Vol.2020-CSEC-90 No.9
Vol.2020-SPT-38 No.9
2020/7/20

Windowsにおけるハッシュ値の伝播によるスレッドインジェクション機能を持つマルウェアの特定手法

田中　大樹[1,a)]　川古谷　裕平[2]　岩村　誠[2]　鄭　俊俊[1]　毛利　公一[1]

概要：マルウェアによる被害の増加に伴いフォレンジックスの需要が高まっている．しかし，マルウェアには解析を妨害する機能を持つものが存在し，現状では解析が難化，高コスト化している．特に，マルウェアを隠蔽することで解析を妨害するスレッドインジェクション機能は，良性プロセスに悪性挙動を行わせるためマルウェアを特定する大きな障害となる．そこで，本論文ではスレッドインジェクションによる悪性コードの伝播を追跡可能とする仕組みを提案する，実行ファイルのハッシュ値を Windows カーネル内のスレッドごとのデータ構造に格納し伝播させることでスレッドインジェクションを持つマルウェアを特定可能とする．評価では，マルウェアに見立てたスレッドインジェクションを行う実行ファイルを動作させ，その実行ファイルを特定できることを確認した．

キーワード：スレッドインジェクション, マルウェア追跡, オペレーティングシステム

Identification Method of Malware with Thread Injection Function by Propagating Hash Value on Windows

Abstract: The demand for forensics is increasing with the increase in malware. However, some malware has a function that interferes with analysis, and at present, many speculations are included in forensics, making the analysis difficult and expensive. In particular, the thread injection function that obscures the analysis by hiding the malware is a major obstacle for identifying the malware by forensics because the benign process is forced to perform the malicious behavior. In this paper, we present a method for identifying malware with thread injection by storing and propagating the hash value of the executable file in the data structure of each thread in the Windows kernel. In the evaluation, an executable file that performs thread injection and is considered as malware was run, and it was confirmed that the executable file could be specified.

Keywords: Thread Injection, Malware Trace, Operating System

1. はじめに

マルウェアを利用したサイバー攻撃による被害は年々増加しており [1]，中でも特定の組織や企業を標的とし，機密情報の収奪や特定システムの破壊を目的とした標的型攻撃は，会社や組織間さらには国家をまたぐ問題となっている．この問題が発生した場合の原因調査方法としてフォレンジックスがある．フォレンジックスは，ログやメモリ，記憶媒体などから原因を調査する．しかし，フォレンジックスでは分析とその結果からの原因推測に多くの時間やコストが必要となる課題があり，効果的なフォレンジックス手法が求められる．

フォレンジックスに多くの時間を要する要因として，マルウェアの高機能化，複雑化が挙げられる．特にマルウェアが攻撃処理を他のプロセスに紛れ込ませて自己のプロセスを隠蔽するコードインジェクション攻撃は，フォレンジックスを難化させ証拠の発見を遅らせる大きな要因となる．そのため，コードインジェクションが行われても確実かつ迅速にフォレンジックスを行う手法が求められる．コードインジェクションを持つマルウェアを確実にフォレンジックスで特定するためにはコードインジェクションに関係する処理の全てについてログとして残すことが考えられる．しかし，その場合はスレッド生成を全て残す必要が

1　立命館大学
2　日本電信電話株式会社 NTT セキュアプラットフォーム研究所
a)　dtanaka@asl.cs.ritsumei.ac.jp

ⓒ 2020 Information Processing Society of Japan

ある．一般的なコンピュータでのスレッド生成は 300–400 回/分発生するためログの保存期間と保存情報によってはログが膨大となる．さらに，コードインジェクション機能を持つマルウェアは長期間の潜伏が可能であり，ログのローテーションにより情報が失われる可能性がある．そのため，コードインジェクション機能を持つマルウェアの特定に関しては，情報をログとして保存することは不適切である．

本論文では，コードインジェクションの中でも悪性コードをスレッドの形で挿入するスレッドインジェクションに焦点を当て，スレッドインジェクション機能を持つマルウェアを特定する手法について述べる．具体的には，マルウェアに関する情報をスレッドオブジェクトに残し，テイントのように伝播させることでスレッドが挿入されてもマルウェアを特定できるようにする．

以下，本論文では，2 章でスレッドインジェクション攻撃について述べ，3 章で提案手法について述べる．4 章で提案手法の実装について述べ，5 章で評価について述べる．6 章で関連研究について述べ，7 章でまとめる．

