CL-004

攻撃ユーザプロセスの利用するカーネルコードの追跡と特定手法の提案と評価

葛野 弘樹[1)]　　山内 利宏[2)]
Hiroki Kuzuno　Toshihiro Yamauchi

概要

オペレーティングシステムへの攻撃として，カーネル脆弱性を利用したメモリ破壊による特権奪取や競合条件の発生による動作停止が指摘されている．攻撃困難化を目的としたカーネルの攻撃面最小化手法として，kRazor，ならびに KASR はユーザプロセスに対して最小限のカーネルコードやデータのみを提供する手法を提案している．kRazor および KASR では，ユーザプロセス毎のカーネルコード実行許可判定やカーネルの再構成を行うため，アプリケーションの更新対応やユーザプロセス切替えに伴うカーネル構成変更を必要とする．我々は，新たな攻撃面最小化手法として，ユーザプロセスの実行中に脆弱なカーネルコードのみを利用不可とし，迅速な攻撃対応と低負荷の実現を目指したセキュリティ機構を提案している．しかし，脆弱なカーネルコードの仮想アドレス範囲の特定と登録は手動で行う必要がある．本稿では，先行提案の課題を解決するため，カーネル脆弱性に関するカーネルコードを動的に追跡し，仮想アドレスの範囲を特定する手法を提案する．提案手法では，ユーザプロセスによるカーネルへの攻撃実行時，および攻撃未実行時に利用するカーネルコードを追跡し，カーネルのデバッグ情報を参照することで仮想アドレスの範囲を特定する．提案手法を Linux にて実現し，擬似的に複数のカーネル脆弱性を導入した環境において，攻撃を行うユーザプロセスの利用する脆弱なカーネルコードの追跡と仮想アドレス範囲の特定可能性について評価した．

1　はじめに

オペレーティングシステム（OS）において，脆弱性を含むカーネルコード（カーネル脆弱性）を利用した攻撃への対策が課題とされている．カーネル脆弱性を利用した攻撃により，カーネル仮想記憶空間の改竄による特権奪取や強制的に競合条件を引き起こし，カーネルを停止させる Denial of Service （DoS）がある．

カーネル脆弱性を利用した攻撃への対策として，KASLR (kernel address space layout randomization) は，カーネルの仮想記憶空間上にあるカーネルコードやカーネルデータをランダム配置することで攻撃時の仮想アドレス推測を困難化する [1]．しかし，攻撃者のユーザプロセスにてカーネル脆弱性および攻撃対象のカーネルデータの仮想アドレスを特定された場合，カーネルへの攻撃は成功する．

kRazor ならびに KASR （kernel attack surface reduction）はカーネルにおける攻撃面最小化の手法を提案している．kRazor はアプリケーション（以降，AP と略す）の動作に必要なカーネルコード一覧を事前収集し，ユーザプロセスとして実行する際に事前収集したカーネルコード一覧に含まれるカーネルコードのみ許可することで攻撃面最小化を行う [2]．KASR は仮想マシンモニタ（以降，VMM と略す）を利用することで仮想マシン上のカーネルの起動から終了，および AP 動作に必要なカーネルコードをカーネルを事前動作させながら取得する．そして，ユーザプロセスの実行時に特定のカーネルコードのみ実行可能なカーネルを動作させることで攻撃面最小化を行う [3]．

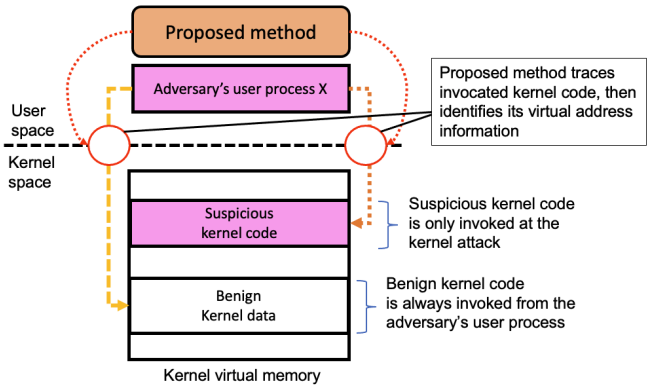

図 1　提案手法の概要

kRazor および KASR による攻撃面最小化により，ユーザプロセスの動作において攻撃に利用可能なカーネル脆弱性と関連するカーネルコードを必要としない場合，攻撃自体は制限可能である．しかし，次の課題があると考えている．

課題：AP 単位での攻撃面最小化

攻撃面最小化のため，AP の実行に必要とするカーネルコードを事前収集し，予め利用可能なカーネルコードの情報を AP 毎に用意する必要がある．大規模な AP の全ての動作に対して必要なカーネルコードを網羅することは困難であり，また，AP の更新に対応しなければならない．

我々は， ユーザプロセス毎にカーネルの仮想記憶空間を操作することで，脆弱なカーネルコードのみ利用不可とし，攻撃の困難化を図るセキュリティ機構を提案している [4]．提案しているセキュリティ機構はカーネルコードのページ参照を動的に制御する機能を備える．一方，課題として，利用不可とするカーネル脆弱性に関するカーネルコードの網羅と仮想アドレス範囲の特定と登録は手動で行わなければならい．

本稿では，先の提案手法 [4] の課題を解決するため，ユーザプロセスの攻撃時に利用するカーネル機能に着目し，カーネルコードの追跡と仮想アドレス範囲の特定を行う手法を提案する．図 1 に提案手法の概要を示す．提案手法は，ユーザプロセスの利用するカーネルコードを動的に追跡し，仮想アドレスの範囲を特定する．提案手法においては，攻撃を行う Proof of concept（PoC）コードを追跡対象のユーザプロセスとして動作させる．提案手法では，PoC コードのプログラムコードサイズが小規模であることを想定し，次の 2 種類のカーネルコード一覧の取得を行う．

1）　セコム株式会社 IS 研究所
2）　岡山大学学術研究院自然科学学域

- 攻撃実行時：PoC コードによる攻撃が行われる場合に呼出されるカーネルコードの一覧.
- 攻撃未実行時：PoC コードより，攻撃を行う箇所を除いた，攻撃が行われない場合に呼出されるカーネルコードの一覧.

