# プロセスの依存関係に基づく分散システムのセキュリティ機構

光来 健一† 千葉 滋‡ 益田 隆司†

†東京大学大学院 理学系研究科 情報科学専攻
‡筑波大学 電子・情報工学系, 科学技術振興事業団 さきがけ研究 21

**要旨**

インターネットの拡がりに伴ってシステムが外部から攻撃される機会が多くなっており、ファイアウォールなどによるセキュリティ対策が行われている。しかしながら、これらの対策の多くは一旦システムに侵入されると無力になるので、侵入された後も行動を追跡して不正行為を防ぐことができるセキュリティモデルが必要とされている。そこで我々はプロセスの依存関係に基づくセキュリティ機構を提案する。このセキュリティ機構はプロセスが周囲のプロセスに 与えた影響を分散システム内部において追跡し、その情報をアクセス制限に利用する。プロセスの依存関係としては、プロセスの通信依存関係とプロセスの親子関係を考える。我々は汚染という概念を導入することにより、このセキュリティ機構を Linux カーネルに効率よく実装した。

## 1 はじめに

インターネットが普及し、様々な組織内部でイントラネットが構築され、分散システムとして運用されるようになってきている。しかしイントラネットはオープンなインターネットの技術を転用したものなので、十分なセキュリティ対策を行わなければ、外部からのシステムへの侵入を容易に許してしまうことになる。外部からの攻撃には様々なものが考えられるが、セキュリティホールを利用する方法やユーザになりすましてログインする方法などが最も一般的である。

外部からの攻撃を防ぐためにファイアウォールを構築したり認証を強化したりといった方法が使われているが、これらの方法では分散システムの境界に作られた防衛ラインを突破されるとそれ以降の攻撃に対して無力になってしまう。ファイアウォールは外部からのパケットを制限することによって内部のシステムを守るが、一旦侵入されるとファイアウォールは効力を失う。また、認証も基本的にはパスワードが合ってさえいれば成功するので、一旦パスワードを知られてしまうと信頼できるユーザとして振舞われてしまう。

この防衛ラインモデルの問題を解決するには、システムに侵入された後もその行動を追跡して不正行為を防ぐことができるセキュリティモデルが必要になる。我々はこの行動追跡モデルに基づいたセキュリティシステムを実現するために、プロセスの依存関係を利用したセキュリティ機構を提案する。このセキュリティ機構はプロセスの依存関係に基づいて、プロセスが周囲のプロセスに与えた影響を分散システム内部で追跡し、その情報をアクセス制限に利用する。プロセスの依存関係としては、プロセス同士の通信によって生じる依存関係とプロセスの親子間の依存関係を考える。

行動追跡モデルに基づくセキュリティ機構を実現するために、我々は Compacto というシステムを設計した。Compacto ではプロセスの依存関係を追跡できるように汚染という概念を導入し、プロセスに汚染情報を保持させている。汚染はプロセスが周囲のプロセスによって与えられた影響を蓄積したものであり、より信頼度の低いホストと通信したプロセスの汚染レベルはより高くなる。汚染は汚染レベルと汚染の原因となったホスト集合で表され、汚染レベルで階層的なアクセス制限、汚染の原因となったホスト集合で排他的なアクセス制限を可能にしている。また、汚染はプロセスの依存関係に沿って、分散システム内部のホストで動いているプロセスからプロセスへと伝播する。我々はこのセキュリティ機構を Linux カーネルに実装し、TCP オプションを用いた汚染情報のピギー・バックなどの工夫により、行動追跡モデルを効率よく実現することができている。

以下、2 章ではセキュリティ問題と従来の対策について述べ、3 章で我々の提案するセキュリティ機構について述べる。4 章ではそのセキュリティ機構を実現するシステム Compacto の設計について述べ、5 章ではその設計の妥当性、問題点、制限について論じる。さらに 6 章で、提案するセキュリティ機構のオーバヘッドを調べるために行った実験とその結果について述べる。そして 7 章で関連研究に触れ、8 章

で本稿をまとめて今後の課題を述べる。

## 2 背景

分散システムでは複数のホストが協調して処理を行えるように、外部に対してデータベース検索やリモートアクセスなどのサービスを提供しており、常に外部からの攻撃の危険にさらされている。これらのサービスはサーバ・プロセスを動かすことによって実現されており、そのためのプログラムをサービス・プログラムと呼ぶ。最も一般的な攻撃として、サービス・プログラムのセキュリティホールを利用した攻撃が頻繁に行われている。例えば、プログラムのバッファをオーバフローさせることで任意のコードを実行させることが可能になる。また、サービス・プログラムが他のプログラムを呼び出している場合には任意のプログラムを実行させたり、そのプログラムのセキュリティホールを利用することが可能かもしれない。これらのセキュリティホールはプログラムのバグやチェック漏れといったプログラマの不注意が原因であることが多いが、それを完全になくすのは難しい。

