Computer Security Symposium 2017
23 - 25 October 2017

仮想計算機モニタを用いたWindows 10 64bit環境におけるスタックトレースの実現

山下 雄也[1,a)]　明田 修平[1]　瀧本 栄二[1]　齋藤 彰一[2]　毛利 公一[1]

概要：64bit マルウェアの出現に伴って 64bit マルウェアの解析が求められるようになった．我々は仮想化技術を用い，Windows 10 上で 64bit マルウェアの解析を可能とするシステムコールトレーサ Alkanet 10 を開発している．現在，マルウェアによるコードインジェクションの挙動追跡機能の Alkanet 10 への実装を試みているが，x64 における呼出し規約は x86 と異なりフレームポインタを利用しないため，スタックトレースの実現は容易ではない．本論文では，Windows の内部構造である VAD ツリーと PE ファイル内の.pdata セクションからスタックトレースを実現する方法を提案する．

キーワード：MWS，動的解析，スタックトレース，Windows 10

Implementation of Stack Trace on Windows 10 x64 using Virtual Machine Monitor

Yuya Yamashita[1,a)]　Shuhei Aketa[1]　Eiji Takimoto[1]　Shoichi Saito[2]　Koichi Mouri[1]

Abstract: Along with an advent of 64-bit malware, analysis of 64-bit malware is now required. We are developing Alkanet 10, which is a system call tracer for 64-bit malware analysis on Windows 10 using virtualization technology. At present, we are attempting to implement a stack trace on Alkanet 10 to trace code injection behavior by malware. However, it is not easy to realize the stack trace because the calling convention on x64 does not use a frame pointer unlike x86. In this paper, we propose a way to implement the stack trace using VAD tree and .pdata section in PE file.

Keywords: MWS, runtime analysis, stack trace, Windows 10

1. はじめに

近年，オペレーティングシステムの 64bit 環境の普及に伴い，マルウェアの 64bit 化が進んでいる [1]．実際に，64bit マルウェアによる被害も発生しており，今後被害はさらに増加していくと考えられる．マルウェアへの対策のためには，その挙動を明らかにする必要があり，64bit マルウェアの解析環境の構築が求められている．マルウェアの解析手法は，マルウェアを実際に動作させて挙動を追跡する動的解析とマルウェアのコードを逆アセンブルして挙動を解析する静的解析の 2 つに大別される．我々は，動的解析に着目し，Windows 上で動作するマルウェアの解析を可能とするシステムコールトレーサ Alkanet をこれまで開発してきた [2]．Alkanet は，仮想計算機モニタ (VMM; Virtual Machine Monitor) である BitVisor[3] の拡張機能として動作し，ゲスト OS である Windows XP 上のプロセスが発行したシステムコールをスレッド単位でトレースできる．また，我々は，解析対象となる Windows のバージョンを XP から 10 にするための Alkanet の拡張も行っている [4]．以下，本論文では，Windows XP x86 を対象とする Alkanet を Alkanet XP，Windows 10 x64 を対象とする Alkanet を

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525-8577, Japan
[2] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466-8555, Japan
[a)] yyamashita@asl.cs.ritsumei.ac.jp

© 2017 Information Processing Society of Japan

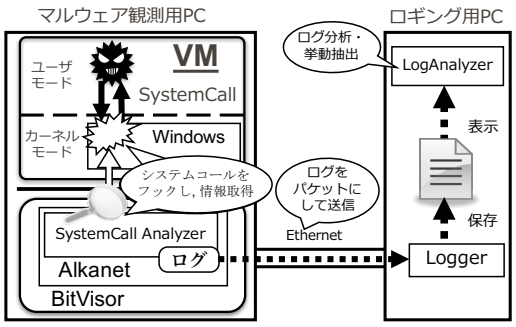

図 1 Alkanet の構成

Alkanet 10 と呼称する．

マルウェアの中には，他プロセスにコードを挿入し，そのプロセスに悪性な挙動を行わせることで自身の活動を隠蔽するものがある．しかし，一般にシステムコール・API トレース機能では，マルウェアが行うコードインジェクションに起因する他プロセスの挙動を関連づけて追跡できない．Alkanet は，この課題をスタックトレース機能によって解決している [5]．Alkanet にスタックトレース機能を実装することで，システムコール発行に至るまでの関数呼出し階層を取得でき，この呼出し階層とシステムコールトレースログを組み合わせることで，当該課題の解決が可能である．

Alkanet XP では，スタックに積まれているフレームポインタの値を順に辿ることでスタックトレースを実現している [5]．しかし，x64 における呼出し規約は x86 と異なりフレームポインタを使用しないため，同様の手法を Alkanet 10 に適用することができない．そのため，Alkanet 10 にスタックトレース機能を実装するためには，フレームポインタに頼らない手法を検討する必要がある．我々の調査では，Windows 10 x64 環境におけるスタックトレース実現手法を明らかにした研究は存在しない．以上の背景から，本論文では，Windows の内部データ構造である VAD ツリーと PE ファイル内の.pdata セクションから Windows 10 x64 環境上でスタックトレースを実現する手法を提案する．また，提案手法を Alkanet 10 に実装し，提案手法により Windows 10 x64 環境上でスタックトレースを実現できることを示す．

