平成16年度 修士論文

カーネル感染型有害プログラム検出システムの設計と実装

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0350036 花田 智洋

指導教員 多田 好克 助教授
Vytautas Reklaitis 教授
古賀 久志 講師

提出日 平成17年1月31日

目 次

第 1 章　はじめに **6**

第 2 章　背景と目的 **9**

2.1 背景 . . . 9

2.2 目的 . . . 12

第 3 章　有害プログラム分析 **14**

3.1 ユーザレベル感染型有害プログラム . . . 15

3.1.1 攻撃手法 . . . 15

3.1.2 対策方法 . . . 16

3.2 カーネル感染型有害プログラム . . . 16

3.2.1 攻撃手法 . . . 17

3.2.2 対策方法 . . . 24

3.2.3 カーネル感染型有害プログラムのサンプル . . . 24

3.2.4 カーネル感染型有害プログラムによる特徴的な動作 . . . 26

第 4 章　関連研究 **27**

4.1 侵入検知システム . . . 27

4.1.1 検出方法による分類 . . . 27

4.1.2 取得する情報源による分類 . . . 28

4.2 システムの整合性検査ツール . . . 30

4.2.1 Tripwire . . . 31

4.2.2 Tripwire の限界 . . . 31

4.3 有害プログラム検出ツール . . . 33

4.4 システムコールを監視する侵入検知システム . . . 34

4.4.1 Execution Path Analysis 34
4.4.2 BlueBox 35
4.4.3 Independent Auditor 35

第 5 章 設計 36

5.1 設計方針 36
5.2 性能モニタリング 37
5.2.1 P6 ファミリ・プロセッサにおける性能モニタリングの限界 . 37
5.2.2 Pentium 4 プロセッサにおける性能モニタリングの拡張 . . . 39
5.2.3 Pentium 4 プロセッサにおける性能モニタリングの利用 . . . 42
5.3 監視対象とするシステムコール 43
5.4 検出アルゴリズム 44

第 6 章 実装 47

6.1 実装概要 47
6.2 性能モニタリング・カウンタの設定 48
6.2.1 システムコール実行時の総実行命令数カウント 48
6.2.2 システムコール実行時間のカウント 52
6.3 検出システムのアーキテクチャ 52
6.3.1 システムコールリダイレクトによるカウント 53
6.3.2 カウンタの開始 53
6.3.3 カウンタの停止と読み取り 55
6.3.4 カウンタ読み取り後の処理 56
6.3.5 有害プログラム検出に使用するしきい値 56
6.4 カーネル感染型有害プログラム検出実験 59
6.4.1 knark 59
6.4.2 Adore-ng 60

6.4.3 SucKIT 61
6.5 考察 63
6.5.1 本システムを利用してできること 63
6.5.2 本システムでは対処できない問題 64
6.5.3 オーバヘッド 65
6.5.4 今後の課題 66

第 7 章 おわりに **67**

図 目 次

2.1 ユーザレベルで攻撃を行う有害プログラム 10
2.2 カーネルレベルで攻撃を行う有害プログラム 10
2.3 カーネル感染型有害プログラムの概念図 11

3.1 ユーザプログラムのバイナリを改ざんする有害プログラム 16
3.2 システムコールの改ざん 20
3.3 システムコールテーブルの改ざん 21
3.4 システムコールテーブル呼び出し先アドレスの改ざん 22
3.5 idtr レジスタの改ざん 23

5.1 イベント選択制御レジスタ ([26] より転載) 40
5.2 カウンタ設定制御レジスタ ([26] より転載) 41
5.3 性能モニタリング・カウンタ ([26] より転載) 41
5.4 検出アルゴリズム 45

6.1 カウントするイベント選択の設定 50
6.2 カウント開始の設定 50
6.3 カウント読み取りの設定 51
6.4 カウント停止の設定 51
6.5 カウント停止の設定 52
6.6 システムコールリダイレクトのルーチン 54

表 目 次

6.1 実験環境 47
6.2 性能モニタリング・カウンタの対応表 48
6.3 リタイヤメント時カウント用性能モニタリング・イベント 49
6.4 総実行命令数のしきい値 57
6.5 実行時間のしきい値 58
6.6 knark が感染したシステム上での実験結果 59
6.7 Adore-ng が感染したシステム上での実験結果 61
6.8 SucKIT が感染したシステム上での実験結果 62
6.9 オーバヘッドの測定 65

第 1 章

はじめに

計算機システムに対する不正アクセスが，大きな社会問題となっている [1, 2, 3]．計算機システムへの攻撃者は，システムへの侵入後に今後のアクセス手段の確立，システムへの攻撃，証拠の隠ぺいなどを行う．これらの活動を簡単かつ円滑に行うために，攻撃者は有害プログラムを使用する．有害プログラムとは，計算機利用者に対して有害な機能を持ったプログラム [4, 5] のことである．

近年では，オペレーティング・システムのカーネル内で動作する有害プログラムが出現し，攻撃者が不正アクセスを行う際に使用することが問題になってきている [6, 7]．

攻撃者がこの有害プログラムをカーネルに組み込むと，攻撃者の活動や有害プログラムの存在を隠ぺいするために，カーネルの処理結果を改ざんする．カーネルの処理結果を改ざんすることによって，攻撃者にとって都合のよい情報のみを計算機利用者へ返すことが可能となる．

システム管理者が有害プログラム検出を試みる場合，ユーザプログラムを利用してカーネルの情報を取得する．このような有害プログラムに汚染されたシステムでは，有害プログラムによって改ざんされた情報を得ることとなる．

このように，ユーザプログラムを用いた検出ではカーネルに感染する有害プログラムの検出に限界がある．それゆえ，カーネルに感染する有害プログラムを検出するシステムが必要とされている．

そこで本研究では，オペレーティング・システムのカーネルに感染する有害プロ

グラム (以下，カーネル感染型有害プログラム) を検出するシステムを設計し実装する．

システム管理者の目から逃れた隠密活動をするために，攻撃者はカーネルを攻撃者にとって都合の良い動作をさせることを考える．そこで，カーネルを自由自在に操るために，攻撃者はカーネルスペースへ有害プログラムを組み込む．有害プログラムをカーネルスペースへ組み込む方法として，攻撃者はカーネルモジュールなどを利用する．

攻撃者は有害プログラムをカーネルに組み込むことで，システムコールに関連する機能を改ざんする．攻撃者がユーザレベルとカーネルレベルの間に存在する唯一のインタフェースとなるシステムコールに関連する機能を狙うことは当然であり，簡単に汚染することができる [6]．カーネル感染型有害プログラムの大部分は，システムコールの機能を改ざんすることで，攻撃者の活動や有害プログラムの存在を隠ぺいする．システムコールの機能を改ざんすることで，システム管理者はカーネルの情報を正しく取得できなくなる．

このように攻撃者がカーネル感染型有害プログラムを使用して汚染したシステムでは，システムコール実行に必要な総実行命令数やシステムコール実行に要する時間が増加するという特徴的な動作を見ることが可能である．本研究で作成したシステムを利用すれば，このようなカーネル感染型有害プログラムに特徴的な動作をリアルタイムに取得することができる．

本研究では，カーネル感染型有害プログラムによって汚染されたシステム上で見ることが可能な，システムコール実行時の総実行命令数や実行時間やが増加する，という特徴的な動作を監視することで，システムの異常検出を行った．

本研究で行う検出では，システムの情報取得に CPU に備わっている性能モニタリング機能を利用する．CPU の機能を利用することには様々な利点がある．ハードウェアを利用した検出を行うので，オペレーティング・システムの機能を利用した測定よりも，粒度が細かく正確なカウントが可能となる．

本研究では，Intel 社の NetBurst 系プロセッサ上で動作する，Linux カーネルの Red Hat ディストリビューション上で実験を行った．本研究での考え方は，NetBurst 系プロセッサ上で動作する他のオペレーティング・システムでも適用することができる．

本論文の構成は以下の通りとなっている．2 章では，有害プログラムと有害プログラム検出に関する議論を行い，本研究の目的に関して述べる．3 章では，有害プログラムに関する分類と分析を行う．はじめに，ユーザレベルで攻撃を行うユーザ感染型有害プログラムが行う攻撃手法，ユーザ感染型有害プログラムへの対処法に関して述べる．続いて，オペレーティング・システムのカーネルに対して攻撃を行うカーネル感染型有害プログラムが行う攻撃手法，カーネル感染型有害プログラムへの対処法に関して述べる．3 章の最後では，本研究で着目するカーネル感染型有害プログラムに汚染されたシステム上での特徴的な動作に関しても言及する．4 章では，関連研究に関して述べる．既存の侵入検知システムで利用されている技術，利点や問題点に関して議論を行う．5 章では，本研究で作成するカーネル感染型有害プログラム検出システムの設計に関して述べる．設計方針，システムに必要な要件，実現する機能に関して述べる．6 章では，カーネル感染型有害プログラム検出システムの実装に関して述べる．はじめに実装概要に関する議論を行い，実装したシステムの詳細を述べる．実装したシステムの有効性を示すために，カーネル感染型有害プログラムを用いて実験を行った．最後の 7 章では本研究のまとめを行う．

第 2 章

背景と目的

2.1 背景

ネットワークベースの計算機システムがますます重大な役割を果たすにつれて，計算機システムは攻撃者から侵入の目的となってきた [1]．重大な役割を果たすにも関わらず，現在の計算機システムは脆弱である．日々様々な脆弱性が発見され，攻撃手法や対策方法などに関する活発な議論が行われている [6]．

攻撃者がシステムに対して不正アクセスを行う理由として，計算機リソースの不正利用や破壊，DDoS 攻撃 [8] の踏み台，情報の不正入手や改ざんなどがあげられる．攻撃者は，これらを簡単かつ円滑に行うために，計算機利用者に対して有害な機能を備えたプログラムを作成し，利用する．このようなプログラムのことを有害プログラム [4, 5] という．代表的な有害プログラムには，ウィルス，ワーム，トロイの木馬などがあげられる．

従来の有害プログラムは，図 2.1 のようにユーザレベルで攻撃を行うものであった．しかし近年では，攻撃の技術がより洗練されてきており，図 2.2 のようにオペレーティング・システムのカーネルレベルで動作する有害なプログラムが出現しはじめた [6, 7, 13, 18]．攻撃者が不正アクセスを行う際に，この有害プログラムを使用することが問題となっている．

ユーザレベルで攻撃を行う有害プログラムは，主にユーザプログラムを改ざんする攻撃を行う．攻撃者はユーザプログラムを改ざんすることにより，システムの

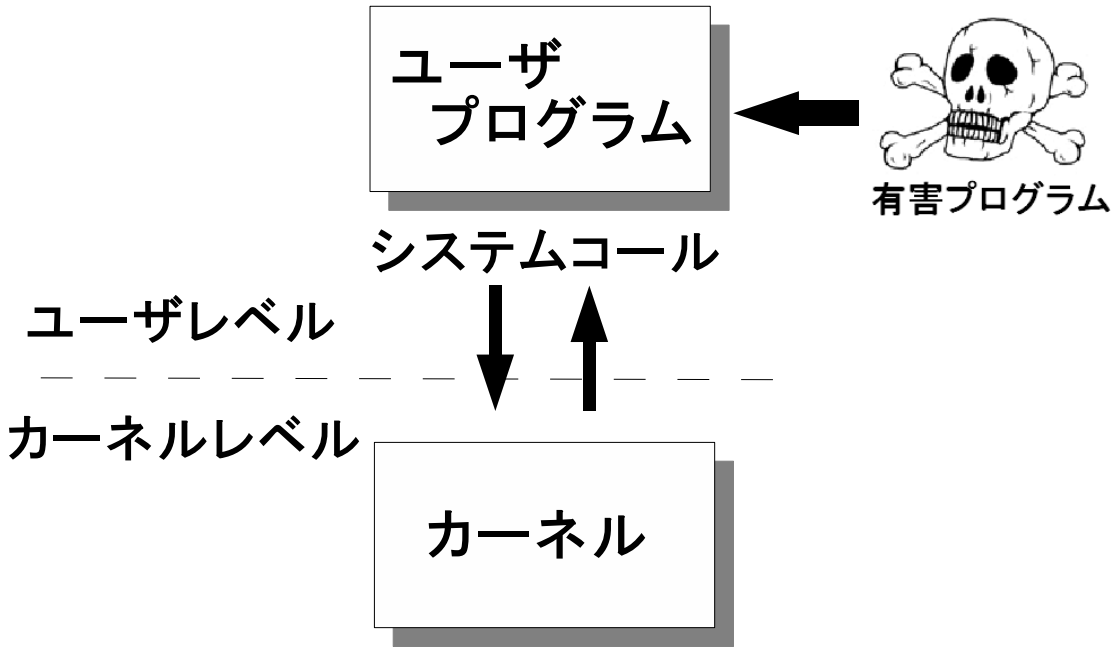

図 2.1: ユーザレベルで攻撃を行う有害プログラム

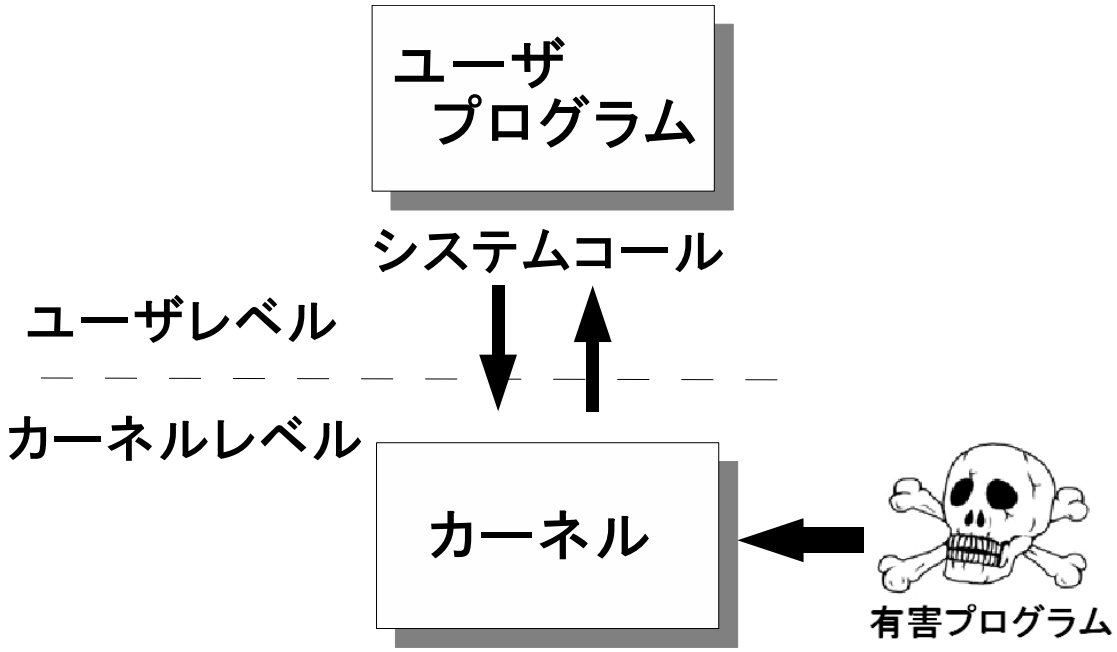

図 2.2: カーネルレベルで攻撃を行う有害プログラム

動作を攻撃者にとって都合の良い動作に変更する．この種の攻撃が行われた場合，改ざんされたプログラムを正常なプログラムと比較することで，システムの汚染を検出することが可能である．

