Computer Security Symposium 2018
22 - 25 October 2018

独自のカーネル用仮想記憶空間を用いたカーネルモジュール監視手法

葛野 弘樹[1,2]　山内 利宏[1]

概要：計算機上のオペレーティングシステム（OS）を不正利用するための脆弱性を用いた攻撃への対策が重要となっている．多くの攻撃は OS カーネルが管理する仮想記憶空間を操作し，任意のプログラムコードを実行して特権を取得する．Linux における OS カーネルへの攻撃の対策手法として，KASLR による仮想記憶空間へのカーネル関数・データ部配置のランダム化，NX ビット，SMAP 及び SMEP など仮想記憶空間のデータ参照・実行を制御する機構，また，KPTI によるユーザモードとカーネルモードの仮想記憶空間を分離する手法がある．これら対策手法の組合せにより，ユーザモードとカーネルモードの両方を利用する脆弱性の攻撃緩和は可能である．しかし，カーネルモードのみで完結する脆弱性による攻撃の場合，カーネルの仮想記憶空間が不正に操作される可能性がある．本稿では，カーネルが管理する仮想記憶空間を監視するために，新たな仮想記憶空間を用意し，その仮想記憶空間上でカーネルの仮想記憶空間の監視を行う手法を提案する．提案手法による保護機能として，カーネルの仮想記憶空間への不正操作を監視することができ，意図しないカーネルモジュールの挿入などの検出が可能となる．Linux における提案手法の実現方式，評価において不正なカーネルモジュール挿入を検出可能であること，及びベンチマークでの実行性能への影響を示す．

キーワード：仮想記憶空間，カーネル保護，オペレーティングシステム，システムセキュリティ

Hiroki Kuzuno[1,2]　Toshihiro Yamauchi[1]

1. はじめに

オペレーティングシステム（OS）のうちカーネルの脆弱性が多数報告されており [1], [2]，その対策が必要とされている．攻撃の多くは OS 上の特権ユーザを狙っており，カーネルの脆弱性を利用し特権を奪取することを試みる．

攻撃への対策のうち，OS でのアクセス制御として，SELinux[3] ではアクセス制御ポリシによるプロセスのアクセス範囲や特権ユーザの権限の制限，ケイパビリティ [4] ではネットワーク，ファイル操作など OS の持つ機能を細分化し，特権ユーザが奪取されたとしても OS への被害軽減を図ることを可能としている．実際に，カーネルの脆弱性を利用した攻撃では，カーネルの特定の関数を起点として仮想記憶空間を不正に操作する．このため，スタック上の関数戻り番地の監視 [5] による攻撃の検出，KASLR (Kernel Address Space Layout Randomization) の仮想記憶空間におけるカーネル関数，データ部配置のランダム化 [6] による攻撃の困難化が行われている．仮想記憶空間へのアクセス制限として，仮想記憶空間に配置されたデータへの実行を制御するための NX ビット（No eXecute bit），カーネルのスーパバイザモードからユーザモードの仮想記憶空間にあるデータへのアクセスを防ぐ SMAP（Supervisor Mode Access Prevention）[7]，ならびにスーパバイザモードからユーザモードの仮想記憶空間に置かれたデータの実行を防ぐ SMEP (Supervisor Mode Execution Prevention)[7] が提案され，用いられている．また，Meltdown によりユーザモードからカーネルモードの仮想記憶空間へのサイドチャネル攻撃が示され，Linux では KPTI（Kernel Page Table Isolation）にてユーザモードとカーネルモードの間で仮想記憶空間を明確に分離する手法 [8] が提案され，実装が行われた．

これらの対策は，カーネルの脆弱性を利用した攻撃の困

[1] 岡山大学 大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University
[2] セコム株式会社 IS 研究所
Intelligent Systems Laboratory, SECOM Co., Ltd., Japan

難化，攻撃成功後の特権ユーザの制限による被害の最小化を目的としており，ユーザモードとカーネルモードの両方を利用する脆弱性の攻撃緩和は可能である．しかし，カーネルモードのみで完結する脆弱性による攻撃の場合，カーネルの仮想記憶空間は保護していないため，仮想記憶空間が不正に操作される可能性がある．

そこで，本稿では，カーネルに独自の仮想記憶空間を用意し，カーネルの管理する仮想記憶空間の保護をより強固にするセキュリティ機構を提案する．提案手法では，不正なカーネルモジュール（以下，モジュール）やカーネル脆弱性を用いた攻撃によるカーネルの仮想記憶空間の書換えによる影響を回避するため，特定のタイミングにおいて，カーネルの仮想記憶空間の切替えを行い，独自の仮想記憶空間上において監視を行う．これにより，カーネルの仮想記憶空間への不正操作の検出を実現する．具体的な効果としては，カーネルの脆弱性を利用され，特権ユーザが奪取された後，意図しないモジュールが挿入されたことをカーネルの仮想記憶空間上で検出することが可能となる．また，カーネルの脆弱性を利用され，カーネルの仮想記憶空間の書き換えが起こったとしても，独自の仮想記憶空間上のセキュリティ機構は別の空間に存在するため，書き換えの影響を受けない．我々は，提案手法を KPTI に対応した Linux において実現し，実際に不正なモジュールの挿入を検出できることを評価した．また性能評価として，ベンチマークにより性能オーバヘッドを評価した．

