.3: 間接侵入攻撃

高い」。しかし、検知したイベントの信頼性は、監視対象のホストの信頼性に比例する。つまり、対象ホストが「健全」な状態ならば検知されたイベントの信頼性は高いが、すでに侵入を許しているホストにおいては、検知されたイベントの信頼性はゼロに等しい。例えば、侵入後「rootkit」と呼ばれるツールを仕掛けられた場合、特定のイベントのみが出力されないような状態にされることもありえる。このような状態のホストに、導入したとしても検知できるイベントはまったく信用できない。OS レベルで行う場合の信頼性は対象ホストの信頼性に比例することに注意し、セキュアなホストを構築した上で導入する必要がある。

最近では、重要な情報をやり取りする際には、暗号化通信を利用するケースが増えている。暗号化された通信に含まれる疑わしいパケットについては、ネットワーク上で行う方法では検知できない場合が多い。しかし、ネットワーク上では暗号化されていても、ホスト内部では暗号化されていない状態でやり取りされるので、検知できる。

OS レベルで行う場合は、監視対象のホストごとに導入する必要があるので、対象の範囲を広げれば、当然コストも高くなる。また、バージョンアップやパッチの適用といったメンテナンスをこまめに行う必要がある。

また、監視対象となるホストの多くは、日常的にクライアントとして利用されているか、サーバとしてサービスを提供している。日常的にかかっている負荷に加えて、更なる負荷が加わることになる。その結果、通常のサービスを正常に提供できなくなる可能性がある。出力するログファイルなども要注意である。イベントが頻繁に検出されるような状況になると、

ホストの CPU 負荷が高まるとともに、ログファイルがストレージに収まらなくなり、通常のサービスを継続できなくなってしまう。
以下では、OS 内部でセキュリティ対策を行うシステムについて述べる。

2.2.2　セキュア OS

これまでの OS は以下のような弱点を抱えている。

- 任意アクセス制御 (DAC) 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すべ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。プロセスがのっとられた場合、システムに大きな被害を及ぼす

セキュア OS は上記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。MAC は root アカウントにさえも強制力を持つ。root アカウントがファイルのアクセス権を変更するには、特定の手順を用いてシステムの上体を変更してからアクセス権を変更する必要がある。ちなみに MAC には次の二つの実装方法がある。

- MLS(Multi Level Security)

  機密レベルの異なる複数のユーザが同時にシステムへアクセスできること。MLS を採用したシステムでは、各リソースに重要度に応じたラベルがつけられ、ユーザはそのラベルに応じてアクセスを許可されたデータだけにアクセスできる。

- RBAC(Role-Based Access Control)

  オブジェクトに対するアクセス権を役割 (role) に関連付け、この役割をユーザに割り当てることによって、ユーザにその役割に許可された権限を獲得させるアクセス方式。同時に最小特権の原理の包含する (最小特権については後述を参照)。

もう一つの特徴は「最小特権」である。プロセスに特権を与える際に、すべ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 と最小特権という二つの機能を満たしているものである。Linux 用の代表的なセキュア OS モジュールは、

- SELinux
- LIDS
- RSBAC

がある。

2.2.3 SELinux

SELinux は、Linux カーネルのセキュリティ強化モジュールとして提供されている。Linux カーネルに SELinux を組み込むことにより、Linux カーネルのアクセス制御機能が大幅に強化される。具体的には、次の 4 つの機能が新たに実装される。

- 機能 1：MAC…アクセス制御の徹底
- 機能 2：TE…プロセスに最小限の特権を付与
- 機能 3：ドメイン遷移…権限の付与と昇格防止
- 機能 4：RBAC…ユーザに最小の権限を付与

これらの機能により、たとえ攻撃者が SELinx を用いたシステムに侵入できたとしても、大きな被害を与えることはできない仕組みになっている。

MAC と RBAC については先に述べたとおりであるので、TE とドメイン遷移について述べる。

TE(Type Enforcement)

TE は、「ドメイン」と「タイプ」という属性情報を用いて、プロセスのアクセス制御を行う。SELinux を用いたシステムでは、すべてのプロセスにドメインと呼ぶ属性情報が与えられる。このドメインは、「プロセスに与えられる権限の名前」と考えられる。一方タイプは、システムの資源に与えられる属性情報である。例えば、ファイルやディレクトリやネットワーク・インタフェースごとに特定のタイプが与えられる。

よく利用するアプリケーションやシステム資源ようのドメインやタイプは、あらかじめ SELinux に用意されている。また、管理者が独自のいドメインやタイプを設定することもできる。

SELinux では、どのドメインがどのタイプにどのようなアクセスが可能かをあらかじめ設定しておくことにより、きめ細やかなアクセス制御が可能となる。

ドメイン遷移

上で述べたプロセスへのドメイン割り当ては、TE の機能の一部である「ドメイン遷移」により実現されている。このドメイン遷移は、あるプロセスが子プロセスを起動したときのドメイン割り当てを制御し、新規のプロセスが不必要に大きな権限を持たないようにする。

ここで、親プロセスが子プロセスを起動した場合を考える。SELinux の標準設定では、子プロセスは親プロセスのドメインを継承する。つまり、親プロセスと同じ権限を持つ。しかし、これでは不都合が生じる場合があるため、子プロセスのドメインをより権限が少ないものに変更することができるようになっている。

2.2.4　LIDS

LIDS は「設定が直感的」なことが特徴としてあげられる。LIDS の設定方法は、例えば「プログラム A がファイル B のディレクトリ C にアクセスしたときの権限は D である」という表記で、「ACL(Access Control List：アクセス制御リスト)」を記述していく方法を取っている。このため、ファイルにアクセス可能なプログラムを制限したい際に、非常に簡単に設定を行うことができる。

また、LIDS はそのシンプルさが特徴となっている反面、SELinux をはじめとしたその他のセキュア OS に比べ、セキュリティの粒度の荒い状態になっている。アクセスを制限できる対象も、今のところプロセスに対するファイル/ディレクトリへの権限操作とケーパビリティ、さらにネットワーク機能の一部の制限だけになっていて、SELinux のようにセマフォや共有メモリに対してのアクセス制御を行うことはできない。また、RBAC によるユーザ管理なども行わない。

しかし、特定の用途に限定してセキュリティの担保を担いたい場合には、充分な機能を備えており、さらに設定方法が直感的で簡単だという点は、複雑なセキュア OS のっ機能を OFF にしたがるユーザを対象として考えた場合にはメリットとなるのではないだろうか。

権限の設定方法に関しては、通常のセキュア OS では以下の二つのアプローチが考えられる。

- 最初に厳しくしておいて、必要な権限を追加する

- 最初穏やかにしておいて、できることを制限していく

SELinux などは前者を採用しているのに対して、LIDS は後者を採用している。このことから、セキュリティの本質論から言えば「甘い」セキュリティとなるが、「このフォルダだけ書き込み不可能にする」などといった、特定の用途にしようしたい場合には簡単に設定できるため便利である。

さらに LIDS には他のセキュア OS にはない機能がいくつか搭載されている。その 1 つが ACL に「ステート」という概念を導入していることである。このステート機能とは、システム全体の状態を以下の 3 つの状態にわけ、それぞれに対して ACL を設定できるというものである。

- 起動状態：ブーストから login まで
- 運用状態：login 以降
- シャットダウン状態：コマンドで状態を変更してからシステムが停止するまで

例えばシステム起動時とシャットダウン時には、全てのサービスに対して「/var/run/*.pid」というファイルに書き込めるように設定しておき、通常運用時は READONLY にしておくということが可能である。この機能を使うことで、より適切な権限を設定することが可能になる。

さらに「NF_MARK」という機能では「iptables」や「iproute2」と同じように、ソケット作成時に MARK 値を設定することができる。これを使うと、LIDS と iproute2 を用いて各プログラムごとにネットワークのルーティングを変更することが可能となる。

2.2.5　不正侵入の検知と遮断 (IDS と IPS)

ネットワークにおけるセキュリティ対策製品としてまず基本になるのはファイアウォールであるが、攻撃者はその攻撃手法を高度化させており、いまやファイアウォールだけではあらゆる攻撃を排除できないということが常識となっている。

これに対し、ファイアウォールが防げない攻撃を検知したり防御したりするものが「IDS(Instrusion Detection System：侵入検知システム)」「IPS(Instrusion Prevention System(侵入防止システム)」である。

ファイアウォールの基本は、外から来る通信の IP アドレスを見て、社内のシステムへの入り口にあたる TCP/UDP ポートの ON/OFF を行うことといえる。しかし、IP アドレスは偽装できるし、アプリケーションゲートウェイのようなファイアウォール技術でも欺くことはハッカーにとってはそう難しくない。ファイアウォールは巧妙な不正アクセスを正当

なユーザからのアクセスと区別することができず、とりあえず壁を通してしまう。そこで、あるパケットが正当なものであるかどうかを見分けるために使うのが IDS である。IDS はネットワークを流れるパケットを監視したり、サーバ上で受信データやログを調べて不正侵入を見つけ、あらかじめ設定されていたアクション (例えば警告メールを管理者へ送信) を起こす。ファイアウォールとの連携により、ポートを一時的に閉じたり、特定のセッションを切ったりすることによってその後に続く不正アクセスを防ぐこともできる。IDS には大きく分けてネットワーク型 IDS とホスト型 IDS がある。

IDS は、コンピュータネットワークの通信において不正アクセスを検知しても、あくまで通知するのみである。また、コンピュータネットワークに流れる通信内容をいったんコピーしてから解析を行うため、最初の侵入は防ぐことができず、新手の攻撃や亜種の攻撃に対して弱いという一面を持っている。このため、こうした受身的発想の IDS を発展させ、不正アクセスに対して、自動的に通信遮断やサーバのシャットダウンを行うといった能動的発想で構築されているのが IPS である。このため、IDS では、管理者が事前に不正アクセスのパターンを予測して運用する必要があったが、IPS では、管理者はアラーとやログを確認してからでも充分に対応できるようになっている。IPS にも IDS と同様にネットワーク型とホスト型の 2 種類がある。

