社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－OS－108 (8)
2008／4／23

カーネルレベルルートキットの検知システムの構築

小倉寛之[†1,*1] 大山恵弘[†1] 岩崎英哉[†1]

近年，クラッカーがコンピュータシステムへ侵入した後に，ルートキットと呼ばれるツールを使用することがある．ルートキットは，典型的にはコンピュータシステムへ侵入した痕跡や自身の存在を，システム管理者から隠蔽する機能を持つツールセットである．特に，カーネルレベルルートキットと呼ばれるルートキットは，OS カーネルから返される情報を改竄できるため，検知は単純ではない．本研究では，我々はカーネルレベルルートキットの検知システム Sunny を Linux 上に実装し，実験により有効性を評価した．Sunny はカーネルモジュールとして実装されており，本来不変であるはずのカーネルメモリ領域を監視し，その領域の変化をルートキットの存在を示すものとして検知する．具体的には，カーネルメモリ領域を固定長のメモリブロックに分割し，各ブロックの正しい内容のハッシュ値を特別な領域に保存しておく．そして，システムコールが発行される度に一つのブロックに対して，更新されたかどうかをハッシュ値によって検査する．頻繁にシステムコールが呼び出される場合には，ルートキットが入ってから短い時間でルートキットを検知することが可能である．

Development of Kernel-Level Rootkit Detection System

HIROYUKI OGURA,[†1,*1] YOSHIHIRO OYAMA[†1] and HIDEYA IWASAKI[†1]

Recently, crackers use rootkits after they intrude into a computer system. Rootkits are toolsets that typically have a mechanism for hiding the signs of intrusions and themselves from system administrators. In particular, detecting kernel-level rootkits is not easy because they can modify the information returned by the underlying operating system kernel. In this work, we developed a kernel-level rootkit detection system *Sunny* on Linux, and conducted experiments to evaluate its effectiveness. Sunny is implemented as a loadable kernel module, which watches the kernel-level memory areas that should not be modified in normal execution, and detects the modification of the areas as a sign of a rootkit. Specifically, it partitions the areas into fixed-length memory blocks, and stores the hash values of the correct content for each block in a special memory area. Then, it checks the modification to each block using the hash values each time a system call is invoked. If system calls are invoked frequently, it detects rootkits in a short time after the intrusion.

1. はじめに

コンピュータシステムへ侵入したクラッカーが，侵入の痕跡や自身の存在を隠蔽するために，ルートキットを利用するケースがある．ルートキットとは，不正プログラムが詰め合わせられたツールセットのことである．クラッカーに侵入されたシステムは，システム管理者がその痕跡を発見し，侵入に対する対処を行うまでの間，クラッカーの制御下におかれてしまう．そのため，システム管理者は侵入の痕跡をいち早く見つける必要がある．しかし，ルートキットの利用により，システム管理者は侵入の痕跡を発見し難くなり，侵入の発生からその事実を確認するまでに要する時間が長引いてしまうことが問題となっている．

また，ワームやウイルスなどの各種マルウェアには，ルートキットの隠蔽技術を利用し，長期間システムに留まろうとするものがある．そのため，クラッカーによる侵入の検知やマルウェアの検知がいっそう困難になっている．

本論文では，Linux 上に構築したカーネルレベルルートキットの攻撃検知システム Sunny の設計と実装について述べる．Sunny は，カーネルが提供する Loadable Kernel Module（LKM）の機能を利用し，カーネルに改変を加えることなく，カーネルレベルルートキットの攻撃検知機能をカーネルへと組み込む．

本システムが検知対象とするカーネルレベルルートキットは，カーネルの初期化後には本来不変であるはずのカーネルメモリ領域を改竄するものである．多くのカーネルレベルルートキットは，カーネルメモリの本来不変であるべき部位を改竄し，カーネルの処理をフックして様々な情報の隠蔽を企む．そこで，本シス

†1 電気通信大学情報工学科
Department of Information Science, The University of Electro-Communications
*1 現在，日本ヒューレット・パッカード株式会社
Presently with Hewlett-Packard Japan, Ltd.