攻撃実行時および攻撃未実行時に呼出されるカーネルコードの一覧を比較し，カーネル脆弱性と想定される攻撃に関連するカーネルコード（図 1 における Suspicious kernel code）のみの仮想アドレスの範囲を特定し，その結果をプロファイルとしてカーネル脆弱性の無効化に用いることを可能にする.

実攻撃を想定し，かつ AP への影響を抑えた攻撃面最小化を行うため，プロファイルの先の提案手法 [4] での利用を検討する．これにより，AP 毎に必要な事前調査を必要としない．また，攻撃面最小化を最小限のカーネルコードの無効化にて実現に繋がると考えている.

本稿での研究貢献は以下の通りである：

1. ユーザプロセスの利用するカーネルコードの動的な追跡と仮想アドレスの範囲を特定するための機構の設計と実装を行い，提案手法を実現した Linux にて，機能検証を進めた.
2. 提案手法の評価結果として，擬似的なカーネル脆弱性を攻撃に利用するユーザプロセスを動的に追跡し，カーネルコード一覧の取得と仮想アドレスの範囲を特定可能であることを確認した．また，ユーザプロセスおよびカーネル動作に影響を及ぼす可能性について検討した.

2 背景知識

2.1 カーネル脆弱性を用いた攻撃

カーネルへの攻撃に利用可能とされる実装不備として，カーネル脆弱性は，実装の内容から 10 種類に分類されている [5]．カーネル脆弱性毎に攻撃利用時にカーネルに与える効果は異なる．代表的な効果として，特権奪取ならびにカーネルの動作停止（DoS）がある．提案手法の評価に用いるため，擬似的に導入するカーネル脆弱性として利用するメモリ管理の不備，および競合条件の発生に関するカーネル脆弱性を以下に述べる.

- バッファオーバーフロー：スタック領域あるいはヒープ領域の上書きが行われる．バッファオーバーフローにより，権限情報が改竄された場合，特権奪取に繋がる.
- 解放済みメモリの参照（Use-After-Free）：解放済みメモリ領域の参照が行われる．不定な値の読込みと利用によりカーネル動作が不定となり，DoS に繋がる.
- 競合条件の発生：フラグ操作処理の誤りによる意図的な無限ループやデッドロックの発生が行われる．カーネル動作の処理待ちが発生し，DoS に繋がる.

2.2 脅威モデル

本稿での脅威モデルとして，攻撃者の目標はカーネル脆弱性を利用したバッファオーバーフローによる特権奪取，ならびに解放済みメモリ利用や競合条件の発生に起因した DoS によるカーネルの動作停止とする．想定する攻撃シナリオとして，攻撃者はカーネル脆弱性と関連するカーネルコードを呼出すユーザプロセスを実行し，カーネルへの攻撃を行う。脅威モデルにて想定する攻撃者の要件，および環境を以下にまとめる.

- 攻撃者： 非特権ユーザ権限を有し，カーネル脆弱性を呼び出すための攻撃コードを実行する PoC コードをユーザプロセスとして実行可能.
- カーネル：攻撃者に利用可能なカーネル脆弱性を含む．実行中のユーザプロセスに対して，アクセス制御機構以外のセキュリティ制限の適用は行わない.
- カーネル脆弱性：攻撃者の実行するユーザプロセスより特権奪取のために利用可能なカーネル脆弱性，ならびに DoS を発生させ，カーネルの停止に利用可能なカーネル脆弱性.
- 攻撃対象： 攻撃者の実行するユーザプロセスにおいて特権奪取のために攻撃対象として指定される仮想アドレスに配置されるカーネルデータ（例．ユーザプロセスの権限情報），ならびに DoS を引き起こすための解放済みメモリのカーネルデータ，ならびにフラグ情報（例，ファイルシステムのロック）.

3 提案手法と実現方式

3.1 提案手法の要件

提案手法では，攻撃を行うユーザプロセスの利用するカーネル機能に付随するカーネルコードの動的な追跡と仮想アドレス範囲を特定するために，設計において以下の要件を満たすことを目指した.

要件：ユーザプロセスの実行に必要とされるカーネル機能において，システムコールを介して呼出されるカーネルコードの動的な追跡，およびカーネルコードとカーネルの仮想記憶空間上の仮想アドレス範囲の紐付けを行う.

3.2 設計

要件を満たす設計として，図 2 に提案手法の処理概要を示す．提案手法では，動作中のカーネルにおいて，ユーザプロセスの実行に対応するカーネルコードの実行を追跡し，攻撃実行時，および攻撃未実行時のカーネルコードを呼出処理に応じて取得する．カーネルコードの取得後，カーネル仮想空間に配置されるカーネルコードからカーネルコードと仮想アドレス範囲の紐付けを行う．攻撃実行時のみに含まれるカーネルコードと仮想アドレスから，カーネル脆弱性に関するカーネルコードと仮想アドレス範囲を特定する．提案手法における，一連の処理を以下で述べる.

