メモリ拡張によるアドレスに依存しないブレークポイント技術の提案

中山　心太† 　青木　一史† 　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
{nakayama.shinta , aoki.kazufumi , kawakoya.yuhei , iwamura.makoto , itoh.mitsutaka }@lab.ntt.co.jp

あらまし　近年，多数のマルウェアが出現しており，挙動解析の効率化が求められている．マルウェアの概要を把握する上で API トレースは有効な手法であるが，特定のアドレスへのアクセスで例外を発生させるようなブレークポイントを用いた API トレース手法は，stolen bytes と呼ばれるアンチデバッグ手法により，回避されてしまう．stolen bytes とは，API の全体もしくは一部をメモリ上にコピーし利用する手法である．本稿では，仮想マシンを用いてメモリ拡張を行い，特定のアドレスに設定したブレークポイントがメモリコピー時に伝播するブレークポイント技術を提案する．提案手法により，CCC Data Set2010 マルウェア検体の API トレースを収集し，その有効性の評価を行う．

Address independent breakpoint using extended memory function

Shinta Nakayama† Kazufumi Aoki† Yuhei Kawakoya† Makoto Iwamura† Mitsutaka Itoh†

†NTT information Sharing and Platform Laboratories
9-11, Midori-Cho 3-Chome, Musashino-shi, Tokyo 180-8585 Japan
{nakayama.shinta , aoki.kazufumi , kawakoya.yuhei , iwamura.makoto , itoh.mitsutaka }@lab.ntt.co.jp

Abstract Recent days, many malwares appeared, and efficiency improvement of malwares behavior analysis are requested. API trace is a effective technique to know the outline of malwares. But API trace technique that raise interrupt when access to the specific address are evaded by the anti debugging technique that is called “stolen bytes”. “stolen bytes” is a technique that copy whole or part of API functions to memory, and use it. In this paper, we propose breakpoint technique based on extend virtual machine's memory, that spread breakpoint when specified breakpoint to address are copied. We evaluated the proposed technique by the CCC DATA Set 2010 malware samples.

1　はじめに

マルウェアの詳細な解析を行うには，デバッガが用いられいる．デバッガでは，ブレークポイントを用いて任意の箇所でプログラムの動作を停止させ，解析を行う．しかしながら，昨今のマルウェアは高度化しており，ブレークポイントに対するアンチデバッグ機能を備えるものも少なくない．たとえば，stolen bytes と呼ばれるアンチデバッグ手法では，プログラムの一部をコピーして実行することで，ブレークポイントを設置した箇所を迂回されてしまう．

本稿では，マルウェアのアンチデバッグ機能によって検知・迂回されない，アドレスに依存しないブレークポイント技術の提案を行う．CCC DATA set 2010 マルウェア検体に対して，API トレースを行うことで，提案手法の評価を行った．

2　既存技術

既存のブレークポイントには，CPU の割り込み命令を利用したソフトウェアブレークポイント，CPU のデバッグレジスタを利用したハードウェアブレークポイント，川古谷らの仮想マシンを利用したブレークポイントがある．

2.1　ソフトウェアブレークポイント

ソフトウェアブレークポイントとは，プログラムを停止させたいアドレスの値とそのアドレスを記録し，割り込み命令(x86 アーキテクチャでは int 3)に置き換える．そしてプログラムが割り込み命令で停止した際に，どこのアドレスで停止したかを調べ，アドレスに対応する元の値に置き換えプログラムを再開する．API ト

レースは API 関数の先頭にソフトウェアブレークポイントを設置することで，実現することが出来る．

しかし，ソフトウェアブレークポイントを利用したブレークポイントは，命令を置き換えるため，プログラムのチェックサムを監視するようなアンチデバッグ機能に検知されてしまうという問題がある．

2.2 ハードウェアブレークポイント

ハードウェアブレークポイントとは，CPU のデバッグレジスタを利用したブレークポイントであり，メモリを改変しないため，チェックサムの監視等のアンチデバッグに検知されないという特徴がある．

しかし，一般にデバッグレジスタは数が限られており，Intel 社の x86 アーキテクチャ[5]では 4 つのアドレスしか指定することができない．そのため，たとえば WindowsXP SP0 の kernel32 には 928 個の API 関数が存在するため，これら API のすべてを監視することができない．

