仮想計算機モニタを使ったマルウェアの挙動解析

野村 和裕† 吉村 拓也† 毛利 公一†

†立命館大学情報理工学部情報システム学科
525-8577 滋賀県草津市野路東 1–1–1

{knomura,tyosimura,mouri}@asl.cs.ritsumei.ac.jp

あらまし マルウェアの技術は着実に進歩しており，日々新しいマルウェアが出現している．これらの対策のためには，マルウェアの挙動を解析し，その結果を侵入検知システムなどへフィードバックさせる必要がある．従来は逆アセンブルやデバッガによる解析が可能であったが，最近では難読化やデバッガ検出機能を有するマルウェアがある．また，そういったマルウェアであっても，OS 内からであれば挙動を観測しやすいが，マルウェアの主たる対象となっている Windows はプロプライエタリソフトウェアであるため，それが難しい．そこで，我々は，ハイパーバイザ型の仮想計算機モニタを利用してマルウェアの挙動を解析しようと試みている．本論文では，BitVisor に変更を加えることで，Windows 上で動作するマルウェアを観測する手法について述べ，その開発状況について報告する．

Analyzing Behavior of Malware by Virtual Machine Monitor

Kazuhiro Nomura† Takuya Yoshimura† Koichi Mouri†

†Department of Computer Science, Ritsumeikan University
1–1–1 Nojihigashi, Kusatsu, Shiga 525-8577 Japan

{knomura,tyosimura,mouri}@asl.cs.ritsumei.ac.jp

Abstract Technology of malwares progresses steadily, and new ones are appearing everyday. For provision against them, we need to analyze their behavior, and its results should be fed back to intrusion detection system and so on. Old-fasioned malwares can be analyzed by a disassembler or a debugger. But, recently, some ones are packed not to be analyzed easily. Some other ones have a mechanism for detecting a debugger, and they change their behavior when a debugger was detected. Even if they have such mechanisms, it is easy to analyze them from inside operating systems. However, actually, it is not easy from Windows because it is a proprietary software. To solve these problems, we are trying to analyze malware's behavior by hypervisor-based virtual machine monitor. In this paper, we discuss how to improve BitVisor to monitor malware's behavior on Windows.

1 はじめに

社会基盤がコンピュータとネットワークに依存している現在，マルウェアによる脅威が問題となっている．この問題に対して，マルウェアの侵入を防止・検出・排除するための対策が行われている．しかし，マルウェアは継続的に進歩し，新種・亜種が日々でてきているため，対策が先回りすることは難しい．一般的には，新種・亜種が出てきた場合，それらを解析し，その

結果をマルウェア対策にフィードバックさせる．

マルウェアを解析する場合，従来は，それを逆アセンブルするなどして解読したり，実際に動作させながらデバッガで観測するといった方法が採られる．しかし，最近のマルウェアは，難読化によって解読しづらくするといった手法がとられているような場合がある．また，デバッガ検出機能を有し，自身がデバッガの対象であると検出すると，自身の振舞を変えてしまうようなマルウェアが報告されている [1]．このような機能を持ったマルウェアを解析する方法として，オペレーティングシステム (OS) の中から観測することが考えられる．マルウェアのメモリ領域を参照したり，マルウェアが発行するシステムコールを追跡したりすることができる．しかし，マルウェアの主な攻撃対象となっている Windows は，プロプライエタリソフトウェアであるため，こういった手法を採ることが困難である．

以上の問題点を解決するために，我々は，ハイパーバイザ型の仮想計算機モニタ (VMM) である BitVisor[2] に変更を加え，Windows 上で動作するマルウェアの挙動を観測するための仕組みを構築した．本論文では，その観測のための仕組みについて述べるとともに，ある種のマルウェアに対してそれが有効であることについて述べる．

以下，2 章ではマルウェアの観測手法についてその概要を述べ，3 章でその実装の具体的な方法について述べる．さらに，4 章では実際にマルウェアを観測した結果を示し，5 章でそれに対する考察を行なう．

2　マルウェア観測手法

2.1　全体構成

本論文で提案する観測手法は，図 1 に示す構成となっている．マルウェアをはじめとするユーザモードで動作するプロセス，カーネルモードで動作し，プロセスに種々のサービスを提供する OS，さらに，OS よりもさらに特別な VMM 用のモードで動作する VMM から成る．