別の攻撃方法としては、システム内部のユーザになりすまして侵入する方法がある。システムにログインする際には一般的には認証が行われるが、利便性のために特定のホストからならパスワード入力を省略できるようにしている場合がある。これを悪用されると簡単にシステムへの侵入を許してしまうことになる。そうでなくても、辞書攻撃や総当たり攻撃によってパスワードを類推されて侵入されてしまう場合もある。他にもネットワーク上を流れるパスワードを盗み見られて、パスワードを知られてしまうこともあり得る。こうして不正にログインしてきた侵入者は、正規ユーザと区別するのが難しい。

このようなシステムへの攻撃を防ぐために様々な対策がとられている。最も簡単で有効な方法は、ファイアウォールを構築して外部から不正にアクセスされる危険をなくしてしまうことである。ファイアウォールを通過できるパケットを制限することで、サービス・プログラムにそもそも不正なパケットが届かないようにすることができる。これによりプログラムにセキュリティホールがあっても利用するのが難しくなり、認証の機会すら与えないようにすることも可能である。

また、サービス・プログラムにパケットが届いた場合でも、StackGuard [2] と呼ばれる手法を用いることによりバッファ・オーバフロー攻撃を無効化できる。バッファ・オーバフロー攻撃はプログラムのスタックを溢れさせて任意のコードを書き込んでおき、関数から戻る時にそのコードを実行させるというものである。StackGuard では関数を呼び出す時にスタックに印をつけておき、関数から戻る時に印が上書きされていないかどうかを調べる。もし上書きされていればバッファがオーバフローさせられていると判断することができる。

認証に関しては、通信路の暗号化 [1, 4] やワンタイム・パスワード (OTP) [6, 7] などによって認証を強化するという対策がとられている。通信路の暗号化によって、パスワードは暗号化されてネットワーク上を流れる。仮に暗号化されたパスワードを盗み見られたとしても、別の通信路では異なる暗号化がなされるので、盗まれた暗号化パスワードが不正利用されることはない。OTP ではクライアントはサーバから送られてきた種から使い捨てパスワードを計算して、それをサーバに送る。この場合も使い捨てパスワードは毎回異なるので、盗まれても問題にはならない。

しかしながら、これらの対策はどれも信頼できるホストと信頼できないホストの境界に防衛ラインを敷き、内側に入ってこられないようにするというものである (防衛ラインモデル)。このモデルでは、一旦防衛ラインを突破されてしまうとそれ以降の攻撃に対して無力になってしまう。ファイアウォールによるパケット制限も、ファイアウォール・マシンに侵入されて設定を無効にされると無力化する。サービス・プログラム自身が外部からのアクセスに対して行っている制限も、そのプログラムが破られてしまうと無意味である。また、認証もソフトウェアのみでやっている限りは、パスワードが知られてしまうと無力になることが多い。

## 3 プロセスの依存関係に基づくセキュリティ機構

### 3.1 行動追跡モデル

防衛ラインモデルの問題を解決するには、防衛ラインを突破されてシステムに侵入された後もその行動を追跡することができるセキュリティモデル (行動追跡モデル) が必要になる。行動追跡モデルの下で

は、あるプロセスが周囲のプロセスからどういう影響を受けているかという情報を用いてアクセス制限を行うことが可能になる。これにより、例えば、サービス・プログラムのセキュリティホールを利用して入ってきた侵入者に対しては、ほとんどのアクセスを拒否するようにすることができる。また、近くの信頼できるホストを踏み台として使われて、信頼できるユーザとしてログインされた場合でも、その侵入者がいた元々のホストを考慮して行動を制限することができる。ちなみに、行動追跡モデルは従来の防衛ラインモデルと相補的なものであり、共存が可能である。

我々は行動追跡モデルに基づくセキュリティシステムを実現するために、プロセスの依存関係を利用したセキュリティ機構を提案する。このセキュリティ機構はプロセス同士の依存関係に基づいて、プロセスが周囲のプロセスに与えた影響を追跡し、その情報を用いてアクセス制限を行うことができる。例えば、どのプロセスがどういう順番で実行されたかという情報によって、システムの提供するサービスが正しく使われているかどうかを判断することができる。これにより、サーバ・プロセスから直接シェルを起動しようとした場合には、サービス・プログラムのセキュリティホールを使って侵入してきたと見なしてシェルの起動を失敗させることが可能になる。また、侵入された形跡のある近くのホストからログインを試みられた場合には、侵入者がそのホストを踏み台として利用している可能性があると考えて、ログインを拒否することができる。

## 3.2 プロセスの依存関係

プロセスの依存関係はプロセスが他のプロセスに何らかの影響を与えられた時に生じる関係である。図 1にプロセスの依存関係の一例を示す。提案するセキュリティ機構ではプロセスの依存関係として、プロセス同士の通信による依存関係とプロセスの親子間の依存関係を考えている。プロセスの通信依存関係は、異なるホストまたは同じホスト上で動いている二つのプロセスが通信を行うことによって生じる。プロセス間で行われる通信とは、ソケットやパイプ、ファイルといった OS のリソースを介してデータをやりとりすることである。セキュリティ面から考えた場合、受信側のプロセスが送られてきたデータによって影響を受けるので、受信側のプロセスから送信側のプロセスに対して依存関係が生じると考えることができる。ただし、一般にはプロセスは互いに通信し合うので相互に依存関係が生じることが多い。

