コンパイラとOSの連携による強制アクセス制御向けプロセス監視手法

表 雄仁 †　森山 壱貴 †　桑原 寛明 ‡　毛利 公一 ‡　齋藤 彰一 ††
上原 哲太郎 ‡‡　國枝 義敏 ‡

† 立命館大学大学院理工学研究科　‡ 立命館大学情報理工学部情報システム学科
†† 名古屋工業大学大学院工学研究科　‡‡ 京都大学学術情報メディアセンター

あらまし 本論文では，コンパイラとの連携によるOSのプロセス監視システムを提案する．本システムでは，コンパイラがシステムコールやアクセスするリソースを解析することでプログラムの正常な動作を記述したポリシーを生成する．そして，OSはコンパイラが生成したポリシーに基づき実行時に動的解析を行い，プロセスの不正な動作もしくは不正なアクセスを禁止する．本稿では，ポリシーに基づいた実行プロセスの監視やアクセス制御を行うための手法を述べる．

A Process Monitoring for Mandatory Access Control by Collaboration of Compiler and Operating System

Yuji Omote †　Ikki Moriyama †　Hiroaki Kuwabara ‡　Koichi Mouri ‡
Shoichi Saito ††　Tetsutaro Uehara ‡‡　Yoshitoshi Kunieda ‡

†Graduate School of Science and Enginering Ritsumeikan University
‡Department of Computer Science, College of Information Science and Engineering, Ritsumeikan University
††Graduate School of Engineering, Nagoya Institute of Technology
‡‡Academic Center for Computing and Media Studies, Kyoto University

Abstract In this paper, we propose a Secure system as follow. Compiler generates the policy data by analyzing system calls and resource accesses of a given source program. By using the policy data, OS watches and analyzes the user's running process. OS can detect anomaly behaviors and unjust accesses infringing the policy data. In this paper, both idea and a concrete technique for access control based on policy data are described.

1 はじめに

近年，ネットワーク技術の発展から起きた弊害として不正アクセスが急増してきている．特に，バッファーオーバフロー攻撃などのソフトウェアの脆弱性をついた攻撃は年々増加しており，大きな問題になっている．Linuxではそうした攻撃で管理者権限を奪われてしまうとシステム全体を乗っ取られたことになる．この問題を解決する手段として，強制アクセス制御機能を付加したSecure OSが提供されている [1][2]．強制アクセス制御とは，あらかじめ定義されたルール (ポリシー) の許容範囲内でのみプログラム実行を可能とする機構である．従って，管理者プロセスに対してもユーザプロセス同様のアクセス制限を付加することができる．この手法を組み込むことにより，管理者権限を奪われてしまったとしても被害を最小限に抑えることができる．

Linuxにおいて，ファイルやディレクトリ，ソケットなどリソースへアクセスするには全てシステムコールが使用される．従って，プロセスが

使用するシステムコールをOS側で管理，制御することで，攻撃されてもシステムへの被害を最小限に軽減できる．この考えに基づき，Secure OS の多くはシステムコール発行時に，アクセス制御を行うことで実行プロセスの監視を行っている．このアクセス制御のために，プロセスやリソースに対するアクセス権限を細かく限定させ，それをポリシーとしてまとめる．例えば，Secure OS の 1 つである SELinux[1] ではプロセスやリソースに対する制御をセキュリティ管理者がポリシーとして予め記述しておくことで，そのポリシーに記述されていないプロセスの動作やリソースへのアクセスを停止させる．しかし，すべてのプロセスが発行するシステムコールやリソースへのアクセスを予め正確に把握することは通常は不可能であり，適切なポリシーの記述を行うことが困難であることが指摘されている [3]．そのため，TOMOYO Linux[2] のように，許可したい操作を一通り実行することで，許可したいファイルやプロセスへのアクセスを自動学習させる方式がある．[2] によれば，自動学習により必要なアクセスの 90%を定義することできるとしているが，必ずしも学習時に正しい入力やプロセスの遷移が行われるとは限らない．結果的に，人間が学習時の実行アクセスパターンや学習の正確性を保証しなければならない．すなわち，どのようなアクセスパターンを学習させるのかを判断する必要がある．

こうした現状を踏まえ，我々は，コンパイラの解析からポリシーを生成し，生成されたポリシーを基に実行時に動的解析を行うセキュアシステム提案する．コンパイラとOSが連携することでユーザに負担のない高信頼性のシステムを構築することを目標とする．ここで，高信頼性のシステムとは，プロセスの安全性を保証したシステムとする．プロセスの安全性は，プログラムや実行中のプロセスに対する悪意を持ったコードの書き換えなどにより，ユーザの意図に反した方法でリソースにアクセスできないこととする．本システムでは，コンパイラがソースコードを解析することでポリシーを生成するので，ユーザに負担がなく，さらに細かな制御情報をポリシーとして生成できる利点がある．OSがポリシーを基に動的解析を行うことで，コード書き換えやプロセスの乗っ取りによる不正なプログラムの実行，不正なファイルアクセスも防止することができる．OS によるアクセス制御を行うことで，システム管理者にもユーザと同様のアクセス制御を施すことができる．その意味で一般の Secure OS と同等の強制アクセス制御が実現できると考える．