テムではそれらのカーネルメモリ領域を監視することにより，カーネルレベルルートキットの攻撃を検知する．本システムをカーネルに組み込む時点でのカーネルメモリの状態を正常な状態であると仮定し，本来不変であるべきカーネルメモリの状態を，本システムの組込み時の状態から変化がないか監視する．

可能な限り早期に攻撃を検知するため，システムコールが呼び出される度に，監視対象のカーネルメモリ領域を検査する．これは，システムコールの処理を実行するシステムコールハンドラをフックすることで実現する[1)]．また，検査によるオーバヘッドを考慮し，1 回の検査で監視対象のカーネルメモリ領域全体の検査は行わず，その領域を固定長に分割したメモリブロックを，システムコールの度に一つずつ順に検査していく方法をとる．

本来不変であるはずのカーネルメモリ領域の一部分のみを検査するようなシステムはこれまでに提案されてきたが，本システムのようにその領域全体を検査するシステムは，我々の知る限り存在しない．

2. ルートキット

2.1 概　　要

ルートキットとは，クラッカーが攻撃目標のコンピュータシステムを制御するためのツールセットのことを指す．例えば，一度侵入したシステムへ再度侵入するためのバックドアや，ネットワークを流れるデータを傍受するスニファ，ユーザのキー入力を記録して送信するキーロガー，侵入の痕跡を隠蔽するトロイの木馬，ログを消去するログクリーナなどがツールセットとしてまとめられていた．しかし，ルートキットの一機能にすぎなかった侵入の痕跡を隠蔽する機能が，ルートキットの特徴として大きな意味を持つようになり，近頃ではルートキットは侵入者の痕跡を隠蔽するためのツールを指すことが多くなっている．

ルートキットは，その動作レベルに応じてユーザレベルルートキットと，カーネルレベルルートキットに分類することができる．以下，それぞれのルートキットについて説明する．

2.2 ユーザレベルルートキット

ユーザレベルルートキットとは，ユーザレベルで動作するルートキットであり，バイナリの置換やログファイルの改竄により，侵入者の痕跡を隠蔽しようとする．例えば，クラッカーが動作させた悪意のあるプロセスを隠蔽する場合，ps コマンドのプログラムファイルをルートキットが用意した偽物のプログラムファイルに置き換える．偽物の ps コマンドは，本物の ps コマンドが出力するプロセスの情報から，悪意のあるプロセスの情報を除いたものを出力することで，悪意のあるプロセスの存在を隠蔽する．

ユーザレベルルートキットに有効な対策方法として，

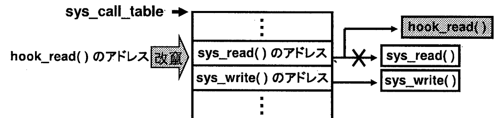

図 1　システムコールテーブルの改竄による sys_read() のフック
Fig. 1　Interception of sys_read() by rewriting the system call table.

システムの整合性検査がある．これは，正常時のシステムと現在のシステムの状態を比較し，差分を見出すことにより，システムの異常を検知する方法である．システム正常時のバイナリとの比較や，ログファイルの監視により，ユーザレベルルートキットの攻撃を検知することが可能である．

2.3 カーネルレベルルートキット

カーネルレベルルートキットとは，カーネルレベルで動作するルートキットである．これは，カーネルメモリを改竄することにより，本来行われるべきカーネルの処理をフックする．そして，そのフック先で情報操作を行い，カーネルの出力を改竄する．よって，カーネルレベルルートキットの攻撃を受けたカーネルが出力する情報は，信頼することができず，ユーザレベルルートキットよりも検知が困難である．著名なカーネルレベルルートキットとして，adore[2)] や adore-ng[3)]，heroin[4)]，SucKIT[5)6)] などがある．

2.3.1 攻 撃 手 法

カーネルレベルルートキットがカーネルメモリのどの部分を改竄し，どの処理をフックして情報操作を行うのかについて説明する．大きく分けて，以下の 5 通りの手法が存在する．

(1)　システムコールテーブルの改竄

システムコールテーブルに格納されているシステムコールサービスルーチンのアドレスを，ルートキットが用意した関数のアドレスに改竄する．これにより，システムコールサービスルーチンの処理がルートキットの用意した関数にフックされ，その関数内で情報操作が可能となる．例えば，**図 1** のように `read` システムコールの処理を行う `sys_read()` をフックすることで，隠蔽したい情報の読み込みを防ぐことができる．

(2)　システムコールテーブルのアドレスの改竄

システムコールハンドラ内で利用されるシステムコールテーブルを，ルートキットが用意した偽のシステムコールテーブルに差し替える．具体的には，**図 2** のように，システムコールハンドラ内で呼び出される `call *sys_call_table(,%eax,4)` という命令を，`call *fake_sys_call_table(,%eax,4)` のように改竄する．これにより，偽のシステムコールテーブルが利用されるようになり，ルートキットは任意のシステムコールサービスルーチンの処理のフックが可能となる．

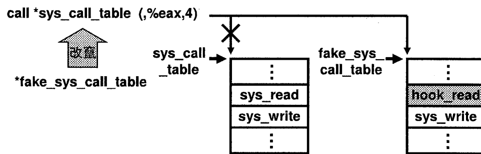

図 2　システムコールテーブルのアドレスの改竄による sys_read() のフック

Fig. 2　Interception of sys_read() by rewriting the address of the system call table.

