平成20年度　　卒業研究論文

侵入防止システムにおける動作規則の保護機構の開発

指導教官
齋藤　彰一　准教授

名古屋工業大学　情報工学科
平成17年度入学　17115134番

古屋　雄介

目 次

第1章

はじめに

近年、コンピュータネットワークの発達と共に悪意のあるユーザによるコンピュータへの不正アクセスが多発している。コンピュータが不正アクセスされ、コンピュータ内部に保管されている機密情報を漏洩されたり、他のコンピュータに対する更なる不正アクセスのための踏み台にされるなどといった被害が多数報告されている。

一般的に不正アクセスはシステム内に存在するセキュリティーホールを利用して行われる。セキュリティホールとはシステム内に存在する脆弱性のことである。これにより悪意のあるユーザにより意図しない操作をされてしまう危険性がある。不正アクセスの種類は様々だが、その中でもコンピュータプログラムに存在するセキュリティホールを利用するものが顕著である。プログラムはある程度知識のあるユーザならば容易に作成することができるが、その反面正しい例外処理が施されていない場合、セキュリティホールが入りやすいためである。プログラム作成段階においてセキュリティホールを最小限にすることを意識したセキュアプログラミングが行われている。しかし、セキュアプログラミングを行っていてもセキュリティホールが完全に排除される訳ではなく、セキュリティホールを持たないシステムを構築することは困難である。

そこで、システム内の異常を検知し、不正アクセスを防止する侵入防止システムが注目されている。一般的に侵入防止システムはプログラムの正常な動作を予め定義した動作規則に基づいてプログラムの異常を検知し、侵入を防止する。しかし、この動作規則が改竄されると侵入防止システムが機能しなくなるという問題点がある。

本論文では侵入防止システムの動作規則を改竄から保護する手法を提案する。本提

案手法は動作規則が存在する場所によって改竄を検知または防止するものである。ファイル中の動作規則に関してはデジタル署名を付加することで改竄の検知を行い、メモリ中の動作規則に関しては各改竄を防止する。

2章では不正アクセスによる被害と代表的な攻撃手法及び既存の対策を述べ、3章では提案手法について述べ、4章では提案手法の実装方法について述べる。5章では実験と考察について述べ、6章で結論を述べる。

第2章

不正アクセスとその防止手法

本章では不正アクセスと不正アクセスによる被害、不正アクセスを行う上でよく利用されるバッファ・オーバーフロー、また既存の対策について述べる。

2.1 不正アクセス

不正アクセスとはアクセス制御機能を持つシステムにアクセス権限を持たない者が不正にアクセスをしたり、試みることである。これには、自分以外の者が持つ正規のパスワードを使用してアクセスする場合とシステムのセキュリティホールを利用してアクセスする場合がある。

インターネットの普及により多くのコンピュータが相互に通信可能な環境が構築され、不正アクセスの件数は増加した。そのため 1999 年、本国において不正アクセス行為の禁止等に関する法律 (不正アクセス禁止法) が制定されるなど様々な対策が講じられているが、未だに多くの不正アクセスが報告されているのが現状である。

2.2 不正アクセスによる被害

IPA 情報処理推進機構へ報告された不正アクセス件数 [1] は 2000 年から 2001 年にかけて約 4 倍に急増し、その後、同じような件数で推移している。代表的な不正アクセスの手法として SQL インジェクションがある。これはデータベースアプリケーションに不正なクエリーを与える行為であり、一般的に Web アプリケーションのセキュリティ

ホールを利用して行われる。企業の顧客管理システムがSQLインジェクションの被害に遭い、顧客情報が流出する事件が多発している。

一方、不正アクセスによる実被害件数は減少傾向にある。これは近年企業が不正アクセスにより深刻な被害が受け、企業側の意識が変わってきているものと考えられる。しかし、個人ユーザに関しては不正アクセスへの意識が低く、多数の被害が報告されている。不正アクセスはそれ自体を阻止することは難しいが、システムを堅牢にすることにより被害を最小限に抑えることができる。

2.3 バッファオーバーフロー

不正アクセスはコンピュータプログラムのセキュリティホールを利用したものが多く、その中でもバッファオーバーフローと呼ばれるセキュリティホールを利用したものが顕著である。

現在最も普及しているノイマン型コンピュータによるプログラムではバッファと呼ばれる記憶領域が用いられており、一般的にメモリ上に確保される。人為的な不注意によってバッファが意に反して溢れたり、悪意のある者がプログラムの脆弱性を利用して意図的に溢れを発生させることがある。これをバッファオーバーフローと呼ぶ。バッファオーバーフローが発生すると当然プログラムが意図したように動作せず、不都合が生じる。

また、バッファオーバーフローはC言語を用いてプログラムを作成した場合に発生しやすい。これはC言語が元々オペレーティングシステムを記述するための低級言語であり、さらに古い言語なのでチェック機構が不十分なためである。

2.4 バッファオーバーフロー攻撃

ノイマン型コンピュータを用いてプログラムを作成する場合、プログラムコードとデータが同一のメモリ空間上に配置される。故に、バッファオーバーフローの発生に伴い、コードや制御情報が改竄され、プログラムの制御を奪うことが可能である。これをバッファオーバーフロー攻撃と呼ぶ。