本稿における我々の研究による貢献ならびに得られた結果は以下の通りである：

- カーネルの仮想記憶空間に対し，独自の仮想記憶空間から監視するセキュリティ機構を提案した．この方式は，仮想化技術を用いずに，従来のカーネル内に実現するよりもセキュリティ機構の安全性を高めることができる．また，ベアマシンで起動する OS に適用可能であり，仮想計算機上の OS でも利用可能なため，仮想化技術を用いたセキュリティ技術とも共存可能である
- KPTI に対応した最新の Linux への提案手法の実現方式，および事例として不正なモジュール挿入の検出方式を示した．ベンチマークによる性能評価により，システムコール 1 回あたりのオーバヘッドは 0.470 μs から 0.889 μs であることを示した

2. 背景知識

2.1 仮想記憶空間の管理

仮想記憶空間は実行中のプロセスに専用の記憶空間を割当てる技術であり，カーネルの管理する物理的な記憶空間を大幅に超えた領域を提供することを可能としている．Linux x86_64 ではプロセスごとに仮想記憶空間を管理している．仮想記憶空間にある仮想アドレスを物理アドレスに変換するためには多層のページテーブルが使われる（図 1 を参照）．CR3 レジスタにプロセスごとの多層のページテーブルの物理アドレスを格納し，順にページテーブルを参照していく．最終的に，MMU（Memory Management Unit）および TLB（Translation Lookaside Buffer）を利用して物理アドレスへのアクセスを行う．

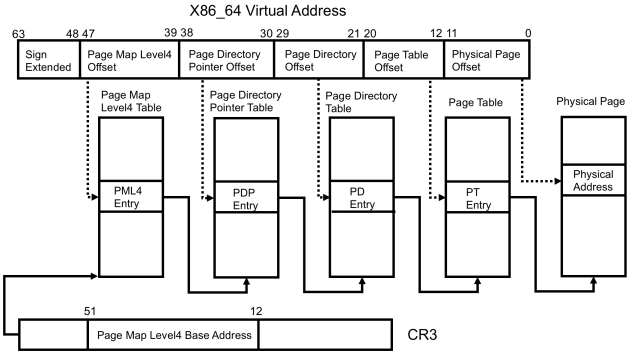

図 1　多層のページテーブルによる仮想アドレスから物理アドレスへの変換

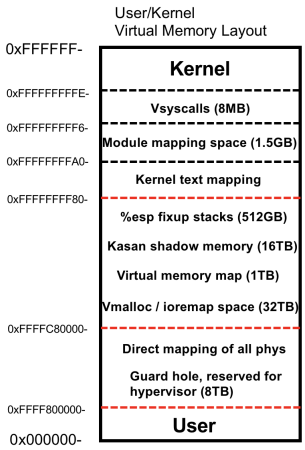

図 2　Linux x86_64 における仮想記憶空間レイアウト

仮想記憶空間に配置されるデータのレイアウトはカーネルおよび CPU アーキテクチャごとに異なる．Linux x86_64 における仮想記憶空間は 48 ビットの仮想アドレスにより 256TBytes となり，ユーザ領域（128TBytes）とカーネル領域（128TBytes）からなる．カーネルの仮想記憶空間上にはカーネル関数，データ部，モジュール，および vmalloc などのメモリ確保のための領域が決められたアドレス範囲（図 2 を参照）で配置されている．

2.2 想定する脅威モデル

本稿での脅威モデルとして，攻撃者はカーネルモードで完結するカーネルの脆弱性を利用して，OS の特権を取得し，不正なカーネルモジュールを挿入することを想定する．カーネルの仮想記憶空間においては，脆弱性の実行時に書換えの対象となる領域，およびモジュールの挿入先の領域以外は改ざんされないものとする．また，BIOS，CPU，MMU，TLB，ならびにその他ハードウェアは安全とする．

3. 提案手法と実現方式

3.1 提案手法の目的と要件

本研究では，提案手法において，意図しないモジュール挿入をカーネルの仮想記憶空間の不正な書換えとして検出することを目的とする．カーネルの仮想記憶空間は，ユーザモードからカーネルモードに遷移した後においてのみ読書き，実行可能である．このため，カーネルの仮想記憶空間の不正な書換えはカーネルモードでのカーネルコードの実行中に発生する．

カーネルモードにおいてカーネルの仮想記憶空間の監視を行う手法の要件を 2 つ挙げる．

要件 1：カーネルコードから参照されない記憶領域の確保

要件 2：カーネルコードから参照されない記憶領域での監視実行

これらの要件を満たすために，独自のカーネル用仮想記憶空間を用意し，本来のカーネルの仮想記憶空間からの記憶領域の隔離，および監視実行を実現するセキュリティ機構を提案する．

3.2 提案手法の構成

独自のカーネル用仮想記憶空間による監視の全体像を**図 3** に示す．要件 1 を満たすため，独自のカーネル用仮想記憶空間（Kernel mode Secret Virtual Memory）を作成する．このカーネル用仮想記憶空間は元の仮想記憶空間からの複製となり，複製元のカーネルコード，データならびに監視用のカーネルコード（Monitoring code）とデータ（Monitoring Valid Data）は同じとなる．カーネルモードにおいて，仮想記憶空間が完全に分離されることで，監視対象となるカーネルコードが実行中に利用する仮想アドレスからは，独自のカーネル用仮想記憶空間にはアクセスすることができない．また，元の仮想記憶空間にある監視用カーネルコードならびにデータはページテーブル上にてアクセス制限のフラグ操作を行い参照不可とする．