これに対して，カーネルレベルで攻撃を行う有害プログラムは，ユーザプログラムを改ざんする攻撃を行わない．ユーザプログラムを改ざんする代わりに，ユーザプログラムが呼び出すシステムコールの機能を変更する．攻撃者がシステムコールの機能を変更することで，オペレーティング・システムの処理結果から汚染に関する情報を取り除くことなどが可能である．その結果，図 2.3 のように，システム管理者はカーネルの正しい情報を取得できなくなる．

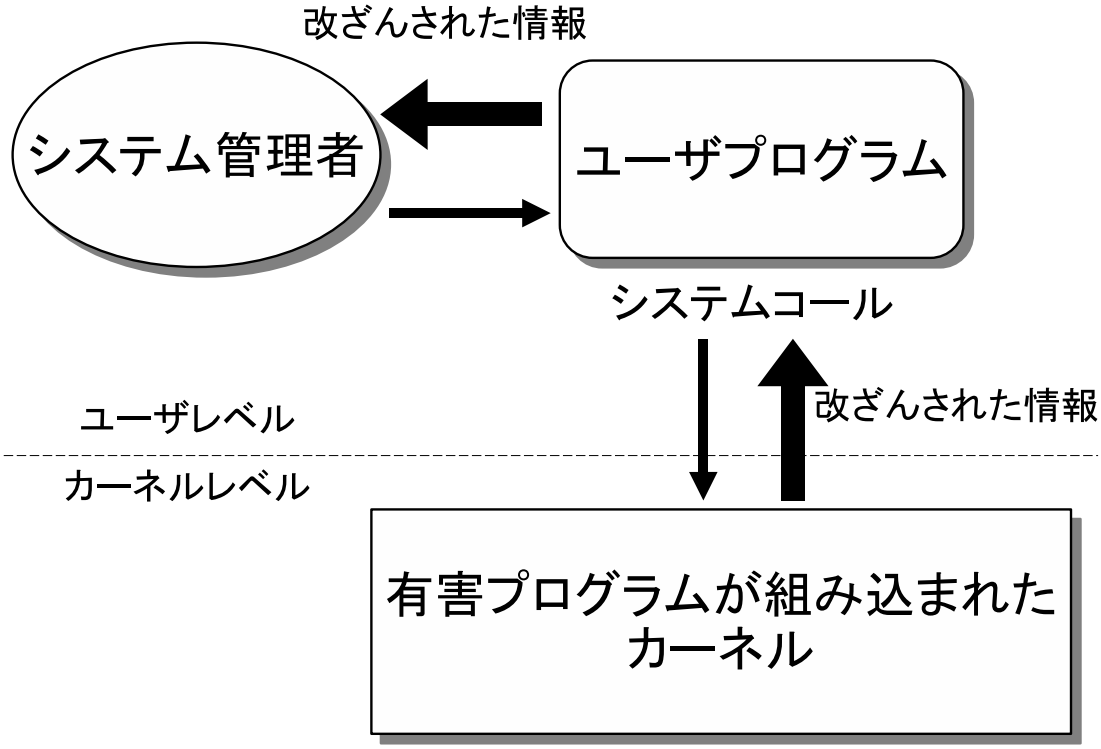

図 2.3: カーネル感染型有害プログラムの概念図

このような攻撃を受けた場合，ユーザプログラム自体には，正常時から一切変更が加えられない．それゆえ，ユーザプログラムを正常時のユーザプログラムと比較する方法によって，システムの汚染を検出することができない．

このように，カーネルレベルで攻撃を行う有害プログラムに対して，ユーザプログラムを利用したシステムの汚染検出には限界がある．攻撃者によってカーネルの処理結果を改ざんされてしまうため，ユーザが信頼できる情報を得ることができないからである．

攻撃者によってカーネル感染型有害プログラムを使用したシステムへの攻撃が行われると，システム管理者はシステムの汚染を検出することが非常に困難になる．カーネル感染型有害プログラムを絶対的に検出できるソフトウェアやシステムはいまだに登場していない．そのため，カーネル感染型有害プログラムを検出するためのシステムが必要とされている．

実際に，カーネル感染型有害プログラムによるシステムの汚染に気づくことはほとんどない．汚染が発覚したきっかけには，他のサイトからの警告によりシステムの汚染が分かったという事例が大多数を占める [9]．

このような背景から，本研究ではカーネル感染型有害プログラムを検出するシステムを設計し，実装を行った．

2.2 目的

本研究の目的は，オペレーティング・システムのカーネルに感染するカーネル感染型有害プログラムを検出するシステムを設計し実装することである．本研究での検出対象は，オペレーティング・システムのカーネルに感染し，システムコールに関連する部位に対する改ざんを行う有害プログラムとする．

攻撃者がカーネル感染型有害プログラムを使用して汚染したシステムでは，システムコール実行に必要な総実行命令数やシステムコール実行に要する時間が増加する，という特徴的な動作を見ることが可能である．本研究で作成するシステムでは，このような特徴的な動作に着目し，値を取得する．

値の取得には CPU に備わっている機能を利用する．CPU に備わっている機能

を利用することで，粒度が細かく正確な値を求めることができる．

本システムでは，攻撃者がカーネル感染型有害プログラム利用して汚染したシステムから，カーネル感染型有害プログラムをリアルタイムに検出できることを目指す．

第 3 章

有害プログラム分析

本章では，有害プログラムを分類し分析する．

有害プログラムが隠密活動をするための技術をステルス技術という．ステルス技術には，有害プログラム自体の隠ぺい，有害プログラムが実行されていることの隠ぺいなどがある．すべての有害プログラムがステルス技術を用いた隠密活動をするとは限らない．しかし，隠密活動は多くの有害プログラムに見られる特徴といえる．

代表的なユーザレベル感染型有害プログラムは，ウィルス，トロイの木馬，ワーム，ルートキットなどがある．ウィルスは，Frederick Cohen が 1984 年のセキュリティ会議で発表した論文 [32] で初めて定義された．ウィルスとは，自身のコピーを含ませるために，他のプログラムを修正することで感染するプログラムのことである．ワームとは，単独で自己増殖が可能で，ネットワークを利用して感染，増殖を行うプログラムのことである．トロイの木馬とは，有用な機能をもつように見えるが，潜在的に悪意のある機能を持つプログラムのことである．ルートキットとは，不正アクセスに利用できる有害なプログラムの集合体のことである．ルートキットには，アクセス維持や再侵入のためのバックドア [33]，プロセスや活動の隠ぺい，システム情報の信頼性破壊など，攻撃者に有用な機能が備わっている [6]．

有害プログラムには，ユーザレベルに感染して攻撃を行うもの，オペレーティング・システムのカーネルに感染して攻撃を行うものと，大きく 2 種類に分類することができる．

従来の有害プログラムはユーザレベルで感染して攻撃を行うものであった．しかし，近年ではカーネル感染型有害プログラムが出現してきた [7]．有害プログラムがカーネルに感染することで，攻撃者はシステム管理者からの検出を逃れようとする．

以下で，これらの有害プログラムに関しての分析を行っていく．ここでの攻撃に関する議論は，Unix のようなオペレーティング・システム上での議論のみを行う．しかし，他のすべてのオペレーティング・システムにおいてもこのような種類の攻撃が行われており，本論文での議論が適用できる．

3.1 ユーザレベル感染型有害プログラム

ここでは，ユーザレベルで感染して攻撃を行う，ユーザレベル感染型有害プログラムに関して述べる．攻撃者がユーザレベル感染型有害プログラムを用いて攻撃を行う場合，攻撃はユーザレベルで行われる．ユーザレベルでの攻撃手法には主に，有害なファイルの設置，ユーザプログラムのバイナリ改ざんなどが行われる．以下では，ユーザレベル感染型有害プログラムの攻撃手法，対策方法をそれぞれ述べる．

3.1.1 攻撃手法

ユーザレベル感染型有害プログラムを用いて，攻撃者はユーザプログラムのバイナリ改ざんを行う．攻撃者は図 3.1 のように，システム上に存在するユーザプログラムのバイナリを改ざんする．このバイナリは，攻撃者にとって都合の良い動作をするバイナリである．

この攻撃が行われてしまうと，有害なファイルやプロセスの隠ぺい，システム汚染の隠ぺい，システムコールに関わる機能の改ざん，ファイルサイズが増加していないように見せること，チェックサムをオリジナルのものから変化させないように

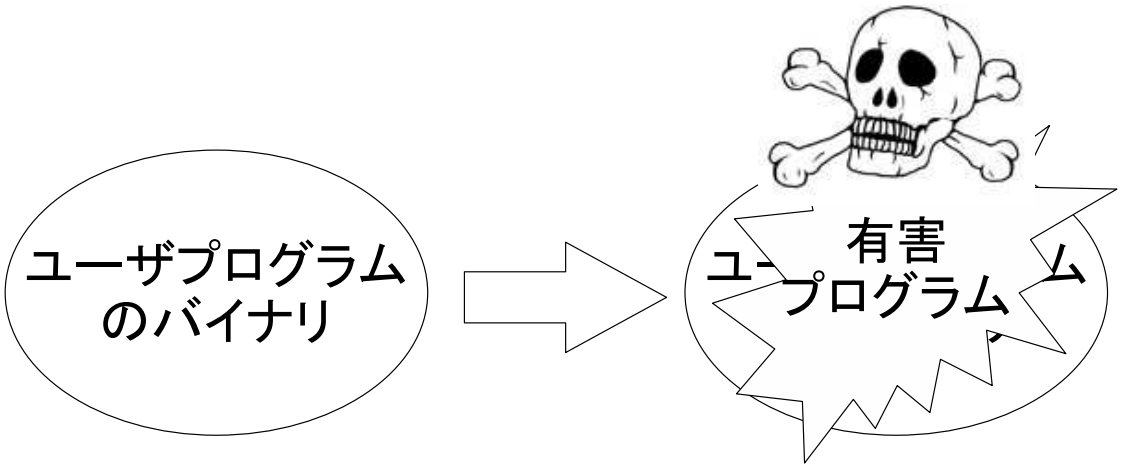

図 3.1: ユーザプログラムのバイナリを改ざんする有害プログラム

すること，カーネル構造を改ざんすることなどができる

攻撃者によって改ざんの対象となるユーザプログラムの多くは，/bin，/sbin，/usr/bin の中にあるシステム管理用のユーティリティである．

3.1.2 対策方法

ユーザレベル感染型有害プログラムを用いて，攻撃者はユーザプログラムのバイナリ改ざんを行う．このタイプの攻撃を受けた場合，改ざんされたユーザプログラムを正常なプログラムと比較することで，システムの異常を検出することが可能である [20]．ユーザプログラム比較する方法には，パターンマッチングによるバイナリの比較，ハッシュ値などによる整合性検査などがあげられる．

比較するために用意したユーザプログラムが正常であると保証されている場合，この攻撃に対処することができる．

3.2 カーネル感染型有害プログラム

ユーザレベル感染型有害プログラムに対して，カーネルレベルで攻撃を行うカーネル感染型有害プログラムが出現してきた [6, 7, 13, 18]．

カーネルレベルで攻撃を行うカーネル感染型有害プログラムは，ユーザプログラムを直接改ざんしない．ユーザプログラムを直接改ざんする代わりに，ユーザプログラムが呼び出すシステムコールの機能を改ざんする．したがって，ユーザプログラムは正常時と同じままシステムの機能を変更することができる．

初めて出現したカーネル感染型有害プログラムは，システムコールの機能を汚染することに狙いを定めた．攻撃者がユーザスペースとカーネルスペース間に存在する唯一のインタフェースとなるシステムコールに関連する機能を狙うことは当然であり，簡単に汚染することができる [6]．

以下では，ユーザレベル感染型有害プログラムの攻撃手法，対策方法をそれぞれ述べる．

3.2.1 攻撃手法

攻撃者がカーネルへ有害プログラムを組み込むと，システムコールに関連する機能を改ざんする．これより，以下のような攻撃ができる．

- 有害なファイルやプロセスの隠ぺい
- 改ざんの隠ぺい

攻撃者がカーネル感染型有害プログラムを用いてシステムへ行う攻撃には，カーネルへの侵入と改ざんの 2 つの段階がある．

カーネルへの侵入

カーネルへの侵入とは，攻撃者が有害プログラムをカーネルへ組み込むこととする．攻撃者が有害プログラムをカーネルへ侵入させる方法を，以下の 2 種類に分類する．

- カーネルモジュールを使用した侵入

- /dev/kmem を使用した侵入

以下で，それぞれの侵入方法に関して詳しく述べる．

カーネルモジュールを利用した侵入

カーネルモジュールを利用した侵入とは，ローダブルカーネルモジュール (LKM: Loadable Kernel Module) の技術を利用して，有害プログラムをカーネル中にカーネルモジュールとして組み込む方法である．ローダブルカーネルモジュールとは，デバイスドライバなどのモジュールを必要な場合にカーネルに読み込むことができ，不要になった場合にはカーネル上から削除できる機能である．ローダブルカーネルモジュールは特権モードで動作するため，カーネルが提供する機能の変更を行うことができる．

/dev/kmem を利用した侵入

/dev/kmem を利用した侵入とは，ユーザレベルのプロセスから悪意のあるコードをカーネルに組み込む方法である．このタイプの有害プログラムは，/dev/kmem ファイルを使用してカーネルメモリへ直接アクセスする．Linux カーネルは，/dev/kmem ファイルを使用して，ユーザスペースからカーネルメモリへアクセスする機能を提供する．ローダブルカーネルモジュールの使用が制限されているシステムでも，この方法を利用することにより，有害プログラムをカーネルへ組み込むことができる．

改ざん

カーネルスペースへの侵入後，攻撃者はシステムコールに関連した機能を改ざんする．改ざん方法を以下の 4 種類に分類する．

- システムコールの上書き

- システムコールテーブルの改ざん
- システムコールテーブル呼び出し先アドレスの上書き
- idtr レジスタの上書き

以下で，それぞれの改ざん方法に関して詳しく述べる．

システムコールの改ざん

システムコールの改ざんでは，個々のシステムコールを改ざんする (図 3.2) . システムコールを直接改ざんすることで，システムコールの機能を変更する．この攻撃は，システムコールテーブルへの改ざんは行わない．

システムコールテーブルの改ざん

システムコールテーブルの改ざんでは，攻撃者がシステムコールテーブルの一部もしくは全体を改ざんする．システムコールテーブル中にある個々のシステムコールの呼び出し先アドレスを，攻撃者が用意した新たなシステムコールへリダイレクトするように改ざんする (図 3.3) .

