Computer Security Symposium 2017
23 - 25 October 2017

テイントフォレンジックスによるIAT再構築

川古谷 裕平[1]　岩村 誠[1]　三好 潤[1]

概要：マルウェアの静的解析において，Windows Application Programming Interface（API）はマルウェアの機能を効率的に読み解くための重要な情報源である．しかし，その有用性のため，マルウェア作者はマルウェアに難読化を施し，インポートしている API を隠蔽する傾向がある．本論文では，マルウェアが利用する API 難読化手法とそれらを解析する既存手法を整理し，既存手法が API や Dynamic Link Library（DLL）のメモリ上の配置位置を難読化する解析妨害に脆弱であることを示す．次に，この問題を解決するため，テイント解析により得た情報に基づきメモリダンプ内の Import Address Table（IAT）の API 名前解決を行う手法を提案する．本提案手法を用いることで，マルウェアが配置位置難読化を施した場合でも，インポートしている API を正確に特定できることを実験にて示す．

キーワード：フォレンジックス，テイント解析，マルウェア，IAT 再構築，Windows API

Taint-assisted Forensics for IAT Reconstruction

Yuhei Kawakoya[1]　Makoto Iwamura[1]　Jun Miyoshi[1]

Abstract: Windows API is an important data source for analysts to effectively understand the functions of malware in static analysis. However, malware authors are likely to hide the imported APIs in their malware by taking advantage of various obfuscation techniques. In this paper, we first point out that existing IAT reconstruction techniques are vulnerable to position obfuscations, which are an anti-analysis technique obfuscating the positions of loaded APIs or DLLs. Next, we propose API name resolution on the basis of taint analysis for IAT reconstruction to defeat position obfuscations. Last, we experimentally demonstrate that the system in which our proposed API name resolution has been implemented enables us to correctly identify the imported APIs even though malware authors apply various position obfuscations for their malware.

Keywords: Forensics, Taint Analysis, Malware, IAT Reconstruction, Windows API

1. はじめに

背景 マルウェアの詳細な動作を知るため，解析者は手動で静的解析を行うことがある．一般的に，プログラムは膨大な数の機械語命令から構成されるため，それらを 1 つ 1 つ読み解く静的解析は時間的なコストがかかる．そのような状況の中，Windows API はこれら機械語命令よりも，多くのコンテキスト情報を保持しているため，静的解析を効率的に進める上で重要な情報源となる．

攻撃者もこの状況を十分把握しているため，彼らのマルウェアの中で利用している API を隠そうと難読化が行われる（以下，API 難読化）．具体的には，Import Name Table（INT）の削除や，INT と IAT を Portable Executable（PE）ヘッダから非参照とする方法がある．これらが施されると，PE ヘッダからインポートされている API の一覧を取得できない．また API を呼び出してるコール命令を見ても，何の API を呼び出しているかわからなくなる．一方で，API 難読化が施されたマルウェアは難読化対象となったコード（以下，オリジナルコード）がインポートしている API を独自に管理し，実行時に復元する．そのため，オリジナルコードからの API 呼び出しは問題なく行われる．

解析者はこのような API 難読化が施されたマルウェア

[1] NTT セキュアプラットフォーム研究所
NTT Secure Platform Laboratories

を解析する際，一般的には以下のステップで解析を進め，難読化されたインポート API の情報を復元する．この一連の処理を IAT 再構築と呼ぶ．

(1) **メモリダンプの取得** マルウェアを実行し，オリジナルコードの IAT に API のアドレスが埋められた時点でメモリダンプを取得する．
(2) **IAT の位置特定** メモリダンプを解析し，IAT の位置を特定する．
(3) **API 名前解決** IAT に格納されているアドレスから API 名前解決を行い，IAT の各エントリに対応する API 名を特定する．
(4) **PE ヘッダの復元** 必要に応じて，削除された INT を復元し，PE ヘッダから INT と IAT を参照させる．

この IAT 再構築を行うことで，静的解析の際，オリジナルコードのコール命令が，どの API を呼び出しているかわかるようになる．

既存手法とその問題点 しかし，マルウェアが配置位置難読化の解析妨害を行うと API 名前解決，上記の（3），がうまくいかず IAT 再構築に失敗する．

通常，API 名前解決は IAT のエントリと API が配置されたメモリアドレスの一覧を比較し，値が一致した場合そのエントリと API 名を関連づける．API が配置されたメモリアドレスは，DLL のロードされた位置とその DLL がエクスポートしている API の Relative Virtual Address（RVA）から決定する．一般的に，DLL のロードされた位置は Process Environment Block（PEB）や Virtual Address Descriptor（VAD）等から取得し，API の RVA は DLL の PE ヘッダから参照されているエクスポートテーブルから取得する．