以下，本論文では 2 章で Alkanet 10 におけるスタックトレース実現の課題について述べ，3 章で提案手法について述べる．4 章で評価について述べ，5 章で今後の課題について述べる．最後に，6 章でまとめる．

2. Alkanet 10 におけるスタックトレース実現の課題

2.1 Alkanet の概要

Alkanet の構成を図 1 に示す．Alkanet は，VMM である BitVisor をベースとしたシステムコールトレーサで，ゲスト OS である Windows 上で動作するマルウェアが発行したシステムコールをトレースできる．Alkanet は，VMM をベースとすることで，多くのマルウェアが持つアンチデバッグ機能の回避を実現している．システムコールトレースは，システムコールの出入口にハードウェアブレイクポイントを設置し，処理をフックすることで実現している．システムコールのフック後は，発行されたシステムコールの引数や戻り値などの情報を取得し，Alkanet 内部のバッファにログとして保存する．保存したログは，Ethernet ケーブルを経由してロギング用 PC へ送信され，ログファイルに保存される．ログ取得後は，ログ解析ツールを用いることで，マルウェアの挙動に関するサマリを取得できる．

2.2 Alkanet XP でのスタックトレースの実現方法

Alkanet XP では，スタックに積まれているフレームポインタの値を順に辿ることでスタックトレースを実現している．Alkanet XP が対象とする Windows XP x86 では，Windows API の呼出しに stdcall 呼出し規約 [6] が使用される．この呼出し規約では，関数の先頭で現在の EBP の値を PUSH し，EBP を現在の ESP の値で上書きする．そのため，EBP が指す領域には前の EBP の値が存在し，その 4 バイト後ろにはリターンアドレスが存在する．したがって，EBP の値をスタックから順に取り出していき，その都度 EBP+4 のアドレスを読み出すことでスタックトレースを実現できる．

2.3 Alkanet 10 におけるスタックトレース実現の課題

Alkanet 10 が対象とする Windows 10 x64 では，Windows API の呼出しに Microsoft x64 呼出し規約 [7] が使用される．この呼出し規約では，stdcall 呼出し規約と異なりフレームポインタを使用しない．そのため，Alkanet 10 にスタックトレース機能を実装するためには，フレームポインタに頼らない別の手法を検討する必要がある．

3. 提案手法

3.1 提案手法の全体像

Microsoft x64 呼出し規約では，関数の先頭数命令（プロローグコード）で必要なスタック領域を確保し，関数の末尾数命令（エピローグコード）で確保したスタック領域を開放する．そのため，関数のプロローグコードの内容が分かれば，当該関数が消費するスタックフレームサイズを計算でき，フレームポインタに頼らないスタックトレースを実現できる．関数のプロローグコードの内容は，当該関数が定義されている PE ファイルの.pdata セクションから取得できる．提案手法では，この点に着目し，Windows 10 x64 環境におけるスタックトレースを実現する．

提案手法の全体像を図 2 に示す．提案手法は，以下の 7 つの手順から構成される．

© 2017 Information Processing Society of Japan

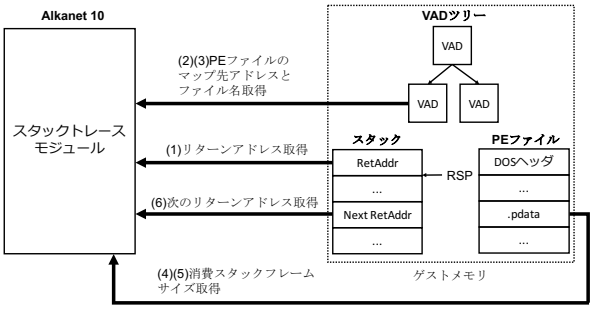

図 2　提案手法の全体像

手順 (1)　フック時にスタックの先頭に積まれているリターンアドレスを取得する．

手順 (2)　VAD（後述）ツリーを探索し，リターンアドレスをカバーするメモリ領域を見つける．

手順 (3)　そのメモリ領域の情報（VAD）から，呼出し元関数が定義されている PE ファイルのマップ先アドレスとファイル名を取得する．

手順 (4)　PE ファイルのマップ先アドレスから，当該 PE ファイルの.pdata セクションのアドレスを求める．

手順 (5)　.pdata セクションの情報から，呼出し元関数が消費するスタックフレームサイズを計算する．

手順 (6)　消費スタックフレームサイズの計算結果を用いて，スタック上の次のリターンアドレスを取得する．

手順 (7)　以下，手順 (2) から繰り返す．

上記手順それぞれの詳細については次節以降で述べる．

3.2　手順 (2) の詳細

3.2.1　VAD の概要

VAD (Virtual Address Descriptor) は，プロセスが確保したメモリ領域を管理するための Windows 10 内部データ構造である．VAD は，プロセスが動的メモリ割当てやファイルマッピングを行う毎に作成され，対応するメモリ領域を管理するための情報を保持する．VAD が保持する情報には，自身が管理する仮想アドレスの範囲やその領域にマップされているファイル情報などがある．