システムコールテーブル呼び出し先アドレスの改ざん

システムコールテーブル呼び出し先アドレスの改ざんでは，攻撃者がシステムコールテーブル全体をリダイレクトする . 割り込みディスクリプタテーブル (IDT : Interrupt Descriptor Table) の 0x80 番地にはシステムコールテーブルの呼び出し先アドレスが入っている . 攻撃者はこの番地から呼び出されるシステムコールテーブルのアドレスを，攻撃者が用意した新たなシステムコールテーブルへリダイレクトするアドレスへ改ざんする (図 3.4) . 元々あったシステムコールテーブルは改ざんしない．

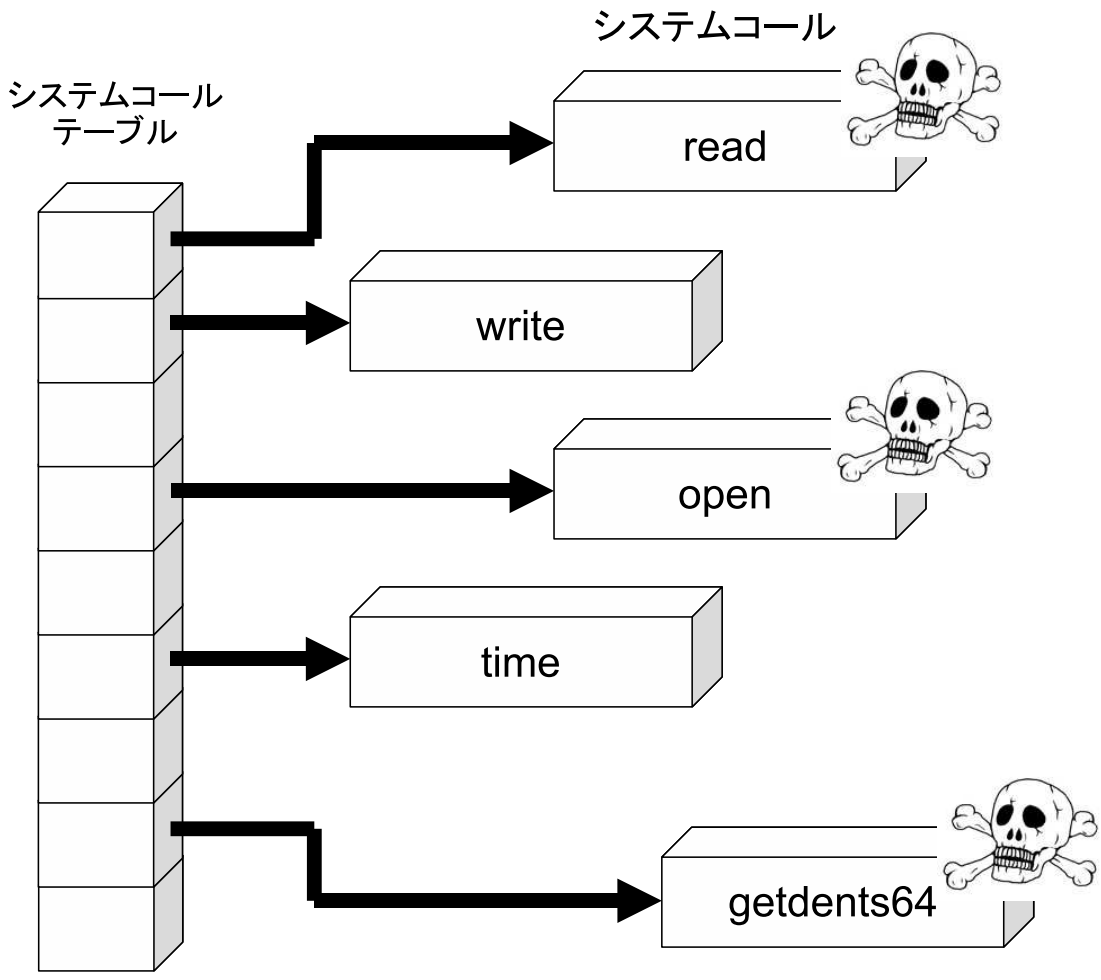

図 3.2: システムコールの改ざん

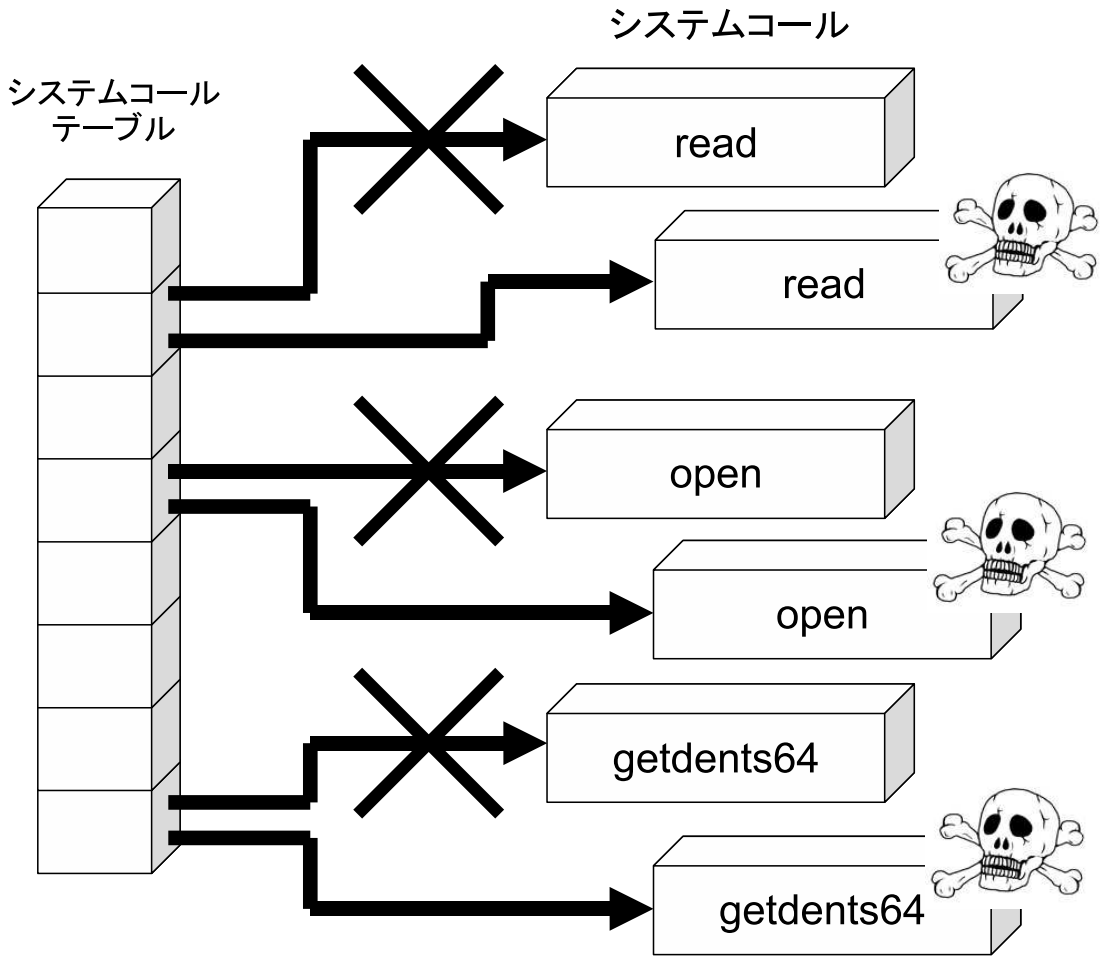

図 3.3: システムコールテーブルの改ざん

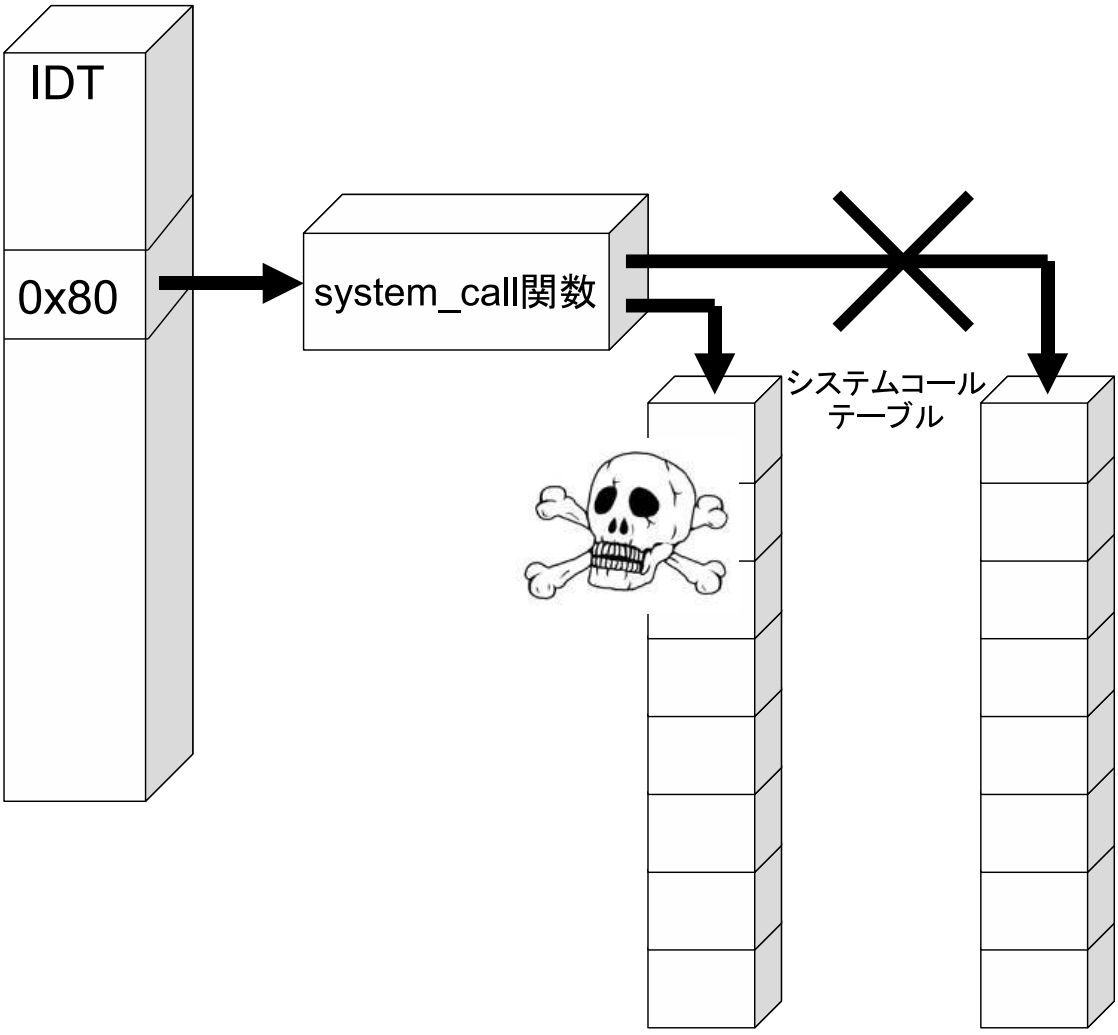

図 3.4: システムコールテーブル呼び出し先アドレスの改ざん

idtr レジスタの改ざん

idtr レジスタの改ざんでは，攻撃者が idtr レジスタに入っているアドレスを改ざんする．idtr レジスタには，割り込みディスクリプタテーブルの先頭アドレスがロードされる．そのアドレスを，攻撃者が用意した新たな割り込みディスクリプタテーブルへリダイレクトするアドレスへ改ざんする（図 3.5）．元々あった割り込みディスクリプタテーブルは改ざんしない．