2. スレッドインジェクション攻撃

2.1 スレッドインジェクションに使用する API

スレッドインジェクション攻撃は，マルウェアプロセスが攻撃コードを他の良性プロセスに挿入し，スレッドの形で実行する手法である．マルウェアは，良性プロセス内で実行するコードによって自身のプロセスを終了したりファイルを削除したりすることで自己隠蔽することができる．スレッドインジェクションには，CreateRemoteThread と呼ばれる他のプロセスのアドレス空間にスレッドを作成する Windows API が利用される．CreateRemoteThread を利用するためには，実行する前にインジェクションするコードをインジェクション対象プロセスのアドレス空間に書き込む必要がある．これを実現するための API として，ターゲットプロセス内に書き込むデータを格納するためのメモリ領域を確保する VirtualAllocEx と，確保したメモリ領域にデータを書き込む WriteProcessMemory が一般的に利用される．

2.2 マルウェアがとりうるスレッド伝播のパターン

マルウェアが悪性コードの伝播を複雑にすることで解析を妨害するために行うスレッドインジェクションのパターンについて述べる．

2.2.1 複数回インジェクション

マルウェアは，良性プロセスに挿入したスレッドからさらに他の良性プロセスにスレッドを挿入することが可能である．これにより，挿入されたスレッドの作成者情報が得られても伝播元のマルウェアの特定が難しくなる．

2.2.2 挿入先プロセス内でスレッド生成

マルウェアは，挿入先の良性プロセス内にスレッドの作成を行い，作成したスレッドで攻撃処理を行わせることが可能である．これにより，攻撃処理を行ったスレッドの作成者はスレッドを挿入された良性プロセスとなるため，マルウェアの特定ができない．

2.3 既存のマルウェア特定手法

フォレンジックスにおいてスレッドインジェクションを持つマルウェアを特定するための既存手法には，Sysmon[2] が残したイベントログを解析する方法とインシデント発生時に取得されたメモリダンプを解析する方法が存在する．以下，それぞれついて述べる．

2.3.1 Sysmon が残すログを解析する方法

スレッドインジェクション攻撃に一般的に利用される関数である CreateRemoteThread を検知するソフトウェアとして，Microsoft 社が提供する Sysmon が存在する．Sysmon は，様々なシステムアクティビティを記録し，イベントログに出力する．しかし，フォレンジックスは攻撃を受けたコンピュータに対して行われるため，ファイルとして保存されるログファイルは，攻撃者にとって削除や書き換えることが容易なものであるため信頼性が十分であるとは言えない．また，ログファイルの保存期間も問題となる．ログは無制限に保存できるわけではなく，決められた容量を超えると古いログから削除される．レポート [3] によると，サイバー攻撃が発生してから検知されるまでの期間は 2000 日を超える事例がある．この期間にログが失われるケースが考えられる．また，全てのログを残すとログファイルが巨大なサイズとなることに加え，企業においては複数台のログの保管が必要となる．

2.3.2 メモリダンプを解析する方法

主記憶の情報を利用するメモリダンプを解析する方法として，Windows にはプロセスやスレッドなどのオブジェクトの作成者情報を保存するデバッグ機能を用いる手法がある．このデバッグ機能を有効にすると，カーネル内にあるスレッドを管理するためのオブジェクトの Optional Headers 領域に OBJECT_HEADER_CREATOR_INFO 構造体が保存されるようになる．この構造体には，オブジェクトの作成者情報として作成元プロセスの PID が保存される．しかし，PID がわかっても一般的にスレッドインジェクションを行うマルウェアはスレッド挿入後悪性プロセスを終了させるため解析時に取得した PID のプロセスは存在しない．さらに PID は再利用されるものであるため，解析時にマルウェアでない実行ファイルによって作成されたプロセスが解析で取得した PID を利用している可能性があり，マルウェアではないプロセスをマルウェアだと誤認する可能性がある．よって現状のデータ構造ではマルウェアの特定には十分であると言えない．