一方，配置位置難読化は，API のメモリ上の位置を難読化する API 配置位置難読化と，API をエクスポートしている DLL の配置位置を難読化する DLL 配置位置難読化がある．API 配置位置難読化は，API を構成する一部，またはすべての命令列をローダによりロードされた場所とは別の場所にコピーして実行する Stolen Code[14] や Copied API Obfuscation[13] がある．また，DLL 配置位置難読化は，ロードした DLL の場所を管理しているデータ構造体の情報を改ざんする DLL Unlinking[8] や，元々それらに記録されない方法で DLL をロードする Stealth Loader[7] がある．

既存の IAT 再構築の手法が配置位置難読化に対して脆弱な理由として，API 名前解決を行う際，以下の二つの前提に暗黙のうちに依存しているためだと考える．

- API は DLL がロードされた領域内に存在している．
- ロードされた DLL は OS が管理しており，OS の特定のデータ構造体にその情報が保存されている．

上述の配置位置難読化の解析妨害はこれらの前提を攻撃し，API 名前解決を失敗させることで，IAT 再構築を妨害している．

提案手法とシステム 本論文では，上記の 2 つの前提に依存しない API 名前解決手法として，テイントタグに基づく API 名前解決を提案する．本提案手法は，動的解析とダンプ解析から構成される．

動的解析では，マルウェアを動作させる前に，各 API の命令列に対してそれぞれ識別可能なテイントタグを設定する．その後，マルウェアを動的解析環境上で実行する．この際，API を含んだ DLL や API のコードがコピー操作等を通して移動すると，テイントタグを伝搬させて，その移動を追跡する．一定期間実行させた後，メモリダンプに加え，テイントタグの情報が格納されたシャドウメモリのダンプも取得する．また，ページインしていないページを解析するため，ディスクイメージとそれに関するテイントタグを格納したシャドウディスクのダンプも併せて取得する．

ダンプ解析では，解析対象のコード領域から間接コール命令，例えば `call [0x1001000]`，を探し出し，当該命令が参照するメモリ領域，上の例では 0x1001000，を IAT エントリ候補とする．この IAT エントリ候補を集め，それらが集中している箇所を IAT とする．IAT の特定後，IAT の各エントリのアドレスを API の名前と結びつけるため，API 名前解決を行う．この際，提案手法ではシャドウメモリを参照し，対応するテイントタグから API 名を取得する．また，メモリ上にないページの場合は，目的とするデータが存在するディスク上の場所を計算し，その場所に対応するシャドウディスクからテイントタグを取得する．

本提案手法をシステムとして実装した（以下，提案システム）．動的解析には，API Chaser[6] を利用し，ダンプ解析には，The Volatility Framework[8]（以下，Volatility）を独自に拡張した *TaintVolatility* を利用した．TaintVolatility はメモリダンプ解析のほか，上記のシャドウメモリダンプ，ディスクイメージ，シャドウディスクダンプを読み込み，メモリダンプと一緒に解析する機能を持つ．

実験と結果 提案システムの有効性を示すため，上述の 4 種類の配置位置難読化（Stolen Code, Copied API Obfuscation, DLL Unlinking, Stealth Loader）を利用し，Windows の標準的な 4 つの実行ファイルに API 難読化を施し，これらをデータセットとして実験を行った．比較対象の既存ツールとして，Volatility のプラグインであり，PEB から DLL の情報を取得する impscan，VAD から DLL の情報を取得する impscan++，IAT 再構築ツールの Scylla[9] を用意した．提案システムと比較対象の既存ツールでデータセットを解析し，API 難読化を施す前に調べておいた各実行ファイルがインポートしている API をすべて取得できるかを実験した．

この実験の結果，すべての既存ツールが配置位置難読化に影響を受けるなか，提案システムだけが配置位置難読化が施されたすべての Windows 実行ファイルを正確に解析

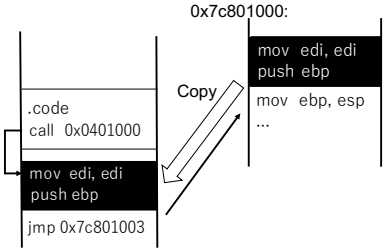

図 1 API 配置位置難読化 (Stolen Code)
Fig. 1 API Position Obfuscation (Stolen Code)

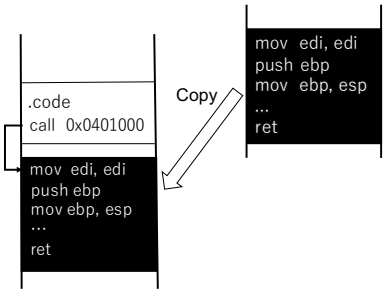

図 2 API 配置位置難読化 (Copied API Obfuscation)
Fig. 2 API Position Obfuscation (Copied API Obfuscation)

