Vol.2019-OS-147 No.8
2019/7/25

バイナリ解析に基づく仮想デバイスへの不正 I/O 要求のフィルタリング

庄司 豊[1]　石黒 健太[1]　河野 健二[1]

概要：ハイパーバイザはマルチテナント型のクラウド環境において, テナント間の保護を実現するための基盤となっている. しかしながら, Xen や KVM などのハイパーバイザには多くの脆弱性が指摘されており, 特に仮想デバイスのエミュレーションが脆弱性の温床となっている. これは, デバイス・エミュレーションの複雑さおよび検証の煩雑さに起因する. National Vulnerability Database (NVD) [1] によれば, 2013 年から 2018 年までに報告された危険度の高い QEMU[2] の 危険性 (CVSS v2 スコア 7.0 以上) 51 件のうち, 35 件 (約 69%) がエミュレーションに関するものである. 本論文では, 仮想デバイスの脆弱性を突く攻撃の多くがデバイスの仕様にそぐわない I/O 要求であることに着目し, そのような I/O 要求をフィルタリングする仕組みを提案する. 不正 I/O 要求は, ゲスト環境で動作するデバイス・ドライバ が生成する I/O 要求とは, その発行順序, I/O 要求の引数などが異なっている. デバイスドライバをバイナリ解析することで, I/O 要求のいわば “正規” の発行順序および引数を抽出し, それに反する I/O 要求をフィルタリングする. 既存のデバイス・ドライバに対して本方式を適用し, 不正 I/O 要求を排除できる精度のフィルタが生成できることを確認する.

キーワード：仮想デバイス, デバイスドライバ, 脆弱性, バイナリ解析

1. はじめに

ハイパーバイザは仮想化技術を用いるクラウド環境において, テナント間の保護のための基盤となっている. ハイパーバイザは物理ハードウェアと仮想マシン (VM) の間に存在し, 物理ハードウェアを多重化・仮想化して, VM から発行される物理ハードウェアへのアクセスを調停する. これによって, VM が物理ハードウェアを自らが占有する前提で動作する仮想化環境において, 物理ハードウェアへのアクセス競合の阻止や一貫性 (consistency) の維持が実現できる. 同時に, ハイパーバイザは VM の分離 (isolation) も実現する. VM の分離とは, VM が他の VM から独立していることである. これによって, 物理的には同一のマシン上に存在する VM であっても, お互いのメモリにアクセスするなどの相互干渉が不可能になる.

ハイパーバイザには脆弱性が数多く存在する. National Vulnerability Database (NVD) [1] によれば, KVM+QEMU[2], [3] について, 2013 年から 2018 年までに報告された脆弱性は 232 件である. また, Xen[4] には 239 件の脆弱性が見つかっている. これらのうち, KVM+QEMU では 51 件 (21%) の脆弱性が, Xen では 47 件 (19%) が CVSS v2 スコアが 7 以上の特に深刻な脆弱性である. CVSS スコアとは, 脆弱性の深刻度を 10 段階で定量的に表した数値であり, 数値が 10 に向けて大きくなるほど, 深刻な脆弱性であることを示す [5].

この事実はマルチテナント型のクラウド環境では深刻な問題である. 脆弱性の悪用によって, VM Escape が引き起こされる可能性があるためである. VM Escape とは, VM の分離が回避され, ホスト上で任意のコードが実行可能となることである. これによって, 攻撃者はハイパーバイザや同一ホスト上の他の VM を攻撃できてしまう.

ハイパーバイザの脆弱性の多くがデバイスエミュレーションに起因する. KVM+QEMU では 51 件の深刻な脆弱性のうち, 35 件 (約 69%) がデバイスエミュレーションに起因する. 例えば, VENOM (CVE-2015-3456) という脆弱性がある. この脆弱性はフロッピーディスクコントローラ (Floppy Disk Controller, FDC) のエミュレーションに関するもので, CVSS v2 スコアが 7.7 と危険度が高い. この脆弱性を悪用することによって VM Escape を引き起こし, ハイパーバイザの管理者権限や任意のコード実行が可能となる [6].

本論文では, このようなデバイスエミュレーションに関

[1] 慶應義塾大学
Keio University

係する脆弱性の悪用からハイパーバイザを保護することを目標とする．そのために, 不正な I/O 要求のフィルタリング機構を提案する．この機構は VM とハイパーバイザの間に設置され, VM からの不正な I/O 要求をフィルタリングする．

仮想デバイスに対する不正攻撃の多くは, ゲスト環境で動作するデバイスドライバが発行する I/O リクエストのシークエンスとは異なったものであることが多い．これは, 通常のデバイスドライバが生成する I/O シークエンスに対しては, 仮想デバイスの動作検証も十分に行われていることが多いからである．そのため, 仮想デバイスの脆弱性の多くは, 通常のデバイスドライバが生成することのないシークエンスとなっていることが多い．実際, VENOM ではフロッピーディスクドライブ (FDD) へ発行されたコマンドが処理されている最中に, バッファにアクセスするというシークエンスが用いられる．この I/O シーケンスは通常のデバイスドライバが生成することはない．なぜなら, コマンドが処理されている最中のバッファへのアクセスは, ハードウェアレベルで禁止されているためである．

