仮想環境に依存せず詳細な解析能力を持つ動的解析システムの設計と実装

Fourteenforty Research Institute, Inc.

株式会社フォティーンフォティ技術研究所

もくじ

著作権

免責事項

当文書は AS-IS （現状有姿）にて提供され、フォティーンフォティは明示的かつ暗示的にも、いかなる種類の保証をも行わないものとします。この無保証の内容は、商業的利用の可能性・特定用途への適応性・他の権利への無侵害性などを保証しないことを含みます。たとえフォティーンフォティがそうした損害の可能性について通知していたとしても同様です。また「この文書の内容があらゆる用途に適している」あるいは「この文書の内容に基づいた実装を行うことが、サードパーティー製品の特許および著作権、商標等の権利を侵害しない」といった主張をも保証するものではありません。そして無保証の範囲は、ここに例示したもののみに留まるものではありません。

また、フォティーンフォティはこの文書およびその内容・リンク先についての正確性や完全性についても一切の保証をいたしかねます。

当文書内の記載事項は予告なしに変更または中止されることがありますので、あらかじめご了承下さい。

更新履歴

2011-07-26	1	

文書情報

発行元：　株式会社フォティーンフォティ技術研究所

連絡先：　株式会社フォティーンフォティ技術研究所

sales@fourteenforty.jp

〒162-0808

東京都新宿区天神町 8 番地　神楽坂 U ビル 2 階

概要

本研究では、マルウェアの解析に要する時間を削減するため、仮想環境に依存せず詳細なマルウェアの解析能力を持つ動的解析システムの設計と実装を行った。また、その有効性を評価するために、本システムのコンセプト実装である egg を用いて、実際のマルウェア 1 体と、複数のサンプルプログラムを解析した。マルウェアの解析の結果は、本システムがマルウェアの実行を 1 命令単位で解析することで、その実行履歴をもとに関数間の遷移情報や分岐トレース情報を生成し、また API の引数をファイルに保存するなど、従来の動的解析システムよりも詳細にマルウェアを解析できることを示した。サンプルプログラムの解析の結果は、マルウェアが他のプロセスにコードを注入したり、動的にファイルを生成し実行したりする挙動に対して、本システムが自動的にそれらを追跡し解析できることを示した。また、別のサンプルプログラムの解析の結果は、本システムが、従来の動的解析システムと同様にパックされたプログラムに対しても機能すること、広く知られたアンチデバッグ技法の多くに対して不可視性を持っていることを示した。以上の点から、本システムが、従来の動的解析システムより詳細な解析能力を持ち、手動での解析に要する時間、ひいてはマルウェア解析全体に要する時間を削減できるものと結論した。

- キーワード

Windows, Ring0, Malware analysis, Dynamic analysis, Taint tracing

1. イントロダクション

セキュリティベンダーにとってマルウェアの解析は重要な意味を持つ。セキュリティベンダーは、マルウェアを解析することで、アンチウイルスソフトのシグネチャの作成や、新たな攻撃手法の把握などができるようになる。

マルウェアを迅速に解析し、脅威にタイムリーに対応するためには、適切な解析手法を用いることが重要である。マルウェア解析の手法に着目すると、「マルウェアを動作させる/動作させない」と「手動で行う/自動で行う」という 2 つの視点で分類でき、一般にマルウェアを動作させる解析手法を「動的解析」といい、動作させない解析手法を「静的解析」という。

未知のマルウェアを解析する場合には、通常、はじめに「自動」の「動的解析」を行う。「自動」の「動的解析」は、短時間でマルウェアの挙動を確認できるためである。

しかし、自動の動的解析だけで十分な解析結果が得られることは少ない。それは、自動の動的解析には、マルウェアが実際に動作した範囲でしか情報を得られない、自動で動的解析を行うシステム（以後、動的解析システムという）が収集する以上の情報が得られない、という欠点があるためである。

そのため多くの場合、次のステップとして、柔軟な情報収集ができる「手動」の動的解析や静的解析を行う。

だが、手動での解析は、マルウェアがパック（圧縮・難読化処理）されていた場合には大きな困難を伴う。マルウェアがパックされている割合は、2003 年では 20%[2]、2005 年では 55%[2]、2007 年では 80%[3]と年々増加している。セキュリティベンダーは急増する新種・亜種のマルウェアにも対応しなければならず、その全てを手動で解析することは困難である。

このような背景により、今日のセキュリティベンダーにとっては「自動」の「動的解析」でいかに多くの情報を収集し、手動での解析に要する時間を削減するかが重要になってきている。

しかし、従来の動的解析システムは、手動での解析に要する時間を削減するという目的に対して解析機能が不十分である。多くの動的解析システムは、取得できる情報は API の呼び出し履歴程度であり、カーネルモードで動作するマルウェアに対しては機能しない。加えて、他のプロセスに悪意のコードを注入するなどの方法で、プロセスを越えて拡散するマルウェアを解析することも困難である場合が多い。そのため、自動の動的解析でマルウェアの全容を理解することは難しく、手動での解析に多くの時間を割かなければならないことがほとんどである。

既存の研究の中には、仮想マシンやエミュレーター（以後、仮想環境という）を解析環境として用いることで、動的解析システムの解析機能を高めたものもある。これらの研究による動的解析システムは、マルウェアの実行を 1 命令単位で解析する機能を持ち、一部にはプロセスを越えて拡散するマルウェアを追跡し、解析するものもある。

だが、マルウェアには、仮想環境で動作させた場合その存在を検知し挙動を変えるものもあり、仮想環境に依存した動的解析システムでは本来の動作結果を得られないことがある。

そこで本研究では、仮想環境に依存せずマルウェアの実行を 1 命令単位で解析し、かつプロセスを越えて拡散するマルウェアを追跡し解析できる動的解析システム（以後、本システムという）の設計と実装を行い、その有効性を評価した。

本稿の構成は以下のとおりである。2 章では、従来の動的解析システムや関連研究が持つ問題点から本研究で解決すべき課題と目標を定義する。3 章では、それらの課題を解決する本システムの設計と実装を検討する。4 章では、本システムの有効性を実際のマルウェアやサンプルプログラムの解析を通して評価する。5 章では、評価結果を踏まえて本研究で解決すべき課題と目標がどのように達成されたかを結論付け、今後の展望を示す。

2. 目標

本章では、従来の動的解析システムや関連研究が持つ問題点から本研究で解決すべき課題と目標を定義する。まず従来の動的解析システムの課題を確認し、次いで、仮想環境を用いて従来の動的解析システムの課題を解決する複数の研究を確認する。仮想環境を用いた動的解析システムについては、その特徴的な機能と、仮想環境を用いることで新たに発生する課題を確認し、最後にこれらを踏まえて、本研究において解決すべき課題と目標を定義する。

2.1. 従来の動的解析システムにおける課題

従来の動的解析システムの中で広く利用されているものとして、Process Monitor[1]とSysAnalyzer[2]があげられる。これらの動的解析システムは、マルウェアの挙動を API フックや、Windows が提供するコールバック機構を用いて監視し表示する機能を持つ。これらの動的解析システムは、マルウェアがパックされていても解析が可能であり、またマルウェアにデバッガーとしてアタッチしないため、アンチデバッグ技法によって解析の妨害を受けることが少ない。

しかしながら、これらの動的解析システムには大きく 2 つの課題がある。

1 つは、従来の動的解析システムによって収集できる情報が少ないことである。従来の動的解析システムが収集する情報は、API の呼び出し履歴やコールバック機構がサポートするリソ

[1] http://technet.microsoft.com/en-us/sysinternals/bb896653
[2] http://labs.idefense.com/software/malcode.php

ースへのアクセス履歴などに限定されている。そのため、たとえばどのファイルにアクセスしたかの情報は得られても、どのような内容のアクセスを行ったかという情報は得られない。またマルウェアが、どのような順序で内部関数を実行したかという情報も得られないため、マルウェアの詳細な挙動や内部構造に関する情報などは手動での解析時に調査しなければならない。

従来の動的解析システムが持つ別の課題は、プロセスを越えて拡散するマルウェアへの対応力に乏しいことである。たとえば、マルウェアが他のプロセスに悪性コードを書き込み、スレッドを注入して実行した場合、Process Monitor でこれを追跡することは難しい。SysAnalyzer はこのような場合を追跡するが、SetWindowsHookEx 関数を用いた単純な DLL 注入や、バッチファイル等を利用した間接的なサブプロセスの起動など、多くのケースでは追跡できない。また、どちらの動的解析システムもカーネルモードコードを解析することはできず、マルウェアの全容を把握することは難しい。

これらの課題により、従来の動的解析システムだけでは十分な解析結果が得られず、手動での解析に多くの時間が費やされることで、結果として、マルウェアへの対応速度を低下させてしまっている。

