仮想デバイスへのリクエストフィルタの導入による ハイパーバイザの脆弱性回避

小笠原 純也[1,a)]　河野 健二[1]

概要：クラウド等で仮想化が一般的になり，仮想化を実現するハイパーバイザの脆弱性が注目されている．ハイパーバイザでは，仮想マシンに提供する仮想デバイスのエミュレーションを行うものの，デバイスの仕様が複雑である上，実デバイスでは回路の特性上考慮する必要のない状態まで正確にエミュレートする必要があり，しばしば脆弱性の要因となる．本論文では，デバイスの仕様に反する不正なリクエストを拒否するフィルタを提案する．このフィルタの導入により，エミュレーションの誤りに起因する脆弱性を事前に回避できる．実際，エミュレーションの脆弱性を突く VENOM と同等の攻撃を行ったところ，脆弱性があっても攻撃を防止できることを示す．

キーワード：ハイパーバイザ，仮想デバイス，エミュレーション，脆弱性，VENOM

1. はじめに

近年，クラウド等における計算資源の効率的な利用のために仮想化が一般的になっている．仮想化は複数の OS を 1 つの物理サーバー上で動かすことを可能にすることで，複数の OS 間で計算資源を分配し効率的に利用することを可能にした．また，クラウドプロバイダーにおいては物理サーバの運用台数を大幅に減らすことを可能とし，コスト削減のために広く用いられるようになっている．

仮想化の実現にはハイパーバイザというソフトウェアが用いられる．ハイパーバイザは主に物理デバイスを仮想化し，ハイパーバイザ上で動く複数のゲスト OS に対してあたかも各 OS 専用のデバイスが存在するかのように認識させる．これによりハイパーバイザ上で各 OS は他のゲスト OS を関知することなく，物理マシン上で動作している状態と同じように動作することが可能になる．

しかしながら，このハイパーバイザのデバイス仮想化における脆弱性が度々報告されている．例えば，2009 年の VMware の仮想ビデオデバイスにおける脆弱性である Cloudburst [1] や，2015 年に発見された QEMU の仮想 Floppy Disk Controller (FDC) における脆弱性の VENOM [2] である．デバイスの仮想化において脆弱性が生じてしまうのは，物理デバイスの仕様が複雑である上に，物理デバイスでは回路の特性上考慮する必要のない状態まで考慮する必要があるためである．そのため物理デバイスを正確にエミュレートするということ自体が非常に困難であり，エミュレーションにおいて仕様に沿わない部分が少しでも存在すれば，脆弱性の原因となり得る．

このような脆弱性は物理デバイスの仕様に沿わないようなリクエストによって，物理デバイスでは起こりえない挙動が生じることによって引き起こされる．一般に仕様に沿ったリクエストに対してはテストが入念に行われるが，仕様に沿わないリクエストに対しては想定外のものが必ず存在してしまい，テストのカバレッジを上げることが難しい．そのため，脆弱性による被害を防ぐためにはデバイスの仕様に沿わないようなリクエストが仮想デバイスに対して発行されないことが望ましい．

そこで，本論文ではゲスト OS から仮想デバイスへ発行されるリクエストに対するフィルタを導入することにより，ハイパーバイザのデバイス仮想化における脆弱性による被害を回避する手法を提案する．具体的には，まず物理デバイスの仕様を表すオートマトンを用意する．そしてゲスト OS と仮想デバイスの間にリクエスト監視のためのレイヤーを挿入することで，仮想デバイスへ発行されるリクエストを常にモニタリングする．予め用意したオートマトンを基に，リクエストが物理デバイスの仕様に沿わない不正なリクエストか否かを判別し，不正なリクエストであると判断した場合にはそのリクエストを拒否する．これにより，物理デバイスの仕様では本来受け付けないはずの状態

[1] 慶應義塾大学
Keio University
[a)] oga@sslab.ics.keio.ac.jp

へ仮想デバイスの状態が遷移することを妨げることが可能となる．すなわち，たとえ仮想デバイスのエミュレーションに脆弱性が存在したとしても，この新たに導入するフィルタが不正なリクエストを拒否することで脆弱性による被害を未然に防ぐことを可能にする．

本論文では実験として，本提案が不正確なデバイスエミュレーションに起因する脆弱性に対する攻撃を検知できるかをシミュレーションによって確認した．実験対象の仮想デバイスには VENOM の原因となった QEMU の FDC を用いた．まず，ゲスト OS と仮想デバイス間にレイヤーを挿入し，ゲスト OS から仮想デバイスへのリクエストログ収集した．そして，物理デバイスの仕様を表すオートマトンを作成しリクエストフィルタを実装した．このリクエストフィルタに対して取得したログを入力として与え，本提案で導入するフィルタの有効性を確かめることを行った．結果として正常系のログは全てフィルタを通過した一方で， 異常系のログはフィルタに検知されることが確認された．なお，異常系のログには VENOM 脆弱性の Proof of Concept (PoC) コードをゲスト OS にて実行した際に収集されたものを用いた．