ここでは、ホスト型について説明する。ネットワーク型についてはネットワーク上でのパケットフィルタリングのセクションで説明する。

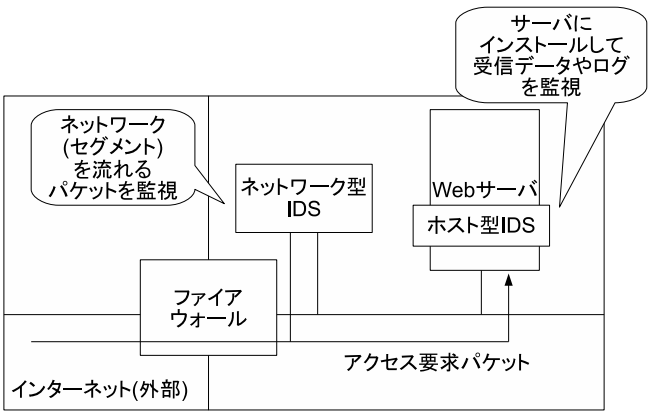

図 2.4: ネットワーク型 IDS とホスト型 IDS

2.2.6　ホスト型 IDS

ホスト型 IDS の機能

ホスト型 IDS は監視の対象とするホストに直接導入して利用する。そして、以下を監視する機能を主に備える。

- 監視対象ホストが受信したパケット
- 監視対象ホストが出力するイベントやログ
- 監視対象ホストのファイルの変更
- 監視対象ホストのシステムコール

これらを監視し、あらかじめ設定された特定の変化やイベントを検知すると、管理者へ通知といった特定の動作を行う。検知後の動作についてはネットワーク型 IDS とほとんど同じである。

ホスト型 IDS を導入するメリット

次にホスト型 IDS を導入および運用するメリットは以下である。

- 検知したイベントの信頼性が高い

  ネットワーク型 IDS は、ネットワークに流れているパケットを監視する。そして疑わしいパケットを検出した場合に警告などを発する。あくまでも疑わしいパケットを検知するだけなので、そのパケットが受信ホストにどういった影響を与えるのか、本当に悪影響を与えるのか、までを確認することができない。例えば、ネットワーク型 IDS が多数の警告を発しても、パケットを受信したホストには何の影響もないことがありうる。それに対して、ホスト型 IDS は監視対象ホストのイベントを監視するので、警告が発せられた場合には、そのホストが影響を受けている可能性は非常に高い。つまり、ネットワーク型 IDS は、影響の可能性を警告するのに対して、ホスト型 IDS は、結果を警告するので、検知したイベントの信頼性が高い。とはいえ、これをもって「ホスト型のほうがネットワーク型よりも優れている」とはいえない。「検出したイベントの信頼性が高い」ことを示すだけで、トータルの有用性についてネットワーク型が劣っていることを示しているわけではない。さらに、ホスト型 IDS で検知したイベントの信頼性は、監視対象ホストの信頼性に比例する。対象ホストが健全な状態では、確かに検出されたイベントの信頼性は高いが、IDS 導入前すでに侵入を許しているホストにおいては、検知されたイベントの信頼性はゼロに等しい。

- 暗号化通信へ対応できる

  最近では、重要な情報をやり取りする際には SSL や SSH、VPN といった暗号化通信を利用するケースが増えている。暗号化された通信に含まれる疑わしいパケットについては、ネットワーク型 IDS では検知できない場合が多い。しかし、ネットワーク上では暗号化されていても、ホスト内部では暗号化されていない状態でデータやイベントがやり取りされるので、ホスト型は検知できる。不正アクセス以外のホストの異常を検知できることもホスト型 IDS のメリットである。具体的には、オペレーション・ミスや特定のアプリケーションエラーなどを検知できる。ホスト型 IDS の設定によっては、ユーザが誤ってファイルを消した場合や、設定を変更した場合にも検知し、復旧することが可能である。

- 不正アクセス以外のイベントも検知できる

  具体的には、オペレーションミスや特定のアプリケーションのエラーなどを検知できる。ホスト型 IDS の設定によっては、ユーザが誤ってファイルを消した場合や、設定を変更した場合にも検出し、復旧することが可能である。

ホスト型 IDS の不正アクセス検知方法

- ログのモニタリング

  特定の OS のログ出力を監視し、異常なイベントを検出する。例えば認証の失敗が繰り返された場合に警告をコンソールに表示するといったものである。OS ばかりでなく、Apache や DNS などの主なアプリケーションについても同じように同じようにログ監視を行う。

- ファイルの改ざんチェック

  ファイルの属性を記録しておき、正当に更新されていないのに属性が変わったものを発見する。属性とは、グループ、オーナー等に加え、ファイルサイズをハッシュしたデータを用いてチェックできるようになっている場合がある。

- サーバへの不正なパケット検知と遮断

  ネットワーク型 IDS のインライン配置と同様に、ローカルなファイアウォールとして機能することができる。特定のパケットを拒否したり、特定のイベントに反応してブロックしたりして、アクセスを遮断できる。

2.2.7　ホスト型 IPS

ホスト型 IPS は、ソフトウェアの形で提供され、サーバマシンにインストールする。バッファオーバーフローを利用した不正アクセスを OS レベルで防いだり、一般ユーザによる管理者権限の取得ができないようにしたり、アクセスログの改ざんを防止するなどの機能を持っている。OS が提供するよりも細かいレベルでのアクセス制御を実現したり、サーバ管理者とセキュリティ管理者を分離して万能の権限を持った管理者を存在させないことで侵入された際の被害の拡大を抑える機能を持ったものもある。

2.3　ネットワーク上でのセキュリティ対策

2.3.1　特徴

ホスト以外の機器に接続し、接続されているネットワークセグメントの全てのトラフィックを監視する。ネットワーク上のパケットをリアルタイムで監視する。すでに知られているパターンなどをネットワーク上に流れる情報から検知する。つまり、既知のパターンの攻撃や登録されているパターンの攻撃しか検知することができず、未知の攻撃には無防備である。ネットワーク上の全ての通信内容を検査するので、不正アクセスの疑いがあると思われるものは全て検知できるが、疑いがあるもの全てを検知してしまい、実際には不正アクセスではないものである場合もある。しかし、ネットワークセグメントごとに設置すればよいので導入はホスト型に比べて簡単である。

2.3.2　ファイアウォール

ファイアウォールを使うと、ポート番号や IP アドレスを基準に、不要な通信を通さないようにできる。しかし、ファイアウォールの通過する通信を利用した攻撃を防ぐことはできない。例えば、HTTP の通信を使って web サーバのセキュリティホールを攻撃するようなものは防げない。ポート番号や IP アドレスだけでなく、通信の中身を見て、通信の制限をするアプリケーションゲートウェイと呼ばれるタイプのファイアウォールも存在するが、あらかじめ登録しておいた攻撃しか防ぐことはできず、また未知の攻撃には対応することも難しい。

2.3.3　ネットワーク型 IDS

ネットワーク型 IDS の特徴

監視対象となるネットワーク (セグメント) に設置して使用する。ネットワーク (セグメント) を流れる通信パケットを監視することによって不正侵入や攻撃がないかをチェックする。ただし、ネットワークトラフィックが増加すると、ネットワーク型 IDS はパケットを取りこぼしてしまうこともある。従って、DoS 攻撃に弱い体質を持っている。最悪の場合は、DoS 攻撃によって IDS がダウンしてしまうこともありえる。

ネットワーク型 IDS の特徴としては以下があげられる。

- ネットワーク上に流れるパケットを収集して、そのプロトコルヘッダやデータを解析する。
- 暗号化された通信では、複合された情報が不正ものであるかどうかの判断ができない。
- 監視対象となるホストには余計な設定を施さないので、ホストに負荷をかけない。

ネットワーク型 IDS は、攻撃のパターンをシグネチャベースやアノマリベースで検知したり、連続するパケットを (フラグメント化された) 組み立てなおして解析し攻撃の有無を判断する。監視したパケットがシグネチャに一致したり、通常ではありえない以上な動作を監視したりするために、スイッチのポートミラーにネットワーク型 IDS を接続する。

ネットワーク型 IDS の不正アクセス検知方法

- シグネチャ検知 (Signature detection)

  パケットをチェックし、不正アクセスに使われる特徴的な文字列が含まれているかを検査するもので、この特徴的な文字パターンを「シグネチャ」といい、シグネチャに一致した場合不正アクセスとみなす。シグネチャは、ウイルス対策ソフトで使うウイルスパターンと同様なものと考えてよい。IDS はパケットをリアルタイムで捕らえて、登録されているシグネチャとパターンマッチングを行い、不正とみなされるパケットを検出する。したがって常に最新のシグネチャを登録しておく必要があり、充分な数のシグネチャがなければならない。また、パケットの取りこぼしがあると、不正アクセスを見落とす可能性がある。最新のネットワークの高速化に伴い、高速なマッチング性能が求められる。

- アノマリ検知 (Anomaly detection；異常検知)

  通常のシステムではありえない行動を検出するもので、検知対象として以下のような異常行動がある。例えば、ping コマンドを使ってネットワーク機器の状態を確認することは通常の管理行動の一つだが、コマンドを送出するのはせいぜい 1 秒に 1 回とか 1 分に 20 回といった頻度であろう。それが 1 秒間に 100 回送られたとすれば、それは異常行動といえる。目的のはっきりしない管理コマンドやメールの送付件数が異常に多くなったなどの状況が捉えられれば不正アクセスの発見につながる。これには、正常な状態を登録している必要がある。

2.3.4　ネットワーク型 IPS

ネットワーク型 IPS は、専用のアプライアンス (装置) として提供され、ネットワークの境界に設置する。コンピュータウイルスや DoS 攻撃などパケットが持つ特徴的なパターンがあらかじめ記憶されていて、侵入検知時には通信の遮断などの防御をリアルタイムに行い、管理者への通知やログ記録の機能を持つ。

2.4　仮想マシンにおけるサーバ管理

2.4.1　仮想マシンの復活

仮想マシン環境は、古くて新しい技術である。仮想マシン環境の歴史は古く、大型汎用機用 OS が現れた直後にはすでに仮想マシン環境が使われていた。仮想マシン環境とは、ハードウェアを仮想化し、その仮想ハードウェア上で OS を動作させる技術である。仮想的なハードウェアを複数容易することで、同時に複数の OS を実行することも可能である。

大型汎用機時代に仮想マシン環境が重宝された最大の動機は、過去の資産の継承であった。汎用機の置き換えを行った場合、通常 2 つの汎用機の間には互換性はなく、古い汎用機用に作られた高価なアプリケーションプログラムは、新しい汎用機では動作しない。この課題を解決するために、仮想マシン環境が普及した。仮想マシン環境が古い汎用機のハードウェアをエミュレートすることで、アプリケーションプログラムをその仮想マシン環境の中で使い続けることができた。

最近、この古い技術がまた脚光を浴びるようになってきた。今日の仮想マシン環境を導入しようというモチベーションは、汎用機時代の最大の動機である「資産継承」よりも、マシン代数の削減に目が向いている。イン