することができた．

貢献 本論文の貢献は以下の 4 点である．

- 静的解析における API 難読化手法を整理し，既存の API 名前解決手法が配置位置難読化に脆弱であることを示した．
- 既存の API 名前解決手法が配置位置難読化に脆弱な理由として，暗黙的に依存している二つの前提を明らかにした．
- 配置位置難読化に対して，テイント解析を利用した API 名前解決手法を提案し，IAT 再構築のためのシステムとして実装した．
- 提案システムが配置位置難読化の問題を解決可能なことを実験で示した．

2. 配置位置難読化

ここでは，既存の IAT 再構築手法では解析困難な配置位置難読化の解析妨害について述べる．配置位置難読化は，API 配置位置と DLL 配置位置の難読化に分けられる．また既存の IAT 再構築手法が配置位置難読化に対して脆弱な原因を分析する．

2.1 API 配置位置の難読化

API 配置位置難読化は，ローダによりメモリ上に配置された API の機械語命令列を，異なる場所にコピーしてから実行する手法である．具体的には，Stolen Code（**図 1**）と Copied API Obfuscation（**図 2**）がある．

Stolen Code は API のコードの一部（主に先頭から数命令）を確保したバッファにコピーし，そのコピーした命令を経由して API を呼び出す方法である．図 1 の例では，`mov edi, edi; push ebp;` の 2 命令をコピーし，そのコピーした命令の後に `jmp` 命令を置き，コピーした命令の後の命令（`mov ebp, esp`）にジャンプする．

Copied API Obfuscation は API のコードのすべての命令を確保したバッファにコピーし，そのコピーした命令列を API と見立てて実行する方法である．図 2 の例では，`mov edi, edi;` から `ret;` までのすべての命令をコピーし，実行している．元の API の命令は一切実行しない．

API 配置位置の難読化が行われると，API のコードまたはその先頭の数命令は，その API をエクスポートしている DLL がロードされたメモリ領域外に存在し，IAT の関数ポインタはその DLL とは無関係のメモリ領域，つまり確保したバッファ，を指すことなる．そのため，IAT に格納されたアドレスと PEB 等のデータ構造体から計算した各 DLL がエクスポートしている API のアドレス一覧とが一致せず，API の名前解決が失敗する．

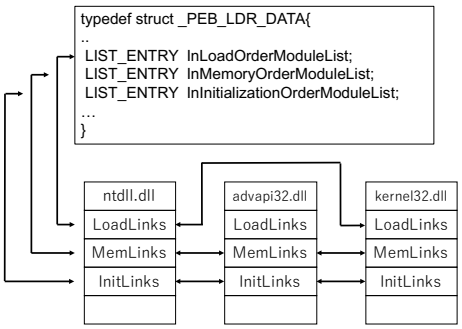

図 3 DLL 配置位置難読化（DLL Unlinking）
Fig. 3 DLL Position Obfuscation (DLL Unlinking)

2.2 DLL 配置位置の難読化

DLL 配置位置難読化は，DLL がロードされた位置を難読化する手法である．具体的には，DLL Unlinking（**図 3**）と Stealth Loader（**図 4**）がある．

DLL Unlinking は PEB に存在するロードされた DLL を管理するリンクリストのデータ構造体から，対象とする DLL のエントリを外すことで，ロード済み DLL の存在を隠す方法である．図 3 では，advapi32.dll に該当するエントリをリンクリストから外している．これにより，例え解析ツールが InLoadOrderModuleList を走査しても，advapi32.dll のエントリは見つからず，advapi32.dll はロード済み DLL とは認識されない．

Stealth Loader は実行ファイルに埋め込まれた独自プログラムローダ（図 4-(a)）が，Windows 標準のプログラムローダとは独立に DLL をロードし実行可能にする手法である．Stealth Loader は Windows のロードした DLL を管理するデータ構造体に痕跡を残さないよう，注意深く設計されている．具体的には，ロードした DLL の情報を

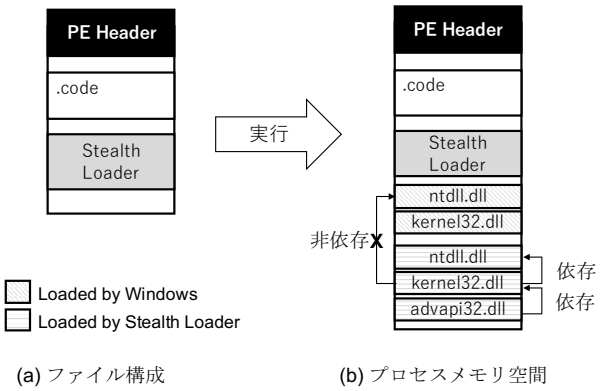

図 4 DLL 配置位置難読化（Stealth Loader）
Fig. 4 DLL Position Obfuscation (Stealth Loader)