関連研究として、仮想環境に依存しない動的解析システムの研究がいくつか存在する。PolyUnpack[1]はマルウェアの実行を 1 命令単位で解析するという手法を採用し、マルウェアの内部構造を解析しながらアンパック（パックされた状態の解除）処理を可能とした。しかしながら、この実装は Windows が提供するデバッグ API に依存しており、アンチデバッグ技法によってマルウェアから容易に検知、妨害されてしまう。Justin[14]はデバッグ API を用いることなくマルウェアの実行を解析し、アンパック処理を行うが、動的解析システムとしての機能はなく 1 命令単位での解析はできない。

2.2. 仮想環境を用いた研究とその問題点

仮想環境を用いた動的解析システムによって、収集できる情報の不足や、プロセスを越えて拡散するマルウェアへの対応力の不足といった課題を解決する研究がこれまでに複数存在する。

Anubis[3]は仮想環境を用いた動的解析システムであり、QEMU 環境下でマルウェアを実行し、その挙動を解析する。しかしながら、収集する情報は API の呼び出し等であり、命令単位の解析はしない[4]。一方、Ether[6]は Xen を用い、マルウェアの実行を命令単位で解析する機能を持つ。また Renovo[7]は、TEMU[8]を利用し、命令単位で解析する機能に加えて、プロセスを越えて拡散するマルウェアを追跡し解析する機能を持つ。

だが、マルウェアには、仮想環境で動作させた場合にその存在を検知し挙動を変えるものもあり、仮想環境に依存した動的解析システムでは本来の動作結果を得られないことがある。仮想環境の検知技法は、Peter Ferrie[9]ら多くの研究者によって具体的な方法が明らかにされており、Themida[10]など検知技法機能を埋め込むことができるパッカーも存在することから、仮想環境を検知するマルウェアは今後さらに一般化していくものと著者は予測している。

[3] http://anubis.iseclab.org/

2.3. 本研究の目標

これまでに論じてきた各動的解析システムの特徴を表 2-1 に示す。

表 2-1 動的解析システムの特徴

特徴	従来の動的解析システム		仮想環境を用いた動的解析システム	
	Process Monitor	PolyUnpack	Ether	Renovo
1 命令単位で解析できる(多くの情報を収集できる)	×	○	○	○
パックされたプログラムを解析できる	○	○	○	○
プロセスを越えて拡散するマルウェアに対応できる	×	×	×	○
カーネルモードコードを解析できる	×	×	○	○
アンチデバッグ技法の影響を受けにくい	○	×	○	○
仮想環境の検知技法の影響を受けない	○	○	×	×

これらの動的解析システムが持つ一連の問題点を踏まえ、ここで本研究の目標を定義し、解決すべき課題を明確化する。

2.3.1. 命令単位の解析機能の実現

より多くの情報を収集するため、マルウェアの実行を 1 命令単位で解析する機能を実現する。また、パックされたプログラムの解析を可能とする。これらにより、自動での動的解析によって、マルウェアの詳細な挙動や内部構造に関する情報を収集し、手動の解析に要する時間を削減することを可能とする。

2.3.2. プロセスを越えて拡散するマルウェアの自動解析の実現

プロセスを越えて拡散するマルウェアを自動で解析するために、マルウェアの影響を受けた

全てのスレッドを自動的に解析する機能を実現する。また、カーネルモードコードの解析機能を実装し、マルウェアの影響を受けたカーネルモードコードの実行を自動で解析する機能を実現する。これらにより、従来の動的解析システムを利用した場合より効率良くマルウェアの全容を把握することを可能とする。

2.3.3. アンチデバッグ技法に対する不可視性の実現

マルウェアによって本システムが検知されることを回避するために、デバッグ API の利用を避け、広く知られたアンチデバッグ技法に対する不可視性を実現する。これにより、アンチデバッグ技法を用いるマルウェアに対する動的解析を可能とする。

2.3.4. 仮想環境への依存性の排除

仮想環境の検知技法によって本システムが検知されることを回避するために、一連の機能を仮想環境に依存しない方法で実現する。これにより、仮想環境を検知するマルウェアに対する動的解析を可能とする。

3. 設計と実装

本章では、本システムの設計を検討し、その実装方法を示す。

3.1. 設計

仮想環境に依存せず、かつプロセスを越えて拡散するマルウェアを解析するために、本システムは Ring0 で動作するデバイスドライバー（egg.sys）として実装する。また、デバイスドライバーへ解析に有用な情報を与える補助プログラムとして、対話的なユーザーモードプログラム（eggclient.exe）を実装する。これらを踏まえた本システムの動作イメージを図 3-1 に示す。

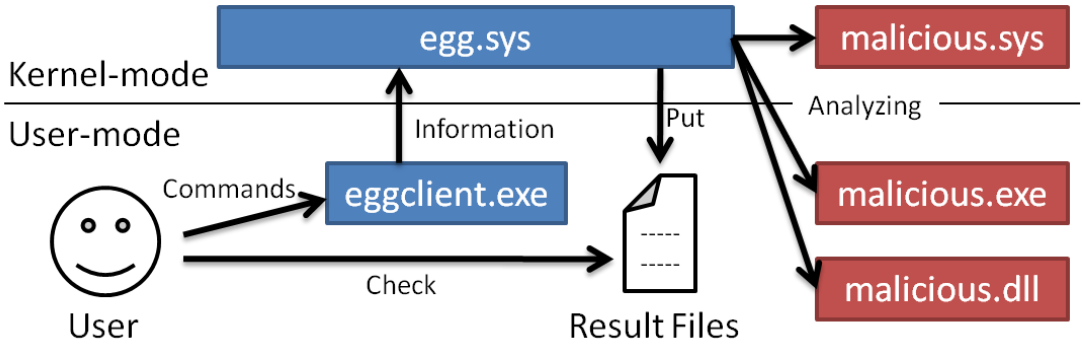

図 3-1 本システムの動作イメージ

命令単位の解析を実現するために、プロセッサーが持つページ保護機能とシングルステップ実行機能を組み合わせた、プログラムの実行制御手法を用いる。本手法は Amit Vasudevan[11]らによって考案されたものあり、カーネルモードコードの解析に適用でき、デバッグ API を用

いず、仮想環境であるか否かを問わず利用できる。

プロセスを越えて拡散するマルウェアを自動的に解析するために、「汚染追跡（Taint tracing）」の概念を用いる。「汚染追跡」は、「汚染」として記録された要素の伝搬を追跡することで、汚染の影響範囲を自動的に管理するものである。本システムでは、マルウェアの影響を受けた仮想メモリー（以後、単に「メモリー」という）、スレッドおよびファイルを汚染の対象とし、汚染されたメモリーとファイルの実行を解析対象とする。

3.2. 実装

3.2.1. 命令単位での解析

本システムが実装する命令単位の解析機能の動作概要を図 3-2 に示す。なお、より詳細な内容については参考文献[11]を参照していただきたい。

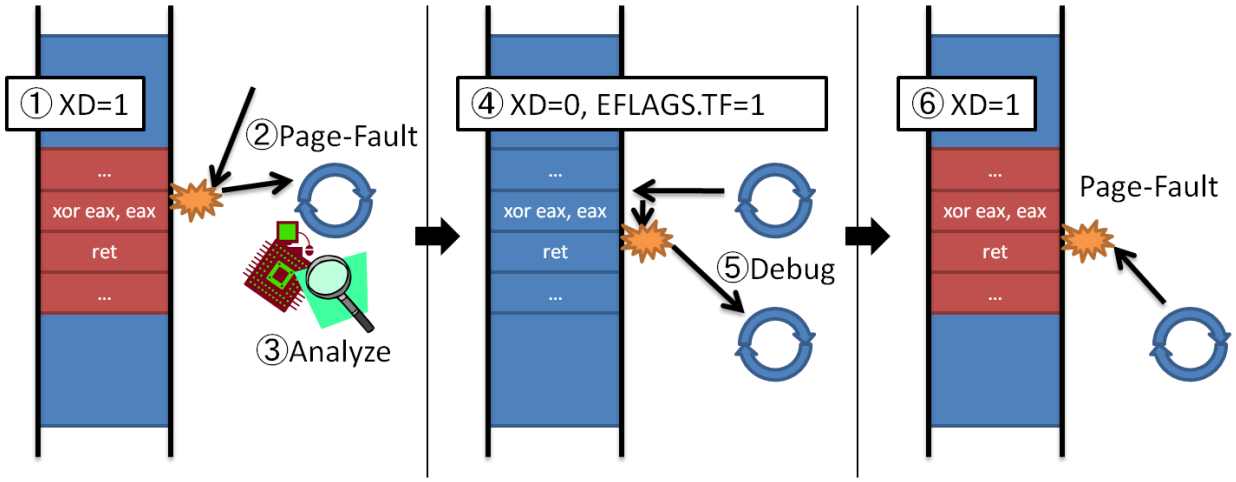

図 3-2 命令単位の解析機能の動作概要

① 実行の解析が必要なメモリー領域に対し、実行禁止属性を設定する。これは当該メモリーに対応する PTE（Page Table Entry）の XD（Execute-disable）ビットを 1 にすることで実現する。

