ステルスデバッガを利用したマルウェア解析手法の提案

川古谷　裕平 †　　岩村　誠 †　　伊藤　光恭 †

†NTT 情報流通プラットフォーム研究所
180-8585 東京都武蔵野市緑町 3-9-11

{kawakoya.yuhei,iwamura.makoto,itoh.mitsutaka}@lab.ntt.co.jp

あらまし　本稿では、高度なアンチデバッグ機能を持ったマルウェアを効率よく解析できるステルスデバッガを提案する。仮想マシンを利用しゲスト OS の外側からデバッグ機能を提供し、仮想ハードウェアを操作することで従来の OS や CPU によるデバッグ支援機構には頼らないデバッグ機能を実現する。これにより、マルウェアが持つ様々なアンチデバッグ機能を無効化しつつ解析を行うことができる。さらにステルスデバッガではゲスト OS 上で動作するプログラムの特徴的な命令列の実行に基づきプログラムをブレークさせるシグネチャブレークポイント機能を実装した。これによりマルウェアのオリジナルエントリポイントへジャンプする瞬間の処理を捉えアンパッキングに応用する手法を提案する。研究用データセット CCC DATAset 2008 のマルウェア検体を利用してステルスデバッガの各機能の評価を行った。

Analyzing Malware with Stealth Debugger

Yuhei Kawakoya†　　Makoto Iwamura†　　Mitsutaka Itoh†

†NTT Information Sharing and Platform Laboratories
9-11, Midori-Cho 3-Chome, Musashino-shi, Tokyo 180-8588 Japan

{kawakoya.yuhei,iwamura.makoto,itoh.mitsutaka}@lab.ntt.co.jp

Abstract　In this papar,we introduce Stealth Debugger, an effective approach that stealthily debug malware that has anti-debug tricks. Stealth Debugger monitors programs running on guest OS from safe position, outside of OS and analyzes these programs by controlling virtual hardwares of the virtual machine monitor. Debugging functions of Stealth Debugger are totally independent from general OS and CPU debugging support. Therefore, we continue with analyzing malware without being detected by various malware debug detection techniques. In addtion, Stealth Debugger has signature breakpoint that capture characterictic instruction's execution of programs running on guest OS. The signature breakpoint is adopted for unpacking malware by capturing the moment of jumping to original entrypoint. To demonstrate its effectiveness, we implemente that system and evaluate it with CCC DATAset 2008 malware.

1　背景

マルウェアの詳細な動作を把握するにはデバッガや逆アセンブラを利用したリバースエンジニアリングによる静的解析が有効である。しかし、近年の高機能なマルウェアには自身が解析されるのを困難にするためパッカーやソフトウェアプロテクタにより難読化、暗号化の処理が施されている。これらパッカーやソフトウェアプロテクタの多くはデバッガや逆アセンブラによる

解析を困難にするため、様々なアンチデバッグ機能が搭載されている [1][2]。マルウェアの持っている機能を解析するには、これらアンチデバッグ機能を回避しつつ難読化や暗号化されている部分を復元し、オリジナルコードを抽出する作業、アンパッキング、を行う必要がある。

2　関連研究と問題点

マルウェアによるアンチデバッグ機能を回避し、オリジナルコードの抽出を行うため多くの研究が行われている。Olly Advanced[3] は OllyDbg のプラグインとして動作し様々なデバッガ検知機能を無効化しつつデバッグ作業を進めることができる。

またアンパッキングを目的としたツールでは、Universal PE Unpacker[4] や OllyBonE[5] などがある。Universal PE Unpacker は IDA pro のプラグインとして動作し、マルウェアのオリジナルコードの展開後の特徴な動作に着目し Original EntryPoint(以下、OEP) 部分の検出を行う。OllyBonE は OllyDbg のプラグインとして動作し、ページ単位でメモリ上にブレークポイントを設定し、そのページ上のコードを実行した瞬間にプログラムをブレークさせる。