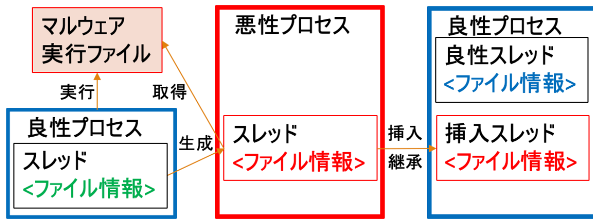

図 1　提案手法

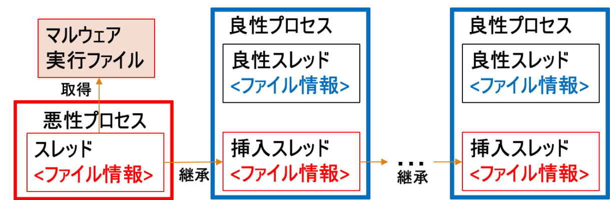

図 2　複数回インジェクションするパターンへの対応

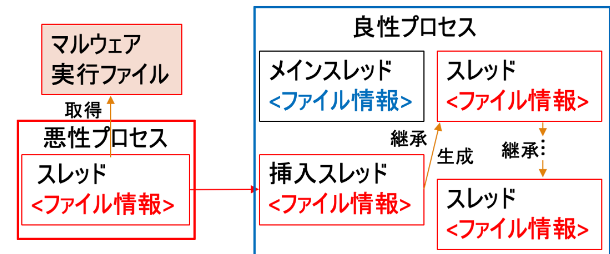

図 3　挿入先プロセス内でスレッド生成するパターンへの対応

3.　提案手法

3.1　概要

前章で述べたようにマルウェアの潜伏期間を考慮するとログにスレッドインジェクション機能を持つマルウェアの情報を残すことは不適切である．そのため，OS のスレッド構造体にフォレンジックスに有効な情報を加える手法を提案する．加えた情報をテイントのように伝播させることで伝播したスレッドが生きている限り情報が喪失することはない．また，本手法において情報は蓄積しないため，マルウェアの潜伏期間や組織内でのコンピュータの台数によるコストの増大は起こらない．よって，本手法を適用することで潜伏期間が長いマルウェアに対してもフォレンジックスに利用可能である．

3.2　情報の伝播アルゴリズム

提案手法が満たすべき要件を以下に示す．

- マルウェアへの改変・削除前の情報が取得できる
- 伝播先の悪性スレッド内に伝播元のマルウェアの情報が格納される

マルウェアは自身のファイルを隠蔽するためにマルウェアの実行ファイルを削除，改変する可能性がある．そのため実行ファイル実行がマルウェアによって改変される前の情報を残す必要がある．よって，情報の取得タイミングは実行ファイルの実行時とする．

スレッドインジェクションでは悪性コードの伝播が，プロセスからプロセスへのスレッド生成によって行われる．また，2 章 2 節 2 項のように挿入されたスレッドがプロセス内でスレッドを生成したり，2 章 2 節 1 項のようにさらに他のプロセスにスレッドを生成したりする可能性がある．よって，スレッド挿入も含めたスレッド生成時にファイル情報を生成元スレッドから生成先スレッドに継承することでファイル情報を伝播させる．以上の提案手法を図 1 に示す．図 1 の良性プロセス内の挿入スレッドには，マルウェアの実行ファイルの情報が格納され，この情報を利用することでマルウェアの特定が可能である．提案手法を適用した場合に 2 章 2 節のマルウェアがとりうるスレッド伝播のパターン全てに対応可能か検討する．

3.2.1　複数回インジェクション

提案手法を適用した場合の複数回インジェクションするパターンにおけるスレッドが持つファイル情報の伝播を図 2 に示す．提案手法では，CreateRemoteThread による他プロセスへのスレッド生成は生成元スレッドの持つファイル情報を生成先のスレッドに継承することとしている．そのため，挿入されたスレッドが持つファイル情報は挿入したスレッドが持つファイル情報であり，スレッドインジェクションが繰り返されたとしてもスレッド情報は変化しない．よって，最終的に侵害を行ったスレッドが持つファイル情報はマルウェアのファイル情報となる．

3.2.2　挿入先プロセス内でスレッド生成

提案手法を適用した場合の挿入先プロセス内でスレッド生成するパターンにおけるスレッドが持つファイル情報の伝播を図 3 に示す．提案手法では，プロセス生成以外のスレッド生成時にファイル情報の継承を行う．そのため，スレッド生成された場合でもマルウェアのファイル情報を取得できる．

3.3　スレッド伝播情報

前節で述べた情報の伝播手法で伝播させる情報について述べる．マルウェアの特定に利用可能な情報はマルウェアを一意に特定できるものであることが望ましい．そこで考えられる情報は以下である．

- ファイルパス
- 物理セクタ番号などのディスク上のアドレス
- ファイル ID
- ハッシュ値

ファイルパスは，比較的容易に取得可能であるがマルウェアがマルウェアによって移動されたりファイル名が変更された場合，パスが示すファイルが存在しない可能性がある．

ディスク上のアドレスは，ファイルが存在する二次記憶装置内のアドレスである．この値は利用するファイルシステムによって異なる．NTFS の場合，ファイルサイズと属

ⓒ 2020 Information Processing Society of Japan

性情報が 1KB を超えると MFT(Master File Table) 外にファイルが保存される．このとき，MFT の$data にファイル本体の先頭クラスタ番号とクラスタ数が格納される．FAT の場合は，ディレクトリエントリ領域のファイル情報内に格納されているファイルデータの先頭クラスタ番号となる．この情報によりディスク内のファイルの場所が分かり，削除されたマルウェアの復元に有用である．しかし，この情報を利用して得られるマルウェアのファイルがマルウェアによって改変されていた場合，改変後の情報となる．さらに，消されたマルウェアが復元できるとは限らない．また，ファイルシステムによって異なるためフォレンジックスには利用がしづらい．