② 何らかのスレッドから当該メモリーの実行が要求されると、プロセッサーによりページフォルト例外が引き起こされる。

③ 本システムはあらかじめページフォルト例外の例外ハンドラーを書き換えておき、書き換えた後の例外ハンドラーで、レジスターやメモリーを解析する。

④ その後、メモリーの実行禁止属性を解除（実行可能に）するとともに、EFLAGS の TF（Trap Flag）ビットに1を設定し、プロセッサーのシングルステップ実行機能を有効にする。

⑤ 例外ハンドラーから戻ると、スレッドは 1 命令だけ実行したのち、プロセッサーによってデバッグ例外が引き起こされる。

⑥ 本システムはここでもあらかじめデバッグ例外の例外ハンドラーを書き換えておき、書き換えた後の例外ハンドラーで、ふたたびメモリー領域に対して実行禁止属性を設定する[4]。

一連のステップにより、プログラムの実行を命令単位で解析できるようになる。

本システムではこの解析で得られたプログラムの実行履歴をもとにして、関数間の遷移情報と分岐トレース情報[5]を生成する。また、命令単位でレジスターやメモリーを解析できることを利用して、特定の命令や API に対する入力引数と出力引数の両方を解析する機能を実装する。API の引数を解析する機能については「API の引数解析機能の実装」で詳しく述べる。

3.2.2. マルウェアによる汚染の追跡

本システムにおける「汚染」の対象要素は、メモリー、スレッドおよびファイルであり、それぞれを「汚染メモリー」「汚染スレッド」「汚染ファイル」という。

汚染メモリーは、汚染スレッドによって変更されたメモリー領域か、汚染ファイルがメモリ

[4] 実行禁止属性の設定は必須である。シングルステップ実行機能のみを使用し、実行可能な状態を維持した場合、他のスレッドによって当該メモリー領域が実行された際にページフォルト例外が発生せず、解析できない。
[5] EIP が「EIP＋実行した命令の長さ」以外の値になった場合に変更前後の EIP を記録したもの。

一上にマップされた領域である。汚染スレッドは、汚染メモリーを一度でも実行したことのあるスレッドである。汚染ファイルは、汚染スレッドによって変更されたファイルである。

本システムの汚染追跡の動作概要を図 3-3 に示す。

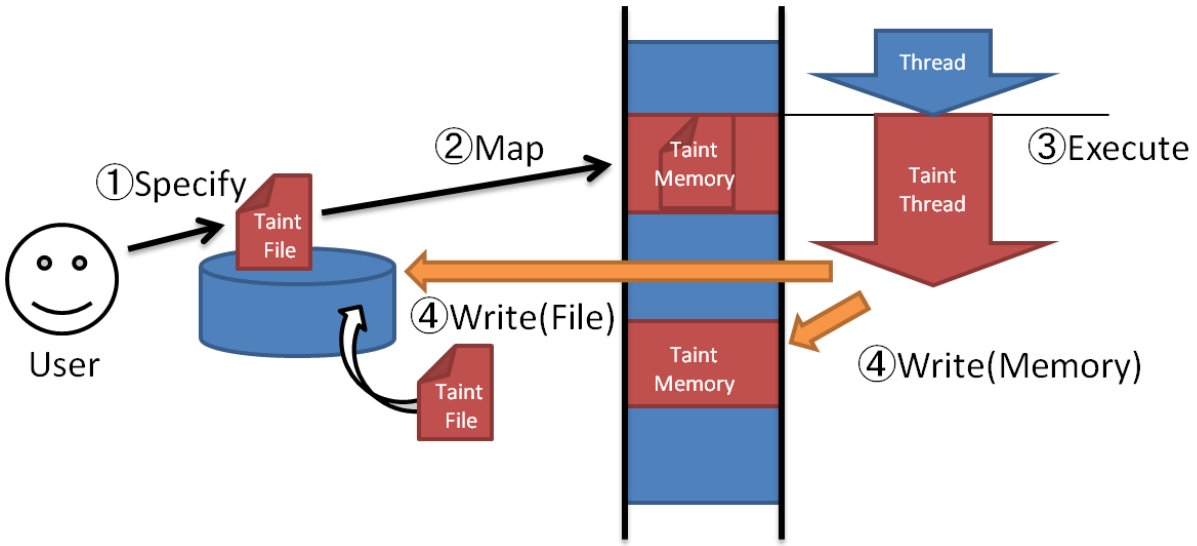

図 3-3 汚染追跡の動作概要

① はじめに、本システムの利用者が解析対象のマルウェアファイルを汚染ファイルとして指定する。

② その後、マルウェアファイルの実行などによりこの汚染ファイルがメモリー上にマップされると、マップされた領域は汚染メモリーとして扱われる。

③ いずれかのスレッドがこの汚染メモリーを実行すると、実行したスレッドは汚染スレッドとして扱われ解析の対象となる。

④ 汚染スレッドによって変更されたファイルやメモリーがそれぞれ汚染ファイル、汚染スレッドとして扱われる。

本システムは、汚染メモリーの実行を「命令単位での解析」で述べた手法により解析する。

次に、メモリーの汚染とファイルの汚染の実装方法を示す。

メモリーの汚染

メモリーの汚染の追跡は、全てのメモリーに書き込み禁止属性を設定し、メモリーへの書き込み処理をトラップすることで実装する。以下にこの流れを示す。

① 汚染スレッドが動作するプロセスのメモリー領域全体に対し、書き込み禁止属性を設定する。これは PTE の R/W(Read/Write)ビットを 0 にすることで実現する。

② 書き込みが禁止されたメモリーへの書き込みが要求されると、プロセッサーによりページフォルト例外が引き起こされる。

③ 本システムは、あらかじめページフォルト例外の例外ハンドラーを書き換えておく。書き換えた後の例外ハンドラーでは、書き込みが要求されたアドレスを汚染メモリーとして記録するとともに、メモリーの書き込み禁止属性を 1 命令実行される間だけ解除(書き込み可能に)する。

本システムでは、上記処理を汚染スレッドからの書き込みに対してのみ行うために、Windows 内部関数である SwapContext 関数および、KiSwapProcess 関数をあらかじめ書き換える。

SwapContext 関数は実行スレッドの切り替え（スレッド・プリエンプション）を行う関数である。本システムが書き換えた後の SwapContext 関数では、汚染スレッドへ実行が切り替わった際に、メモリーの書き込み禁止属性を設定し、逆に汚染スレッドから汚染されていないスレッドに実行が切り替わった際には、書き込み禁止属性を元の値に復元する。

KiSwapProcess 関数はスレッドが動作するメモリー空間を他のメモリー空間に切り替える関数で、WriteProcessMemory 関数などプロセス間のメモリー操作の際に利用される。本システムが書き換えた後の KiSwapProcess 関数では、汚染スレッドが他のメモリー空間を利用するように要求された場合、切り替え先のメモリー空間に対して書き込み禁止属性を設定するとともに、切り替え元のメモリー空間に対しては書き込み禁止属性を元の値に復元する。

一連の内部関数の書き換えにより、汚染スレッドが変更したメモリーが自動的に追跡される

ようになり、WriteProcessMemory 関数などを用いたプロセス間のマルウェアの拡散に対応できるようになる。

ファイルの汚染

ファイルの汚染の追跡は、Windows が提供するファイルシステムフィルタードライバーの仕組みを用い、汚染スレッドのファイル I/O を監視することで実装する。また、汚染ファイルがメモリー上にマップされたことの検知は、PsSetLoadImageNotifyRoutine 関数を用いる。

これにより、汚染スレッドにより生成または変更された実行ファイルの起動や、デバイスドライバー、DLL の実行に対する自動的な解析が可能となる。

カーネルモードに対する汚染追跡の動作制限

本システムは、カーネルモードにおける汚染追跡の動作に以下の制限をもつ。

- 汚染スレッドによるカーネルメモリーの書き込みが発生した場合、そのメモリーを汚染メモリーとして扱わない。
- スレッドがカーネル空間の汚染メモリーを実行した場合、そのスレッドを汚染スレッドとして扱わない。

前者は、汚染を追跡するためにカーネルメモリーを書き込み禁止にすると、本システムの動作も影響を受けてしまう場合があるための制限である。

後者は、カーネルモードの汚染メモリーを実行したスレッドを汚染スレッドとすることが、実用上適切ではないための制限である。具体的な例をあげる。たとえば、マルウェアがファイルシステムフィルタードライバーをインストールした場合、全てのスレッドが、ファイル I/O 時にこのフィルタドライバーのコードを実行する。このとき、フィルタドライバーのコードを実行したスレッドを汚染スレッドとして扱った場合、無数のスレッドが汚染スレッドとなって