- (Step 1-1)：攻撃者はユーザプロセスを実行．カーネルにて，該当ユーザプロセスのみを追跡するため，プロセス ID を追跡対象として登録する.
- (Step 1-2)：攻撃者のユーザプロセスはカーネル脆弱性を利用した攻撃を開始する.
- (Step 1-3)： 追跡対象としたユーザプロセスからのシステムコール要求の場合，システムコールを実行する際に関連する一連のカーネルコードの収集を行い，記録する.
- (Step 1-4)：攻撃者のユーザプロセスにより，カーネル脆弱性を利用した攻撃を行い，攻撃が成功した場合，カーネルコードの収集を終了する．攻撃が行われない場合，ユーザプロセスの終了に伴いカーネルコードの収集を終了する.
- (Step 2-1)：収集したカーネルコードから，デバッ

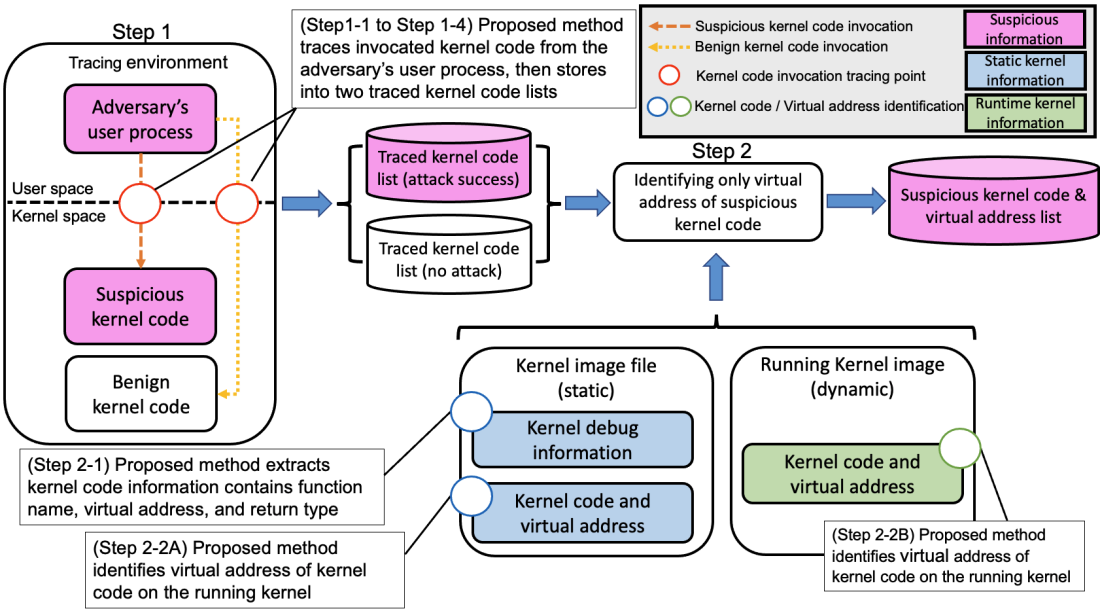

図 2　提案手法の処理概要

グ情報を利用し，詳細な情報（仮想アドレス範囲，定義ファイル，定義位置，関数の型，引数の数，ならびに引数の型，等）を取得する．

- （Step 2-2A）：KASLR を利用しない場合．収集したカーネルコード，およびデバッグ情報から取得した仮想アドレス範囲を紐付けする．攻撃実行時のみ含まれるカーネルコードと仮想アドレス範囲を抽出，カーネルイメージとして含まれていることを確認し，カーネル脆弱性に関するカーネルコードと仮想アドレスの組合せとする．
- （Step 2-2B）：KASLR を利用する場合．攻撃実行時のみ含まれるカーネルコードを抽出し，収集したカーネルコードがカーネルイメージに含まれていることを確認する．KASLR により，カーネルコードの仮想アドレス範囲の値はカーネル起動の際に定まり，都度異なることから，起動時においてカーネル脆弱性に関するカーネルコードと仮想アドレスの組合せを生成する．

3.2.1　プロファイル生成

プロファイルは，提案手法により収集したカーネル脆弱性に関するカーネルコードと仮想アドレス範囲の組合せとする．

プロファイルの生成のため，カーネル脆弱性を利用した PoC コードを用いる．カーネル脆弱性を利用した攻撃が成功する際のカーネルコードの一覧は，攻撃実行時の追跡結果と攻撃未実行時の追跡結果より取得する．

- 攻撃実行時：PoC コード実行時に攻撃が行われ，かつ攻撃が成功したと判定された場合に呼出されるカーネルコードの一覧．
- 攻撃未実行時：PoC コードより，攻撃を行うプログラムコードを除外し，PoC コードを実行．攻撃が試行されない場合に呼出されるカーネルコードの一覧．

カーネルへの攻撃に関与しない，ユーザプロセスの実行に必要とされる汎用的，または一般的なカーネルコードの一覧を追跡結果として得るため，攻撃未実行時として，攻撃を行うプログラムコードを除外した PoC コードを実行する．PoC コードのプログラムサイズが小規模である場合，攻撃未実行時のカーネルコード一覧は攻撃実行時にも含まれると考えている．攻撃実行時のみに含まれるカーネルコードと仮想アドレス範囲の組合せをカーネル脆弱性を用いて攻撃が成功する際のカーネルコードの一覧として抽出し，プロファイルとする．

プロファイルを用いて，先行して提案しているセキュリティ機構 [4] において，利用不可とするべきカーネルコードの登録に用い，カーネルへの攻撃面最小化を実現する（詳細は 5.3 節を参照）．

3.3　カーネル脆弱性を利用した攻撃の追跡基準

追跡対象のユーザプロセスを実行した際，3.2 節で示した一連の処理により，カーネルにおいてユーザプロセスから呼び出されたカーネルコードが列挙され，最終的にカーネルコードに対応する仮想アドレス範囲の一覧が得られる．カーネルへの攻撃に紐づくカーネルコードかの判断は，カーネル脆弱性を利用した攻撃の成功可否を判定して行う．

