情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2009-OS-112 No.7
2009/8/6

侵入防止システムにおける動作規則保護機構の開発

古 屋 雄 介[†1]　齋 藤 彰 一[†1]　松 尾 啓 志[†1]

プログラムの脆弱性を利用した不正アクセスが深刻な問題になっている．これに対し，プログラムを監視し，異常を検知するシステムが提案されている．この異常検知システムを用いることで，プログラムに未知の脆弱性が含まれている場合でもプログラムへの侵入を防ぐことができる．本論文では槙本らが開発した異常検知に基づく侵入防止システムの問題点を挙げ，このシステムが依存する動作規則が改竄され得ることを示す．また，この攻撃に対する保護システムを提案する．実際に保護システムを開発し，評価を行った．

Implementation of Protecting Program Behiviour Rules for Intrusion Prevention System

Yusuke Furuya,[†1] Shoichi Saito[†1] and Hiroshi Matsuo[†1]

Illegal accesses by exploiting vulnerabilities of programs are serious problem. As a solution, some researchers proposed the system which monitors programs and detects its abnormal behaviours. The system prevents programs from being intruded even if programs include unknown vulnerabilities. This paper focuses on an intrusion prevention system Belem based on abnormal detection. we then propose a protection system against the attack which modifies program behiviour rules for Belem and have developed and evaluated the system.

†1 名古屋工業大学
Nagoya Institute Technology

1. はじめに

近年，インターネットの発達とともにコンピュータへの不正アクセスが多発している．不正アクセスにより企業の機密情報が漏洩する実被害も多数報告されている．[1] 攻撃者が行う不正アクセスの手法は 2 つに大別される．ひとつは正規のアクセス権を有しているユーザのパスワードを利用するものである．これにはソーシャルエンジニアリングや総当たり攻撃が利用される．もう一方はターゲットホスト上で動作するプログラムの脆弱性を利用し，そのプログラムの制御を奪うものである．前者はコンピュータ管理者の教育を行うことや，耐攻撃性のあるパスワードを設定することが対策となる．一方，後者を防ぐことは困難である．これはプログラムの脆弱性を事前にすべて見つけることはほぼ不可能なためである．これに対してプログラム作成段階におけるセキュアプログラミングや作成後におけるセキュリティパッチの適用が講じされているが，やはりすべての脆弱性を排除することはできない．

脆弱性に対してプログラムの実行を監視し，異常動作を検知可能なシステムが考案されている[2)–5)]．ここで異常動作とは正常なプログラム動作とは異なる動作のことである．異常動作を検知するためにあらかじめプログラムの正常な動作を作成する必要がある．本論文ではプログラムの正常な動作を**動作規則**と呼ぶ．動作規則の作成には静的解析による方法[2),3)]と動的解析による方法[4),5)] がある．静的解析による方法では実行ファイルを解析し，プログラム内の考えられるすべての遷移情報を元に実行状態オートマトンを作成する．一方，動的解析による方法では，プログラムを安全な環境で一定時間実行させて実行状態の遷移を調べることにより実行状態オートマトンを作成する．異常検知システムはプロセスが実行される間，常時監視を続け，プログラムの実行と動作規則とを照合する．動作規則に反する挙動が見られた場合，侵入されたと見なす．故に異常検知システムは侵入検知システム (IDS) とも呼ばれる．侵入検知システムの内，侵入を受けた後に何らかの処理を施し，侵入を防ぐ機能を有するものを侵入防止システム (IPS) と呼ぶ．

IDS と IPS はプログラムへの侵入を防ぐ上で有効的な機構となり得るが，これらのシステム自体が攻撃を受ける可能性がある．IDS と IPS が攻撃を受けた場合，これらのシステムが監視しているプログラムも侵入される可能性があり，その被害範囲は大きい．そこで本論文では槙本らが開発した Belem[6)] に関して，このシステムの問題点とこれに対する改善策を示す．また，開発した保護システムとその性能評価について述べる．

第 2 章では Belem について述べ，第 3 章で Belem への攻撃について説明する．第 4 章では攻撃に対する保護システムについて説明する．第 5 章では保護システムの実装につい

© 2009 Information Processing Society of Japan

て述べ，第 6 章では保護システムの考察を行う．第 7 章では関連研究について述べる．最後に第 8 章で本論文をまとめる．

2. 侵入防止システム (Belem)

本章では Belem の概要を述べる．まず，動作規則の作成方法について説明する．次に監視対象プログラムの実行状態と動作規則との照合について説明する．

2.1 動作規則の作成

実行プログラムの動作規則の作成は静的解析によって作成する．対象とする実行プログラムは C 言語で記述されていると仮定する．まず，実行ファイルを逆アセンブルし，アセンブリコードを取得する．次にこのアセンブリコード中に含まれる jmp，call などの遷移命令に基づき，各ユーザ関数 (共有ライブラリ関数以外の関数) ごとに内部における関数の実行状態オートマトンを作成する．各ユーザ関数の実行状態オートマトンにはユーザ関数呼び出しまたはライブラリ関数 (共有ライブラリ関数) 呼び出しの実行状態が含まれる．また，Belem では実行プログラムが使用するライブラリ関数内部の動作規則の作成は行わない．代わりに各ライブラリ関数が呼び出すシステムコールの集合であるシステムコール群を定義する．そして，各ユーザ関数に対応する実行状態オートマトンの集合と各ライブラリ関数に対応するシステムコール群の集合から**動作規則**を作成する．なお，動作規則は後述する libelem ファイルに組み込む．