ターネット系サービスを中心に、そのサービス成長に伴って規模がどんどん大きくなり、ハードウェアおよび運用コストが増大傾向にある。これらのコストを下げるための 1 つの手段として、仮想マシン環境の利用が検討されている。

仮想マシン環境を作り出す仕組みはいくつか存在する。1 つは、ある OS(ホスト OS と呼ぶ) の上に、ハードウェアをエミュレートする環境をのせ、その上で別の OS(ゲスト OS と呼ぶ) を動かす方法である。エミュレートはホスト OS が持つ機能を利用して実現する (図 2.5)。

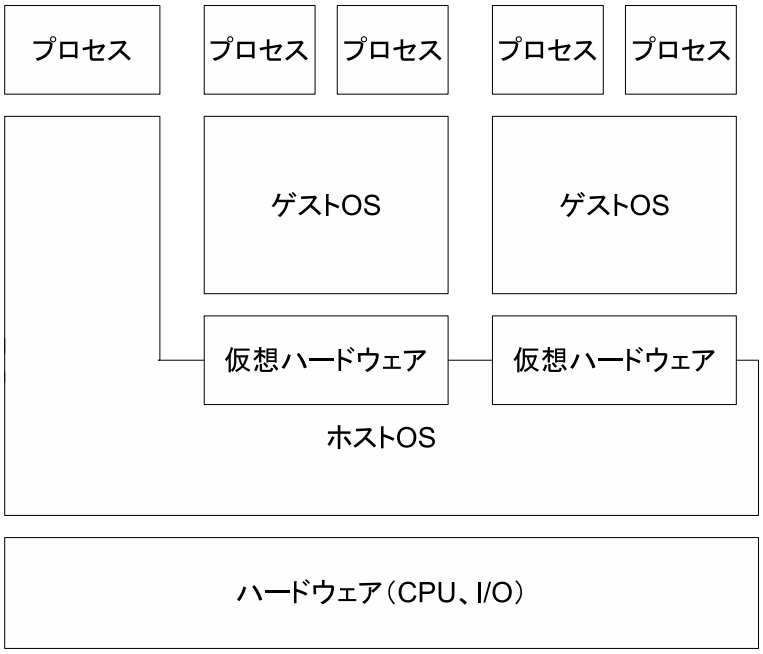

図 2.5: OS 上でのエミュレートによる仮想マシン環境

2 つ目は、ハードウェア上に、直接ハードウェアをエミュレートする専用のプログラムを動作させる方法である (図 2.6)。仮想マシン環境を実現するために必要最小限の機能を、OS と実ハードウェアの間に挟む。このプログラムを仮想マシンモニタと呼ぶ。この方式は、1 つ目の方式よりも小さなオーバーヘッドで仮想マシン環境を実現できる。

本研究では、仮想マシンモニタを使った仮想マシン環境の方に注目するので、以下で仮想マシンと仮想マシンモニタについて説明する。

仮想マシンと仮想マシンモニタ

仮想マシン

仮想マシンの目的は、物理命令セットアーキテクチャと論理命令セットアーキテクチャの分離であり、一度コンパイルされたプログラムは物理命令セットアーキテクチャが何であれ、仮想マシンによって実行することができる。また、論理命令セットアーキテクチャを固定することで、コンパイラに対して複数の物理命令セットアーキテクチャに対応したコードジェネレータをその都度開発しなくてすむ。また、非常に複雑で多岐にわたる物理命令セットではなく、その言語や環境にあわせて洗練された命令セットに対応したコードジェネレータを一回だけ設計すればよく、コンパイラ開発における負担を大いに軽減する。

仮想マシンモニタ

一つの計算機資源を複数の計算機資源としてハードウェア的に利用したり、物理的資源を区分けして特定のワークロードがシステム全体を独占させない目的 (パーティション) のために作られたものである。一台のコンピュータを複数のコンピュータとして、ハードウェアレベルで使うことができる。一台のコンピュータ上で特にオペレーティングシステムの変更なしに、複数のゲストオペレーティングシステムを稼動させたりする。稼動させる複数のゲストオペレーティングシステムは、まったく別のオペレーティングシステムであることも可能である。

元々仮想マシンモニタはメインフレームにおいてその資源を各ワークロードの負荷に応じて動的に分割し、システム全体の効率を上げるために用いられた。計算機が高性能化した現在では、小規模な Web サイトホスティングのような軽い処理を一般的なサーバでも仮想マシンモニタを用いて並列に実行し、資源の有効活用することが行われている。

仮想マシンモニタと普通の OS を比較してみる。

- OS：アプリケーションからハードウェアを隠蔽するものとして生まれた。マルチタスクをサポートするようになり、複数のアプリケーションを OS 上で実行できるようになった。アプリケーションは、タスクやプロセスという環境を与えられ、その中で動作する。
- 仮想マシンモニタ：OS からハードウェアを隠蔽するものとして生まれた。複数の OS を、仮想マシンモニタ上で同時に実行できる。OS は仮想マシンモニタから与えられた仮想マシン環境のなかで動作する。

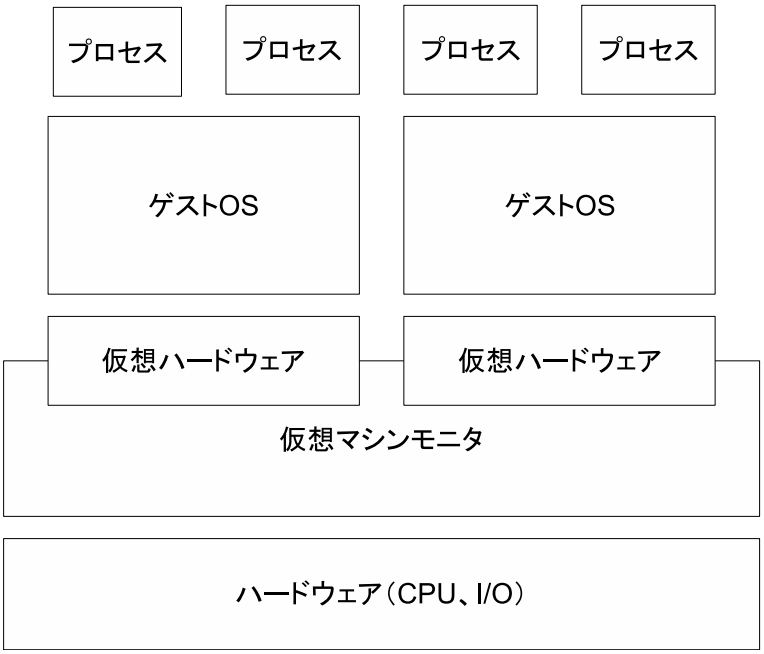

図 2.6: 仮想マシンモニタによる仮想マシン環境

仮想マシンモニタは、OS に対して CPU 時間やメモリなどの資源を割り当てる。一方、OS はその上で動作しているアプリケーションプログラムに対して資源を割り当てる。つまり、「仮想マシンモニタはゲスト OS をスケジューリングする OS」と考えられる。

Xen について

ここでは、仮想マシンモニタ「Xen [1]」を取り上げ、その特徴を見ていく。

Xen は、既存の OS の上でほかの OS を動かすのではなく、複数の OS を動かすための基盤となるプラットフォームだけを提供する。つまり、実際の OS はすべて Xen 上で対等に動作することになる。

Xen の大きな特徴は 2 つある。1 つは仮想マシン環境構築に、準仮想化という手法を採用している点。もう 1 つは、Xen 自体がデバイスドライバを持たないという点である。

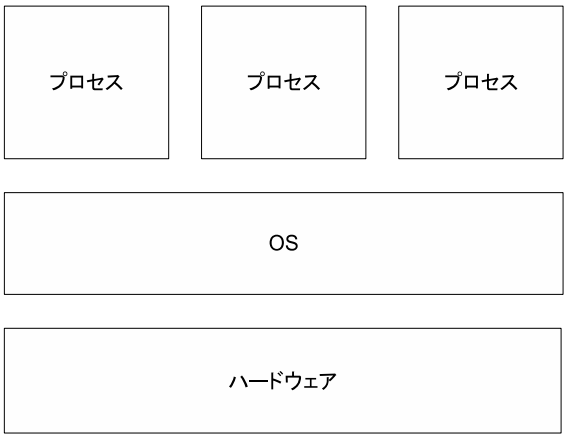

図 2.7: OS とタスク

準仮想化と完全仮想化

仮想化には準仮想化と完全仮想化の二種類がある。ここでは、それぞれについて特徴を説明する。

- 準仮想化

  Xen は準仮想化と呼ばれる実装手法を標準採用している。実在のハードウェアを完全にエミュレートする代わりに、仮想マシン環境を実現するのに都合の良い仮想的なハードウェアを再定義する。この仮想ハードウェアは、実在のハードウェアに似ているが、操作をするためにはハイパーバイザコール[2]を呼び出す必要がある。Xen はこのハイパーバイザコールの要求に応じて、仮想マシン環境に変更を加える。

  この準仮想化という手法は、Xen の上で動作させるゲスト OS に手を加えることを要求する。つまり、ゲスト OS を Xen 仮想ハードウェア上に移植する必要がある。この方法はあらゆる OS に適用できるわけではないためデメリットといえる。しかし、大きなメリットもある。エミュレーションのオーバーヘッドを最小限に抑えることができるため、性能面で大きなアドバンテージを得られる。

  ハードウェアを完全にエミュレートするには、OS からハードウェアや CPU に対し発行された要求を分析し、本来やりたかったことを

[2]プロセスが Linux に要求を出すときに呼び出すシステムコールのようなもの。

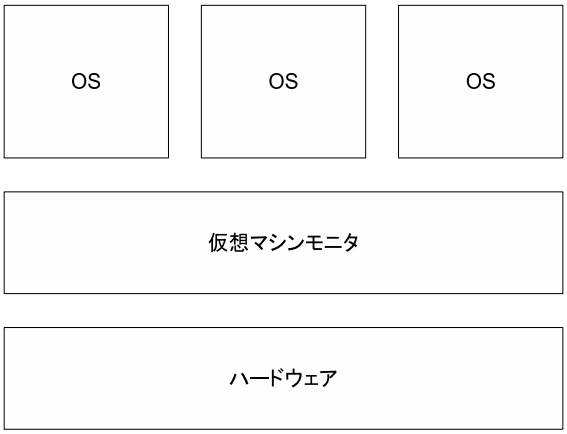

図 2.8: 仮想マシンモニタと OS

推測し、実際の処理を実行する必要がある。一方準仮想化の環境では、ゲスト OS は Xen に対して本来やりたいことを直接要求できるため、処理を単純化できる。

- 完全仮想化

  Xen はハードウェアの完全仮想化機能も提供している。この機能を利用すると、実ハードウェアように容易された OS をそのまま Xen 上で動作させることが可能となる。ただし、完全仮想化機能を利用するためには、VT 対応の IntelCPU 上で Xen を動かす必要がある。