提案手法においては，カーネル脆弱性を介して攻撃可能な PoC コードを実行する際に，以下のカーネル脆弱性毎の攻撃成功可否基準に従い，攻撃の有効性を判定する．

- 特権奪取：攻撃を行うユーザプロセスの実行前後において，権限情報の通常ユーザから特権ユーザへの変化を攻撃成功とみなす．
- DoS：攻撃を行うユーザプロセスの実行中にカーネルが停止した場合，攻撃成功とみなす．カーネルが停止する場合，カーネルコードの呼出し履歴の取得が困難となることから，永続性のある記録媒体へのカーネルコードの呼出毎の保存，またはリモートの計算機へのログ出力を行う．

カーネル脆弱性毎の攻撃成功可否基準を利用し，攻撃成功と判定された場合のみ，攻撃実行時におけるユーザ

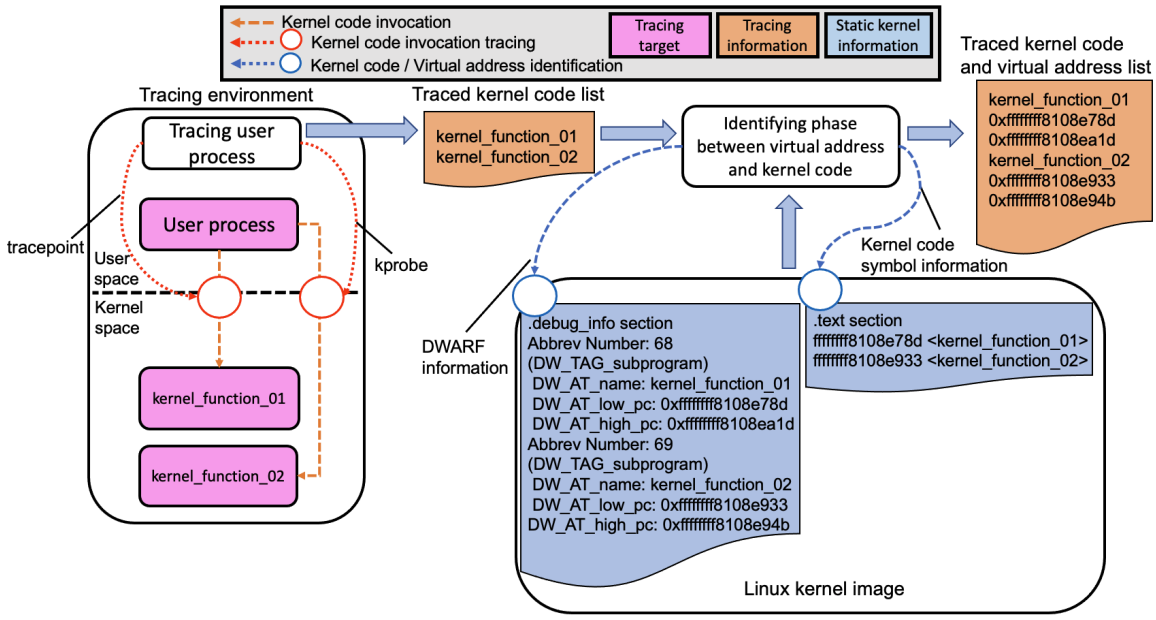

図 3　実現方式の概要

プロセスからの要求に伴い実行されたカーネル脆弱性に関連するカーネルコードの一覧とする．

3.4　実現方式

提案手法の実現環境は x86_64 CPU アーキテクチャの Linux を想定し，特定のユーザプロセスを指定した際にカーネルコード呼出しの追跡ならびにカーネルコードと仮想アドレス範囲の特定を行う実現方式を考案した．

3.4.1　カーネルコードの特定

実現方式において，追跡対象とするユーザプロセスを動作させるカーネルのトレーシング機能を利用して行う．図 3 に実現方式における処理の概要を示す．Linux においては，次のカーネルのトレーシング機能を利用する．

- tracepoints：Linux カーネルにおいて，特定の実行地点においてソースコード側より静的に呼出すトレーシング機能．
- kprobes：Linux カーネルにおいて，カーネルコードのシンボル情報に基づき追跡地点を動的に登録するトレーシング機能．

提案手法においては，`kprobes` に指定するカーネルコードのシンボルは，追跡対象としたユーザプロセスから利用可能なカーネルコードとし，実際に呼出されたカーネルコードからカーネルコードの名称を収集する．`tracepoints` はカーネルのソースコードにおいて静的に挿入されており，補足情報として利用する（例．システムコール関数の呼出し）．

3.4.2　仮想アドレスの特定

実現方式において，追跡対象のユーザプロセスを動作させ取得したカーネル関数名一覧から仮想アドレス範囲の特定は，以下の静的な仮想アドレス特定手段ならびに動的な仮想アドレス特定手段により行う．両方の手段を利用するかはカーネルの仮想空間におけるカーネルコードの位置を起動毎に変化させる KASLR の有無に依存する．

- 静的な仮想アドレス特定手段：KASLR 有無に関わらず利用．カーネルの `.debug_info` セクションに格納される DWARF（Debug With Attribute Record Format）情報，ならびにカーネルコードに関する `.text` セクションの情報を利用して仮想アドレス範囲を特定
- 動的な仮想アドレス特定手段：KASLR 有効時に利用，カーネルの起動時にカーネルコードを直接参照し，仮想アドレス範囲を特定

静的な仮想アドレス特定手段にて用いる DWARF 情報の値を表 1 に示す．DWARF 情報を利用することで，仮想アドレス範囲，定義ファイル，定義位置，関数の型，引数の数，ならびに引数の型等を特定することが可能である．

3.4.3　ユーザプロセスの追跡例

実現方式を用いたユーザプロセスの追跡例を図 4 に示す．追跡対象のユーザプロセスは特定のシステムコールを呼出し，提案手法により，ユーザプロセスの利用するシステムコールに関連するカーネルコードについて，カーネルコードの収集と仮想アドレス範囲の特定が行われる．追跡終了基準の判定のため，特権奪取を模擬し，特権ユーザにてユーザプロセスを実行し，通常ユーザから特権ユーザへの権限変更を行う．