2.3 仮想マシンによるブレークポイント

川古谷らのステルスデバッガでは，仮想マシンを利用したブレークポイント手法が提案されている．

2.3.1 VM ベースハードウェアブレークポイント

VM ベースハードウェアブレークポイントは，VMM のレイヤでプログラムを停止させたいアドレスを記録しておくことにより，設置個数に上限がないハードウェアブレークポイントを実現することが出来る．そのため，プログラムのチェックサムを監視するようなアンチデバッグ手法は回避することが出来る．

しかしこの手法であっても，従来のブレークポイントと同様にアドレスに依存した手法であるため，stolen bytes と呼ばれるアンチデバッグ手法によってブレークポイントを迂回して任意の関数を実行できてしまう．

stolen bytes の例を図 1 に示す．関数の先頭を利用する stolen bytes では，マルウェアが本来の関数の先頭を，自己が確保した 0x2000 の領域にコピーし，その後ろに本来の関数に復帰するためのジャンプコードを置く．これにより，関数が実行されたかどうかを調べるために 0x1000 にブレークポイントを設置したとしても，0x2000 のアドレスを呼び出すことで，本来の関数と同等のプログラムが実行できてしまう．

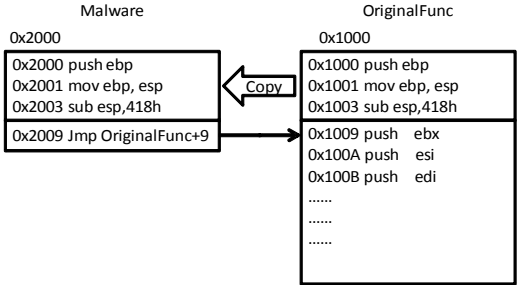

図 1:関数の先頭をコピーする stolen bytes の例

2.3.2 命令列に基づくブレークポイント

命令列に基づくブレークポイントは，あらかじめ設定しておいた命令列が現れた場合，プログラムを停止させるという手法である．これにより，アドレスに依存しないブレークポイントを設置することができる．

しかし図 1 のケースでは，本来の関数が実行する命令列とは異なり，ジャンプコードが間に挟まるため，仮想 CPU が実行する命令列が元の関数とは異なってしまい，プログラムを停止させることが出来ない．また，ブレークポイントとして設定する命令列を短くすることで，stolen bytes に対応することができるが，命令列を短くすることで，本来停止すべきではない箇所がブレークポイントとして検知されてしまう可能性がある．

3 アドレスに依存しないブレークポイントの提案

既存のブレークポイント技術には，マルウェアのアンチデバッグ機能によって，ブレークポイントが検知もしくは迂回されるという問題がある．そのため，マルウェアを効率的に解析することが難しい．

そこで本稿では，仮想マシンの利用により，マルウェアから検知されず，仮想マシンのメモリ拡張により，アドレスに依存しないブレークポイント技術を提案する．

本提案は仮想マシンの物理メモリとレジスタに対して，一対一に対応するブレークポイント用メモリを用意する．図 2 はこの様子を図示したものである．ブレークポイント用メモリは，ブレークポイントの情報を格納することができ，ブレークポイント情報はどこのアドレスに設置したブレークポイントかを示す識別子となる値を格納することができる．

ブレークポイント情報はゲスト OS の演算によって伝播が行われる．たとえばメモリ間の代入演算が行われると，代入演算元のメモリに対応するブレークポイント用メモリに格納さているブレークポイント情報を，代入先のメモリに対応するブレークポイント用メモリに上書きする．

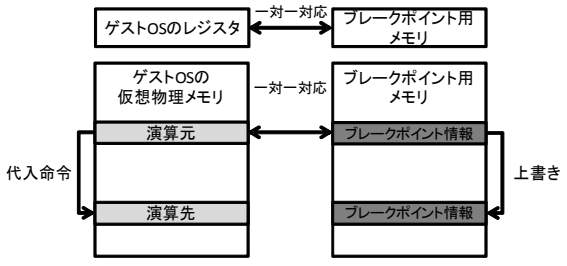

図 2:ブレークポイント情報用メモリの対応

ブレークポイントの伝播は表 1 のルールにしたがって行われる．仮想 CPU がゲスト OS の命令を実行す

る際に，命令を分析し，命令の種類，演算元の種類，演算先の種類を特定する．次に，演算先に対応するブレークポイント用メモリに対し，表 1 のルールに従って演算元のブレークポイント情報が伝播される．