  この完全仮想化機能が提供する仮想マシン環境内の OS は、自分では特権モードで動作していて完全に物理ハードウェアを支配しているを思っている。しかし、実際には特殊なモードの中で OS は動作させられている。

  OS が仮想ハードウェアを制御する命令を実行したとき、CPU はそれを検出し、例外のようなものが発生して Xen に制御を渡す。制御を渡された Xen は、OS が行おうとした処理を分析し、仮想ハードウェアの動作をエミュレートする。

  完全仮想化の環境は、準仮想化と比べると、エミュレーションのためのコストが大きくなる。しかし、ソースコードを手に入れることのできない Windows 等の OS も動かせることは、大きなメリットである。

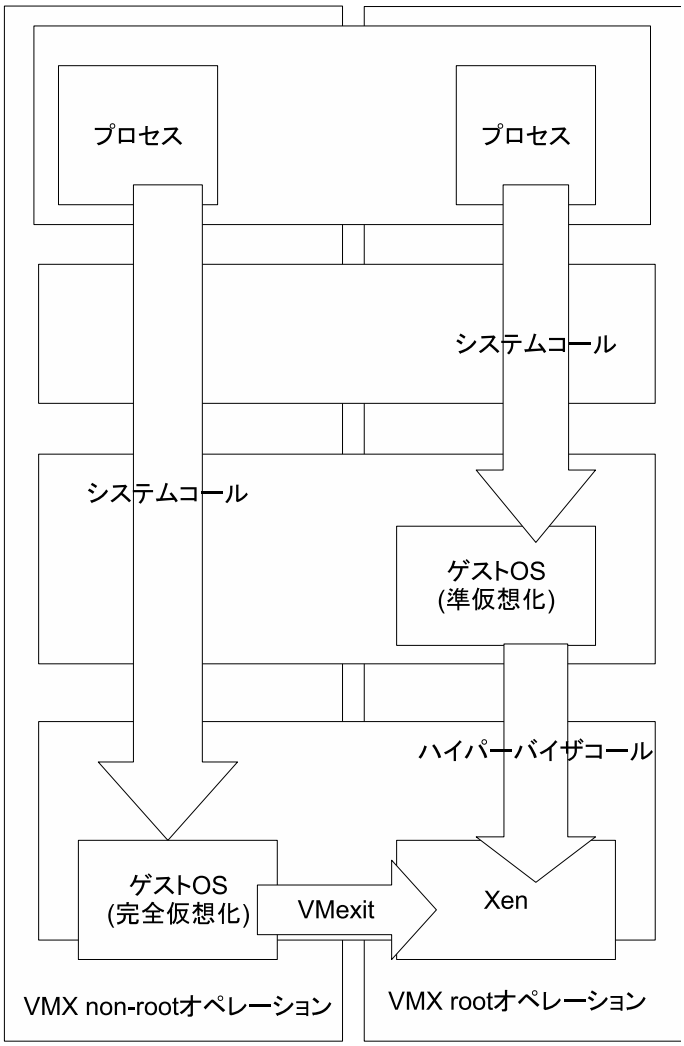

図 2.9: 完全仮想化と準仮想化

デバイスドライバのモデル

仮想マシンモニタには、CPU だけでなく I/O デバイスも仮想化する必要がある。Xen の仕組みでは、I/O デバイスの仮想化に当たって、実際のデバイスでなく、Xen 環境専用の仮想的なデバイスを定義している。ドメイン上のゲスト OS は、この仮想化デバイスに対して I/O アクセス要求を行う。この要求は、実デバイスへの I/O アクセス要求に変換しなければならない。

ところが Xen は自分自身では実デバイスを管理しておらず、ドメイン 0 と呼ばれる特別なドメインに実デバイスの制御を任せている。ゲスト OS が仮想デバイスに I/O アクセスを要求すると、そのままドメイン 0 に転送し、ドメイン 0 が代わりに実デバイスを操作するという仕組みになっている。この実装方法なら、Xen 本体の構造を単純化することに加え、新しい各種デバイスへの追従は Linux カーネルに任せておけばよい。つまり、Linux が動作するハードウェアなら、どこでも Xen を動作させることができる。

Xen では、ネットワークインタフェースも仮想化されている。ドメイン間に仮想的なネットワークを立ち上げており、Linux カーネルが用意しているブリッジ機能を利用して、物理ネットワークインタフェースの先に、仮想ネットワークインタフェースをつなぐ。ドメイン U が送信したパケットは、ドメイン 0 の物理ネットワークインタフェースを通して外の世界に送り出される (受信はその逆)。

それ以外の特徴

Xen が提供する仮想マシン環境はマルチプロセッサに対応しているので、仮想マシン環境 (ドメイン) の中で、マルチプロセッサ対応の OS を動かすことができる。また、各仮想マシン環境 (ドメイン) に対する資源割り当て量を動的に変更できる (現在は CPU 時間とメモリ量のみ制御できる)。

このほか、物理マシンの間で仮想マシン環境 (ドメイン) を移動できる。ドメイン内で動作している OS は、移動したことにほとんど気づかないだろう。この機能を利用すると、物理マシン間にまたがった負担分散や、OS を無停止のまま物理マシンを交換したりすることが可能となる。

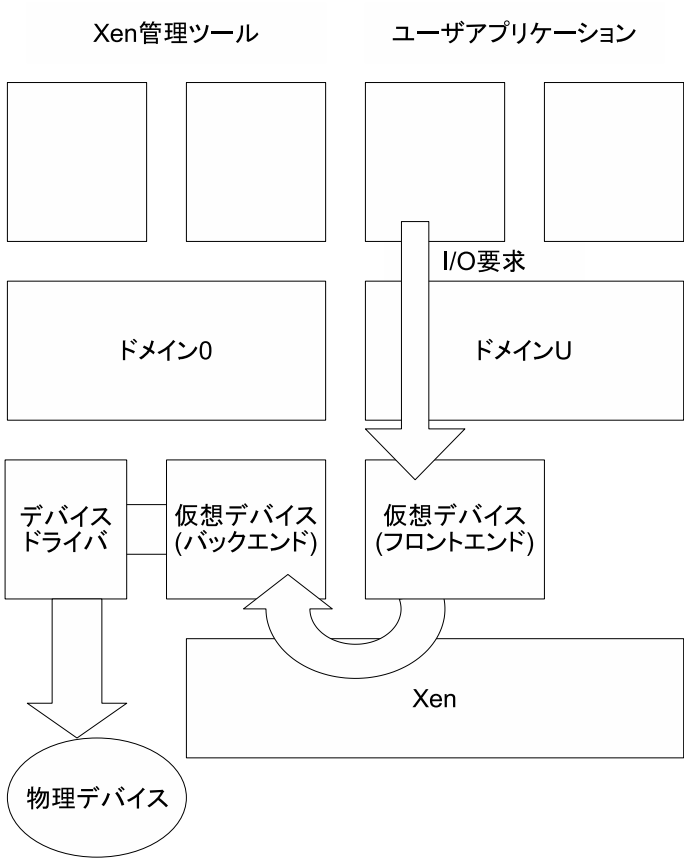
Xen管理ツール
ユーザアプリケーション
I/O要求
ドメイン0
ドメインU
デバイス
ドライバ
仮想デバイス
(バックエンド)
仮想デバイス
(フロントエンド)
Xen
物理デバイス

図 2.10: Xen の構造

2.4.2　仮想マシンモニタによる理想のサーバ管理

仮想マシンモニタ管理者とゲスト OS の管理者

仮想マシンモニタはデータセンターに設置されていて、各ゲスト OS はデータセンターに設置を依頼している状況を考える。ここで、各ゲスト OS の管理は企業の人がしていて、場合によっては本業の片手間に管理しているかもしれない。このような場合、ゲスト OS の管理者全てが OS の管理者として正しく管理できるとは限らない。本業ではないため、サーバ管理に慣れていない人もいるだろう。また、知識があり、サーバ管理に慣れている人でも、フルタイムで管理することは難しいだろう。そこで、知識がある人が常時仮想マシンモニタの管理者を行うのが望ましい。そして、問題が起きたときには仮想マシンモニタ管理者が即座に応急処置をして、OS の管理者がきちんとした対応をするまで外部への不正な攻撃を防ぐ必要がある。その際、問題のある OS ごと切り離してしまうのは簡単だが、OS のホスティングを依頼していることを考えるとできるだけ制限を最小限にするのが好ましい。つまり、脆弱性のあるプログラムの通信のみ遮断できればよい。

常時管理する利点

現在、さまざまなセキュリティ対策がとられているが、OS に対する全ての攻撃を完全に排除することは難しい。先に述べたセキュリティ対策のシステムも完璧ではなく、また攻撃者もその手法を高度化させている。しかも、攻撃によっては早急に対処しなければ、どんどん被害が拡大することもありえるだろう。つまり、早急に対処できる知識のある人が常時管理することで、攻撃を受けたとき被害を最小限に抑えることができる。

仮想マシンモニタ管理者が行う通信制御

既存の方法では、ゲスト OS 内部から制御する方法と、ネットワーク上で仮想マシンモニタから制御する方法が考えられるが、どちらもうまくいかない。ゲスト OS 内部から制御する場合は、プログラム単位、ユーザ単位の細かな制御ができる。しかし、仮想マシンモニタの管理者にゲスト OS のアクセス権がない可能性がある。アクセス権がない場合は当然設定は行えない。仮想マシンモニタから制御する場合は、仮想マシンモニタの管理者の権限で全て制御できる。しかし、OS 内部の情報が使えないため、OS 単位の大雑把な制御しかできない。

つまり、仮想マシンモニタ管理者が通信制御を行う場合、ネットワーク上で通信制御すれば常に制御が行える。。しかし仮想マシンモニタから通信制御する場合、OS 内部の情報は使えないため、OS 内部で行うような細かな制御はできない。

第3章　xFilter

既存のシステムでは、仮想マシンモニタ管理者はプロセス単位やユーザ単位での通信制御を行うにはゲスト OS にログインしなければならない。なぜなら仮想マシンモニタ管理者はゲスト OS 内の情報を得ることができないからである。しかし、仮想マシンモニタ管理者がゲスト OS へのアクセス権があるとは限らない。アクセス権がない場合は、仮想マシンモニタ管理者は仮想マシンモニタから通信制御を行うしかない。仮想マシンモニタから行う通信制御はゲスト OS 内の情報を得ることができないため、ゲスト OS 単位でしか行うことができない。仮想マシン単位で遮断してしまうとそのゲスト OS からのすべての通信を遮断してしまい、無関係なプロセスの通信までも遮断してしまう。