この目標を具体化し，本稿では，上記の考えに基づく，コンパイラの解析情報を基にした強制アクセス制御向けプロセス監視手法を述べる．第 2 章で本システムの概要を述べ，第 3 章において提案するプロセス監視手法を述べる．第 4 章で関連研究について述べ，第 5 章でまとめる．

2　提案システムの概要

提案システムは，プロセスがコンパイル時のプログラム記述以外の動作を起こそうとする不正実行を検出し実行を停止させるシステム (図 1) である．具体的には，コンパイラはソースプ

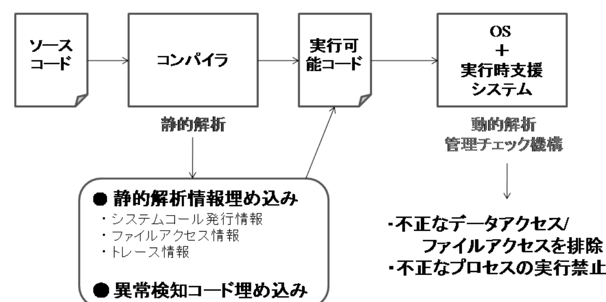

図 1: システム構成

ログラムを静的解析し，ポリシーとして利用する各種情報を抽出する．抽出する情報は，システムコールの種類および各引数情報を記述した許可リストや関数のコールグラフ，システムコールの各引数に使われる変数の追跡情報で構成されている．自動生成されたポリシーは通常の目的コード中に目的コードとは別に独立して埋め込まれて，OS に渡される．

プロセスがファイルなどのリソースへアクセスする場合には，システムコールを発行しなければならない．従って，プロセスが発行できる

システムコールやシステムコールがファイルにアクセスするアクセスパスを制限することができれば，プロセスがファイルへ不正にアクセスすることが難しくなる．つまり，プロセスが発行するシステムコールを制限することで，被害を最小限に抑えることができると考えられる．そこで，OS では，コンパイラにより生成されたポリシーを基にシステムコールの発行を契機として実行時の動的解析を行い，不正なシステムコールの実行を検出し停止させる．脆弱性をつかれてプロセスが乗っ取られた場合でも，ポリシーに記述されていないシステムコールの実行を阻止することができるので，ファイルへの不正なアクセスや不正なプログラムの実行を防ぐことができる．つまり，システムへの被害を最小限に抑えることができる高信頼システムが実現する．

ただし，本システムを実現するためにはコンパイラによって解析されるソースプログラムが悪意を持って書かれていないことを前提とする．

次に本手法で使用するポリシー内の許可リストについて述べる．許可リストとは，静的解析によって生成されたシステムコールに関する情報のことである．生成される許可リストの例を図 2 に示す．

```
syslist   {
    main        {
      execve      "/bin/ls"      argv    NULL
    }
    file_copy   {
      open        from_name      0
      open        to_name        577     0666
      read        from_fd        buff    1024
      write       to_fd          buff    rcount
      close       from_fd
      close       to_fd
    }
}
```

図 2: 許可リストの例

同図中 syslist で囲まれている部分にシステムコールに関連する情報を抽出しまとめる．まとめ方としては，ユーザ関数ごとに発行される全てのシステムコールとその引数情報をリスト形式で与える．ユーザ関数ごとにシステムコールを記述することで，実行時に発行できるシステムコールを限定することができる．また，システムコールごとに実引数の値を記述しておくことで，発行されるシステムコールごとに実行時のファイルパスなどを細かに解析を行うことができる．コンパイラは，コンパイル時に関数ブロック内にシステムコールを検出すると，第 1 列に当該システムコール名，第 2 列以降には当該システムコールの実引数リストを記述する．ここで与えられる実引数は定数，値域 (引数の取り得る値の範囲，またはその集合)，未定義変数 (式) いずれかである．コンパイラの最適化機能により，ソースコード上では実引数が変数 (式) であっても，定数伝幡，定数の畳み込みなどの手法により，値を静的に確定できる場合がある [4]．こうした静的解析機能を用い，可能な限り値域を限定し許可リストに登録する．確定できない場合には，許可リストに未定義変数として記述し，実行時にいかなる値も取り得ることを示す．ただし，その場合も今度は動的解析により，実行時に値を絞り込める場合がある．可能な引数の値を限定することで，許可されない挙動を厳格にチェックすることが可能となる．