上記のプラグインツールは有効ではあるが、マルウェアのアンチデバッグ機能を回避しきるのは困難である。その原因として以下の 2 点があげられる。 1) デバッガとマルウェアが同じ環境上、同じ権限で動作可能である。 2)CPU や OS によるデバッグ支援機構を利用しているためマルウェアによるデバッガの検出が容易である。

1) に関して、デバッガで解析する際、マルウェアをデバッガが動作している環境 (OS やハードウェア) と同じ環境で動作させなければならない。さらにマルウェアによっては OS と同等の権限を取得することも可能である。このため、マルウェアはその環境上での動作に自由度が高くなり様々な手法を利用してデバッガの検出が行えてしまう。また 2) に関し、デバッガは OS や CPU が提供するデバッグ支援機構を利用してマルウェアをデバッグする。これら OS や CPU によるデバッグ支援機構は元来ソフトウェアのバグを調査する目的に利用されるためステルスに動作するようには設計されていない。このためマルウェアにデバッガの存在を隠し切ることが難しくなってしまう。

3　ステルスデバッガの提案

本稿では上記の 2 点の問題を解決するステルスデバッガを提案する。ステルスデバッガは解析対象とは異なる環境上で解析対象よりも高い権限 (ring -1) で動作し、OS や CPU が提供するデバッグ支援機構を利用せずにデバッグ機能を実現している。このため、マルウェアによる各種アンチデバッグ機能に影響されず解析を進めることができる。

以下、ステルスデバッガの構成とデバッグコマンド一覧、ステルスデバッガが持つ特徴的な機能について説明する。

3.1　動作概要

図 1 にステルスデバッガの構成要素を示す。ステルスデバッガは仮想マシンモニタ、ゲスト OS、コントローラによって構成される。ゲスト OS には Windows や Linux などの汎用 OS を利用し、そのゲスト OS 内で解析対象を動作させる。仮想マシンモニタは一般的な PC が持つハードウェアをソフトウェアで表現しており、各種ハードウェアの中にデバッグ用のハードウェア制御機能を搭載している。解析者はコントローラ上からデバッグ用コマンドを発行し仮想ハードウェアを操作することで解析対象のデバッグを行う。仮想ハードウェアとは具体的には、CPU、物理メモリ、ハードディスク、BIOS などがある。

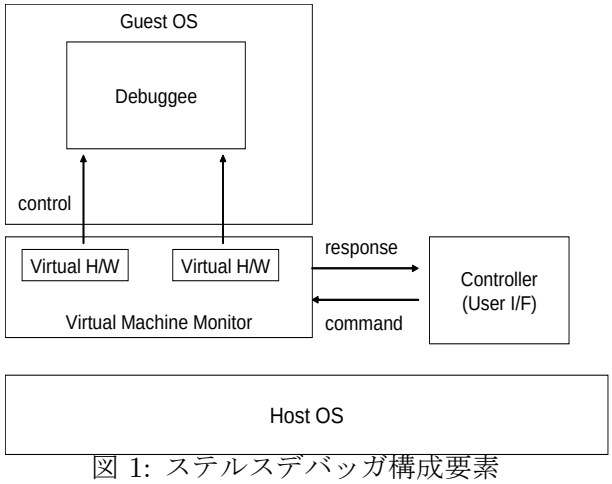

図 1: ステルスデバッガ構成要素

また実際の解析作業においては図 2 に示すようにゲスト OS 内に配置した解析用ソフトウェアも駆使して行うことを想定している。ステルスデバッガは強力なプログラム制御機能を提供するが、得られる情報がハードウェアレベルのプリミティブな情報になるためそれだけで解析対象の詳細な解析は困難である。そのため、ゲスト OS 内に配置した解析用ソフトウェアにより高いレイヤでの情報を収集し、ステルスデバッガで低レイヤからのプログラム制御と情報収集を行うことで効率よく解析を行うことができる。