しまい、実用上不要な情報まで解析結果に含まれてしまう。このような問題を回避するため本制限を設けた。

3.2.3. 不可視性の確保

本システムは、解析対象からの不可視性を確保し、マルウェア本来の動作を解析できるようにするため、ユーザーモードで動作するプログラムに対して透過的に動作する。具体的には、デバッグ API を使用せず、レジスターやメモリーの内容をユーザーモードから可視な形で変更しない。

以下に、これらを実現するうえで課題となる「EFLAGS の管理」と「API の引数解析機能の実装」について述べる。

EFLAGS の管理

「命令単位での解析」で述べた手法は、EFLAGS の値がユーザーモードから可視であることに起因する問題がある。本手法はデバッグ例外を発生させる目的で EFLAGS の TF ビットの値を変更する。しかし EFLAGS の値はユーザーモードコードから pushf 命令や例外処理などを用いて取得可能であり、EFLAGS の値を検査されることで本システムの存在を検知される可能性がある。同様に、EFLAGS の TF ビットの値は popf 命令や icebp 命令などを用いて変更できるため、本システムの正常な動作が妨げられる可能性がある。

これらの問題に対処するために、本システムでは解析対象からの EFLAGS の取得や変更を管理する。本システムは、解析中に pushf 命令、popf 命令、icebp 命令が出現した場合、その実行結果を改ざんし、本システムが存在しないように見せかける。また、例外が発生した場合も、Windows の内部関数である KiDispatchException 関数をあらかじめ書き換え、例外情報

に含まれる EFLAGS の値を変更することで、本システムが存在しないように見せかける。

API の引数解析機能の実装

本システムは、その利便性を高めるために、汚染メモリーからの API の呼び出しを検知し、呼び出された API の引数を解析する機能を実装する。

本機能もユーザーモードから透過的に動作する。API の呼び出しの検知は、ブレークポイントなどユーザーモードから可視な手法は用いず、命令単位の解析機能と連携して、スレッドの EIP が API のエントリーポイントに設定されたか否かによって判断する。API のエントリーポイントの取得は、メモリー上に PE ファイルがマップされたとき、カーネルモードから PE ファイルのエクスポートアドレステーブルを走査して取得する。なお、引数の数や型については、利用者があらかじめ補助プログラムを用いてデバイスドライバーに指定する。

本機能を実装する際に問題となるのは、引数の解析処理が「命令単位での解析」で述べた例外ハンドラー内では安全に行えないということである。Windows は、例外ハンドラー内からのさらなるページフォルトを許可しない[6]。そのため、たとえば、HANDLE 型の引数を解析するために NtQueryObject 関数を用いる場合にも、NtQueryObject 関数にページフォルトを発生させる可能性があるため例外ハンドラー内では利用できないという問題が発生する[7]。

また、引数の解析処理に API フックを用いることもできない。API フックを行うには、API のエントリーポイントやインポートアドレステーブルなど、何らかのメモリーを変更しなければならない。加えてフック関数を配置し実行する必要もあり、ユーザーモードから可視な影響

6 Windows は、プロセッサーを適切な状態に設定しないまま、例外ハンドラー内でページフォルト例外を発生させると、システムストップを起こすように実装されている。
7 NtQueryObject 関数は、PASSIVE_LEVEL でのみ利用可能であると明示されている。これは本関数によってページフォルトが発生する可能性があることを示している。

となってしまう。

そこで本システムでは、ユーザーモードから不可視なまま、安全なタイミングで API の引数を解析する方法を考案した。

具体的な実装方法を述べる。まず、API が呼び出された際の例外ハンドラーでは引数を解析せず、代わりに当該スレッドの EIP の指すアドレスをループ命令（0xfeeb）に書き換える。このとき、書き換え操作によるページフォルトの発生を避けるために、IoAllocateMdl 関数を用いて EIP のアドレスをカーネルメモリーにマッピングしたうえで書き換える。その後、例外ハンドラーから制御を戻すと、当該スレッドは API のエントリーポイントでループ状態となり、引数やレジスターが変更されずに保持される。この間に、本システムは安全なタイミングで引数を解析し、その後、ループ命令から元の命令に復元し、スレッドのループを解除する。

本手法はユーザー空間のメモリーを 0xfeeb に書き換えるが、ユーザーモードからは不可視である。0xfeeb に書き換えられたスレッドは、ループを実行するため書き換えられたかどうかを判断できない。また他のスレッドからは、SwapContext 関数によって他のスレッドに切り替わる際に、0xfeeb を元の命令に復元することで、書き換えられている状態を確認できなくなる。

本手法によって、ユーザーモードから不可視なまま API の引数解析が可能となる。

4. 評価と結果

本システムのコンセプト実装である egg を用いて、実際のマルウェア 1 体と、複数のサンプルプログラムを解析し、本システムを評価した。評価は大きく分けて以下の項目について行った。

① 解析機能と性能

② プロセスを越えて拡散するマルウェアへの対応力

③ アンチデバッグ技法に対する不可視性

全ての評価は 32 ビットの Windows XP SP3 を用い、サンプルプログラムは Visual Studio 2010 でコンパイルした。

4.1. 解析機能と性能

本システムの解析機能および性能を評価した。

はじめに、基本的な解析機能の評価のため、実際のマルウェアを解析した。この解析結果を分析することで、本システムの解析機能の有効性を評価した。次に Windows に含まれる ipconfig.exe を、通常の状態とパックした状態のそれぞれで解析し、両者を比較することで、パックされたプログラムに対する本システムの有効性を評価した。また、性能を評価するため、単純なループプログラムを用いて、通常環境下と本システムによる解析下での実行速度を比較した。

4.1.1. マルウェアの解析

評価方法

本システムの解析機能の有効性を評価するため、実際のマルウェアを解析した。

本システムによる解析を通して得た関数間の遷移情報、分岐トレース情報および API の引数情報を用いて未知のマルウェアを調査し、それらがどのように役立つのかを確認することで、本システムの解析機能の有効性を評価した。

評価結果

解析結果の関数間の遷移情報から、Graphviz[8]を用いて生成したコールグラフの一部を図 4-1 に示す。

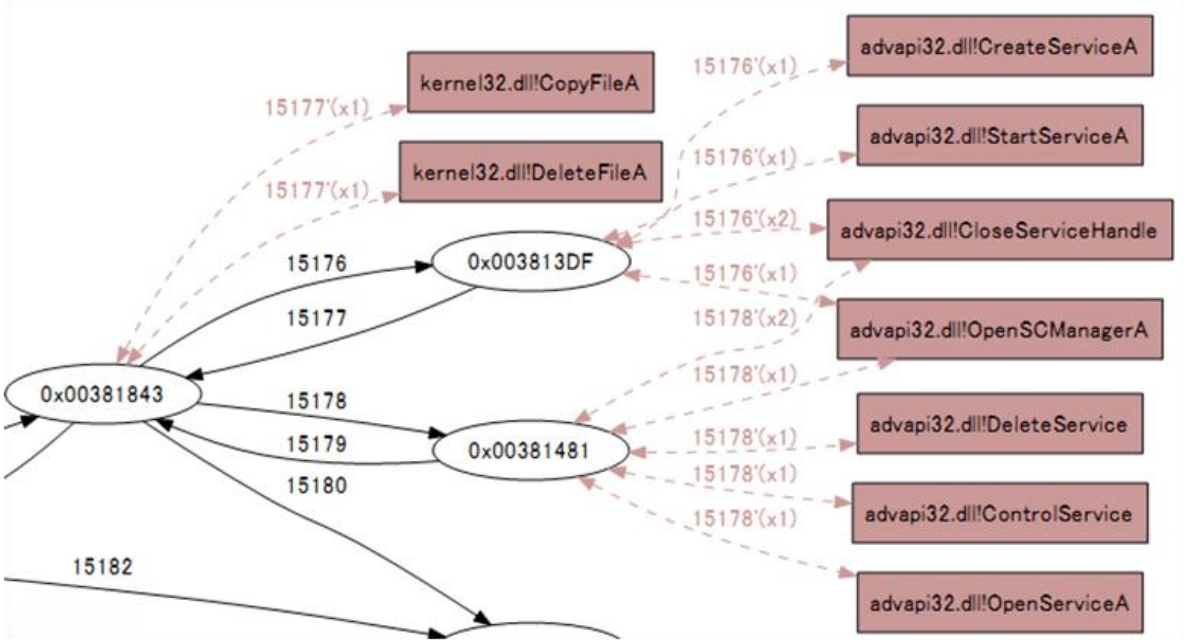

図 4-1 サービス操作が行われている箇所のコールグラフ

[8] http://www.graphviz.org/

図中、線上の番号は実行の遷移番号を示し、楕円と内部の数字は内部関数とそのアドレスを示している。図 4-1 からは、内部関数 0x00381843 から別の内部関数 0x003813DF と 0x00381481 を呼び出しており、0x003813DF では CreateService 関数や StartService 関数を用いてサービスを開始し、0x00381481 では DeleteService 関数を用いてサービスを削除していることが読み取れた。また遷移番号から、サービスの開始（15176）、ファイル操作（15177）、サービスの削除（15178）の順に実行されたことが読み取れ、マルウェアが、サービス開始後にファイルを削除し、さらにサービスも削除することで自身の痕跡を隠そうとしたことが推測できた。