本論文で提案する I/O リクエスト・フィルタでは, ゲスト環境で動作するデバイスドライバの生成しうる I/O リクエストのシークエンスは不正ではないとみなし, ゲスト環境のデバイスドライバが生成し得ない I/O リクエストのシークエンスは不正であるとみなす．ここでいう I/O リクエストのシークエンスとは, デバイスレジスタに対するアクセス順序, およびその際に読み書きされる値のことを指す．I/O リクエスト・フィルタでは, デバイスドライバ毎にそのデバイスドライバが生成しうる I/O リクエストのシークエンスを保持し, そのシークエンスに一致しない I/O リクエストをフィルタリングする．これによって, いわゆるゼロデイ攻撃などの未知の脆弱性に対しても耐性のある手法となっている．

本論文では, デバイスドライバのバイナリコードから正当な I/O シークエンスを抽出する手法を示す．I/O シークエンスを抽出するには, 1) デバイスレジスタに対するアクセスおよび 2) そのアクセスが行われる順序を抽出すれば良い．デバイスレジスタに対するアクセスは, I/O ポートやメモリマップト I/O (MMIO) に対するアクセスとして実現されるため, その抽出は容易である．デバイスレジスタに対するアクセス順序を抽出するには, バイナリコードを基本ブロックに分割し, 基本ブロック間の制御フローに従った順序でアクセスされるものとすればよい．さらに, デバイスレジスタに書き込まれる値を特定するため, データフロー解析を行うようにしている．

バイナリ解析フレームワークである ROSE[7] に提案方式の実装を行い, xv6[8] の IDE ドライバ, Linux kernel の FDC について適用し, それぞれ 100%, 52.2% の I/O シークエンスを抽出することができた．

本論文の構成は, 次のとおりである．第 2 章では, デバイスエミュレーションの脆弱性を例示する．第 3 章では, 本論文の脅威モデルについて説明する．第 4 章では, 本論文で提案するフィルタについて説明する．第 5 章では, 実際にフィルタを作成する．第 6 章では, 関連研究について説明する．第 7 章では, 本論文のまとめを述べる．

2. デバイスエミュレーションの脆弱性

ハイパーバイザに存在する脆弱性の例として, VENOM (CVE-2015-3456) がある [6]．これは QEMU における仮想 FDC エミュレーションの脆弱性である．この脆弱性の悪用によって VM Escape を誘発する可能性があるため, CVSS v2 スコアが 7.7 と高い値となっている．

この脆弱性は, FDD と FDC 間でのデータ転送に利用される FIFO バッファのオーバーフローが原因である [9]．攻撃者は, FDC が発行する特定のコマンドが FDD で実行されている最中という条件下で, FIFO バッファへの書き込みを行い, オーバーフローを引き起こす．

しかし, そもそもエミュレートされたドライバが, 通常のデバイスドライバでは絶対示されない挙動を行えることがより根源的な原因である．FDC において, コマンドを処理する際, コマンド入力状態, コマンド実行状態, 実行結果取得状態の順に状態が遷移する．実際の FDC では FDD での処理に時間がかかるため, FDC がコマンド実行状態である時間が長い．QEMU ではこの処理遅延も再現している．このコマンド実行状態において, 通常のデバイスドライバはハードウェアレベルで FIFO バッファへ書き込みを行うことが禁止されている．しかし, QEMU でのエミュレーションでは, 状態管理の不備によって, コマンド実行状態での遅延エミュレート中に FIFO バッファへの書き込みができてしまう．すなわち, FDC が FDD のコマンド実行を待機している最中に, FIFO バッファに書き込むという, 通常のデバイスドライバでは想定されない不正な挙動が可能となる．

3. 脅威モデル

本論文では, 次のような脅威モデルを想定した：攻撃者は, ゲスト環境における管理者権限を有しているものとする．すなわちゲスト環境内で特権命令などを自由に実行できる．攻撃者はゲスト環境からハイパーバイザや同一ホスト上の他の VM を攻撃する．ここで, ネットワーク経由の攻撃など, 外部からの攻撃は対象としない．

4. I/O リクエストフィルタの自動生成

4.1 Nioh

Nioh[10] では, 攻撃者が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を悪用することを防ぐフィルタを提案する．フィルタは VM とハイパーバイザの間に存在