なお，本論文ではハイパーバイザとして Linux KVM + QEMU を対象としている．

本論文の構成を以下に示す．2 章では背景として仮想化技術について述べる．3 章では仮想化環境における脆弱性について説明する．4 章では本研究が提案する手法について述べる．5 章では VENOM に対する本提案の適用について述べる．6 章では提案手法のフィルタに正常系及び異常系のログを入力として与えた実験の評価について述べる．7 章では関連研究を紹介する．8 章ではまとめを述べる．

2. 仮想化技術

1 章で述べたとおり，仮想化の実現にはハイパーバイザというソフトウェアが用いられる．本章ではまず，Intel CPU による仮想化支援技術 Intel Virtualization Technology [3] の 1 つである Intel VT-x [4] を説明し，次に本論文で対象とするハイパーバイザの Linux KVM [5] 及びデバイスエミュレーションを担う QEMU [6] について述べる．なお，仮想化技術に対して理解がある場合，本章は読み飛ばして頂いて構わない．

2.1 Intel VT-x

Intel VT-x は x86/x64 CPU における仮想化をハードウェア的に支援する技術である [*1]．Intel VT-x は x86/x64 CPU に対して既存のリングプロテクションによる ring0 から ring3 までの動作モードに加え，新たに VMX root mode 及び VMX non-root mode という 2 種類の動作モードを導入した．

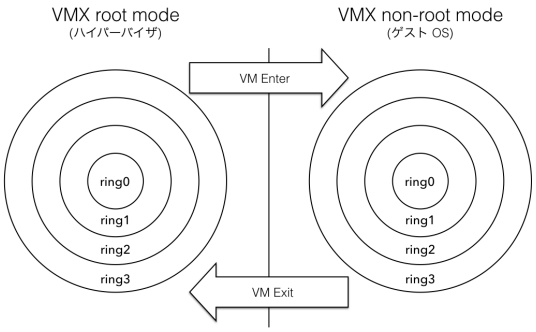

図 1 Intexl VT-x の VMX 動作モード

この新たな 2 つの動作モード導入された背景に，Popek と Goldberg の仮想化要件 [7] がある．この仮想化要件の内の 1 つに

> A virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.
> （全てのセンシティブ命令セットが特権命令セットのサブセットであれば，Virtual Machine Monitor (VMM)[*2] を構築することができる．）

というものがある．これはシステム資源に影響を及ぼす，または依存する命令（センシティブ命令）が ring0 でない，すなわちユーザモードで実行される際には命令がトラップされ，ring0 へ制御が移らなければなければならないということを意味する．ゲスト OS の発行するセンシティブ命令をハイパーバイザがトラップすることができるようになるため，システム資源を管理できるようになる．

しかしながら，x86/x64 の Instruction Set Architecture (ISA) では一部のセンシティブ命令が非特権命令であるため，この仮想化要件を満たすことができなかった．そのため， VMware では Binary Translation によって動的にセンシティブ命令を書き換える事で対応が行われた [8] が，ベアメタル環境に比べて非常にオーバーヘッドが大きかった．また，Xen ではセンシティブ命令の際にはハイパーバイザへ処理が移るよう，ハイパーバイザ用にゲスト OS を書き換える準仮想化という手法によって対応が行われた．しかし，ゲスト OS を書き換えなければならず， Windows 等には適用できない手法であった．

このような問題をハードウェア的に解決するために，Intel VT-x が登場した．**図 1** に示すように，従来のリングプロテクションは保ったまま，リングプロテクションとは全く異なる VMX root mode 及び VMX non-root mode という 2 種の動作モードをハードウェアレベルで導入した．仮想化を行う際には VMX root mode でハイパーバイザが動作し，VMX non-root mode でゲスト OS が動作す

[*1] AMD の CPU においても同様の仮想化支援機能があり，AMD-V と呼ばれる

[*2] ハイパーバイザは Virtual Machine Monitor (VMM) とも呼ばれる

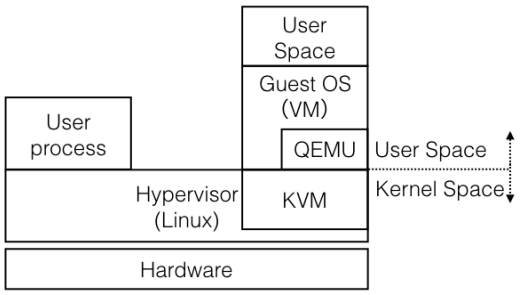

図 2 Linux KVM の仮想化概略図

ることになる．これにより，ハイパーバイザがゲスト OS のコードを実行する際には VM Entry を発生させ，VMX root mode から VMX non-root mode へモードが切り替わることでゲスト OS のコードが CPU 上で直接実行される．一方， ゲスト OS によるセンシティブ命令の発行等によりハイパーバイザ側での処理が必要となった場合，CPU は VM Exit を発生させ VMX non-root mode から VMX root mode へ切り替わる．これにより，ハイパーバイザがゲスト OS によるセンシティブ命令をトラップ可能となり，Popek と Goldberg の仮想化要件を満たすことができるようになる．Intel VT-x によりゲスト OS に変更を加えることなしに高いパフォーマンスを保ったままの仮想化が可能になった．