要件 2 を満たすため，監視の際には作成したカーネル用仮想記憶空間に用いる多層ページテーブルの物理アドレスを仮想記憶空間を指す特定のレジスタに書込み，仮想記憶空間を切替える．切替え後の仮想記憶空間上で監視用のカーネルコードを動作させ，監視終了後，再度，元の仮想記憶空間に用いる多層ページテーブルの物理アドレスを仮想記憶空間を指す特定のレジスタに書込み，仮想記憶空間を切替えた後，元のカーネルコードの処理を継続させる．

3.3 提案手法による監視

独自のカーネル用仮想記憶空間を用いた監視シーケンスを**図 4** に示す．まず，攻撃側の処理の流れを説明する．

(1) 攻撃者はカーネルモードで完結するカーネル脆弱性を利用して特権を奪取する

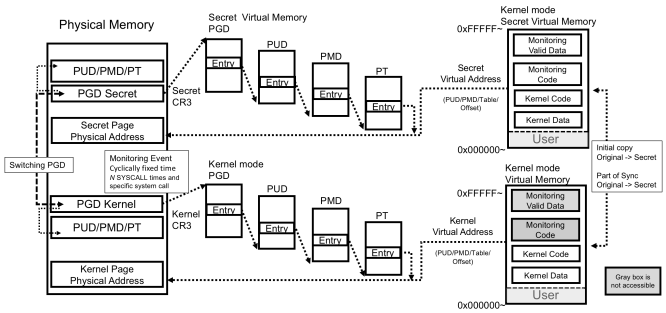

図 3 独自のカーネル用仮想記憶空間を用いた監視の全体像

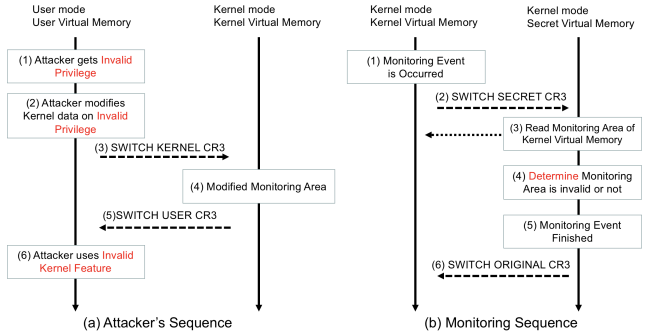

図 4 独自のカーネル用仮想記憶空間を用いる際の各種シーケンス

(2) 攻撃者は insmod コマンドを実行し，不正なモジュールの挿入を行う

(3) insmod コマンドの中で sys_init_module システムコールを発行し，カーネルモードに遷移する

(4) カーネルへのモジュール挿入が行われる

(5) システムコールが終了し，ユーザモードに遷移する

(6) 攻撃者はモジュールの利用が可能となる

続いて，監視側の処理の流れを説明する．

(1) 監視を開始する

(2) 独自のカーネル用仮想記憶空間への切替えを行う

(3) 監視用カーネルコードが，監視対象となる仮想記憶空間からデータ（本稿ではモジュール名リスト）を読込む

(4) あらかじめ決められた正しいデータ（モジュール名リスト）との比較を行う

(5) 比較の結果より，不正か否かの検出を行う

(6) 本来のカーネルの仮想記憶空間への切替えを行う

以上の処理により，独自のカーネル用仮想空間による監視機能を実現し，攻撃の検出を行う．監視のための処理は常時カーネルモードにて待機しており，特定のシステムコール，一定回数のシステムコール発行時，および指定された周期で監視の処理を開始する．