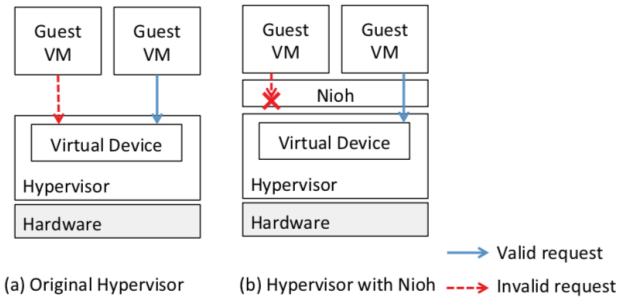

図 1　Nioh [10]

し, VM から発行される仮想デバイスへの I/O 要求をフィルタリングする (図 1(b)). 具体的には, VM が I/O 要求を発行すると VM Exit が発生する. この段階でフィルタに処理を遷移させフィルタリングを行う. これによってハイパーバイザで不正な I/O 要求が処理される前に取り除く.

Nioh では, 脆弱性を引き起こす I/O 要求はデバイスの仕様にそぐわない場合が多いことに着目し, 図 2 のようなデバイスの仕様書から手動で作成されたオートマトンを用いてフィルタを作成する.

このオートマトンでは, デバイスに関する 3 つの情報を管理する. コマンド入力待ち状態やコマンド実行状態といったデバイスの状態と, デバイスの状態で受け付けられるデバイスドライバからの要求, すなわち仕様書に沿った正しい要求, そして正しい要求を処理した後に遷移する状態である. デバイスが取りうる全ての状態について, その状態で受け入れ可能な全ての要求とその遷移先の状態を, 仕様書を調査することで決定し, 手動でオートマトンの形にする. このオートマトンによって, デバイスの仕様書に基づいた正しい挙動が予め定義される. これによって, 仕様にそぐわない場合の多い, 脆弱性を悪用する I/O 要求を取り除く.

4.2　I/O シーケンスの自動抽出

Nioh のように, 仕様書に基づいたオートマトンを手動で作成することで, フィルタを生成することには困難が伴う. なぜなら, 物理デバイスに即したフィルタを間違いなく書くことが難しいからである. デバイスの仕様は複雑な場合が多い. さらに, デバイスの仕様書は数百ページにわたる場合があり, 仕様書を読了すること自体が難しい. そして, 自然言語という曖昧な形で仕様が定義されている場合もある. このような理由から, 仕様書に基づいて物理デバイスに即したフィルタを書くことは難しく, フィルタの作成に困難が伴う. さらに, デバイスによってはそもそも仕様書が非公開の場合もある. この場合, フィルタを作成することは不可能となる.

本論文ではデバイスドライバのバイナリコードを自動で解析することで, I/O シーケンスを自動で抽出し, これを元

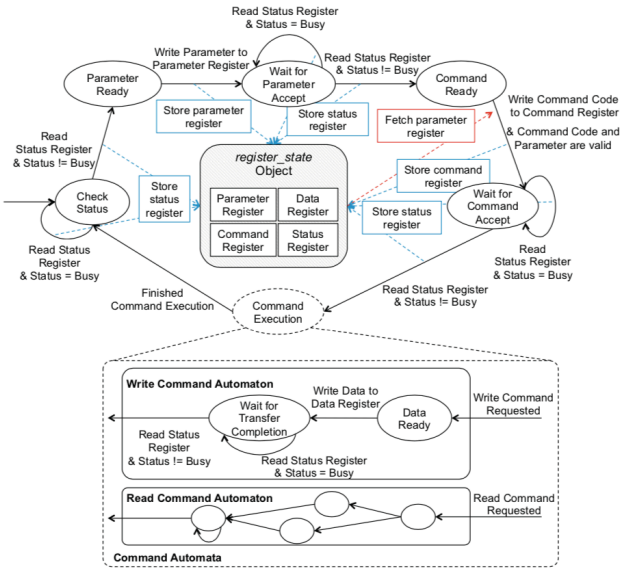

図 2　Nioh で用いられるオートマトン例 [10]

にしたフィルタを生成することを提案する. I/O シーケンスとは, デバイスレジスタに対するアクセス順序とその際読み書きされる値のことである.

以下では, デバイスレジスタにアクセスする際に読み書きされる値の抽出手法について説明する. ここで, 以下で用いる 3 語について次のように定義する.

- ターゲットレジスタ : デバイスレジスタへのアクセス時に読み書きされる値を抽出するために, 現在解析する必要があるレジスタのこと.
- ターゲットレジスタリスト (リスト): ターゲットレジスタが含まれるリスト. 単にリストと言及する場合は, ターゲットレジスタリストを指すものとする.
- 関係命令リスト : デバイスレジスタへのアクセス時に読み書きされる値を抽出するために, 解析する必要がある命令群のこと. 最終的にこのリスト中の命令から読み書きされる値を決定する.

抽出は次のような手順で行われる.