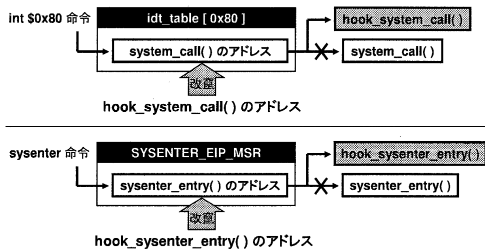

図 3　IDT の 0x80 番目のエントリと SYSENTER_EIP_MSR の改竄によるシステムコールハンドラのフック

Fig. 3　Interception of the system call handler by rewriting idt_table[0x80] and SYSENTER_EIP_MSR.

(3)　関数ポインタの改竄

カーネル内の関数ポインタの値を，ルートキットが用意した関数のアドレスに改竄することにより，本来呼ばれるべき関数の処理をフックする．

(4)　システムコール発行命令が利用するメモリ領域やレジスタの改竄

Linux 2.6 では，int $0x80 命令[*1] と sysenter 命令[*2] によるシステムコールの発行がある．int $0x80 命令による発行の場合，割り込みディスクリプタテーブル (Interrupt Descriptor Table，以下 IDT）の 0x80 番目のエントリに格納された system_call() というシステムコールハンドラへ，sysenter 命令による発行の場合，SYSENTER_EIP_MSR というレジスタに格納された sysenter_entry() というシステムコールハンドラへと処理が移行する．ルートキットは，**図 3** のように IDT の 0x80 番目のエントリや SYSENTER_EIP_MSR を改竄することで，システムコールハンドラの処理をフックする．

(5)　カーネル関数のコードの改竄

図 4 のように，カーネル関数のコードにルートキットが用意した関数へのジャンプコードを上書きすることで，カーネル関数の処理をフックする．この手法は，インライン関数フッキングと呼ばれる．

⋆1 Linux 2.6 以前のカーネルでは，システムコールを発行する唯一の方法．

⋆2 Intel Pentium II プロセッサで登場した命令．Linux ではカーネル 2.6 から使用可能となった．

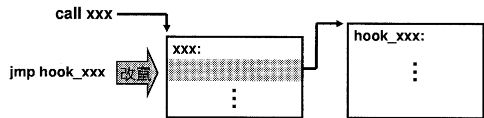

図 4　インライン関数フッキング

Fig. 4　Inline function hooking.

2.3.2　カーネル空間へのアクセス方法

ルートキットがカーネルメモリを改竄するには，カーネル空間へアクセスする必要がある．ここでは，ルートキットがカーネル空間へアクセスする方法を二つ述べる．

(1)　LKM を利用する方法

LKM はカーネル権限で動作し，LKM をカーネルにロードする際，任意の関数を呼び出すことができる．LKM として実装されたルートキットは，この機能を利用し，自身のロード時にカーネルメモリの改竄を行う関数を呼び出す．

(2)　/dev/kmem ファイルを利用する方法[6)]

/dev/kmem は，ユーザ空間からカーネル空間へアクセスするためのキャラクタデバイスファイルである．ルートキットは，open() により /dev/kmem ファイルを開き，lseek() により改竄するカーネルメモリのアドレスを設定した後，write() により改竄を行う．

2.3.3　対　　策

カーネルレベルルートキットの対策として，LKM を利用禁止にすること，/dev/kmem ファイルへの書き込みを禁止することが効果的である．しかし，どちらの機能も便利なものであり，使用を禁止したり制限したりすることは，その恩恵を逃すことになる．そのため，別の方法による対策ツールが開発されてきた．詳しくは 5 章で述べる．

3.　検知システム Sunny

3.1　設 計 方 針

カーネルレベルルートキットの攻撃の特徴的動作である，カーネルメモリの改竄を検知するシステム Sunny について説明する．カーネルレベルルートキットの多くは，カーネルメモリの本来不変であるはずの領域を改竄する．そこで，Sunny は LKM を利用し，カーネルに改変を加えることなく，カーネル内部から本来不変なはずのカーネルメモリを監視する．監視対象のカーネルメモリの状態が，システム正常時の状態と異なっていれば，カーネルレベルルートキットによる攻撃があったとみなす．

本システムは，2.3.1 節で述べた (1)〜(5) のすべてのタイプのカーネルレベルルートキットを検知対象とする．また，本システム導入時にはカーネルの状態は正常であると仮定する．

カーネルメモリの検査は，ハッシュ関数を利用して

行う．本システムのロード時の，正常なカーネルメモリから得られたハッシュ値と，検査時のカーネルメモリから得たハッシュ値を比較し，値が異なっていれば改竄がなされたと判断する．本システムは，検査時にカーネルメモリの内容を読んでハッシュ値を計算する．