表 1:ブレークポイント情報の伝播ルール

演算元＼命令の種類	代入	演算	その他
即値	消去	維持	維持
メモリ かレジスタ	上書き	合成	維持

ただし，x86 アーキテクチャではレジスタのゼロクリアに，xor eax,eax という命令が利用されることがある．この命令はレジスタ同士の演算命令であるが，mov eax,0 と等価であるため，即値の代入とみなし，ブレークポイントの情報をクリアする．これにより演算が行われ，プログラムがコピーされたとしても，ブレークポイント情報を正しく伝播させることができる．

ブレークポイントの検知は，プログラムが実行されるたびに，仮想 CPU のプログラムカウンタのアドレスに対応するブレークポイント用メモリに，ブレークポイント情報が存在するかを調べ，存在した場合，プログラムを停止させる．

4 提案手法を用いた API トレースシステムの実装

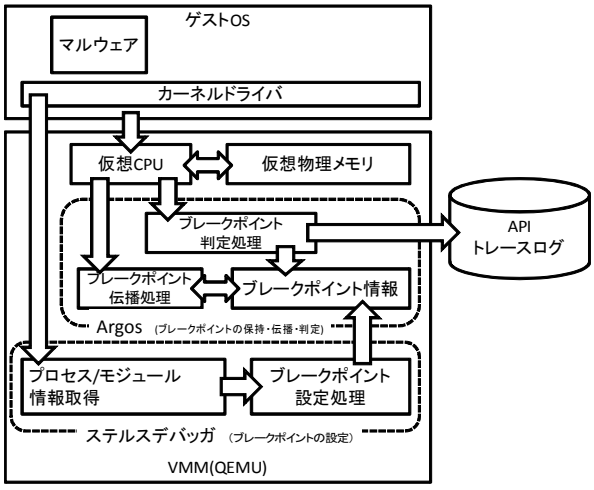

図 3:提案システムの構成図

API トレースとは，マルウェアが利用する API 調べることで，マルウェアの挙動の概要を把握するための自動解析の一手法である．

図 3 は提案システムの構成図である．ブレークポイントの保持・伝播・判定に Argos[2]を利用し，ブレークポイントの設定には川古谷らのステルスデバッガを利用した．

4.1 ブレークポイントの保持・伝播・判定

ブレークポイントの保持・伝播・判定はテイント解析機能を持った Argos を改良して実現した．

テイント解析とは，外部から入力されるデータをテイント，すなわち汚染されたデータであるとみなして，そのデータフローを追いかけることで，外部から入力されたデータがどのような効果を引き起こしたかを追跡する手法である．

Argos のテイント解析では，仮想物理メモリに対して一対一に対応するテイント解析用のメモリ領域を用意し，仮想 LAN デバイスが，受け取ったデータをメインメモリに書き込む際に，対応するテイント解析用メモリ領域にテイント情報を入力する．テイント情報は仮想 CPU が演算を行うつど，表 2 のルールに従い伝播が行われる．

表 2:テイント情報の伝播ルール

演算元＼命令の種類	代入	演算	その他
即値	消去	維持	維持
メモリ かレジスタ	上書き	OR 合成	維持

表 1 のルールと異なる箇所は，演算元がメモリかレジスタであり，命令の種類が演算の場合である．Argos のテイント情報は，汚染されているか否かの 1bit であるため，演算元か演算先のどちらかが汚染されていれば，演算先は汚染されているとする．

仮想 CPU が命令を実行するたびに，実行した命令にテイント情報が存在するかを確認し，テイント情報を伝播させる．これにより外部から送信されたデータが実行されたかどうかを検知することが出来る．