図 4 においては，一時的に通常ユーザ権限として実行されたユーザプロセスがシステムコール `write` を呼出す．カーネルコード `vfs_write` の実行が捕捉され，カーネルコードとして表示されている．ユーザプロセスが特権ユーザに変更した段階で追跡を終了し，収集したカーネルコードの関数名 `vfs_write` から仮想アドレス範囲は `ffffffff811b8cd0` から `ffffffff811b8e64` であることを特定結果として表示している．

4　評価

4.1　評価の目的と評価環境

提案手法に対するユーザプロセスとカーネルにおける追跡性能の調査を目的として，機能評価の項目と内容を

```
1. [*] start tracing process
2. uid=0(root) euid=1000(user) gid=0(root)
3. [*] start system call invocation
4. [*] kernel code information
5. vfs_write pid: 14003
6. [*] finish system call invocation
7. uid=0(root) euid=0(root) gid=0(root)
8. [*] finish tracing process
9. [*] virtual address rage of kernel code list
10. vfs_write ffffffff811b8cd0 ffffffff811b8e64
```

図 4　提案手法によるユーザプロセスの追跡例

表 1　提案手法で利用する DWARF 情報（●：利用，—：利用予定）

Item	Description	Approach
DW_AT_name	関数名	●
DW_AT_decl_file	関数定義ファイル名	—
DW_AT_decl_line	関数定義行番号	—
DW_AT_type	関数戻り値の型	—
DW_AT_low_pc	プログラムコードの開始仮想アドレス	●
DW_AT_high_pc	プログラムコードの終了仮想アドレス	●

以下に示す．

1. カーネル脆弱性に対する評価
   提案手法を用いて，意図的に導入した複数のカーネル脆弱性を攻撃に利用可能なユーザプロセスを実行する．攻撃実行時および攻撃未実行時に動的に追跡し，カーネル脆弱性に関連するカーネルコードの取得，ならびに仮想アドレス範囲を特定可能か評価した．

評価環境として，機能評価に評価用計算機を用いた．評価用計算機は Intel(R) Core(TM) i7 7700HQ（2.80GHz, 4 コア，メモリ 16GB），OS は Debian 9.0，Linux kernel 5.0.0 とし，KASLR は適用しない．提案手法の実現方式の実装は AP として，712 行にて実現した．また，評価用のカーネル脆弱性の実装を Linux kernel 5.0.0 に行い，4 個のファイルに対して 192 行の追加，PoC コードは 143 行にて実現している．

4.1.1　導入するカーネル脆弱性

提案手法の機能評価のため，3 種類のカーネル脆弱性毎に独自システムコールをカーネルに意図的に導入した．

- カーネル脆弱性 1（バッファオーバーフロー）：独自システムコール 1 の実行中，カーネル脆弱性 1 を含むカーネルコード 1-1，1-2 を呼出す．ユーザプロセスの権限情報をバッファオーバーフローにより改竄し，特権奪取可能とする．
- カーネル脆弱性 2（解放済みメモリの参照）：独自システムコール 2 の実行中にメモリ領域を確保，カーネルコード 2-1 を呼出し，該当メモリ領域を開放する．その後，独自システムコール 2 にて，開放済みメモリ領域を参照し，DoS を発生させる．
- カーネル脆弱性 3（競合条件の発生）：独自システムコール 3 実行中，カーネル脆弱性 3 を含むカーネルコード 3-1 を呼出す．カーネルコード 3-1 では，タスク処理のロック確保を発生させ，カーネル動作に影響する DoS を発生させる．

評価において，PoC コードとしてユーザプロセスを実行し，攻撃実行時は導入したカーネル脆弱性を利用できる独自システムコールを呼出し，特権奪取，および DoS 発生を行う．一方，攻撃未実行時は，PoC コードにて，独自システムコールの呼出しは行わない．

```
1.  [*] start tracing process
2.  uid=1000(user) gid=1000(user) groups=1000(user)
3.  [*] start system call invocation
4.  [*] kernel code information
5.  // Buffer overflow
6.  a.out kprobe:__x64_sys_kvuln01 3241
7.  a.out kprobe:support_kvuln01_01 3241
8.  a.out kprobe:support_kvuln01_02 3241
9.  [*] finish system call invocation
10. uid=0(root) gid=0(root) groups=0(root)
11. [*] finish tracing process
12. [*] virtual address range of kernel code list
13. __x64_sys_write ffffffff812685f0
14. __x64_sys_kvuln01 ffffffff8109a109
15. __x64_sys_nanosleep ffffffff81104410
16. __ia32_sys_getuid ffffffff810943d0
17. __ia32_sys_getgid ffffffff81094580
18. __x64_sys_kvuln01 ffffffff8109a109, ffffffff8109a129
19. support_kvuln01_01 ffffffff81099bd0, ffffffff81099e60
20. support_kvuln01_02 ffffffff81099e95, ffffffff81099a6cd
21. __x64_sys_openat ffffffff81264670
22. __x64_sys_lseek ffffffff81265bb0
23. __x64_sys_newfstat ffffffff8126cf10
24. __x64_sys_read ffffffff81268500
25. __x64_sys_close ffffffff812625d0
26. __x64_sys_exit_group ffffffff81084b90

27. // Kernel code symbol information
28. ffffffff8109a109 <__x64_sys_kvuln01>
29. ffffffff81099bd0 <support_kvuln01_01>
30. ffffffff81099e95 <support_kvuln01_02>
```

図 5　カーネル脆弱性 1 を利用した攻撃の追跡結果

4.2　カーネル脆弱性に対する評価

評価においては，提案手法を用いて，3 種類のカーネル脆弱性を利用した攻撃に対して，攻撃発生前よりユーザプロセスを追跡し，攻撃終了後までのカーネルコードの収集と仮想アドレス範囲の特定結果について出力を行う．