監視対象となるカーネルメモリ領域のサイズは非常に大きいため，1 回でその全体を検査するには，多大なオーバヘッドが生じる．例えば 4.2 節の実験で用いた環境では，そのサイズは 2,738,400 バイトであった．そこで，監視するカーネルメモリを固定長に分割し，複数回の検査によって，監視対象のカーネルメモリ全体を検査する方法をとる．また，可能な限り短期間で攻撃を検知するため，頻繁に呼び出される処理であるシステムコールのハンドラをフックし，そのフック先でカーネルメモリを検査する．つまり，システムコールの度に分割したカーネルメモリを一つずつ順に検査していく．

また，本システム自身をルートキットの攻撃から保護するため，本システムがフック処理に用いているメモリ領域とレジスタをカーネルスレッドに監視させる．

3.2 監視のための準備

本システムのロード時に行う準備作業には，以下がある．

- IDT の 0x80 番目のエントリの書き換え
- SYSENTER_EIP_MSR の書き換え
- 監視するカーネルメモリのハッシュ値の取得

まず，システムコールの発行の度にカーネルメモリを検査するために必要となる，システムコールハンドラのフックのための作業を行う．本システムは，**図 5** に示すように，int $0x80 命令が利用する IDT の 0x80 番目のエントリと，sysenter 命令が利用する SYSENTER_EIP_MSR の一部に変更を加える．IDT の 0x80 番目のエントリの上位 16 ビットと下位 16 ビットには，int $0x80 命令が実行された後に処理が移る関数のアドレスの上位 16 ビットと，下位 16 ビットが格納されている．また，SYSENTER_EIP_MSR の下位 32 ビットには，sysenter 命令が実行された後に処理が移る関数のアドレスが格納されている．本システムは，この部分を本システム内の関数のアドレスに書き換えることにより，それぞれの命令に対応するシステムコールハンドラへの処理の移行を，本システム内の関数へとフックする．

次に，本システムのロード時におけるカーネルメモリの内容のハッシュ値を取得する．内容自体ではなくハッシュ値を比較することにより，検査に伴う空間的オーバヘッドを軽減する．監視対象のカーネルメモリを固定長に分割し，分割してできた個々のメモリブロックの内容をハッシュ関数にかける．そうして得られたハッシュ値を，本システムが配列として保持し，カーネルメモリの検査に用いる．

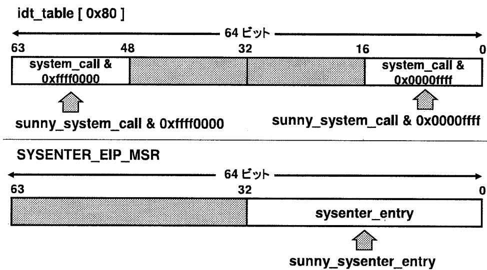

図 5 IDT の 0x80 番目のエントリ と SYSENTER_EIP_MSR のオフセット値の書き換え

Fig. 5 Rewriting of idt_table[0x80] and SYSENTER_EIP_MSR.

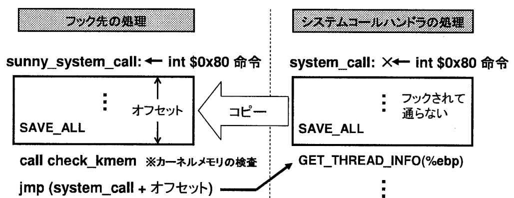

図 6 system_call() のフック先の処理

Fig. 6 Processing after system_call() is hooked by Sunny.

3.3 システムコールの発行時の処理

int $0x80 命令によりシステムコールが発行された場合，本システムは **図 6** のような処理を行う．int $0x80 命令が実行されると，`sunny_system_call()` という本システム内の関数へと処理が移り，この関数内でカーネルメモリを検査する．`sunny_system_call()` はまず，本来のシステムコールハンドラ関数である `system_call()` 内の `SAVE_ALL`[*1] マクロまでと同じ一連の処理を実行する．`SAVE_ALL` マクロまでの処理を行えば，スタックへ退避したレジスタの値をカーネルメモリの検査で自由に使用することができる．`SAVE_ALL` マクロの直後で，カーネルメモリの検査関数を呼び出し，検査が終了すれば，`system_call()` 内の `SAVE_ALL` マクロの直後の処理へ，jmp 命令で移行する．つまり，通常のシステムコールハンドラの処理に，カーネルメモリの検査を挟み込む形になる．

同様に，sysenter 命令によるシステムコールの発行の場合にも，本システムは本来のシステムコールハンドラ関数である `sysenter_entry()` の `SAVE_ALL` マクロまでの一連の処理を行った後，カーネルメモリを検査する．その後，`sysenter_entry()` 内の `SAVE_ALL` マクロ直後の命令へと処理を移す．