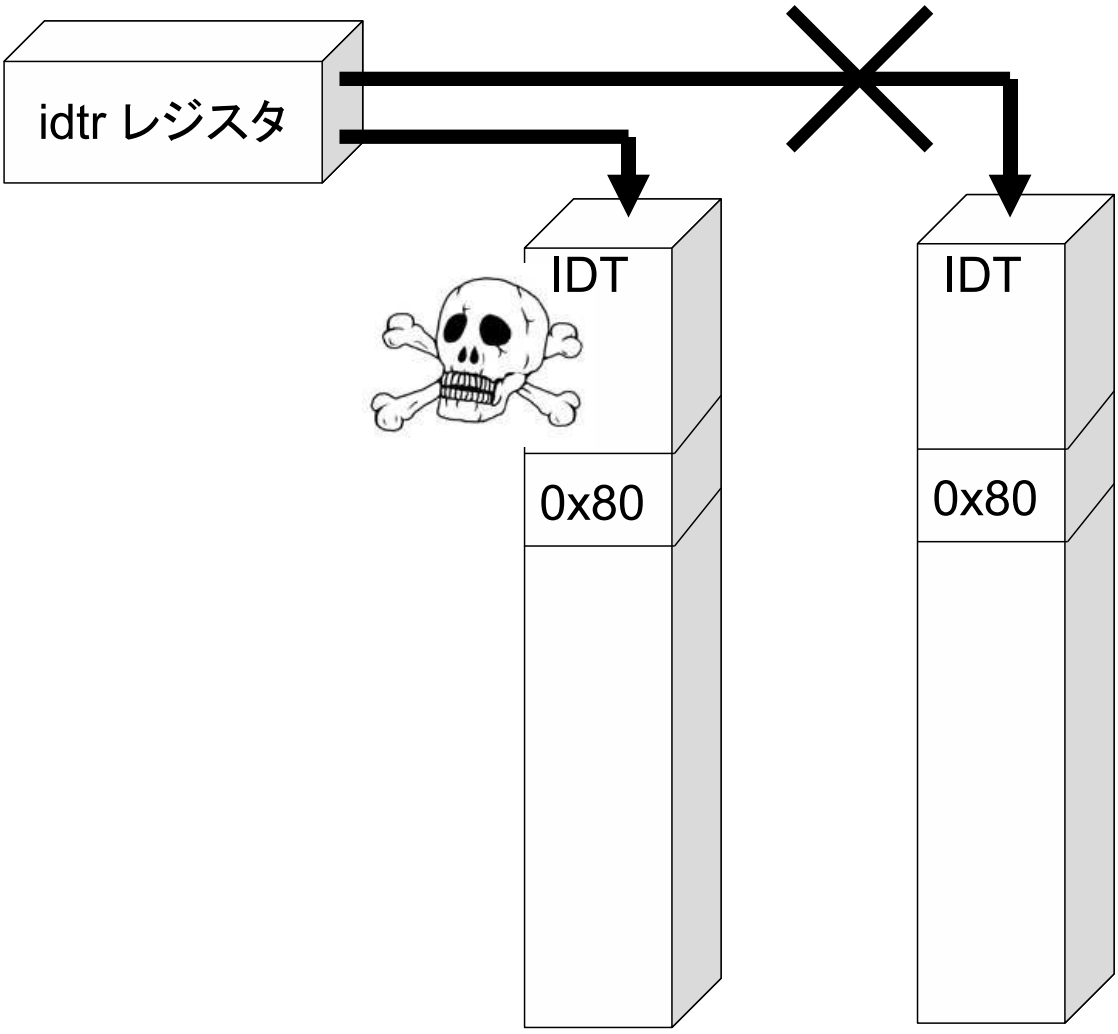

図 3.5: idtr レジスタの改ざん

システムコールの上書きを行う有害プログラム，idtr レジスタの上書きを行う有害プログラムはまだ登場していない．しかし，将来のカーネル感染型有害プログラムでは，これらの機能が実装される可能性がある．

3.2.2 対策方法

カーネル感染型有害プログラムへの検出には様々な対策がされてきたが，問題の解決には至っていない [6]．

既知のカーネル感染型有害プログラムの場合，デフォルトでインストールされるディレクトリ，ファイルを監視することで検出が可能である．しかし，カーネル感染型有害プログラムが既知のものであっても，攻撃者がカスタマイズすると特定のファイルやディレクトリの監視では検出できない．

システム状態を整合性ツールを使用して監視しても，システムの信頼性が失われるため，ツールが意図通りに機能しない．カーネル感染型有害プログラム検出ツールもいくつか提案されているが，どの対策手法も決定的な対策方法とはなり得ない．詳しくは関連研究のところで述べる．

3.2.3 カーネル感染型有害プログラムのサンプル

カーネル感染型有害プログラムの代表的なものに，以下の 4 つをあげることができる

- knark [10]
- Adore [10]
- Adore-ng [11]
- SucKIT [12]

これらのカーネル感染型有害プログラムは，カーネルへの攻撃手法を実証するために作成された物であり，インターネット上で公開されている．攻撃者はこれらの有害プログラムを利用する．

knark

knark は Creed によって作成されたカーネル感染型有害プログラムである．knark はローダブルカーネルモジュールを利用してカーネルへ侵入し，システムコールテーブルを改ざんすることでシステムを汚染する．knark には以下のような機能が備わっている．

- ファイル，ディレクトリの隠ぺい
- プロセスの隠ぺい
- ネットワーク接続の隠ぺい

Adore，Adore-ng

Adore は，Stealth によって作成されたカーネル感染型有害プログラムである．Adore はローダブルカーネルモジュールを利用してカーネルへ侵入し，システムコールテーブルを改ざんすることでシステムを汚染する．Adore には，knark と同様の機能が備わっている．

Adore は現在開発が終了しており，後継のバージョンである Adore-ng の開発が進められている．Adore-ng は，Adore をベースに Stealth が作成したカーネル感染型有害プログラムである．Adore-ng には，Adore の機能に加え，バーチャルファイルシステムに関係のあるシステムコールを汚染する機能が備わっており，カーネルバージョン 2.6 系の Linux で動作できる実装がされている．

SucKIT

SucKIT は，sd と devik によって作成されたカーネル感染型有害プログラムである．SucKIT は /dev/kmem を利用してカーネルへ侵入し，システムコールテーブル呼び出し先アドレスの上書きをすることでシステムを汚染する．SucKIT には knark と同じような機能を備えている．

3.2.4 カーネル感染型有害プログラムによる特徴的な動作

このようにカーネル感染型有害プログラムによって汚染されたシステムでは，システムコール実行に必要な総実行命令数，システムコール実行に要する時間が増加するという特徴的な動作を見ることが可能である [9, 13, 14]．

カーネル感染型有害プログラムはシステムコールに関連する機能を改ざんする．通常の処理の他に，有害プログラムによる追加の処理が発生する．この追加の処理により，システムコール実行に必要な総実行命令数，システムコール実行に要する時間が増加する．

例えば，システムの処理結果からある特定の値を隠ぺいするためには，通常処理の他に，隠ぺいのための追加の命令数を必要とする．追加の命令の影響で，システムコール実行に要する時間も増加する．

そこで，本研究では，このようなカーネル感染型有害プログラムが隠ぺい処理を行うときに生じる特徴的な動作に着目して検出を試みた．

第 4 章

関連研究

4.1 侵入検知システム

有害プログラムや新しい攻撃手法の広まりによって，侵入検知システム (IDS : Intrusion Detection System) [15, 16] の使用が活発になってきた．侵入検知システムは，計算機システムを安全に運用するために使用される [17]．近年では，侵入検知システムの使用により，計算機システムの安全性がこの 10 年ほどで大幅に改善されてきた [7]．

侵入検知システムとは，計算機システムに対する攻撃やユーザの不正な行為を検知するシステムである．侵入検知システムを警報装置のようなもの，とたとえることができる．警報装置は泥棒を防ぐことはできない．しかし，何か問題があれば異常を検知し，警告をすることができる．

新しい攻撃手法やコンピュータ使用環境の変化のために，導入された侵入検知システムはしばしばアップデートする必要がある．

検査データの中に明確な不正活動や侵入の証拠を記録できることが侵入検知の基本的な前提である [1]．

4.1.1 検出方法による分類

侵入検知システムは，検知方法の観点から分類をすると，以下の 2 種類に分類することができる．

- 不正検知
- 異常検知

不正検知

不正検知とは，既知の攻撃手法のパターンと一致したものを攻撃とみなす検知方法である．攻撃者による不正な行動を，あらかじめシグネチャというデータベースに登録しておく．ユーザの行動がシグネチャに合致した場合，侵入が起きたと検知する．

この検知方法の利点は，誤報が少なく検知の精度が高いことがあげられる．しかし，攻撃者の活動がシグネチャに登録されていない場合，攻撃を検知することは絶対にできない．

異常検知

異常検知とは，正常な状態の統計的特徴から逸脱しているものを攻撃とみなす検知方法である．ユーザーが行う通常の活動の傾向を，統計的に記録したプロファイルというデータをあらかじめ記録しておく．ユーザの行動がこのプロファイルから逸脱した場合，侵入が起きたと検知する．

この検知方法の利点は，不正検知では決して見つからないようなものも検知できるということがあげられる．しかし，何が異常かという事例収集，異常パターンの一般的記述方法が難しい．

4.1.2 取得する情報源による分類

侵入検知システムは，取得する情報源の観点から分類をすると，以下の 2 種類に分類することができる．

- ネットワーク型侵入検知システム

- ホスト型侵入検知システム

ネットワーク型侵入検知システム

ネットワーク型侵入検知システムは，ネットワーク上を流れるパケットを情報源として侵入検知を行う．パケットを収集し，ヘッダやペイロードなど，パケットの中身の情報を取得することで不正なパケットを検知する．

ネットワーク型侵入検知システムは，監視対象ホストまでの経路やネットワークのゲートウェイとなる地点に設置される．ネットワーク型侵入検知システムは，設置された場所から取得可能なパケットをモニタリングすることにより侵入を検知する．監視対象の外に配置されるため，監視対象ホストのリソースを消費することがないことが利点としてあげられる．ただし，今後ネットワーク上を流れるパケットがどんどん暗号化されていけば，ネットワーク型侵入検知システムでは検知できなくなる．

ホスト型侵入検知システム

ホスト型侵入検知システムは，ホスト上で得ることのできる情報を情報源として侵入検知を行う．ファイル，ディレクトリ，パーミッションに関する情報を収集し，ホスト上の不正なファイルを検知する．

ホスト型侵入検知システムは，監視対象となるホスト自身へ導入される．ホスト型侵入検知システムは，ホスト上で出力される様々な情報を用いて監視を行うことで侵入を検知する．様々な情報を利用して監視地点を多く設定することで，検知能力を上げることができる．今後ネットワーク上を流れるパケットがどんどん暗号化されていけば，ネットワーク上での監視ができなくなるため，最低限ホスト上監視をする必要がある．しかし，監視するホストすべてに導入する必要があり，侵入検知のためにホストの性能に負担をかけてしまう．

一般に，攻撃者に対して侵入を許してしまうと，侵入検知は非常に難しくなって

しまう [18]．ネットワーク型の侵入検知システムでは，侵入のための怪しい振る舞いを監視する．そのため，攻撃者の侵入後に監視を行っても，攻撃者による汚染を知ることができない．ホスト型の侵入検知システムでは，攻撃者がホストの機能を改ざんするため，改ざんされた情報を用いた検出を強いられる．そのため，攻撃者が侵入したホストを監視しても，攻撃者による汚染を知ることが難しい．

このように，攻撃者が侵入後の汚染検出は非常に難しい．本研究での検出対象であるカーネル感染型有害プログラムはネットワーク型侵入検知システムでは監視できない．そこで，本研究での取得する情報源としては，ホスト型でシステム汚染を監視する必要がある．

本研究で作成するシステムは，有害プログラムの検出方法の観点から異常検出に分類される．また，取得する情報源の観点からホスト型の侵入検知システムに分類される．

以下では，ホスト型の侵入検知システムとして用いられている関連研究に関して詳細に述べていく．

4.2 システムの整合性検査ツール

有害プログラム検出における最近の研究では，チェックサムを用いて正常なシステムと汚染されたシステム間での改ざんや追加されたファイル数などの差分に注目している [6, 9, 18]．昨今では攻撃技術が巧妙になり，攻撃者によってシステムが汚染された場所を特定することが非常に困難になってきている．このような状況を考慮すると，正常時との差分に注目することは妥当である．

システムの汚染場所特定のために，システムの差分を検査するためのツールを，ここではシステムの整合性検査ツールとする．

4.2.1 Tripwire

システムの整合性検査ツールの代表的なものに Tripwire [19] がある．Tripwire は，ファイルの作成や削除，ファイルサイズやパーミッションなどの変更を監視することができる．監視するファイルやディレクトリ情報のハッシュ値を，システムが正常である時にあらかじめデータベースへ保存しておく．そして，現在の状態とデータベースの内容の差分を取ることにより，システムの汚染検出を試みる．Tripwire の実行結果は，レポートとして保存することができる．そのほかに，指定したアドレスへメールを送信することもできる．

変更したはずのないファイルやディレクトリに対する検査を行い，Tripwire によって異常を検知されるということは，ファイルやディレクトリに対して何らかの改ざんが行われたとみなすことができる．

ユーザレベル感染型有害プログラムは，主に有害なファイルの設置やユーザプログラムのバイナリの改ざんを行う．有害なファイルを設置する攻撃が行われた場合，Tripwire を利用して正常時には存在しなかったファイルが設置されたという検出が可能である．ユーザプログラムのバイナリを改ざんする攻撃が行われた場合，Tripwire によって，現在システム上にあるユーザプログラムのバイナリが正常時のバイナリと異なるという検出が可能である．

4.2.2 Tripwire の限界

ユーザレベル感染型有害プログラムは，Tripwire を攻撃することが可能である．なぜならば，Tripwire で使用する実行ファイルやデータベースは監視するマシン中に存在するからである．攻撃方法として，以下のように 2 種類に分類することができる．

- Tripwire の実行ファイルを改ざんする攻撃
- システムの状態を記録したデータベースを改ざんする攻撃

攻撃者によって Tripwire の実行ファイルを改ざんする攻撃では，整合性検査を行っても必ず正常であるという結果しか返さない実行ファイルに変更することが可能である．このように Tripwire の実行ファイルの改ざんをすることで，攻撃者はシステム上で有害なファイルの設置やユーザプログラムのバイナリの改ざんを，Tripwire に検出されずに行うことができる．