次に、この内部関数 0x003813DF に分岐トレース情報を対応づけたものを図 4-2 に示す。

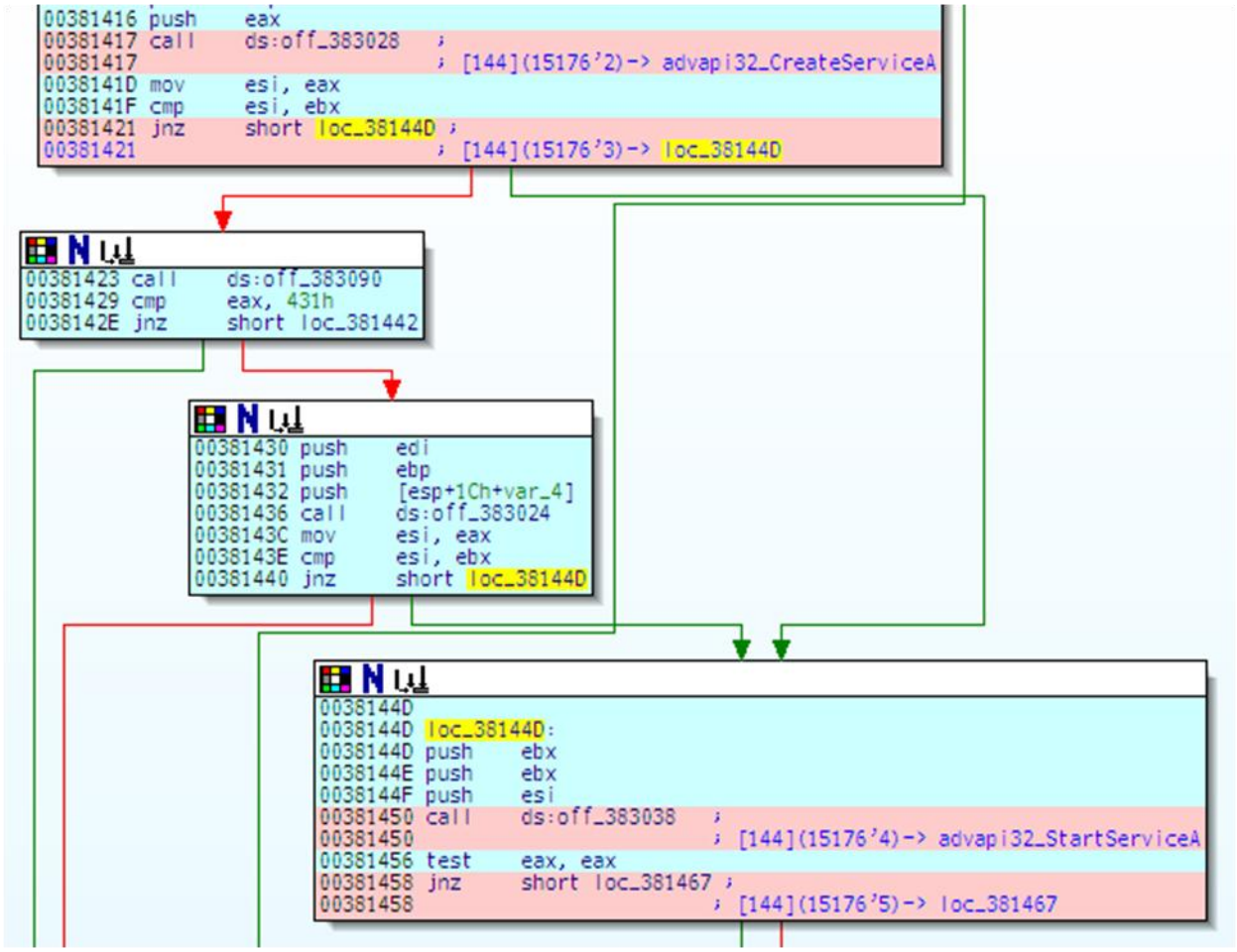

図 4-2 分岐トレース情報と逆アセンブル出力の対応[9]

図中の桜色の箇所が、分岐が発生した箇所である。図 4-2 からは、内部関数 0x003813DF が解析時にどのような分岐で実行したかや、各 call 命令がどの API を呼び出したか、実行されなかった命令はどれか、といった情報を得ることができた。

次に、これらとは別の箇所における API の引数情報をリスト 4-1 に示す。これは特徴的な書き込み操作を記録した箇所である。

[9]青色の文字は『[スレッド ID](遷移番号'分岐番号)->分岐先のアドレスまたはシンボル名』をそれぞれ示している。

```
...
[ 1632:  144] 0038163B call to kernel32.dll!CreateFileA(
[ 1632:  144]     Arg 1 : ¥¥.¥C:,
[ 1632:  144]     Arg 2 : C0000000(-1073741824)
[ 1632:  144] )
[ 1632:  144] 0038163D returned from kernel32.dll!CreateFileA() => 00000064 = File : ¥Device¥HarddiskVolume1
[ 1632:  144] 00381667 call to kernel32.dll!ReadFile(
[ 1632:  144]     Arg 1 : 00000064 = File : ¥Device¥HarddiskVolume1,
[ 1632:  144]     Arg 3 : 00000800(2048)
[ 1632:  144] )
[ 1632:  144] 0038166D returned from kernel32.dll!ReadFile(
[ 1632:  144]     Arg 2 : 0012F184 - 0012F983 is dumped as ¥??¥c:¥desktop¥eval411.exe_01632¥00144_kernel32.dll!ReadFile_Arg02_20110510_233154.328.bin
[ 1632:  144] ) => 00000001(1)
[ 1632:  144] 00381818 call to kernel32.dll!WriteFile(
[ 1632:  144]     Arg 1 : 00000064 = File : ¥Device¥HarddiskVolume1,
[ 1632:  144]     Arg 2 : 00960000 - 00960FFF is dumped as ¥??¥c:¥desktop¥eval411.exe_01632¥00144_kernel32.dll!WriteFile_Arg02_20110510_233206.594.bin,
[ 1632:  144]     Arg 3 : 00001000(4096)
[ 1632:  144] )
[ 1632:  144] 0038181E returned from kernel32.dll!WriteFile() => 00000001(1)
...
```

リスト 4-1 書き込み操作が読み取れる API の引数情報

ここからは、CreateFileA 関数に “¥¥.¥C:” が渡され、その結果、ディスクドライブを指すファイルハンドルが返されたこと、そのファイルハンドルを用いて WriteFile 関数等を呼び出したことが読み取れた（赤色の文字による強調箇所）。このことから、マルウェアが、ファイル名を用いずディスクに直接書き込むことで、アンチウイルスソフトによる検知を回避しようとしていることが推測できた。

本システムの引数解析機能は、WriteFile 関数の第 2 引数の例のように引数の内容をファイルに保存できる。リスト 4-1 に青色の文字で強調した箇所は、本機能によって引数の内容がファイルに保存されたことを示している。保存されたファイルの内容を図 4-3 に示す。