同一仮想アドレス空間に属する VAD は，互いに連結され平衡二分探索木を構成する．また，平衡二分探索木のルートノードのアドレスは，EPROCESS 構造体のメンバ VadRoot に格納される．本論文では，当該平衡二分探索木を VAD ツリーと呼称する．

3.2.2　VAD ツリーの探索に必要な情報

VAD ツリーの探索処理を実現するためには，各 VAD が持つ以下の 2 つの情報を取得する必要がある．

- 管理対象の仮想アドレス範囲
- 子ノードのアドレス

これら 2 つの情報を取得できれば，以下のように VAD ツリーの探索を行うことができる．

```
kd> dt nt!_MMVAD
   +0x000 Core             : _MMVAD_SHORT
   +0x040 u2               : <unnamed-tag>
   +0x048 Subsection       : Ptr64 _SUBSECTION
   +0x050 FirstPrototypePte : Ptr64 _MMPTE
   +0x058 LastContiguousPte : Ptr64 _MMPTE
   +0x060 ViewLinks        : _LIST_ENTRY
   +0x070 VadsProcess      : Ptr64 _EPROCESS
   +0x078 u4               : <unnamed-tag>
   +0x080 FileObject       : Ptr64 _FILE_OBJECT

kd> dt nt!_MMVAD_SHORT
   +0x000 VadNode          : _RTL_BALANCED_NODE
   +0x000 NextVad          : Ptr64 _MMVAD_SHORT
   +0x018 StartingVpn      : Uint4B
   +0x01c EndingVpn        : Uint4B
   +0x020 StartingVpnHigh  : UChar
   +0x021 EndingVpnHigh    : UChar
   +0x022 CommitChargeHigh : UChar
   +0x023 SpareNT64VadUChar : UChar
   +0x024 ReferenceCount   : Int4B
   +0x028 PushLock         : _EX_PUSH_LOCK
   +0x030 u                : <unnamed-tag>
   +0x034 u1               : <unnamed-tag>
   +0x038 EventList        : Ptr64 _MI_VAD_EVENT_BLOCK

kd> dt nt!_RTL_BALANCED_NODE
   +0x000 Children         : [2] Ptr64 _RTL_BALANCED_NODE
   +0x000 Left             : Ptr64 _RTL_BALANCED_NODE
   +0x008 Right            : Ptr64 _RTL_BALANCED_NODE
   +0x010 Red              : Pos 0, 1 Bit
   +0x010 Balance          : Pos 0, 2 Bits
   +0x010 ParentValue      : Uint8B
```

図 3　MMVAD 構造体，MMVAD_SHORT 構造体，RTL_BALANCED_NODE 構造体のメンバ

(a) 手順 (1) により，リターンアドレス RA を取得する．

(b) EPROCESS.VadRoot から VAD ツリーのルートノードのアドレスを取得する．

(c) ルートノードの VAD が管理する仮想アドレス範囲を取得する．この仮想アドレス範囲に RA が含まれていれば探索終了である．RA が含まれておらず，かつ RA が仮想アドレス範囲より小さければ左子ノードを辿り，RA が仮想アドレス範囲より大きければ右子ノードを辿る．

(d) 以下，目的となる VAD が見つかるまで子ノードを辿る．

前述した 2 つの情報は，VAD を表現する構造体である MMVAD 構造体のメンバから取得できる．MMVAD 構造体のメンバを図 3 に示す．なお，本節では，構造体のメンバを示す際に WinDbg[8] の dt コマンドの出力結果を用いる．出力の各列の意味は，左から順にオフセット，メンバ名，メンバの型である．

3.2.3　管理対象の仮想アドレス範囲の取得方法

VAD が管理する仮想アドレスの範囲は，MMVAD.Core のメンバに格納される．MMVAD.Core の型である MMVAD_SHORT 構造体のメンバを図 3 に示す．この構造体のメンバのうち，StartingVpn と StartingVpnHigh が管理対象の先頭仮想アドレスを保持し，EndingVpn と EndingVpnHigh が管理対象の終端仮想アドレスを保持する．管理対象の先頭仮想アドレスは，StartingVpn と StartingVpnHigh

© 2017 Information Processing Society of Japan

に対して以下の演算を行うことで求められる．

$$((StartingVpnHigh << 32)|StartingVpn) << 12$$

終端仮想アドレスについても，EndingVpn と EndingVpnHigh に対して同様の演算を行うことで求められる．

3.2.4 子ノードのアドレスの取得方法

子ノードのアドレスは，MMVAD.Core.VadNode のメンバに格納される．MMVAD.Core.VadNode の型である RTL_BALANCED_NODE 構造体のメンバを図 3 に示す．この構造体のメンバのうち，Left が左子ノードのアドレスを保持し，Right が右子ノードのアドレスを保持する．すなわち，Left から左子ノードの MMVAD 構造体のアドレスを取得でき，Right から右子ノードの MMVAD 構造体のアドレスを取得できる．