システムの状態を記録したデータベースを改ざんする攻撃では，攻撃者が汚染後のシステムの状態を，正常時のシステム状態とであるようデータベースを改ざんすることが可能である．攻撃者は有害なファイルの設置やユーザプログラムのバイナリの改ざんによるシステム汚染を行った後に，汚染されたシステム状態を正常な状態としてデータベースに登録する．汚染された状態を正常状態であると認識させることで，Tripwire による検出を回避することができる．

これらの攻撃に対処するためには，Tripwire の実行ファイル，システムの状態を記録したデータベースを，CD などの安全なメディアにあらかじめ保存しておく．整合性検査を実行する際に，この正常であると保証された実行ファイル，データベースを利用して検出を行えばよい．

カーネル感染型有害プログラムは，Tripwire が使用するシステム機能の信頼性を破壊する攻撃が可能である．Tripwire の実行ファイル改ざんや，システムの状態を記録したデータベースの改ざんなどを行わずに，Tripwire による検出回避をすることが可能となる．

システムの整合性検査ツールは，オペレーティング・システムが正しく機能することに依存する．したがって，攻撃者がオペレーティング・システムのカーネルの機能を改ざんすると，簡単にシステムの整合性検査ツールを回避することが可能である [7]．

このように，カーネル感染型有害プログラムがシステムに組み込まれてしまうと，システムの整合性検査ツールを用いてのシステム汚染の検出を行うことができない．

4.3 有害プログラム検出ツール

有害プログラムを検出するためのツールがいくつか作成されている．以下では，それぞれのツールに関して述べていく．

chkrootkit [20] は，多くのルートキットやトロイの木馬などの有害プログラムを検出できるツールである．chkrootkit はパターンマッチングにより有害プログラム検出を試みる．カーネル感染型有害プログラムの場合，既知のものがカスタマイズされずに使用されていた場合のみ，chkrootkit での検出が可能である．

KSTAT [21] はカーネルのメモリ状態を，正常時と比較することにより，システムの状態を監視するツールである．kstat は /dev/kmem 経由でカーネルの情報を取り出す．2.2 カーネルでは System.map，2.4 カーネルでは make 時に取り出したカーネルの情報と比較することにより，システム汚染を検出する．カーネルが make 時に汚染されていると検出できない可能性もある，

KSTAT と似たような働きをするツールに kern_check [22] がある．kern_check は，システムコールアドレスを，カーネルコンパイル時の System.map から得られる値と現在の値を比較する．

alamo [23] は，ディレクトリエントリを操作するシステムコール getdents と同等の機能を持つツールである．攻撃者が getdents を汚染してリダイレクトすると，ファイルやディレクトリの情報を自由に操作することが可能である．alamo 経由でファイル情報を取得することで，getdents のみをリダイレクトする攻撃には対処が可能である．しかし，getdents 以外のシステムコールがリダイレクトされると無力である．

CheckIDT [24] は，システムコールの 0x80 エントリポイントである割り込みディスクリプタテーブルを復元させるツールである．

検出ツールが監視している部分を攻撃する有害プログラムに関しては，既存のツールで検出が可能である．ただし，どのツールも有害プログラムの一部の情報

を監視するため，監視場所以外の攻撃をされた場合，有害プログラムの検出をすることができない．

4.4 システムコールを監視する侵入検知システム

最近のホスト型侵入検知システムには，システムコールを監視するものが登場してきた．以下では，カーネル感染型有害プログラム検出に利用できる，システムコールを監視する侵入検知システムに関して述べる．

4.4.1 Execution Path Analysis

Execution Path Analysis [13, 14] のコンセプトは，システムサービス中に実行された命令数を数えることである．3.2.4 でも述べたように，攻撃者が有害プログラムを利用してシステム上のプロセスを隠ぺいする場合，システムコール実行に必要な総実行命令数が正常なシステムよりも多くなる．

命令数をカウントするために，Intel プロセッサのシングルステッピングモード [25] を利用する．Execution Path Analysis では，プロセッサをシングルステッピングモードにし，カーネルモードで実行される命令数をカウントする．このようにして Execution Path Analysis を行うと，監視するシステムコールの実行に必要な命令数をカウントすることができる．

Execution Path Analysis には問題点がある．命令数をカウントするためにシングルステッピングモードにすると，システムに大きな負荷をかけてしまう．そのため，Execution Path Analysis を利用してリアルタイムに監視をすることができない．

4.4.2 BlueBox

BlueBox [17] は，アプリケーションプログラムが使用するシステムコール実行を監視する．あらかじめアプリケーションプログラムが使用するシステムコールをポリシとして定義しておく．プログラムを実行するときに，正しいシステムコールが実行されているかどうかを監視する．

BlueBox では，ポリシに違反したシステムコール実行を検出することができる．しかし，適切なシステムコールを使用した攻撃者の活動を監視することができない．また，システムコールに関連する部位を攻撃者が改ざんを行っても，これを検知することができない．

4.4.3 Independent Auditor

Independent Auditor [7] では，PCI バスに接続された組み込みシステムを用いてシステムの動作を監視する．Independent Auditor では，システムコールやハードウェアの動作に対する監視が行える．

Independent Auditor は監視対象のホストの外部から監視を行うため，攻撃者による攻撃を受けにくい．しかし，Independent Auditor と監視対象のホストとの通信量が多くなると，ホストへの負荷が高くなる．また，監視するためには，ホストへ専用のハードウェアを導入する必要がある．

第 5 章

設計

5.1 設計方針

本研究で作成するシステムは，有害プログラムが隠ぺい処理時に示す特徴的な動作を手がかりとして，カーネル感染型有害プログラムを検出するシステムである．

3 章で述べたように，攻撃者がカーネル感染型有害プログラムを使用して汚染したシステムでは，有害プログラムによる隠密活動をするための隠ぺい処理が生じる．この隠ぺい処理のために，システムコール実行時に必要な総実行命令数や実行時間が正常時よりも増加する，という特徴がある．

そこで本研究では，Intel 社の NetBurst 系プロセッサ上に備わっている性能モニタリング機能を利用して，システムコール実行時の総実行命令数，実行時間をリアルタイムに取得する．取得した値を正常時との差を取ることにより，攻撃者によるシステム汚染の異常検出を行う．

リアルタイムに情報を取得することで，短い時間の改ざんでも検知することができる．システムの汚染がシステムの整合性検査ツールによるチェックが行われていない間に生じれば，システムの整合性検査ツールによって汚染されていることを検知できない．

本システムでの検出対象となる有害プログラムは，攻撃者によってカーネルへ組み込まれ，ユーザレベルからカーネルレベルへのインタフェースであるシステムコールに関連する部位に対して攻撃を行うものである．

5.2 性能モニタリング

Intel 社の 32 ビットマイクロプロセッサである Pentium 4 プロセッサ , P6 ファミリ・プロセッサ , および Pentium プロセッサには , モデル固有レジスタ (MSR : Model-Specific Register) が複数装備されている .

モデル固有レジスタは , その名が示すようにプロセッサ固有のものである . モデル固有レジスタは , ハードウェアとソフトウェアに関連したいろいろな機能を制御するものとして提供されている . モデル固有レジスタには , 以下に示すような機能が備わっている .

- 性能モニタリング・カウンタ
- デバッグの拡張
- マシン・チェック例外処理機能とマシン・チェック・アーキテクチャ
- メモリ・タイプ範囲レジスタ

これらの機能の 1 つである性能モニタリング・カウンタでは , さまざまなハードウェアイベントの計測を可能としている .

性能モニタリング・カウンタの機能を利用することで、プロセッサ性能パラメータの範囲をモニタし , 計測することができる [26] . 性能モニタリング・カウンタで計測できるイベントとして , 実行命令数 , プロセッサのクロックサイクル , 分岐命令数 , 分岐予測失敗数 , キャッシュミス数などをあげることができる .

5.2.1 P6 ファミリ・プロセッサにおける性能モニタリングの限界

P6 ファミリ・プロセッサをはじめとする現在利用されているプロセッサでは , 以下のような特徴がある .

1. 深いパイプライン構造

2. 投機的な実行

3. スーパスカラによる複数命令の同時実行

4. アウトオブオーダ実行

パイプライン構造とは，命令の読み込み，解釈，実行，結果の書き込みなど，各段階の処理機構を独立して動作させることにより，流れ作業的に，前の命令のサイクルが終わる前に次の命令を処理し始めることである．

投機的な実行とは，マイクロプロセッサの分岐予測に従って分岐予想先の命令をあらかじめ実行しておくことである．

スーパスカラとは，プロセッサの中に複数のパイプラインを用意し，複数の命令を並列に処理することである．

アウトオブオーダ実行とは，依存関係にない複数の命令を，プログラム中での出現順序に関係なく次々と実行することである．

これらの特徴などが原因で，イベントを発生させた命令とプログラムカウンタの値は必ずしも一致しないという問題がある．

この他にも，P6 ファミリ・プロセッサにおける性能モニタリング・カウンタには，以下に示すいくつかの限界があった [27, 28, 29]．

- キャンセルされた命令のイベントも計測する
- イベントサンプリングの粒度が荒い

P6 ファミリ・プロセッサでは，プロセッサが行った投機的実行が採用されなかった場合，命令をキャンセルする．しかし，P6 ファミリ・プロセッサの性能モニタリング・カウンタでは，キャンセルされた命令が引き起こしたイベントもカウントしてしまう．

イベントサンプリングをする場合，ある回数ごとに例外処理ルーチンが起動される．実際のプログラムカウンタや各種レジスタの情報を収集する時点では，プ

ロセッサの特性と例外処理ルーチンのレイテンシにより，取得したプログラムカウンタの値は実際のイベントを発生させたプログラムカウンタよりもかなり先を行っている [27, 28, 29]．

Dean らの報告 [30] によると，Pentium Pro プロセッサでのサンプリングを行った場合，Pentium Pro プロセッサで取得したプログラムカウンタと実際のプログラムカウンタとの隔たりは 25 命令以上あった．Sprunt の報告 [31] によると，Pentium 4 プロセッサでのサンプリングを行った場合，Pentium 4 プロセッサで取得したプログラムカウンタと実際のプログラムカウンとの隔たりは 65 命令以上あった．

5.2.2 Pentium 4 プロセッサにおける性能モニタリングの拡張

Pentium 4 プロセッサでの性能モニタリング・カウンタには，以下のような機能が拡張された．

- リタイヤメント時カウント機構
- 性能モニタリング・カウンタ高速読み取り機能
- カウント可能な性能イベントの種類
- 同時に計測できる性能モニタリング・カウンタ数

これらの機能により，以前のプロセッサでは不可能だった，より精密なプロセッサ状態のサンプリングが可能となった．

Pentium 4 プロセッサでは，実際に実行した命令だけ，あるいはキャンセルされた命令だけを計測する機能を提供する．この機能を利用することで，より粒度の細かい計測が可能となる．実際に実行した命令だけ計測する機能のことをリタイヤメント時カウント機構という．投機的に実行した命令が確定されるとリタイヤメントが実行される．

Pentium 4 プロセッサの性能モニタリング機構の中で，本研究で利用するものを以下に記す．

- 45 個のイベント選択制御レジスタ (ESCR : Event Selection Control Register) (図 5.1)
- 18 個のカウンタ設定制御レジスタ (CCCR : Cunter Configuration Control Register) (図 5.2)
- 18 個の性能モニタリング・カウンタ (PMC : Performance Monitoring Counter) (図 5.3)

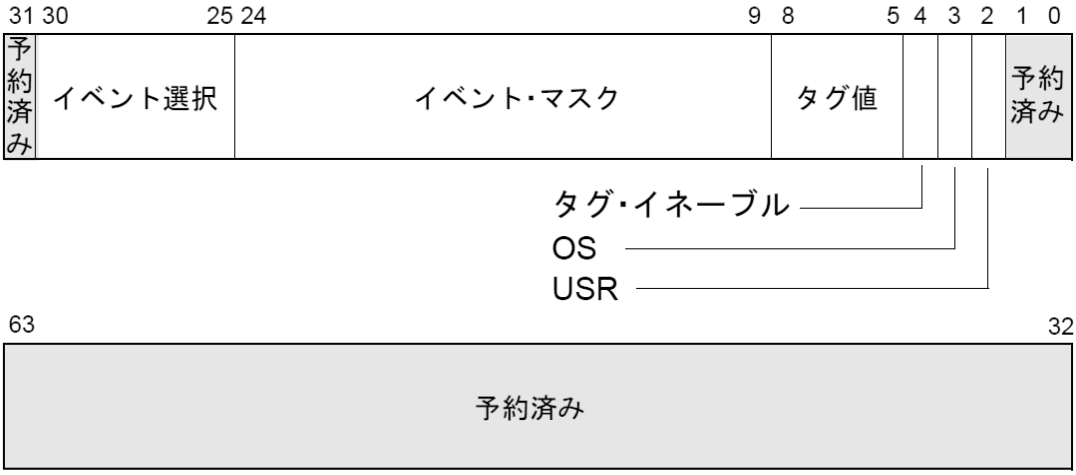

図 5.1: イベント選択制御レジスタ ([26] より転載)

性能モニタリング・カウンタでイベントをカウントするためには，次のように設定する．

1. イベント選択制御レジスタに計測すべきイベントを設定する．
2. カウンタ設定制御レジスタに計測方法と計測すべきイベントを設定したイベント選択制御レジスタを指定する．