4.2.1　カーネル脆弱性 1 を利用した攻撃の追跡

図 5 にカーネル脆弱性 1 を介して攻撃を行うユーザプロセスの追跡結果を示す．

攻撃実行前のユーザプロセスは通常ユーザの権限（ユーザ ID 1,000）であることが 2 行目にて表示されている．6 行目から 8 行目にて，独自システムコール 1 を呼出し，カーネル脆弱性 1 を利用するカーネルコード 1-1，および 1-2 の呼出しについて，追跡結果が表示されている．独自システムコール 1 の終了後，10 行目にてユーザプロセスは特権奪取（ユーザ ID 0）に成功しており，同時に提案手法の追跡は終了したことを確認できる．

カーネルコードの仮想アドレス範囲の特定結果として，13 行目から 26 行目にかけ，提案手法にてユーザプロセスの追跡により収集したカーネルコードと仮想アドレスの組合せを表示している．18 行目から 20 行目にかけては攻撃実行時のみに収集されたカーネルコードであり，独自システムコール 1 およびカーネル脆弱性 1 を含むカーネルコード 1-1，1-2 の仮想アドレス範囲を特定している．また，28 行目から 30 行目は，カーネルイメージに含まれるシンボル情報から攻撃実行時のみに呼出されるカーネルコードと仮想アドレスの組合せを抽出した結果である．提案手法において特定した開始仮想アドレスと同値であることを確認できる．

4.2.2　カーネル脆弱性 2 を利用した攻撃の追跡

図 6 にて，カーネル脆弱性 2 を介して攻撃を行うユーザプロセスの追跡結果を示す．

```
1.  [*] start tracing process
2.  [*] start system call invocation
3.  [*] kernel code information
4.  // Use after free: kmalloc, then free and use
5.  a.out kprobe:__x64_sys_kvuln02 3251
6.  a.out kprobe:support_kvuln02_01 3251
7.  [*] finish system call invocation
8.  // kernel is stopped

7.  [*] virtual address rage of kernel code list
8.  __x64_sys_newfstat ffffffff8126cf10
9.  __x64_sys_kvuln02 ffffffff81099d30, ffffffff81099da3
10.                   ffffffff8109a1a7, ffffffff8109a1f0
11. support_kvuln02_01 ffffffff81099c50, ffffffff81099cab
12.                    ffffffff8109a124, ffffffff8109a15e
13. __x64_sys_write ffffffff812685f0
14. __x64_sys_nanosleep ffffffff81104410

15. // Kernel code symbol information
16. ffffffff81099d30 <__x64_sys_kvuln02>
17. ffffffff81099c50 <support_kvuln02_01>
```

図 6　カーネル脆弱性 2 を利用した攻撃の追跡結果

```
1.  [*] start tracing process
2.  [*] start system call invocation
3.  [*] kernel code information
4.  // Deadlock: task list is locked
5.  a.out kprobe:__x64_sys_kvuln03 3272
6.  a.out kprobe:support_kvuln03_01 3272
7.  [*] finish system call invocation
8.  // kernel is stopped

7.  [*] virtual address rage of kernel code list
8.  __x64_sys_newfstat ffffffff8126cf10
9.  __X64_sys_kvuln03 ffffffff81099e50, ffffffff81099e70
10. support_kvuln03_01 ffffffff8109a1f0, ffffffff8109a2f3
11. __x64_sys_write ffffffff812685f0
12. __x64_sys_nanosleep ffffffff81104410

13. // Kernel code symbol information
14. ffffffff81099e50 <__x64_sys_kvuln03>
15. ffffffff8109a1f0 <support_kvuln03_01>
```

図 7　カーネル脆弱性 3 を利用した攻撃の追跡結果

攻撃時において，5 行目および 6 行目にて，独自システムコール 2 の呼出し，ならびにカーネルコード 2-1 の追跡結果が表示される．攻撃として開放済みメモリの参照を行うため，DoS 攻撃成功後，カーネルは停止する．

再起動後のカーネルにおいて，提案手法により収集したカーネルコードから仮想アドレスの特定結果を 8 行目から 12 行目にかけて表示している．9 行目，10 行目は独自システムコール 2，ならびに 11 行目，12 行目はカーネルコード 2-1 の仮想アドレスの範囲である．提案手法において，カーネルの仮想記憶空間にカーネルコードが断片化して配置された場合においても，仮想アドレスの範囲を特定できることを示している．また，16 行目，17 行目にて攻撃時のみ捕捉されるカーネルコードと仮想アドレスの組合せをカーネルより抽出している．カーネルイメージに含まれる開始仮想アドレスとの一致を確認できる．

4.2.3　カーネル脆弱性 3 を利用した攻撃の追跡

図 7 に，カーネル脆弱性 3 を介して攻撃を行うユーザプロセスの追跡結果を示す．5 行目にて，独自システムコール 3 を呼出し，6 行目にて，カーネル脆弱性 3 を含むカーネルコード 3-1 の呼出しが表示される．競合条件の発生として，カーネルのタスク管理ロックを取得したままとしており，攻撃成功後，カーネルは停止する．提案手法では，カーネルコード 3-1 の実行まで追跡し，収集可能である．

カーネル脆弱性 2 と同じく，再起動後のカーネルにおいて，提案手法により収集したカーネルコードから仮想アドレスの特定結果を 8 行目から 12 行目にかけて表示している．9 行目および 10 行目が攻撃時にのみ呼出されたカーネルコードであり，仮想アドレスの範囲を特定している．また，14 行目，15 行目に攻撃に利用したカーネルコードと仮想アドレスの組合せのカーネルイメージに含まれる開始仮想アドレスを表示している．提案手法による特定結果と一致していることを確認できる．