3.3 手順 (3) の詳細

3.3.1 呼出し元関数が定義されている PE ファイルのマップ先アドレス取得方法

呼出し元関数が定義されている PE ファイルのマップ先アドレスは，手順 (2) で取得した VAD が管理する先頭仮想アドレスと同一である．PE ファイルからプロセスを生成した際，当該 PE ファイルはメモリ上にマップされる．このとき，Windows 10 はマッピング領域を管理する VAD を 1 つ作成する．すなわち，ここで作成される VAD が管理する先頭仮想アドレスは，PE ファイルのマップ先アドレスとなる．手順 (2) で取得した VAD は，リターンアドレスをカバーする VAD であるため，呼出し元関数が定義されている PE ファイルのマッピング領域を管理する VAD である．

3.3.2 呼出し元関数が定義されている PE ファイルのファイル名取得方法

呼出し元関数が定義されている PE ファイルのファイル名は，手順 (2) で取得した MMVAD 構造体からメンバを辿っていくことで取得できる．具体的には，以下の方法で取得する．

(a) MMVAD.Subsection->ControlArea->FilePointer から EX_FAST_REF 構造体を取得する．以下，この構造体を efref とする．
(b) efref.Object & ~0xf を計算し，対応する FILE_OBJECT 構造体のアドレスを取得する．以下，このアドレスを fo とする．
(c) fo->FileName から目的のファイル名を取得できる．

3.4 手順 (4) の詳細

3.4.1 概要

手順 (4) は，PE ファイルに含まれる DOS ヘッダと PE ヘッダの情報を用いることで実現できる．64bit 対応の PE ファイルフォーマットである PE32+フォーマットの全体像 [9] を図 4 に示す．

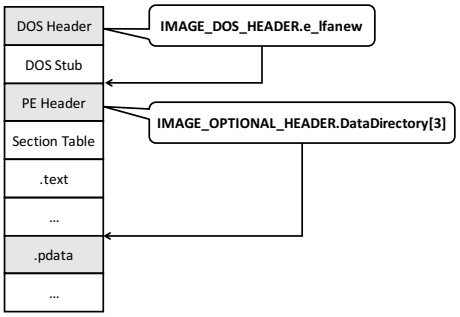

図 4 PE32+フォーマットの全体像と手順 (4) の概要

```
struct IMAGE_DOS_HEADER {
  WORD e_magic;
  WORD e_cblp;
  (省略)
  WORD e_res2[10];
  LONG e_lfanew;
};

struct IMAGE_NT_HEADERS64 {
  DWORD Signature;
  IMAGE_FILE_HEADER FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

図 5 DOS ヘッダ，PE ヘッダの構造

手順 (4) の概要を以下に示す．

(a) DOS ヘッダ内のフィールド e_lfanew を read し，PE ヘッダの先頭仮想アドレスを取得する．
(b) PE ヘッダ内のフィールドである配列 DataDirectory の添字 3 の要素を read し，.pdata セクションの先頭仮想アドレスとサイズを取得する．

次節以降では，上記処理に関連するヘッダである DOS ヘッダと PE ヘッダについて述べる．

3.4.2 DOS ヘッダ

DOS ヘッダは，PE ファイルの先頭に配置される領域であり，MS-DOS 上で当該 PE ファイルを実行する際に参照される．Windows 上で PE ファイルを実行する場合は，後述する e_lfanew 以外のフィールドは使用されない．

DOS ヘッダの構造を図 5 に示す．DOS ヘッダは，IMAGE_DOS_HEADER 構造体として表現される．手順 (4) を実現する上で重要となるメンバは e_lfanew である．e_lfanew は，PE ヘッダのアドレスの RVA (Relative Virtual Address) を保持する．ここで，RVA は PE ファイルのマップ先アドレスからのオフセットである．すなわち，手順 (2) で取得したマップ先アドレスに e_lfanew を加算することで，PE ヘッダの仮想アドレスを求めることができる．

3.4.3 PE ヘッダ

PE ヘッダは，PE ファイルに関する様々な情報を保持するヘッダである．PE ヘッダの構造を図 5 に示す．PE ヘッダは，IMAGE_NT_HEADERS64 構造体として表現さ

© 2017 Information Processing Society of Japan

```
struct IMAGE_OPTIONAL_HEADER64 {
  WORD Magic;
  BYTE MajorLinkerVersion;
  (省略)
  DWORD NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