3.4 実現方式

3.4.1 想定する環境

提案手法を実現し，カーネルの仮想記憶空間の監視を行う環境として OS は Linux を，CPU アーキテクチャは x86_64 を用いることを想定する．

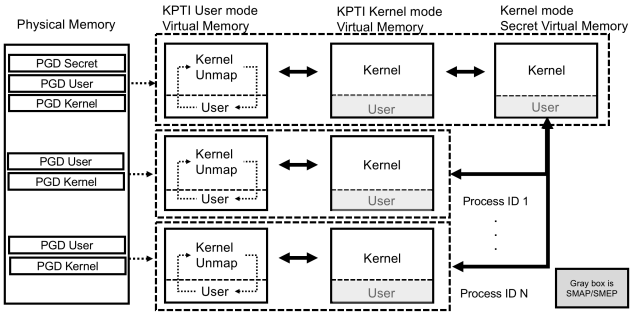

図 5　Linux におけるカーネル用仮想記憶空間を追加した概要図

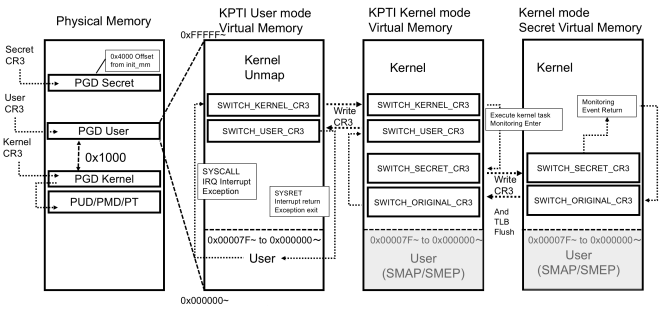

図 6　Linux におけるカーネル用仮想記憶空間への切替え処理

3.4.2　実現方式における仮想記憶空間管理

Linux において提案手法のカーネル用仮想記憶空間を実現した際の概要を図 **5** に示す．最新の Linux では，KPTI により，カーネルおよびプロセスごとに仮想記憶空間が 2 つ作成される．カーネルモードの仮想記憶区間を指す物理アドレスは，`mm_struct` 構造体の `init_mm` 変数がもつ `pgd` 変数および，KPTI のユーザモードでの仮想記憶空間を指す物理アドレスは `init_mm` 変数の `pgd` 変数から 1 ページサイズ分（x86_64 では 4Kbytes）を論理和した値となる．プロセス生成時には，`task_struct` 構造体の `mm_struct` 構造にある `pgd` 変数をユーザモードで用いる KPTI の仮想記憶空間，`pgd` 変数の物理アドレスから 1 ページサイズ分オフセットした物理アドレスにカーネルモードで用いる仮想記憶空間をそれぞれ複製する．

提案手法では，カーネルに対して新たな仮想記憶空間を作成し，物理アドレスを `init_mm` 変数の `pgd` 変数から 4 ページサイズ分（x86_64 では 16Kbytes）を論理和した値としている．カーネルコードおよびデータは `pgd` 変数から複製する．この新たに作成した仮想記憶空間を指す物理アドレスを利用することで，各プロセスでのカーネルモードから直接，監視のためのカーネル用仮想記憶空間への切替えを実現する．

3.4.3　実現方式における仮想記憶空間管理の切替え

KPTI を有効化している Linux におけるユーザモードおよびカーネルモード間の遷移と独自のカーネル用仮想記憶空間への切替え処理を図 **6** に示す．

ユーザモードにおいては，割込み（SYSCALL，IRQ Interrupt）および例外（Exception）を発生させ，カーネルモードへ遷移する．この際，カーネルモードでの仮想記憶空間への切替えをユーザモードの仮想記憶空間にマップされた SWITCH_KERNEL_CR3 関数を呼出し行う．カーネルモード用の仮想記憶空間へ切替えた後，SWITCH_SECRET_CR3 関数を呼出す．この関数は，`init_mm` 変数の `pgd` 変数の物理アドレスから 4 ページサイズのオフセットを計算し，CR3 レジスタに書込むことで監視用の仮想記憶空間への切替えを行う．その後，監視処理を行うかを以下の条件にて判定する．

(1) 監視対象の仮想記憶空間を変更操作するシステムコールまたは関数の呼出し
(2) 割込み，例外を規定回数呼出し
(3) 一定時間周期に達しているか

監視作業を終了後，SWITCH_ORIGINAL_CR3 関数において，`current` 変数（`task_struct` 構造体）の持つ `active_mm` 構造体の持つ `pgd` 変数の物理アドレスを CR3 レジスタに書込むことで，実行中プロセスでのカーネルモードの仮想記憶空間に切替える．それぞれの CR3 レジスタへの書込み後は，TLB フラッシュを行い，仮想アドレスから物理アドレスへの変換キャッシュをクリアする．その後，SWITCH_USER_CR3 関数を呼出し，ユーザモードの仮想記憶空間に切替え，割込みの終了（SYSRET Interrupt return）および例外終了（Exception exit）にてユーザモードに遷移する．

3.4.4　実現方式における仮想記憶空間レイアウト

Linux x86_64 における仮想記憶空間レイアウトにおける提案手法の仮想記憶空間で利用するカーネルコードおよび監視用データの領域を図 **7** に示す．監視用ならびに仮想記憶空間の切替えを行うカーネルコードと監視用データは Kernel text mapping（0xFFFFFFFF80000000 - 0xFFFFFFFF9FFFFFFF，512MBytes），監視対象のモジュールは Module mapping space（0xFFFFFFFFA0000000 - 0xFFFFFFFFFF5FFFFF，1526 MBytes）にそれぞれ配置する．監視用の仮想記憶空間はカーネルの仮想記憶空間からの複製であり，CoW（Copy on Write）を無効化することで，仮想記憶空間を切替えた後でも Module mapping space を参照可能とする．一方，監視用カーネルコードおよび監視用データが複製元であるカーネルの仮想記憶空間において書き換えられることを防ぐため，Page Table Entry にて読み書きをオフとし，参照不可能とする．

3.4.5　実現方式における仮想記憶空間の監視

Linux での独自のカーネル用仮想記憶空間の確保ならびに監視時の処理の流れを図 **8** に示し，以下で説明する．

(1) Linux ブート時にカーネルの仮想記憶空間の初期化処理（mm_init 関数）および KPTI によるユーザモードの仮想記憶空間の初期化処理（kaiser_init 関数）を行う
(2) 独自のカーネル用仮想記憶空間を物理記憶空間に確保