ファイル ID は Windows が管理するファイルに割り当てられる ID であり，ファイル ID を知ることでファイルを一意に特定可能である．しかし，ディスク上のアドレスと同様にファイル ID から得られるマルウェアのファイルは，マルウェアのファイルがマルウェアによって改変されていた場合，改変後となる．

ファイルのハッシュ値は，ファイルを固定長に不可逆変換したものである．ハッシュ計算のタイミングの情報を取得できるメリットがある．また，ハッシュ値はファイルの復元が不可能だった場合でも VirusTotal[4] などでマルウェアの検索が可能であり，既知のマルウェアであった場合は検体を取得することが可能である．一方で未知のマルウェアであった場合は検体の検索が不可能ではあるが，フィルタリング等のブラックリスト制御に利用可能である．さらに，組織内での感染を調査する場合に，他のファイル情報はコンピュータによって異なる情報のため調査に利用するのが難しい．一方でハッシュ値の場合は改変されていないマルウェアであれば値が変化しないため有用である．しかしハッシュ値には計算処理が必要となり，システムの動作に影響を与える可能性がある

以上より，残すべき情報としてファイルパスは不適切であるが，ディスク上のアドレスとファイル ID とハッシュ値はそれぞれ一長一短でありどの情報を利用するかはユースケースによる．本論文では，ハッシュ値を伝播させた場合について述べ，オーバヘッドが問題となるか調査する．

3.4 利用するハッシュアルゴリズム

前節で述べたようにマルウェアが復元できない場合 VirusTotal を利用してマルウェアを取得する．そのため，利用するハッシュアルゴリズムは VirusTotal でマルウェアを検索可能なものである必要がある．VirusTotal では，MD5，SHA-1，SHA-256 が検索可能である．これらのハッシュアルゴリズムの中で最もハッシュ値のサイズが小さく，計算オーバヘッドが小さいものを利用する．ハッシュ計算のオーバヘッドを調査するため，Windows の標準機能である certutil コマンドを利用して 100MB の実行ファイルのハッシュ値を 100 回計算する．そして平均値をオーバヘッドとして比較する．ハッシュアルゴリズムごとのサイズとオーバヘッド調査した結果を表 1 に示す．表 1 より，ハッシュ値のサイズが最小であるのは MD5，ハッシュ計算オーバヘッドが最小であるのは SHA-1 であることが明らかとなった．しかし，MD5 と SHA-1 の計算オーバヘッドに大きな差はないため，よりハッシュ値のサイズが小さい MD5 を今回は採用する．

表 1 ハッシュアルゴリズムごとのサイズとオーバヘッド

ハッシュアルゴリズム	サイズ (bit)	計算オーバヘッド (ms)
MD5	128	231.08
SHA-1	160	214.72
SHA-256	256	446.65

3.5 格納場所

残す情報を格納する場所について述べる．提案手法では，ファイル情報をスレッドに持たせることとしている．Windows カーネル内でのスレッドオブジェクトには，Pool Header と Optional Headers，Object Header そして ETHREAD 構造体が存在する．Optional Headers は Windows のデバッグ機能を有効化しなければ存在しないため適切ではない．また，他の領域は Windows の動作に関わるため既存のメンバを書き換えることは OS の動作を阻害する原因となり得るため不適切である．よって残すファイル情報は，Optional Headers 以外の新しいメンバとして追加することが望ましい．

4. 実装

4 章で述べた提案手法の実装について述べる．本手法では，カーネル空間へのアクセスが必要であるため，カーネルモードドライバによって実現する．ファイル情報は 4 章 4 節で述べたようにファイル情報を Optional Headers 以外の新しいメンバとして追加するとした．しかし WindowsOS は公開されておらずドライバでは新しいメンバを追加することは難しい．そのため，新しいメンバではなく既存のメンバを書き換えることで実現した．既存メンバの書き換えは，OS の動作を阻害する可能性があるため，OS が利用しない Optional Headers 内の OBJECT_HEADER_CREATOR_INFO 構造体を書き換えることとする．本手法を実現するために必要な処理フローを図 4 に示す．

4.1 スレッド生成フック機能

スレッド生成をフックするために，PsSetCreateThreadNotifyRoutine 関数を利用する．この関数を利用すると，スレッドが生成処理が終了した後フックされ，実行される関数を登録することができる．この関数をドライバ組込時