そこで、仮想マシンモニタの管理者がアクセス権がなくとも OS の外からゲスト OS 内部の情報を得ることができれば理想とする管理を行うことができる。これを実現するため、仮想マシンモニタから OS 内部の情報を得て、プロセス単位やユーザ単位の通信制御を行えるシステムとして、xFilter の設計と実装を行った。以下で、xFilter について述べる。

3.1　xFilter の特徴

先にも述べているとおり、xFilter の特徴はゲスト OS の外からゲスト OS 内部の情報を得ることができる点である。プロセス単位、ユーザ単位の制御を行うために、ゲスト OS 内のプロセス ID およびそのユーザのユーザ ID を見ることができる。さらに、通信制御を行うために開いているポート番号、IP アドレスも表示する。

また、そうして得た情報を使ってフィルタリングのルールを自動で生成することもできる。当然、プロセス ID やユーザ ID を指定してパケットのフィルタリングを行う。

3.1.1　xFilter の文法・使用例

まず、一つ目の特徴であるゲスト OS 内部の情報を得る場合、どのドメインの情報を得るのかを指定すればよい。例えば、ドメイン ID が 1 のド

メインのプロセス情報を表示したいときは、以下のように入力する。

- ./xFilter 1

実行結果は図 3.1 のように表示される。

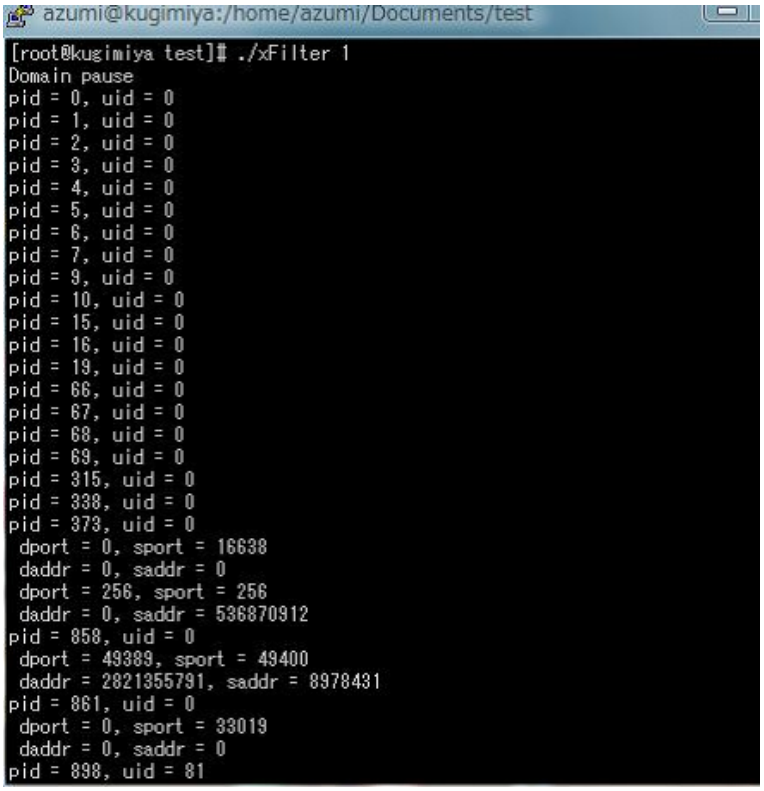

図 3.1: ドメインのプロセス情報の表示

さらにパケットのフィルタリングを行うには、プロセス単位のフィルタリングかユーザ単位のフィルタリングかを選ぶことができ、プロセス ID またはユーザ ID を指定することができる。例えば、ドメイン ID が 1 のドメインでプロセス ID が 900 のプロセスの通信を遮断したいなら、以下のように入力する。

- ./xFilter 1 pid 900

実行結果は図 3.2 のように表示される。

また、ドメイン ID が 1 のドメインでユーザ ID が 0 のプロセスの通信を遮断したいなら、以下のように入力すればよい。

- ./xFilter 1 uid 0

実行結果は図 3.3 のように表示される。

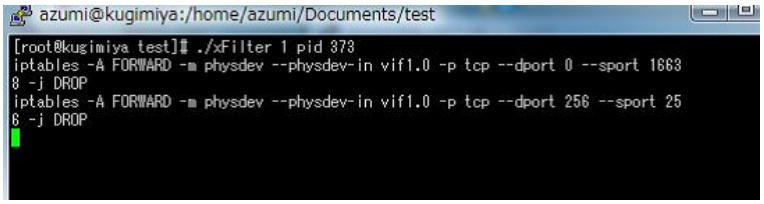

図 3.2: プロセス ID を指定してフィルタリング

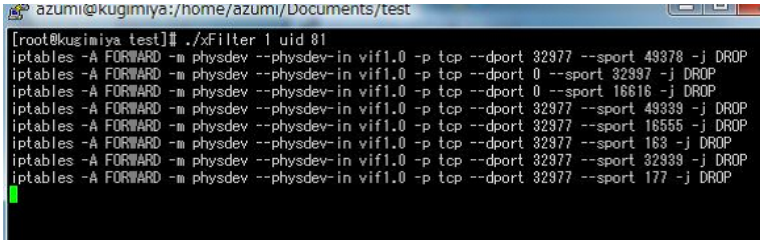

図 3.3: ユーザ ID を指定してフィルタリング

3.1.2　システムの構成

xFilter のシステムは、大きく二つの部分に分けられる。仮想マシンモニタからゲスト OS にログインせずにゲスト OS の情報を得るためのシステムと、その情報を用いて通信制御を行うシステムの二つである。

- ゲスト OS の情報を得るシステム

  ドメイン ID を指定することで、そのドメインのプロセス情報を表示できる。指定したドメインのすべてのプロセスのプロセス ID と、そのプロセスが現在開いてるポート番号、IP アドレスを表示する。この機能を使ってまずフィルタリングを行いたいドメインのプロセス情報を見て、さらにどのプロセスのパケットを遮断するかを決める。

- 通信制御を行うシステム

  プロセス ID かユーザ ID を指定して通信を遮断することができる。プロセス ID を指定した場合は、そのプロセスが開いているポートの通信をすべて遮断する。ユーザ ID を指定した場合は、そのユーザの実行しているプロセスが開いているポートの通信をすべて遮断する。

3.2　DomU 上のプロセス情報の取得

以下では、まずドメイン U のメモリを参照してゲスト OS 内の情報を取得する方法、次にメモリアクセスのロックについて、最後にポート番号を取得する方法について述べる。

3.2.1　DomU の OS 内の情報の解析

ゲスト OS が物理アドレスだと思って扱っているものは、本当の物理アドレスではない。物理アドレスは Xen によって仮想化されており、本当の物理アドレスは、マシンアドレスと呼ばれている。Xen はマシンアドレスで管理されるメモリ領域の一部を、あるドメイン空間の物理アドレスにマップし、そのドメインに貸し出している、Xen の環境では、同じ物理アドレスを持つメモリが複数存在することになるので、多重物理空間と呼べるだろう。Xen はそれぞれのメモリにドメイン ID をつけ、どのドメインの物理アドレスであるかを管理する。実 CPU がメモリを操作するとき、また実デバイスがメモリを操作するときは、マシンアドレスを使わなければならない。

また、Xen はメモリをページサイズ毎に管理している。このページサイズ毎のまとまりをマシンフレーム (machine frame) と呼ぶ。メモリをページサイズごとに区切った、各マシンフレームに machine frame number(mfn) と呼ばれる 0 から始まる連続した番号がついている。ドメインに割り当てられたマシンフレームの machine frame number は連続しているとは限らない。ドメインに割り当てられたマシンフレームを仮想物理フレーム (pseudo physical frame) と呼ぶ。ドメインに割り当てられたマシンフレームに、physical frame number(pfn) と呼ばれる番号がついている。physical frame number は machine frame number の順序に従って順番に番号がついている。ドメイン上のオペレーティングシステムは、この仮想物理フレームを physical frame number の順序に連続した本物のメモリだと思って管理している。

Xen はドメイン 0 のプロセスから呼び、ドメインのメモリを操作することができる関数を提供している。それらはすべて xebctrl.h で宣言されている。その関数を使って、プロセスの仮想アドレスから、そのアドレスが存在する mfn を求め、ドメイン 0 のプロセスのアドレス空間にマシンフレームを割り当てる。こうしてメモリをマッピングすることで、ドメイン 0 のプロセスからドメイン U のメモリにアクセスできるようになる。ここで注意しなければならないのは、マップするサイズはページサイズ分でしかなく、ドメイン U のすべてのメモリを参照できるわけではない。つまり、参照した情報にポインタがあったとする。このポインタがさしている

アドレスは、ドメイン U 上の物理アドレスであり、単純にそのアドレスをみても、それはドメイン 0 上の物理アドレスであるため、まったく別のものである。ポインタの指す先の情報を参照するときは、ポインタの指しているアドレスをさらにドメイン 0 にマッピングしなければならない。

この部分の実装は、田所さん (東工大　千葉研究室) のシステムで使っているものを使った。

3.2.2　メモリアクセスのロック

マルチプロセッサ環境では、メモリ上に配置された資源は同時に複数の CPU からアクセスされる可能性がある。この資源を保護するための仕組みとして、スピンロックなどの排他機構が用意されている。

スピンロック

スピンロックでは、メモリ上にロック変数を配置し、この変数を早い者勝ちで各 CPU が奪い合う。このロック変数で守られた資源の操作を許される。ロック変数を取得できなかった CPU は、ロックが解除されるまでその場でループして待ち続ける (これをビジーウェイトと呼ぶ)。

読み書きスピンロック

通常のスピンロックでは、排他区間を実行できる CPU は 1 つだけであった。つまり、ある資源を操作する処理は、同時に 1 つだけしか実行できない。アクセス競合の発生が少ない場合には、この通常のスピンロックは効率的に動作する。しかし、アクセス競合が多く発生する資源に対しては、ビジーウェイトによるオーバーヘッドが無視できなくなる。

このようなアクセス競合が発生する資源のうち、ほとんどのアクセスが参照アクセスであるような資源を保護する目的で、Linux カーネルは読み書きスピンロックを提供している。読み書きスピンロックは以下のような特徴を持っている。