図 6 IMAGE_OPTIONAL_HEADER64 構造体のメンバ

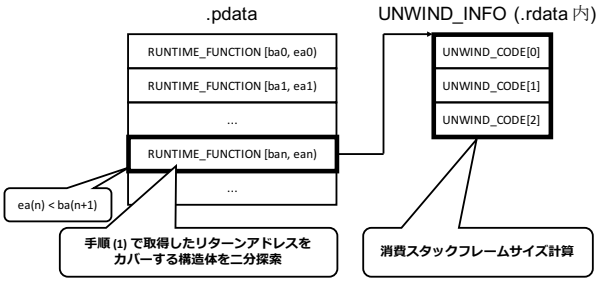

図 7 手順 (5) の全体像

れる．

手順 (4) を実現する上で重要となるメンバは OptionalHeader である．OptionalHeader は，Optional と付いているが PE ヘッダ内に必ず存在するメンバである．OptionalHeader の型である IMAGE_OPTIONAL_HEADER64 構造体のメンバを図 6 に示す．IMAGE_OPTIONAL_HEADER64 構造体のメンバである配列 DataDirectory の添字 3 の要素に，.pdata セクションの RVA とサイズが格納される．メンバ DataDirectory の各要素の型は IMAGE_DATA_DIRECTORY 構造体であるが，この構造体は RVA を表す DWORD 型のメンバ VirtualAddress とサイズを表す DWORD 型のメンバ Size から構成される．また，DataDirectory の添字 3 の要素には，.pdata セクションに関する情報が格納される．すなわち，DataDirectory の添字 3 の要素を dd3 とすると，dd3.VirtualAddress に.pdata セクションの RVA が，dd3.Size に.pdata セクションのサイズが格納される．したがって，これら 2 つのメンバを read することで，.pdata セクションの先頭仮想アドレスとサイズを取得することができる．

3.5 手順 (5) の詳細

3.5.1 概要

呼出し元関数が消費するスタックフレームサイズは，呼出し元関数のプロローグコードの内容から計算できる．関数のプロローグコードの内容は，.pdata セクションの情報を用いることで取得できる．手順 (5) では，.pdata セクションの情報を用いて，呼出し元関数が消費するスタックフレームサイズを計算する．手順 (5) の全体像を図 7 に示す．手順 (5) は，以下の 3 つの処理から構成される．

(a) 手順 (1) で取得したリターンアドレスをカバーする RUNTIME_FUNCTION 構造体を.pdata セクション内から探索する．
(b) RUNTIME_FUNCTION 構造体のメンバから対応する UNWIND_INFO 構造体のアドレスを取得する．
(c) UNWIND_INFO 構造体をパースし，関数の消費スタックフレームサイズを計算する．

次節以降では，上記処理それぞれの詳細について述べる．

```
struct RUNTIME_FUNCTION {
  ULONG BeginAddress;
  ULONG EndAddress;
  ULONG UnwindData;
};

struct UNWIND_INFO {
  UBYTE Version : 3;
  UBYTE Flags : 5;
  UBYTE SizeOfProlog;
  UBYTE CountOfUnwindCodes;
  UBYTE FrameRegister : 4;
  UBYTE FrameRegisterOffset : 4;
  // n == CountOfUnwindCodes
  UNWIND_CODE UnwindCodesArray[n];
  ULONG AddressOfExceptionHandler;
};

struct UNWIND_CODE {
  UBYTE OffsetInProlog;
  UBYTE UnwindOperationCode : 4;
  UBYTE OperationInfo : 4;
};
```

図 8 RUNTIME_FUNCTION 構造体，UNWIND_INFO 構造体，UNWIND_CODE 構造体のメンバ

3.5.2 .pdata セクションの構造と RUNTIME_FUNCTION 構造体

.pdata セクションは，RUNTIME_FUNCTION 構造体 [10] の配列から構成され，その要素数は PE ファイル内で定義されている関数の個数である．RUNTIME_FUNCTION 構造体は，PE ファイル内で定義されている関数と 1 対 1 で対応しており，対応する関数に関する例外処理用の情報を保持する．この情報の中に関数が消費するスタックフレームサイズが含まれている．

RUNTIME_FUNCTION 構造体のメンバを図 8 に示す．メンバ BeginAddress と EndAddress は，当該構造体に対応する関数命令列の開始アドレスの RVA と終了アドレスの RVA を保持する．メンバ UnwindData は，当該構造体に対応する UNWIND_INFO 構造体の RVA を保持する．UNWIND_INFO 構造体については次項で述べる．

手順 (5) では，手順 (1) で取得したリターンアドレスをカバーする RUNTIME_FUNCTION 構造体を探索する必要があるが，そのアルゴリズムには二分探索を使用できる．なぜなら，RUNTIME_FUNCTION 構造体は，対応する関数のアドレスに関して昇順で.pdata セクション内に配置されるためである．すなわち，n 個目の RUNTIME_FUNCTION 構造体のメンバ BeginAddress と EndAddress をそれぞれ $BA(n), EA(n)$ とすると，常に $EA(n) < BA(n+1)$ が成