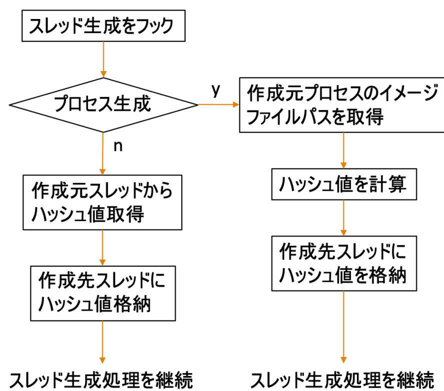

図 4　提案手法の処理フロー

に実行し，図 4 のフック後の処理を行う関数を登録する．

4.2 プロセス生成判定条件

本手法では，プロセス生成時にハッシュ計算，CreateRemoteThread を含むスレッド生成時にファイル情報を継承する．スレッド生成はプロセス生成時にも行われるためスレッド生成全てをフックしプロセス生成によるものかを判定し，処理を変える必要がある．プロセス生成によるスレッド生成は，通常のスレッド生成とはスレッド生成先プロセスとスレッド生成元プロセスとの間に親子関係が存在する点が異なる．生成先プロセスの親プロセスは，作成先プロセスの EPROCESS 構造体の InheritedFromUniqueProcessId メンバに親プロセスの PID が格納される．この値と作成元プロセスの ID が同じ場合プロセス生成と判定する．しかしこの条件だけでは，マルウェアがスレッド挿入対象良性プロセスを生成し，そのプロセスに対してスレッドインジェクションを行う場合に，スレッドインジェクションによるスレッド生成がプロセス生成によるものだと誤認する．プロセス生成と通常のスレッド生成との違いは，生成先プロセスに属するスレッドの数にも現れる．プロセス生成の場合は，生成されたスレッド一つがプロセス内のスレッドのみであり，スレッド生成の場合は，生成されたスレッドに加え最低限一つはスレッドが存在する．つまり，スレッド生成やスレッド挿入の場合は，生成先プロセス内のスレッド数は 2 つ以上，プロセス生成時は一つであることが条件に利用可能である．以上のことを実現するため，プロセス内のスレッドの数を格納している EPROCESS 構造体の ActiveThreads の値を条件に追加した．

4.3 プロセスのイメージファイルパスの取得

プロセスの実行ファイルのハッシュ値を計算するため，プロセスのイメージファイルパスを取得して，open/read をする必要がある．プロセスのイメージファイル名を取得するためには，PsGetProcessImageFileName 関数を利用する方法がある．しかし，この方法はファイルパスではなくファイル名のみしか取得できず，さらに取得できる情報は 14 文字分のみであり，ハッシュ値の計算には利用できない．そのため，API によってファイルパスを取得するのではなく EPROCESS 構造体からファイルパスを取得する．ファイルパスが格納される場所を図 5 に示す．

4.4 カーネルドライバでのハッシュ計算方法

カーネルドライバで MD5 のハッシュ値を計算するため，ユーザ空間のハッシュ値を計算するアプリケーションをカーネルドライバ用に移植した．この時，ユーザ空間の関数をカーネル関数に置き換える必要がある．具体的には fopen を ZwOpenFile，fread を ZwReadFile，fclose を ZwClose に置き換えた．しかし，ZwReadFile で読み込んだ文字数を取得することができない問題が発生した．これは，ユーザ関数では返り値が読み込んだ文字数を示すが，ZwReadFile では返り値は NTSTATUS であることが原因である．そのため，QueryInformationFile 関数によりファイルサイズを取得し，ZwReadFile 関数が呼ばれるたびにファイルサイズから 1024 を引き 1024 を読み取った文字数とし，最後は残りのサイズを読み取った長さとして扱うことで解決した．

5. 評価

評価のため，マルウェアを模したスレッドインジェクションを行う実行ファイルを作成し，本手法を適用した環境で実行，メモリダンプの解析を行い，マルウェアの特定が可能か機能評価した結果について述べる．また，本手法でファイル情報として利用しているハッシュ値は計算に大きなオーバヘッドがかかることが予想できるため，オーバヘッド測定による性能評価についても述べる．

5.1 機能評価

本評価では，2 章 2 節で述べたスレッドインジェクションのパターンを行うマルウェアを作成し本手法適用環境で動作，メモリダンプの解析を行い，マルウェアの特定が可能か調査した．2 章 1 節のうちスレッドインジェクションを繰り返すパターンについて述べる．