ここでは 2 種類のバッファオーバーフロー攻撃についてバッファオーバーフローが発生しやすい言語である C 言語を例に説明する。

2.4.1 スタックオーバーフロー攻撃

C 言語ではプログラムモジュールは関数と呼ばれる単位で作成される。各関数内で用いられるバッファは LIFO(Last In First Out) のアルゴリズム特性を持つため、コールスタックと呼ばれる領域に配置される。このコールスタック上にはプログラムが使用するデータだけではなく、関数の戻りアドレスなどの制御情報も配置される。関数の脆弱性を利用して意図的にスタックオーバーフローを起こし、関数の戻りアドレスを書換えることにより制御を奪うことが可能である。これをスタックオーバーフロー攻撃と呼ぶ。

2.4.2 ヒープオーバーフロー攻撃

バッファオーバーフロー攻撃はスタックオーバーフロー以外によっても引き起こされ、C 言語において malloc 関数や calloc 関数によって確保されるバッファであるヒープを狙った攻撃手法が存在する。

関数の脆弱性を利用してヒープをオーバーフローさせることによりヒープ上の制御情報を書換え、任意のアドレスに任意のデータを書き込むといったことが可能であり、またプログラムの制御を奪うことも可能となる。例えば、C 言語におけるヒープ管理は一般的にライブラリによって管理されているが、このライブラリに脆弱性がある場合、ヒープオーバーフロー攻撃を受ける可能性がある。

2.5 バッファオーバーフロー攻撃の防止手法

バッファオーバーフロー攻撃を防止するために様々な手法が考案されている。安全なプログラミングを意識したセキュアプログラミングやデバッグによる検査、OS レベルやアーキテクチャレベルの対策などがある。それらのいくつかについて述べる。

2.5.1　ソースコードの静的解析

最も初期に比較的容易に行うことができる対策である。これには目視による検査、ツールによる検査がある。

- 目視による検査

  例えばC言語においてscanfやstrcpyなどの関数は引数に与えられたデータの大きさをチェックせずバッファに格納しようとするため、バッファ・オーバーフローが発生する可能性があり、脆弱性がある。

  また、制御情報のひとつである関数ポインタが書換えられることによって制御が奪われるため、関数ポインタを多用している場合も注意が必要である。ソースコード作成段階でこれらの脆弱性を排除することが可能である。

- ツールによる検査

  検査しようとするソースコードが目視できないほど大きい場合にはツールによる検査が有効である。以下に代表的なツールを２つ挙げる。

  - Flawfinder[3]

    C言語における脆弱性のあるライブラリ関数を使用していた場合、これを警告する。

  - Coverity Prevent[4]

    C/C++言語においてこのツールの元でビルドすることにより、コンパイラの構文解析結果を収集しプログラムのロジックを静的に解析する。

2.5.2　安全なライブラリ

セキュリティホールは自ら作成したコードだけではなく使用するライブラリに潜んでいることがある。より安全なライブラリとして米国のBell研究所が作成したLibsafe[5]がある。Libsafeはバッファオーバーフローなどの対策が十分にされていないライブラ

リをバッファオーバーフローが発生しにくいように実装をし直したライブラリであり、もし発生した場合には警告をする。

2.5.3 動的デバッグ

静的な検査では取り除くことのできるセキュリティホールの数に限界がある。作成したプログラムを実行させて動的にデバッグすることでバッファ・オーバーフローを検出する。代表的なツールに Valgrind[6] がある。これは Linux 専用のツールであり、ヒープオーバーフローの検出に適している。

2.5.4 実行時検査

スタックオーバーフロー攻撃はスタックを溢れさせ、スタック上の制御情報である関数の戻り値を書き換えることにより行われる。スタックフレーム作成時に関数の戻りアドレスの手前に検査用データを挿入する。関数が戻るときに検査用データが書き換わっているかを検査することによりスタックオーバーフローの発生の有無を調べ、攻撃を回避することができる。標準の C コンパイラの改良版である StackGuard[7] や GCC オプションである-fstack-protector により行うことができる。

2.5.5 ランダム化

バッファオーバーフロー攻撃によりプログラムの制御を奪う場合、攻撃者は自らが挿入したシェルコードが配置されるメモリアドレスを知っている必要がある。Linux カーネルパッチである ExecShield[8] を用いることでスタックやヒープの配置アドレスがランダム化されるため攻撃が困難となる。

2.5.6 データ実行防止

ノイマン型コンピュータではプログラムコードとデータが同一のメモリ空間上に配置されており、従来の CPU では命令実行に関して両者を区別していない。そのため

バッファオーバーフロー攻撃により攻撃者が埋め込んだシェルコードに制御を移すことが可能となる。データ実行防止機能を持つCPUやオペレーティングシステムを用いることで防ぐことができる。しかし、return to libc攻撃を行われた場合、制御が奪われる可能性がある。

return to libc攻撃とは攻撃者が用意したシェルコードを実行させるのではなく最初からメモリ空間に配置されているライブラリを用いて制御を奪う攻撃である。C言語においてはsystem関数を利用することで可能となる。