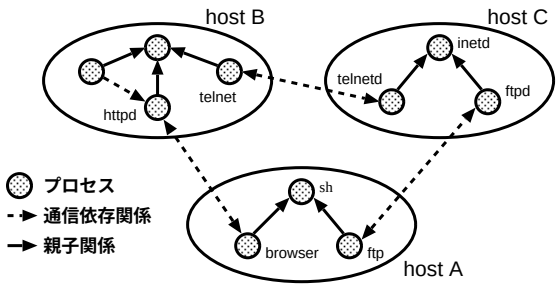


図 1: プロセスの依存関係の一例

一方、プロセスの親子関係は、同じホスト上でプロセスが新しいプロセスを生成することによって生じる。プロセスの生成とは、fork システムコールによるプロセスの複製または、exec システムコールによる別のプログラムの実行のことである。この時、生成元のプロセスを親プロセス、生成されたプロセスを子プロセスと呼ぶ。セキュリティ面から考えた場合、親プロセスが子プロセスの生成を支配しているので、子プロセスから親プロセスへの依存関係が生じると考えることができる。

プロセスの及ぼす影響を追跡できるようにするためには、プロセスの依存関係として通信依存関係と親子関係の両方を考えることが必要不可欠である。プロセスが他のホストのプロセスと通信する分散システムにおいては、プロセス同士の通信によって生じる依存関係を考えないわけにはいかない。さらに、プロセスの影響を追跡するためにはホスト内部でも通信依存関係を考えることが重要になる。パイプなどのプロセス間通信については、パイプでつながった相手のプロセスへの影響を追跡する必要がある。また、ファイルを介して他のプロセスに影響を与える場合も追跡が必要である。例えば、あるプロセスが設定ファイルを書き換えて、サーバがその設定ファイルを読み込むことによっておかしな動作をした場合、設定ファイルを書き換えたプロセスの影響がサーバに及んだことになるからである。

プロセスの親子関係はどのプロセスがどのプロセスを実行したかというプロセスの実行順を追跡するためにも必要であるが、通信依存関係と一緒に考えてプロセスの影響を追跡するためにも必要とされる。プロセスが他のプロセスからデータを受け取った後、そのプロセス自身が別のプロセスと通信する場合に

は通信依存関係のみで影響を追跡することができる。しかし、子プロセスを作って別のプロセスと通信するような場合は親子関係も考慮しなければならない。例えば、ウェブサーバが要求を受け取って CGI プログラムを実行し、その CGI プログラムが通信を行うような場合である。

# 4 Compacto

我々は行動追跡モデルに基づくセキュリティ機構を実現するシステム Compacto を設計した。このシステムでは行動追跡モデルを効率よく実装できるようにするために、汚染という概念を導入している。

## 4.1 汚染

### 4.1.1 汚染レベルと汚染元のホスト集合

Compacto ではプロセスの依存関係に基づいてプロセスの与えた影響を追跡できるように、汚染という概念を導入してプロセスに汚染情報を保持させている。汚染はプロセスが他のプロセスによって与えられた影響を蓄積したものである。他のプロセスに影響を与えられる例としては、プロセスが他のプロセスからデータを受け取った場合や親プロセスによって生成された場合などが挙げられる。つまり、汚染はプロセスがどのくらい外部から攻撃されているかという危険度を表す指標になる。プロセスの汚染レベルは、そのプロセスに影響を与えたプロセスが動いているホストの信頼度に応じて決まる。信頼度の低いホストで動いているプロセスに直接または、間接的に影響を与えられていれば、それだけプロセスの危険度も高くなると考えられるので、汚染レベルも高くなる。

汚染は汚染レベルと汚染の原因となったホスト集合によって表される。汚染レベルは汚染の度合を表しており、値が大きい程ひどく汚染されていることを表す。汚染レベルが最小の時はプロセスが全く汚染されていない状態であり、他のホストで動いているプロセスからいかなる影響も受けていないことを表す。一方、汚染レベルが最大の時はひどく汚染された状態であり、そのプロセスまたはそのプロセスに影響を与えたプロセスが信頼できない外部のホストと通信したことを表す。Compacto の現在の実装では、汚染レベルに 4 ビットの数値が割り当てられており、最小値は 0、最大値は 15 である。

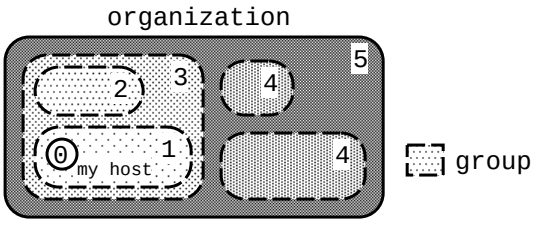


図 2: 自ホストから見た組織内グループの汚染レベルの対応例 (数値は汚染レベル)