表 2　提案手法による攻撃ユーザプロセスの追跡結果（✓：成功）

Item	Description	Target kernel code	Approach
カーネル脆弱性 1	バッファオーバーフロー	3	✓
カーネル脆弱性 2	開放済みメモリの参照	2	✓
カーネル脆弱性 3	競合条件の発生	2	✓

5　考察

5.1　評価に対する考察

提案手法による，カーネル脆弱性を介した攻撃を行うユーザプロセス追跡機能の評価結果を表 2 に示す．提案手法は，評価に用いた 3 種類のカーネル脆弱性を利用した PoC コードそれぞれに対して，動作中のユーザプロセスの利用するカーネルコードを動的に収集し，攻撃実行時および攻撃未実行時の追跡結果からカーネル脆弱性に関連するカーネルコードの名称，および仮想アドレス範囲を特定できることを示した．

評価結果より，提案手法を利用することで，PoC コードのみを利用し，攻撃発生時のカーネル脆弱性を含むカーネルコードの動的な追跡と仮想アドレス範囲の特定し，プロファイルとして利用可能であると言える．また，評価において，特権奪取ならびに DoS 状態によるカーネル停止の発生について，カーネルおよびユーザプロセスの動作は想定通りであり，提案手法は影響を及ぼしていないことを確認した．

5.2　提案手法の考察

カーネルにおける攻撃面最小化のため，大規模な AP の利用するカーネルコードとその実装を網羅的に把握することは AP 全ての動作を実行する必要があり，困難を伴う．一方，提案手法において，評価に利用したカーネル脆弱性を利用する PoC コードは 143 行，カーネル脆弱性に関するカーネルコードは 192 行と動作の全体像を把握可能なコードサイズである．提案手法による攻撃実行時，攻撃未実行時の追跡結果から，攻撃に関するカーネルコードのみが差分として抽出されることから，攻撃面最小化のために把握対象としなければならないカーネルコードの絞込みが可能であることを示している．

5.3　先行提案との連携

我々が先行して提案しているセキュリティ機構 [4] では，カーネルの仮想記憶空間を構成するページテーブルを直接操作し，ユーザプロセスに対して予め指定したカーネルコードの実行を禁止可能とする．先の提案手法 [4] の実現方式において，カーネルコードの実行禁止対象への事前登録は，KASLR が無効化されている場合はカーネルコード名と仮想アドレス範囲の両方を登録可能である．KASLR が有効化されている場合，カーネルコード名のみ登録可能である．

本稿での提案手法を用いることで，PoC コードが存在する場合，カーネル脆弱性と想定するカーネルコード一覧の把握と仮想アドレス範囲の特定が可能となる．その

表 3　カーネルコードの追跡と特定手法の比較

機能	kRazor [2]	KASR [3]	提案方式
追跡・特定対象	AP 全体	AP 全体	PoC コード
追跡・特定実現方式	カーネルのトレーシング機能	VMM からのメモリ操作監視	カーネルのトレーシング機能
限界	AP・カーネル更新への対応	AP・カーネル更新への対応	カーネル更新への対応

ため，先の提案手法 [4] の実現方式において，本稿での提案手法を用いて得られたカーネル脆弱性に関連すると判定したカーネルコード一覧と仮想アドレス範囲を動的に追加し，必要に応じて更新することで効率的にユーザプロセスからの呼出しを制限することが可能になる．これにより，カーネル脆弱性の発見時に，迅速に攻撃面最小化を行い，カーネルへの攻撃防止を実現可能になると考えている．今後，先行研究 [2, 3] との攻撃面最小化効果の比較，ならびに実際の脆弱性に対しての有効性の評価を検討している．

5.4　限界

提案手法では，カーネル脆弱性を利用する PoC コードが入手可能な場合に適用可能であると考えている．そのため，PoC コードの入手が困難な場合，実際の攻撃に利用されるカーネルコードを特定できない可能性がある．また，カーネル脆弱性を含むものの，ユーザプログラムから利用頻度が高い一般的なカーネルコードの場合（例，プロセス生成など），提案手法により特定は可能ではあるが，利用自体を制限することはカーネル動作を維持するためには難しいといえる．

全てのユーザプロセスに対し，一律にカーネルコードの利用制限を行うだけでなく，攻撃対象となる可能性の高さから制限対象とするユーザプロセスを絞り込むことも必要であると考えている．また，カーネルへの攻撃を防ぐためカーネルコード実行時の引数情報の追跡を行い，利用制限に用いることや，従来技術である強制アクセス制御，KASLR に加えて，スタックおよびメモリ監視との組合せが重要になると考えている．

6　関連研究

仮想記憶空間の分離と保護

カーネルにおける仮想記憶空間の分離機構として，KPTI はユーザモード用のページテーブルに最低限のカーネルコードおよびカーネルデータ，残りをカーネルモード用のページテーブルに配置することで，ユーザプロセスによるサイドチャネル攻撃からカーネルモード用のページテーブル参照を保護できることを示した [6]. カーネルにおける仮想記憶空間の保護として，kR^X はカーネルコードおよびカーネルデータの読込み権限と書込み権限を排他的に管理することでカーネルへのメモリ破壊攻撃を緩和できることを示した [7].

攻撃面最小化

カーネルにおける攻撃可能な領域を削除する攻撃面最小化の手法では，事前に AP の利用するカーネル機能の調査を伴う．kRazor は，ユーザプロセス実行時に呼出されたカーネルコードを捕捉し，利用可否判断を行う [2]. KASR は，ユーザプロセス実行時に，最低限のカーネルコードとカーネルデータのみをカーネルの仮想記憶空間に配置し利用可能とする [3].

不正コード実行防止

脆弱性を利用した攻撃の防止手法として，Control Flow Integrity（CFI）はコード呼出し順を検査し，不正コードの実行を防ぐ [8]．KCoFI は割込み制御等の非同期処理を行うカーネルにおいても CFI を適用可能とした [9].