他にも CPU による仮想化支援として Extended Page Table (EPT) 等も Intel VT-x の機能に含まれるが本論文では割愛する．

2.2 Linux KVM + QEMU

Linux KVM は 2.1 で述べた Intel VT-x による支援を前提として作成されたハイパーバイザであり，標準で Linux Kernel へ組み込まれているのが特徴である．Google Compute Engine (GCE) 等のパブリッククラウドでも広く利用されている [9] ．**図 2** に Linux KVM による仮想化の概略図を示した．図にあるように，Linux KVM は次の 2 つのソフトウェアから構成される．KVM カーネルモジュールと QEMU である．

KVM カーネルモジュールは，Intel VT-x や AMD-V の機能を用いて仮想マシンを作成し，仮想マシンのコードを CPU で実行する制御や，メモリの仮想化といった役割を担う．VM Entry 等のための vmlaunch 命令や vmresume 命令は特権命令であり，かつセンシティブ命令であるのでカーネルモジュールとして実装されている．

一方， QEMU はユーザ空間で動作し，I/O デバイスのエミュレーションを担う．QEMU は元々，フルシステムエミュレーションを行うためのソフトウェアとして開発され，広く利用されてきた．そのため，多数のデバイスのエミュレーションをサポートしており，また多数の開発者によってメンテナンスが行われている．KVM はこれを利用し，デバイスエミュレーション部分は全て QEMU に任せることにした．Xen や VirtualBox といった他の仮想化ソフトウェアも同様に QEMU をデバイスエミュレーションのために利用している [10], [11].

Linux KVM + QEMU が I/O 命令のようなセンシティブ命令が発行された際に，どのようにしてハードウェアエミュレーションを行うのかを説明する．ゲスト OS のコードは 2.1 にて説明した通り VMX non-root mode にて動作している．VMX non-root mode の中で I/O 命令のようなセンシティブ命令が発行されると，CPU は VM Exit を発生させ，VMX root mode へ CPU の動作モードが遷移する．それに伴ってハイパーバイザへコンテキストスイッチが生じ，VM Exit の際にハイパーバイザへ Exit Reason という VM Exit が生じた原因を表す情報が伝えられる．ハイパーバイザはこの Exit Reason から I/O 命令のエミュレーションを行わなければならないと検知する．すると，ハイパーバイザは QEMU へと制御を移し，デバイスエミュレーションを行う．そして，デバイスエミュレーション終了後，ハイパーバイザは VM Enter を行いゲスト OS の動作を継続させる．これを繰り返すことで，Linux KVM + QEMU はハードウェアの仮想化を行っている．

3. 仮想化環境における脆弱性

1 章にて，ハイパーバイザにおける脆弱性が度々報告されていることを述べた．ハイパーバイザの脆弱性を用いた攻撃は時に深刻な被害を及ぼす．例えば， Deninal of Service (DoS) 攻撃や VM Escape といった攻撃である．特に VM Escape はハイパーバイザ特有の攻撃手法である．VM Escape はゲスト OS がハイパーバイザの脆弱性を利用することで，ハイパーバイザ空間でのコード実行を可能とする攻撃である．これにより悪意のある攻撃者がゲスト OS からハイパーバイザ空間へ脱出し，ハイパーバイザーの権限を得ることができてしまう．攻撃者は VM Escape を行うことで，同一ハイパーバイザ上で動くゲスト OS のメモリや暗号化されていないゲストのストレージ等の情報を覗くことができるようになる．

パブリッククラウドにおいて，こういった攻撃による被害は甚大である．パブリッククラウド上では様々な顧客の仮想マシンが同一のハイパーバイザ上で動作している．そのため，悪意のあるゲスト OS が DoS 攻撃を行うことで，通常ユーザの仮想マシンの正常な動作を妨げてしまう．また，VM Escape 攻撃を行うことでハイパーバイザを乗っ取り，同一ハイパーバイザ上で動く他の顧客の仮想マシンに対する全コントロールを取得することで，顧客の機密情報が漏洩する可能性がある．

このような脆弱性による被害を防ぐ解決策として最も望ましいのは，ハイパーバイザに脆弱性が全くない状態にすることで脆弱性による被害が生じ得ない状況にすることで

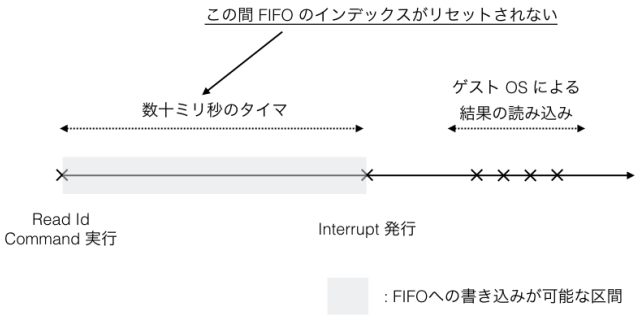

図 3 FDC の Read Id Command 実行の流れ

ある．しかしながら，脆弱性が全くない状態にするのは非常に困難である．これを裏付けるように，今日に至るまでハイパーバイザに関する脆弱性のレポートは継続的に上がっている．そのため，脆弱性の存在を全く許さない方針でハイパーバイザのセキュリティレベルを上げるのではなく，脆弱性があったとしても深刻な被害を引き起こさないようにすることが求められる．