PEB に記録しない．VAD による検知を避けるため，マップ関数を利用せずにコードをメモリに展開する．Windows によってロードされている DLL には依存関係をつくらない（図 4-(b)），といった方針で設計されている．Stealth Loader を利用すれば，Windows のデータ構造体には DLL をロードした痕跡が残らないので，ロード済み DLL とは認識されない．

DLL 配置位置が難読化された場合，対象とする DLL がロード済みとは認識されない．そのため，隠蔽された DLL がエクスポートしている API は，API 名前解決の際，候補 API として一覧に名前が上がってこない．これにより，API 名前解決が失敗する．

2.3 問題分析

前述のように，API 配置位置や DLL 配置位置の難読化が行われると，API 名前解決に失敗し，結果として IAT 再構築に失敗する．この原因は，既存ツールは以下の 2 つの前提に暗黙のうちに依存しており，この前提を元に手法，ツールが設計されている点にある．

- **前提 1** API は DLL がロードされたメモリ領域内に存在している．
- **前提 2** ロード済み DLL の位置情報は OS が適切に管理している．

Stolen Code と Copied API Obfuscation は前提 1 に対する攻撃である．どちらも API を構成する命令列をローダが配置したメモリ位置からコピーし，そこで実行する．これにより，OS や解析環境が把握している DLL が配置されたメモリ領域の外側で API のコードが実行される．

DLL Unlinking と Stealth Loader は前提 2 に対する攻撃である．OS の管理情報を改ざん，またはそもそも DLL の配置情報が OS の管理構造体に残らない方法で DLL を実行可能にすることで，OS や解析環境からロードされた DLL の存在を隠蔽している．

3. 提案

ここでは，前章で述べた 2 つの前提に依存せずに API 名前解決を行う，テイント解析に基づく API の名前解決手法を提案する．まず，本提案手法の対象範囲について述べ，次に提案手法の概要を述べる．そして，システム構成と動作の流れを説明し，その後，実装の詳細について述べる．

3.1 対象範囲

本論文の提案が対象とする解析妨害は 2 章で述べた配置位置難読化である．つまり，IAT 再構築の API 名前解決に関する問題を解くことが目的である．IAT 再構築を妨害するには，API 名前解決を攻撃する以外にも方法は考えられる．例えば，メモリダンプの取得を妨害する方法 [14] や IAT を経由させずに API を呼び出す方法 [13] などがあるが，これらは本論文の対象外とする．

3.2 基本アイデアと効果

本提案の基本的なアイデアは以下の 2 点である

(1) 各 API にテイントタグを設定し，マルウェアを動作中にテイントタグを設定した API のコードが操作された場合，テイントタグをあらかじめ定めたルールに基づき伝搬させる．
(2) メモリダンプ取得時にテイントタグ情報を格納したシャドウメモリのダンプも一緒に取得し，シャドウメモリ内のテイントタグに基づき API のメモリ上の位置を特定し，API 名前解決を行う．

このアイデアの効果は，API 毎に異なる種類のテイントタグを設定する事で API 単位で位置を管理できる．API 内の一部の命令列が別の場所にコピーされても，テイントタグを伝搬させて追跡できる．つまり，API のコードが DLL がロードされたメモリ領域外に存在したとしても識別可能であり，2．3 節の前提 1 に依存せず API のコードの位置を把握できる．

また，テイントタグの伝搬，管理は OS とは独立して行われるので，OS のコンテキストとは独立して DLL の位置を追跡できる．つまり，2．3 節の前提 2 に依存せずに DLL の位置を把握できる．

これらの効果により配置位置難読化の解析妨害に対して，その影響を受けずに API の位置を特定することができ，この API のメモリ上の位置情報に基づき，API 名前解決を行うことで，配置位置難読化に強い IAT 再構築手法を実現できる．

3.3 システム構成と動作の流れ

図 5 に提案システムの構成と動作の流れを示す．提案システムは，動的解析部分とダンプ解析部分から構成される．

動的解析はマルウェアを実行させ，テイント解析を行い，メモリダンプ，シャドウメモリダンプ，ディスクイメージ，シャドウディスクダンプ（以下，ダンプファイル一式）を生成する．ダンプ解析は，動的解析により生成されたダンプファイルを解析し，IAT 再構築を行う．