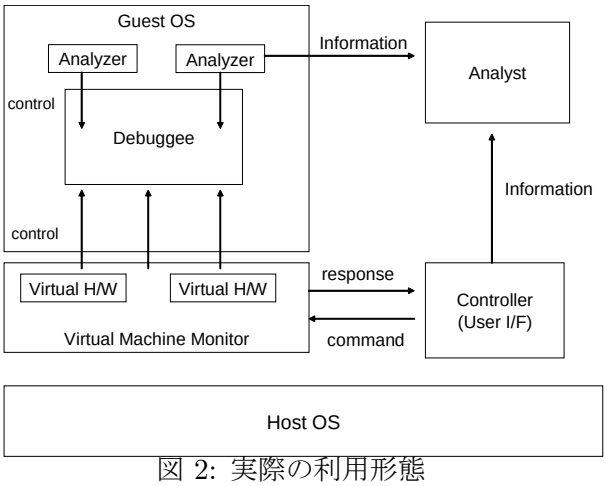

図 2: 実際の利用形態

3.2 デバッグコマンド

表 1 にステルスデバッガで利用可能なコマンド一覧を掲載する。一般的なデバッガで用いられるブレークポイントやトレース実行、シングルステップ実行、メモリ操作、レジスタ操作機能があり、これらは全て仮想ハードウェアを制御することで実装されている。それに加え、ステルスデバッガ独自のリソースアクセスモニタ機能、シグネチャブレークポイント機能を実装した。

3.3 タイムコントロール機能

高度なアンチデバッグ機能を備えたマルウェアの中には時間を利用し自身がデバッグされているかを判断するものがある。ある特定の関数の実行にかかる時間 (ミリセカンドやクロック数) を予め計測しておき、その関数実行時に計測した時間との実実行時間を比較する。デバッグを行いつつマルウェアを動作させるとブレークポイントによる実行の一時停止や、トレースログ出力のためのオーバヘッドがかかり、通常の動作よりも時間を浪費してしまう。そこでステルスデバッガでは、ブレークポイントで実行を一時停止させる場合やトレースログを取りログ出力を行う場合などのデバッグ操作中はゲスト OS 内の CPU クロックを完全に停止し、全ての仮想ハードウェアの動作を一時停止させる。これにより、デバッグ操作中にかかるオーバヘッドを隠蔽することができ時間を利用したアンチデバッグ機能を無効化している。

3.4 実行命令内容に基づくブレークポイント

ステルスデバッガでは、デバッグ機能の一つとして柔軟なブレークポイント設定を行うことができる。従来のデバッガでは、ブレークポイントはアドレスに基づき設定するものであった。具体的にはブレークさせたいアドレスの命令 1 バイトを 0xCC と置き換えることにより実現する。0xCC の命令を CPU がフェッチすることでデバッグ例外 (INT 01h) を発生させ OS のデバッグ例外ハンドラに制御を渡している。このようなアドレスに基づくブレークポイントでは予め実行アドレスを知っておく必要があるため、コードの動的展開、自己書き換え等を頻繁に行うマルウェアの解析には適していない。そこで、ステルスデバッガでは、従来のアドレスに基づくブレークポイントに加え実行命令の内容に基づくブレークポイントを実現している。具体的には、「ret 命令が実行された」などの事象を捉え、該当箇所でブレークさせることができる。また実効命令を組み合わせて指定することも可能であり「pop, pop, ret」といった一連の命令列の実行で解析対象をブレークさせることも可能である。以上のような柔軟なブレークポイントを利用することで解析対象の特徴的な命令列の実行を捉え解析を行うことができる。

3.5 リソースアクセスモニタ

ステルスデバッガでは仮想マシン上で動作しているプロセスの特権モードへ移行する命令の実行を捉えることでゲスト OS 内でのファイルリソースへのアクセスを監視している [6]。従来