2.2 実行状態と動作規則との照合

監視対象プログラムの実行状態の把握方法について述べる．次に実行状態と動作規則との照合タイミングについて述べ，最後に照合方法について述べる．

2.2.1 監視対象プログラムの実行状態の把握

プログラム実行中，ある関数内で関数が呼び出されると，呼び出し元の関数に戻ることができるようにローカル変数などと共に戻りアドレスがコールスタック上に積まれる．この関数呼び出しの度に積まれる戻りアドレスの情報からプログラムの実行状態を把握することができる．しかし，コールスタック上に積まれた戻りアドレス情報は呼び出された関数が戻る際に失われる．よって，コールスタックを調べる機会が多いほど，プログラムの実行状態をより詳細に把握することが可能となる．従来の研究[2)–4)] では，プログラムからのシステムコール発行を契機にコールスタックを調べる．これにはカーネルの修正のみが必要であり，実行プログラムの修正は必要ない．しかし，システムコールをほとんど発行しないプログラムからは得られる実行状態が少いため，検知率の低下につながる．そこで，Belem はプログラムのライブラリ関数呼び出しごとにコールスタックから戻りアドレスを調べ，履歴を作成する．一般にシステムコール呼び出しよりもライブラリ関数呼び出しの回数の方が多いため，より詳細にプログラムの実行状態を把握することができる．このためにライブラリ関数フック用ライブラリ (libelem) を作成し，プログラムにリンクする．監視対象プログラムが実行されると，対応する libelem ファイルも同時にユーザ空間にロードされる (図 1(1) 参照)．そして，監視対象プログラムがライブラリ関数を呼び出すと libelem がコールスタックからライブラリ関数の**呼び出し履歴**を生成する (図 1(2))．

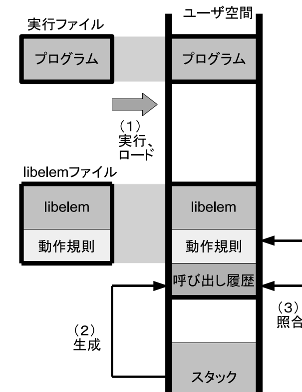

図 1 プログラム実行から照合までの流れ

2.2.2 照合タイミング

攻撃者はプログラムの制御を奪った後，一般にシステムコールを発行する．これはプログラムの挙動が外部に反映されるのはシステムコールによってのみであり，効果的な攻撃を行うにはシステムコールを呼び出す必要があるからである．よって，監視対象プロセスがシステムコールを発行すると，カーネルによりライブラリ関数の呼び出し履歴と動作規則の照合が行われる (図 1(3) 参照)．

2.2.3 照 合 方 法

監視対象プログラムがシステムコールを発行する度にカーネル内に組み込まれた侵入防止システムに制御が移る．ここで，前回システムコールが発行されてから今回システムコールが発行されるまでの呼び出し履歴に，動作規則に一致する遷移が存在するかを調べる．動

© 2009 Information Processing Society of Japan

Vol.2009-OS-112 No.7
2009/8/6

作規則に一致する遷移が存在し，さらに発行されたシステムコールが当該ライブラリ関数のシステムコール群に含まれる場合は，正常とみなして発行されたシステムコールを実行し監視対象プログラムに制御を戻す．一方，動作規則に一致するか否かに関わらず，発行されたシステムコールが当該システムコール群に含まれていない場合は侵入されたと見なし，プログラムの実行を中断する．なお，Belem は実行状態オートマトンにシステムコール呼び出しを含めていないため，ライブラリ関数以外から発行されたシステムコールの評価を行うことができない．故に Belem では実行プログラムはライブラリ関数によってのみシステムコール発行を行うことを仮定する．これはシェルコードによる直接のシステムコール発行を禁止することにもつながる．

ここで，照合を行う際に侵入防止システムは呼び出し履歴と動作規則にアクセスする必要がある．呼び出し履歴は libelem の.bss セクションとしてユーザ空間に存在する．呼び出し履歴をカーネル空間に配置した場合，セキュリティは向上するがアクセスするためにシステムコール発行を必要とする．そのため，ライブラリ関数呼び出しの度にシステムコールが発行されることになり負荷が高くなる．また，動作規則も libelem の.data セクションとしてユーザ空間に存在する．侵入防止システムが動作するカーネル空間に動作規則を配置すれば改竄されることはないが，カーネル仮想領域は各監視対象プログラムの動作規則を保持するに十分なサイズを有していない．そのため動作規則は各監視対象プロセスのユーザ空間に配置されている．

3. Belem への攻撃

前章で述べたように Belem は侵入防止を行うにあたり，照合のために動作規則と呼出し履歴に依存している．これらのデータが確かなものであるという前提のもとに成り立っていると言える．しかし，実際には動作規則と呼出し履歴はユーザ空間に配置されているため，改竄攻撃を受ける恐れがある．その場合，侵入防止システムは正常に検知作業ができなくなる．本章では Belem への攻撃の例を示し，問題点を述べる．