本提案手法の実装では，Argos のテイント情報の保持・伝播・判定機能をブレークポイント情報の保持・伝播・判定として利用した．実装するにあたり，Argos では，外部から来たデータか否かしか取り扱っておらず，非ゼロであれば 0xFF を代入するという処理が行われていたため，これを修正し，任意の値をブレークポイント情報の識別子として伝播可能にした．また，Arogs の仮想 LAN デバイスから受け取ったデータに対してテイント情報を入力する機能は，テイント情報をブレークポイント情報として利用するため無効化した．

4.2 ブレークポイントの設定

ブレークポイントの設定は，川古谷らが提案したステルスデバッガの機能を，Argos に実装することで実現した．

ステルスデバッガは VMM のレイヤでデバッガの機能を提供しているが，VMM レイヤの情報だけではプリミティブな情報しか利用することが出来ないため，たとえばデバッグに必要なプロセスごとの仮想アドレスの情報を利用することができない．そのため，ステルスデバッガではゲスト OS 内に配置したカーネルドライバとの連携が行われている．カーネルドライバは，ゲスト OS が発行するイベントに対してコールバック関

数を登録することで，ゲストOS内で発生したプロセスの生起等のイベントを監視し，VMMのレイヤに伝達する．VMMはカーネルドライバから送られたプロセス情報やモジュール情報にもとづき，任意のプロセスに対してアクセスすることが可能になる．

本提案では，ステルスデバッガのカーネルドライバを利用し，プロセスの生成と消滅，モジュールのロードを監視する．

本提案は仮想マシンの物理メモリに一対一対応したブレークポイント用メモリに対してブレークポイント情報を設置するため，モジュールがすべて物理メモリにマッピングされている必要がある．そのため，カーネルドライバでは，モジュールがロードされた際に，ロードされたメモリ領域に対して，一度読み込み処理を行うことで，確実にページインさせている．その後ロードされたモジュールのPIDと仮想アドレスをVMMに伝えることで，VMM側からブレークポイントを設置することが可能になる．

VMM側では，ロードされたモジュール名と関数一覧とオフセットを受け取り，モジュールの関数に対してブレークポイント情報を設置していく．設置したブレークポイントは，ブレークポイント情報の値と，設置したアドレスのペアのリストによって管理される．

プログラムカウンタのアドレスにブレークポイント情報が存在していた場合，ブレークポイント情報の値から，そのブレークポイントがもともと設置されていたアドレスを調べ，プログラムカウンタのアドレスと比較する．もし一致していれば，通常の方法で関数が呼び出されたと判断し，APIトレースログを出力する．異なっていれば，stolen bytesが発生したとして検知し，本来のAPI名を出力する．

OSが提供するAPIに対してブレークポイントを設置すると，APIはさまざまなプロセスで利用されるため，マルウェアとは関係のないプロセスでAPIトレースが行われてしまう．そこで，ステルスデバッガのプロセス監視の機能を用い，監視対象にあるプロセスのみAPIトレースログを出力するようにした．また，監視対象にあるプロセスが子プロセスを生成した場合，子プロセスは自動的に監視対象になる．

Argosの場合，仮想物理メモリ1バイトに対して，1バイトのテイント情報を割り当てているため，ブレークポイント情報は識別子として，0～255の1バイト分の値しか取れない．そのため，255個のブレークポイントしか設定することができず，限られたAPIしかブレークポイントを設定することができない．そこで，ブレークポイント情報の設定はプレフィックスとしてマジックナンバーの0xCCを埋め込み，その後ろに2バイトにブレークポイントの識別子を埋め込むようにした．これにより，2バイト分，65535個のブレークポイントを区別できるようになった．

なお，ブレークポイントを設定するAPIは，マルウェアが頻繁に利用するkernel32.dll，advapi32.dll，ntdll.dll，shell32.dll，user32.dll，ws2_32.dllからエクスポートされているAPI関数とした．

5　予備実験

本提案の効果を確認するために予備実験を行った．ホストOSとしてCentOS5.3を利用し，ゲストOSにはWindowsXP SP0を利用した．　また，ゲストOSはメモリのページアウトを防ぐために，ディスクキャッシュを無効にした．

```
1.  #include <windows.h>
2.  #include <stdio.h>

3.  typedef DWORD (WINAPI* GetVersionFP)(void);
4.  int main(void)
5.  {
6.      {
7.          DWORD version = GetVersion();
8.          printf("%x¥n", version);
9.      }
10.     {
11.         GetVersionFP fp = (GetVersionFP ) VirtualAlloc(
12.             NULL,
13.             1024,
14.             MEM_COMMIT,
15.             PAGE_EXECUTE_READWRITE
16.             );
17.         memcpy(fp, (void*)GetVersion, 100);
18.         DWORD version = fp();
19.         printf("%x¥n", version);
20.     }
21.     return 0;
22. }
```