本論文ではハイパーバイザにおける深刻な脆弱性の例として 2015 年に話題となった VENOM を紹介する．

3.1 VENOM

2015 年に QEMU の 仮想 FDC のエミュレーション部分に脆弱性が発見され，VENOM と名付けられた．VENOM は VM Escape を引き起こしうる脆弱性であり，VENOM による攻撃が成功すると最悪の場合，ハイパーバイザの制御が攻撃者によって乗っ取られる恐れのあるものだった．その結果，他のゲスト OS の情報等が漏洩する可能性があったことから，脆弱性の評価指標である Common Vulnerability Scoring System (CVSS) の Base Score は 7.7 の「危険」，攻撃が成功した場合の影響度を示す Impact Subscore は最大値の 10.0 [12] と評価され話題となった．このように，ハイパーバイザのデバイスエミュレーションにおける脆弱性はセキュリティに対する影響度が非常に大きいことが分かる．

VENOM の原因について説明する．VENOM の直接の原因は，FIFO バッファへの書き込み時に配列のインデックスに対する範囲チェックが不完全であったことである．そのため FDC の制御コマンドである Read Id Command 及び Drive Specification Command 実行中の特定条件下で FIFO のサイズを超えた書き込みが行われてしまい，バッファオーバーフローが生じてしまうことがあった．これにより，QEMU の動作しているハイパーバイザ空間にてゲスト OS が任意のコードを実行可能となり，VM Escape が生じる可能性があった．

VENOM の Read Id Command における特定条件について説明する．Read Id Command は Floppy Disk Drive (FDD) の現在の磁気ヘッドの位置を取得するコマンドである．FDC および FDD は物理デバイスであるので，実行に CPU 時間と比較して大きな時間がかかる．そのため QEMU における FDC の Read Id Command のエミュレーションでは，この遅延をエミュレートするために**図 3** に示すように結果を返すまでに数十ミリ秒のタイマを設定している．このタイマの時間経過後，FDC は Interrupt を発行してゲスト OS へ処理が完了したことが通知し，ゲスト OS は Read Id Command の実行結果の読み込みを行う．

バッファオーバーフローが生じた直接の原因は，この条件下で図 3 に示すように数十ミリ秒のタイマの間，FIFO のインデックスがリセットされないことである．これにより，範囲外への書き込みが生じてしまった．

しかし根本的な原因は，そもそもタイマ実行中に FIFO への書き込みが行われていた点にある．物理デバイスの仕様上，FDC は Read Id Command の実行中に FIFO への書き込みを許可していない．しかし，数十ミリ秒のタイマ実行中に FIFO への書き込みが可能な状態にあってはならないにも関わらず，QEMU の仮想 FDC では書き込みが可能となっていた．これは QEMU の仮想 FDC のエミュレーションにおける FDC の状態管理に不備があったことを意味する．つまり，本来電気的には受け付けられないはずの状態にも関わらず，その状態を正しく管理できていなかったために FIFO への不正な書き込みを引き起こすこととなったのが VENOM の根本原因である．

4. 提案

本論文ではゲスト OS から仮想デバイスへのリクエストに対するフィルタを導入することにより，物理デバイスの仮想化に起因する脆弱性による被害の回避手法を提案する．本提案ではゲスト OS から仮想デバイスへのリクエストが物理デバイスの仕様に沿っているかの判断を行うために，物理デバイスの仕様を表すオートマトンを作成し用いる．本章では，本提案がデバイスの仕様に基いてどのようなオートマトンを用いるかについて述べる．

物理デバイスは動作する上で現在の状態を持っている．各状態は受け入れ可能な OS からの入力や OS に要求する操作が仕様上決まっている．物理デバイスは OS から操作があると，現在の状態と操作内容に応じて処理の内容を決定し，処理を行った後に現在の状態と操作に対応する次の状態へ遷移を行う．物理デバイスを正しく利用するには，物理デバイスの正規の利用者はこの状態を正しく把握し操作を行う必要がある．そのため，正規のユーザや正しく作られたドライバはこの仕様に沿った操作を行う．

本提案では物理デバイスの仕様を基に物理デバイスの各状態において受け入れ可能なリクエストを予め定義し，各状態において受け入れ可能なリクエストが発行された場合の次の状態も定義する．これにより，**図 4** に示すようなゲ