2.6 侵入防止システム

侵入防止システム(IPS:Intrusion Prevension System)とはネットワークやコンピュータへの不正侵入を防止するシステムであり、近年注目を浴びている。この背景としては、システムのセキュリティホールを無くす努力がされてきたがセキュリティホールを完全に排除することは困難であり、システムに発生した異常を能動的に検知、防止する機構が必要であると考えられてきたことが挙げられる。

ネットワークへの侵入を防止する侵入防止システムをネットワーク型侵入防止システム(Network-based IPS)、コンピュータへの侵入を防止する侵入防止システムをホスト型侵入防止システム(Host-based IPS)と呼ぶ。また、侵入を検知する方式として不正検知方式と異常検知方式がある。

不正検知方式では観測された行為と予め登録しておいたシステムに対する不正行為とを照合し、一致した場合は侵入行為とする。そのため誤検知は無いが登録されていない不正行為をされた場合は検知できない。異常検知方式ではあらかじめシステムの正常な動作を定義しておき、その動作規則に反する動作を異常とする。動作規則を正確に定義するほど検知率は上がるが検知に要するオーバーヘッドも増加する。

2.6.1 Network-based IPS

近年、インターネットの発達とともに様々な不正アクセスが行われている。ネットワークを機能させなくするDos攻撃や、コンピュータへの侵入のために事前に行われ

るポートスキャンなどである。ネットワーク型侵入防止システムはネットワーク上のトラフィックを監視し、これらの異常を検知し侵入を防止する機能をもっている。

不正検知方式のネットワーク型侵入防止システムではあらかじめ不正アクセスにおいて使用されるデータを登録しておき、トラフィック上を流れるデータと照合し、一致した場合データの侵入を防止する。一方、異常検知方式のネットワーク型侵入防止システムではネットワーク上のトラフィックを監視し、通信の統計を算出し、正常な状態を定義する。これに当てはまらない場合は異常とみなし侵入を防止する。

2.6.2 Host-based IPS

ホストへの侵入とはそのホスト上で動いているプロセスを乗っ取り制御を奪うことである。ホスト上で動いているプロセスが乗っ取られてしまうとそのプロセスのもつ権限の範囲内であらゆることができる。具体的にはWebページの改竄や顧客情報の流出などの被害を受けたり、他のホストへの更なる不正アクセスのための踏み台にされる可能性がある。

ホスト型侵入防止システムはコンピュータへの侵入を検知し防止する。ネットワーク型侵入防止システムがネットワークに1つ設置すれば良いのに対しホスト型侵入防止システムは各コンピュータにそれぞれ設置する必要がある。

第3章

既存の防止手法と提案手法

本章では前章で述べた不正アクセス防止手法である侵入防止システムのうち異常検知に基づく Host-based IPS についてその動作原理を述べる。その上でセキュリティーホールとなり得る問題点を挙げ、その解決手法を提案する。

3.1 Host-based IPS における防止手法

異常検知に基づく侵入防止システムは監視対象となるプログラムの正常な動作を定義した動作規則を元に侵入を検知、防止する。このとき、動作規則の作成には学習による作成と静的解析による作成がある。尚、本章で紹介する手法 [2] ではプログラムが発行するシステムコールを元に動作規則を作成している。また、OS カーネルは乗っ取られないという前提のもとで侵入防止システムは OS カーネルに組み込まれている。

3.1.1 学習による動作規則の作成

学習による動作規則の作成では監視対象プログラムを一定期間安全な環境で動作させ、その間に発行されたシステムコールから動作規則を作成する。この手法の場合、監視対象プログラムに特徴的な動作規則を作成することができるので動作規則の肥大化を防ぐことができるが、システムコールの発行パターンを全て網羅していないので誤検知も発生することになる。また、動作規則の作成にも時間がかかる。

3.1.2 静的解析による動作規則の作成

静的解析による動作規則の作成では監視対象プログラムのソースファイルや実行ファイル、または逆アセンブルしたアセンブリファイルを解析し動作規則を作成する。学習による動作規則の作成とは違い、プログラムの動作をすべて網羅することができるので誤検知は発生しない。しかし動作規則の肥大化や、それに伴う監視オーバーヘッドも増加する。さらにプログラムごとに動作規則の作成が必要となる。

3.1.3 問題点

プログラム監視中、動作規則はカーネル空間に配置されるのが理想的だが一般的にカーネル空間の大きさには限りがあるため各監視対象プロセスのメモリ空間内に配置されており、無防備な状態である。また、静的解析により作成された動作規則は上記に加えてプログラムが実行されてメモリ空間に配置されるまでも無防備な状態である。

ホスト型侵入防止システムではプログラムを監視する上で動作規則に依存しているため、正しい動作規則をもとに監視をしなければ機能しない。動作規則が改竄されると、侵入防止システムが機能しなくなるだけではなく、プロセスが乗っ取られるという問題点がある。つまり、動作規則の改竄は侵入防止システムへの攻撃を意味する。