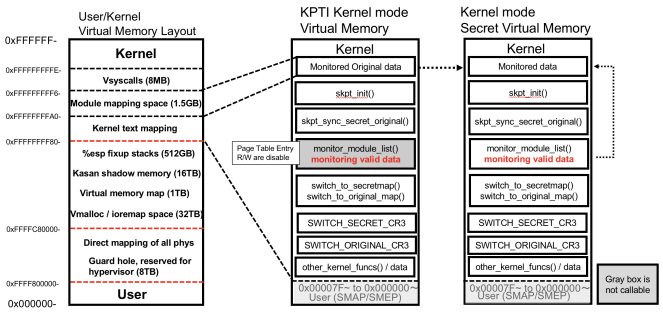

図 7　Linux x86_64 の仮想記憶空間レイアウトでのカーネル用仮想記憶空間のカーネルコードおよびデータ配置

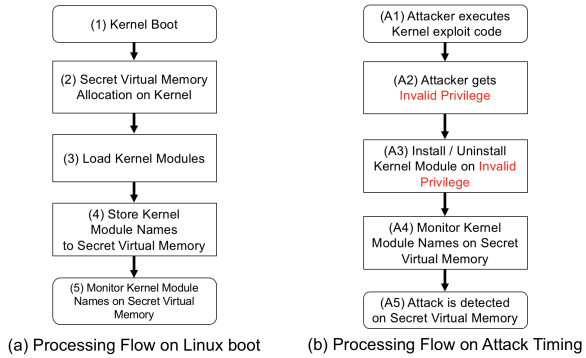

図 8　Linux における独自のカーネル用仮想記憶空間を用いた監視の処理フロー

し初期化処理を行う

(3) カーネルにおいてモジュール読込処理（load_default_modules 関数）を実行

(4) カーネルの仮想記憶空間と独自のカーネル用仮想記憶空間にて監視に必要となる領域を複製

(5) 独自のカーネル用仮想記憶空間にて監視を開始

攻撃者によるモジュールの操作に対する Linux におけるモジュール名リストの監視処理の流れを説明する．

（A1）攻撃者はユーザ権限にてカーネル脆弱性を実行

（A2）攻撃者はカーネル脆弱性により特権を奪取

（A3）攻撃者は不正なモジュールを挿入（sys_init_module システムコール），またはモジュールの除去（sys_delete_module システムコール）を行う

（A4）モジュール関係のシステムコール発行を捕捉（entry_SYSCALL_64 関数内）し，モジュールの初期化関数が呼ばれる前（add_unformed_module 関数）に監視用の仮想記憶空間に切替える

（A5）モジュール名リスト（`modules` 変数）とあらかじめ決められたモジュール名リストを比較し，不正なモジュールの検出を行う．その後，実行中プロセスの `current` 変数（`task_struct` 構造体）の仮想記憶空間へ切替える．

表 1　提案手法の評価環境

OS	Debian 9.0 (Linux Kernel 4.4.138, x86_64)
CPU	Intel(R) Core(TM) i7-7700HQ @ 2.80GHz
Memory	16 GB

4. 評価

4.1 評価の目的と評価環境

本研究で行った評価の目的と内容を以下に示す．

(1) カーネルモジュール監視判断の実験
提案手法を導入し，仮想記憶空間の切替え後にモジュールへの監視判断が適切にできるか評価し，また，不正な場合の検出にかかる時間を測定した．

(2) 提案手法の導入によるオーバヘッドの測定
提案手法の導入後において，ベンチマークの測定を行い．仮想記憶空間の切替えおよび監視環境でも動作すること，およびオーバヘッドを測定した．

上記の評価はクラウド環境の計算機上で行った．評価に利用した環境を**表 1** に示す．提案手法の実装は Linux Kernel 4.4.138 に行い，CPU アーキテクチャは x86_64，ディストリビューションは Debian 9.0 を用い，起動時に CPU を 1 つのみ認識するように設定した．Linux Kernel の 15 個のファイルについて，仮想記憶空間の切替えおよび監視機能を追加し，271 行で実現した．

4.2 カーネルモジュール監視判断の実験

評価は，正規および不正とする 2 つのモジュールを自作し，それぞれの挿入後に仮想記憶空間の切替え後に監視判断が有効に動作するかをログへ出力させることで行った．正規のモジュール名はあらかじめ監視用の仮想記憶空間に Linux 起動時に保持させておく．また，不正なモジュールでは，初期化関数においてカーネルのモジュール名リストから隠蔽のためにモジュール名を削除する処理を行うものとする．監視時のログにおいては，正規のモジュールの場合は “Valid kernel module”，不正なモジュールの場合は “Invalid kernel module” と文字列にて表示される．

提案手法による監視判断のログ出力を**図 9** に示す．左から，カーネル起動時からの時刻，ログ出力文字列を表している．提案手法では，正規のモジュールの判断は正しく行われており，挿入処理開始から 51.32 ms で監視判断が行われ，監視から 61.29 ms 後に初期化関数が呼ばれた．

不正なモジュールに対する判断も正しく行われており，挿入処理開始から 80.19 ms で監視判断が行われ，その後，44.37 ms 後に初期化関数およびモジュール内に定義したモジュール名リスト上書き関数が呼ばれた．

4.3 提案手法の導入によるオーバヘッドの測定

提案手法の導入前の Linux Kernel と導入後の Linux

```
[   72.398684] victim_module: module license 'unspecified' taints kernel.
[   72.401408] Disabling lock debugging due to kernel taint
// Switching to secret virtual memory and monitoring
[   72.449999] Valid kernel module valid_ module
// Switching to kernel virtual memory
[   72.511291] Valid Module Init
```