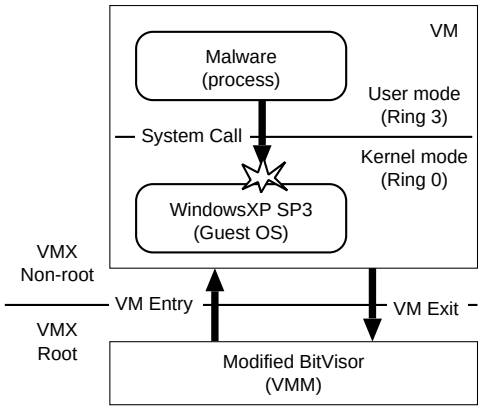


図 1: 提案手法の概要

2.2　プロセスと OS

図 1 の構成要素のうち，プロセスと OS については，VMM が作り出す仮想計算機 (VM) 環境の中で動作する．観測をするにあたって，特に必要となる変更はない．プロセスが OS の機能を呼び出す場合には，ライブラリ等を通じてシステムコールが発行されることで実現される．

2.3　仮想計算機モニタ

VMM は，VM を生成し，VM が動作するに必要な環境を提供することが目的である．提案手法では，VM 内でプロセスや OS がどのような動作をしているかを観測する機能を追加した．

一般的な VMM は，複数の VM を実現することが目的であるため，CPU，メモリ，デバイスを，各 VM にどのように割り当てるかを調整することが主たる処理内容となる．そのために，特権命令，入出力・割込み関連，主要レジスタへのアクセスなどを VM 上で動作するプログラムが実行すると，VMM へ処理が移るような仕組みとなっている (VM Exit)．さらに，VMM がそれを適切に処理した後，VM の動作を継続させる (VM Entry)．

本観測手法では，この VM Exit/Entry の仕組みを利用した．VM 上のプロセスがシステムコールを発行すると，カーネル内のシステムコールハンドラの入り口で VM Exit が発生するよう

にした．ここで，システムコールに関する情報を取得することで観測可能としている．

このような手法とすることで，OS がプロプライエタリソフトウェアであっても，VM Exit が発生するタイミングでその挙動を解析するための情報を取得することができる．全てのレジスタとメモリの内容を取得可能である．

今回，OS として WindowsXP SP3 を対象とした．また，VMM は，BitVisor を改変することで，システムコール番号と，仮想記憶のページディレクトリのアドレスを指すレジスタの値 (CR3 レジスタ; プロセスごとに値が異なる) を取得する機能を実現した．

3 実装

3.1 CPU による仮想化支援

CPU による仮想化支援には，Intel VT や AMD-V などがある．ここでは，Intel VT について簡単に説明する．

Intel x86 アーキテクチャでは，特権レベルとして Ring0 から 3 までが存在し，一般に OS は Ring0 で，プロセスは Ring3 で動作する．Intel VT では，このような OS やプロセスが動作するためのモードを VMX non-root モードと呼ぶ (図 1 参照)．VMX non-root モードで動作する VM は複数生成することができる．また，VM 上で動作する OS を特にゲスト OS と呼ぶ．さらに，Intel VT では，新たに Ring とは別の，VMM を動作させるための特別なモードである VMX root モードを作った．VMX non-root モードでゲスト OS が特権命令を発行すると，VMX root モードに切り替わり，VMM に処理が移る．この遷移を VM Exit と呼ぶ．逆に，VMX root モードから VMX non-root モードへの遷移を VM Entry と呼ぶ．

それぞれの VM や VMM のプロセッサの状態の保存や，挙動を制御するためのデータ構造体として，VMCS(Virtual Machine Control Structure) がある [3]．その内容には，表 1 に示す 6 つの領域がある．特にここで重要なのは Guest-state area であり，これによって各種レジスタの値が取得できる．

3.2 BitVisor

BitVisor は，セキュア VM プロジェクトにより開発されている国産の VMM である．BitVisor は様々な特徴を有しているが，特に我々にとって特徴的であるのは次の点である．