*1 システムコールハンドラ内で使用する可能性があり，割り込みにより自動的に退避されないレジスタ群をスタックに退避させるマクロ．

3.4 カーネルメモリの監視

ここでは，Sunny によるカーネルメモリの監視方法について詳しく説明する．

3.4.1 監視対象領域の設定

カーネルメモリには，コード領域とデータ領域がある．データ領域は更に，初期化されたデータ領域と初期化されないデータ領域とに分けられる．これらの領域は，以下に示す 4 つのシンボルにより区切られている．

_text カーネルコードの先頭アドレス．
_etext カーネルコードの末尾アドレス．
_edata 初期化されたカーネルデータの末尾アドレス．
_end 初期化されないカーネルデータの末尾アドレス．

なお，_etext の直後から始まる初期化されたカーネルデータ領域には，まず読み込み専用であるデータが並んでおり，その後に読み込み専用でないデータが並んでいる．つまり，この領域は更に細かく読み込み専用のデータ領域と，読み込み専用でないデータ領域に分けられる．

本システムは，カーネルのコード領域全体と初期化されたデータ領域の一部を監視する．コード領域には，カーネル関数などの実行コードが存在しており，通常この領域が変更されることはない．よって，コード領域に何らかの変化が生じれば，ルートキットによる攻撃を受けた可能性が高いと判断する．一方，データ領域には読み込み専用のシステムコールテーブルや，読み込み専用と定義されていないものの本来変更されるべきでない IDT などのデータと，プロセスリストなどの変更があり得るデータとが混在している．従って，データ領域に関しては初期化されたデータのうち，初期化後に変更されるべきでないデータについてのみ監視する．以上をまとめると，具体的な監視領域は以下の通りである．

- コード領域全体
- データ領域の読み込み専用領域全体
  - **sys_call_table** など
- データ領域の読み込み専用領域以外の一部
  - **proc_root_inode_operations**
  - **proc_root_operations**
  - **ext3_dir_operations**
  - **idt_table**

本システムはロード時に，vmlinux★1 から得られるシンボル情報を利用し，監視領域のアドレスを取得している．System.map★2 ファイルからもこのシンボル情報を得ることは可能であるが，このファイルがルートキットにより改竄されると，OS の再起動時に再び本システムをインストールする際に改竄の影響を受けてしまう．よって，本システムは vmlinux を利用する．

★1 カーネルのイメージファイル．
★2 カーネル内の変数・関数のシンボルとその型，アドレスが記載されたファイル．

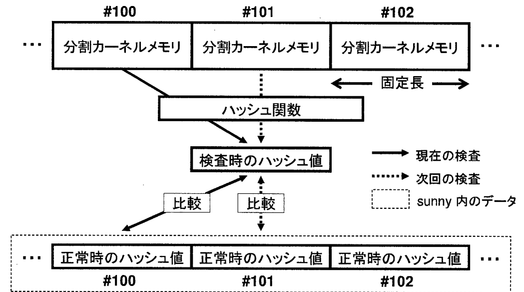

図 7 カーネルメモリの検査の様子
Fig. 7 Checking kernel memory.

3.4.2 メモリ領域の内容の検査

本システムはカーネルメモリを 256 バイトのメモリブロックに分割し，システムコールの発行の度に 1 ブロックの検査を行う．よって，分割個数回のシステムコールが呼ばれたとき，監視領域全体を 1 周検査したことになる．なお，この 256 バイトという分割サイズは実験により決定した．

本システムがカーネルメモリを検査する様子を，**図 7** に示す．システムコールの発行の度に，カウンタが示す番号の分割カーネルメモリをハッシュ関数にかけ得られたハッシュ値と，ロード時にあらかじめ保持しておいた正常な分割カーネルメモリのハッシュ値を比較する．両ハッシュ値が一致すれば，この分割カーネルメモリは正常であるが，一致しなければ改竄されている．改竄されていた場合には，カーネル・ログ・バッファに警告を書き込む．システム管理者は，このログを確認することにより，カーネルレベルルートキットに攻撃されたことを知る．

そして，次のシステムコールが呼ばれると，カウンタが次の番号に移り，その番号が示す分割カーネルメモリを検査する．全ての分割カーネルメモリを検査し終えれば，カウンタを 0 に戻し，再び最初の分割カーネルメモリから検査をやり直す．

3.5 本システム自身の保護

本システムは，システムコールハンドラをフックし，そのフック先でカーネルメモリを検査する．しかし，ルートキットの攻撃により，IDT の 0x80 番目のエントリや SYSENTER_EIP_MSR が改竄され，本システムがシステムコールハンドラをフックして行う処理を更にフックされる可能性がある．そうすると，本システムはカーネルメモリの検査ができず，無力化されてしまう．

そこで，本システムはカーネルスレッドを利用し，この二つの改竄されては困る情報を監視する．本システムのロード時に，本システム保護用のカーネルスレッドを生成する．このカーネルスレッドは，上の情報にロード時の状態から変化がないか検査する．**図 8**

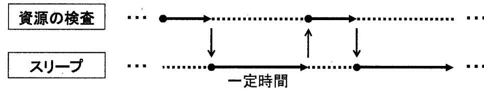

図 8　本システムを保護するカーネルスレッドの挙動
Fig. 8　Kernel thread for protecting Sunny.