(1) I/O 命令の探索
(2) I/O 命令において, デバイスレジスタにアクセスする際に読み書きされる値の抽出

このフローは, コード 1 の擬似コードが示すように, デバイスドライバのバイナリコードが含む関数のコントロールフローグラフ (CFG) ごとに, さらに CFG が含むベーシックブロックごとに行われる.

各 I/O 命令について, デバイスレジスタへのアクセス時に読み書きされる値の抽出は次のような手順で行われる.

(1) リストを初期化する.
(2) 解析対象の命令を含んでいるベーシックブロック内の解析を行う.
(3) 解析が完了しない場合は, ベーシックブロックを遡っ

```
1 foreach cfg in all CFGs in the binary
2   foreach basicblock in all basic blocks
        in the cfg
3     foreach ioInst in all I/O instruction
        in the basicblock
4       extract the value read or written in
          the I/O instruction
```

Listing 1 I/O シーケンスの自動抽出の流れ

```
1 initialize targetRegisterList;
2 analyze the basic block named bb to which
    the I/O instruction belongs;
3 if finish analyzing
4   return;
5 foreach basicblock in all previous basic
    block in bb
6   analyze;
```

Listing 2 I/O シーケンスの自動抽出の流れ 2

て解析を行う。

リストの初期化では, 解析対象の I/O 命令で読み書きされるレジスタを格納する. 例えば, ポート I/O 方式で in 命令を解析する場合, デバイスレジスタから何らかの値を, ソースレジスタが示す I/O ポート経由で読み込む. 従って, ソースレジスタをリストに格納する. out 命令の場合は, 出力する値も必要であるのでデスティネーションレジスタも格納される. そして, まず始めに解析対象の命令が属すベーシックブロック内の解析を行う. この際, 解析対象の直前の命令から遡って解析する. 次に, 解析対象の命令が属すベーシックブロック内の解析が終了した段階で, デバイスレジスタへのアクセス時に読み書きされる値の抽出が完了しない場合, さらに命令を遡る. この際, 解析対象の命令が属すベーシックブロックの前の各ベーシックブロックについて, ベーシックブロックの最後の命令から遡って解析する (コード 2).

解析対象の命令が属すベーシックブロックについて解析を行うときも, そのベーシックブロックの前のベーシックブロックについて解析を行うときも, 解析の手法は次のように行われる. ただし, 命令は遡って探索される. これは, レジスタの値の確定を行うためには, レジスタへの変更が加えられる順序とは逆に辿る必要があるためである.

(1) リスト内のレジスタが変更される命令を探す.
(2) 発見した場合, 関係命令リストに追加し, リストを更新する.

解析対象の命令が属すベーシックブロックのその命令よりも前にある命令について, あるいはベーシックブロック内の命令について, リストに属すレジスタが変更されるかを判定する. そして, もし変更する場合は, その命令を関係命令リストに追加し, その命令に応じてリストを更新する. もし, 解析が終了できる場合は終了し, できない場合は解析を続行する (コード 3).

```
1 if analyzing the basic block to which that
      the I/O instruction belongs to
2   instList includes all instructions in
      the basic block before the I/O
      instruction;
3 else
4   instList includes all instructions in
      the basic block;
5
6 foreach instruction in instList
7   if instruction changes register in List
8     add the instruction to involved
        instruction;
9     update List;
10    if finish analyzing
11      return;
```

Listing 3 I/O シーケンスの自動抽出の流れ 3

リストの更新は, コード 3 で発見された命令の種類とデスティネーションレジスタのサイズに応じて更新を行う. コード 3 で発見された命令のソースレジスタは, 必ずリストに加えて追跡する必要がある. しかし, リスト中のレジスタのサイズと取得した命令のデスティネーションレジスタのサイズの関係によっては, リスト中のレジスタをリストから削除する場合と, しない場合がある. 例えば, x64 の場合, リスト中のレジスタが dx で, 取得した命令のデスティネーションレジスタが rdx というような状況である.

取得した命令が add 命令のような命令の場合, リスト中のレジスタは削除しない (表 1). なぜなら, この命令が実行される前のデスティネーションレジスタの値も把握する必要があるからである. add dst, src の場合, この命令の実行による dst の値を確定するためには, src の値と, この命令が実行される前の dst の値の 2 つを把握しなければならない. したがって, デスティネーションレジスタの内容は引き続き解析対象にしておく必要がある.

表 1 mov 命令以外のターゲットレジスタリスト更新

reg0 へのアクセスサイズ	命令	更新後のターゲットレジスタリスト
reg0 の下位ビットが変更される場合	add lower-bit-of-reg0, src	reg0, src
reg0 が同じ場合	add reg0, src	reg0, src
reg0 が上書きされる場合	add reg0, src	reg0, src

しかし mov 命令の場合, リスト中のレジスタを削除する場合がある (表 2). なぜなら, mov 命令ではデスティネーションレジスタの内容はソースレジスタの内容に置き換わ

るためである. mov dst, src の場合, dst の値は src の値であり, この命令実行前の dst の値は関係がない. しかし, もしリスト中のターゲットレジスタの一部のみを mov 命令によって変更する場合は, リスト中のレジスタの値が全て確定できるわけではないため, リストに残す必要がある. 例えば, リスト中のレジスタ A が 4 バイトの場合に, mov 命令で A の下位 2 バイトが変更されたとする. この場合, 4 バイトのうち上位 2 バイトの値は未確定であり, 引き続き解析する必要がある.