汚染レベルの最小値と最大値以外については、各分散システムでホストの信頼度に応じて自由に対応させることができる。一般には、分散システムの中でホストの信頼度は階層的になっていることが多いので、自ホストを最も信頼できるホストとし、より大きなグループに属するホストをより信頼できないホストと考える。そして分散システム外部のホストを最も信頼できないとする。例えば、自ホストの属する部門内のホストにはより小さな汚染レベルを対応させ、それ以外の部門のホストにはより大きな汚染レベルを対応させるなどが可能になる。図 2は自ホストから見た組織内のグループに対応する汚染レベルの例である。このように分散システム内部でもホストの信頼度に応じて異なる汚染レベルを対応させることにより、内部犯による攻撃を防ぐこともできる。

汚染レベルを用いることにより、アクセス制御を階層的に行うことが可能となる。階層的なアクセス制御とは、強い権限を持ったプロセスは弱い権限しか持たないプロセスにできることは全てできるというものである。つまり、汚染レベルの低いプロセスに適用されるアクセス制限は、それよりも汚染レベルの高いプロセスにも必ず適用される。

一方、汚染の原因となったホスト集合を汚染レベルと併わせて記録しておくことで、より細かなアクセス制御が可能になる。汚染レベルはアクセス権限を階層的にコントロールするのには役立つが、排他的なアクセス制限をするのは難しい。汚染元のホスト集合に関する情報を用いることにより、同一の汚染レベルのホストに対して異なるアクセス制限を適用することができる。例えば、あるホストには ftp だけを許可し、別のホストには telnet だけを許可するといったことが可能になる。

Compacto の現在の実装では、汚染元のホスト集

合は 12 ビットのビット集合で表されている。汚染の原因となったホスト集合を 12 ビットにエンコードするために、各ビットにあらかじめいくつかのホストからなるグループを割り当てておく。この割り当て方は各分散システムにおいて、汚染レベル毎に決めることができる。各ビットがホストを一意に識別できるようにするのが理想であるが、オーバヘッドが大きくなる上、全てのホストを区別できるようにする必要がないことも多いので、少ないビット数に圧縮している。

#### 4.1.2 汚染の伝播

プロセスが周囲のプロセスに与えた影響をシステムが追跡できるようにするために、影響を与えた段階で汚染を即座に伝播させる。プロセス間での汚染の伝播はプロセスの依存関係に基づいて行い、プロセス同士の通信によって伝播する場合と、プロセスの親子間で伝播する場合が考えられる。

プロセス同士の通信では、ソケットやパイプ、ファイルといった OS のリソースを介して汚染が伝播する。そのため、各リソースについても汚染情報を記録しておく。プロセスがリソースからデータを読み込むと、プロセスはそのリソースの汚染レベルにまで汚染される。逆に、プロセスがリソースにデータを書き込むと、そのリソースはプロセスの汚染レベルにまで汚染される。汚染の伝播においては、元の汚染レベルと伝播された汚染レベルのより高い方が有効になり、汚染レベルが下がることはない。また、汚染元のホスト集合は、汚染レベルが変化しない場合には元のものと伝播されたものとの和集合になり、変化する場合には伝播されたものと等しくなる。

汚染を異なるホスト間で伝播させるためには、汚染情報を通信相手のホストに通知する必要がある。Compacto ではこのオーバヘッドを最小にするために、データを送るために使われるパケットのプロトコル・オプションを使って汚染情報をピギー・バックする。例えば、TCP オプションを使う場合には、オプションの種類 (1 バイト)、オプションのサイズ (1 バイト)、汚染情報 (2 バイト) の計 4 バイトの増加で済む。このパケットを受け取ったプロセスは、パケットの送信元のアドレスから決まる汚染レベルと、送られてきた汚染情報が示す汚染レベルのより高い方をパケットの汚染レベルと見なす。このような汚染情報の伝播は分散システム内部のホスト同士で通信する場合にのみ行い、外部からのパケットに関しては送信元のアドレスのみを利用して汚染レベルを決定する。そのため、カーネルに変更が必要なのは分散システム内部のホストだけである。

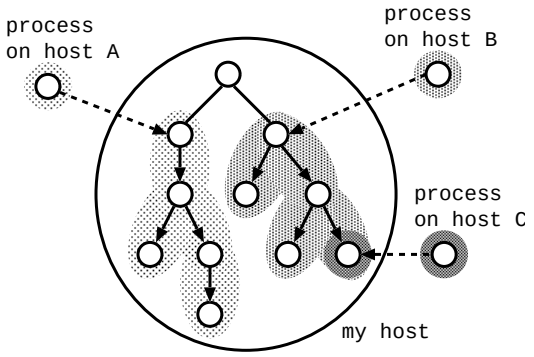


図 3: プロセス木を汚染が拡がる様子

一方、プロセスの親子間ではプロセスが新しいプロセスを生成する時に、親プロセスから子プロセスへと汚染が伝播する。この場合には、子プロセスの汚染レベルはその時点での親プロセスの汚染レベルと等しくなる。また、汚染元のホスト集合も親プロセスのものと等しくなる。図 3にプロセス木を汚染が拡がる様子を示す。