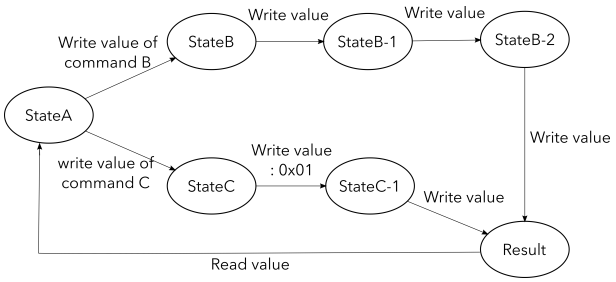

図 4 本提案におけるオートマトンの例

スト OS からのリクエストによって遷移する物理デバイスの仕様を表すオートマトンを記述することができる．このオートマトンを利用することで本提案におけるリクエストフィルタを実現した．図 4 を用いて具体的に説明する．

図 4 において State A は物理デバイスがアイドルな状態である．State A では Command B 及び Command C の値の書き込みが仕様上定められており，それ以外の値や操作は定められていない．このため，State A においてリクエストフィルタを通過するようなリクエストは，

$$\text{書き込み} \wedge \text{入力値が } (CommandB \vee CommandC)$$

を満たすものに限られる．State A において上記のリクエストが発行された場合，フィルタは通過し次の状態に遷移する．一方，読み込みのリクエストや，Command B でも Command C の値でもない書き込みが行われた場合，そのようなリクエストはフィルタを通過せず，不正なものであるとして拒否される．

同様にして，State B においては書き込みのみが許可されているため，読み込みのリクエストが発行されれば不正なリクエストとして拒否される．Result では読み込みのみが許可されているため，書き込みのリクエストが発行されれば不正なリクエストとして拒否される具合である．

このように，ゲスト OS から仮想デバイスのリクエストをオートマトンを用いて検証し，不正なリクエストを検知して拒否することでデバイスエミュレーションの脆弱性による被害を未然に防ぐことを可能とする．具体的にどのようなオートマトンを作成するのかについては，5 章にて FDC に対する例を紹介する．

5. VENOM に対する対策の適用

本論文では本提案を実際に Linux KVM + QEMU へ組み込む前段として，ハイパーバイザの脆弱性回避に対する本提案の有効性を確認するためにリクエストフィルタのシミュレーションを行った．対象の仮想デバイスとして，3.1 にて紹介した VENOM の原因となった QEMU の仮想 FDC を用いた．そしてゲスト OS 内でフロッピーディスクへ正常なリクエストを行った場合と，VENOM の PoC を実行した場合の 2 つの条件で実行結果の比較を行った．実行結果の比較については 6 章にて述べる．

本章では対象である FDC のオートマトンの仕様及び仮想デバイスへのリクエストログをどのように取得したかについて述べる．

5.1 FDC のオートマトン

QEMU の FDC は Intel 82078 という IC をエミュレーションしている．そこで，本論文ではオートマトンを Intel 82078 の仕様書 [13] に基いて作成した．

まず最初に，Intel 82078 の仕様について概要を説明する．Intel 82078 は OS とのインタフェースとして FIFO バッファを持っており，OS が FIFO への読み書きを行うことで FDC の制御を行う．OS はこの Intel 82078 の FIFO に読み書きを行う際，Main Status Register (MSR) というレジスタを参照することによって FIFO が読み書き可能かを確認しなくてはならない．もし，FIFO が読み書き可能でない場合は読み書きが可能になるまで待たなければならない．

Intel 82078 はこの FIFO への入出力に応じて動作する 29 のコマンドを持っており，各コマンドは次の 3 つのフェーズに分かれて動作する．OS からコマンドの受け入れを行う Command Phase，受け入れたコマンドに応じた実行を行う Execution Phase，OS へコマンドの実行結果を返す Result Phase である．

コマンド待ち状態で OS から FIFO への入力が行われると，入力値が仕様上存在するコマンドの場合 Command Phase へ遷移する．Command Phase 内では Command 毎に必要なパラメータを OS から受け取り，パラメータが充足されると Execution Phase へ遷移する．Execution Phase において必要な処理が完了すると，FDC は Interrupt を発生させ，Result Phase へ遷移する．Result Phase では OS が結果を FIFO から読み込むことを行い，全ての結果の読み込みが完了すると再びアイドル状態へ遷移するという流れである．これら各遷移においても OS は前述の MSR を読み，読み書き不可能な場合は適切に待つ必要がある．

このコマンドの仕様の例として，Intel 82078 のコマンドの 1 つである READ ID コマンドの仕様を仕様書 [13] より抜粋し，**表 1** に示す．Data Bus は FIFO への入出力線である．表 1 は READ ID コマンドの各フェーズにおいてどのような操作が受け入れ可能かを示している．Data Bus の行において，0 または 1 ではなく文字列が入っている箇所は 0 と 1 のどちらも取りうることを表現している．(文字列の詳細な意味は仕様書を参照のこと．) 例えば，READ ID コマンドを実行する際の最初の値は 0x0a (=0b00001010) または 0x4a (=0b01001010) のどちらかでなくてはならないことが仕様によって定義されていることが分かる．