に示すように，一定の周期（現在は 10 sec）で検査とスリープを繰り返し，それらの情報が改竄された場合には，ロード時の状態に復元し，本システムの無力化を防ぐ．

このカーネルスレッドをデーモン化させ，あらゆるシグナルを拒否させることにより，クラッカーに SIGKILL や SIGQUIT などのシグナルを送られ，強制終了させられることを防いでいる．本システムのカーネルモジュールは，本システム保護用のカーネルスレッドと監視フラグを共有しており，本システムのモジュールのアンロード時に監視フラグを 0 にすることで，このカーネルスレッドを終了させる．

3.6 議　　論

カーネルメモリを監視するだけであれば，本システムをカーネルスレッドとして実装することも可能である．そのカーネルスレッドは，一定周期でカーネルメモリの検査とスリープを繰り返し，分割したカーネルメモリを順に検査する．カーネルスレッドとして実装した場合，本システムのようにシステムコールハンドラをフックする必要がない．そのため，本システムに比べて実装がし易く，3.5 節で述べたシステム保護用のカーネルスレッドを生成する必要もなくなる．加えて，実際に本システムの攻撃検知機能をカーネルスレッドで実装し，4.2 節の (1) の実験を行うと，本システムよりもオーバヘッドが小さくなることが確認できた．よって，カーネルメモリの監視のみに限れば，カーネルスレッドで実装する方が適しているかもしれない．

本システムがシステムコールハンドラをフックする方法をとった理由は，拡張性である．システムコールハンドラをフックすることで，システムコール番号などの有益な情報を得ることが可能となり，カーネルメモリの改竄検知以外の様々なセキュリティ処理を組み合わせることができる．

ここで，本システムの拡張の一例を挙げる．本システムは，ルートキットの攻撃を検知すると，カーネル・ログ・バッファに警告を書き込む．しかし，本システムを詳しく知るクラッカーが，カーネル・ログ・バッファの内容を出力する際に利用する `write` システムコールの処理をフックし，その出力を改竄することで，システム管理者に警告を知られるのを防ごうと企むかもしれない．そういった場合にも，システムコールハンドラをフックしておくことで対処が可能となる．システムコールハンドラのフック先では，発行要求のかかっているシステムコール番号を知ることができる．よって，`write` システムコールの発行要求の際には，改竄されている恐れのあるカーネルの代わりに，本システムが `write` システムコールの処理を代替することで対処できる．`write` システムコール以外にも，カーネルレベルルートキットが隠蔽工作を行うにあたってよく狙う，`read` や `getdents64` などのシステムコールについても，本システムにその処理を代替させることでシステムコールの安全性を高めることができる．このように，本システムを更に拡張するにあたり，システムコールハンドラのフックは重要といえる．

表 1　攻撃検知テストに用いたカーネルレベルルートキット
Table 1　Kernel-level rootkits used in experiments.

ルートキット	改竄箇所	アクセス方法
adore	SCT	LKM
adore-ng	FP	LKM
heroin	SCT	LKM
SucKIT	ASCT	/dev/kmem

SCT：システムコールテーブル　FP：関数ポインタ
ASCT：システムコールテーブルのアドレス

4. 実験と評価

4.1 攻撃検知実験

Sunny の有効性を評価するため，表 1 に示す著名なカーネルレベルルートキットを用いた攻撃検知実験を行った．実験に用いたルートキットは，Linux 2.4 の環境でのみ動作するものが多かったため，Sunny を Linux 2.4 の環境で動作するよう作成し実験した．ただし，heroin は Linux 2.4 では動作しないので，修正を加えたものに対して実験を行った．また，SucKIT は `/dev/kmem` ファイルから IDT の 0x80 番目のエントリを経由してシステムコールハンドラの `system_call()` にアクセスし，システムコールテーブルのアドレスなどの情報を得ることでカーネルメモリの改竄を行うアドレスを設定するのだが，本システムの IDT の 0x80 番目のエントリの書き換えにより，SucKIT は本システム内の `sunny_system_call()` にアクセスするようになり，`system_call()` で得るはずだった情報を取得できなくなってしまうので，必要な情報を与える修正を SucKIT に加えた．実験環境は以下の通りである．