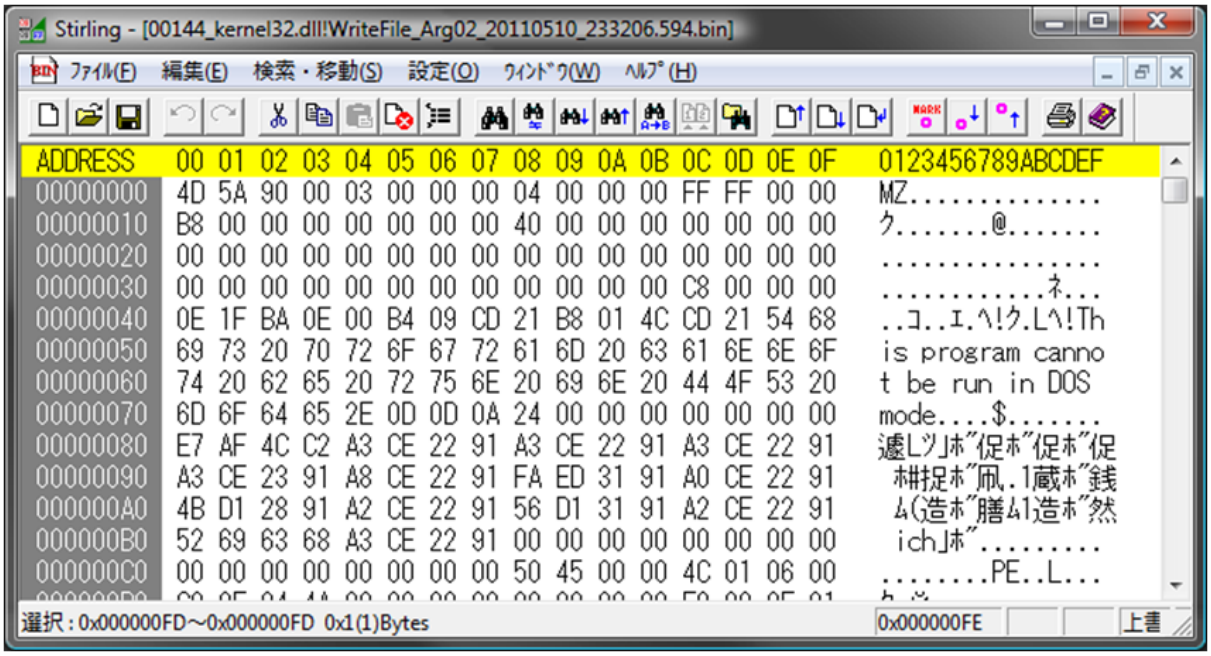

図 4-3 ファイルに保存された WriteFile 関数の第 2 引数

保存されたファイルの内容から、WriteFile 関数の第 2 引数には PE ファイルが渡されたことが読みとれ、マルウェアがディスクドライブに直接 PE ファイルを書き込んだことがわかった。

4.1.2. パックされたプログラムの解析

評価方法

パックされたプログラムに対する本システムの有効性を評価するため、通常の状態のプログラムと、パックした状態のプログラムを解析した。

Windows XP SP3 に標準で含まれる ipconfig.exe を、それぞれの状態で実行し、プロセスが終了するまでの所要時間および解析結果のコールグラフを比較することで、解析対象がパックされていた場合の影響を評価した。パッカーは、マルウェアによって利用されることがもっとも多いもの[2][3][4]として UPX 3.07[10]を使用した。

評価結果

評価の結果、プロセスが実行を終了するまでの所要時間は、通常の場合は 4000 ミリ秒程度であるのに対し、パックされている場合 50,000 ミリ秒程度となった。このことから、プログラムがパックされている場合には解析に要する時間が増加することがわかった。

次に、通常の場合とパックされている場合の双方のコールグラフを図 4-4、図 4-5 に示す。

[10] http://upx.sourceforge.net/

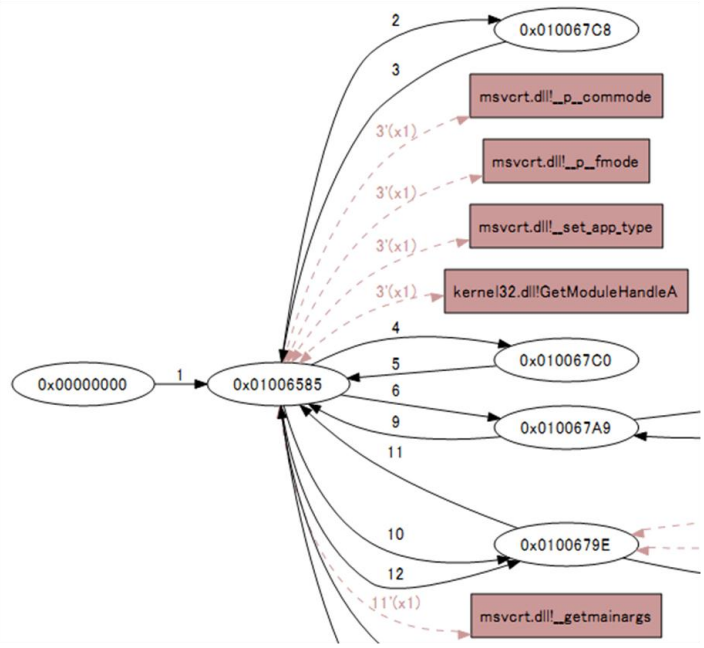

図 4-4 実行開始直後の ipconfig(通常)のコールグラフ

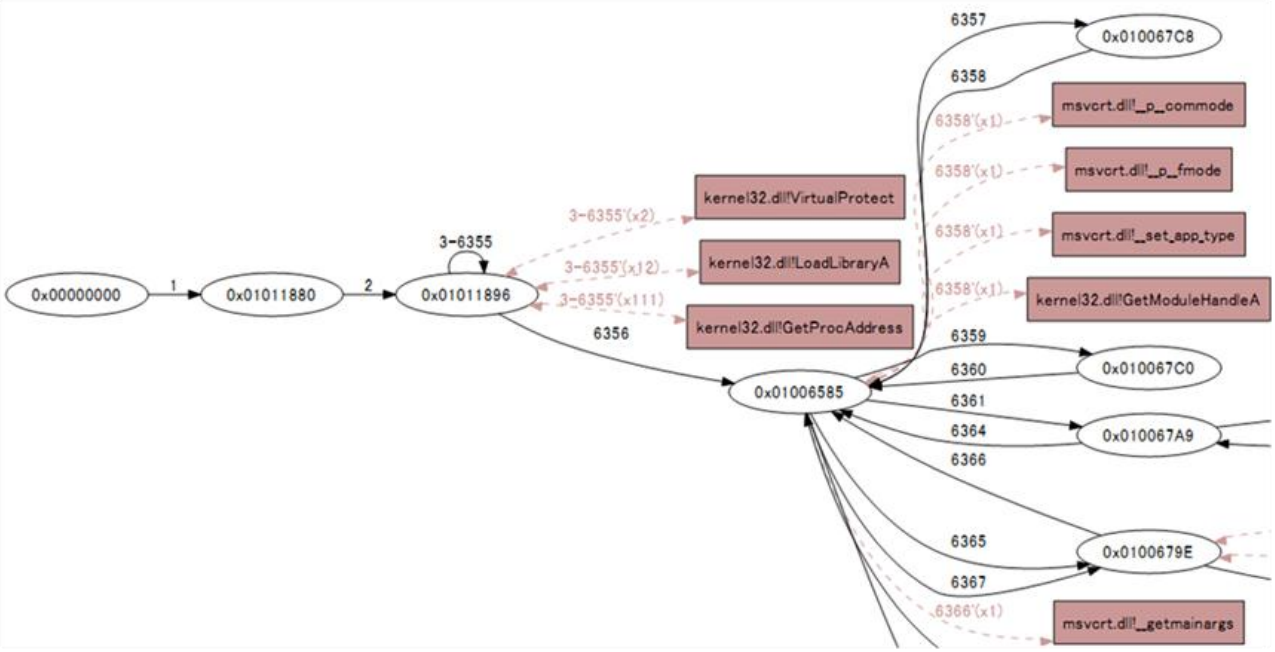

図 4-5 実行開始直後の ipconfig(UPX パック済)のコールグラフ

それぞれのコールグラフを比較すると、パックされたものは展開処理が実行されたことでプログラム実行開始直後の内容が異なるものになったが、その後は同じ内容となった。

4.1.3. ループプログラムの解析

評価方法

本システムの性能を評価するため、ループプログラム(eval413.exe)を解析した。

ループプログラムは、指定された秒数だけ空の for ループを実行し、ループの実行回数を計測するプログラムである。このプログラムを用いて、本システムが動作していない状態と、本システムによって解析されている状態とで、1 秒間に実行できるループの回数を比較して性能を評価した。

評価結果

ループプログラムをそれぞれ 5 回実行した結果の平均値を表 4-1 に示す。

表 4-1 ループの実行回数の比較

項目	ループの実行回数	本システムが動作していない状態に対する倍率
本システムが動作していない状態	900,180,667	1
本システムによって解析されている状態	3,155	285,319

本システムによって解析されている場合のループの実行回数は、通常の 30 万分の 1 程度となった。

4.2. プロセスを越えて拡散するマルウェアへの対応力

本システムの、プロセスを越えて拡散するマルウェアへの対応力を評価した。

はじめに、他のプロセスにコードを注入するサンプルプログラムを解析し、プロセス間で拡散するマルウェアへの対応力を評価した。次に、動的に生成したデバイスドライバーをロードするサンプルプログラムを解析し、ファイルを用いて拡散するマルウェアへの対応力を評価した。