表 1 から，READ ID コマンドに対するオートマトンを**図 5** のように表現できる．仕様に沿った入出力が行われ

表 1　Intel 82078 における READ ID コマンド仕様

Phase	R/W	Data Bus D7	D6	D5	D4	D3	D2	D1	D0
Command	W	0	MFM	0	0	1	0	1	0
	W	0	0	0	0	0	HDS	DS1	DS0
Execution									
Result	R	———				ST0			———
	R	———				ST1			———
	R	———				ST2			———
	R	———				C			———
	R	———				H			———
	R	———				R			———
	R	———				N			———

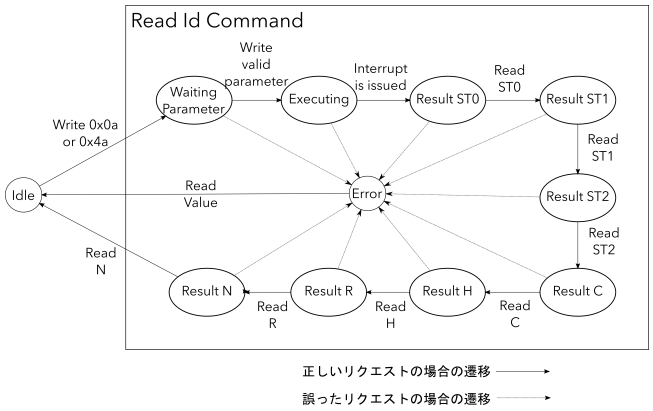

図 5　Intel 82078 における Read Id コマンドのオートマトン

た場合には次の状態へ遷移し，仕様に沿わない誤った入出力が行われた場合には Error 状態へ遷移する．Error 状態では仕様上 FIFO にエラー原因が出力されており，これを OS は読み込まなければならない．それにも関わらず，書き込み等が行われた場合には不正なリクエストとして判断し，拒否することを行う．また，各状態において FIFO への入出力のリクエストが発行された際には常に MSR を確認し，書き込み不可にも関わらず書き込みが行われた場合や，読み込み不可にも関わらず読み込みが行われた場合はオートマトンの状態遷移に含まれないためリクエストを拒否する．

このようなオートマトンを 29 全てのコマンドに対して作成し，ログの入力に伴ってオートマトンの状態遷移を行い，不正なログが入力された場合には拒否するプログラムを実装した．評価にはこのプログラムを用いた．

5.2　仮想デバイスへのリクエストログの取得

ゲスト OS から仮想デバイスへのリクエストログの取得は，QEMU にフックを追加することにより実現した．

OS とデバイスのやり取りは I/O ポートを通じて I/O 命令を用いて行う方法や，MMIO を通じてメモリアクセス命令を用いて行う方法がある．2.2 にて述べたように Intel VT-x 環境においては，ゲスト OS から I/O 命令や MMIO 領域に対するメモリアクセス命令が発行されると，CPU は VM Exit を発生させ，VMX non-root mode から VMX root mode へ移りハイパーバイザへ処理が渡る．

本論文で対象とする Linux KVM + QEMU の場合には，VM Exit が生じた際に KVM はデバイスエミュレーションの必要性を判断し，必要なければ VM Exit に対する適切な処理を行った後に再び VM Entry を行う．一方，必要であれば QEMU へ処理が渡り， QEMU がデバイスエミュレーションを行い， QEMU がその結果を KVM へ渡し，再び VM Entry することで結果がゲスト OS へ通知される．

そこで，本論文ではゲスト OS から仮想デバイスへのリクエストを取得するために，QEMU の KVM から QEMU へ処理が渡ってくる箇所に着目した．FDC には Port-mapped I/O が用いられ，I/O ポートの 0x3F0 ~ 0x3F7 （0x3F6 を除く）に対して I/O 命令を発行することでやりとりが行われる (FDD が 1 つの場合)．そのため，Exit Reason が 5 (= I/O 命令による VM Exit) かつ，アクセス先ポート番号が 0x3F0 ~ 0x3F7 (0x3F6 を除く) に含まれる場合，FDC に対するアクセスであると判断し次の項目をログとして出力した．

(1) Read 又は Write アクセスのどちらであるか
(2) アクセス先ポート番号
(3) 入出力値

また，5.1 で述べたように，物理デバイスから OS への通知には割り込みが用いられるため，仮想デバイスからの割り込みのログも取得する必要がある．FDC の IRQ 番号は 6 であるので，QEMU が KVM に対して割り込みを入れる箇所に IRQ 番号が 6 の場合にのみログを出力するフックを追加した．

6.　評価

5 章で説明した実験についての評価結果を示す．まず評価環境について説明し，続いて正常系及び異常系のログに対する本提案の結果を示す．

```
#include <sys/io.h>
#define FIFO 0x3f5
#define READ_ID 0x0a
#define DUMMY_VAL 0x42

int main() {
    iopl(3); /* Change I/O privilege level */

    outb(READ_ID, FIFO); /* READ_ID Command */
    for (int i = 0; i < 10000000; i++) {
        /* write to FIFO*/
        outb(DUMMY_VAL, FIFO);
    }
}
```