分類	コマンド	概要
ブレークポイント	.setbp [address]	ブレークポイントの設定
	.delbp [address]	ブレークポイントの削除
	.showbp	ブレークポイントの一覧
	.setsig [signature]	シグネチャ(signature) の設定
	.setsig [signature]	シグネチャ(signature) の設定を解除
トレース	.trace start [pid] [output file]	実行命令トレース開始
	.trace end [pid] [output file]	実効命令トレース終了
ステップ実行	.ss	シングルステップ実行
	.tb	ブロックステップ実行
メモリ操作	.mem [address] [bytes]	address から bytes 分メモリ表示
	.write [address] [value]	address に値 (value) を書き込み
アクセスモニタ	.mon start	ファイルアクセス監視を開始
	.mon end	ファイルアクセス監視を終了
レジスタ操作	.showreg	レジスタ値の一覧
	.setreg [reg] [value]	レジスタ (reg) に値 (value) を設定

表 1: ステルスデバッガコマンド一覧

のモニタツールでは、モジュールやドライバをシステムやプロセス内に挿入し、特定関数の呼び出しを監視することでリソースへのアクセスモニタを実現していた。しかし、この手法ではドライバやモジュールを挿入するために、システム内のメモリやファイルの特定部分を書き換える必要があり、この変更箇所をチェックすることでマルウェアは容易にモニタされているかを判断することができた。

ステルスデバッガで行っているモニタは仮想 CPU 内で特権命令の実行監視し、それを契機にシステムリソース情報を調査する。システム内の如何なる箇所も改変していないため、仮想マシン上で動作しているプログラムからはモニタの存在を確認することはできない。このため、アンチデバッグ機能に検知されることなくモニタリングを行うことができる。

3.6 実装環境

表 2 ステルスデバッガを以下の環境上に実装し評価を行った。

CPU	Intel Core2 Quad 2.66GHz
メモリ	SDRAM 2GB
Host OS	Windows XP SP3
Guest OS	Windows XP SP0

表 2: 実装環境

4 評価

本稿で提案するステルスデバッガの評価として、研究用データセット CCC DATAset 2008 検体 (以下 CCC2008 検体) を利用した。CCC2008 検体は各種アンチデバッグ機能が搭載されており、一般的なデバッガでは解析を行うことが非常に困難である。またデバッガ検知だけではなく、動的解析で利用する Sysinternals[8] のモニタツールも CCC2008 検体実行時に存在が検知され利用することができない。

4.1 パッカーの特定

CCC2008 検体はデバッガ検知時に図 3 のようなダイアログを出現する。このダイアログを元に検索エンジンで調査した結果、利用されてい

るパッカーは SDProtector[7] だと推定される。

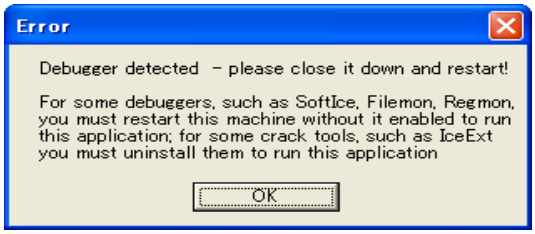

図 3: デバッガ検出時のダイアログ

4.2 ファイルアクセスの監視

最初に CCC2008 検体をステルスデバッガ上で動作させ、.mon コマンドによるファイルアクセスのモニタを行う。図 4 の.mon によるモニタ結果から以下のことが分かる。