提案システムの詳細な動作の流れを以下に示す．具体的には，前処理，動的解析，ダンプ解析（IAT 特定），ダンプ解析（API 名前解決）の 4 ステップで解析が行われる．

(1) **前処理** DLL 内の API コードのディスク上の位置を特定し，テイントタグを設定する．
(2) **動的解析** 動的解析システム上で解析対象マルウェアを一定期間動作させ，ダンプファイル一式を生成する．
(3) **ダンプ解析（IAT 特定）** メモリダンプを解析し，IAT の位置を特定する．
(4) **ダンプ解析（API 名前解決）** シャドウメモリダンプを解析し，IAT 内のポインタが差す API を特定し，その名前を得る．

前処理では，解析対象マルウェアを動作させるディスクイメージを走査し，監視対象の API を含む DLL ファイルのディスク上の位置を特定する．その後，DLL のヘッダ情報を読み込み，各 API の RVA を取得し，各 API のディスク上の位置を特定する．そして，各 API の位置に対応するシャドウディスクのエントリにその API を一意に識別可能なテイントタグを設定する．

動的解析では，動的解析システムを起動し，解析対象マルウェアを動作させ，テイント解析を行う．一定期間動作させた後，メモリダンプを取得する．また，同時にシャドウメモリ，シャドウディスクのテイントタグ情報をファイルに書き出す．このステップへの入力として，各 API にテイントタグが設定されたシャドウディスクとマルウェアの実行ファイルを受け取る．また出力として，ダンプファイル一式を書き出す．

ダンプ解析（IAT 特定）では，取得したメモリダンプの中から解析対象のプロセスを特定し，IAT を特定する．IAT の位置特定は従来の IAT 再構築手法と同じものを利用した．つまり，コード領域中の間接コール命令を見つけ出し，当該命令が参照するメモリ領域を IAT エントリ候補とする．IAT エントリ候補が集中するメモリ領域を IAT とする．

ダンプ解析（API 名前解決）では，前ステップで特定した IAT の各エントリの 4 バイトの値を API 名と関連づける．IAT のエントリ内のアドレスが指すメモリ領域に設定されているテイントタグをシャドウメモリから取得し，そのテイントタグと関連付けられた API 名を取得する．仮に，IAT が指すメモリアドレスを含むページがページアウトしていた場合，当ページを管理する VAD を取得し，ディスク上の位置を特定する．その位置情報に基づき，シャドウディスクダンプからテイントタグを取得する．このステップの出力として，解析対象プロセスの VAD ダンプと解決済み API 名を逆アセンブルコード上にマッピングする IDC（IDA Script) を書き出す．

3.4 実装

ここでは，提案システムの実装について述べる．提案システムは，前処理に The Sleuth Kit（TSK）[3] を利用し，動的解析に API Chaser を利用し，ダンプ解析に TaintVolatility を利用している．

3.4.1 前処理：TSK

提案システムでは，前処理ステップにおいて，ゲスト OS がインストールされたディスクイメージファイル中の DLL の位置を特定するため，TSK を利用した．TSK を利用し，ディスクイメージ上にインストールされたファイルシステムを解析し，対象とする DLL のセクタ番号，オフセット，サイズを取得する．その後，DLL の PE ヘッダ部分を取り出し，エクスポートテーブルを走査し，当該 DLL がエクスポートしている API 一覧とその RVA を取得し，各 API のディスク上での位置を特定する．上記の前処理はすべて動的解析を開始する前に行う．

3.4.2 動的解析：API Chaser

提案システムの動的解析環境として，API Chaser を利用した．API Chaser は，QEMU[2]（Argos[10]）をベースに開発されたテイント解析機能を有する動的解析システムである．仮想 CPU 内で実行される機械語命令に設定されているテイントタグをチェックし，そのタグに基づき解析対象コードの実行か否かを判断するコードテイント機能を持つ．更に，API コード毎に設定されたテイントタグとマルウェアのコードに設定されたテイントタグのそれぞれが設定された命令間の実行の遷移を捉え，API コールを識別することで API 監視を行う．

テイントタグの伝搬ルールとして，Argos と同様のものを利用した．つまり，データの移動とコピーに関しては移動，コピー元オペランドのテイントタグを，移動，コピー先のオペランドへ伝搬させる．また，1 項算術演算の場合は，元のオペランドのテイントタグを保持し，2 項の場合は，どちらか一方のオペランドのテイントタグを演算結果へ伝搬させる．