© 2017 Information Processing Society of Japan

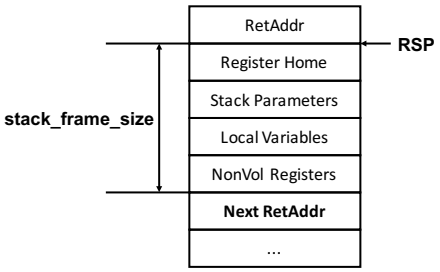

図 9　スタックフレームレイアウトと消費スタックフレームサイズの関係

り立つ．

3.5.3　UNWIND_INFO 構造体

UNWIND_INFO 構造体 [11] は，対応する関数に関する例外処理用情報をメンバとして保持する構造体である．UNWIND_INFO 構造体のメンバを図 8 に示す．

手順 (5) の実現に必要なメンバは UnwindCodesArray である．UnwindCodesArray は，UNWIND_CODE 型の配列であり，各要素がそれぞれ関数のプロローグコードにおける 1 命令を表す．すなわち，UnwindCodesArray から関数のプロローグコードの命令列を判別でき，その結果から関数の消費スタックフレームサイズを計算できる．

3.5.4　UNWIND_CODE 構造体

UNWIND_CODE 構造体 [12] は，プロローグコード内の 1 命令を表すための構造体である．UNWIND_CODE 構造体のメンバを図 8 に示す．

消費スタックフレームサイズの計算には，メンバ UnwindOperationCode とメンバ OperationInfo を使用する．これら 2 つのメンバには，対応する命令がどのような命令であるかが格納される．機械語命令と UnwindOperationCode，OperationInfo の対応を表 1 に示す．表 1 中の next slot は，UnwindCodesArray における次の要素の値を表す．すなわち，UNWIND_CODE 構造体 1 つ分の領域を使用して，オペレーションに応じた情報を表す．また，next two slots は，UnwindCodesArray における次の要素 2 つ分の値を表す．このように，UNWIND_CODE 構造体からプロローグコードにおける命令を判別でき，当該命令における RSP の減少量が分かる．したがって，UnwindCodesArray をイテレートし，UNWIND_CODE 構造体をパースすることで，プロローグコードにおける RSP の減少量の合計，すなわち関数の消費スタックフレームサイズを算出できる．

3.6　手順 (6) の詳細

スタックフレームレイアウトと消費スタックフレームサイズの関係を図 9 に示す．図 9 より，以下の処理を行うことで，スタック上の次のリターンアドレスを取得できる．

(1) RSP に 8 を加算する．
(2) RSP に手順 (5) で求めた消費スタックフレームサイズを加算する．
(3) RSP が指すアドレスから 8 バイト read する．この read 結果が次のリターンアドレスである．

4.　評価

4.1　評価目的と方法

提案手法により，Windows 10 64bit 環境におけるスタックトレースを実現できることを確認するために，評価を行った．本評価は，提案手法を Alkanet 10 に実装することで行った．具体的には，Alkanet 10 の SYSCALL フック時にスタックトレース処理を呼び出すよう実装した．ただし，現在 5 章で後述するページインの課題が解決できていないため，本評価では，WinDbg を用いてあらかじめ必要なページをページインさせておくこととする．

以下，具体的な評価方法について述べる．評価方法は前半部と後半部に分けられる．評価方法の前半部を以下に示す．

前半部: WinDbg によるスタックトレース

前半部 (1)　hello world プロセスに WinDbg をアタッチする．ここで，hello world プロセスは，標準 C ライブラリに含まれる printf 関数を用いて，”hello world” という文字列を標準出力に出力するプロセスである．

前半部 (2)　ntdll!NtWriteFile にブレークポイントを設置する．Windows 環境における printf は，最終的に NtWriteFile システムコールを発行する．本評価では，当該システムコール発行までの関数呼出し階層に着目する．

前半部 (3)　hello world プロセスを実行する．実行後，ntdll!NtWriteFile の先頭でブレークする．

前半部 (4)　WinDbg の k コマンドを実行し，スタックトレースを行う．ここで得られた結果を保存する．

評価方法の後半部を以下に示す．

後半部: 提案手法によるスタックトレース

後半部 (1)　hello world プロセスに WinDbg をアタッチする．

後半部 (2)　WinDbg の db コマンドを実行し，提案手法の実現に必要なセクションにアクセスする．これにより，アクセスしたページがページインされるため，5 節で述べた課題を回避できる．

後半部 (3)　Alkanet 10 によるシステムコールトレースを開始し，hello world プロセスを実行する．

後半部 (4)　Alkanet 10 による NtWriteFile フック時，提案手法を用いたスタックトレースを行う．スタックトレースでは，評価に必要な情報をデバッグログに出力する．具体的には，取得したリターンアドレスの RVA とマップファイルのファイル名を出力する．これら 2 つの情報と PDB ファイルに含まれるシンボル情報を用いることで，リターンアドレスを対応する関数名に変換できる．

© 2017 Information Processing Society of Japan

表 1　機械語命令と UnwindOperationCode，OperationInfo の対応 (主要なもののみ)