図4:stolen byteのサンプルコード

予備実験では，図4のstolen bytesのサンプルコードを実装し，実験を行った．サンプルコードはWindowsのバージョンを求めるAPIであるGetVersionを正規の方法で呼び出す方法と，VirtualAllocを利用して，実行権限をつけたメモリ領域を確保し，GetVersionの関数をコピーし，実行する方法の二つからなる．なお最適化によって関数呼び出しが消されることを防ぐために，printfによって印字をしている．

実験の結果，一度目のGetVersionの呼び出しも，二度目の呼び出しも，共にGetVersionの呼び出しであると検知することができた．

二度目の呼び出しはブレークポイント情報の識別子から，本来のGetVersionのアドレスが調べられ，プ

ログラムカウンタのアドレスと異なっていたため，stolen bytes が発生していると検知された．

6 実験

実験は CCC DATA Set 2010 マルウェア検体[1]50 件と，参考情報として提供された，CCC DATA Set 2008 マルウェア検体1件，CCC DATA Set 2009 マルウェア検体 10 件の計 61 検体に対して，本提案手法による API トレースを行った．

実験の結果，2008 年度の検体から，API トレース中に stolen bytes が検知された．stolen bytes が発生した箇所に関する分析結果を図 5 に図示する．

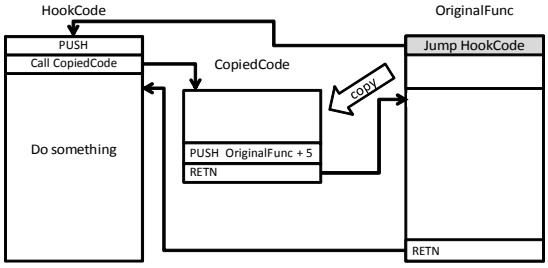

図 5:API フックの構造

API トレースの結果，実態としては stolen bytes ではなく，関数の先頭をコピーして利用するフックコードであった．API フックは元のコードをコピーして実行するため一種の stolen bytes といえる．そのため，stolen bytes の検知として，本提案手法は正しく機能しているといえる．

API トレースの結果，stolen bytes として検知されたのは，kernel32 が提供する API である FindFirstFileEx と FindNextFileW であった．表 3 は API トレースの結果の一部である．PID はプロセスの ID であり，Function name はブレークポイント情報の値から，もともと設置された API の名前を引いた結果である．isValid は，ブレークポイントで停止した箇所が，コピーされたブレークポイントか否かを示している．EIP はブレークポイントで停止した際の EIP のアドレスであり，Break point addr は本来のブレークポイントのアドレスである．

表 3:API トレース結果

PID	Function name	isValid	EIP	Break point addr
988	_GetSystemDirectoryW@8	Valid	77e6a961	77e6a961
988	_CreateFileW@28	Valid	77e779b1	77e779b1
988	_FindFirstFileW@8	Valid	77e78a39	77e78a39
988	_FindFirstFileExW@24	Copied	77f5036c	77e78848
988	_WriteFile@20	Valid	77e79d8c	77e79d8c
988	_WriteFile@20	Valid	77e79d8c	77e79d8c
988	_WriteFile@20	Valid	77e79d8c	77e79d8c
988	_FindNextFileW@8	Copied	77f503d8	77e7f2c4
988	_LoadLibraryA@4	Valid	77e805d8	77e805d8
988	_LoadLibraryExA@12	Valid	77e805b8	77e805b8

FindFirstFileEx 関数の本来の関数の先頭のバイトコードを図 6 に示す．図 7 にマルウェア感染後のバイトコードを示す．

Address	Hexdump	Dissassembly
77E78848	55	PUSH EBP
77E78849	8BEC	MOV EBP, ESP
77E7884B	81EC B4020000	SUB ESP, 2B4
77E78851	837D 0C 01	CMP DWORD PTR [EBP+C], 1
77E78855	53	PUSH EBX
77E78856	56	PUSH ESI
77E78857	57	PUSH EDI