耐故障性の向上

カーネルにおける耐故障技術として，デバイスドライバと主要なカーネル機能の動作する仮想記憶空間を分離し，障害発生によるメモリ破壊や動作停止の影響最小化を行う手法がある．ユーザスペースドライバは，ユーザモードでデバイスドライバを動作させている [10, 11]. また，iKernel は仮想マシンでデバイスドライバを動作させ [12], SIDE はカーネルとデバイスドライバのページテーブルを分離する機構を提案している [13].

6.1　先行研究との比較

先行研究 [2, 3] との比較を表 3 に示す．カーネルの攻撃面最小化のため，kRazor や　KASR では，カーネルと AP 動作全体の網羅を必要としている．そのため，先行研究における AP の利用するカーネルコードの追跡と特定手法では，AP やカーネルの更新の都度，AP の動作に必要とされるカーネルコードを適切に追跡し特定していくことが求められ，大規模な AP や更新内容が多岐にわたる場合への対応は困難である．提案手法においては，カーネルへの攻撃を実現するための PoC コードは一般に最小限のプログラムコードで構成される点に着目した．PoC コードのプログラムサイズが小規模であれば，攻撃実行時，および攻撃未実行時の切替えは比較的容易に実現でき短時間の動作でプログラムコード全体の実行を網羅可能である．そのため，カーネル脆弱性を利用した攻撃の実行時と未実行時から，ユーザプログラムがカーネルへの攻撃に利用するカーネルコードのみ抽出可能であり，カーネル脆弱性に関する網羅性を担保可能であると考えている．

KCoFI では，CFI の適用のためにカーネルに対して非同期処理の呼出しを管理するため，呼出し順の維持を可能とするための独自アーキテクチャを要求している [9]. CFI は，既知及び未知の脆弱性に関わらず，不正なコード呼出し等の呼出し順序を操作する攻撃へは有効である．しかし，単一のカーネルコードにおいて，開放済みメモリ参照や競合条件の発生させる攻撃に対しては防止できない．提案手法では，PoC コードの実行からカーネルへの攻撃に利用するカーネルコードの特定が可能である．提案手法を利用して，カーネル脆弱性の種別を明らかにすることで CFI の適用可否に利用できると考えている．

7　おわりに

カーネルに対する攻撃への対策として，kRazor や KASR はユーザプロセス毎にカーネルを再構成し，利用可能なカーネルコードを制限することで攻撃面最小化を実現している．これらの手法は事前にユーザプロセスの利用するカーネルコードの情報収集を必要とし，また制御対象のカーネルコードは多岐にわたる．

我々は，ユーザプロセス毎に利用するカーネルコードを収集し，実行可否を制御するのではなく，脆弱性を含むカーネルコードの利用可否のみを制限可能とする手法を提案している [4]．しかし，制御対象とするカーネル脆弱性に対して，カーネルコードならびに仮想アドレス範囲の特定は手動で行う．

本稿では，カーネル機能の実行追跡機能とデバッグ情報を活用し，ユーザプロセスによるカーネルへの攻撃実行時，および攻撃未実行時に利用するカーネルコードの呼出し結果の違いから，カーネル脆弱性を介して攻撃を行う際に利用するカーネルコードを特定し，仮想アドレス範囲を取得する手法を提案した．

評価として，提案手法を実現した Linux において，特権奪取および DoS によるカーネル停止を発生させるカーネル脆弱性を備えたカーネルコードを意図的に導入し，ユーザプロセスからの攻撃に利用された脆弱なカーネルコードの動的な追跡と仮想アドレス範囲の特定を可能であること示した．

謝辞

本研究の一部は，JSPS 科研費 JP19H04109 の助成を受けたものです．

参考文献

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

[2] Kurmus, A., et al.: Quantifiable Run-Time Kernel Attack Surface Reduction. the 11th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), vol. 8550, pp. 212-234 (2017).

[3] Zhang, Z., et al.: KASR: a reliable and practical approach to attack surface reduction of commodity os kernels. the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), vol. 11050, pp. 691-710 (2018).

[4] 葛野 弘樹, 山内 利宏, カーネル仮想記憶空間における排他的ページ参照機構の実現方式と性能評価, 第 54 回 情報通信システムセキュリティ研究会 (ICSS), 電子情報通信学会技術研究報告, vol.120, no.384, pp.138-143, (3, 2021)

[5] Chen, H., et al.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems. In: Proceedings of the Second Asia-Pacific Workshop on Systems, pp. 1-5, (2011).

[6] Gruss, D., et al.: KASLR is Dead: Long Live KASLR. In: Proceedings of 2017 International Symposium on Engineering Secure Software and Systems (ESSoS), vol. 10379, no. 3, pp. 161-176, (2017).

[7] Pomonis, M., et al.: kR^X: comprehensive kernel protection against just-in-time code reuse. In Proceedings of the twelfth European Conference on Computer Systems, pp. 420-436 (2017).

[8] Abadi, M., et al.: Control-Flow Integrity Principles, Implementations, and Applications. In Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS), pp. 340-353 (2005).

[9] Criswell, J., et al.: KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In Proceedings of IEEE Security and Privacy, pp. 292–307 (2014)

[10] Herder, J., et al.: Fault Isolation for Device Drivers. In Proceedings of the 39th Annual IEEE / IFIP International Conference on Dependable Systems and Networks, pp. 33-42, (2009).

[11] Butt, S., et al.: Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In Proceedings of the 2009 Annual Computer Security Applications Conference, pp. 301-310, (2009).

[12] Tan, L., et al.: iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support. In Proceedings of the third IEEE International Symposium on Dependable, Autonomic and Secure Computing, pp. 134-144, (2007).

[13] Sun, S., et al.: SIDE: Isolated and efficient execution of unmodified device drivers. In Proceedings of the 43rd Annual IEEE / IFIP International Conference on Dependable Systems and Networks, pp. 1-12, (2013).