5.1.1 評価方法

今回評価に利用するマルウェアに見立てた実行ファイルは，notepad.exe(pid:8180) にスレッドを挿入する．挿入されたスレッドは notepad.exe(pid:3980) に対してさらにスレッドを挿入する．踏み台として挿入したスレッドとマルウェアに見立てたプロセスは挿入処理実行後終了する．その後メモリダンプをとり解析する．メモリダンプの解析では，取得したメモリダンプから挿入されたスレッドの OBJECT_HEADER_CREATOR_INFO 構造体の中身を確認してマルウェアに見立てた実行ファイルのハッシュ値と同一か確認する．

ⓒ 2020 Information Processing Society of Japan

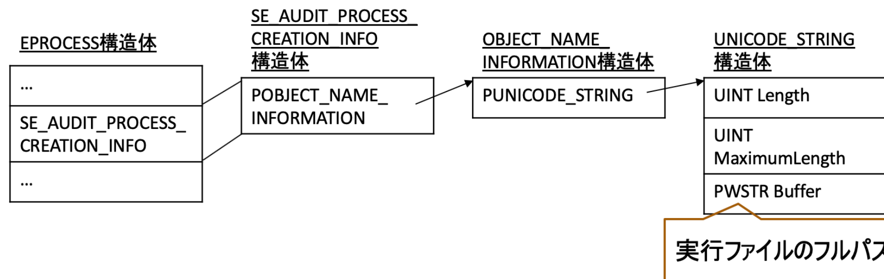

図 5　イメージファイルパスの格納場所

表 2　機能評価結果

スレッドインジェクションのパターン	評価結果
複数回インジェクション	○
挿入先プロセス内でスレッド生成	○

5.1.2　評価結果

今回作成したマルウェアに見立てた実行ファイルのハッシュ値は，「3f2badd55b6b637abbc414ff7f91088」である．取得したメモリダンプを解析した結果を図 6 に示す．図 6 の赤枠内の値が挿入されたスレッドの OBJECT_HEADER_CREATOR_INFO 構造体の中身である．その結果，今回作成したマルウェアに見立てた実行ファイルのハッシュ値と同一であるため，本手法はスレッドインジェクションを繰り返すマルウェアの特定に有効である．

上記と同様に 2 章 2 節で述べたスレッドインジェクションの全パターンに対して機能評価した結果を表 2 に示す．表 2 に示すようにスレッドインジェクションのパターン全てでマルウェアに見立てた実行ファイルの特定が可能であった．よって，本手法はスレッドインジェクション機能を持つマルウェアの特定に有効である．

5.2　性能評価

性能評価として本手法のオーバヘッドの測定を行った．本手法は，プロセス生成時とスレッド生成時の 2 パターンの処理に分かれており，2 パターンそれぞれでオーバヘッド計測を行った．また，実機において提案手法を適用させた場合におけるプロセス生成時のオーバヘッド計測を行った結果についても述べる．

5.2.1　評価環境

プロセス生成時，スレッド生成時それぞれのオーバヘッド計測を行った環境を以下に示す．

- VMWare Workstation 15 Player
- CPU:Intel Core-i7 7700 2 コア
- メモリ:8GB
- OS:Windows10 64bit personal
- バージョン:17763.1098

実機でのオーバヘッド計測で利用する環境を以下に示す．

- CPU:Intel Core-i7 7700
- メモリ:16GB
- OS:Windows10 64bit personal
- バージョン:18363.720

5.2.2　評価方法

プロセス生成時のオーバヘッドは，実行してハッシュ値を求める対象の実行ファイルによって大きく異なる．そのため，42KB，100KB，1MB，10MB，100MB の実行ファイルを作成し，これらを 100 回ずつ実行して平均オーバヘッドを求める．

一方で本手法ではスレッド生成時にファイル情報の継承のみが行われる．そのためファイルサイズによるオーバヘッドへの影響はない．そのため，オーバヘッド計測では CreateThread を 100 回繰り返し，その平均オーバヘッドを算出した．オーバヘッド計測には Visual Studio[5] の Performance Profiler 機能を利用し，プロセスとスレッドを生成する CreateProcess，CreateThread 関数の実行時間を計測する．

上記の評価では，決められた実行ファイルを短時間で実行している．そのため，通常の動作とは異なる可能性があり，通常の環境ではオーバヘッドに様々な影響を及ぼす可能性がある．実機でのオーバヘッド計測は，5 章で述べた実装にファイルサイズとオーバヘッドをログとしてファイルに出力する機能を追加した．本手法のオーバヘッドは実行ファイルのサイズが小さい場合μｓオーダーとなる可能性がある．しかし，Windows のシステム時間の分解能は 1ms であるため高精度の時間測定が必要である．そのため時間計測には，KeQueryPerformanceCounter 関数を利用する．KeQueryPerformanceCounter 関数は，High Precision Event Timer(HPET) と呼ばれるクロック周波数が一定であるハードウェアタイマを利用する．この関数で得られる値をクロック周波数で割ることでシステム起動からのより詳細な時間取得が可能であり，正確なオーバヘッドの取得が可能である．