3.1 攻　撃　例

Belem に対する攻撃例を示す．まず，攻撃者は事前に監視対象プログラムに存在するオーバーフローの脆弱性がある関数を調べる．次に稼働している監視対象プロセスに対してシェルコードを送り込み，脆弱性のある関数内でオーバーフローを発生させる．オーバーフローが発生することによって監視対象プロセスの実行は攻撃者が送り込んだシェルコードに移る．一般にシェルコードは execve システムコールを利用してシェルを起動させる内容となっている．監視対象プロセスが root 権限で実行されていた場合，攻撃者は root 権限を持ったシェルを利用することができる．しかし，2 章で説明したように Belem ではライブラリ関数以外からのシステムコール呼び出しを禁止している．そのため，システムコールを直接呼ぶシェルコードの実行は Belem により検知可能である．これに対して，攻撃者はシステムコールがライブラリ関数から呼ばれているかのように模倣することで，Belem に検知されることなくシステムコールを発行できる．ここで攻撃方法は，監視対象プログラムが execve システムコールを発行するライブラリ関数を実行するか否か，つまり監視対象プログラムの実行状態オートマトンに execve システムコールを発行するライブラリ関数呼び出しが含まれるか否かで変わる．以下に攻撃方法を示す．

3.1.1 execve システムコールを発行するライブラリ関数を含む場合

監視対象プログラムの動作規則の当該ライブラリ関数に対応するシステムコール群に，execve システムコールが含まれる場合について述べる．この場合はオーバーフローを引き起こした関数から execve システムコールを発行する可能性のあるライブラリ関数呼び出しまで，実際に遷移したかのように呼び出し履歴を模倣することで Belem に検知されずに execve システムコールを発行できる．以下に，具体的な呼び出し履歴の改竄を伴う侵入方法を説明する．図 2 は監視対象プロセスが実行中の関数 D(関数 A,B,C を経て呼ばれる．) 内における動作規則を表している．Entry と Exit ノードはそれぞれ関数 D における実行の始まりと終わりを表している．Lib と書かれているノードはライブラリ関数呼び出しを表し，User と書かれているノードはユーザ関数呼び出しを表している．ノード間の遷移を矢印で表している．また，ライブラリ関数 B は execve システムコールを発行する可能性のある関数とする．

まず，攻撃を受けずに関数 D の実行が Entry から Exit まで進む場合を説明する．監視対象プロセスがユーザ関数 A，B，C を経て関数 D の実行に移る．次にライブラリ関数 A を実行しようとするとき，libelem により呼び出し履歴が生成される (図 3 の t1)．図 3 の四角はそれぞれ関数呼び出しを表しており，下部から順に積まれる．さらに，図 3 下部の時間軸は各時刻において保持される呼び出し履歴を表す．ここで，ライブラリ関数 A によりシステムコールが発行され，Belem により照合された後，監視対象プロセスに制御が戻り，libelem により呼び出し履歴は消去される．次にユーザ関数 F，G と実行を進め，ライブラリ関数 C を呼び，呼び出し履歴が生成される (図 4 の t2)．このとき関数 C はシステムコールを発行しない関数であるため，呼び出し履歴は消去されない．最後にライブラリ関数 B において呼び出し履歴が生成され (図 3 の t3)，t2 において生成された呼び出し履歴と合

ⓒ 2009 Information Processing Society of Japan

Vol.2009-OS-112 No.7
2009/8/6

わせて Belem により照合され，関数 D が終了する．

次に，ユーザ関数 D の実行中に攻撃された場合を説明する．ユーザ関数 D 内のユーザ関数 F にはオーバーフローの脆弱性があるとする．ユーザ関数 D の実行が Entry から始まり，ユーザ関数 F まで進み，ここでオーバーフローが発生してシェルコードに実行が移る．シェルコードは呼び出し履歴を改竄し，実行状態がユーザ関数 G からライブラリ関数 C，B の順に遷移したかのように模倣する (図 4 の t2)．そして execve システムコールを発行し，シェルを起動する．この例から分かるように呼び出し履歴の改竄を行うこで攻撃者は侵入することが可能となる．

3.1.2 execve システムコールを発行するライブラリ関数を含まない場合

監視対象プログラムの動作規則に，execve システムコールを含むシステムコール群が含まれていない場合について述べる．このような場合には，プログラムが呼び出す任意のライブラリ関数に対応するシステムコール群に execve システムコールを追加する．そして，現在の実行状態からそのライブラリ関数まで実際に遷移したかのように模倣することで Belem に検知されずに execve システムコールを発行できる．以下に具体的な動作規則と呼び出し履歴の改竄を伴う侵入方法を説明する．前項と同様に監視対象プロセスが実行中の関数 D 内における動作規則を図 2 に示すが，今回はライブラリ関数 B が execve システムコールを発行しない点が異なる．

今，ユーザ関数 F においてオーバーフローが起き，シェルコードに実行が移ったとする．攻撃者は次に遷移可能なシステムコール発行の可能性のあるライブラリ関数 Lib B まで遷移したかのように呼び出し履歴領域を改竄する (図 4 の t2)．さらに，本来 execve システムコールを発行しないライブラリ関数 B のシステムコール群を改竄し，execve システムコールを追加する．そして execve システムコールを発行し，シェルを起動する．この例から分かるように呼び出し履歴と動作規則の改竄を行うことで攻撃者は侵入することが可能となる．