### 4.2 プロセス木の拡張

プロセスの依存関係としてプロセスの親子関係を利用するためには親子関係を正確に表すプロセス木が必要になるが、従来の UNIX の保持しているプロセス木では情報が不足している。従来の UNIX では、親プロセスが終了した場合その子プロセスは孤児プロセスとなり、プロセス木の根である init プロセスの直接の子供になってしまう。これではプロセスをバックグラウンドで動かしたままログアウトされたような場合に、祖先プロセスの情報を利用することができない。また、exec システムコールを実行するとプロセス内で動いているプログラムが置き換わり、exec 前のプロセスの情報が失われてしまう。一旦 fork システムコールでプロセスを複製してからすぐに別のプログラムを exec するような場合には問題ないが、直接 exec した場合には元のプロセスの情報を利用できなくなる。

そこでプロセスの親子関係を正確に保持できるように、従来の UNIX が保持しているプロセス木を拡張する。拡張されたプロセス木では、プロセスが終了してもその子プロセスがいる限りは親プロセスの

情報を残しておく。これにより、全てのプロセスから常に全ての祖先プロセスに関する情報が利用できることになる。また、このプロセス木には exec する前のプロセスに関する情報も残しておくので、どの祖先プロセスの情報も失なわれることはない。ただし、プロセスが終了した時点でプロセスの持つリソースは全て解放され、プロセスに関する最小限の情報以外は残らない。それゆえ、プロセス木を拡張しても OS のリソースが圧迫されるような事態にはならない。

## 4.3 アクセス制限

### 4.3.1 ポリシー

プロセスの依存関係をプロセスのアクセス制限に利用できるようにするために、ポリシーという形でアプリケーションのアクセス制限を記述する。さらに、ポリシーにはアプリケーションの中のセキュリティに関するコードも分離して記述される。これまで OS によって強制される以上のアクセス制限を必要とする時にはアプリケーション毎に独自に実装していたが、アプリケーションのバグやプログラマの不注意による影響を受けやすく、セキュリティホールの原因になってきた。セマンティクスの影響しない範囲でセキュリティに関するコードをポリシーとして分離することで、セキュリティホールを減らすことができ、修正も容易になる。

ポリシーにはシステムコールレベルでのアクセス制限を記述することができ、プロセスの依存関係はそれらのアクセス制限を適用するための条件として利用される。プロセスの汚染に関しては、汚染レベルが一定値以上である場合や特定のホストによって汚染されている場合などに、アクセス制限を適用するようにさせることができる。また、プロセスの親子関係に関しては、特定のプロセスを祖先に持つ場合や setuid システムコール等によって変更される前のユーザ ID・グループ ID が特定のものであった場合などに、アクセス制限を適用するようにさせることができる。

さらに、ポリシーは適用されているプロセスの子孫に対しても影響を及ぼすことができる。これにより一つのアプリケーションが複数のプロセスを使って作業を行う場合でも、アプリケーション全体に一貫したポリシーを適用するのが容易になる。例えば、ウェブサーバの起動する全ての CGI プロセスに対してアクセスできるディレクトリを限定したりすることが可能になる。また、プロセス木の根である init プロセスにポリシーを適用することによって、全てのプロセスに共通のアクセス制限を課すことができる。

ポリシーの設定・変更はセキュリティにおいて最も重要なので、全く汚染されていない状態でのみ許可する。全く汚染されていない状態にするためにはコンソールから直接ログインする必要があるので、他のホストから侵入されてポリシーを変更されてしまうということはあり得ない。しかし実際には、コンソールからのログインでしかポリシーを設定できないのは不便なので、汚染されている状態であってもアクセスをより制限する方向でのポリシーの設定は許している。

### 4.3.2 踏み台攻撃の防御の実例

Compacto では汚染情報を利用することによって、セキュリティの弱いホストを踏み台にした攻撃を防ぐことができる。攻撃者のいる外部ホストの汚染レベルを 15、分散システム内部にあるセキュリティの弱いホストの汚染レベルを 1、自ホストの汚染レベルを 0 とする。外部ホストからセキュリティホールなどを使って内部ホストに侵入された時、攻撃を受けたプロセスの汚染レベルは 15 になる。そのプロセスまたはそのプロセスと依存関係にあるプロセスが自ホストを攻撃してきた時には、その汚染レベル 15 が伝播されるので、その情報を基にアクセスを制限することができる。例えば、攻撃者が侵入した内部ホストから自ホストに rlogin してきた時には、汚染レベルが 15 であることより、次のようなルールを記述しておくことによってログインを拒否することができる。このルールは汚染レベルが 2〜15 の時には in.rlogind の実行を禁止するという意味である。

```
deny exec "in.rlogind" at-prange 2-15
```

一方、汚染情報が利用できない場合にはパケットの送信元のアドレスから判断するしかなく、汚染レベル 1 の内部ホストからの rlogin に見えてしまうのでアクセスを制限できない。