3.2 提案手法

静的解析により作成された動作規則に基づく侵入防止システムの問題点は学習により作成された動作規則に基づく侵入防止システムの問題点を包括している。よって、本節では動作規則の改竄を防ぐことにより、静的解析により作成される動作規則に基づく侵入防止システム [2] が攻撃されるという問題の解決手法を提案する。また、本論文では動作規則の改竄を防ぐことを保護と呼ぶ。

プログラム実行前の動作規則とプログラム監視時に参照される各プロセスのメモリ空間上に配置された動作規則の保護手法についてそれぞれ説明する。また、具体的な手法は実装環境に依存するため本節では概要について述べ、詳細については 4 章にて述べる。

3.2.1　プログラムファイル中の動作規則の保護

監視対象プログラムの動作規則はプログラム実行時に必要となるものであるためプログラムファイル中に一緒に格納する。動作規則が格納されたプログラムファイルを保護することになり、プログラムへの改竄に対しても有効となる。

動作規則への改竄主体は多く、各々に対応し防止することは困難である。そこで、本節ではプログラムファイルにデジタル署名を付加することにより改竄を検知する手法を提案する。

プログラムファイルフォーマットの拡張

プログラムファイルへの署名にともない、プログラムファイルフォーマットを拡張し、署名のファイル内オフセット及び動作規則のロードされるアドレスを付加する。署名のファイル内オフセットは署名の検証時に必要となり、動作規則のロードアドレスはメモリ中の配置された動作規則の保護に必要となる。

プログラムファイルへの署名

プログラムファイルが作成されてから実行されるまでの間には様々な改竄が考えられる。作成環境における改竄や実行環境への移動中における改竄など、改竄場所や改竄主体は多種多様でり、すべての改竄に対応し防ぐことは困難であり現実的ではない。そこでプログラムファイルにデジタル署名を付加することによりプログラムファイルへの改竄を検知する手法を提案する。デジタル署名により得られる完全性によりプログラムファイルが改竄されていないことが保証されるだけでなく、真正性によりプログラム作成者が保証される。なお、ファイルへの署名時はファイルが改竄される恐れのない安全な場所でプログラムに署名する。

署名の検証

署名の検証は信頼できる者が行わなければならない。プログラム実行時に、カーネルに組み込まれた侵入防止システムによりデジタル署名の検証が行われる。デジタル署名が正しいものであればプログラムファイルは改竄されていないと判断し、プログラムを実行させる。

3.2.2　メモリ中の動作規則の保護

プログラム実行時に署名の検証がされ、改竄されていないことが保証されても実行中に改竄されることも考えられる。静電気による改竄などを除けばメモリ中の動作規則への改竄主体は限られている。そのため各々の改竄に対する機能を追加することで改竄の検知ではなく防止することが可能となる。

一般的な実行環境はノイマン型アーキテクチャに基づいている。プログラムはメモリ中に配置され、それを CPU が逐次処理をしていく。動作規則をソフトウェア的に改竄する場合、それは全て CPU により行われることになる。そのため CPU による動作規則の書き換えを制限する。

一般的な実行環境はオペレーティングシステムによりメモリが管理されており、仮想メモリ管理が行われている。仮想メモリ管理とは各プロセスに固有のメモリ空間を割り当てる仕組みのことで、メモリ空間をページと呼ばれる単位で管理している。また、CPU には仮想メモリ管理のために MMU と呼ばれるメモリ管理ユニットが搭載されている。CPU によるメモリアクセス時に MMU により当該ページの仮想アドレスを実アドレスに変換している。この仮想アドレスと実アドレスの対応表をページテーブルと呼び、これには各ページへのアクセス制御フラグも含まれている。フラグによりアクセスが不許可の設定になっている場合、CPU による該当ページへのアクセスはできない。

そこで監視対象プロセスのページテーブルのアクセス制御フラグを設定することで、動作規則への改竄を防ぐ。また、一般的なオペレーティングシステムではカーネルとプロセスに異なる動作権限を与えており、それぞれの動作モードをカーネルモードと

ユーザモードと言う。ページテーブルのアクセス制御フラグも動作モードに応じて設定が可能である。

通常、ユーザモードにおけるフラグの設定はプログラム実行時にプログラムフォーマット中で指定する。予めこのフラグを設定することでユーザモードでのアクセスを禁止することができる。しかし、一般的なオペレーティングシステムでは実行時にアクセス権を変更するシステムインタフェースを提供しており、これにより容易にアクセスが可能となる。また、オペレーティングシステムによってはデバッグ利用としてメモリを書き換えるシステムインタフェースを提供している場合がある。この場合、ユーザモードにおけるアクセスが禁止されていてもメモリの書き換えはカーネルモードで行われるためアクセスが可能となる。上に挙げたシステムインタフェースを利用した場合の改竄に対応するため各システムインタフェースにより実行されるカーネルコードを変更することで、動作規則への改竄を行う場合はそれを防ぐ。