3.2 動作規則と呼び出し履歴に関する問題

前節での Belem への攻撃例からわかるように Belem は動作規則と呼び出し履歴を改竄されることで攻撃を受け得る．動作規則と呼び出し履歴は監視対象プロセスのユーザ空間に配置されている．さらにそれらの配置されるアドレスは固定されている．そのため，監視対象プロセスのユーザ空間に注入されたシェルコードにより容易に改竄が可能となる．

また，監視対象プロセス外からも改竄を受け得る．Linux 等の一般的なオペレーティングシステムでは各プロセスに固有のアドレス空間を提供している．このような環境では各プロセスは他のプロセスのメモリ空間にアクセスできない．しかしこれらのオペレーティングシステムはまたデバッグインタフェースを備えている．例えば Linux において ptrace システムコールを用いると他プロセスのメモリ空間の読み書きができる．つまり，他プロセスから監視対象プロセスのユーザ空間内の動作規則と呼び出し履歴の改竄が可能となる．

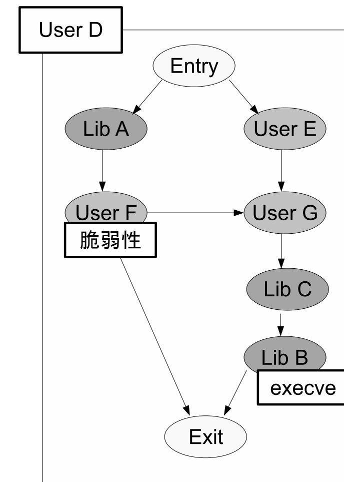

図 2 関数 D の動作規則

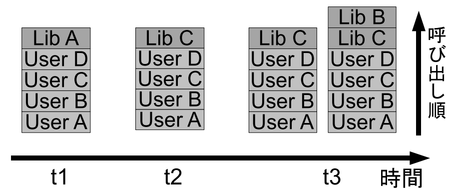

図 3 呼び出し履歴 (通常実行)

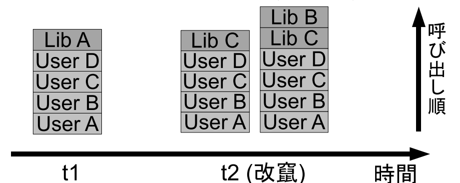

図 4 呼び出し履歴 (シェルコードにより改竄)

また，動作規則に関しての改竄はプログラム実行時以外でも発生し得る．監視対象プログラムが実行される前までは動作規則は監視対象プログラムに対応する libelem ファイルに格納されている．そのため，libelem ファイルをダウンロードして実行するまでの間に改竄があり得る．

4. 保護システム

動作規則及び，呼び出し履歴の保護手法について述べる．本章では侵入防止システムのための保護システムを提案する．保護システムは動作規則の保護と呼び出し履歴の保護の 2 機構からなる．以下にそれぞれの詳細を述べる．

4.1 動作規則の保護

3.2 で述べた動作規則の問題点に対する保護方式について述べる．

© 2009 Information Processing Society of Japan

4.1.1 動作規則の実行ファイルへの格納

動作規則は実行ファイルと一対一に対応するものであり，実行される上で必ず必要となる．つまり，何らかの関連付けを行う必要があるが，その方法として実行ファイルに格納することを提案する．これにより従来システムでは実行ファイルとこれに対応する libelem ファイルが独立していたのに対し，実行ファイルひとつで済む．また libelem を汎用的に使用可能となる．

4.1.2 署名及び検証

プログラムが作成されてから実行されるまでの間では様々な改竄が発生し得る．そこで，動作規則が格納された実行ファイルに対する改竄を検知するために実行ファイルにデジタル署名を付加する．そして，プログラム実行時に署名の検証を行う．これにより，プログラム実行時までに発生し得る改竄を検知する．

4.1.3 動作規則への書き込みの禁止

監視対象プログラム実行中のメモリに配置された動作規則に対して改竄が発生し得る．そこで，プログラム実行時に動作規則を書き込み禁止に設定することでシェルコードからの改竄攻撃を防ぐ．また，他プロセスからの ptrace システムコールによる動作規則の改竄に対しては ptrace システムコールの動作を制限することで動作規則への改竄攻撃を防ぐ．すべての ptrace システムコールを監視し，監視対象プロセスの動作規則が配置されたメモリへの書き込みを禁止する．これにより攻撃者は動作規則を改竄することはできない．

4.2 呼び出し履歴の保護

3.2 で述べた呼び出し履歴の問題点に対する保護方式について述べる．

4.2.1 libelem のランダム配置

2 章で述べたように呼び出し履歴は libelem の.bss セクションに存在する．.bss セクションの libelem 内オフセットはコンパイル時に静的に決定される．つまり，libelem の配置アドレスから呼び出し履歴の配置アドレスを知ることができる．よって，呼び出し履歴を保護するには libelem の配置アドレスを隠す必要がある．libelem は共有ライブラリとして提供され，監視対象プログラム実行時に dynamic linker によりユーザ空間にマッピングされる．そこで，このマッピングアドレスをランダマイズすることで，libelem をユーザ空間上にランダムに配置する．このため攻撃者はまず呼び出し履歴のアドレスを探す必要があり，このとき観測される挙動から攻撃を検知することができる．これには，`/proc/<PID>/maps` の参照のための open システムコール発行による異常検知や総当り攻撃によるセグメンテーションフォルトの検知などが当てはまる．また，動作規則同様，他プロセスからの ptrace システムコールによる改竄に対してはその利用範囲を限定することで libelem への改竄攻撃を防ぐことができる．以下，libelem の配置アドレスによる情報漏れの対策について述べる．