UwOpCode	オペレーション名	RSP の減少量	意味
0	UWOP_PUSH_NONVOL	8	不揮発性レジスタの退避 (PUSH)
1	UWOP_ALLOC_LARGE	OpInfo=0: next slot * 8 (136 to 512K - 8) OpInfo=1: next two slots * 8 (512K to 4G - 8)	スタック領域確保 (SUB)
2	UWOP_ALLOC_SMALL	OpInfo * 8 + 8 (8 to 128)	スタック領域確保 (SUB)
4	UWOP_SAVE_NONVOL	0	不揮発性レジスタの退避 (MOV)

後半部 (5)　デバッグログに出力した情報から，取得したそれぞれのリターンアドレスを関数名に変換する．ここで得られた結果と評価方法の前半部 (4) の結果が一致するか確認する．

4.2　評価結果と考察

評価方法の前半部 (4) の結果を図 10 に示す．図 10 より，NtWriteFile システムコール発行までの関数呼出し階層は，ntdll!RtlUserThreadStart から ntdll!NtWriteFile までの合計 20 個の関数から構成されている．また，WinDbg の k コマンドでは，インライン関数やラムダ式もトレースできている．

次に，評価方法の後半部 (5) の結果を図 11 に示す．図 11 の出力のうち，[ALKANET] で始まる行がデバッグログへの出力，#で始まる行が追記したコメントである．図 11 より，提案手法を用いたスタックトレースでは，インライン関数を除く全ての関数を正しくトレースできている．インライン関数をトレースできない理由は，インライン関数の呼出しには CALL 命令が使用されず，リターンアドレスがスタックに積まれないためである．しかし，文献 [13] では，インライン関数をトレースするための情報が PDB ファイル内に格納されると述べられている．そのため，スタックトレースのみではインライン関数をトレースできないが，PDB ファイル内の情報を活用することでトレースが可能になると考えられる．WinDbg の k コマンドは，インライン関数もトレースできているが，これは PDB ファイル内の情報を活用して実現していると考えられる．以上より，提案手法により Windows 10 64bit 環境におけるスタックトレースを実現できることを確認した．

5.　今後の課題

提案手法では，ゲストメモリを read し，その結果を用いてスタックトレースを行う．しかし，read 対象のページがメモリ上に存在しない場合，ゲストメモリの read に失敗し，それ以上処理を継続できないという課題がある．提案手法で対象としている Windows 10 は，オンデマンドページングを採用しているため，ページがメモリ上に存在しないケースは十分考えられる．

当該課題に対する解決策として，以下の 2 つを検討している．

(1) ゲストメモリの read 失敗時，ページフォルトをゲスト OS にインジェクションする．
(2) NtCreateProcess システムコールの処理をフックし，あらかじめ必要なページをページインする．

まず，解決策 (1) について述べる．解決策 (1) は，read に失敗する毎に Windows 10 のページフォルトハンドラを呼び出し，該当ページをページインさせる方法である．ページイン処理を行うページフォルトハンドラは，CR2 レジスタからページイン対象の仮想アドレスを判別する．そのため，ゲストメモリの read に失敗した場合は，CR2 に read できなかった仮想アドレスを書き込み，ゲスト OS にページフォルトをインジェクションすることで，Windows 10 のページフォルトハンドラに当該ページをページインさせることができると考えられる．

次に，解決策 (2) について述べる．解決策 (2) は，あらかじめ提案手法の実現に必要となるページをページインさせておく方法である．Windows 10 では，NtCreateProcess システムコールを用いてプロセスを生成する．そのため，NtCreateProcess システムコールの処理のうち，適切な部分にフックポイントを設置することで，提案手法の実現に必要となるページをページインさせた状態でプロセスを生成できると考えられる．

6.　おわりに

本論文では，仮想計算機モニタを用いた Windows 10 64bit 環境におけるスタックトレースの実現手法を提案した．また，提案手法を Alkanet 10 に実装し，評価を行った．評価の結果から，提案手法を用いることで Windows 10 64bit 環境上でスタックトレースを実現できることを確認した．

今後は，5 章で述べた課題を解決する．現在，この課題に対する解決策を 2 つ検討しているため，まずはこれら 2 つの解決策実現に向けた調査を行っていく予定である．

© 2017 Information Processing Society of Japan

```
0:000> k
#       Child-SP          RetAddr           Call Site
00      00000088`ae3ae128 00007ffb`2856d458 ntdll!NtWriteFile
01      00000088`ae3ae130 00007ff6`a6b446ce KERNELBASE!WriteFile+0x88
(省略)
06      00000088`ae3af720 00007ff6`a6b10c88 hello_world!__acrt_stdio_end_temporary_buffering_nolock+0x1e
07      (Inline Function) --------`-------- hello_world!__acrt_stdio_temporary_buffering_guard::{dtor}+0xb
08      00000088`ae3af750 00007ff6`a6b08726 hello_world!<lambda_a31071773b9a94e445cbd5d04ef99106>::operator()+0x104
(省略)
0e      00000088`ae3afd20 00007ff6`a6af621f hello_world!printf+0x41
0f      00000088`ae3afd60 00007ff6`a6af647d hello_world!main+0x1f
10      (Inline Function) --------`-------- hello_world!invoke_main+0x22
11      00000088`ae3afd90 00007ffb`28732d92 hello_world!__scrt_common_main_seh+0x11d
12      00000088`ae3afdd0 00007ffb`2b059f64 KERNEL32!BaseThreadInitThunk
13      00000088`ae3afe00 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

図 10 評価方法の前半部 (4) の結果