第4章

実装

本提案手法を Linux2.6 上に実装した。本章ではプログラム ファイル中の動作規則の保護とメモリ中の動作規則の保護についてそれぞれ実装方法を述べる。

4.1 プログラムファイル中の動作規則の保護

本節ではプログラムファイル中の動作規則の保護の実装について説明する。プログラムファイル中に動作規則を組み込む方法として既存のプログラムファイルを静的に変換する方法をとった。

まず、拡張対象となるプログラム ファイルフォーマットである ELF について述べ、続いて ELF の拡張方法、動作規則の付加方法、拡張後の ELF ファイルへの署名方法、Linux におけるプログラム の実行の流れ、プログラム 実行時における署名の検証方法について順に述べる。

4.1.1 ELF: Executable and Linking Format

ELF[9] とは Executable and Linking Format の略であり、Linux においてデファクトスタンダードなプログラム ファイルフォーマットである。UNIX において長い間使われてきたプログラム ファイルフォーマットである a.out に変わるものである。本項ではまず ELF の 4 種類の形式について述べ、次に ELF を構成する各ヘッダについて説明する。

ELF の形式

ELF は 4 種類のファイル形式を取りうる。以下にそれぞれについて述べる。

1. 再配置可能オブジェクト形式

   ソースファイルをコンパイルして得られる再配置可能なプログラム形式である。シンボルのアドレス解決はされておらず、リンカによって静的に結合される。

2. 共有オブジェクト形式

   プログラム実行時にダイナミックリンカにより動的にマッピングされる。各プログラム間で物理メモリが共有されるためメモリの節約になる。また、プログラムに静的にリンクされないのでディスクの節約にもなる。

3. 実行可能形式

   再配置可能オブジェクトファイルを静的にリンクして得られる実行可能なファイル形式であり、いわゆる実行ファイルである。シンボルのアドレス解決もされている。本研究ではこの形式を拡張する。

4. コア形式

   プログラムがエラーにより不正終了した際、メモリやレジスタの内容が記録される形式である。

ELF のヘッダ

以下で ELF を構成する各ヘッダについて述べる。

1. ELF ヘッダ

   以下に説明するプログラムヘッダやセクションヘッダの位置などの ELF ファイル自体の情報を持つ。elf.h による構造体定義を図 4.1 に示す。

2. プログラムヘッダ

   プログラム実行時に必要となる情報への参照を持つ。プログラムヘッダで参照される部分をセグメントと呼ぶ。ローダやダイナミックリンカにより参照される。elf.h における構造体定義を図 4.2 に示す。

3. セクションヘッダ

プログラム作成段階やデバッグ時に必要となる情報への参照を持つ。セクションヘッダで参照される部分をセクションと呼ぶ。コンパイラやリンカにより参照される。elf.h における構造体定義を図 4.3 に示す。

```
typedef struct {
  unsigned char e_ident[EI_NIDENT]; //マジックナンバーなど
  uint16_t      e_type;             //ファイル形式
  uint16_t      e_machine;          //必要とされるアーキテクチャ
  uint32_t      e_version;          //ファイルのバージョン
  ElfN_Addr     e_entry;            //プログラムのエントリーポイント
  ElfN_Off      e_phoff;            //プログラムヘッダテーブルのオフセット
  ElfN_Off      e_shoff;            //セクションヘッダテーブルのオフセット
  uint32_t      e_flags;            //プロセッサ固有のフラグ
  uint16_t      e_ehsize;           //ELF ヘッダサイズ
  uint16_t      e_phentsize;        //プログラムヘッダサイズ
  uint16_t      e_phnum;            //プログラムヘッダ数
  uint16_t      e_shentsize;        //セクションヘッダサイズ
  uint16_t      e_shnum;            //セクションヘッダ数
  uint16_t      e_shstrndx;         //セクション名文字列テーブルのインデッ
クス
} ElfN_Ehdr;
```

図 4.1: ELF ヘッダ

```
typedef struct {
  uint32_t   p_type;     //セグメントタイプ
  Elf32_Off  p_offset;   //セグメントのファイルオフセット
  Elf32_Addr p_vaddr;    //セグメントが配置される仮想アドレス
  Elf32_Addr p_paddr;    //セグメントが配置される物理アドレス
  uint32_t   p_filesz;   //セグメントのファイルイメージのバイト数
  uint32_t   p_memsz;    //セグメントのメモリイメージのバイト数
  uint32_t   p_flags;    //セグメントへのアクセス権フラグ
  uint32_t   p_align;    //セグメントのアライメント
} Elf32_Phdr;
```

図 4.2: プログラムヘッダの構造体定義

```
typedef struct {
  uint32_t   sh_name;      //セクションヘッダ文字列テーブル内のインデックス
  uint32_t   sh_type;      //セクションのタイプ
  uint32_t   sh_flags;     //様々な属性フラグ
  Elf32_Addr sh_addr;      //配置されるメモリアドレス
  Elf32_Off  sh_offset;    //セクションのファイルオフセット
  uint32_t   sh_size;      //セクションサイズ
  uint32_t   sh_link;      //セクションヘッダテーブルインデックスリンク
  uint32_t   sh_info;      //追加情報
  uint32_t   sh_addralign; //セクションのアライメント
  uint32_t   sh_entsize;   //セクション内のエントリサイズ
} Elf32_Shdr;
```