4.2.2 他プロセスによる監視対象プロセスの proc 情報参照の禁止

Linux では proc ファイルシステムを参照することで対象プロセスのメモリ空間レイアウトを知ることができる．このため，監視対象プロセスのユーザ空間に配置された libelem の配置アドレスを知ることができる．監視対象プロセスのユーザ空間に注入したシェルコードが proc を参照する場合，open システムコールの発行により Belem により検知される．しかし，他プロセスからは監視対象プロセスのメモリ空間レイアウトを知ることができる．そこで得た libelem の配置アドレスを元にシェルコードを用いて libelem を改竄することが可能である．これに対して，監視対象プロセスのメモリ空間レイアウトの参照は監視対象プロセス自身による参照のみに限定する．これにより，シェルコードと他プロセスのどちらも libelem の配置アドレスを知ることはできない．

4.2.3 ライブラリ関数アドレスの非再利用化

監視対象プログラム実行時，dynamic linker により実行プログラムが呼び出すライブラリ関数のアドレス解決が，libelem から順に行われる．libelem には実行プログラムが呼び出すライブラリ関数と同名のフック関数がそれぞれ定義されているため，実行プログラムが呼び出すライブラリ関数はすべて libelem を経由する．dynamic linker によりライブラリ関数のアドレスが解決されると，そのアドレスは実行プログラム内の GOT[7] の対応するエントリに保持される．実行プログラムはこのテーブルを参照することで目的のライブラリ関数を呼び出すことができる．よって，実行プログラム内の GOT を参照することで libelem の配置アドレスを知ることができる．そこで，ライブラリ関数のアドレスを GOT に保持しないようにすることで libelem のアドレスを隠す．アドレスを隠すことで実行プログラムからも参照できなくなるが，これに関しては GOT を用いないライブラリ関数呼び出し機構を導入することで解決する．

5. 実　　装

本章では保護システムの実装について述べる．まず，システムの概要を示し，次に各々について詳しく説明する．

5.1 保護システムの全体像

本保護システムは 2 つの保護機構から成る．ひとつは動作規則の保護機構であり，もう一方は呼び出し履歴の保護機構である．保護システムの全体像を図 5 に示す．

© 2009 Information Processing Society of Japan

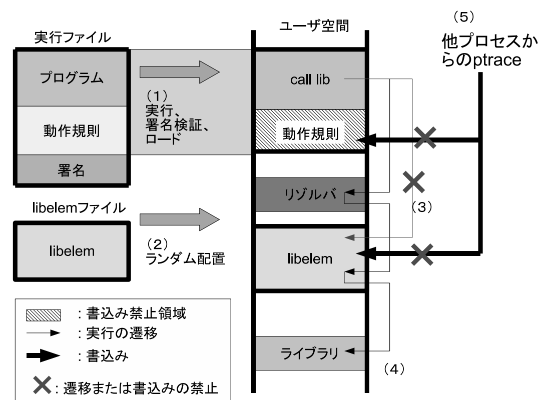

図 5 保護システムの全体像

まず，動作規則の保護システムの概要を述べる. 実行ファイルには予め動作規則と共にプログラム作成者が署名を付加する (図 5 左部分参照). これによりプログラム実行時までに発生し得る改竄に対応する. 次にユーザがプログラムを実行するとカーネルが署名を検証し，改竄されていなければプログラムと動作規則はユーザ空間にロードする (図 5(1) 参照). このとき，動作規則を書き込み禁止領域に配置する. これにより同プロセスからの改竄攻撃を防ぐ. また，他プロセスからの ptrace による攻撃も 4.1.3 で述べた方法により防ぐ (図 5(5) 参照).

次に，呼び出し履歴の保護システムの概要を述べる. ユーザによりプログラムが実行されると対応する libelem もユーザ空間にロードされる. このとき，libelem をランダムに配置 (図 5(2) 参照) することで，libelem に含まれる呼び出し履歴のアドレスを特定困難とする. また，実行プログラム内の GOT に libelem のアドレスが含まれる問題に対しては実行プログラムから GOT を利用した直接の libelem への遷移を禁止する (図 5(3) 参照) ことで解決する. 実行プログラムがライブラリ関数を呼び出す度にリゾルバによるアドレスの解決と遷移を行う (図 5(4) 参照). これにより実行プログラム内の GOT エントリには libelem の関数アドレスが含まれない. また，動作規則の保護と同様に他プロセスからの ptrace による攻撃も防ぐ (図 5(5) 参照). 以降，保護システムについて詳細に述べる.

5.2 動作規則の保護

動作規則保護の実装を，以下に述べる. まず，実行ファイルへの動作規則の格納について述べ，次に動作規則を格納した実行ファイルへの署名と検証について述べる. 最後に動作規則への書き込み禁止について述べる.