表 2 mov 命令時のターゲットレジスタリスト更新

reg0 へのアクセスサイズ	命令	更新後のターゲットレジスタリスト
reg0 の下位ビットが変更される場合	mov lower-bit-of-reg0, src	reg0, src
reg0 が同じ場合	mov reg0, src	src
reg0 が上書きされる場合	mov reg0, src	src

解析が終了となるのは, リストが空になるかベーシックブロックを遡ることができなくなるときである. リストからターゲットレジスタが削除されうるのは, mov 命令によって即値が格納されるときあるいは, lea 命令などで rip レジスタを用いて定数にアクセスする場合である. これはすなわちリスト中のそのレジスタの値が確定できたことを示す. 従って, リストが空になる場合, 解析対象の I/O 命令において, デバイスレジスタにアクセスする際に読み書きされるレジスタの値を確定できたことになる. 一方、 ベーシックブロックをそれ以上遡れない場合は, コントロールフローの解析失敗か, 関数の最初のベーシックブロックに到達した場合である. いずれの場合も, 解析は完了しなったことを意味する.

ターゲットレジスタリストの更新例としてコード 4 を考える. まず, 5 行目に in 命令がある. in 命令では入力ポートを確認するため, リストに dx が格納される. 以下, dx が変更される命令を遡って探索する. 4 行目では dx (2 バイト) に対して, edx (4 バイト) の mov 命令が実行されている. この mov 命令によって, eax の値がわかれば, edx の値がわかり, 結果的に dx の値が判明する. したがって, 4 行目の解析が終了した段階では, dx は削除され eax のみがリストに格納される. 一方 3 行目では, add 命令によってターゲットレジスタリストにある eax が変更されている. add 命令では, デスティネーションレジスタにソースレジスタの値を加算するので, eax, ebx 共に解析が必要であり, リストには eax, ebx が格納される. 2 行目では ebx に 0x6 が mov される. この段階でリスト中の ebx の値は決定できる. したがって, 2 行目終了時では ebx はリストから削除され eax のみが残留する. 同様にして 1 行目では　eax の値が決定できるため, eax はリストから削除され, リストは空となる. この例ではリストが空になったので, 解析が完了したことになる.

```
1  mov     eax,0x3f0
2  mov     ebx,0x6
3  add     eax,ebx
4  mov     edx,eax
5  in      al,dx
```

Listing 4 ターゲットレジスタ更新例

```
1  #include <sys/io.h>
2
3  void test() {
4      unsigned char value = 0xff;
5      int port, cnt, flag = 2;
6      for (cnt = 0; cnt < 100; cnt++) {
7          if (inb(0x1f7) != 0) {
8              flag =0;
9              break;
10         }
11     }
12     if (flag)
13         port = 0x3f0;
14     else
15         port = 0x3f1;
16
17     outb (value, port);
18 }
```

Listing 5 サンプルコード

コード 5 では, ループが含まれる場合や, ベーシックブロックを遡る例について示す. ここで, ベーシックブロック内での解析はコード 4 で説明したため, 省略する. また, 図 3 はコード 4 の CFG であり, 0x4000dc にあるベーシックブロックの out 命令を考えるものとする.

まず, dx レジスタはこのベーシックブロックでは確定できない. したがって, ベーシックブロックを遡る必要がある. このベーシックブロックには 0x4000d7, 0x4000d0, 0x4000c9 の 3 つのベーシックブロックが存在する. それぞれのブロックで, edx に 0x3f0 ないしは 0x3f1 が格納されている. したがって, 解析結果は 3 つ存在し, 0x4000d7, 0x4000d0 の場合は ”mov edx, 0f3f1”, 0x4000c9 の場合は”mov edx, 0x3f0” が出力される.

また, 0x4000bf の in 命令を解析する際, ベーシックブロックを遡る関係上, 0x4000c4 のブロックも解析される. この場合, 0x4000bf, 0x4000c4 でループが発生してしまう. これに対しては, 探索できるベーシックブロックの数を制限することで対応を行った.