図 4.3: セクションヘッダの構造体定義

4.1.2 ELF の拡張

ELF のヘッダのうち ELF ヘッダを拡張し、ELF ヘッダの末尾に動作規則のロードアドレスと署名のファイル内オフセットを格納する。拡張した様子を図 4.4 に示す。

図 4.4: ELF ヘッダの拡張

4.1.3 動作規則の付加

新たなプログラムヘッダを追加して動作規則をロードする。追加するプログラムヘッダが指すセグメントタイプは PT_LOAD である。PT_LOAD に設定した場合、当該セグメ

ントはプログラムヘッダの p_vaddr で指定されるアドレスからプログラムヘッダの p_memsz で指定されるサイズ分がマッピングされる。また、アクセス権フラグはアクセス不許可の設定にする。

4.1.4 拡張 ELF ファイルへの署名

拡張 ELF ファイルへの署名に用いたアルゴリズムと署名の流れを以下で述べる。

デジタル署名に用いたアルゴリズム

公開鍵暗号方式に RSA を使用した。RSA における鍵は OpenSSL により作成した。米国国立標準技術研究所 (NIST)[11] が 2010 年までに安全でない暗号技術の使用を停止することを発表し、RSA の場合、鍵長は 2048bit 以上を用いるとしたことにより鍵長は 2048bit とした。暗号化には OpenSSL の暗号化ライブラリを使用せず、付属の多倍長整数ライブラリを使用した。これは署名における暗号化方式と検証における復号方式を一致させるためである。

また、ハッシュ関数として SHA256 を使用した。これは近年、MD5 などのハッシュ関数への攻撃の成功例が報告されており、NIST もまた SHA2 以上を使用すると発表したことによる。実際のハッシュ値計算には GPG[10] を使用した。

署名の流れ

署名の流れは以下の通りである。

1. プログラムファイルのハッシュを計算
2. ハッシュデータを暗号化
3. 暗号化されたハッシュデータをファイルの末尾に付加

4.1.5　プログラム実行の流れ

Linux におけるプログラム実行の流れを述べる。一般的に新規プログラムを実行する場合、fork システムコールを発行しコピープロセスを作成した後に execve システムコールを発行してプロセスのアドレス空間を新規プログラムに書換える。execve システムコール以下の流れを図 4.5 に示す。

execve システムコールが発行されるとカーネルに制御が移る。カーネル内ではまず execve システムコールに対応する関数である `sys_execve` 関数が実行され、次に実際の処理を行う `do_execve` 関数が実行される。`do_execve` 関数では `search_binary_handler` 関数によりプログラムファイルフォーマットの特定と対応するローダが選択される。プログラムファイルフォーマットが ELF の場合は `load_elf_binary` 関数によりプログラムがメモリにロードされ実行される。

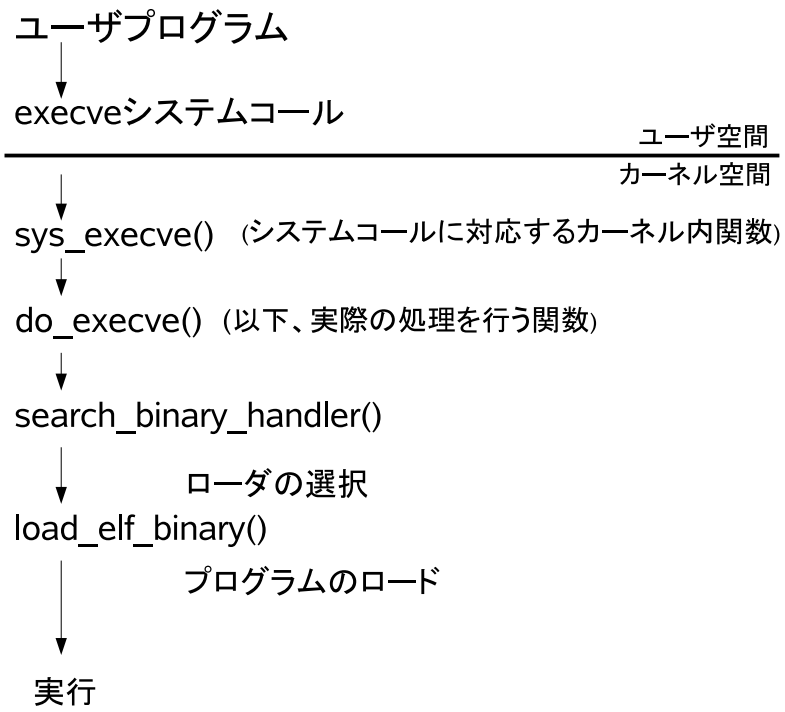

図 4.5: プログラム実行の流れ

4.1.6　署名の検証

前項で説明したプログラム実行の流れの中に署名の検証機能を追加する。追加した様子を図 4.6 に示す。`load_elf_binary` 関数によりプログラムがロードされる前に、追加した `sign_verify` 関数によってプログラムに付加された署名の検証を行う。検証によりプログラムが改竄されていないと判明した場合はプログラムをロードし、そうでなければエラーを返す。また、検証にはカーネルに RSA 暗号ライブラリである RSA algorithm patch[12] を用いた。