5.2.1 動作規則の実行ファイルへの格納

監視対象プログラムから得られた動作規則をその実行ファイルへ格納する方法を述べる. 動作規則を格納するにあたり，本研究で対象とした実行ファイルフォーマットは ELF[8)]である. ELF ファイルへの格納方法は次の通りである. まず，監視対象の ELF ファイルの末尾に動作規則を追加する. これは，ELF の各セクションはファイル内オフセットで指定されるため，オフセットがずれることを防ぐためである. さらに，追加された動作規則がプログラム実行時にユーザ空間へロードされるように ELF ファイルへプログラムヘッダを追加する. このとき，動作規則のロードアドレスは任意の設定で良いが，本実装では実行プログラムが配置されるセグメントの次ページアドレスを設定している. プログラムヘッダを単に追加しただけではプログラムは実行不能になる. プログラムコードは ELF ヘッダとプログラムヘッダに続いて同セグメント内にロードされるようになっており[⋆1]，プログラムヘッダサイズ分だけ以降のプログラムコードやデータのロードアドレスがずれるためである. そこで，コードやデータのロードアドレスを変えずにプログラムヘッダを追加する必要がある. ELF ヘッダとプログラムヘッダとプログラムコードを含むセグメントの内，ELF ヘッダとプログラムヘッダはロードアドレスに依存しないデータであり，そのロードアドレスに影響されない. そこで，このセグメントを 1 ページ分低位に配置し[⋆2]，新規プログラムヘッダを追加した後，プログラムコードに関してはロードアドレスが変わらないようにプログラムヘッダとプログラムコードとの間にパディングをした. これにより，実行プログラムの動作に支障をきたすことなく，動作規則ロード用のプログラムヘッダを新たに追加することができる. なお，この作業を行うプログラムを C 言語により作成した.

5.2.2 署名及び検証機構

まず，署名された実行ファイルの利用について述べる. 実行ファイルへの署名はプログラム作成後にそのプログラム作成者が行う. そのため，ユーザは信頼のできるプログラム作成者から署名検証のための公開鍵を事前に受け取る必要がある. 本システムでは受け取った公開鍵をカーネルに静的に組み込む方式である.

次に署名方法について述べる. まず，ELF ファイルの先頭から末尾までのすべてからハッ

⋆1 実行時に dynamic linker がヘッダを参照する.
⋆2 ページ境界に設定する必要がある.

© 2009 Information Processing Society of Japan

シュ値を計算する．得られたハッシュ値を暗号化し，ELF ファイルの末尾に追加する．署名が付加されたファイルオフセットは ELF ヘッダに格納した．これは ELF ヘッダは拡張性があるため，拡張しても実行ファイルローダの動作に支障をきたすことがないためである．次に，ELF ファイルに付加された署名の検証について述べる．execve システムコールが発行されると，カーネル内部でのフォーマットチェックなどを経て，`load_elf_binary` 関数が実行される．`load_elf_binary` 関数内に ELF ファイルに付加された署名の検証機構を追加した．ここで実行ファイルが改竄されていないと判断されればプログラムを実行する．逆に，改竄されていると判断されれば，execve システムコールはエラーを返す．

5.2.3 動作規則への書き込みの禁止

監視対象プロセスのユーザ空間に注入されたシェルコードによる改竄に対しては動作規則の配置されるページを書き込み禁止にすることで防ぐ．これには動作規則の実行ファイルへの格納時に動作規則に対応するプログラムヘッダのアクセス権設定フラグを書き込み禁止に設定する．これによりプログラム実行時に動作規則を書き込み禁止領域としてユーザ空間に配置される．

他プロセスからの動作規則の改竄に対しては ptrace システムコールの利用範囲を限定することで改竄を防ぐ．これには Linux カーネル内の ptrace システムコールの変更を行う．ptrace システムコールに渡された引数を調べ，その内容が監視対象プロセスの動作規則に対する書き込みであれば，システムコールをエラーで返す．

5.3 呼び出し履歴の保護

呼び出し履歴保護に関して述べる．まず，libelem のランダム配置について述べ，次に他プロセスによる監視対象プロセスの proc 情報参照の禁止について述べる．最後に監視対象プロセスが呼び出すライブラリ関数アドレスの非再利用について述べる．

5.3.1 libelem のランダム配置

libelem は共有ライブラリとして作成されるため，監視対象プログラム実行時に dynamic linker によりユーザ空間にロードされる．そこで，dynamic linker を変更し，共有ライブラリのアドレスをランダマイズした．共有ライブラリすべてについてランダマイズを行うため，libelem 以外の他の共有ライブラリもランダムにロードされる．これは libelem の位置を隠すことの他に，他のライブラリ関数への return into libc 攻撃を防ぐ効果もある．また，動作規則と同様に ptrace システムコールの改変も行うことで他プロセスからの改竄に対応する．

5.3.2 他プロセスによる監視対象プロセスの proc 情報参照の禁止

他プロセスが proc ファイルシステムを参照することで監視対象プロセス内の呼び出し履歴のアドレスを知ることができることに対応するために，Linux カーネル内の open システムコールを変更を行う．open システムコールに渡された引数を調べ，proc ディレクトリの監視対象プロセスのディレクトリ以下に対するものであればシステムコールをエラーで返す．