# 5 議論

## 5.1 汚染を導入する妥当性

プロセスの依存関係を利用してプロセスの与える影響を追跡するために、我々は汚染という概念を導入している。しかし、プロセスの依存関係全体は有

向グラフを構成するので、このグラフを直接扱う方法も考えられる。このグラフにおいて、ノードはプロセスを表し、有向エッジはプロセス同士の依存関係を表す。ノード (プロセス) から有向エッジを遡ってたどっていくことで、そのプロセスに影響を与えたプロセスを見つけることができる。

我々がプロセスの依存関係のグラフを直接扱わずに、汚染という概念を導入したのにはいくつかの理由がある。第一に、汚染という概念を導入することにより、維持管理しなければいけないデータ量を減らすことができる。依存関係のグラフはプロセスの影響を正確にたどれるようにするために、過去に行われた通信や終了したプロセスに関する情報も含んでいる。それゆえ、特に大規模な分散システムにおいてはグラフが非常に大きくなる可能性がある。また、ホスト間で通信を行って依存関係が生じる度に、その依存関係からたどられる依存関係全てを相手のホストに送らなければならない。このため、依存関係のグラフが複雑になればなるほど通信量が増えることになる。これに対して汚染を使う場合には、各プロセスが数バイトの汚染情報を保持するだけで済み、ホスト間で送る必要があるデータもわずかである。

第二に、汚染情報を使ってアクセス制限することにより、アクセス制限を行う際のオーバヘッドを減らすことができる。依存関係のグラフを使ってアクセス制限する時には、まず、グラフをたどってどのホストを通ったかを調べ、それから通ったホストを元にアクセス制限を行う。それに対して汚染を使う場合には、汚染レベルや汚染の原因となったホスト集合の比較という簡単な演算だけで済ませることができる。

第三に、依存関係のグラフの持つ全ての情報がアクセス制限に必要なわけではないという点が挙げられる。依存関係のグラフをたどれば、どういう経路で侵入してきたかという情報を得ることができる。しかし、外部の攻撃からシステムを守るという観点からは、その経路上に存在するホストの中で最も信頼度の低いホストが分かれば十分である。また、依存関係のグラフを用いればホスト単位ではなく、プロセス単位でアクセス制限を行うことができる。例えば、自ホストの telnetd プロセスと通信できるのはホスト A の telnet プロセスだけに限定する、などである。我々のシステムでは汚染とともに、プロセス木もアクセス制限に利用しているので、自ホスト側ではプロセス単位でアクセス制限が可能である。リモートホスト側のアクセス制限はホスト単位でしかできないが、性能とのトレードオフを考えると妥当であると思われる。

## 5.2 汚染の伝播を抑制する方法

我々のシステムでは、汚染はリソースからプロセスへ、プロセスからリソースへ、親プロセスから子プロセスへと自動的に伝播するが、実用性を考えると汚染の伝播を抑制できるようにする必要がある。これは一旦プロセスやリソースが汚染されてしまうと汚染レベルを下げることができないためである。汚染の伝播を放っておくと、外部からのアクセスによって重要なプロセスやリソースが高いレベルに汚染されてしまい、危険性を考慮すると身動きがとれなくなる可能性がある。例えば、一旦外部からアクセスされたウェブサーバ・プロセスは汚染されてしまうので、イントラネット内部からの非公開情報へのアクセス要求を拒否してしまうかもしれない。

このような事態を防ぐために、ポリシーで汚染の伝播を抑制する方法が考えられる。プロセスがリソースからデータを読み込む時やリソースにデータを書き込む時には、汚染を伝播させないようにすることができる。また、プロセスが新しいプロセスを生成する時には、親プロセスの汚染を子プロセスに伝播させないようにすることができる。しかし、ポリシーで汚染の伝播を抑制するとシステムのセキュリティが弱まってしまう。これは汚染を伝播させないようにすることによって、その先で行われる攻撃に対処できなくなる可能性があるからである。例えば、ブラウザで外部のウェブサーバにアクセスした後で、イントラネット内部のウェブサーバにアクセスできるようにするには、ブラウザが汚染されないようにする必要がある。このような場合にダウンロードされたコンテンツによってブラウザがおかしな挙動を示しても、システムは対処できないかもしれない。ただし、汚染の伝播を抑制しても我々の提案するセキュリティ機構を使わなかった場合に比べてセキュリティが弱まるということはない。

汚染の伝播を抑制する別の方法としては、プログラムの作り方や運用の仕方を工夫することが考えられる。例えば、サーバの場合は一つのプロセスで様々なホストからの要求を処理する代わりに、通信相手のホスト毎、もしくは汚染レベル毎にプロセスを生成して処理させるようにすれば、汚染をそのプロセスだけに局所化することができる。ただし、いくつ

かのサーバで使われているようなプロセス・プールの手法は使えなくなるので、性能に影響を及ぼす可能性がある。また、ブラウザの場合は外部のウェブサーバにアクセスする場合とイントラネット内部のウェブサーバにアクセスする場合とで別々のブラウザを使うことが考えられる。しかし、この方法はユーザに負担を強いることになる。