4.3 フィルタ生成

フィルタの生成のために, I/O 命令の引数とその発行順

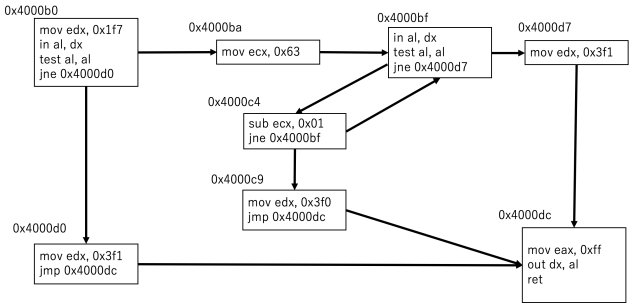

図 3 コード 5 の CFG

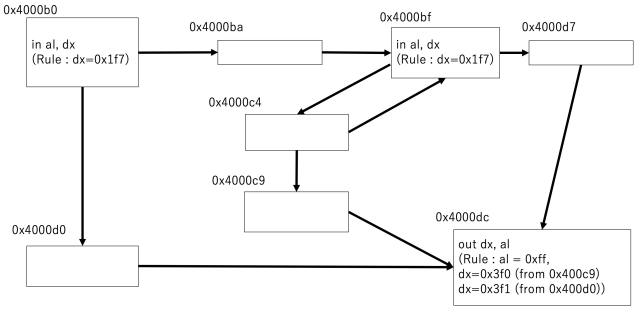

図 4 コード 5 のフィルタ

序を把握する必要がある. 前者については, 4.2 節によって把握できる. 後者については, CFG をデバイスドライバの状態遷移図のように扱うことで把握できる. 例えばコード 5 の場合, このような状態遷移を持つフィルタは図 4 となる. インストラクションポインタから, 現在実行中の命令が属すベーシックブロックを確認し, I/O 命令が発行された際は, その命令のルールを確認する. また, ベーシックブロックに含まれる jmp 命令などから, 次に遷移する状態を決定する.

このように I/O 命令を含むベーシックブロックのみのフローを状態遷移として扱い, 各ベーシックブロックが含む I/O 命令について, 解析の結果取得できた情報を I/O 命令のルールとするようにフィルタを記述することで, フィルタを生成できる.

4.4 実装

I/O 要求の順序と引数の解析を行う環境は表 3 の通りである. またポート I/O 方式を対象として実装を行なった. この環境で解析することを想定して, I/O 要求の順序と引数の解析を行う機構を ROSE を用いて C++ で実装した. ROSE[7] は Lawrence Livermore National Laboratory (LLNL) で開発されているバイナリ解析ツールである. ROSE にはバイナリから CFG やベーシックブロック, そしてベーシックブロックに含まれるアセンブリ命令の情報を取得する機能がある. この機能を用いて解析機構を実装した.

また, I/O 要求の順序と引数についての情報からフィルタを作成する際の環境は表 4 の通りである. 本論文では, 仮想

表 3 解析環境

Host System	Ubuntu 18.04.2 LTS (Bionic Beaver)
HostCPU	Intel Xeon CPU X5560 @ 2.80GHz * 4 Cores (8 logical processors)
Host Memory	32 GB
ROSE	0.9.10.213 (Commit: 9cd82edd986a7d818becc9c960b18121311800e5)

表 4 フィルタの実装環境

Host System	CentOS Linux release 7.3.1611 (Core)
HostCPU	Intel Xeon CPU X3480 @ 3.07GHz * 4 Cores (8 logical processors)
Host Memory	16 GB
Host QEMU	Modified 2.9.50 (Commit: 6db174aed1f70215b681aaf3a6a9e23e2c7ba86d)
Guest System	Ubuntu 18.04.1 LTS (Bionic Beaver)
Guest VCPU	1 VCPU
Guest Memory	1 GB

化基盤として KVM+QEMU を使用した. KVM+QEMU における I/O の流れは, 4.4.1 章で述べる.

4.4.1 KVM+QEMU

KVM[3] は Linux カーネルをハイパーバイザとして利用するためのモジュールである. KVM を利用する場合, I/O 処理のようなフロントエンドのエミュレートは QEMU[2] を利用する. QEMU はユーザ空間で動作するデバイスエミュレータである.

KVM では次のようにしてデバイスエミュレーションが行われる (図 5) まず, VM が I/O 要求を発行する (1) と, VM Exit が発生し, 処理が KVM に移る (2). I/O 要求による VM Exit なので, この要求が QEMU に渡され処理される (3). 処理が終了すると KVM に処理が戻り (4), VM Entry して, 処理が VM に戻る (5, 6).

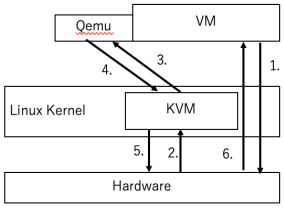

図 5 KVM におけるデバイスエミュレーション

5. 実験

本実験では, 4 章で提案した手法によって I/O 要求の引数情報が抽出できることを示す. そのために, 表 3 の環境で解析を行なった. 解析対象は xv6 の ide.o と linux カーネルの floppy.ko とした. xv6 [8] は MIT で開発された教育用オペレーティングシステムである. 表 5 に, xv6 と linux カーネルのバージョン情報を示す. また, ポート I/O 方式を対象に実験を行なった.