5.3.3 ライブラリ関数アドレスの非再利用化

まず GOT を用いたライブラリ関数呼び出しについて述べる．GOT とは実行プログラムや共有ライブラリから他の共有ライブラリへアクセスするためのテーブルである．このテーブルを書き込み可能領域に配置させることで，dynamic linker はプログラム実行時に各 GOT を初期化する．共有ライブラリ関数呼び出しにおいて，GOT は他の共有ライブラリ内の関数アドレスを保持するために用いられる．これにより，ライブラリ関数を呼び出すプログラムは GOT の当該エントリを参照することで意図するライブラリ関数を呼び出すことができる．

また，dynamic linker による GOT の初期化タイミングには 2 種類ある．まずはプログラム実行時にすべての GOT エントリを初期化する方法である．しかし，一般的にプログラムはすべてのライブラリ関数を呼び出さず，エラー処理関数などは呼ばれないことが多い．そこで，無駄なアドレス解決を省くために最初のライブラリ関数呼び出し時に初めてアドレス解決を行うしくみが標準となっており，これが 2 つ目の方法で lazy binding と呼ばれる．lazy binding を行う場合，GOT の他に PLT と呼ばれるテーブルが用いられる．PLT の各エントリは GOT の各エントリに対応しており，小さなプログラムコードが格納されている．実行プログラムのライブラリ関数の呼び出し先はこの PLT の各コードになっている．実行プログラムがライブラリ関数を呼ぼうとすると各関数に対応する PLT エントリに遷移することになる．もし対応する GOT エントリが初期化されていなければライブラリ関数のアドレスを解決し，GOT エントリを初期化する．一度初期化されると，そのライブラリ関数は次回からは直接呼ばれることになる．

次に，前述した lazy binding に着目したライブラリ関数アドレスの非再利用化について述べる．GOT の初期化を lazy binding にした場合，各ライブラリ関数の最初の呼び出しでアドレスが解決され，対応する GOT エントリが初期化されることになる．このとき，GOT エントリを初期化しないようにすることはライブラリ関数呼び出しごとに毎回アドレス解決を行うことを意味する．このように，リゾルバにより解決されたライブラリ関数のアドレスを GOT に初期化せず再利用しないようにすることで GOT 内の libelem で定義されて

© 2009 Information Processing Society of Japan

表 1 実験環境

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

いる関数のアドレスを知られることを防ぐことができる．次に，攻撃者がアドレス解決を行うリゾルバプログラム (`_dl_runtime_resolve`) を実行する可能性がある．しかし，この `_dl_runtime_resolve` は trampoline code[9] であり，対応するライブラリ関数に遷移して呼び出し元に戻ることはない．もしそのライブラリ関数がシステムコールを発行すれば侵入防止システムにより異常を検知することができる．しかし，システムコールを発行しない関数であった場合には，攻撃者は libelem への戻りアドレスや退避ベースポインタをコールスタック上から探し出すことで libelem の位置を特定することができる．これに対しては，libelem から実行プログラムに戻る直前に使用していたコールスタック領域をゼロクリアすることで解決することができる．

6. 評　価

署名検証に要する時間とライブラリ関数配置アドレスの非再利用によるオーバヘッドの測定実験と評価を行った．実験に用いた環境を表 1 に示す．

6.1 署名検証に要する時間

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を測定した．実験には約 10K バイトと約 100K バイトのサイズのプログラムを使用した．署名検証システムの有無での測定結果を表 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．署名システムによる処理時間増加を検証するために，署名検証システムの主な処理ごとの測定結果を表 3 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である．

表 2 署名検証システム有無での測定結果

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 3 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)

表 4 再利用の有無での実行に要する時間

プログラム名	再利用 (通常)	非再利用 (通常)	再利用 (Belem 上)	非再利用 (Belem 上)
wc	1.459(sec)	19.808(sec)	3.131(sec)	22.272(sec)
inetd	0.030(sec)	0.036(sec)	0.036(sec)	0.041(sec)

6.2 ライブラリ関数アドレスの非再利用によるプログラム実行オーバヘッド

wc と inetd についてライブラリ関数アドレスを再利用する場合と再利用しない場合で実行に要する時間を評価した．wc は，1MB のテキストファイルを与えてから結果を出力するまでの時間を計測した．inetd は，inetd が稼働するマシンに対して同じ LAN 内の別マシンから 10 回接続を行った時間を計測した．結果を表 4 に示す． 表 4 の各列は左からそれぞれ，実験に使用したプログラム名，通常実行でアドレスを再利用した場合の計測時間，通常実行でアドレスを再利用しなかった場合の計測時間，Belem 上でアドレスを再利用した場合の計測時間，Belem 上でアドレスを再利用しなかった場合の計測時間を表す．wc では通常実行における再利用の有無での時間の差は 18.3sec，Belem 上実行における再利用の有無での時間の差は 19.1sec となり，ほぼ等しい結果となった．また，inetd の通常実行における再利用の有無での時間の差は 0.006sec，Belem 上実行における再利用の有無での時間の差は 0.006sec となり，wc 同様にほぼ等しい結果となった．これは通常実行と Belem 上実行においてアドレス解決時間が影響されないことを示している．また，wc ではプログラム実行中に約 4500 万回のライブラリ関数を呼び出し，各関数呼び出しにおける平均アドレス解決時間は約 0.42μsec となった．inetd では約 1 万回のライブラリ関数を呼び出し，各関数呼び出しにおける平均アドレス解決時間は約 0.60μsec となった．プログラムごとに若干の差はあるが，平均アドレス解決時間はだいたい同じ結果になることが確かめられた．つまり，アドレス非再利用によるプログラム実行オーバヘッドはプログラムが呼び出すライブラリ関