OS　Vine Linux 2.5 (kernel 2.4.18-0vl3)
CPU　Intel Pentium 4 (2.40 GHz)
メモリ　513.5 MB

カーネルメモリの監視領域は，コード領域全体，読み込み専用のデータ領域全体，読み込み専用でないデータ領域の一部（`proc_root_inode_operations`，`proc_root_operations`，`ext3_dir_operations`，`idt_table`，`sys_call_table`[*1]）とした．

*1 Linux 2.6 では読み込み専用のデータ領域に存在するが，Linux 2.4 では読み込み専用でないデータ領域に存在する．

表 2 攻撃検知テストの結果
Table 2 Results of the experiments.

ルートキット	検知結果	検知箇所
adore	○	sys_call_table
adore-ng	○	proc_root_inode_operations, proc_root_operations, ext3_dir_operations
heroin⋆1	○	sys_call_table
SucKIT⋆1	○	system_call, tracesys

表 3 カーネルの再構築による実験でのオーバヘッド
Table 3 Overhead in building the kernel.

Sunny	分割サイズ [byte]	実行時間 [sec]	オーバヘッド [%]
未導入	—	580	—
導入	64	607	4.7
	128	615	6.0
	256	626	7.9
	512	649	12
	1024	709	22

表 4 mprotect の呼び出しによる実験でのオーバヘッド
Table 4 Overhead measured in the mprotect test.

Sunny	分割サイズ [byte]	実行時間 [μsec]	オーバヘッド [μsec]
未導入	—	0.217	—
導入	64	4.728	4.51
	128	5.876	5.66
	256	8.095	7.88
	512	12.734	12.5
	1024	22.020	21.8

表 5 ルートキットを検知するのにかかる平均時間.
Table 5 Average time for detection of rootkits.

分割サイズ [byte]	分割個数 [個]	検知までの平均時間 [sec]
64	42,788	2.77
128	21,394	1.40
256	10,697	0.714
512	5,349	0.370
1024	2,675	0.202

実験結果を表 2 に示す．この結果から，本来不変なはずのカーネルメモリ領域の監視により，カーネルレベルルートキットの攻撃を検知できることを示せた．

4.2 オーバヘッド

本システムの導入により生じるオーバヘッドを測定するため，(1) カーネルの再構築による実験と，(2) mprotect システムコールを繰り返し呼び出すプログラムによる実験を行った．これらの実験を行った環境は以下の通りである．

OS Ubuntu 7.04 (kernel 2.6.20.3-ubuntu1)
CPU Intel Pentium 4 (2.66 GHz)
メモリ 503.9 MB

なお，これらの実験において，3.5 節で述べた本システムを保護するカーネルスレッドは，10 秒毎に資源を検査するよう設定した．

カーネルの再構築による実験では，本システムを導入した状態と未導入の状態とで，カーネルの再構築に要する時間を測定し，オーバヘッドを求めた．また，本システムが監視するカーネルメモリの分割サイズを変更しての測定も行った．測定結果を表 3 に示す．表 3 から，カーネルメモリの分割サイズが大きくなるにつれ，オーバヘッドが大きくなっているのが分かる．これは，分割サイズが大きくなると，検査時に利用するハッシュ関数へ渡すデータの量が増すためである．本システムが採用している 256 バイトの分割サイズでは，オーバヘッドは 7.9% であり，十分小さいといえる．

mprotect システムコールの呼び出しによる実験では，本システムを導入した状態と未導入の状態とで，1 回の mprotect システムコールの呼び出しにかかる時間を測定した．測定結果を表 4 に示す．表 4 から，それぞれの分割サイズにおいて，システムコールが 1 回呼ばれる度にどの程度のオーバヘッドが加算されるのか，おおよその値を計算することができる．例えば，256 バイトの分割サイズで 100 万回のシステムコール呼び出しが行われると，約 8 秒程度のオーバヘッドが生じる．

4.3 検知までに要する時間

4.2 節のカーネルの再構築による実験の測定結果を元に，本システムがカーネルレベルルートキットの攻撃を検知するまでに要する時間について検証する．まず，カーネルの再構築による実験で呼び出されたシステムコールの総数を調べると，4,686,426 回であった．これをカーネルの再構築に要した時間で割れば，カーネルの再構築時に呼び出されるシステムコールの頻度が求まる．カーネルメモリの分割個数をこの頻度で割ると，監視対象のカーネルメモリ全体を 1 周検査するのに要する時間が分かり，その時間の半分が，ルートキットの攻撃を検知するまでにかかる平均検知時間である．カーネルメモリの分割サイズごとの結果を，表 5 に示す．