また，解析対象マルウェアを一定時間実行後，マルウェアの実行を停止し，物理メモリ，シャドウメモリ，シャドウディスクの各々のダンプを取得する機能を追加した．さらに，一定時間の経過を待たずにマルウェアが終了する場合を考慮し，解析対象マルウェアの実行プロセスが停止する際にもダンプファイル一式を自動的に取得できるようにした．

3.4.3 ダンプ解析：TaintVolatility

ダンプ解析のため，動的解析ステップで生成されたダンプファイル一式を入力として読み込み IAT 再構築を行う

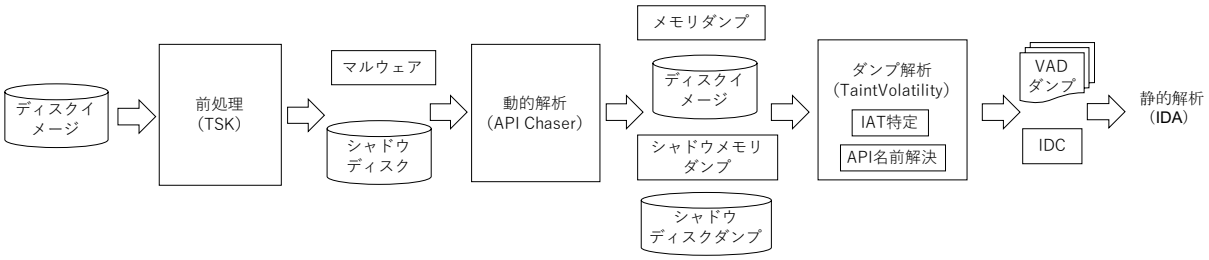

図 5 提案システムの動作の流れ
Fig. 5 The Workflow of Our System

TaintVolatility を実装した．

TaintVolatility は Volatility を拡張したものである．拡張点としては，ディスクイメージファイル，シャドウメモリダンプ，シャドウディスクダンプを読み込むオプションを追加した．また，IAT の位置特定には impscan の実装を利用した．API 名前解決において，impscan は PEB からロードされた DLL の一覧情報とそれぞれのロードアドレスを取得するが，TaintVolatility では，シャドウメモリダンプ，シャドウディスクダンプから該当するアドレスのテイントタグ情報を参照し，そのタグと関連付けられた API 名を取得する．

また，遅延ロードにより，まだメモリ上にロードされていないメモリページを解析するため，メモリフォレンジックスとディスクフォレンジックスを連携させた．具体的には，Volatility でメモリを解析中，ロードされていないメモリページを見つけた場合，そのメモリページに対応する VAD を見つけ，そこにマップされているファイルのパスを特定する．その後，ファイルオフセットを計算し，対応するページ部分を取得する．ディスクイメージの解析には TSK を利用し，Volatility からアクセスするために TSK の Python ラッパーである Pytsk[11] を利用した．

ダンプ解析ステップの出力として，各 VAD が指す領域のダンプを Volatility のプラグインである vaddump を利用して生成する．また，IDC の出力は impscan のコードを再利用した．

4. 実験

ここでは，本提案システムの有効性を示すために行った実験について説明する．まず，実験方法と実験に利用したツールについて説明する．次に，実験結果について説明する．

4.1 実験方法

この実験の目的は，２章で説明した配置位置難読化に対して提案システムが適切に処理可能だと示すことである．

実験用に４つの Windows 実行ファイルを用意し，それぞれに対して，Stolen Code, Copied API Obfuscation, DLL Unlinking, Stealth Loader を適用し，実験用のデータセットとした．また，提案システムの比較対象として，impscan, Scylla[9] を用意した．さらに impscan を独自に拡張し，ロードされた DLL の配置位置を VAD から見つけ出す impscan++も用意した．

実験手順としては，以下の通りである．

(1) ４つの Windows 実行ファイル（calc.exe, winmine.exe, cmd.exe, notepad.exe）に関して，それぞれがインポートしている API 一覧（インポート API 一覧）を予め取得しておく．
(2) ３つの既存ツールと提案システムの計４つのシステムで，４種の難読化が施された４つの Windows（合計 16 実行ファイル）に対して IAT 再構築を行う．
(3) IAT 再構築により得られた API 一覧と（1）で取得しておいた API 一覧と一致するかを確認する．

4.2 実験結果

表 1 に実験結果を示す．提案システムはすべての解析妨害手法に影響されることなくインポート API 一覧をすべて取得することができた．impscan と Scylla はすべての解析妨害に影響を受け，インポート API 一覧の取得に失敗した．一方，impscan++は DLL Unlinking のみを回避できた．これは，DLL Unlinking は PEB の DLL 情報管理データ構造体から特定の DLL のエントリを消すのに対して，impscan++は DLL の情報を VAD から取得しているため，DLL Unlinking の影響を受けなかったためである．