3 プロセス監視手法

提案手法は，プロセスがシステムコールの発行を監視し，そのシステムコールやシステムコールの実引数を動的解析し，ポリシーを基に検査する．ユーザプロセスがシステムコールを発行すると，そのシステムコールが許可リストに登録されているシステムコールかの判定を行い，システムコールの引数の検査を行う．

3.1 システムコールの許可判定

システムコールの許可判定では，ユーザプロセスのシステムコールの発行時，発行されたシステムコールが許可リストに記述されているかどうかの判定を行う．ユーザ関数内で複数のシステムコールの発行順序が一意であることが静的解析で保証されている場合，その順序を考慮し判定を行う．現在，本手法ではシステムコール

の発行順序を考慮しておらず，どのユーザ関数から発行されたシステムコールかを判断することができない．システムコールの発行された関数の特定の確認を行うために，スタックフレーム中の実行時の呼び出し関係の解析を行い，そのシステムコールがどのユーザ関数から発行されたシステムコールかを解析する．これにより，呼び出し関係からも不正実行を検出する．

3.2　システムコールの引数検査

システムコールの引数検査には，基本的に定数，値域の場合は許可リストとのパターンマッチングを行う．コンパイラによって解析されたポリシー内の許可リストでは，システムコールの各引数は定数，値域，未定義の 3 種類の場合が存在する．値域もしくは未定義だった場合，値が確定されないため，プロセスが乗っ取られた場合に，リソースへ不正にアクセスされる危険がある．従って，実行時に値が確定できるのであれば，ポリシーを動的に書き換え，システムコールの実引数の値を確定もしくは絞り込みを行う必要がある．そこで，実行時に実引数の値を確定もしくは絞り込みを行う手法を以下に述べる．

- 許可されていないパスへの侵入の禁止
  “../” を使用したディレクトリ階層を遡る場合に，許可されていないパスを通ることを禁止する．例えば，システムコールの発行時の引数の動的解析時に “/home/user/../../bin/ls” が実引数に入っていたと仮定する．“/home/user/../../” までのパスを評価を行うと，“/” となる．このとき，“/” が許可されているパスでなければ，不正なアクセスであると判断し停止さする．

- 複数のシステムコールの同一変数使用時の値の固定化
  read システムコールや write システムコールで使用されるファイルディスクリプタ変数は，コンパイラの解析結果では値を確定できない場合がある．これらのシステムコールで使用されるファイルディスクリプタは open システムコール発行時に値が決まり，close システムコール発行時に値が破棄される．従って，コンパイラの解析時に値が確定できなかったとしても，この間の変数値が不変であるとコンパイラにより保証されていれば，実行時に値を確定することができる．このことから，open システムコール発行時に許可リストの未定義になっている戻り値であるファイルディスクリプタ変数を動的解析により定数に置き換える．さらに，open 後の read/write システムコールに関する許可リスト中のファイルディスクリプタに対応する実引数を調べ，同一変数のものは，open と同様に定数に置き換える．そして，close システムコール発行時にこれらの置換した定数を元の未定義変数に戻す．

3.3　プロトタイプシステムの実装

今回は，有効性を検証するために提案システムのプロトタイプを作成した．このプロトタイプシステムは，ユーザプログラムを監視する実行時システムである．プロトタイプシステムでは，発行されたシステムコールの順序は考慮せず，システムコールがポリシーの許可リストに登録されいるかどうかの検査と定数もしくは値域におけるシステムコールの引数の検査を行う．プロトタイプシステムは，ユーザプログラムを子プロセスとして動作させ，自身は監視プロセスとしてユーザプロセスの監視を行う．以下に処理手順を述べる．

1. ユーザプロセスがシステムコールの発行を行うと，監視プロセスはユーザプロセスを一旦停止させる．
2. 発行されたシステムコールが登録されているシステムコールかどうか検査する．
3. システムコールの実引数の動的解析を行い，ポリシーの許可リストと照合しシステムコールの引数を検査する．
4. 監視プロセスはユーザプロセスが正常動作していると判断すると，制御をユーザ

プロセスに戻し，システムコールを実行させる．ポリシーに記述されていない動作を検知すると，監視プロセスはユーザプロセスが不正実行すると判断し，ユーザプロセスにシグナルを送りその実行を強制的に停止させる．