ただし, ソースファイルの行数にはヘッダファイルは含まないものとする.

表 5　xv6 と linux カーネルのバージョン情報

xv6 version	Commit: b818915f793cd20c5d1e24f668534a9d690f3cc8
linux kernel version	4.15.0-47-generic

表 6　ide.o と floppy.ko のサイズ

ファイル名	ソースファイル行数	アセンブリ命令数	バイナリファイルのサイズ	ベーシックブロック数
ide.o	168	225	3613 バイト	72
floppy.ko	4982	9665	133990 バイト	2305

また, ide.o に含まれる I/O 命令の総数は, in が 4 命令, out が 10 命令の合計 14 命令, floppy.ko に含まれる I/O 命令の総数は in が 11 命令, out 命令が 33 命令の合計 44 命令であった.

ide.o について, 全ての I/O 命令を解析することができた. また floppy.ko について, 解析できた I/O 命令は, 23 命令で 52.2% 存在した. 残りの 21 命令については, 引数を用いた命令のために失敗した命令が 14 命令, 間接ジャンプが利用されていることが原因の命令が 7 命令存在する. 解析に成功した命令のうち, ide.o, floppy.ko のそれぞれ 2 命令は xor eax, eax という eax レジスタの 0 初期化に対応する必要があり, これに対処した.

間接ジャンプは, コード中で switch 文が利用されている部分に存在した. ROSE ではこの部分の解析に失敗した. これが原因である. 具体的には switch 文のそれぞれの case 文を含むベーシックブロックについて, ベーシックブロックに入るエッジの解析に失敗し, ベーシックブロックを遡ることに失敗した. 間接ジャンプに対応するためには, switch 文では用いられる制御遷移先配列を調査し, ベーシックブロックに入るエッジの解析ができるようにする必要がある. また, 関数の引数を用いた命令のために解析が失敗したものについては, 関数を呼び出す側でのレジスタの解析を行う必要がある.

6. 関連研究

ハイパーバイザの脆弱性に関して, 先行研究では様々な角度からのアプローチがなされている. 本セクションでは, その中から 3 つ取り上げ, 本論文との比較を行う.

6.1 Nioh

Nioh[10] では, 本論文と同様に, ハイパーバイザにおいてデバイスエミュレーションが脆弱性の温床となっていることに注目している. そして, 脆弱性を悪用する VM から発行される I/O 要求をフィルタリングする機構を提案している.

この機構では, VM から仮想デバイスに向けて発行される全 I/O 要求をフィルタリングする. フィルタは, デバイスの仕様書を元に, 完全なデバイスの状態遷移と全デバイスレジスタの値を管理する. これによって, デバイスの仕様にそぐわない脆弱性を悪用する不正な I/O 要求を弾くことができる.

Nioh では, 仕様書を元にフィルタを作成するため, 脆弱性を幅広く網羅できる. しかし一方で, 仕様書は文章量が多く, さらに自然言語で仕様が書かれているなど, 必ずしも明確に仕様が規定されているわけではない場合がある. さらに, デバイスの状態遷移は複雑な場合がある. これらのことから, フィルタの作成には労力がかかると言える.

本論文では, デバイスドライバのバイナリを元にフィルタを作成し, デバイスエミュレーションに適用するため, 仕様書を読み込んだり, デバイスの状態遷移を把握する必要はない. しかし, デバイスドライバそのものに脆弱性があった場合は本提案では対応することが難しい.

6.2 ハイパーバイザの攻撃面の削減

ハイパーバイザのコードベースが大きいと, その分脆弱性の数は増える. 従って, コードベースを削減することで脆弱性を削減しようという研究がなされている.

Delusional Boot[11] では, クラウドの VM が汎用の VM に比べて, 使用するデバイスが限定されていることに注目し, Min-V というハイパーバイザを提案している.

この Min-V では, クラウド上での VM の実行には重要ではない仮想デバイスを全て無効化する. さらに, 重要な仮想デバイスについては, クラウド環境での実行には必要でない機能を削除する. これによって, 脆弱性が存在しうるコードベースの絶対量を削減し, ハイパーバイザの安全性を高めている.

しかし, 多くの OS は, Min-V で無効化した仮想デバイスがないとブートしない問題がある. そこで, delusional boot という機構を提案し, この問題に対処している. Delusional boot は, ブートサーバという専用のサーバ上で行われる. このサーバ上で通常通り VM をブートした後に, 仮想デバイスを無効化した安全な VM 設定で再度ブートする.

Delusional boot では重要ではない仮想デバイスの無効化や, 不要な機能の削除を行なっているが, 無効化や削除の対象とならなかった仮想デバイスの脆弱性に対応することは難しい.