5. 関連研究

ここでは，本論文と関連する既存研究について取り上げ，本研究との違いや関係についてに述べる．

API Chaser[6] は著者らによって提案された，テイント解析を利用したマルウェア動的解析システムである．本論文の基本的なアイデアは API Chaser のテイントタグによる API コール特定の仕組みを静的解析用に拡張したものである．API Chaser は動的解析システムであるため，実行されたコード部分しか API を特定することはできない．一方で，本論文の手法は動的解析で実行されたなかった API に関しても識別可能である．

Eureka[12] は，オリジナルコードの Control Flow を解

表 1 実験結果
Table 1 Experimental Result

-	Stolen Code	Copied API	DLL Unlinking	Stealth Loader
提案システム	✓	✓	✓	✓
impscan	-	-	-	-
impscan++	-	-	✓	-
Scylla	-	-	-	-

「✓」は解析妨害に影響を受けずにインポートしている API をすべて取得できたことを意味する．一方で，「-」は影響を受けてしまい，API の取得に失敗したことを意味する．

析し，呼ばれている API を特定する．DLL とアドレスの関連づけは，NtMapViewOfSection API を監視し，DLL ファイルがメモリにマップされる度にそのアドレスとファイルを対応づけることで実現している．Eureka は DLL 単位で API の位置を特定しているため，API 配置位置難読化には脆弱である．また，DLL 配置位置難読化に関しては，DLL がロードされる際に，メモリアドレスを取得しているため，PEB から特定の DLL のエントリを隠蔽する DLL Unlinking に対しては耐性があるものの，そもそもマップ関数を利用せずに DLL をロードする Stealth Loader に対しては脆弱である．一方，実験で示したように本提案システムはこれらの解析妨害にも影響を受けない．

Choi ら [5] は，API コードに対するメモリアクセスを監視し，コードのコピー先を特定することで，Stolen Code や Copied API Obfuscation に対処する．本提案手法とは実装は異なるが，バイト単位で API コードの移動を捉える点は我々の提案手法と類似している．実装方法として，Choi らはメモリアクセストレースを利用し，我々はテイント解析を利用している．しかし，Choi らの研究はそもそもメモリ上にロードされた DLL を正確に識別できるかといった点には言及されていない．例えば，Stealth Loader を利用された場合，メモリ上にロードされた DLL を正確に識別することが難しい．一方，提案手法では，Stealth Loader が処理を行う前にテイントタグをディスク上のファイルに対して設定し，それをディスク，メモリ間で伝搬させることで API の位置を追跡，特定している．そのため，例え，OS が DLL の位置を正確に管理できなかったとしても，テイントタグから API の位置を特定できる．

岩村ら [15] の研究は，可変長である x 8 6 命令列の逆アセンブルの不確実性の問題を指摘し，確率的に IAT の場所を見つけ出す手法を提案している．彼らの研究と本論文はどちらも IAT 再構築に関するものだが，対象としている問題が異なる．我々の提案は，API 名前解決に関するものであり，彼らの提案は IAT の場所特定に関するものである．提案システムの中で岩村らの研究成果を組み合わせることで，より解析妨害耐性を持った IAT 再構築手法が実現可能だと考える．

6. 考察

ここでは提案システムの制限について考察する．

6.1 制限

IAT と API コードの間にスタブを挟まれる解析妨害の場合（API Redirection[14]），本提案手法では API 名前解決ができないことがある．IAT のエントリにはスタブのアドレスが格納されており，スタブは API に設定したどのテイントタグも保持しない．そのため，単純に IAT エントリが指すメモリ領域のテイントタグを見るだけでは API 名前解決はできない．このようなマルウェアに対しては，Eureka[12] のように Control Flow を解析することで IAT エントリと API を結び付けられると考える．IAT エントリがスタブを指している場合，そのスタブから Control Flow を解析し，テイントタグを持つコードに到達した時点で，そのテイントタグを利用して API 名前解決を行うことで，IAT エントリと API コードを結び付けられる．

しかし，IAT と API コード間に，動的に呼び出す API 先を決定するスタブが利用された場合，静的な Control Flow の解析だけでは解決できない．例えば，スタブが API のディスパッチャとして動作し，各コール元の何らかの情報，例えばコール元アドレス，を元に呼び出される API が動的に決定される場合，静的解析だけではコール先の API を決定することができず，より高度な解析が必要となる．これに関しては，本論文の残課題としたい．