図 6　VENOM の PoC

6.1　評価環境

実験サーバには Intel Xeon E5-2680 2.70GHz 32 コア，メモリ 64GB のものを用いた．ホスト OS，ゲスト OS 共に Ubuntu 16.04 LTS を用いた．KVM には Ubuntu 16.04 LTS のデフォルトのカーネルバージョンである Linux 4.4.0 のものを用い，QEMU には VENOM 脆弱性が発見され，修正パッチが当てられる前のバージョンである 2.3.0 を用いた．ゲスト OS へは 1VCPU, メモリ 4GB の割当てを行った．

6.2　正常系のログ

正常系としてゲスト OS を起動し，Floppy Disk のマウント先へファイル操作を行った後にゲスト OS をシャットダウンするという操作を行った．具体的にログの取得の為に行った操作を次に示す．

(1) ゲスト OS の起動
(2) Floppy Disk のマウント
(3) 512KB サイズのファイルの作成 & 512KB の追記
(4) /tmp 以下へ作成したファイルのコピー
(5) 作成したファイルの削除
(6) Floppy Disk のアンマウント
(7) ゲスト OS のシャットダウン

この操作により，計 5,071 行のログを得た．この 5,071 行のログを本提案で実装したリクエストフィルタのシミュレータに入力したところ，全てのログが正しく受け入れられることが確認できた．

6.3　異常系のログ

5 章で触れたように，異常系には VENOM の PoC コードを用いた．実際に実験に用いた PoC コードを図 6 に示す．この PoC コードは oss-security のメーリングリスト [14] にて紹介されているものにコメント等少し手を加えたものである．

具体的にログの取得の為に行った操作を次に示す．

(1) ゲスト OS の起動
(2) PoC コードの実行
(3) （ゲスト OS がクラッシュ）

この操作により，計 211,035 行のログを得た．この 211,035 行のログを本提案で実装したリクエストフィルタのシミュレータに入力したところ，VENOM の PoC の実行部分で拒否されることが確認できた．

3.1 で述べたとおり VENOM は Read Id Command の入力後，短時間の間に大量の値を FIFO へ書き込む．今回の PoC では Read Id Command を発行後，0x42 を FIFO へ書き込み続けている．5.1 で述べたように，Read Id Command 入力後の FDC の状態はパラメータの入力待ち状態で，受け入れ可能な値は 0x0a もしくは 0x4a である．しかし，今回の PoC の入力である 0x42 はこれに含まれない．そのため拒否された．たとえパラメータの入力において受け入れ可能な 0x4a を入力し，その後任意のペイロードを FDC へ入力しても拒否される．なぜなら，Read Id Command の Execution Phase においては FIFO の書き込みは許可されておらず，FIFO への書き込みを行った時点で仕様にないリクエストと判断されるためである．そのため，任意の攻撃コードを攻撃者が書き込むことは出来ない．

7.　関連研究

ハイパーバイザのセキュリティを高める研究は数多くなされている．

Virtual CPU Validation [15] は，Intel の物理 CPU に対するテストツールを用いて KVM の仮想 CPU をテストすることで，仮想 CPU が物理 CPU の仕様に正しく沿っているかを調査している．これにより，物理 CPU の仕様に沿わないことで生じる仮想 CPU の脆弱性を数多く発見することに成功している．本論文では脆弱性が存在したとしても，その脆弱性による被害を防ぐことのできる手法を提案している．

また，HyperSafe [16] や HyperVerify [17] はハイパーバイザの実行時に攻撃者がエクスプロイトを行うことを困難にするハードニングを行っている．HyperSafe は Control-Flow Integrity を，HyperVerify は Non-Control Data Integrity を保つための機構をハイパーバイザへ組み込むことでハードニングを可能にすると提案している．本論文では薄いフィルタリングレイヤーを挟むことで，既存のハイパーバイザへの変更が非常に小さいにも関わらずハードニングが可能である手法を提案している．

ハイパーバイザのコードサイズを最小限にすることで，Trusted Computing Base (TCB) に対する Attack Surface を最小限にすることや，TCB の形式的検証を可能にすることを目的とした研究も行われている．NOVA [18] は マ

イクロカーネルアーキテクチャを基に，ハイパーバイザの機能を分解する．そして機能ごとに microhypervisor としてホスト OS のゲスト空間へ分離することで，ホスト OS のカーネル空間で動作する TCB のコードサイズを最小限に抑える試みを行っている．また，NoHype [19] はゲスト OS の起動時にのみハイパーバイザによってリソース割り当てを行い，実行中はハイパーバイザを介在させずにゲスト OS が直接ハードウェアにアクセスすることで，実行時の TCB のコードサイズをほぼゼロにする手法を提案している．本論文では，既存の広く用いられているハイパーバイザのアーキテクチャを大きく変更する必要なしに，どのようにセキュリティを向上させるかについて着目し，提案を行っている．