### 5.3 汚染の正しい伝播のための制限

現在の我々のシステムでは、ホスト間で汚染情報を正しく伝播できるようにするためにいくつかの制限がある。第一に、分散システム内部のホストのカーネルは侵入者によって書き換えられないものとしている。カーネルを書き換えれば汚染情報を偽ることができるようになるので、そのホストの汚染状況を正しく伝播させられなくなるからである。ただし、パケットの送信元のアドレスから決まる汚染レベルは伝播するので、それよりも低い汚染レベルであると偽ることはできない。一方、外部のホストから送られてきたパケットの場合には、送られてきた汚染情報は信用せず、常に最大の汚染レベルになるので問題ない。

第二に、パケットのヘッダに書かれる送信元のアドレスは詐称されないものとしている。Compactoではホストのアドレスに基づいて汚染の制御を行っているので、パケットの送信元のアドレスが詐称されると汚染を正しく制御することができなくなる。今のところ Compacto ではアドレスの詐称に対して何の対策も行っていないが、ファイアウォールを使うことにより、外部から送られてきたパケットの送信元のアドレスが内部のアドレスなら拒否することができる。また、内部から送られてきたパケットの場合でもゲートウェイ等でチェックすることが可能である。

## 6 実験

我々は行動追跡モデルに基づくセキュリティ機構を実現するシステム Compacto を Linux 2.2.11 上に実装した。分散システムにおいてはホストに既に侵入されている可能性を考えると各ホストの OS カーネル以外は信頼できないため、全ての機能をカーネル内に実装している。

我々の提案するセキュリティ機構がシステムに与える影響を調べるために、二種類の実験を行った。一つはホスト間で汚染を伝播させるのにかかるオーバヘッドを測定することであり、もう一つはシステムコールレベルでポリシーのチェックを行うのにかかるオーバヘッドを測定することである。この実験は 100Mbps のイーサネットで接続された二台の PC(Pentium II/400MHz, メモリ 128MB) を使って行い、ネットワーク・インタフェース・カードには 3COM Fast EtherLink を用いた。

| | レイテンシ ($\mu s$) | | スループット |
|---|---|---|---|
| | $1B$ | $1430B$ | ($Mbps$) |
| 伝播あり | 147.2 | 553.8 | 67.04 |
| 伝播なし | 141.9 | 548.6 | 67.41 |

表 1: 汚染伝播の有無による TCP/IP の性能の変化

### 6.1 汚染を伝播させるオーバヘッド

現在の実装では異なるホスト間で汚染を伝播させるために、TCP オプションを使って汚染情報をピギー・バックしている。汚染を伝播するために必要な TCP オプションのサイズは 4 バイトである。汚染を伝播させるためのオーバヘッドには、パケットサイズが増加することによるオーバヘッド、TCP オプションを処理するオーバヘッド、汚染情報を更新するオーバヘッドが含まれる。

汚染を伝播させるのにかかるオーバヘッドを調べるために、netperf というベンチマーク・プログラムを用いて TCP/IP のレイテンシとスループットを測定した。レイテンシについては、パケットのデータサイズが 1 バイトの時と 1430 バイトの時とで往復のレイテンシを測定した。さらに、比較のために汚染情報をピギー・バックしない場合についてもそれぞれ測定を行った。この実験結果を表 1に示す。

TCP オプションを用いて汚染情報を伝播させることにより、レイテンシにおいて最大で 3.7%のオーバヘッドを被っている。ただしこれはデータサイズが 1 バイトと非常に小さい時であり、平均的なデータサイズではオーバヘッドはもっと小さくなる。実際、一つの TCP パケットで送ることができる最大のデータサイズ (1430 バイト) でのオーバヘッドは 0.9%である。一方、スループットに関しては 0.6%と十分小さなオーバヘッドで済んでいる。

### 6.2 ポリシーチェックのオーバヘッド

我々の提案するセキュリティ機構では、システムコールが呼ばれる度にポリシーに記述されたルールをチェックすることでアクセス制限を行う。一般に、ルールが増えれば増えるほどこのオーバヘッドは大きくなる。しかし、適用されるべきルールが全くない場合でも、ルールの有無をチェックするために多少のオーバヘッドがかかってしまう。

そこで、全くアクセス制限を行わないプロセスに対して、どの程度余分なオーバヘッドがかかるかを調べる実験を行った。この実験では、getpid システムコールを実行する際にポリシーチェックを行う場合と行わない場合について、それぞれの実行時間を測定した。その実験結果は、ポリシーチェックを行った場合には $0.81\mu s$、行わなかった場合には $0.77\mu s$ となり、5%程度のオーバヘッドがかかっている。しかしながら、getpid はほとんど何も行わないシステムコールなので、平均的なシステムコールについてはこのオーバヘッドは相対的に小さくなると考えられる。またプロセス全体で見た場合には、getpid システムコールを毎秒 100 万回以上呼んだ時に 5%のオーバヘッドがかかるということであり、実際のプロセスではこのようなことはまずありえない。