- CCC2008 検体は 4 回の起動を行う
- User1\Local Settings\Temp\ フォルダの下に添付ファイルを作成し、実行時に参照している。
- 2、4 回目起動時に SICE,NTICE,FILEMON などのドライバにアクセスし存在確認を行っている
- 2 回目起動時に C:\Windows\System32\ 以下に NVCOM.EXE というファイルを作成している
- 4 回目起動時に C:\Windows\System32\drivers\etc\hosts ファイルにアクセスしている

```
[2020] C:¥DOCUME~1¥User1¥LOCALS~1¥Temp¥~temp02023628362.tmp
[2028] C:¥DOCUME~1¥User1¥LOCALS~1¥Temp¥~temp02023628362.tmp
[2028] C:¥share¥ccc_malware.exe
[2028] ¥SICE
[2028] ¥NTICE
[2028] ¥SIWDEBUG
[2028] ¥SIWVID
[2028] ¥FILEMON
...
[2028] ¥NTICE
[2028] C:¥share¥ccc_malware.exe
[2028] C:¥WINDOWS¥System32¥NVCOM.EXE
[112] C:¥DOCUME~1¥User1¥LOCALS~1¥Temp¥~temp02023628362.tmp
[120] C:¥DOCUME~1¥User1¥LOCALS~1¥Temp¥~temp02023628362.tmp
[120] C:¥WINDOWS¥System32¥NVCOM.EXE
[120] ¥SICE
[120] ¥NTICE
[120] ¥SIWDEBUG
[120] ¥SIWVID
...
[120] ¥NTICE
[120] C:¥WINDOWS¥System32¥drivers¥etc¥hosts
```

図 4: ファイルアクセスログ

4.3 実行トレース差分解析

次に.trace コマンドを利用して CCC2008 検体の全ての実行命令列を取得し、1 回目と 2 回目の起動のトレースログを比較した。1 回目と 2 回目の起動は最初は同じ命令実行を行うが、9,344,706 ステップ目から命令が異なる。以下に 1 回目と 2 回目のトレースログ結果を示す。

1 回目のトレースログ。

```
0x004a6575: jne   0x4a6593
0x004a6593: cmp   DWORD PTR [esp+28],0xc
0x004a6598: jae   0x4a6610
0x004a659a: call  0x4a6396
0x004a6396: rdtsc
0x004a6398: ret
```

2 回目のトレースログ。

```
0x004a6575: jne   0x4a6593
0x004a6593: cmp   DWORD PTR [esp+28],0xc
0x004a6598: jae   0x4a6610
0x004a6610: push  esi
0x004a6611: call  0x4a9cc6
0x004a9cc6: call  0x4a9ccc
```

上記のトレースログを比較すると、0x4a6593 における cmp 命令の結果が 1 回目と 2 回目の実行の差異の原因となっている。この周辺を静的解析して調査してみると、[esp+28] には temp02023628362.tmp ファイルを読み込んだ際の読み込みバイト数が格納されている。以上より、1 回目の起動の際は temp02023628362.tmp ファイルは存在しないので [esp+28] には 0 が入っている。2 回目の起動の際は、1 回目の起動で作成された temp02023628362.tmp ファイルを読み込みそのバイト数は 0Ch(12 バイト) 以上になり、異なる実行パスを通るようになっている。

4.4 実行命令の特徴に基づくアンパッキング

次に CCC2008 検体をアンパックするため、OEP の検出を行う。CCC2008 検体は SDProtector によりパッキングされていると想定される。SDProtector はアンパッキング処理の終了時に以下のような特徴的な命令列を実行し OEP へジャンプする。以下の例は Windows に標準搭載されているマインスイーパーを評価版 SDProtector でパッキングしたプログラムの OEP へのジャンプ箇所である。

```
0x0108f38e:  cmp    BYTE PTR [eax],0xe9
0x0108f391:  jne    0x108f30a
0x0108f397:  mov    BYTE PTR [eax],0xe8
```

```
0x0108f39a:  popf
0x0108f39b:  popa
0x0108f39c:  ret
0x01004055:  ret
～ret の繰り返し～
0x01003f48:  ret
0x01003e21:  push   0x70 <- OEP
0x01003e23:  push   0x1001390
```