8. まとめ

近年，クラウド等における計算資源の効率的な利用のために仮想化の利用が一般的になっている．そのような中で，仮想化を支えるソフトウェアであるハイパーバイザにおける脆弱性が度々報告され，脆弱性が利用された場合に起こり得る被害が非常に大きいことがわかっている．本論文ではハイパーバイザのデバイスエミュレーションにおける脆弱性が存在したとしても，物理デバイスの仕様に沿わないリクエストを拒否するリクエストフィルタを導入することでその脆弱性による被害が生じないような回避手法を提案した．実験では，本提案のリクエストフィルタに対するシミュレーションを行った．結果として，QEMU の仮想 FDC における脆弱性である VENOM の PoC が正しく拒否され，ハイパーバイザのデバイスエミュレーションにおける脆弱性に対して本提案のリクエストフィルタが十分に有効であることが確認された．本論文により，本提案のデバイスエミュレーションの脆弱性に対する有効性が確認できたため，今後は実際に Linux KVM + QEMU へ実装を行う．本論文では FDC のみを対象としていたが，NIC 等にも対象を広げていくとともに，作成したオートマトンから未知の脆弱性を発見するための攻撃コードを自動生成することも試みる予定である．

参考文献

[1] Kortchinsky, K.: Cloudburst, https://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf.

[2] CrowdStrike: VENOM Vulnerability, http://venom.crowdstrike.com/.

[3] Intel: Intel Virtualization Technology(Intel VT), http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[4] Dong, Y., Li, S., Mallick, A., Nakajima, J., Tian, K., Xu, X., Yang, F., Yu, W. and Yaozu DongShaofan LiAsit MallickJun NakajimaKun TianXuefei XuFred YangWilfred Yu: Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel® Virtualization Technology, *Intel Technology Journal*, Vol. 10, No. 3, p. 193 (online), DOI: 10.1535/itj.1003 (2006).

[5] Kivity, A., Lublin, U., Liguori, A., Kamay, Y. and Laor, D.: kvm: the Linux virtual machine monitor, *Proceedings of the Linux Symposium*, Vol. 1, pp. 225–230 (online), DOI: 10.1186/gb-2008-9-1-r8 (2007).

[6] Bellard, F.: QEMU , a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference. Proceedings of the 2005 Conference on*, pp. 41–46 (2005).

[7] Popek, G. J. and Goldberg, R. P.: Formal requirements for virtualizable third generation architectures, *Communications of the ACM*, Vol. 17, No. 7, pp. 412–421 (online), DOI: 10.1145/361011.361073 (1974).

[8] Ro, J.: VMWare and CPU Virtualization Technology, http://download3.vmware.com/vmworld/2005/pac346.pdf.

[9] Google: Google Compute Engine FAQ, https://cloud.google.com/compute/docs/faq.

[10] Xen Project: Hvmloader - Xen, http://wiki.xenproject.org/wiki/Hvmloader#qemu.

[11] ORACLE: Developer - Oracle VM VirtualBox, https://www.virtualbox.org/wiki/Developer_FAQ.

[12] NIST (National Institute of Standards and Technology): National Vulnerability Database, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456.

[13] Intel: 82078 CHMOS SINGLE-CHIP FLOPPY DISK CONTROLLER, http://download.intel.com/design/archives/periphrl/docs/29047403.pdf.

[14] oss security: 'Re: [oss-security] VENOM - CVE-2015-3456' - MARC, https://marc.info/?l=oss-security&m=143155206320935&w=2.

[15] Amit, N., Tsafrir, D., Schuster, A., Ayoub, A. and Shlomo, E.: Virtual CPU validation, *SOSP*, pp. 311–327 (online), DOI: 10.1145/2815400.2815420 (2015).

[16] Wang, Z. and Jiang, X.: HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity, *Proceedings - IEEE Symposium on Security and Privacy*, IEEE, pp. 380–395 (online), DOI: 10.1109/SP.2010.30 (2010).

[17] Ding, B., He, Y., Wu, Y. and Lin, Y.: HyperVerify: A VM-assisted architecture for monitoring hypervisor non-control data, *Proceedings - 7th International Conference on Software Security and Reliability Companion, SERE-C 2013*, IEEE, pp. 26–35 (online), DOI: 10.1109/SERE-C.2013.20 (2013).

[18] Steinberg, U. and Kauer, B.: Nova : A Microhypervisor-Based Secure Virtualization Architecture, *Proceedings of the 5th European conference on Computer systems - EuroSys '10*, p. 209 (online), DOI: 10.1145/1755913.1755935 (2010).

[19] Szefer, J., Keller, E., Lee, R. B. and Rexford, J.: Eliminating the Hypervisor Attack Surface for a More Secure Cloud Categories and Subject Descriptors, *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS'11)*, pp. 401–412 (online), DOI: 10.1145/2046707.2046754 (2011).