(a) Monitoring of Valid Kernel Module

```
[   69.043874] malicious_module: module license 'unspecified' taints kernel.
[   69.045217] Disabling lock debugging due to kernel taint
// Switching to secret virtual memory and monitoring
[   69.124065] Invalid kernel module malicious_module
// Switching to kernel virtual memory
[   69.168439] Attack Module Init
[   69.169387] calling change_module_name_struct_attack()
```

(b) Monitoring of Invalid Kernel Module

図 9 提案手法によるモジュールの監視ログ

表 2 提案手法の仮想記憶空間切替えによるオーバヘッド（単位：μs）

システムコール	提案手法導入前	提案手法導入後	オーバヘッド
fork+/bin/sh	337.797	383.715	45.918
fork+execve	72.013	75.553	3.540
fork+exit	68.204	69.804	1.600
open/close	1.007	1.954	0.947
read	0.261	1.074	0.813
write	0.229	1.038	0.809
stat	0.486	1.385	0.899
fstat	0.280	1.115	0.835

Kernel にてベンチマークソフトウェア lmbench をそれぞれ 10 回実行し，平均値からシステムコールの実行にかかるオーバヘッドを算出した．仮想記憶空間の切替えと監視処理の実行によるオーバヘッドへの影響を正確に測定するため，システムコール呼出毎に仮想記憶空間の切替えのみを行う場合，および仮想記憶空間の切替えに加えて一定回数毎のシステムコール呼出し時（100 回，1,000 回，10,000 回）に監視処理を行う場合をそれぞれ測定した．評価結果についてを**表 2**，および**表 3** に示す．lmbench において，fork+/bin/sh は約 54 回，fork+execve は 4 回，fork+exit は 2 回，open/close は 2 回，および，その他は 1 回のシステムコール呼出しが行われる．

表 2 から，オーバヘッドの上位 2 つのシステムコールは open/close（0.470 μs），また fork+exit（0.800 μs），下位 2 つでは fork+execve （0.885 μs）および stat（0.899 μs）となり，提案手法の導入後でシステムコール 1 回あたりのオーバヘッドは 0.470 μs から 0.889 μs となった．表 3 から，システムコール 100 回毎から 10,000 回毎の監視処理の実行頻度の違いにより，fork+/bin/sh（0.374 μs），およびその他のシステムコール（0.010 μs）程度のオーバヘッド低減となった．性能評価で用いた lmbench は，性能測定においてモジュールの挿入は行わない．このため，オーバヘッドはシステムコール発行毎に提案手法による監視判断のための仮想記憶空間の切替え，および監視処理となる．

表 3 提案手法導入後の監視処理によるオーバヘッド（単位：μs）

システムコール	100 回毎	1,000 回毎	10,000 回毎
fork+/bin/sh	382.781	383.780	382.407
fork+execve	76.027	75.803	76.486
fork+exit	69.251	69.678	69.200
open/close	1.947	1.954	1.937
read	1.069	1.069	1.067
write	1.033	1.030	1.031
stat	1.395	1.397	1.384
fstat	1.115	1.106	1.108

5. 考察

5.1 評価に対する考察

提案手法を実現した環境にて，正規および不正としたモジュール監視判断の実験の結果，いづれの場合でも，モジュールの初期化関数が実行される前に仮想記憶空間の切替えと監視を行えることを確認できた．これにより，多くの不正なモジュールが行う，初期化関数でのモジュール名リストの上書き処理の影響を受けることなく検出可能であると考える．評価においては，モジュールの初期化関数を実行させたが，検出時点ではモジュールの挿入処理は未完了のため，提案手法にて，検出後の挿入防止は可能である．

lmbench を用いた評価では，提案手法の導入によるシステムコール 1 回あたりのオーバヘッドは 0.470 μs から 0.889 μs であることを示された．主要なオーバヘッドはシステムコール発行時，監視判断のために独自のカーネル用仮想記憶空間に切替え，監視判断終了後，実行中プロセスのカーネル用仮想記憶空間に切替える処理により CR3 レジスタへの書込み，および TLB Flush を行う影響が大きい．この影響を低減するためには ASID （Address Space Identifier，x86_64 では PCID （Processor Context ID）），による TLB における仮想アドレスのキャッシュへのタグ付けを行う必要があると考えている．また，アプリケーション実行時のオーバヘッドは，アプリケーションのシステムコール発行頻度による．つまり，アプリケーションのユーザモードでの実行ではオーバヘッドは発生せず，カーネルモードでの実行時における仮想記憶空間の切替え処理の割合が高く，つづいて監視処理にて少しのオーバヘッドが生じている．測定において，システムコールの種類によりオーバヘッドは大きくは変わらないことから，アプリケーション全体の処理時間への影響は，ユーザモードからのシステムコール発行によるカーネルモードにおける仮想記憶空間の切替えと監視処理の実行回数に比例すると推察できる．

今後，他のベンチマークソフトウェアによるアプリーション実行時のオーバヘッド，監視対象を増やした場合の影響や攻撃に対する適切な監視頻度による効率化の調査を

検討している．

5.2 提案手法に対する考察

提案手法では，仮想記憶空間の切替え処理を監視対象の仮想記憶空間に配置し呼出している．攻撃者が切替え処理を攻撃対象とする可能性があるが，KASLR の対象としてカーネルの仮想記憶空間上でランダムに配置され一定の保護が行われる．