この検証結果から，カーネルの再構築のような，頻繁にシステムコールを呼び出す作業を行う環境では，本システムは数秒以内でルートキットの攻撃を検知することができるといえる．

また，表 3 と表 5 から分かるように，オーバヘッドと検知に要する時間の関係は，トレードオフとなっている．よって，本システムはトレードオフを考慮し，256 バイトの分割サイズを採用した．

⋆1 オリジナルに修正を加えたもの．

5. 関連研究

カーネルレベルルートキットの攻撃を検知する方法として，正常時のカーネル内部の状態と現在のカーネル内部の状態を比較するものがある．例えば，正常時のシステムと現在のシステムとで，システムコールテーブルの内容を比較する方法[7)8)]や，システムコールテーブルのアドレスを比較する方法[7)]，カーネル内の関数ポインタの値を比較する方法[7)]，IDT の状態を比較する方法[9)]がある．しかし，これらの方法は，カーネルメモリの本来不変であるべき領域の一部のみを検査しているに過ぎず，その部位以外への攻撃を検知することはできない．本システムは，本来不変であるべきカーネルメモリ領域を包括的に検査することができる．

他の検知方法として，既知のルートキットを特徴づけるパターン（シグネチャ）とのパターンマッチによる検知方法[10)]や，システムコールの処理の監視による検知方法[11)]がある．シグネチャとのパターンマッチによる検知方法は，既知のルートキットの特徴を元に検査を行うため，新種のものや修正が加えられたものについては検知不可能である．

kern_check[7)]は，System.map ファイルから得られる情報を元に，現在のシステムコールテーブルの状態及び，カーネル内に存在する一部の関数ポインタの値を検査する．システムコールテーブルや関数ポインタの状態を調べるため，`/dev/kmem` を利用してカーネルメモリにアクセスする．kern_check は，System.map を利用して検査を行うため，System.map が改竄されると正確な検査結果を得ることはできない．また，カーネル関数のコード領域を改竄するような攻撃は検知不可能である．

花田らのシステム[11)]は，システムコールの処理に要する時間と命令数を利用した，カーネルレベルルートキット検知システムである．カーネルレベルルートキットにより攻撃されたシステムでは，正常なシステムに比べてシステムコールの処理に要する時間と命令数が増加する．これは，通常のシステムコールの処理に，ルートキットによる隠蔽工作の処理が付加されるからである．花田らのシステムは，あるシステムコールの処理に要する時間と命令数に一定の閾値を定め，この閾値を超えればカーネルレベルルートキットによる攻撃を受けたとみなす．このシステムは，システムコールに関連する部位以外の改竄を検知することは不可能である．

6. まとめと今後の課題

本論文では，カーネルの初期化後には本来不変であるはずのカーネルメモリ領域を監視することにより，カーネルレベルルートキットの攻撃を検知するシステム Sunny の設計と実装について述べた．本システムは，LKM として実装することで，カーネルの再構築なしに攻撃検知機能をカーネルに組み込むことができる．実験による評価の結果，本システムは著名なカーネルレベルルートキットを検知できることを確認したと共に，オーバヘッド，検知までに要する時間共に実用範囲内であるということを確認した．

今後の課題としては，カーネルの初期化後にも変更があり得るカーネルメモリ領域に対する攻撃も，検知できるようにしたい．これらの領域は，正常な変更操作と，ルートキットによる改竄操作との判別が難しいため，今回は監視対象外とした．また，カーネルスレッドの有用性を考慮し，カーネルメモリはカーネルスレッドを用いて監視し，他の機能拡張をシステムコールハンドラのフックにより実現する方法についても検討したい．

参考文献

1) 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).
2) adore: http://packetstormsecurity.org/groups/teso/adore-0.42.tgz.
3) adore-ng: http://packetstormsecurity.org/groups/teso/adore-ng-0.41.tgz.
4) heroin: http://althing.cs.dartmouth.edu/secref/resources/kernel/rootkits/heroin.c.
5) SucKIT: http://packetstormsecurity.org/UNIX/penetration/rootkits/sk-1.3a.tar.gz.
6) devik, sd: Linux on-the-fly kernel patching without LKM, Phrack Magazine, Vol.11, Issue. 58, File.7, (2001), http://www.phrack.com/issues.html?issue=58&id=7.
7) kern_check: http://la-samhna.de/library/kern_check.c.
8) KSTAT: http://www.s0ftpj.org/.
9) kad: Handling Interrupt Descriptor Table for fun and profit, Phrack Magazine, Vol.11, Issue.59, File.4 (2002), http://www.phrack.com/issues.html?issue=59&id=4.
10) chkrootkit: http://www.chkrootkit.org/.
11) 花田智洋：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修士論文 (2005).