5.2.3　評価結果

プロセス生成時のオーバヘッド計測した結果を表 3 に示す．また，スレッド生成時のオーバヘッド計測した結果を表 4 に示す．さらに実機でのオーバヘッド計測をした結果を表 5 に示す．

ⓒ 2020 Information Processing Society of Japan

```
2: kd> dt nt!_OBJECT_HEADER_CREATOR_INFO ffff808b667e9030
   +0x000 TypeList              :  _LIST_ENTRY [ 0xffff808b`667e9030 - 0xffff808b`667e9030 ]
   +0x010 CreatorUniqueProcess : 0x7a636b5b`d5ad2b3f Void
   +0x018 CreatorBackTraceIndex : 0xcbb
   +0x01a Reserved1             : 0x4f41
   +0x01c Reserved2             : 0x8810f9f7
```

図 6　挿入されたスレッドオブジェクト内の OBJECT_HEADER_CREATOR_INFO 構造体

表 3　プロセス生成のオーバヘッド

実行ファイルサイズ	提案手法なし (ms)	提案手法あり (ms)	提案手法のオーバヘッド (ms)
42KB	5.96	7.52	1.56
100KB	4.63	6.53	1.90
1MB	5.15	23.77	18.62
10MB	5.26	139.85	134.59
100MB	5.26	1431.35	1426.09

5.2.4　考察

スレッド生成のオーバヘッドは，表 4 に示すように 0.01ms であるため問題とならない．一方でプロセス生成のオーバヘッドは表 3 に示すように実行するファイルの大きさで大きく異なり，オーバヘッドも 100MB の実行ファイルの場合 1429ms と大きい．そのため，このオーバヘッドが無視できるのか，それとも小さくする必要があるのか考察する．

このオーバヘッドは実行ファイルのサイズによって大きく異なることからハッシュ計算によるものであると推測できる．ハッシュ計算は，マルウェアが実行された場合には必ず行わなければいけないが，クリーンウェアの場合は行わなければいけないわけではない．そのため，マルウェアのサイズがオーバヘッドの小さいファイルサイズに収まればハッシュ計算を行う実行ファイルのサイズに上限をかけることが可能である．そのためマルウェアとクリーンウェアのファイルサイズについて FFRI データセット 2019[6] で調査し，サイズの分布を調べる．FFRI データセットには，クリーンウェアとマルウェアそれぞれ 25 万検体分の表層解析のデータが含まれる．マルウェアについて調査した結果を表 6 に，クリーンウェアについて調査した結果を表 7 に示す．表 6 に示すように，50MB を超えるマルウェアは存在しない．50MB の実行ファイルによるプロセス生成のオーバヘッドは約 600ms であり，この程度であれば問題とならない．また表 5 で示すように，一般的な動作で実行されるファイルは 20MB 未満であり，オーバヘッドも 124ms と問題とならない程度ある．一方で表 7 に示すようにクリーンウェアの分布は，0 検体となるのは 1GB を超えるものであった．そのため，クリーンウェアのハッシュ値計算をする場合最大で 1000 ms 前後かかる可能性がある．これは実行ファイルの実行において過大なオーバヘッドである．以上のことから，ハッシュ計算対象の実行ファイルのサイズに上限を設けることが必要である．そのためには，上限をどのサイズにするかを決める必要がある．しかし上限を定めると，その上限値を超えるサイズのマルウェアとすることで本手法を回避可能である．そのため，上限値はユーザの環境によって設定の有無や上限の値を定めることが有用であると考える．例えば，サーバ用コンピュータなど実行するファイルが固定されており，その実行ファイルのサイズも大きくない場合，上限値を定めることなく運用することがセキュリティ面でも最適であると言える．一方で，個人のコンピュータなど実行されるファイルのサイズが大きくなる可能性がある場合，可用性を優先し，上限を定めることが最適である．よって，上限値の有無や値はユーザに委ねるのが良いと考える．

6.　関連研究

コードインジェクション機能を持つマルウェアへの対応として山本らの研究 [7], [8] では，侵害されていない環境を用意し，その環境のプロセスのメモリイメージと検査対象環境でのプロセスのメモリイメージを比較し，注入されたコードを特定する手法が提案されている．しかし，IDS による通信のアラートが発生しないと検証が行われない．また，マルウェアの抽出が行われるのはマルウェアが通信を行ったタイミングであり，通信が行われる前にマルウェアが自身を削除したり改変した場合に対応できない．