- BSD ライセンスに基づくオープンソースのため，改変を加えることが可能である．
- ハイパーバイザ型の VMM であること．
- Windows，Linux を改変することなく実行させることが可能である．

3.3 システムコールのフック手法

マルウェアの挙動の解析を行なうためには，プロセスが特権を必要とする処理を行なう際に必ず呼び出すシステムコールをフックし，その情報を取得・解析することが効果的と考えられる．

Windows XP 以降では，システムコールを発行する際には sysenter 命令が用いられる．sysenter 命令が発行されると，sysenter_eip_msr レジスタに格納されているシステムコールエントリポイントへと処理が移行する (図 2 参照)．しかし，Windows 2000 以前に使われていた int 0x2e の命令とは異なり，sysenter 命令はソフトウェア例外ではないため，単純にシステムコールが発行 (sysenter 命令が実行) されるだけでは VM Exit は発生しない．そこで，VM Exit を発生させるため，図 2 にあるように，システムコールエントリポイントにブレークポイント命令 (INT3) を埋め込んだ．これにより，システムコールが呼び出されると，ブレークポイント例外が発生し，VM Exit が起こって VMM へ制御を移すことができる．

本来，VMM ではブレークポイント例外が発生すると，ゲスト OS に対しその旨を通知するが，上記で埋め込んだブレークポイントにより発生した VM Exit の場合に関してのみ，ゲスト OS に対する通知を行なわず，システムコール番号などを取得する処理を行う．これらの一連の処理手順を以下に示す．

表 1: VMCS の領域

Guest-state area	VM Exit が発生したとき，VM のプロセッサのレジスタの値をセーブし，VM Entry が発生したときにロードする．
Host-state area	VM Entry が発生したとき，VMM のプロセッサのレジスタの値をセーブし，VM Exit が発生したときにロードする．
VM-execution control fields	VMX non-root モードでのプロセッサの動作を制御する． VM Exit を引き起こした要因も保持する．
VM-Exit control fileds	VM Exit の振る舞いを制御する．
VM-Entry control fields	VM Entry の振る舞いを制御する．
VM-Exit information fields	最後に発生した VM Exit の情報を保持する．