4　関連研究

システムコールを契機としたプロセス監視手法として，Wagner[5] らのシステムがある．彼らは専ら静的解析に基づく侵入検知システムを提案している．その方式は，コンパイラによりシステムコールが発行される可能性のある関数呼び出しに関する順序関係のパターン可能なを抽出し，実行時に発行されたシステムコールが元のソースコードに記述された順序通り呼ばれているか検査する．このとき，システムコールが発行される順序関係のあらゆる可能性をすべて検査するので，実行時のオーバーヘッドが大きい．さらに，攻撃の見逃しも多いことが指摘されている．加藤ら [6] はこうした問題点についてこのシステムを改良している．まず，スタックフレームを実行時にトレースし，システムコール発行時の関数の呼び出しの遷移を検査することで攻撃の見逃しを削減しつつ，実行時の解析結果を学習することで，静的解析で得られた順序関係で，実行時にあり得ないパターンを削減し，除去することで検査を高速化している．しかし，これらのシステムはシステムコールの引数を全く検査していないので，もし乗っ取られてしまった場合，いかなるリソースに対してもアクセスすることが可能である．また，システムコールの発行順序を考慮に入れているのでマルチスレッドモデルのプログラムや setjump/longjump 関数を使用したプログラムは解析できない．一方，一方，今回の提案手法は，システムコールの順序を考慮に入れていないので，これらのモデルのプログラムにも対応できる．また，引数の取り得る範囲を解析し，リソースへのアクセスを制限しているので，攻撃の見逃し押さえることができると考える．

リソースへのアクセスを制限するシステムとして Software Pot[7] のようなサンドボックスシステムがある．ユーザが設定したポリシーにしたがって，システムコールの発行やリソースへのアクセスを制限することができる．しかし，プログラムの全体の設定しか行うことができないので，バッファオーバーフロー攻撃で乗っ取られた場合，データを盗まれたり改ざんされてしまう可能性がある．また，ポリシーの記述が適切でない場合はプログラムを動かすことができなくなってしまう可能性がある．一方，我々はコンパイラの解析結果を利用することにより，システムコールごとにリソースへのアクセスを制限することが可能である．したがって，攻撃者は許可リストに書かれている情報を正確に知っておく必要があり，正確な攻撃をすることが困難である．プログラムが動かなくなることも無い．また，スタック領域に配置された実行コードの実行を禁止する Non-executable stack という手法 [8] もある．実行時のオーバヘッドが少なくスタックオーバフローを完璧に防ぐことが可能となっている．しかし，実行コードがヒープ領域に配置される場合や return to lib と呼ばれるリターンアドレスに悪意を持ったコードではなく，標準ライブラリのように予め判っているアドレスへ改ざんするような攻撃には対応できない．また，再帰処理を行うプログラムが実行不可能になってしまう [9]．提案手法では，発行するシステムコールが予め列挙されており，また，システムコールの各引数の取り得る値が限定されているので，例え，リターンアドレスがライブラリへと書き換えられたとしても検出可能である．

5　おわりに

本稿では，コンパイラの解析情報を基にした強制アクセス制御向けプロセス監視手法を提案した．本システムでは，コンパイラが生成したポリシーに基づき，実行時にプロセスが発行するシステムコールやシステムコールがアクセスするファイルパスを検査を行う．システムコールごとに制御を行うことで，既存の Secure OS よりもユーザプロセスに関して，さらに細かな

監視，異常抑制を可能にする．また，コンパイラにより自動生成されたポリシーを使用するこで，ユーザに負担の少ない高信頼性のシステムなっている．

今後の課題として，本手法の有効性を示すための False Positive や False Negative の誤検知精度評価を行う必要がある．そのために，まず，提案手法の未実装部分を実装を行う必要がある．また，Linux Kernel への実装を行う．現在は，ファイルへのアクセスパスの検査は文字列比較のみしか行っていない．従って，シンボリックリンクなどでのファイルパスが変更された場合には対応できない．これに対応することも今後の課題である．

参考文献

[1] Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[2] 原田季栄，保理江高志，田中一男：“TOMOYO Linux - タスク構造体の拡張によるセキュリティ強化 Linux”，Linux Conference 2004.

[3] 原田 季栄，保理江高志，田中一男：“プロセス実行履歴に基づくアクセスポリシー自動生成システム” Network Security Forum 2003.

[4] 森山壱貴，表雄仁，桑原寛明，毛利公一，齋藤彰一，上原哲太郎，國枝義敏：“コンパイラと OS の連携による強制アクセス制御向け静的解析”．Computer Security Symposium 2007(投稿中)，2007

[5] Wagner，D. and Dean，D.：“Intrusion Detection via Static Analysis”，Proc 2001 IEEE Symposium on Security and Privacy，Oakland，pages.156-168，May 2001.

[6] 阿部洋丈，大山恵弘，岡瑞起，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.SIG3(ACS5)，pages. 11-20，2004年3月．

[7] 大山恵弘，神田勝規，加藤和彦：“安全なソフトウェア実行システム SoftwarePot の設計と実装”，コンピュータソフトウェア，Vol.19，No.6，pages.2-12，2002.

[8] OpenWall Project. http://www.openwall.com/.

[9] 情報処理進行事業協会セキュリティセンター．オープンソースソフトウェアのセキュリティ確保に関する調査. http://www.ipa.go.jp/security/fy14/reports/oss_security/.