大月らの研究 [9] では，システムコールトレーサによるマルウェアの動的解析システムにおいて，スレッド生成全てを記録することでスレッドインジェクションを追跡する機能を実現した．この研究ではマルウェア情報の取得は仮想計算機モニタで行っている．そのためマルウェアによるログの改ざんは発生しない．しかしこのシステムはマルウェアの解析のためのシステムであり，改ざんされていないマルウェアを別途取得する必要がある．

7.　おわりに

本論文では，スレッドインジェクション機能を持つマルウェアを特定する手法として実行ファイル情報をテイントのように伝播させる手法について述べた．本手法を実現するためにスレッド生成を全てフックし，プロセス生成時に実行ファイルのファイル情報を取得し，スレッド生成時に

表 4 スレッド生成のオーバヘッド

提案手法なし (ms)	提案手法あり (ms)	提案手法のオーバヘッド (ms)
0.20	0.21	0.01

表 5 実機でのオーバヘッド

ファイルサイズ	平均オーバヘッド (μ s)	回数
0-100KB	1959	423
100KB-500KB	6584	1810
500KB-1MB	19009	69
1MB-20MB	124754	32
20MB-100MB		0

表 6 FFRI データセットのマルウェアサイズ分布

実行ファイルサイズ	検体数
50MB〜	0
30MB〜50MB	5
〜30MB	249995

表 7 FFRI データセットのクリーンウェアサイズ分布

実行ファイルサイズ	検体数
1GB〜	0
100MB〜1GB	37
50MB〜100MB	46
30MB〜50MB	81
〜30MB	249836

継承する機能を実装し，実際にマルウェアの特定が可能か機能評価を行った．その結果，スレッドインジェクション全てのパターンでスレッドが持つファイル情報からマルウェアの特定が可能であることが明らかとなった．また本論文では，残すファイル情報の中で最も利用価値が高いがオーバヘッドが大きくなると推測されるハッシュ値について計算機能を実装し，実際に動作させてオーバヘッドを計測をすることで性能評価を行った．その結果，コンピュータをユーザが一般的に利用する場合は問題となるオーバヘッドではないが，100MB を超えるような実行ファイルが実行される環境の場合は問題となることが明らかとなった．そのため，本手法を利用する場合は環境によってはユースケースでハッシュ計算を行う上限を設定する必要がある．また，本手法が利用可能なのはスレッドインジェクションを行うマルウェアの中でも exe 形式の実行ファイルのみである．PowerShell スクリプトや DLL などのマルウェアの場合，スクリプトエンジンなどマルウェアではない実行ファイルのファイル情報が伝播し，マルウェアの特定ができない．今後はまず，スレッドインジェクション機能を持つ DLL マルウェアへの対応を行う．

参考文献

[1] キヤノンマーケティングジャパン株式会社：2019 年　年間マルウェアレポート，, 入手先 ⟨https://eset-info.canon-its.jp/files/user/malware_info/images/ranking/pdf/MalwareReport_2019.pdf⟩（参照 2020-06-16）.

[2] Russinovich, M. and Garnier, T.: Sysmon v10.42, , available from ⟨https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon⟩ (accessed 2020-06-16).

[3] ファイア・アイ株式会社：M-Trends 2020 レポート．

[4] Google Inc.: Virus Total, , available from ⟨https://www.virustotal.com/⟩ (accessed 2020-06-16).

[5] Microsoft Corporation: Visual Studio, , available from ⟨https://visualstudio.microsoft.com/⟩ (accessed 2020-06-16).

[6] 押場 博光 千葉 大紀 畑田 充弘寺田 真敏荒木 粧子：マルウェア対策のための研究用データセット〜MWS Datasets 2019〜，情報処理学会， Vol. 2019-CSEC-86, No. 8, pp. 1–8 (2019).

[7] 桜井 鐘治山本 匠：不審プロセス特定手法の提案, コンピュータセキュリティシンポジウム 2013 論文集, Vol. 2013, No. 4, pp. 634 – 641 (2013).

[8] 桜井 鐘治山本 匠：不審プロセス特定手法の実装及び評価，研究報告マルチメディア通信と分散処理（DPS），Vol. 2014-DPS-158, No. 33, pp. 1 – 8 (2014).

[9] Yuto Otsuki, Eiji Takimoto, Takehiro Kashiyama, Shoichi Saito, Eric W. Cooper, and Koichi Mouri: Alkanet: A Dynamic Malware Analyzer based on Virtual Machine Monitor, *World Congress on Engineering and Computer Science 2012 (WCECS 2012)*, Vol. 1, pp. 36–44 (2012).