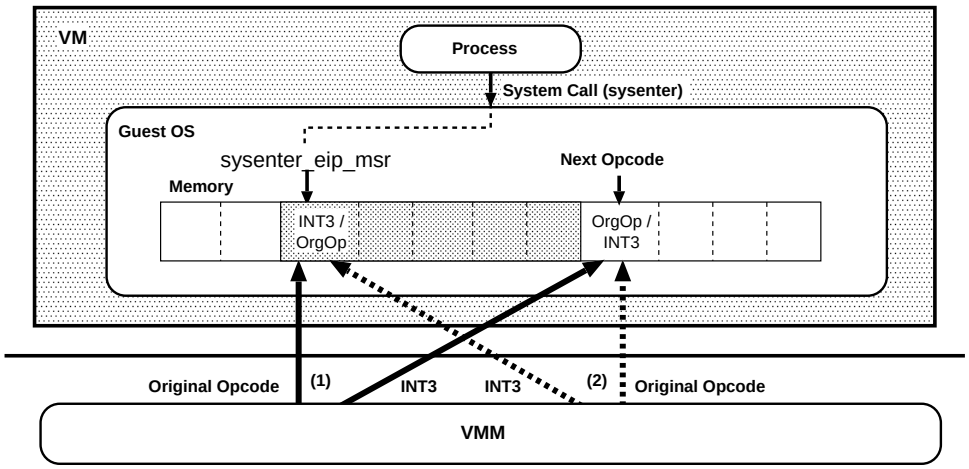


図 2: ブレークポイントの埋込み

1. VMM は，解析情報取得のための初期化処理として，次の処理を行う．
   - sysenter_eip_msr の位置にある命令と，その次 (NextOpcode の位置) にある命令のバックアップをとる．
   - sysenter_eip_msr の位置へ INT3 を埋める．
2. プロセスがシステムコールを発行すると，ゲスト OS 内にある sysenter_eip_msr の位置の命令を実行しようとするため，ブレークポイント例外が発生し，それに伴って VM Exit も発生する．これによって VMM へ制御を移す．
3. VMM 内ではシステムコール番号等の情報を取得する．
4. 次の命令書き換えを行う (図 2 の (1))．
   - 処理を継続させるために，sysenter_eip_msr の位置にある命令を元の命令に書き戻す．
   - 次回のシステムコールをフックする準備の処理をするために，次の命令の場所 (NextOpcode) へ INT3 を埋める．
5. ゲスト OS に処理を戻す (VM Entry)．
6. 次の命令 (NextOpcode) が INT3 であるため，また直ちに VMM へ制御が移る．
7. 次の命令書き換えを行う (図 2 の (2))．
   - 次回のシステムコールをフックするために，sysenter_eip_msr の位置に INT3 を埋める．
   - 処理を継続させるために，NextOpcode の位置に元の命令を書き戻す．

8. ゲスト OS に処理を戻す (VM Entry)．以降，システムコールが発行される度に 2 から処理を繰り返す．

以上のように，1 つのシステムコールが発行される度に 2 回のブレークポイント例外を発生させることで，情報を取得するとともに，継続的なフックを実現している．

Windows では，システムコール番号が EAX レジスタに格納されている．よって，上述の 3 のタイミングで，EAX レジスタの値を取得すればよい．

ただし，全てのプロセスのシステムコールが同じ箇所を通るため，どのプロセスによって発行されたシステムコールかを区別する必要がある．現在の実装では，上述の 3 のタイミングで，CR3 レジスタの値を取得している．CR3 レジスタは，ページディレクトリの物理アドレスを指すレジスタである [4]．すなわち，仮想アドレス空間が異なれば値が異なるため，プロセスを区別することができる．

取得した情報は，VMM のメモリ領域に格納している．この情報は，ハイパーバイザコール (VMCALL 命令) を使用して，ゲスト OS から取得することができる．

4 実行結果

以上で述べたシステムを用いて，実際のマルウェアを観測した．今回対象とするマルウェアは，Win32.Agobot と呼ばれるものを用いた．Win32.Agobot は，自身をファイルとしてコピーしたり，自動起動のためにレジストリを書き換えるといった動作をする [5]．また，デバッガ検出機能を有しており，自身がデバッガの対象となっていることを検出すると，動作を停止して動的解析を困難にする．さらに，パッカーで圧縮されおり，静的解析を困難にしているなど，様々な解析対策が施されたマルウェアである．

なお，Win32.Agobot は，サイバークリーンセンター (CCC) から提供された研究用データセットである “CCC2009 Data Set” の通信記録の中にも，ネットワーク的な挙動が記録されていたマルウェアである．

動作実験では，我々が改変を加えた BitVisor 上で WindowsXP SP3 を動作させ，さらにマルウェアを実行させた時のシステムコールのフックを行った．取得した情報は，システムコール番号と CR3 レジスタの値である．

取得した情報の結果の一部を図 3 に示す．左から，システムコールをフックする度にカウントしている通し番号，システムコール番号とそのシステムコール名 [6][7]，CR3 レジスタの値となっている．CR3 の値は 3 種類あることから，3 つのプロセスがこの間システムコールを用いていることがわかる．このうち，afc42000 のものが実行させたマルウェアと考えられ，NTCreatekey や NTQueryValueKey といった，レジストリ操作関係のシステムコールを多く発行していることが確認できた．

5 考察

システムコールをフックした結果より，デバッガの検出機能を有するマルウェアにおいても，それに検出されることなくマルウェアは通常通りに動作することを確認した．また，マルウェアが発行したシステムコールを獲得することができることを確認できた．さらに，マルウェアとシステムコールの関係では，通常のプログラムを動作させた時より，レジストリ操作やファイル操作のシステムコールが大量に起こる特徴があることを確認できた．

今後，提案手法について実装を進め，システムコール番号のみでなく，システムコールの引数も同時に取得していくことにより，ファイルアクセスの内容やレジストリ操作の内容が解析できるようになると考えている．

さらに，プロセスの CR3 の値をキーとして，プロセス ID やプロセス名 (コマンドライン) を求める機構を構築することで，簡単にプロセスが区別できるようになり，マルウェアの動作をより詳細に解析できるようになると考えている．