また，Linux x86_64 では Direct mapping に物理記憶空間（16TB）がカーネルモードの仮想記憶空間にマッピングされており，攻撃対象となる可能性がある．ダイレクトマッピング領域は，ページ単位で管理された物理記憶空間であり，特定の処理が配置されたページを選択して攻撃することは難しいが，一部のページをマッピングしないようにするなどで対応可能であると考えている．

カーネルへの攻撃においては，脆弱性の種別 [1] により，攻撃の対象や影響を受ける領域が異なり，その対策としてのセキュリティ機構も複数存在する．提案手法は，システムコール発行毎の仮想記憶空間の切替えと監視を実現しており，ユーザモードからカーネルの脆弱性を用いた攻撃の起点時やカーネルの仮想記憶空間へ影響を及ぼす一連の処理を捉えることが可能であると考えている．

モジュール挿入中の処理において，仮想記憶空間の切替えと監視を行なう場合でもカーネルを動作停止させることはなかった．監視処理をカーネルにおける任意の箇所で行えることから，監視対象の仮想記憶空間を追加することで他のセキュリティ機構の保護や連携できると考えている．また，独自の仮想記憶空間にてセキュリティ機構を動作させることで，カーネルの脆弱性を用いて攻撃された際にセキュリティ機構の監視機能バイパスなどを困難化させることが可能といえる．

6. 関連研究

OS におけるセキュリティ機構は，アクセス主体からデータや機能へのアクセスを権限やポリシによる細粒制御にて実現しており，ハードウェアからソフトウェアの複数のレイヤおよびモデルでのアクセス制御手法の提案がある [9]．

Linux でのセキュリティ機構としては SELinux[3], [10]，ケイパビリティ [4] および KASLR[11] などを実装しており，CPU による仮想記憶空間の保護は NX-bit[12] や SMAP/SMEP[7] による実行・アクセス制御が可能である．これらのセキュリティ機構に対して，カーネルモードの仮想記憶空間への攻撃として [6], [8], [13] などサイドチャネルを利用した KASLR の回避が提案されており，最新の Linux では，ユーザモードとカーネルモードの間で仮想記憶空間を分離した．しかし，ROP（Return Oriented Programming）とダイレクトマッピングを利用し，カーネルモードのみで脆弱性コードを実行する手法 [2], [14] も存在するためカーネルの仮想記憶空間の監視は必要となる．

カーネルに対する仮想記憶空間の監視手法としては，ハイパーバイザやセキュアモードを利用する手法 [15], [16], [17], [18] がある．これらは，カーネルへの攻撃の影響を回避できる利点がある．提案手法は，カーネル内部で監視を行うことから，ハイパーバイザやセキュアモードで動作するカーネル自体へ組込むことができる．カーネルの完全性の検証を行い，検証済みのカーネルコードのみ動作させる手法もあり，カーネルの改ざんを防ぐのに有効である．TCB（Trusted Computing Base）[19] では，起動時，SecVisor[15] では実行時に検証を行い，GRIM では GPU からの検証を行う [20]．これらは動作中の仮想記憶空間のデータは対象としていないため，提案手法にて動的な監視と組合せることが有効であるといえる．

仮想記憶空間の保護としては，仮想記憶空間をドメインやプロセス内の特定スレッド単位に分割し細粒度のアクセス制御を行う手法 [21], [22], [23], [24]，VMM 上のカーネルでモジュールとカーネルの仮想記憶空間を分け，耐故障性を向上させる手法 [25] がある．

動作中のカーネルコードやデータに対しては，制御フロー，データフローを追跡する手法 [26], [27], [28]，スタックを監視する手法 [5], [29], [30]，そして，権限情報の監視を行う手法 [31] などが提案されている．これらは，カーネル脆弱性を用いた攻撃による仮想記憶空間上のカーネル関数や権限情報のデータの上書き，不正なモジュールによるスタック書換えを検出することを可能としている．提案手法では，制御フロー，データフローやスタックは監視対象としていない．これらの情報は，監視処理のタイミングや監視先を絞り込むなどより効率的な攻撃検出のために有効と考えている．一方，提案方式は，仮想記憶空間を切替えによりカーネル脆弱性を利用した攻撃の影響を受けずに監視を行うことから既存手法でのセキュリティ機構やデータを保護でき，共存可能であるといえる．

今後，カーネル脆弱性の攻撃に対し，既存研究の手法に加えて，SELinux，KASLR や SMAP/SMEP などのセキュリティ機構と提案手法がどのような場合に有効かの比較，また，提案手法にて仮想記憶空間を切替え中に既存のセキュリティ機構と連携可能か否か検証する予定である．

7. おわりに

OS の脆弱性を利用した多様な攻撃に対して被害の軽減ならびに攻撃の困難化が求められてきている．攻撃の多くはカーネルの管理する仮想記憶空間を狙い，任意のプログラムコードを配置，または組み合わせて実行することで OS の特権奪取を行う．特権の最小化を行う強制アクセス制御やケイパビリティ，攻撃の防止を目的としたアドレス空間のランダム化や仮想記憶空間の実行権限の管理や，ユーザモードとカーネルモードの仮想記憶空間の分離は実現され

ているが依然として攻撃が成功する可能性はある．

我々はカーネルに独自の仮想記憶空間を追加することで，本来のカーネルの仮想記憶空間を保護するセキュリティ機構を提案し，不正なモジュールの挿入を仮想記憶空間上で検出することを可能とした．また，Linux を用いて実現し，事例とした不正なモジュールの挿入を検出できることを示した．性能評価では，ベンチマークにおいて性能への影響はシステムコール 1 回あたり 0.470 μs から 0.889 μs であることを示した．

参考文献

[1] Chen, H., et al.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems, the 2nd Asia-Pacific Workshop on Systems (APSys), (2011).