検証の流れ

検証の流れは以下の通りである。

1. プログラムファイルのハッシュを計算
2. 署名データを復号
3. プログラムファイルのハッシュデータと署名データの復号データを比較

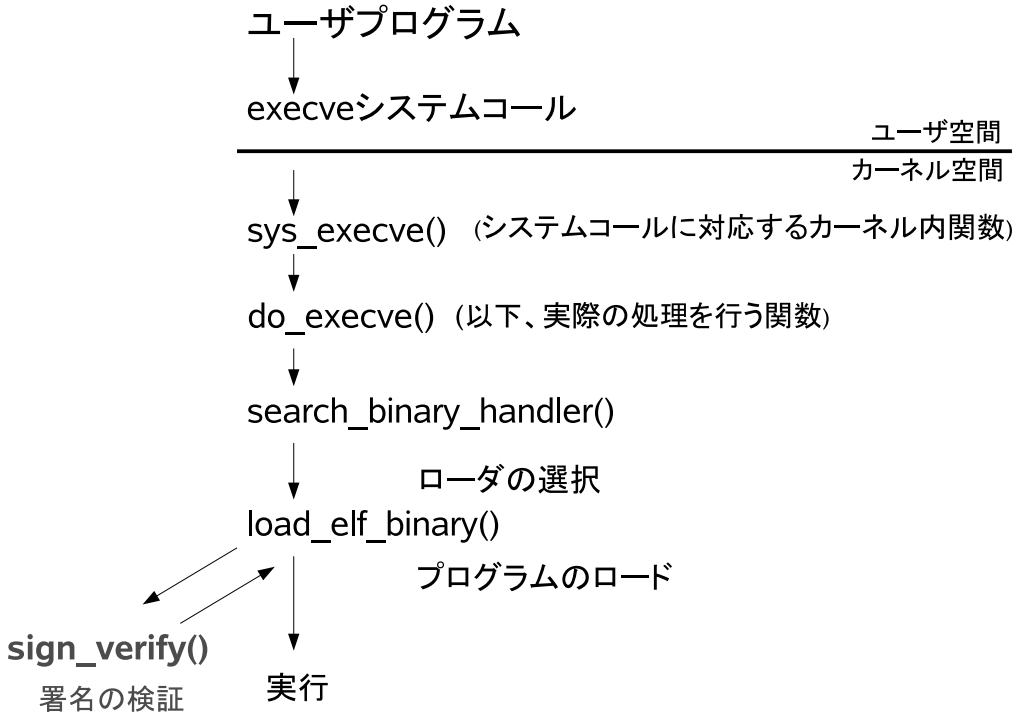

図 4.6: 署名検証機能の追加後のプログラム実行の流れ

4.2 メモリ中の動作規則の保護

本節ではメモリ中の動作規則の保護の実装について述べる。まず、各主体についてメモリ中の動作規則を改竄する可能性の有無を述べる。次に各改竄主体による改竄及びその防止手法について述べる。

4.2.1 各主体による改竄の可能性

カーネルはカーネルモードで動作するため監視対象プロセスの動作規則を書換えることができる。しかし、そもそもカーネルは侵入防止システムが動作する部分なのでカーネルは乗っ取られていない前提であるため改竄主体にはならない。

監視対象プロセスは動作規則の改竄を行うことはないが、乗っ取られることで動作

規則の改竄が起こる。

監視対象外プロセスは ptrace システムコールを利用すると改竄を行うことができる。Linux では仮想メモリ管理によって各プロセスには固有のメモリ空間が割り当てられているため他プロセスのメモリ空間にアクセスすることはできない。しかし、ptrace システムコールを実行するとアクセスが可能となる。

4.2.2 監視対象プロセスによる動作規則の改竄とその防止

以下で監視対象プロセスによる動作規則の改竄手法と防止手法について述べる。

mprotect システムコールを利用した改竄

監視対象プロセスが使用する mprotect システムコールを利用した改竄が考えられる。mprotect システムコールとはプロセスのメモリへのアクセス権を設定するシステムコールである。mprotect システムコールを発行する関数に脆弱性があった場合、その関数への引数が改竄され動作規則へのアクセスが可能となる。これにより新規にシステムコールを発行することなく動作規則の改竄が可能となる。

防止手法

システム内で発行される全ての mprotect システムコールを監視することで改竄を防止する。mprotect システムコールに対応する `sys_mprotect` 関数内で現在の PID と mprotect システムコールに渡された引数のチェックを行う。PID が監視対象プロセス ID であり、引数が動作規則へのアクセスを解除するものであれば mprotect システムコールをエラーで返す。