- 参照アクセス目的のロックは多重に行える。複数の CPU が資源を同時に参照できる。
- すでに参照アクセス目的でロックされている場合、更新アクセス目的のロックを行おうとした CPU は、ビジーウェイトで待たされる。

表 3.1: 読み書きスピンロック

	更新アクセス	参照アクセス
更新アクセス目的ロック	アクセス不可	アクセス不可
参照アクセス目的ロック	アクセス不可	アクセス可能

- すでにある CPU が更新アクセス目的でロックしている場合、ロック要求を出したほかの CPU はすべてビジーウェイトで待たされる。別の更新アクセス要求を出した CPU、参照アクセス要求を出した CPU とも、現在の更新アクセスが完了するまでロックできない。

メモリのロックは読み書きスピンロックで行われている。

外からメモリにアクセスするためには、カーネルがメモリ情報を更新していないときに操作する必要がある。カーネルがメモリ情報を更新している間は、正しい情報になっていない。正しくない情報を取得しても意味がない。

本研究では、外部からメモリにアクセスするときに、カーネルがメモリ情報を更新中でないことを保証するために、外部からメモリの読み書きスピンロックを調べ、更新アクセス目的ロックが取られていたらカーネルがメモリ情報を更新中であると判断する。

3.2.3　ポート番号取得の方法

Linux は実行する基本的な単位をプロセスとして管理しており、それらに関する情報は task_struct 構造体に格納されている。プロセスは、プログラムを動作させるためのさまざまな情報を持っている。カーネルがプロセスを識別するためのプロセス ID、プロセス自身の状態、使用しているメモリやファイル、ソケット、シグナルなどである。task_struct 構造体はこれらの情報を全て一括して管理している。

task_struct 構造体には tasks という名前で list_head 構造体をメンバに持っている。list_head 構造体はメンバに list_head 構造体へのポインタを二つ持っていて、それぞれ next、prev という名前である。この tasks の持つ next はプロセス ID の順で次にくるプロセス (最後までいったら ID は 0) の tasks のアドレスを指している。つまり、この next の指すアドレスをたどっていけば全てのプロセスの情報を得られる。今回はプロセス ID が 0 の task_struct である、init_task を最初に使う。

task_struct 構造体はファイルディスクリプタを管理する files_struct 構造体を持っている (図 3.4)。オープンしているファイルは file 構造体によって管理されている (図 3.5)。files_struct には file 構造体の配列へのポインタを持つ fdtable 構造体を持っている。よって、files_struct 構造体から fdtable 構造体を経由してオープンしているファイルを管理している file 構造体の情報を得る。

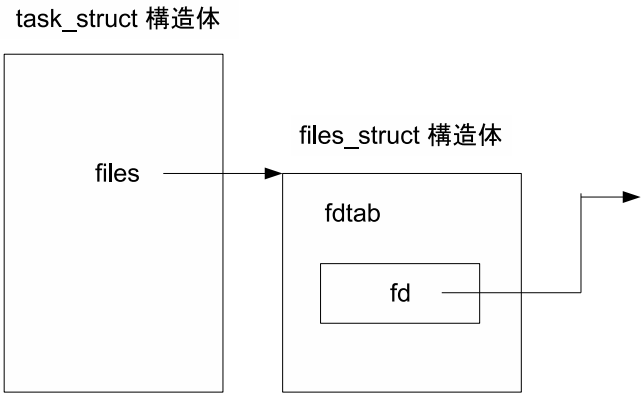

図 3.4: プロセス情報管理

ファイルの実態は inode 構造体に格納されている。inode 構造体は、各ファイルシステムに共通の管理情報を持っている。file 構造体は dentry 構造体を経由して inode 構造体をポイントしている (図 3.6)。

ソケットの実態は socket 構造体と sock 構造体である。socket 構造体はプロトコルにあまり依存しない処理を行い、sock 構造体はよりプロトコルに依存した処理を行う。このように分離されているのは、

- sock 構造体が大きいので sock 構造体を小さく保つため。
- socket 構造体と sock 構造体を切り離して管理するプロトコル (TCP) があるため。

という二つの理由による。

ところで、今調べているプロセスがソケットであるとは限らない。そこで、inode 構造体の i_mode というメンバを見てファイルのモードを調べる。ファイルのモードがソケットだった場合、socket 構造体を取ってくる。実は、inode 構造体と socket 構造体はこの二つをメンバに持つ socket_alloc 構造体によって管理されている (図 3.7)。つまり、inode 構造体の先頭アド

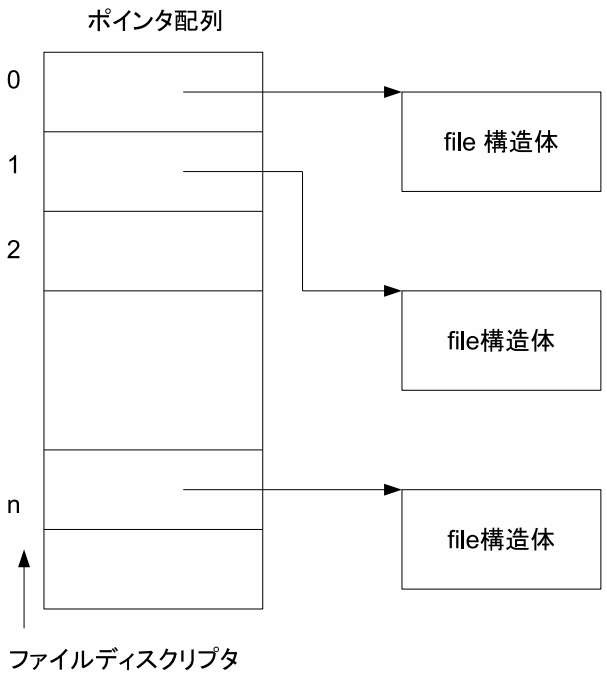

図 3.5: ファイル情報管理

レスから socket 構造体の大きさ分引いた値がそのファイルに関する socket 構造体の先頭アドレスとなっている。

各ソケット固有の情報は sock 構造体に保持されているが、このほかに IP/TCP/UDP 固有の情報も保持している。これらは、inet_sock、inet_connecting_sock、udp_sock、tcp_sock 構造体に保持されている。今回は inet_sock 構造体の情報を使う。sock 構造体から inet_sock 構造体を得るには、単純にキャストするだけでよい。inet_sock 構造体には現在開いているポートや IP アドレス等の情報が入っている。今回必要なのは、送信先のポート、送信元のポート、送信先の IP アドレス、送信元の IP アドレスである。これらの情報はそれぞれ、dport、sport、daddr、saddr に入っている。

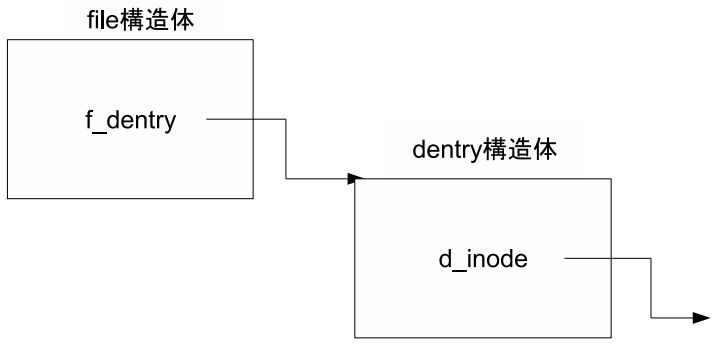

図 3.6: ファイルの実態

3.2.4　通信の遮断

通信の遮断を行うシステムは、iptables を使って実装している。以下では、まず iptables について、次に Xen のネットワークシステムについて、最後に実際にフィルタリングルールの追加を行う方法について述べる。

3.2.5　iptables

ターゲット

iptables は Linux カーネルの IP パケットフィルタルールのテーブルを設定・管理・検査するために使われる。複数の異なるテーブルを定義できる。各テーブルにはたくさんの組み込み済みチェインが含まれており、さらにユーザ定義のチェインを加えることもできる。

各チェインは、パケット群にマッチするルールのリストである。各ルールにはマッチしたパケットに対して何をするかを指定する。これは「ターゲット」と呼ばれ、同じテーブル内のユーザ定義のチェインにジャンプすることもできる。

一つのファイアウォールルールでは、パケットを判断する基準とターゲットとが指定される。パケットがマッチしない場合、チェイン内の次のルールが評価される。パケットがマッチした場合、ターゲットの値が次のルールを指定する。ターゲットの値はユーザ定義チェインの名前、または特別な値 ACCEPT、DROP、QUEUE、RETURN の内の一つである。

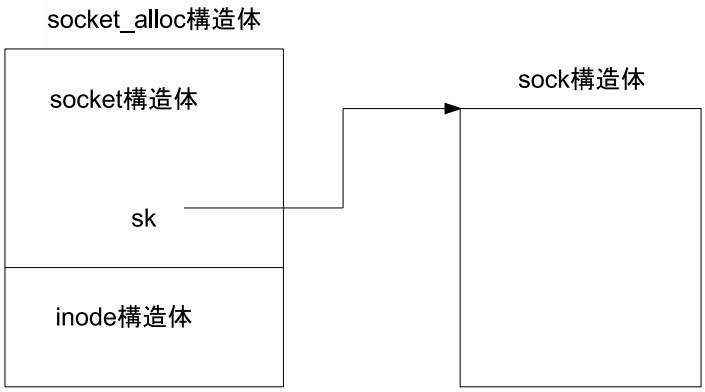

図 3.7: ソケット情報

ACCEPT はパケットを通すという意味である。DROP はパケットを床に落とす (捨てる) という意味である。QUEUE はパケットをユーザ空間に渡すという意味である (カーネルがサポートしている場合)。RETURN は、このチェインの検討を中止して、以前の (呼び出し元) チェイン内の次のルールから検討を再開するという意味である。組み込み済みチェインの最後に到達した場合、または組み込み済みチェインでターゲット RETURN を持つルールにマッチした場合、チェインポリシーで指定されたターゲットがパケットの行方を決定する。

テーブル

現在のところ 3 つの独立なテーブルが存在する。
-t –table *table*
このオプションは、このコマンドを適用するパケットマッチングテーブルを指定する。テーブルは以下のとおりである。

- filter：

  (-t オプションが渡さなければ) これがデフォルトのテーブルである。これは INPUT(マシン自体に入ってくるパケットに対するチェイン)・FORWARD(マシンを経由するパケットに対するチェイン)・OUTPUT(ローカルマシンで生成されたパケットに対するチェイン) という組み込み済みチェインが含まれる。