4.2.1. 他のプロセスにコードを注入するサンプルプログラムの解析

評価方法

プロセス間で拡散するマルウェアへの本システムの対応力を評価するため、他のプロセスにコードを注入するサンプルプログラム（eval421.exe）を解析した。

本サンプルプログラムは、WriteProcessMemory 関数、CreateRemoteThread 関数等を用いて他のプロセスにコードを注入して実行するプログラムである。本手法は、マルウェアによって一般的に利用されるものである。本サンプルプログラムを解析対象として実行し、explorer.exe にコードを注入して実行させ、explorer.exe に注入されたコードも解析できるか否かを確認することで、典型的なマルウェア拡散手法への対応力を評価した。

評価結果

サンプルプログラムの解析ログをリスト 4-2 に示す。

```
...
1:47:39.760       INF  1328   796 eval421.exe     Call/Jmp executing on 004011EF to 7C802213(   34'), as kernel32.dll!WriteProcessMemory
1:47:40.244       INF  1328   796 eval421.exe     Call/Jmp executing on 0040120F to 7C8104CC(   34'), as kernel32.dll!CreateRemoteThread
1:47:51.541       INF  1328   796 eval421.exe     Call/Jmp executing on 00401224 to 78AB1EC6(   34'), as msvcr100.dll!__iob_func
1:47:51.541       INF  1328   796 eval421.exe     Call/Jmp executing on 0040122E to 78B03FB2(   34'), as msvcr100.dll!fprintf
1:47:51.556       INF  1328   796 eval421.exe     Call/Jmp executing on 00401495 to 78AC8040(   37'), as msvcr100.dll!exit
1:47:51.931       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE0012 to 7C872451(    1'), as kernel32.dll!AllocConsole
1:47:52.244       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE0037 to 7C812FD9(    1'), as kernel32.dll!GetStdHandle
1:47:52.244       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE003F to 7C81CC5D(    1'), as kernel32.dll!WriteConsoleA
1:47:52.244       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE004C to 7C802446(    1'), as kernel32.dll!Sleep
1:47:53.291       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE0037 to 7C812FD9(    1'), as kernel32.dll!GetStdHandle
1:47:53.291       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE003F to 7C81CC5D(    1'), as kernel32.dll!WriteConsoleA
1:47:53.291       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE004C to 7C802446(    1'), as kernel32.dll!Sleep
1:47:54.353       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE0037 to 7C812FD9(    1'), as kernel32.dll!GetStdHandle
1:47:54.353       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE003F to 7C81CC5D(    1'), as kernel32.dll!WriteConsoleA
1:47:54.353       INF  1612  1292 explorer.exe    Call/Jmp executing on 00CE004C to 7C802446(    1'), as kernel32.dll!Sleep
...
```

リスト 4-2 explorer にコードを注入するプログラム eval421.exe の解析ログ（抜粋）

このリストから、eval421.exe が WriteProcessMemory 関数、CreateRemoteThread 関数を呼び出し explorer.exe にコードを注入した後、explorer.exe の関数呼び出しが記録され始めており、注入されたコードを解析できたことが読み取れた。これは、サンプルプログラムが注入したコードが、プロセスを越えて汚染メモリーとして扱われ、汚染メモリーを実行したスレッドが、汚染スレッドとして解析の対象になったことによるものである。

4.2.2. 動的に生成したデバイスドライバーをロードするサンプルプログラムの解析

評価方法

ファイルを用いて拡散するマルウェアへの対応力を評価するため、動的に生成したデバイスドライバーをロードするサンプルプログラム（eval422.exe）を解析した。

本サンプルプログラムは、Windows に標準で含まれる Beep.sys を独自のデバイスドライバーに上書きしたうえで、Beep サービスを再起動させることで独自のデバイスドライバーをロードさせるプログラムである。サンプルプログラムは、Beep サービスの再起動に StartService 関数や ControlService 関数を利用せず、Windows に標準で含まれる net コマンドを利用する。これは従来の動的解析システムでは、正常なプログラムを経由した間接的な処理の解析が難しいことを受けて、マルウェアが利用することの多い手法である。

本サンプルプログラムを解析対象として実行し、上書きされた後の Beep.sys の実行を解析できるか否かを確認することで、ファイルを用いて拡散するマルウェアへの対応力を評価した。

評価結果

サンプルプログラムの解析ログをリスト 4-3 に示す。また、これに対応するサンプルプログラム、および独自のデバイスドライバーのソースコードを抜粋したものをリスト 4-4、リスト 4-5 に示す。

```
...
2:00:16.666  INF   196  1024 eval422.exe     Call/Jmp executing on 004010E2 to 7C810800(   30'), as kernel32.dll!CreateFileW
2:00:16.666  INF   196  1024 eval422.exe     Call/Jmp executing on 00401106 to 7C810E27(   30'), as kernel32.dll!WriteFile
2:00:16.963  INF   196  1024 eval422.exe     Call/Jmp executing on 0040110F to 7C809BE7(   30'), as kernel32.dll!CloseHandle
2:00:16.978  INF   196  1024 eval422.exe     Call/Jmp executing on 00401122 to 78B056B4(   30'), as msvcr100.dll!printf
2:00:16.978  INF   196  1024 eval422.exe     Call/Jmp executing on 0040112F to 78B027BA(   30'), as msvcr100.dll!system
2:00:26.994  INF   196  1024 eval422.exe     Call/Jmp executing on 00401136 to 78B056B4(   30'), as msvcr100.dll!printf
2:00:27.041  INF   196  1024 eval422.exe     Call/Jmp executing on 0040113D to 78B027BA(   30'), as msvcr100.dll!system
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D0781F to 8052D8B4(    4'), as ntkrnlpa.exe!DbgPrintEx
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D0783A to 8053079C(    4'), as ntkrnlpa.exe!RtlInitUnicodeString
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D07845 to 8053079C(    4'), as ntkrnlpa.exe!RtlInitUnicodeString
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D0786A to 80577912(    4'), as ntkrnlpa.exe!IoCreateDevice
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D0787D to 80575BC2(    4'), as ntkrnlpa.exe!IoCreateSymbolicLink
2:00:35.619  INF     4  1884 System         Call/Jmp executing on F8D0789E to 805D313E(    4'), as ntkrnlpa.exe!PsCreateSystemThread
2:00:35.619  INF     4  1416 System         Call/Jmp executing on F8D076CF to 8052D8B4(    1'), as ntkrnlpa.exe!DbgPrintEx
2:00:35.619  INF     4  1416 System         Call/Jmp executing on F8D076E9 to 804FC87A(    1'), as ntkrnlpa.exe!KeDelayExecutionThread
2:00:35.822  INF   196  1024 eval422.exe     Call/Jmp executing on 00401144 to 78B056B4(   30'), as msvcr100.dll!printf
2:00:35.838  INF   196  1024 eval422.exe     Call/Jmp executing on 0040114B to 78B056B4(   30'), as msvcr100.dll!printf
2:00:35.838  INF   196  1024 eval422.exe     Call/Jmp executing on 0040115C to 7C82F87B(   30'), as kernel32.dll!CopyFileW
2:00:35.853  INF   196  1024 eval422.exe     Call/Jmp executing on 004012FB to 78AC8040(   31'), as msvcr100.dll!exit
2:00:36.744  INF     4  1416 System         Call/Jmp executing on F8D076CF to 8052D8B4(    1'), as ntkrnlpa.exe!DbgPrintEx
2:00:36.744  INF     4  1416 System         Call/Jmp executing on F8D076E9 to 804FC87A(    1'), as ntkrnlpa.exe!KeDelayExecutionThread
2:00:37.744  INF     4  1416 System         Call/Jmp executing on F8D076CF to 8052D8B4(    1'), as ntkrnlpa.exe!DbgPrintEx
...
```

リスト 4-3 動的にデバイスドライバーをロードするプログラム eval422.exe の解析ログ(抜粋)

```
static void RestartBeep()
{
    printf("Stopping Beep driver¥n");
    system("net stop beep  > NUL 2>&1");
    printf("Starting Beep driver¥n");
    system("net start beep > NUL 2>&1");
}
```

リスト 4-4 サンプルプログラム(eval422.exe)のソースコード(抜粋)

```
EXTERN_C NTSTATUS DriverEntry(
    __in PDRIVER_OBJECT DriverObject,
    __in PUNICODE_STRING  RegistryPath)
{
    PAGED_CODE();
    UNREFERENCED_PARAMETER(RegistryPath);
    DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_ERROR_LEVEL, "%s¥n", __FUNCSIG__);

    UNICODE_STRING DeviceName,Win32Device;
    PDEVICE_OBJECT DeviceObject = NULL;
    RtlInitUnicodeString(&DeviceName,L"¥¥Device¥¥Driver0");
    RtlInitUnicodeString(&Win32Device,L"¥¥DosDevices¥¥Driver0");
// ...
```