© 2009 Information Processing Society of Japan

数の回数に依存する．wc は文字列処理プログラムであるため，ライブラリ関数呼び出しあたりの平均システムコール発行数が少なく，且つプログラム実行の間，ライブラリ関数が断続的に呼び出される．このようなプログラムの場合，アドレス非再利用による実行時オーバヘッドは高くなる．一方，inetd はサーバプログラムであり，要求があった場合のみ処理を行う．また，呼び出すライブラリ関数も入出力を伴うシステムコールを発行し，またライブラリ関数呼び出しあたりの平均システムコール発行数が大きい．このようなプログラムの場合，アドレス非再利用の影響が少なく，オーバヘッドは許容できる範囲内と考えられる．

7. 関連研究

動作規則及と呼び出し履歴の保護に関連する研究を以下に述べる．

7.1 実行ファイルへの署名

実行ファイルへの署名機構として bsing[10] と DigSig[11] がある．bsign は実行ファイル形式として ELF を対象とした署名ユーティリティである．また，署名検証機能も有するが，検証がユーザランドで行われるため，検証プロセスが乗っ取られていた場合や，検証後から実行時までに改竄された場合は改竄を検知することができない．一方，本システムではプログラム実行時に毎回署名検証が行われるため ELF ファイルへの改竄を確実に検知することができる．

一方，DigSig は bsign で署名された ELF ファイルを実行時に検証する機構を提供する．実装には LSM を利用し，LKM により作成されているため，セキュリティ上の問題があり．Linux バージョン 2.6.24 からは LKM による実装は不可となっている．一方，本システムは Linux カーネルに組み込まれる形で実装されているため LKM で提供される DigSig に比べ安全性が高いと言える．

7.2 データ配置のランダマイズ機構

データ配置のランダマイズ機構としてと Pax[12] がある．Pax は mmap が行うマッピングアドレスをランダマイズすることでデータへの不正なアクセスを防ぐ．しかし，ランダマイズは mmap ベースに対してであり，プログラム実行時に一度だけ行われる．そのため複数の共有ライブラリをマッピングさせるとき，それらの配置アドレスは変わるが，配置順序は変わらず，ひとつの共有ライブラリの配置アドレスから他の共有ライブラリの配置アドレスも分かるという問題点がある．一方，本システムでは実行プログラムが必要とする各共有ライブラリは dynamic linker によりそれぞれがランダムに配置される．

8. おわりに

本論文では異常検知に基づく侵入防止システムである Belem を例に挙げ，Belem が深く依存する動作規則と呼び出し履歴が改竄され得ることを示し，これに対する保護システムを提案した．また，実際に保護システムを開発し，評価を行った．本システムを導入することで動作規則と呼び出し履歴は改竄攻撃から保護され，Belem が監視対象プロセスの実行状態と動作規則を正しく把握できる環境が得られた．今後の課題としては公開鍵基盤に対応した，より柔軟な署名検証システムを導入することが挙げられる．現状では署名されたプログラムを実行するために事前にその作者が配布する公開鍵をカーネルに組み込んでおく必要がある．これに対して公開鍵基盤によってプログラム作者の公開鍵の一元管理を行うことで，ユーザは第三者機関の公開鍵のみを組み込んでおくだけで良い．

参考文献

1) IPA: 情報処理推進機構，http://www.ipa.go.jp/.
2) 阿部洋丈，大山恵弘，岡　端起，加藤和彦：静的解析に基づく侵入検知システムの最適化，情報処理学会論文誌， Vol.45, No.SIG 3(ACS 5), pp.11–20 (2004).
3) Wagner, D. and Dean, D.: Interusion Detection via Static Analysis, *In Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp.156–168 (2001).
4) 大山恵弘，王　維，加藤和彦：異常検知システムにおける正常動作データのモジュール化，コンピュータシステム・シンポジウム論文集，pp.45–52 (2003).
5) 島本大輔，大山恵弘，米澤明憲：System Service 監視による Windows 向け異常検知システム機構，情報処理学会論文誌，Vol.47, No.SIG 12(ACS 15), pp.420–429 (2006).
6) 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-OS-110, No.9, pp.3–10 (2009).
7) Dreper, U.: How to Write Shared Libraries, http://people.redhat.com/.
8) Nishida, W.: SkyFree.org, http://www.skyfree.org/linux/references/.
9) the GCCTeam: the GCC Compiler Collection, http://gcc.gnu.org/ml/java/2004-07/msg00187.html.
10) debian: BSign Debian package, http://packages.devian.org/stable/admin/bsign.
11) Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M. and Roy, V.: Digsig: Run-time authentication of binaries at kernel level, *Proceedings of LISA Eighteenth Systems Administration Conference*, pp.59–66 (2004).
12) Team, T.P.: Homepage of The Pax Team, http://pax.grsecurity.net/.

© 2009 Information Processing Society of Japan