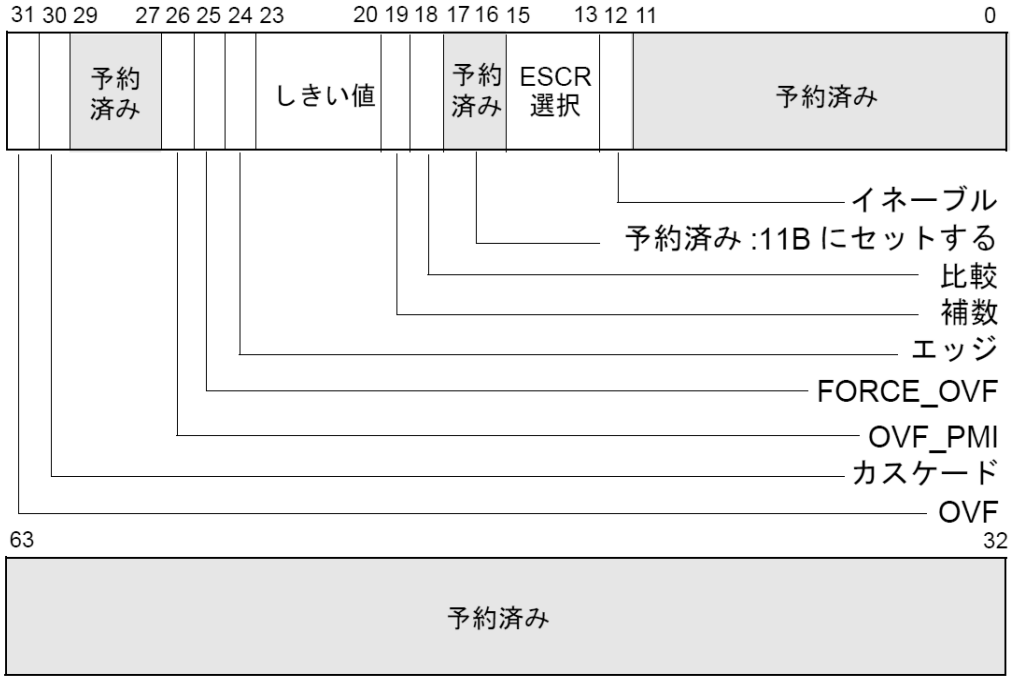

図 5.2: カウンタ設定制御レジスタ ([26] より転載)

図 5.3: 性能モニタリング・カウンタ ([26] より転載)

このようにして計測を開始すると，イベント数が性能モニタリング・カウンタに格納される．

5.2.3 Pentium 4 プロセッサにおける性能モニタリングの利用

本研究で行う測定は，このような Pentium 4 プロセッサに備わっている性能モニタリング機能を利用する．性能モニタリング・カウンタを用いて本研究で行うことを以下に記す．

- リタイヤメント時カウントによるシステムコール実行時の総実行命令数カウント．
- システムコール実行に要する時間のカウント．

システムコール実行時の総実行命令数カウント

性能モニタリング・カウンタでは，リタイヤメント時カウントをすることができる．今回は，instr_retired というイベントを選択し，カウンタのセットアップを行う．性能モニタリング機能を利用してリアルタイムに取得した値と，正常時に測定しておいた値を比較することにより，カーネル感染型有害プログラムを検出をする．

システムコール実行に要する時間のカウント

性能モニタリング・カウンタを使用して，クロックをカウントするように設定することができる．性能モニタリング機能を利用してリアルタイムに取得した値と，正常時に測定しておいた値を比較することにより，カーネル感染型有害プログラムを検出をする．

5.3 監視対象とするシステムコール

本システムでは，オーバヘッドを少なくするために，監視する対象のシステムコールを限定して情報を取得する．監視対象とするシステムコールは，有害プログラムを用いて攻撃者が狙う可能性の高いシステムコールとする．以下では，攻撃が行う汚染の種類とシステムコールの関係，攻撃の可能性があるシステムコール引数に関して述べる．

攻撃者が行うシステム汚染とシステムコールとの関係

攻撃者が行うシステム汚染とシステムコールとの関係を以下に記す．

- ファイル，ディレクトリ，プロセス，ネットワークなどの隠ぺい
  - open
  - getdents64
  - read
  - write
- ネットワークインタフェースのプロミスキャスフラグ隠ぺい
  - ioctl

攻撃者は，getdents, read，open，write などのシステムコールを攻撃することで，システム管理者が取得する情報を改ざんする．システム管理者がシステムの情報を取得するために利用するツールは，これらのシステムコールを使用する．それゆえ，システム管理者は信頼性のあるシステム情報を得ることができなくなる．

攻撃者がネットワークを盗聴するために，ネットワークインタフェースのプロミスキャスフラグをセットする．これにより，ホストへ到着する別のホスト宛のパケットも受信することができる．攻撃者はこのフラグを隠ぺいするために，ioctl システムコールを攻撃し，盗聴の事実を隠ぺいする．

攻撃の可能性があるシステムコール引数

カーネル感染型有害プログラムによって汚染されているシステム上で，先に記したシステムコールが呼び出されるときに，以下に記す引数が指定される場合に攻撃の可能性がある [10, 11, 12] ．

- /（ルートディレクトリ）
- /proc
- /dev/kmem
- /etc/passwd
- /proc/net/tcp
- /proc/net/udp

本研究では，監視対象のシステムコールがこれらの引数を指定して呼び出された場合，システムコール実行時間，総実行命令数を測定する．システムコールの種類は，システムコールテーブルを参照するときのシステムコール番号で判断をする．

5.4 検出アルゴリズム

本システムにおける検出アルゴリズムは，図 5.4 にあらわすように，以下の手順で行われる．

1. システムコールの発行
2. 発行されたシステムコール番号の保存
3. 発行されたシステムコールが監視対象であるかの判定
4. システムコールの引数が監視対象であるかの判定

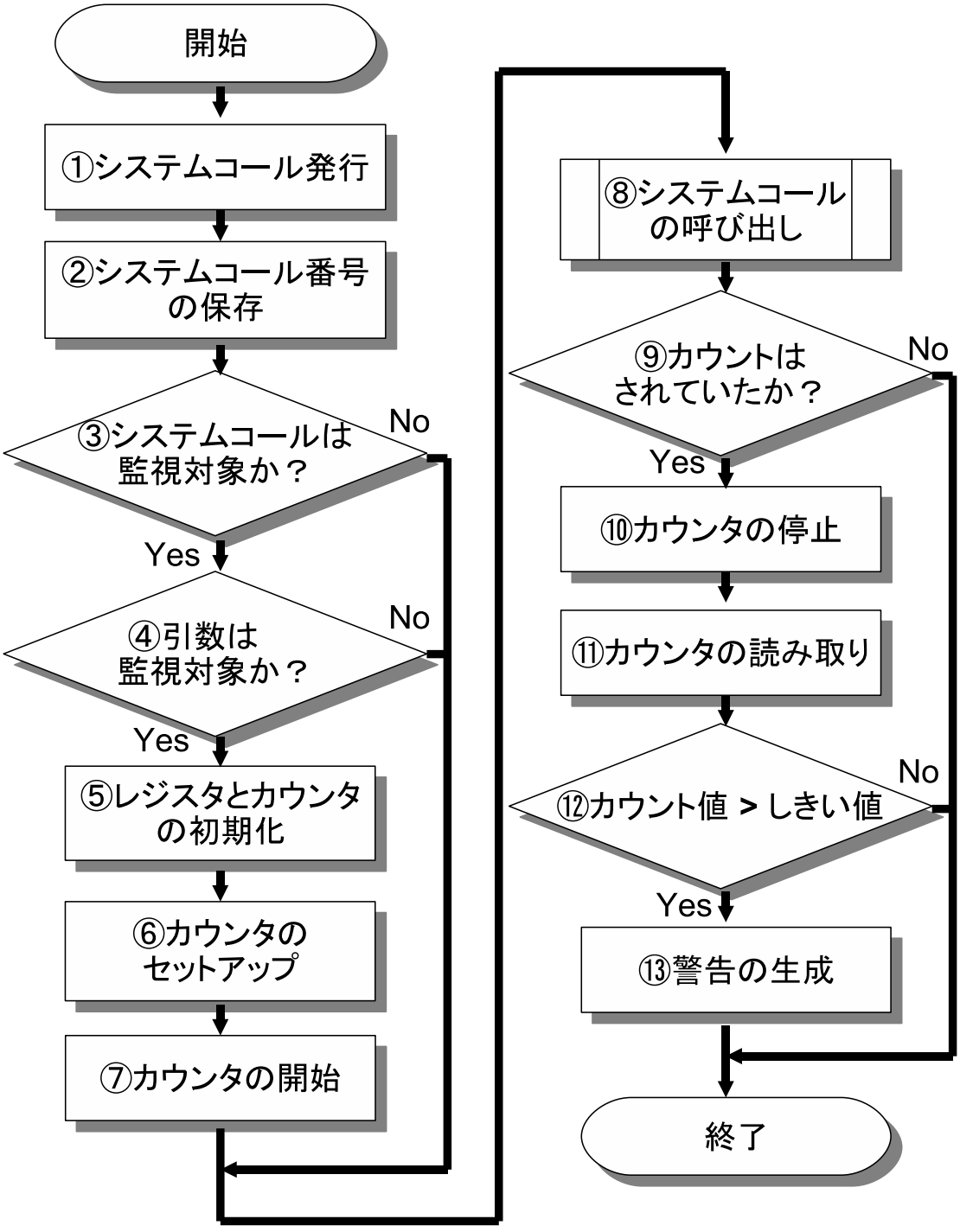

図 5.4: 検出アルゴリズム

5. レジスタとカウンタの初期化
6. カウンタのセットアップ
7. カウンタの開始
8. システムコール呼び出し
9. カウントされていたかどうかの判定
10. カウンタの停止
11. カウンタの読み取り
12. 取得した値としきい値の比較
13. 警告の生成

第 6 章

実装

6.1 実装概要

本章では，本研究で作成したカーネル感染型有害プログラム検出システムに関して議論する．本研究で作成したシステムは，オペレーティング・システムのカーネルを変更・拡張することで実装を行った．そこで，ソースコードが公開されている Linux オペレーティング・システムの Red Hat Linux version 9.0 (Kernel 2.4.20-8) を採用した．

本システムを実装した環境は，表 6.1 のようになっている．本システムでは Pentium 4 プロセッサになって拡張された性能モニタリング機能を利用する．そのため，CPU には Pentium 4 の 3.0 GHz を選定した．

表 6.1: 実験環境

オペレーティング・システム	Red Hat Linux version 9.0 (Kernel 2.4.20-8)
CPU	Intel Pentium 4 （3.00 GHz)
主記憶装置	1.0 GB

以下の節で，実装の詳細に関して述べていく．

6.2 性能モニタリング・カウンタの設定

Pentium 4 プロセッサに備わっている性能モニタリング機能を利用してカウントする場合，カウンタのセットアップが必要になる．以下では，本システムでカウントするシステムコール実行時に必要な総実行命令数カウント，システムコール実行時間のカウントに関して記す．

性能モニタリング・カウンタの設定では， Intel のマニュアル [26] に記載されている性能モニタリング・カウンタの対応表，リタイヤメント時カウント用性能モニタリング・イベントの表と照らし合わせながら設定する必要がある．必要となる項目を表 6.2，表 6.3 に記す．レジスタには WRMSR 命令を使用して値を設定する．

表 6.2: 性能モニタリング・カウンタの対応表

カウンタ			カウンタ設定制御レジスタ		イベント選択制御レジスタ		
名前	No.	番地	名前	番地	名前	No.	番地
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0	4	3B8H
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1	4	3B9H

6.2.1 システムコール実行時の総実行命令数カウント

Pentium 4 プロセッサの性能モニタリング・カウンタでは，リタイヤメント時カウントをすることができる．ここではリタイヤメントした命令をカウントするため， instr_retired というイベントを選択し，カウンタのセットアップを行う．

カウントするイベントの選択

性能モニタリング・カウンタでカウントするイベントを選択するには，以下の手順を行なう．

表 6.3: リタイヤメント時カウント用性能モニタリング・イベント

イベント名	イベント・パラメータ	パラメータ値
instr_retired		
	イベント選択制御レジスタ限定	MSR_CRU_ESCR0 MSR_CRU_ESCR1
	イベント選択制御レジスタごとのカウンタ番号	ESCR0: 12, 13, 16 ESCR0: 14, 15, 17
	イベント選択制御レジスタイベント選択	02H
	イベント選択制御レジスタイベント・マスク	ビット 0: NBOGUSNTAG タグ付けされていない リタイヤメントした命令
	カウンタ設定制御レジスタ選択	04H

1. カウントするイベントである instr_retired 選択する．
2. イベント選択制御レジスタ限定フィールドから，instr_retired をカウントするために使用するイベント選択制御レジスタ MSR_CRU_ESCR0 を選択する．
3. イベント選択制御レジスタごとのカウンタ番号フィールドから，イベントをカウントするために使用するカウンタの番号 12 を選択する．
4. 表 6.3 中の値を，図 6.1 のように WRMSR 命令を使用して，イベント選択制御レジスタ，カウンタ設定制御レジスタにそれぞれ値を書き込む．

```
/* set MSR_P4_ESCR */
wrmsr(0x3b8, (0x02 << 25) | (0x1 << 9) | (2 << 2), 0);

/* set MSR_P4_CCCR */
wrmsr(0x36c, (3 << 16) | (0x04 << 13), 0);
```

図 6.1: カウントするイベント選択の設定

Pentium 4 プロセッサでは，タグ付けの機能が拡張されているが，今回カウントするイベントにはタグ付けがされていない．したがって，イベント選択制御レジスタのイベント・マスクには，タグ付けされていないリタイヤメントした命令をマスクする NBOGUSNTAG を設定する．

カウントの開始

イベントのカウントを開始するには，以下の手順を行なう．

- 図 6.2 のように WRMSR 命令を使用して，カウンタ設定制御レジスタのイネーブル・フラグをセットする．

```
/* set MSR_P4_CCCR_ENABLE */
wrmsr(0x36c,  1 << 12, 0);
```

図 6.2: カウント開始の設定

カウンタ設定制御レジスタのイネーブル・フラグのセットは，図 6.1 に記したカウントするイベント選択の設定と同時に書き込むことができる．

カウントの読み取り

イベントのカウントを読み取るには，以下の手順を行う．

- 図 6.3 のように，カウンタ番号 12 を引数に指定して，RDPMC 命令を実行する．

```
/* read MSR_IQ_COUNTER0 */
rdpmc(0x80000011, instr_retired_count);
```