4.2.3 監視対象外プロセスによる動作規則の改竄とその防止

以下で監視対象外プロセスによる動作規則の改竄手法と防止手法について述べる。

ptrace システムコールによる改竄

ptrace システムコールはプロセスをトレースするためのデバッグシステムコールであり、他プロセスのメモリを書換えることもできる。また、ptrace 実行時の CPU 動作モードはカーネルモードであるため動作規則へのアクセス権に関係なく改竄が可能となる。

防止手法

システム内で発行される全ての ptrace システムコールを監視することで改竄を防止する。ptrace システムコールに対応する `sys_ptrace` 関数内で現在の PID と ptrace システムコールに渡された引数のチェックを行う。PID が監視対象プロセス ID であり、引数が動作規則への改竄を行うものであれば ptrace システムコールをエラーで返す。

第5章

実験

プログラム実行時における署名検証機能に関する実験を行った。以下に実験内容を述べる。

5.1 実験環境

実験環境は以下の通りである。

OS	Fedora Core5
kernel	2.6.17.8
CPU	Pentium4 3.4GHz
Memory	1GB

5.2 実験方法

execve システムコールが発行されてからプログラムが実行されるまでに要する時間を署名検証システム有りと無しの場合で比較した。また、署名検証機能内の各処理に要する時間も測定した。実験には 10K バイトと 100K バイトのサイズのプログラムを使用し、時間計測には do_gettimeofday 関数を用いた。

5.3 結果と考察

署名検証システムの有無での測定結果を表 5.2 に示す。システム無しに比べてシステム有りの場合、10K バイトのプログラムでは約 13 倍の時間を要し、100K バイトの

プログラムでは約 16 倍の時間を要した。どちらの場合も倍率は十数倍となっている。また、システム無しの場合においての測定結果はそれぞれ 317μsec、320μsec となっており、ほぼ同じ値になった。これは、プログラム実行時にプログラムの一部だけをプロセス空間にロードする Linux の仮想メモリ管理システムによる効率化のための結果だと考えられる。

次に署名検証システムの主な処理ごとの測定結果を表 5.2 に示す。主な処理とはハッシュ値を計算するためのプログラムファイル読込み処理、そのハッシュ値計算処理、署名の復号処理である。測定結果からプログラムファイル読込み処理とハッシュ値計算に要する時間はプログラムサイズに比例していることがわかる。復号処理に関しては署名サイズが固定値であるためプログラムサイズによらずほぼ同じ値になった。

表 5.1: システムの有無での測定結果

プログラムファイルサイズ	10KB	100KB
システム無し	317(μsec)	320(μsec)
システム有り	4607(μsec)	5877(μsec)

表 5.2: 処理別の測定結果

プログラムファイルサイズ	10KB	100KB
プログラムファイル読込み	7(μsec)	72(μsec)
ハッシュ値計算	128(μsec)	1271(μsec)
復号	4276(μsec)	4372(μsec)

第6章

まとめ

本研究では侵入防止システムが機能する上で必要となる動作規則の保護手法を提案した。保護手法はプログラムファイルに格納されている動作規則とメモリ上に配置されている動作規則を保護する。これを Linux カーネル 2.6 上に実装し、オーバーヘッドの測定を行った。動作規則が保護されることにより侵入防止システムが正常に動作することが保証された。なお、本研究では動作規則が実際に改竄された場合の検証が十分になされていないので今後検証する予定である。

謝辞

本研究のために、多大な御尽力を頂き、御指導を賜った名古屋工業大学の齋藤彰一准教授に深く感謝致します。また、本研究の際に多くの助言、協力をして頂いた齋藤研究室並びに松尾・津邑研究室の方々に深く感謝致します。

参考文献

[1] IPA: 情報処理推進機構 (IPA), http://www.ipa.go.jp/.

[2] 槙本裕司, 鶴田浩史, 齋藤彰一, 上原哲太郎, 松尾啓志: システムコールとライブラリ関数の監視による侵入防止システムの実現, 情報処理学会 研究報告, Vol. 2009, No. 6, pp. 3–9 (2009).

[3] David A. Wheeler: Flawfinder, http://www.dwheeler.com/flawfinder/.

[4] Coverity: Coverity Prevent Static Analysis for C/C++, http://www.coverity.com/html_ja/prod_prevent.html.

[5] A. Baratloo, N. Singh, T. Tsai: Protecting critical elements of stacks, http://www.research.avayalabs.com/.

[6] valgrind.org: Valgrind, http://valgrind.org/.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton: Automatic adaptive detection and prevention of buffer-overflow attacks, 7th USENIX Security Conference, pp. 63–78 (1998).

[8] redhat: Limiting buffer overflows with ExecShield, http://www.redhat.com/magazine/009jul05/features/execshield/.

[9] Webmasters: Manpage of ELF, http://www.linux.or.jp/JM/html/LDP_man-pages/man5/elf.5.html.

[10] Wemer Koch: GnuPG, http://www.gnupg.org/.

[11] 米国標準技術研究所: National Institute of Standards and Technologies, http://www.nist.gov/.

[12] LWN.net: crypto API:RSA algorithm patch (kernel version 2.6.20.1), http://lwn.net/Articles/226660/.