この特徴的な命令列をステルスデバッガにシグネチャとして持たせブレークポイントを設定し実行を行った。これにより、2 回目の起動でブレークした。以下にブレークした箇所のプログラムを示す。

```
0x0049f389:  add    eax,0xffffff4b
0x0049f38e:  cmp    BYTE PTR [eax],0xe9
0x0049f391:  jne    0x49f30a
0x0049f397:  mov    BYTE PTR [eax],0xe8
0x0049f39a:  popf
0x0049f39b:  popa
0x0042bdcb:  ret
0x0042bdab:  ret
～ret の繰り返し～
0x00427dfd:  ret
0x00427d20:  call   0x4a5f52 <- OEP?
0x004a5f52:  call   0x49f3be
0x0049f3be:  call   0x49f3c4
```

以上により、OEP と想定される地点で検体の実行を停止し、検体のメモリダンプを取得した。以下にダンプした検体の文字列抽出結果を示す。これによるとボットのコマンドらしき文字列が見えるためアンパック処理は終了しているものと考えられる。

```
oo432200 JOIN %s %s\r\n
00432210 PONG %s\r\n
0043221C PING
00432228 NICK %s\r\nUSER %s 0 0 :%s
00432244 PASS %s\r\n
00432250 %d.%d.%d.%d
0043225C ipconfig.exe
0043226C /flushdns
0043227C irc.server
00432288 changes the server the bot
004322B0 irc.reconnect     ...
```

以後、Import Address Table を再構築すれば実際の CCC2008 検体のオリジナルコードの静的解析が可能となる。

4.5 マルウェアのアンチデバッグ機能

CCC2008 検体を解析した結果、CCC2008 検体には複数の以下のようなアンチデバッグ機能が備えられていた。ステルスデバッガを利用して解析を行うことでこれらのアンチデバッグ機能に検知されることなく、解析やモニタを進めることができる。

- API 呼び出しの 0xCC チェック
  パッカーの中から呼び出される API の先頭アドレスに 0xCC がセットされていないかをチェックする。
- 例外を利用した状態遷移
  独自で用意した例外ハンドラを登録し、意図的に例外を発生させることでデバッガによるトレースを困難にしている。
- シングルステップチェック
  自身がシングルステップ状態で実行されていないかを pushf、popf を利用してフラグチェックを行う。これによりデバッガによるトレースを困難にしている。
- VMware 検知
  自身が VMware 内で動作しているかをチェックする。VMware 内だと検知された場合は正常の実行パスと異なる実行パスを通りダイアログを出力する。

5 まとめ

本稿ではマルウェアの持つアンチデバッグ機能を無効化しつつ解析作業を行えるステルスデバッガについて提案した。ステルスデバッガはゲスト OS のさらに下の階層で動作し、従来の OS や CPU のデバッグ支援機構に頼らずにデバッグを行うことができる。これにより、マルウェアの様々なアンチデバッグ機能を回避しつつ解析を行うことが可能なことを示した。また、パッキングされたマルウェアが OEP へジャンプする直前の特徴的な動作をシグネチャで定義し、デバッガをブレークさせて捉えることで効率的にアンパッキングを行えることを示した。

参考文献

[1] Mark Vincent Yason. The Art of Unpacking.Blackhat USA 2007.

[2] Nicolas Falliere. Windows Anti-Debug Reference.http://www.securityfocus.com/ infocus/1893

[3] Olly Advanced. http://www.openrce.org/ downloads/details/241/Olly_Advanced

[4] Data Rescue. Universal PE Unpacker plug-in.http://www.hex-rays.com/ idapro/unpack_pe/unpacking.pdf

[5] OllyBonE. http://www.joestewart.org/ollybone/

[6] Yuhei Kawakoya.VM-Based Malware Detection System. 16th Usenix Security Symposium WiPs.

[7] SDProtector. http://www.sdprotector.com/

[8] Windows Sysinternals. http://technet.microsoft.com/en-us/ sysinternals/default.aspx