図 6.3: カウント読み取りの設定

ビット 31 がセットされている場合，RDPMC 命令は選択されたカウンタの下位 32 ビットのみを読み込む．ビット 31 がクリアされている場合，カウンタの全 40 ビットがすべて読み込まれる．Pentium 4 プロセッサでは，32 ビットの読み込みの方が高速に実行されるため，ここではビット 31 をセットする．

カウントの停止

イベントのカウントを停止するためには，以下の手順を行う．

- 図 6.4 のように WRMSR 命令を実行して，カウンタ設定制御レジスタのイネーブル・フラグをクリアする．ここではカウンタ停止のために全ビットをクリアしている．

```
/* clear MSR_P4_CCCR */
wrmsr(0x371, 0, 0););
```

図 6.4: カウント停止の設定

リタイヤメント時カウントでは，投機的に実行した命令の中で，リタイヤメントした作業を表すイベントだけがカウントされる．投機的に実行した実行された中で，リタイヤメントしなかった作業は無視される．

Pentium 4 プロセッサの性能モニタリング・カウンタは，RDPMC 命令または RDMSR 命令で読み取ることができる．これらの命令を使用すると，カウント中または，カウントが停止中に性能カウンタを読み取ることができる．

6.2.2 システムコール実行時間のカウント

性能モニタリング・カウンタを使用して，クロック数をカウントするように設定することができる．今回はリタイヤメントした命令に関してのクロック数を利用して時間をカウントする．そこで，instr_retired というイベントを選択し，カウンタのセットアップを行う．性能モニタリング・カウンタでクロックをカウントするには，図 6.5 のように WRMSR 命令を使用して，イベント選択制御レジスタ，カウンタ設定制御レジスタにそれぞれ値を書き込む．

```
/* set MSR_P4_ESCR */
wrmsr(0x3b9, (0x02 << 25) | (0x1 << 9) | (2 << 2), 0);

/* set MSR_P4_CCCR */
wrmsr(0x371, (0x0F << 20) | (0x01 << 19) | (0x01 << 18)
               | (3 << 16) | (0x04 << 13) | (1 << 12), 0);
```

図 6.5: カウント停止の設定

6.3 検出システムのアーキテクチャ

本検出システムは，以下に記す 5 つの機能から構成される．

- システムコールをリダイレクトするルーチン
- カウンタの開始をする関数

- カウンタの停止と読み取りをする関数
- カウンタの値を読み取り後の処理をする関数
- しきい値

以下の節で，それぞれの機能に関して述べる．

6.3.1 システムコールリダイレクトによるカウント

本システムでは，システムコールをリダイレクトして，システムコール実行に必要な総実行命令数，システムコール実行時間をカウントする．この機能を実現するために，システムコール呼び出し部分である entry.S を図 6.6 のように変更する．追加した箇所は，3 行目，10 ~ 12 行目，15 ~ 18 行目である．

3 行目では，呼び出されるシステムコールのシステムコール番号を sys_call_num 変数に保存している．この値を利用して，監視対象のシステムコールかどうかの判定を行う．また，カウントした値を比較するために利用するしきい値のテーブルを検索する用途にも，この値を利用する．

本システムで作成した関数がレジスタを破壊しないように，10 行目，15 行目で EAX レジスタの値を退避し，12 行目，18 行目で退避したレジスタを戻している．

11，16，17 の各行でカウントを開始する count_start 関数，カウンタの停止と読み取りをする count_read_and_stop 関数，カウンタのカウンタの値を読み取り後の処理をする after_operation 関数をそれぞれ呼び出している．

6.3.2 カウンタの開始

カウンタの開始をする関数名は，count_start とする．この関数では，以下のような処理を行う．

- システムコール監視の判定

```
1  ENTRY(system_call)
2   pushl %eax                    # save orig_eax
3   mv %eax,sys_call_num
4   SAVE_ALL
5   GET_CURRENT(%ebx)
6   testb $0x02,tsk_ptrace(%ebx)  # PT_TRACESYS
7   jne tracesys
8   cmpl $(NR_syscalls),%eax
9   jae badsys
10  pushl %eax        # SAVE EAX Register for counter
11  call count_start  # count start
12  popl %eax         # RESTORE EAX Register for counter
13  call *SYMBOL_NAME(sys_call_table)(,%eax,4)
14  movl %eax,EAX(%esp)  # save the return value
15  pushl %eax           # SAVE EAX Register for counter
16  call count_read_and_stop # count read and count stop
17  call after_operation     # after operation
18  popl %eax         # RESTORE EAX Register for counter
```

図 6.6: システムコールリダイレクトのルーチン

- システムコール引数の判定
- レジスタの初期化
- カウンタの初期化
- カウンタのセットアップ
- カウンタの開始

システムコールが発行されると，count_start 関数が呼び出される．count_start 関数では，はじめに発行されたシステムコールが監視の対象であるかどうかの判定を行う．監視対象のシステムコールが発行された場合，count_execution フラグを立てる．発行されたシステムコールが監視の対象でなかった場合，以降の処理は行われない．

ここでのシステムコール監視の判定は，システムコール番号によって判断する．カウント対象のシステムコール番号を保持することで，事後処理でしきい値を読み込む際にも利用する．

count_execution フラグが立っている場合，システムコール引数の判定を行う．システムコールの引数が監視対象である場合，dir_flag フラグを立てる．

count_execution フラグ，dir_flag フラグのフラグが共に立てられているときに，レジスタの初期化，カウンタの初期化，カウンタのセットアップ，カウンタの開始をそれぞれ行う．レジスタの初期化，カウンタの初期化はそれぞれに 0 を書き込むことによって行う．カウンタのセットアップ，カウンタの開始は，6.2.1, 6.2.2 に記した手順で行う．

6.3.3 カウンタの停止と読み取り

カウンタの停止とカウンタの読み取りをするための処理の関数名は count_read_and_stop とする．この関数では，以下のような処理を行う．

- カウンタが開始されているかどうかの判定
- カウンタの停止
- カウンタの読み取り

システムコール実行後 ,count_read_and_stop 関数が呼び出される .count_read_and_stop 関数では，はじめに発行されたシステムコールがカウントされていたかどうかの判定を行う．システムコールがカウントされていた場合，カウンタの停止を行う．カウンタの停止を行なったのち，カウント値保存用の変数に取得した値を代入する．システムコールがカウントされていなかった場合，これらの処理を行わない．

6.3.4 カウンタ読み取り後の処理

カウンタ読み取り後の処理をするための関数名は after_operation とする．この関数では，以下のような処理を行う．

- カウンタの読み取りが行われたかどうかの判定
- 取得した値としきい値の比較
- 警告の発行

count_read_and_stop 関数の後に ,after_operation 関数が呼び出される .after_operation 関数では，はじめにカウンタの読み取りが行われたかどうかの判定を行う．カウンタの読み取りが行われていた場合，カウンタで取得した値とシステムコール番号に応じたしきい値を比較する．しきい値を超えていた場合に，コンソールへアラートを表示する．しきい値を超えていなかった場合，何もしない．

6.3.5 有害プログラム検出に使用するしきい値

カウントした値を比較するために利用するしきい値は，システムが正常であるときにあらかじめ取得しておく必要がある．本研究では，システムが正常である

ときに，監視対象とするシステムコールを呼び出すプログラムを実行することで値を取得した．

表 6.4，表 6.5 にシステムが正常時に取得した値を記す．表の値はシステムコールを 100 回実行した時の平均と分散である．また，getdents64 システムコール，read システムコールを実行する場合，はじめに引数のファイルを open システムコールで開いた上で実行している．なお，ここでの実行命令数は回数，実行時間はクロック数であらわされている．

表 6.4: 総実行命令数のしきい値

システムコール	引数	総実行命令数	分散
open	/dev/kmem	1465	1
open	/etc/passwd	1482	0
getdents64	/	9208	553676
getdents64	/proc	33951	1
read	/dev/kmem	1305	10354
read	/proc/net/tcp	1705496	824609
read	/proc/net/udp	1286	0

表 6.4 の結果より，/etc/kmem を引数とした open システムコール，/etc/passwd を引数とした open システムコール，/proc を引数とした getdents64 システムコールが発行される場合，分散が小さく，総実行命令数カウントによるシステムの汚染検出にしきい値として利用できる．しかし，それ以外のシステムコールと引数の組み合わせは分散が大きいため，総実行命令数カウントによるシステムの汚染検出の指標とすることができない．

以下では，/etc/kmem を引数とした open システムコール，/etc/passwd を引数とした open システムコール，/proc を引数とした getdents64 システムコールが発行される場合に，総実行命令数カウントによるシステムの汚染検出を行う．

表 6.5: 実行時間のしきい値

システムコール	引数	実行時間	分散
open	/dev/kmem	4242	44
open	/etc/passwd	3770	279
getdents64	/	30941	18134216
getdents64	/proc	110759	1496591
read	/dev/kmem	10845	4652095
read	/proc/net/tcp	3349358	17965741669
read	/proc/net/udp	3070	1972331

また，表 6.5 の結果より，/etc/kmem を引数とした open システムコール，/etc/passwd を引数とした open システムコールが発行される場合，分散が小さく，実行時間カウントによるシステムの汚染検出にしきい値として利用できる．しかし，それ以外のシステムコールと引数の組み合わせは分散が大きいため，実行時間カウントによるシステムの汚染検出の指標とすることができない．

以下では，/etc/kmem を引数とした open システムコール，/etc/passwd を引数とした open システムコールが発行される場合に，実行時間カウントによるシステムの汚染検出を行う．

6.4 カーネル感染型有害プログラム検出実験

本節では，本研究で作成されたシステムを利用して，カーネル感染型有害プログラムを検出する実験を行う．本システムでは，システムを利用して取得したシステムコール実行時の総実行命令数と実行時間を，正常時に取得しておいた値と比較することにより，システムの汚染を検出する．

カーネル感染型有害プログラム検出実験では，packet storm [10] や Stealth のサイト [11] などで公開されているカーネル感染型有害プログラムを用いて実験を行う．

6.4.1 knark

knark は，システムコールテーブルの改ざんによって，ファイル，ディレクトリ，プロセスなどの隠ぺいを行うカーネル感染型有害プログラムである．knark が感染したシステム上で取得した値を表 6.6 に記す．

表 6.6: knark が感染したシステム上での実験結果

システムコール	引数	総実行命令数		実行時間	
		正常時	汚染時	正常時	汚染時
open	/dev/kmem	1465	1466	4242	4076
open	/etc/passwd	1482	1482	3770	3811
getdents64	/proc	33951	39565	—	—
read	/proc/net/udp	1286	1309	—	—

knark が感染したシステム上で取得した表 6.6 の結果より，/proc を引数とする

getdents64 システムコールが発行されたとき，総実行命令数が約 5600 命令増加していることが分かる．このことから，攻撃者がシステム汚染で使用した有害プログラムにより，getdents64 システムコールが発行されたときに，命令のリダイレクトや追加の処理が行われていると判断することができる．

実際に knark は，getdents64 システムコールをリダイレクトすることにより，プロセスなどの隠ぺいを行っている．リダイレクトすることで，システムコール実行時の総実行命令数，実行時間が正常時よりも増加している．

/proc/net/udp を引数とした read システムコールが発行されたとき，23 命令増加している．しかし，/proc を引数とする getdents システムコールが発行された場合と比較して，はっきりと命令数が増加したわけではない．knark は/proc/net/udp を引数とした read システムコールに対するリダイレクトを行わない．しかし，getdents64 システムコールをリダイレクトすることにより，/proc/net/tcp 中の文字列を隠ぺいする．この影響で，/proc/net/udp を引数とした read システムコール発行時の命令数が若干増加したものと考えられる．

以上の結果より，システムコールテーブルの改ざんによって getdents64 システムコールをリダイレクトし，ファイル，ディレクトリ，プロセスなどの隠ぺいなどを行うカーネル感染型有害プログラムを検出できることが示せた．

6.4.2 Adore-ng

Adore-ng は，knark と同様に，システムコールテーブルの改ざんを行うことによって，ファイル，ディレクトリ，プロセス，ネットワーク接続などの隠ぺいなどを行うカーネル感染型有害プログラムである．Adore-ng が感染したシステム上で取得した値を表 6.7 に記す．

Adore-ng が感染したシステム上で取得した表 6.7 の結果より，/proc を引数とする getdents64 システムコールが発行されたとき，総実行命令数は約 72400 命令増加していることが分かる．このことから，攻撃者がシステム汚染で使用した有害

表 6.7: Adore-ng が感染したシステム上での実験結果

システムコール	引数	総実行命令数		実行時間	
		正常時	汚染時	正常時	汚染時
open	/dev/kmem	1465	1466	4242	4042
open	/etc/passwd	1482	1482	3770	4007
getdents64	/proc	33951	106390	—	—
read	/proc/net/udp	1286	1286	—	—

プログラムにより，getdents64 システムコールが発行されたときに，命令のリダイレクトや追加の処理が行われていると判断することができる．

Adore-ng は knark と同様に，getdents64 システムコールをリダイレクトすることにより，プロセスなどの隠ぺいを行っている．リダイレクトすることで，システムコール実行時の総実行命令数，実行時間が正常時よりも増加している．

以上の結果より，Adore-ng のようにシステムコールテーブルの改ざんによって，ファイル，ディレクトリ，プロセス，ネットワーク接続などの隠ぺいなどを行うカーネル感染型有害プログラムを検出できることが示せた．

6.4.3 SucKIT

SucKIT は，/dev/kmem 経由でカーネルへ侵入し，システムコールテーブル呼び出し先アドレスの改ざんをすることによって，ファイル，ディレクトリ，プロセス，ネットワーク接続の隠ぺいなどを行うカーネル感染型有害プログラムである．