```
# ntdll!NtWriteFile
[ALKANET] #1 stack_trace: retaddr_rva: 935ca
[ALKANET] #1 stack_trace: filename: \Windows\System32\ntdll.dll
# KERNELBASE!WriteFile+0x88
[ALKANET] #1 stack_trace: retaddr_rva: 2d458
[ALKANET] #1 stack_trace: filename: \Windows\System32\KernelBase.dll
(省略)
# hello_world!__acrt_stdio_end_temporary_buffering_nolock+0x1e
[ALKANET] #1 stack_trace: retaddr_rva: 474ba
[ALKANET] #1 stack_trace: filename: \Users\yyamashita\hello_world.exe
# インライン関数hello_world!__acrt_stdio_temporary_buffering_guard::{dtor}+0xbが
# スキップされている
# hello_world!<lambda_a31071773b9a94e445cbd5d04ef99106>::operator()+0x104
[ALKANET] #1 stack_trace: retaddr_rva: 20c88
[ALKANET] #1 stack_trace: filename: \Users\yyamashita\hello_world.exe
(省略)
# hello_world!printf+0x41
[ALKANET] #1 stack_trace: retaddr_rva: 72e1
[ALKANET] #1 stack_trace: filename: \Users\yyamashita\hello_world.exe
# hello_world!main+0x1f
[ALKANET] #1 stack_trace: retaddr_rva: 621f
[ALKANET] #1 stack_trace: filename: \Users\yyamashita\hello_world.exe
# インライン関数hello_world!invoke_main+0x22がスキップされている
# hello_world!__scrt_common_main_seh+0x11d
[ALKANET] #1 stack_trace: retaddr_rva: 647d
[ALKANET] #1 stack_trace: filename: \Users\yyamashita\hello_world.exe
# KERNEL32!BaseThreadInitThunk+0x22
[ALKANET] #1 stack_trace: retaddr_rva: 12d92
[ALKANET] #1 stack_trace: filename: \Windows\System32\kernel32.dll
# ntdll!RtlUserThreadStart+0x34
[ALKANET] #1 stack_trace: retaddr_rva: 9f64
[ALKANET] #1 stack_trace: filename: \Windows\System32\ntdll.dll
```

図 11 評価方法の後半部 (5) の結果

参考文献

[1] Deep Instinct: Beware of the 64-bit Malware, http://info.deepinstinct.com/whitepaper-beware-of-the-64-bit-malware (2017).

[2] 大月勇人，瀧本栄二，齋藤彰一，毛利公一：マルウェア観測のための仮想計算機モニタを用いたシステムコールトレース手法，情報処理学会論文誌，Vol. 55, No. 9, pp. 2034–2046 (2014).

[3] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E. et al.: Bitvisor: a thin hypervisor for enforcing i/o device security, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ACM, pp. 121–130 (2009).

[4] 大月勇人，中野進，明田修平，瀧本栄二，齋藤彰一，毛利公一：Windows 10 x64 環境を対象とするシステムコールトレーサの実現手法，コンピュータセキュリティシンポジウム 2015 論文集， Vol. 2015, No. 3, pp. 839–846 (2015).

[5] 大月勇人，瀧本栄二，齋藤彰一，毛利公一：Alkanet におけるシステムコールの呼出し元動的リンクライブラリの特定手法，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. 753–760 (2013).

[6] Microsoft: __stdcall, https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx (2017).

[7] Microsoft: x64 Software Conventions, https://msdn.microsoft.com/en-US/library/7kcdt6fy.aspx (2017).

[8] Microsoft: Download Windows Debugger (WinDbg) tools - Windows Hardware Dev Center, https://developer.microsoft.com/en-us/windows/hardware/download-windbg (2017).

[9] Microsoft: Microsoft Portable Executable and Common Object File Format Specification, https://www.microsoft.com/en-us/download/details.aspx?id=19509 (2017).

[10] Microsoft: struct RUNTIME_FUNCTION, https://msdn.microsoft.com/en-us/library/ft9x1kdx.aspx (2017).

[11] Microsoft: struct UNWIND_INFO, https://msdn.microsoft.com/en-us/library/ft9x1kdx.asp://msdn.microsoft.com/en-us/library/ddssxxy8.aspx (2017).

[12] Microsoft: struct UNWIND_CODE, https://msdn.microsoft.com/en-us/library/ck9asaa9.aspx (2017).

[13] Microsoft: Debugging Optimized Code and Inline Functions — Microsoft Docs, https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-optimized-code-and-inline-functions-external (2017).

© 2017 Information Processing Society of Japan