- nat：

  このテーブルは新しい接続を開くようなパケットに対して参照される。これにはPREROUTING(パケットが入ってきた場合、すぐにそのパケットを変換するチェイン)・OUTPUT(ローカルで生成したパケットをルーティング前に変換するためのチェイン)・POSTROUTING(パケットが出て行くときに変換するためのチェイン) という 3 つの組み込み済みチェインが含まれる。

- mangle：

  このテーブルは特別なパケット変換に使われる。これには、カーネル 2．4.17 までは PREROUTING(パケットが入ってきた場合、すぐにそのパケットを変換するためのチェイン)・OUTPUT(ローカルで生成されたパケットをルーティング前に変換するためのチェイン) という 2 つの組み込み済みチェインが含まれる。カーネル 2．4.18 からは、これらのほかに INPUT(マシン自体に入ってくるパケットに対するチェイン)・FORWARD(マシンを経由するパケットに対するチェイン)・POSTROUTING(パケットが出て行くときに変換するためのチェイン) という 3 つの組み込み済みチェインも含まれる。

今回の実装では、デフォルトの filter テーブルを使う。

オプション

今回は、特によく使うものだけ紹介する。

コマンド

コマンドは、実行する特定の動作を指定する。

- -A, –append　チェイン　ルールの詳細

  選択されたチェインの最後に 1 つ以上のルールを追加する。送信先や送信元の名前が 1 つ以上のアドレスに解決された場合は、可能なアドレスの組み合わせそれぞれに対してルールが追加される。

- -D, –delete　チェイン　ルールの詳細

- -D, –delete　チェイン　ルール番号

  選択されたチェインから 1 つ以上のルールを削除する。このコマンドには 2 つの使い方がある：チェイン内の番号 (最初のルールを 1 とする) を指定する場合と、マッチするルールを指定する場合である。

- -F, –flash [チェイン]

  選択されたチェイン (何も指定しなければテーブル内の全てのチェイン) の内容を全削除する。これは全てのルールを 1 個ずつ削除するのと同じである。

パラメータ

パラメータはルールの仕様を決める。

- -p, –protocol *[!] protocol*

  ルールで使われるプロトコル、またはチェックされるパケットのプロトコル。指定できるプロトコルは、tcp、udp、icmp、all のいずれか 1 つか、数値である。数値には、これらのプロトコルのどれかないし別のプロトコルを表す数値を指定することができる。/etc/protocols にあるプロトコル名もいていできる。プロトコルの前に”!”を置くと、そのプロトコルを除外するという意味になる。数値 0 は all と等しい。プロトコル all は全てのプロトコルとマッチし、このオプションが省略された際のデフォルトである。

マッチングの拡張

iptables は拡張されたパケットマッチングモジュールを使うことができる。これらのモジュールは、-p または–protocol で暗黙のうちに指定されるか、-m または–match の後にモジュール名を続けて指定される。これらのモジュールの後ろには、モジュールに応じて他のいろいろなコマンドオプションを指定することができる。複数の拡張マッチングルールを一行で指定することができる。大部分のものは、!を前に置くことによってマッチングの意味を逆にできる。今回使ったものだけ紹介する。

physdev

このモジュールは、ブリッジデバイスのスレーブされた、ブリッジポートの入出力デバイスにマッチする。このモジュールは、ブリッジによる透過的な IP ファイアウォールの基盤の一部であり、カーネルバージョン 2.5.44 以降でのみ有効である。

- –physdev-in *name*

  パケットが受信されるブリッジのポート名 (INPUT、FORWARD、PREROUTING チェインに入るパケットのみ)。インタフェース名

が”+”で終わっている場合、その名前で始まる任意のインタフェース名にマッチする。ブリッジデバイスを通して受け取れなかったパケットは、’!’ が指定されていない限り、このオプションにマッチしない。

- –physdev-out *name*

  パケットを送信することになるブリッジのポート名 (FORWARD、OUTPUT、POSTROUTING チェインに入るパケットのみ)。インタフェース名が”+”で終わっている場合、その名前で始まる任意のインタフェース名にマッチする。nat と mangle テーブルの OUTPUT チェインではブリッジの出力ポートにマッチさせることができないが、filter テーブルの OUTPUT チェインではマッチ可能である。パケットがブリッジデバイスから送られなかった場合、またはパケットの出力デバイスが不明であった場合は、’!’ が指定されていない限り、パケットはこのオプションにマッチしない。

tcp

これらの拡張は’–protocol tcp’ が指定された場合にロードされる。

- –source-port *[!] port[:port]*

  送信元ポートまたはポート範囲の指定。サービスまたはポート番号を指定できる。port,port という形式で、2 つの番号を含む範囲を指定することもできる。最初のポートを省略した場合、”0”を仮定する。最後のポートを省略した場合、”65535”を仮定する最初のポートが最後のポートより大きい場合、2 つは入れ換えられる。フラグ–sport は、このオプションの便利な別名である。

- –destination-port *[!] port[:port]*

  送信元ポートまたはポート範囲の指定。フラグ–dport は、このオプションの便利な別名である。

ターゲットの拡張

iptables はターゲット拡張モジュールを使うことができるが、今回は割愛する。

3.2.6　Xen のネットワークシステム

Xen の仮想ネットワークは、ドメイン上に仮想的なネットワーク・インタフェースを作成し、それらに対して MAC アドレスや IP アドレスなどを設定することで通信できるようになっている。シンプルに設計されており、基本的なネットワークの組み合わせで、複雑なネットワークを実現できる。

仮想イーサネットで接続

Xen のドメイン 0 から新たにドメイン U を起動すると、ドメイン 0 とドメイン U を接続する 1 対の仮想的なネットワーク・インタフェースが作成される。仮想的なクロス・テーブルで、ドメイン 0 とドメイン U が直接接続されているイメージである (図 3.8)。

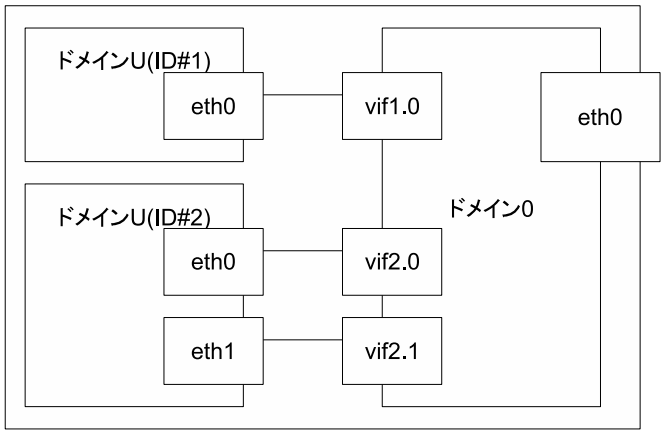

図 3.8: Xen のネットワークの概念図

このときドメイン U 側のデバイス名には、通常の Linux の場合と同様に、eth0,eth1,…といった「eth」＋「通し番号」の名前がつけられる。これに対してドメイン 0 側には、vif1.0,vif2.0 といったデバイス名が作成され、1 つ目の数字 (ピリオドの前) は「xm list」コマンドで確認できるドメイン U の ID 番号、2 つ目の数字 (ピリオドの後) はドメイン U 内にあるインタフェースの通し番号になる。ドメイン U が削除されると、これらの vif で始まる仮想インタフェースも削除される。

ブリッジ・モードで構築

ドメイン U を作成すると、ドメイン 0 とドメイン U を接続するための仮想インタフェースが作成されるが、このままではドメイン U はドメイン 0 に接続されている物理的なネットワークにアクセスできない。またドメイン同士を接続するためのネットワークもないため、ドメイン同士の通信もできない。

そこで Xen は、「ブリッジ・モード」「ルーティング・モード」「NAT モード」の 3 種類のネットワーク・モードを用意し、外部ネットワークとの接続や、ドメイン同士の通信ができるようにしている。

このうち最も代表的なネットワーク・モードはブリッジ・モードである。Xen をインストールした際のデフォルトは、ブリッジ・モードに設定されている。ここではこのブリッジ・モードを使用したネットワーク環境の構築方法について紹介する。

ドメイン 0 を起動した後、「ifconfig」コマンドを実行すると、「xenbr0」という名前のブリッジ･インタフェースが作成されていることが分かる。Xen ではこのブリッジ･インタフェースを使用することで、仮想ネットワーク上にあるドメイン U を、物理ネットワーク上にある独立したホストと同じように見せられる (図 3.9)。xenbr0 というブリッジ･インタフェースは、Xen のデーモンである xend を起動する際に呼び出される「/etc/xen/script/network-bridge」スクリプトの中で定義され、作成されている。

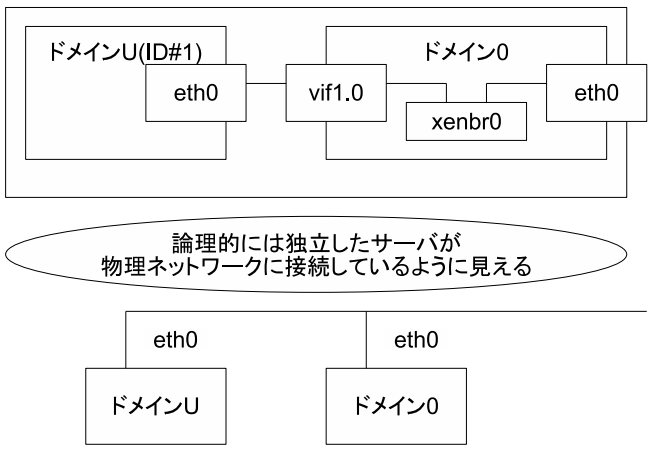

図 3.9: ブリッジ接続を利用してドメイン U から物理ネットワークにアクセス

3.2.7　フィルタリングルールの追加

Xen の規定の設定では、ドメイン 0 の中で、すべてのドメインが個々のホストとしてネットワーク上に存在することができるブリッジ・モードを使っている。ドメイン 0 で iptables を広範囲にわたって使用するなら、ブリッジ・モードの影響を考えなければならない。なぜならブリッジされたパケットは、iptables の PREROUTING、FORWARD、POSTROUTING チェインを通るからである。iptables の FORWARD ならすべてのパケットに対して設定を行うことができる。

また、ブリッジデバイスを通っているので、physdev モジュールを使う。今回は、ドメイン U からのパケットのフィルタリングを行うので、インタフェース名は vifx.0 を指定する (x はドメイン U のドメイン ID である)。