また，今回様々な環境チェックを行うマルウェアを実行させることができたことから，今後，より多くのマルウェアを実行調査しシステムの精度を高めていきたい．

```
                    :
13340 101(NtTerminateProcess)     afc5f000
13341  19(NtClose)                afc42000
13342  77(NtOpenKey)              afc5f000
13343  b1(NtQueryValueKey)        afc5f000
13344  19(NtClose)                afc5f000
13345  b1(NtQueryValueKey)        afc42000
13346  19(NtClose)                afc42000
13347  29(NtCreateKey)            afc42000
13348  29(NtCreateKey)            afc42000
13349  53(NtFreeVirtualMemory)    afc5f000
13350  19(NtClose)                afc42000
13351  29(NtCreateKey)            afc42000
13352  19(NtClose)                afc42000
13353  29(NtCreateKey)            afc42000
13354  e5(NtSetInformationThread) afc5f000
13355  19(NtClose)                afc42000
13356  e5(NtSetInformationThread) afc5f000
13357  19(NtClose)                afc5f000
13358  19(NtClose)                afc5f000
13359  c8(NtRequestWaitReplyPort) afc5f000
13360  77(NtOpenKey)              afc42000
13361  c2(NtReplyPort)            afc5a000
13362  19(NtClose)                afc5a000
13363  29(NtCreateKey)            afc42000
13364  19(NtClose)                afc5a000
13365  29(NtCreateKey)            afc42000
13366 101(NtTerminateProcess)     afc5f000
13367  19(NtClose)                afc42000
13368  29(NtCreateKey)            afc42000
13369  19(NtClose)                afc5a000
13370  c3(NtReplyWaitReceivePort) afc5a000
13371  19(NtClose)                afc42000
13372  19(NtClose)                afc42000
13373  29(NtCreateKey)            afc42000
13374  19(NtClose)                afc42000
13375  b1(NtQueryValueKey)        afc42000
13376  19(NtClose)                afc42000
13377  29(NtCreateKey)            afc42000
13378  29(NtCreateKey)            afc42000
13379  19(NtClose)                afc42000
13380  29(NtCreateKey)            afc42000
13381  19(NtClose)                afc42000
13382  29(NtCreateKey)            afc42000
13383  19(NtClose)                afc42000
13384  b1(NtQueryValueKey)        afc42000
13385  19(NtClose)                afc42000
                    :
```

図 3: フックしたシステムコールと CR3

6 おわりに

本論文では，VMM を用いてマルウェアを解析するために，VMM でゲスト OS となる Windows のシステムコールをフックする手法について述べた．さらに，実行結果を示し，その考察について述べた．

現在の実装状況では，CR3 によりプロセスの区別は行えているが，それぞれの何のプロセスかまでははっきりと判明させることはできていない．今後，システムコールの引数の情報などの解析をより詳細に行い，それぞれのシステムコールを使用しているプロセス名や，プロセス ID といった情報も取得可能にし，マルウェアの解析を行っていこうと考えている．

さらに，マルウェアを実際に解析してみるようなチャレンジをしていきたい．

参考文献

[1] 岩村 誠，伊藤 光恭，村岡 洋一: “コンパイラ出力コードモデルの尤度に基づくアンパッキング手法，” 情報処理学会シンポジウム論文集，Vol. 2008，No. 8，pp. 103–108，2008.

[2] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato: “BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, ” In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009), pp. 121-130, 2009.

[3] Intel: “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide Part 2, ” http://www.intel.com/Assets/PDF/manual/253669.pdf, 2009.

[4] Intel: “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide Part 1, ” http://www.intel.com/Assets/PDF/manual/253668.pdf, 2009.

[5] CA: “Win32.Agobot, ” http://www.casupport.jp/virusinfo/2005/win32_agobot.htm

[6] The Metasploit Project: “Windows System Call Table (NT/2000/XP/2003/Vista), ” http://www.metasploit.com/users/opcode/syscalls.html

[7] egggarden: “Windows System Call Table on Vista(SP0), ” http://d.hatena.ne.jp/egggarden/20080220/1203519277