## 7 関連研究

Java のセキュリティ機構 [5] は、アクセス制限を行う際にメソッドの呼び出し元を追跡していき、全てのメソッドにアクセス権が与えられている時だけアクセスを許可する。我々の提案しているセキュリティ機構は、アクセス制限のためにプロセスの依存関係を追跡していくという点で Java のセキュリティ機構に似ている。しかしながら、Java ではメソッドの呼び出し元という形で明確に検査する対象が定まるのに対し、我々の対象としている実際のシステムではどの依存関係が問題になるかを判断するのは非常に難しい。その点で、実際のシステムで Java のセキュリティ機構のようなものを実現するのはそれほど簡単ではない。

Deeds [3] では、プログラムがどのリソースにアクセスしたかという履歴によってアクセスを制限することができる。例えば、ソケットをオープンしていなければファイルをオープンするのを許すなどである。このようなシステムでは一般にログが膨大になりがちだが、Deeds では基本的にはアクセスのログを残さずに、ソケットをオープンしたかどうかなどの単純なパラメータだけを記録するようにしている。我々の提案しているセキュリティ機構では、汚染の伝播のためにリソースに汚染情報を保持させており、これをリソースに対するアクセスの履歴と考えることもできる。

侵入検知システムの中には、行動追跡モデルに基づくセキュリティを実現することが可能なものが存在する。IDA [9] では、疑わしいアクセスがなされたらリモートログイン元の追跡を行う。しかし、必要になった時点で追跡を行うために、アクセスの正当性を判断するのに時間がかかる。また、セキュリティホールを攻撃される可能性も考えると、各ホストで全てのネットワーク通信に関する情報を記録しておかなければならず、ログが膨大になる可能性がある。それに対して、我々のシステムは侵入経路の正確な検出ではなくアクセス制限を目的としており、追跡に時間がかからず、保持しておくべきデータも少なく済んでいる。

GrIDS [8] ではホストをノードとし、ホスト間の通信をエッジとするグラフを作り、システムへの攻撃を検出する。このシステムでは組織の階層構造に基づいてグラフを簡易化することで、大規模なシステムにも対応できるようにしている。だたし、GrIDS ではホストからホストへ自動的に被害が拡がっていくワーム攻撃を検出するためにグラフを用いており、アクセス制限を目的としている我々のシステムとは異なる。

## 8 まとめと今後の課題

分散システムにおいて行動追跡モデルに基づくセキュリティシステムを実現するために、プロセスの依存関係を利用したセキュリティ機構を提案した。このセキュリティ機構はプロセスの依存関係に基づいて、プロセスが周囲のプロセスに与えた影響を分散システム内部において追跡し、その情報をアクセス制限に利用する。この機構により、万一外部から侵入されたとしても分散システム内部のホストを守り続けることができ、内部犯による不正アクセスもある程度防ぐことが可能になる。我々はこのセキュリティ機構を実現するシステム Compacto を実装し、汚染という概念を導入することによって効率よく行動追跡モデルを実現することができている。

現在の汚染の実装ではホストを識別するのに IP

アドレスを用いているので、IP アドレスの正当性をチェックできるようにする必要がある。IP アドレスを詐称されると、汚染されるべきプロセスが汚染されなかったために攻撃を受けて侵入されたり、意図的に汚染の範囲を拡げられてサービスの提供が阻害される恐れがある。

また、プロセスの認証情報も汚染情報とともに伝播させることが考えられる。現在のシステムではプロセスの認証情報は自動的には通信相手に伝わらず、認証情報を利用するためにはアプリケーションが明示的に送ってやる必要がある。認証情報を自動的に伝播させることにより、よりきめ細かなアクセス制御や汚染の伝播の制御が可能になる。

## 参考文献


[1] Atkinson, R.: Security Architecture for the Internet Protocol, RFC 1825 (1995).

[2] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, in *Proceedings of the 7th USENIX Security Symposium*, pp. 63–78 (1998).

[3] Edjlali, G., Acharya, A. and Chaudhary, V.: History-based Access Control for Mobile Code, in *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pp. 38–48 (1998).

[4] Freier, A. O., Karlton, P. and Kocher, P. C.: The SSL Protocol Version 3.0, Internet Draft (1996).

[5] Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, in *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1997).

[6] Haller, N.: The S/KEY One-Time Password System, in *Proceedings of the ISOC Symposium on Network and Distributed System Security* (1994).

[7] Haller, N., Metz, C., Nesser, P. and Straw, M.: A One-Time Password System, RFC 2289 (1998).

[8] Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R. and Zerkle, D.: GrIDS – A Graph Based Intrusion Detection System for Large Networks, in *Proceedings of the 19th National Information Systems Security Conference* (1996).

[9] 浅香緑：モーバイルエージェントによる侵入検出システムのための情報収集方式, ソフトウェアエージェントとその応用シンポジウム論文集, pp. 64–69 (1997).