リスト 4-5 独自のデバイスドライバーのソースコード(抜粋)

リスト 4-3 からは、サンプルプログラムの system 関数の実行後、PID4 の System による DbgPrintEx 関数の呼び出しが記録されており、上書きされた後の Beep.sys の実行を解析できたことが読み取れた。これは、サンプルプログラムが Beep.sys に書き込みを行ったことで、Beep.sys が汚染ファイルとして扱われ、その汚染ファイルをマップしたメモリーが汚染メモリーとして解析の対象になったことによるものである。

4.3. アンチデバッグ技法に対する不可視性

評価方法

本システムのアンチデバッグ技法に対する不可視性を評価するため、複数のアンチデバッグ技法を実装したサンプルプログラム（eval43.exe）を解析し、それによって本システムが検知されるか否かを調査した。

評価に用いるアンチデバッグ技法は、その実装方法が特に広く知られているものを選択した。評価に用いたアンチデバッグ技法を表 4-2 に示す。それぞれの詳細は参考文献[12]を参照していただきたい。

表 4-2 評価に用いたアンチデバッグ技法(各カテゴリおよび名称は参考文献より引用)

カテゴリ	アンチデバッグ技法の名称
Exploiting memory discrepancies	(1) kernel32!IsDebuggerPresent
	(2) PEB!IsDebugged
	(3) PEB!NtGlobalFlags
	(4) Heap flags
Exploiting system discrepancies	(1) NtQueryInformationProcess
	(2) kernel32!CheckRemoteDebuggerPresent
	(3) UnhandledExceptionFilter
	(5) kernel32!CloseHandle and NtClose
	(10) Ctrl-C
CPU anti-debug	(2) "Ice" Breakpoint
	(3) Interrupt 2Dh
	(4) Timestamp counters
	(5) Popf and the trap flag

評価結果

一連のアンチデバッグ技法に対する解析の結果を表 4-3 に示す。

表 4-3 アンチデバッグ技法による検知テストの結果

アンチデバッグ技法の名称	検知されたか否か
(1) kernel32!IsDebuggerPresent	検知されない
(2) PEB!IsDebugged	検知されない
(3) PEB!NtGlobalFlags	検知されない
(4) Heap flags	検知されない
(1) NtQueryInformationProcess	検知されない
(2) kernel32!CheckRemoteDebuggerPresent	検知されない
(3) UnhandledExceptionFilter	検知されない
(5) kernel32!CloseHandle and NtClose	検知されない
(10) Ctrl-C	検知されない
(2) "Ice" Breakpoint	検知されない
(3) Interrupt 2Dh	検知されない
(4) Timestamp counters	検知された
(5) Popf and the trap flag	検知されない

この結果から、(4) Timestamp counters では本システムが検知されるものの、他の多くのアンチデバッグ技法では本システムを検知できないことがわかった。

5. 考察

本システムの評価結果と他の動的解析システムとの比較を表 5-1 に示す。

表 5-1 本システムの評価結果と他の動的解析システムとの比較

特徴	本システム	従来の動的解析システム		仮想環境を用いた動的解析システム	
	egg	Process Monitor	Poly-Unpack	Ether	Renovo
1 命令単位で解析できる(多くの情報を収集できる)	○	×	○	○	○
パックされたプログラムを解析できる	○	○	○	○	○
プロセスを越えて拡散するマルウェアに対応できる	○	×	×	×	○
カーネルモードコードを解析できる	△	×	×	○	○
アンチデバッグ技法の影響を受けにくい	○	○	×	○	○
仮想環境の検知技法の影響を受けない	○	○	○	×	×
解析時の速度低下が少ない	△	○	○	○	○

本研究の評価結果は、本システムがアンチデバッグ技法に対する不可視性を持ちながらも、命令単位の解析機能やプロセスを越えて拡散するマルウェアを自動解析する機能を有しており、従来の動的解析システムを用いた場合より、マルウェアの手動での解析に要する時間、ひいてはマルウェア解析全体に要する時間を削減できることを示している。以下に、その論拠を述べる。

「マルウェアの解析」の評価結果は、本システムが、マルウェアの内部関数や分岐の実行、API の呼び出しに対して、従来の動的解析システムよりも詳細に解析できることを示した。API の引数情報は、ディスクドライブに直接 PE ファイルを書き込む、といった具体的な動作を読み取ることを可能にし、従来の動的解析システムを用いた場合より、マルウェアの詳

細な挙動の確認を容易にする。また、引数情報をファイルに保存する機能によって、従来は手動での解析で収集・確認していた情報が、自動での動的解析で得られるようになり、結果としてマルウェアの解析に必要な時間を削減する。コールグラフや分岐トレース情報は、利用者がマルウェアの内部構造に関する情報を得ることを可能にし、手動での解析が必要な範囲を絞り込むことができるようになる。

「パックされたプログラムの解析」の評価結果は、本システムがパックされたプログラムに対しても機能することを示した。パックされたプログラムは、手動での解析が困難な場合が多く、本システムが、従来の動的解析ツールと同様にパックされたプログラムに対しても機能することは重要である。

「他のプロセスにコードを注入するサンプルプログラムの解析」および「動的に生成したデバイスドライバーをロードするサンプルプログラムの解析」の評価結果は、マルウェアが他のプロセスにコードを注入したり、動的にファイルを生成し実行したりする挙動に対して、本システムが自動的にそれらを追跡し解析できることを示した。これらの挙動は、マルウェアが頻繁に用いる技法でありながらも、従来の動的解析システムでは追跡が困難である場合が多い。そのため、本システムを用いて解析することで、従来の動的解析システムを利用した場合より、プロセスを越えて拡散するマルウェアの全容を効率よく把握できるようになる。本システムと似た動的解析システムに Cobra[13]があるが、このようなマルウェアの拡散に対する追跡機能は持っていない。

「アンチデバッグ技法に対する不可視性」の評価結果は、本システムが、広く知られたアンチデバッグ技法の多くに対して不可視性を持っていることを示した。これは、本システムがデバッグ API を利用しなかったことによるものである。Xu Chen[5]らは、デバッグ API を利用しマルウェアにデバッガーをアタッチした場合、40%のマルウェアがその挙動を変え

ること報告している。本システムは、これら 40%のマルウェアに対しても、従来の動的解析ツールと同様に本来の挙動を解析できる。

また、本稿ではその評価を省略したが、本システムは仮想環境に依存しないため、仮想環境の検知技法による影響を受けることがない。

以上の点から、従来の動的解析システムを用いた場合と比較し、本システムによってより詳細な解析が可能となり、手動での解析に要する時間、ひいてはマルウェア解析全体に要する時間の削減ができるものと結論した。

今後は、カーネルモードコードに対する解析機能の強化や、本システムの解析速度の改善を図る必要がある。本システムは「カーネルモードに対する汚染追跡の動作制限」で述べたように、カーネルモードコードに対する汚染追跡機能に制限を持っている。また「ループプログラムの解析」の評価結果は、本システムが大きな速度低下を招く可能性があることを示している。速度の低下は、「パックされたプログラムの解析」の評価の際には 1/10 程度であったことから、常に大きな影響を与えるわけではないものの、数値演算などで多数の命令を実行するプログラムにおいては、本システムでの解析に膨大な時間を要する可能性があることを示唆している。今後はこれらの課題を解決し、実用化に向けた改善を進める必要がある。

6. 参考文献

[1] PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing MalwarePaul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee

[2] McAfee Sage 第 1 号 「オープンソースの利点とその代償」 http://www.mcafee.com/japan/security/publication.asp

[3] Malware Formation Stats, Panda Research, 2007 http://research.pandasecurity.com/malwareformation-statistics/

[4] A View on Current Malware Behaviors Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel

[5] Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, Jose Nazario

[6] Ether: Malware Analysis via Hardware Virtualization Extensions Artem Dinaburg, Paul Royal, Monirul Sharif, Wenke Lee

[7] Renovo: A Hidden Code Extractor for Packed Executables Min Gyung Kang, Pongsin Poosankam, and Heng Yin

[8] TEMU: The BitBlaze Dynamic Analysis Component http://bitblaze.cs.berkeley.edu/temu.html

[9] Peter Ferrie

http://pferrie.tripod.com/

[10] Oreans Technologies, Themida

http://www.oreans.com/themida.php

[11] Stealth Breakpoints

Amit Vasudevan and Ramesh Yerraballi

[12] Windows Anti-Debug Reference

http://www.symantec.com/connect/articles/windows-anti-debug-reference

[13] Cobra: Fine-grained Malware Analysis using Stealth Localized-executions

Amit Vasudevan and Ramesh Yerraballi

[14] A Study of the Packer Problem and Its Solutions

Fanglu Guo, Peter Ferrie, Tzi-cker Chiueh