6.3 ハイパーバイザの信頼性向上

ハイパーバイザをテストすることでバグを発見・修復し, ハイパーバイザの信頼性を向上させようという研究もなされている.

Virtual CPU Validation[12] は, 仮想 CPU に存在するバグを発見するためのテスト環境を提案する. 仮想 CPU に存在するバグは脆弱性につながり, ハイパーバイザを攻撃の危険に晒してしまう. そこで, 仮想 CPU のバグが実際の CPU の仕様書に反した動作を引き起こすことに着目する. そして, CPU ベンダーが物理 CPU のテスト時に利用するテスト環境を, ハイパーバイザに応用する. これによって,

仮想 CPU が CPU の仕様に反した動作を引き起こすバグを発見, 削減することができる. 結果的に, ハイパーバイザに潜む脆弱性の数を減らすことができる.

Virtual CPU Validation では, バグによって仮想ハードウェアがその仕様に反した動作をする点に注目している. 本論文では, 仮想デバイスがデバイスドライバの本来の I/O 要求という動作に反した動きをする点に注目しており, 関連した論文と言える. しかしながら, Virtual CPU Validation では対象としているハードウェアが CPU であり, テスト段階で動作の正当性を確認することに焦点を当てている一方で, 本論文では仮想デバイス一般を対象とし, VM が動作中にその動作の正当性を確認している. この点で, 対象としているハードウェアや動作の正当性を確認するタイミングが異なっており, 異なる論文であると言える.

7. まとめ

ハイパーバイザには脆弱性が存在し, 特にデバイスエミュレーションに関する脆弱性が数多く存在する. この脆弱性を悪用することによってハイパーバイザや同一ホスト上の他のマシンが乗っ取られる可能性がある.

本論文では, デバイスエミュレーションの悪用が仕様にそぐわない不正な I/O によって実行されることに着目し, この不正な I/O 要求を弾くフィルタを提案した. 具体的にはデバイスドライバのバイナリを解析し, I/O 要求が発行される順序や引数の情報を取得し, 正当な I/O 要求を定義した.

本論文で提案した手法を既存のデバイスドライバに適用し, 不正な I/O 要求を排除できる精度のフィルタが生成できることが確認された.

本提案では in/out 命令によって実行されるポート I/O 方式を対象としてフィルタを作成した. その過程で in/out 命令について解析を行った. この手法を mov 命令に応用することで, MMIO 方式に対応したフィルタを作成することができる.

謝辞 本研究は, JSPS 科研費 JP19K11906 の助成を受けたものである.

参考文献

[1] National Institute of Standards and Technology: National Vulnerability Database, National Institute of Standards and Technology (online), available from ⟨https://nvd.nist.gov/⟩ (accessed 2019-6-17).
[2] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, USENIX Association, pp. 41–46 (2005).
[3] Kivity, A., Lublin, U., Liguori, A., Kamay, Y., and Laor, D.: kvm: the Linux virtual machine monitor, *Proceedings of the Linux Symposium 1*, pp. 225–230 (2005).
[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, ACM, pp. 164–177 (online), DOI: 10.1145/945445.945462 (2003).
[5] FIRST.org: Common Vulnerability Scoring System SIG, (online), available from ⟨https://www.first.org/cvss/⟩ (accessed 2019-6-17).
[6] CrowdStrike: VENOM Vulnerability, (online), available from ⟨https://venom.crowdstrike.com⟩ (accessed 2019-6-17).
[7] Lawrence Livermore Nationa Laboratory: ROSE compiler infrastructure, (online), available from ⟨http://rosecompiler.org⟩ (accessed 2019-6-17).
[8] R. Cox, M. F. Kaashoek, and R. T. Morris: Xv6, a simple Unix-like teaching operating system, 2017, (online), available from ⟨http://pdos.csail.mit.edu/6.828/xv6⟩ (accessed 2019-6-17).
[9] Petr, M.: git.qemu.org Git - qemu.git/comitdiff e907747bc0718795fedee2e824c, (online), available from ⟨https://git.qemu.org/⟩ (accessed 2019-6-17).
[10] Ogasawara, J. and Kono, K.: Nioh: Hardening The Hypervisor by Filtering Illegal I/O Requests to Virtual Devices, *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, New York, NY, USA, ACM, pp. 542–552 (online), DOI: 10.1145/3134600.3134648 (2017).
[11] Nguyen, A., Raj, H., Rayanchu, S., Saroiu, S. and Wolman, A.: Delusional Boot: Securing Hypervisors Without Massive Re-engineering, *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, New York, NY, USA, ACM, pp. 141–154 (online), DOI: 10.1145/2168836.2168851 (2012).
[12] Amit, N., Tsafrir, D., Schuster, A., Ayoub, A. and Shlomo, E.: Virtual CPU Validation, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 311–327 (online), DOI: 10.1145/2815400.2815420 (2015).