さらにポート番号を指定するために、p オプションで tcp を指定する。こうすることで、送信元ポート、送信先ポートそ指定することができるようになる。

これらをふまえて iptables のルールを書くと、以下のようになる。ここでは、ドメイン ID が 1 のドメインの送信元ポート 80、送信先ポート 0 のパケットをフィルタリングするとする。

- iptables -A FORWARD -m physdev –physdev-in vif1.0 -p tcp –dport 0 –sport 80 -j DROP

ログファイル

フィルタリングルールを追加するとき、すでに追加されているルールを追加するのは無意味であるし、また現在までに追加されているルールを見たいという場合もあるかもしれない。そこで、追加したフィルタリングルールのログを保存するようにした。新たにルールを追加するときは、ログを見てすでに追加されているルールは追加しないようにした。これにより、同じファイアウォールルールが複数追加され、性能が落ちることを回避した。

3.3　関連研究

3.3.1　Antfarm

Antfarm [2] とは、仮想マシンモニタ上からドメインに手を加えずにプロセスの状態を取得する技術である。さらに、取得したプロセスの状態

を使って仮想マシンモニタ上で anticipatory I/O scheduling を実装している。

プロセスとアドレス空間は一対一に対応するので、アドレス空間を観察すればプロセスの状態が分かる。例えば i386 では、CR3 レジスタの値が変化したら context switch したと分かり、CR3 レジスタの値が今までにない新しい値だったら新しいプロセスが生成されたことが分かり、CR3 レジスタの値に対応するページマッピングが存在せず TLB が flush されたら、その CR3 の値に対応するプロセスが終了したことが分かる。

ドメイン上のオペレーティングシステムのソースコードに手を加えずに、オペレーティングシステムの情報を取得する点は本研究と同じである。しかし、取得できる情報はプロセスの状態の変化だけであり、何のプロセスかまでは分からない。オペレーティングシステムの内部構造まで理解していないので、取得できる情報は限られている。ドメイン上のオペレーティングシステムに依存せず、Linux 以外でも通用する技術であるという点で本研究より優れている。

3.3.2　Geiger

Geiger [3] とは、仮想マシンモニタ上からドメインに手を加えずにバッファキャッシュの状態を取得する技術である。

Antfarm と同様に、ドメイン上のオペレーティングシステムのソースコードに手を加えずに、オペレーティングシステムの情報を取得する技術である。

3.3.3　ドメイン 0 から物理マシン全体のスケジューリングを行うシステム

tadokoro システム (仮) とは、仮想マシンモニタ上からゲスト OS のランキューを操作することで、ドメイン間にまたがるプロセスのスケジューリングが行えるシステムである。

Xen 仮想マシンモニタがすべてのプロセスをスケジューリングすることにより、ゲスト OS 内のプロセスの優先度を物理マシン全体の中で決めることができる。物理マシン全体の中での優先度を決めることにより、ドメインの優先度に依存してその中の優先したくないプロセスまで優先してしまう問題を解決した。

ゲスト OS のメモリには、Xen の機能を用いてドメイン 0 のプロセスからアクセスする。本研究は、この部分を実装に使わせてもらっている。

第4章　実験

ベンチマークとして httperf [4] を使って、通信の性能を測定する。

4.1　httperf

httperf は http サーバベンチマークツールである。Apache の ab との違いは、毎秒のリクエスト数を指定できるところである。ab では CPU は常にフル稼働になってしまう。ab では、「毎秒 100 リクエストを送った場合の CPU の負荷がどうなっているか」が分からない。

httperf の特徴として以下の設定が利用できる。

- rate

  1 秒間に生成するコネクションの数。1 秒間に発生するリクエスト数は rate x num-call になる。

- num-conn

  総コネクション数。生成したコネクションの数が numm-conn に達するとベンチマークが終了する。

- num-call

  ひとつのコネクションにつき、いくつのリクエストを送るか。総リクエスト数は num-call x num-conn になる。

4.2　実験環境

- Virtual Machine Monitor：Xen3.1.0
- Domain0 の OS：Linux Kernel 2.6.18
- DomainU の OS：Linux Kernel 2.6.18
- CPU：Athlon(tm) 64 Processor 3500+
- MEMORY:1Gbyte(Domain0 に 512Mbyte DomianU に 256Mbyte 割り当てた)

4.3　ドメインのポーズ時間

httperf を使った測定の前に、実際にドメイン U をポーズしている時間を測定する。実験は図 4.1 で測定した。

```
struct timeval
{
    time_t              tv_sec        /* 秒*/
    suseconds_t     tv_usec/*マイクロ秒*/


double
gettimeofday_sec()
{
    struct timeval tv;
    gettimeofday(&tv, NULL)
    return tv.tv_sec + (double)tv.tv=usec*1e-6
}

int
xFilter(....)
{
for(int i = 0; i < 10000; i++) {
  double t1, t2;
  t1 = gettimeofday_sec();
 /*   Domain pause  */
         .
         .
         .
 /*   Domain unpause   */
  t2 = gettimeofday_sec();
  }
}
```

図 4.1: 時間計測

結果は表 4.1 の通りである。

ポーリングごとに行うポーズの時間は決して小さいとはいえない値である。スループットとしては約 1%であった。しかし、xFilter は仮想マシンモニタ管理者が危険な状況であると感じたときのみしようするので、この

表 4.1: 実験結果 (ドメイン U のポーズ時間 [ミリ秒])

平均	最大	最小
19	52	15

表 4.2: 実験結果 (コネクションの処理時間 [ミリ秒])

	実験所要時間	最小	平均	最大	中央点	標準偏差
xFilter なし	666.6600	0.2	0.3	27.2	0.5	0.2
xFilter(15 秒)	666.6600	0.2	0.3	30.9	0.5	0.8
xFilter(10 秒)	666.6600	0.2	0.4	31.0	0.5	1.0
xFilter(5 秒)	666.6600	0.1	0.4	31.2	0.5	1.4
xFilter(3 秒)	666.6600	0.2	0.5	33.4	0.5	1.7

オーバーヘッドは常にあるわけではない。

4.4　ポーリングオーバーヘッド

フィルタリングにかからないルールを指定して、間隔を変えてポーリングを行う。rate を 150、num-conn を 100000、num-call を 1 にして実験を行った。実験は xFilter を使わない場合、ポーリング間隔を 15 秒にした場合、10 秒にした場合、5 秒にした場合、3 秒にした場合の 5 通りで行った。結果は図 4.2、図 4.3 の通りである。

ポーリングの時間を短くすると平均処理時間は長くなった。これはメモリマップしているときにドメインをポーズしていためでると思われる。さらに最小、最大、中央値はほぼ変わっていないにもかかわらず標準偏差がポーリング間隔が短くなるにつれて大きくなっていることを見ても、最大

表 4.3: 実験結果 (コネクション成立にかかる平均時間 [ミリ秒])

	コネクション成立にかかる時間
xFilter なし	0.1
xFilter(15 秒)	0.1
xFilter(10 秒)	0.1
xFilter(5 秒)	0.2
xFilter(3 秒)	0.2

値に近い大きな値を取る回数が大きくなっていることが分かる。つまりドメインのポーズによって処理に時間がかかることが影響している。

　また、実験所要時間はすべて同じだった。つまりスループットには影響はなかった。これはサーバにまだ余裕があったためと思われ、実験としては性格な性能実験とはいえないかもしれない。しかし、毎秒 150 コネクション程度ならばスループットには影響がないということが分かった。

第5章　まとめ

本研究では、仮想マシンモニタの管理者がゲスト OS にアクセス権を持たなくとも、ゲスト OS 内の情報を取得し、利用できるシステムである xFilter の提案と実装を行った。本システムを利用することで、仮想マシンモニタ管理者はゲスト OS にアクセス権を持たなくても、ゲスト OS のパケットをゲスト OS 内の情報を用いてフィルタリングを行うことができる。

ゲスト OS 内の情報の取得には、ゲスト OS のメモリをマッピングすることでプロセスの情報を参照できるようにした。さらに、取得した情報を用いてのパケットフィルタリングには、iptables を用いて実装した。ポーリングを行い、定期的にゲスト OS のプロセス情報をチェックすることにより、新たな通信が張られた場合も即座に対処できる。

本システムは、仮想マシンモニタ管理者が通信制御を行う場合ゲスト OS にアクセスできなければ OS 内の情報を利用できず、OS 単位の大雑把なフィルタリングしかできないという問題を解決した。仮想マシンモニタ管理者がゲスト OS に対して細粒度なフィルタリングが行えるようになったことで、ゲスト OS がクラックされたとき、仮想マシンモニタ管理者の権限で即座に通信を制御でき、また問題のあるプログラム以外まで制限してしまうことはなくなった。

フィルタリングのルールに指定できるものとして、プロセス ID とユーザ ID を用意した。プロセス ID の場合は、そのプロセスの開いているポートからの通信をすべて拒否する。ユーザ ID の場合は、そのユーザのプロセスが開いているポートからの通信をすべて拒否する。また、ID を指定するためにプロセスの情報を本システムのユーザが分からなければならない。そこで、ドメイン ID だけを指定するとそのドメインの持つプロセス情報が表示されるようにした。

実験では、毎秒 150 コネクション程度ならばスループットには影響がないことを確認した。

今後の課題

今回の実装ではポーリングを採用したが、ポーリングでは一度チェックしてから次のチェックまで一定の時間があり、その間はドメイン U の情報

は更新されないので、拒否すると指定したプロセスが新たな通信 を始めても気づかずに通過を許してしまう。それを避けるためにポーリングの間隔を短くすると、頻繁にポーズがかけられるのでオーバーヘッドが大きくなってしまう。そこで、ドメイン 0 のデバイスドライバを改造し、パケットが送られてきたときにドメイン U のメモリをチェックし、パケットの通過を許すか判断するようにすれば、パケットがフィルタリングからもれることはなく、セキュリティ性が高まる。しかも、通信があまり行われていないときはポーズの時間も少なく、オーバーヘッドはほとんどなくなる。

　また、今回の実験ではサーバに余裕があったためスループットには影響がなかったが、今後さらに負荷をかけてスループットに影響がある状況で実験したい。

参考文献

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM Press, pp. 164–177 (2003).

[2] Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Antfarm: tracking processes in a virtual machine environment, *USENIX-ATC'06: Proceedings of the Annual Technical Conference on USENIX'06 Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 1–1 (2006).

[3] Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Geiger: monitoring the buffer cache in a virtual machine environment, *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, ACM, pp. 14–24 (2006).

[4] Mosberger, D. and Jin, T.: httperf-a tool for measuring web server performance, *SIGMETRICS Perform. Eval. Rev.*, Vol. 26, No. 3, pp. 31–37 (1998).