[2] Kemerlis, P, V., et al.: ret2dir - Rethinking Kernel Isolation, the 23rd USENIX Conference on Security Symposium, pp. 957–972, (2014).

[3] Security-enhanced Linux, available from `http://www.nsa.gov/research/selinux/`, (accessed 2018-08-10).

[4] Linden, A. T.: Operating System Structures to Support Security and Reliable Software, ACM Computing Surveys (CSUR), vol. 8, no. 4, pp. 409445, (1976).

[5] Kemerlis, P, V., et al.: kGuard - Lightweight Kernel Protection against Return-to-User Attacks, the 21st USENIX Conference on Security symposium, (2012).

[6] Hund, R., et al.: Practical Timing Side Channel Attacks against Kernel Space ASLR, 2013 IEEE Symposium on Security and Privacy, pp. 191–205, (2013)

[7] Mulnix D.: Intel Xeon Processor D Product Family Technical Overview, `https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview`, (2015), (accessed 2018-08-10).

[8] Lipp, M., et al.: KASLR is Dead - Long Live KASLR, 2017 International Symposium on Engineering Secure Software and Systems (ESSoS), vol. 10379, no. 3, pp. 161–176, (2017).

[9] Shu, R., et al.: A Study of Security Isolation Techniques, ACM Computing Surveys (CSUR), vol. 49, no. 3, pp. 137, (2016).

[10] Spencer, R., et al.: The Flask Security Architecture: System Support for Diverse Security Policies, the 8th Conference on USENIX Security Symposium, (1999).

[11] Shacham, H., et al.: On the effectiveness of address-space randomization. the 11th ACM Conference on Computer and Communications Security (CCS), pp. 298–307, (2004).

[12] Ingo Molnar, [announce] [patch] NX (No eXecute) support for x86, 2.6.7-rc2-bk2, available from `http://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html`, (2004). (accessed 2018-08-10).

[13] Jang, Y., et al.: Breaking Kernel Address Space Layout Randomization with Intel TSX, the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 380-392. (2016).

[14] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), the 14th ACM Conference on Computer and Communications Security (CCS), pp. 552-561, (2007).

[15] Seshadri, A., et al.: SecVisor - a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, the 21st ACM SIGOPS symposium on Operating systems principles (SOSP), pp. 335–350, (2007).

[16] Azab, A., et al.: SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM, the 2011 Network and Distributed System Security Symposium (NDSS), (2016).

[17] Zhang, F. and Zhang, H.: SoK A Study of Using Hardware-assisted Isolated Execution Environments for Security, the Hardware and Architectural Support for Security and Privacy 2016, pp. 1-8, (2016).

[18] Cho, Y., Kwnon, D., Yi, H. and Paek, Y.: Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM, the 2017 Network and Distributed System Security Symposium (NDSS), (2017).

[19] Trusted computing group. tpm main specification. `"http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2003, (accessed 2018-08-10).

[20] Koromilas, L., et al.: GRIM - Leveraging GPUs for Kernel Integrity Monitoring, the 19th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 3–23 (2016).

[21] Witchel, E, Rhee, J. and Asanovic, K.: Mondrix - memory isolation for linux using mondriaan memory protection, the 20th ACM symposium on Operating systems principles (SOSP), pp. 31–4, (2005).

[22] Castro, M., et al.: Fast byte-granularity software fault isolation, the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP), pp. 45–58, (2009).

[23] Hsu. C, T., et al.: Enforcing Least Privilege Memory Views for Multithreaded Applications, the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 393–405, (2016).

[24] Litton, J., et al.: Light-Weight Contexts - An OS Abstraction for Safety and Performance, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2016).

[25] Srivastava, A. and Giffin, T, J.: Efficient Monitoring of Untrusted Kernel-Mode Execution, the 18th Annual Network and Distributed System Security Conference (NDSS), (2011).

[26] Abadi, M., Budiu, Mihai., Erlingsson, U. and Ligatti, J.: Control-Flow Integrity Principles, Implementations, the 12th ACM Conference on Computer and Communications Security (CCS), pp. 340–353, (2005).

[27] Song, C., Lee, B., Lu, K., Harris, W., Kim, T. and Lee, W.: Enforcing Kernel Security Invariants with Data Flow Integrity, the 2016 Annual Network and Distributed System Security Symposium (NDSS), (2016).

[28] Ge, X., Cui, W. and Jaeger, T.: GRIFFIN: Guarding Control Flows Using Intel Processor Trace, the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS), pp. 585–598, (2017).

[29] Ikegami, Y. and Yamauchi, T.: Proposal of Kernel Rootkits Detection Method by Comparing Kernel Stack," IPSJ Journal, vol. 55, no. 9, pp. 2047-2060, (2014).

[30] Huang, W., et al.: LMP: Light-Weighted Memory Protection with Hardware Assistance, the 32nd Annual Conference on Computer Security Applications (ACSAC), pp. 460–470, (2016).

[31] 赤尾洋平, 山内利宏. : システムコール処理による権限の変化に着目した権限昇格攻撃の防止手法", コンピュータセキュリティシンポジウム 2016，pp.542–549 (2016).