SucKIT が感染したシステム上で取得できる値を表 6.8 に記す．

SucKIT が感染したシステム上で取得した表 6.8 の結果より，/dev/kmem を引数

表 6.8: SucKIT が感染したシステム上での実験結果

システムコール	引数	総実行命令数		実行時間	
		正常時	汚染時	正常時	汚染時
open	/dev/kmem	1465	4692	4242	16369
open	/etc/passwd	1482	4704	3770	7209
getdents64	/proc	33951	406162	—	—
read	/proc/net/udp	1286	4660	—	—

とする open システムコールが発行されたとき，総実行命令数は約 3200 命令，実行時間は約 12000 クロック増加している．また，/etc/passwd を引数とする open システムコールが発行されたときに，総実行命令数は約 3200 命令，実行時間は約 2500 クロック増加していることが分かる．

また，/proc を引数とする getdents64 システムコールが発行されたとき，総実行命令数は約 372000 命令増加していることが分かる．/proc/net/udp を引数とした read システムコールが発行されたとき，総実行命令数は約 3400 命令増加している．

これが生じた原因は，SucKIT によるシステムコールテーブル呼び出し先アドレスの改ざんにより，攻撃者が用意したシステムコールテーブルを参照させるようにしたことが原因である．

以上の結果より，SucKIT のようにシステムコールテーブル呼び出し先アドレスの改ざんによって，ファイル，ディレクトリ，プロセス，ネットワーク接続の隠ぺいなどを行うカーネル感染型有害プログラムを検出できることが示せた．

6.5 考察

ここでは，本研究で作成したシステムによるカーネル感染型有害プログラム検出に関しての考察を行う．ここでは，はじめに本システムを利用すればできることを述べ，本システムでは対処できない問題と今後の課題に関して議論する．

6.5.1 本システムを利用してできること

本システムを利用してできることを以下に記す．

- システムコール実行時の総実行命令数を取得すること．
- システムコール実行時間を取得すること．
- システムコールに関連する部位への汚染を包括的かつリアルタイムに検出できること．
- システムの汚染場所特定のために有益な情報を提供すること．

本研究で作成したシステムでは，CPU の機能を利用してシステムコール実行時の総実行命令数と実行時間を取得することができる．これらの情報を利用することにより，攻撃者がカーネル感染型有害プログラムによるシステム汚染を行ったかどうかを検出することができる．本システムでは，値の取得に CPU の機能を利用した．これにより，粒度が細かい正確なカウントが可能である．

本システムでの値の取得と評価はリアルタイムに行われる．それゆえ，攻撃者によるシステムの汚染が非常に短い時間であったとしても，システムの汚染を検出することができる．

本システムで取得した値は，システムの汚染が生じた場合に汚染場所の特定に有用な情報を提供することが可能である．警告を生成した原因のシステムコールと引数に関する情報を手がかりに，システム管理者はシステムの汚染場所を行うことができる．

6.5.2 本システムでは対処できない問題

本システムでは対処できない問題を以下に記す．

- 検出システムへの直接攻撃
- 検出システムが利用するカーネル関数への攻撃
- カーネルを丸ごと改ざんする攻撃
- カーネルのデータ構造改ざんに対する攻撃
- 監視していないシステムコールへの攻撃
- しきい値の自動更新
- 性能モニタリング機能のない CPU 上での動作

本システムでは，検出システムへの攻撃に関しては対処できない．検出システムへの攻撃には，カウント機能の無効化，しきい値の改ざん，警告を生成する箇所の無効化などが考えられる．攻撃者によって検出システムが攻撃されないように，これらの問題に対する保護をすべきである．検出システムはすべてホスト上に設置されるため，ホスト上で安全にデータを保護する仕組みが必要とされる．ホスト上でのデータを安全に保護にする研究成果が望まれる．

本システムは，ホスト上のカーネル関数を利用して実装している．本システムが利用するカーネル関数が攻撃者によって汚染されてしまうと，システムの状態を正しく取得できなくなる．また，攻撃者がカーネル全体を改ざんする攻撃にも対処できない．本システムは，利用するカーネル関数が正しく機能する場合のみシステム汚染を検出できる．

今回の実装では余計なオーバヘッドを回避するために，既知の有害プログラムが利用したことのあるシステムコールと引数のみに監視対象を絞った．監視していないシステムコールへの攻撃が行われた場合，検出システムは意味をなさない．

本システムを導入している計算機システムのシステム状態が更新されると，検出のためにに利用するしきい値の更新が必要となる．しきい値はシステム状態が変わるごとに手動で再取得する必要がある．

本システムでの値の取得は，CPU に備わっている性能モニタリング機能を利用している．当然のことながら，性能モニタリング機能がない CPU 上ではシステムを動作させることができない．

6.5.3 オーバヘッド

本システムを利用してシステムコールの監視を行った場合のオーバヘッドを time コマンドによって計測した．それぞれのシステムコールと引数につき 3 回計測し，平均を取った．計測を行った結果を，表 6.9 に記す．

表 6.9: オーバヘッドの測定

システムコール	引数	監視無し	監視あり	オーバヘッド
open	/dev/kmem	2.26s	4.6s	104%
open	/etc/passwd	2.68s	4.99s	86%
getdents64	/proc	4.46s	6.32s	42%
read	/proc/net/udp	0.113s	2.63s	2227%

表 6.9 から分かるように，システムコールの監視によってオーバヘッドが生じている．今回の実装では，システムコールが発行されるごとに毎回監視を行っているため，このような大きなオーバヘッドを生む結果となった．今後は，システムコールの監視が必要なときにだけ検出システムを動的に組み込めるようにする，システムコール発行ごとではなく，数回につき 1 回カウントを行うような工夫をする

ことで，オーバヘッドを削減することを考えなければならない．

6.5.4 今後の課題

本システムは，Linux オペレーティング・システム上に作成した．Limux オペレーティング・システムの制約から，カーネルを書き換えることにより実装を行った．検出に利用するしきいの値更新，監視するシステムコールの追加などの必要が生じたとき，カーネルの再構築をしなければならない．ローダブルカーネルモジュールなどを利用して，システムを動的に更新できる機能が必要である．

現在の実装では，既知の有害プログラムが利用したことのあるシステムコールと引数のみに監視対象を絞った．現在の実装では監視対象となるシステムコールと引数のパターンが少ない．したがって，オーバヘッドを増加させず監視範囲を広げられるようなシステムの拡張が必要である．

第 7 章

おわりに

本研究では，カーネル感染型有害プログラムを検出するシステムを設計し，実装した．本研究で作成したシステムでは，カーネル感染型有害プログラムによって汚染されたシステム上で見ることが可能な特徴的な動作を取得することができる．この特徴的な動作とは以下のとおりである．

- システムコール実行時の総実行命令数が増加する．
- システムコールの実行に要する時間が増加する．

本研究で作成したシステムでは，この特徴的な動作に着目をして，カーネル感染型有害プログラム検出を行う．

本システムの利点を以下にあげる．

- リアルタイムにカーネル感染型有害プログラムを検出が可能．
- システムコールに関連する部位を包括的に監視が可能．
- 特別な装置を用意することなく，ハードウェアによる粒度が細かい正確な値のカウントが可能．
- システムの汚染場所特定のために有益な情報を提供することが可能．

本研究で作成したシステムの有効性を示すために，公開されているカーネル感染型有害プログラムを用いて実験を行った．そして，これらのカーネル感染型有害

プログラムに汚染されたシステム上から，有害プログラムの特徴的な動作を取得できることを示した．

本研究で作成したシステムは，システムコールに関連する部位に対する攻撃には有効に機能し，侵入を検知することができる．しかし，システムコールに関連する部位に異常なふるまいが発生しないと，侵入を検知することができない．このように，本研究で作成したシステムだけではすべての攻撃の可能性に対処することができない．

攻撃者に不正アクセスを許してしまうことがそもそもの問題ではあるが，早急に侵入の事実に気付くことは非常に大切なことである．防御側が攻撃を封じるとすぐに，攻撃側は新たな攻撃手法を探し始める．ひとつのセキュリティ技術だけでなく，様々なセキュリティ技術を組み合わせることで，システムを包括的に守ることが重要である．本研究では，オペレーティング・システムのカーネルを包括的に監視するためのひとつの手法として，カーネル感染型有害プログラム検出システムの設計および実装を行った．

謝辞

本研究を遂行するにあたり，たくさんの方々にお世話になりました．

はじめに，指導教員の多田好克先生には日頃から熱心なご指導，そしてご鞭撻を賜わりました。佐藤喬助手と安田絹子助手には、研究方針や研究内容に関して多くの御指導をいただきました．多田好克先生，佐藤喬助手には，ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり，大変貴重なご助言をいただきました．博士後期課程の福田伸彦さんには，研究内容やシステムの実装に関して多くの御指導をいただきました．実装に行き詰まったときにはいつも相談に乗ってくださいました．博士後期課程の滝田裕さんには，研究方針や実装に関してのご助言をいただきました．ここに厚く御礼申し上げます．

そして、本研究を遂行できたことは，研究方針や研究内容に関して議論をし，共に研究生活をおくってきた多田研究室，Vytas 研究室の学生諸氏おかげでもあります。さいごにこれらのみなさんに深く感謝いたします。

参考文献

[1] Wenke Lee，Salvatore J．Stolfo and Kui W．Mok: “A Data Mining Framework for Building Intrusion Detection Models,” 1999 IEEE Symposium on Security and Privacy，pp.120–132，May 1999．

[2] 独立行政法人 情報処理推進機構: “2004 年不正アクセス届出状況”, http://www.ipa.go.jp/security/txt/2005/documents/2004all-cra.pdf，Junuary 2005.

[3] 警察庁: “平成 16 年上半期の不正アクセス行為の発生状況等について”, http://www.npa.go.jp/cyber/statics/h16/html17.htm，August 2004．

[4] 土居範久，佐々木良一，内田勝也，岡本栄司，菊池浩明，寺田真敏，村山優子: “情報セキュリティ事典”, 共立出版，ISBN4-320-12070-1，July 2003．

[5] 内田勝也，高橋正和: “有害プログラム –その分類・メカニズム・対策– サイバーセキュリティ・シリーズ 2”, 共立出版，ISBN4-320-12109-0，July 2004．

[6] Julian B. Grizzard, John G. Levine and Henry L. Owen: “Re-establishing Trust in Compromised Systems: Recovering from Rootkits That Trojan the System Call Table,” ESORICS，pp.369–384，September 2004．

[7] Jesus Molina and William Arbaugh: “Using Independent Auditors as Intrusion Detection Systems,” Proceedings of the Fourth International Conference on Informationand Communications Security，pp.291–302，December 2002.

[8] Jelena Mirkovic，Janice Martin and Peter Reiher: “A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms,” Computer Science Department University of California，Los Angeles Technical report ♯020018，July 2002．

[9] Raul Siles Pelaez: “Linux kernel rootkits protecting the system’s “Ring-Zero”,” http://www.sans.org/rr/whitepapers/honors/1500.php , May 2004 .

[10] packet storm: http://packetstormsecurity.org/

[11] Adore-ng: http://stealth.openwall.net/rootkits/adore-ng-0.45.tgz

[12] sd and devik: “Linux on-the-fly kernel patching without LKM,” Phrack Magazine , Volume 11, Issue 58 , File 7 , 2001 .

[13] Jan K. Rutkowski: “Execution path analysis: finding kernel based rootkits,” Phrack Magazine , Volume 11, Issue 59 , File 10 , 2002 .

[14] Jan K. Rutkowski: “Advanced Windows 2000 Rootkit Detection (Execution path analysis) ,” Black Hat Briefings USA, July 2003 .

[15] Ludovic Me and Cedric Michel: “Intrusion Detection: A Bibliography,” Technical report SSIR-2001-01, Supelec, September 2001 .

[16] Stefan Axelsson: “Research in Intrusion Detection Systems: A Survey,” Technical Report No 98-17, Dept. of Computer Engineering, Chalmers University of Technology, August 1999 .

[17] Suresh N. Chari and Pau-Chen Cheng: “BlueBox: A Policy–driven , Host–Based Intrusion Detection system,” Network and Distributed System Security Symposium Conference Proceedings , February 2002.

[18] Adam G . Pennington , John D . Strunk , John Linwood Griffin , Craig A . N . Soules and Garth R . Goodson: “Storage-based Intrusion Detection: Watching storage activity for suspicious behavior,” Proceedings of the 12th USENIX Security Symposium , pp.137–151 , August 2003 .

[19] tripwire: http://www.tripwire.org/

[20] chkrootkit: http://www.chkrootkit.org/

[21] KSTAT: http://www.s0ftpj.org/

[22] kern_check: http://la-samhna.de/library/kern_check.c

[23] alamo: http://www.rackspace.com/alamo/

[24] kad: “Handling Interrupt Descriptor Table for fun and profit,” Phrack Magazine，Volume 11, Issue 59，File 4，2002．

[25] Marc Khouzam and Thomas Kunz: “Single Stepping in Event-Visualization Tools,” Proceedings of the 1996 conference of the Centre for Advanced Studies, November 1996.

[26] Intel: “IA-32 インテル アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル 下巻：システム・プログラミング・ガイド”, Order Number 245472J, 2002．

[27] 吉岡弘隆: “Intel 系 (IA-32) プロセッサのパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツール”, 第 44 回プログラミングシンポジウム，pp.99–110，2003．

[28] 吉岡弘隆: “OLTP 性能向上を目的としたメモリプロファイリングツール”, 第 14 回データ工学ワークショップ，電子情報通信学会，2003．

[29] 吉岡弘隆: “平成 14 年度未踏ソフトウェア創造事業 OLTP 性能向上を目的としたメモリプロファイリングツール成果報告書”, 独立行政法人 情報処理推進機構，2003．

[30] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl and George Chrysos: "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors," Proceedings of Micro-30, December 1997 .

[31] Brinkley Sprunt: "Pentium 4 Performance Monitoring Features," IEEE Micro Volume 22, Issue 4 , July 2002 .

[32] Fred Cohen: "Computer Viruses : Theory and Experiments ," In 7th DOD/NBS Computer Security Conference Proceedings , September 1984 .

[33] Yin Zhang and Vern Paxson: "Detecting Backdoors," Proceedings of the 9th USENIX Security Symposium , August 2000 .