動的解析時に API の名前解決を行うコード部分まで実行が到達しなかった場合，本提案手法では IAT の再構築が行えない．解析妨害手法には，実行環境の特徴を捉えて，例えば仮想マシン環境であるか否かをチェックし，途中で実行を停止するものがある．このようなマルウェアを解析する際，オリジナルコードの IAT の API 名前解決を行うコードの前で実行が停止すると，IAT の中には API のアドレスが含まれていないため，API の名前解決を行うことができない．

DLL が静的リンクされた場合 [1]，動的解析時にテイントタグが設定されたディスク上の DLL をロードしないので，メモリ上に配置された解析対象コードの一部に含まれ

る API コードと API の名前を紐付けることができない．しかし，DLL の静的リンクは多くの制限事項を抱えている．例えば，バージョン間の依存関係の問題が発生する可能性が高いうえ，ファイルサイズも大きくなり可搬性も失う．他にも，ntdll.dll や kernel32.dll のようなシステムに近い DLL の場合，初期化がうまく動作しない可能性も高く，その場合利用できる API に大きな制限がかかる．

テイント解析の制限事項として，テイントタグの伝搬切れの問題がある．[4] で述べられているように，テイントタグの伝搬は，直接の依存関係のないコードフロー（Implicit Flow）ではうまく働かないことが知られている．仮に，マルウェアが API のコードをコピーする前に Implicit Flow のコードを通した場合，テイントタグの伝搬がそこで途切れてしまう．結果として，API のコードに関連付けられたテイントタグが失われてしまう．

7. まとめ

本論文では，動的解析時のテイント解析の情報を利用して，メモリダンプの解析を行うことで配置位置難読化に影響を受けずに IAT の再構築を行える手法を提案した．今回は，IAT 再構築のためにテイント解析の情報を利用したが，それ以外にも多くの応用が考えられる．従来のメモリフォレンジックスはメモリダンプを取得した時点でのコンテキスト情報しか得ることができない．しかし，テイントタグに実行時のコンテキスト情報やマルウェアの実行中に行った挙動等の情報を保持させ，メモリダンプ取得時にシャドウメモリダンプを同時に取得し，これをメモリダンプと共に解析することで，メモリフォレンジックスを高度化できると考える．

今後の予定として，残課題の解決とテイント解析の情報をよりフォレンジックスに応用する手法の研究を進めていきたい．

参考文献

[1] Abrath, B., Coppens, B., Volckaert, S. and De Sutter, B.: Obfuscating Windows DLLs, *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, IEEE, pp. 24–30 (2015).

[2] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator., *USENIX Annual Technical Conference, FREENIX Track*, USENIX, pp. 41–46 (2005).

[3] Carrier, B.: The Sleuth Kit(TSK), http://www.sleuthkit.org/ (online), available from ⟨http://www.sleuthkit.org/⟩ (accessed 2017-08-17).

[4] Cavallaro, L., Saxena, P. and Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment, *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, Berlin, Heidelberg, Springer-Verlag, pp. 143–163 (2008).

[5] Choi, S.: API Deobfuscator: Identifying Runtime-obfuscated API calls via Memory Access Analysis, *Black Hat Asia* (2015).

[6] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: API Chaser: Anti-analysis Resistant Malware Analyzer, *Research in Attacks, Intrusions, and Defenses: 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, pp. 123–143 (2013).

[7] Kawakoya, Y., Shioji, E., Otsuki, Y., Iwamura, M. and Yada, T.: Stealth Loader: Trace-free Program Loading for API Obfuscation, *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, Gerogia, September 18-20, 2017. Proceedings* (2017).

[8] Ligh, M. H., Case, A., Levy, J. and Walters, A.: *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, Wiley Publishing, 1st edition (2014).

[9] NtQuery: https://github.com/NtQuery/Scylla (online), available from ⟨https://github.com/NtQuery/Scylla⟩ (accessed 2017-08-17).

[10] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation, *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, New York, NY, USA, ACM, pp. 15–27 (2006).

[11] py4n6: https://github.com/py4n6/pytsk (online), available from ⟨https://github.com/py4n6/pytsk⟩ (accessed 2017-08-17).

[12] Sharif, M. I., Yegneswaran, V., Saidi, H., Porras, P. A. and Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis., *ESORICS* (Jajodia, S. and Lopez, J., eds.), Lecture Notes in Computer Science, Vol. 5283, Springer, pp. 481–500 (2008).

[13] Suenaga, M.: A Museum of API Obfuscation on Win32, *Symantec Security Response* (2009).

[14] Yason, M. V.: The Art of Unpacking, *Black Hat USA Briefings* (2007).

[15] 岩村　誠，川古谷裕平，針生剛男：IAT エントリ格納場所の特定方法，マルウェア対策研究人材育成ワークショップ 2011(MWS2011) (2011).