図 6:本来の FindFirstFileEX

Address	Hexdump	Dissassembly
77E78848	E9 337B0D00	JMP 77F50380
77E7884D	B4 02	MOV AH, 2
77E7884F	0000	ADD [EAX], AL
77E78851	837D 0C 01	CMP DWORD PTR [EBP+C], 1
77E78855	53	PUSH EBX
77E78856	56	PUSH ESI
77E78857	57	PUSH EDI

図 7:マルウェア感染後の FindFirstFileEx

図 7 から，マルウェア感染によって関数の先頭が 5 バイトのジャンプコードに上書きされていることが分かる．ジャンプコードの飛び先を図 8 に示す．

Address	Hexdump	Dissassembly
77F50380	FF7424 18	PUSH DWORD PTR [ESP+18]
77F50384	FF7424 18	PUSH DWORD PTR [ESP+18]
77F50388	FF7424 18	PUSH DWORD PTR [ESP+18]
77F5038C	FF7424 18	PUSH DWORD PTR [ESP+18]
77F50390	FF7424 18	PUSH DWORD PTR [ESP+18]
77F50394	FF7424 18	PUSH DWORD PTR [ESP+18]
77F50398	E8 CFFFFFFF	CALL 77F5036C
77F5039D	83F8 FF	CMP EAX, -1

図 8:フック先のコード

ジャンプ先では，CALL 77F5036C で，stolen bytes が検知された箇所が呼び出されている．stolen bytes が検出された箇所を図 9 に示す．

Address	Hexdump	Dissassembly
77F5036C	55	PUSH EBP
77F5036D	8BEC	MOV EBP, ESP
77F5036F	81EC B4020000	SUB ESP, 2B4
77F50375	68 5188E777	PUSH 77E78851
77F5037A	C3	RETN

図 9:stolen bytes が検知された箇所

stolen bytes が検知された箇所は，本来の FindFirstFileEx 関数の先頭の 5 バイトと同一であることが分かる．その後，push reten によって，本来の FindFirstFileEx の関数の先頭から+5 バイトの地点へジャンプしている．

以上から，図 5 に図示された API フックが行われていたことがわかる．API フックでは API 関数の先頭を 5 バイトのジャンプコードで置き換えるため，先頭 5 バイトにかかる命令は実行されなくなってしまう．そのため，API フックから復帰したときのために，関数の先頭の命令をコピーし，実行する本来の関数に復帰する必要がある．したがって，3つの命令が本来の関数からコピーされたため，ブレークポイント情報が伝播され，stolen bytes として検知されたと考えられる．

なお，本来の関数の先頭アドレスが API トレースの結果に現れなかったのは，フックコードへのジャンプ命令によって，上書きされたため，表 1 のルールに従い，ブレークポイント情報の伝播が行われ，ブレークポイントが存在しないという情報によって上書きされたためである．

7 まとめ

本稿ではメモリトレースを利用した，アドレスに依存しないブレークポイントを提案した．実装はテイント解析機能を持った仮想マシンである Argos に，ゲスト OS のカーネルと連携することで，効率的にマルウェアの解析が行えるステルスデバッガの機能を移植することで実現された．これにより stolen byte によってコードがコピーされたとしても伝播するブレークポイント手法を実現することができた．

実験は CCC DATA Set 2010 マルウェア検体 50 件と，参考情報として提供された，CCC DATA Set 2008 マルウェア検体1件，CCC DATA Set 2009 マルウェア検体 10 件の計 61 検体に対して，本提案手法による API トレースを行った．実験の結果，2008 年の検体から stolen bytes が検知された．

2008 年の検体を精査したところ，stolen bytes ではなく，API フックが行われていたことがわかった．API フックでは，API の先頭をコピーして利用するため，stolen bytes と同様の処理が行われるため，これが stolen bytes として検出された．そのため，本手法の有効性を確認することが出来た．

本実験の範囲では，stolen byte を利用しているマルウェアは存在しなかった．そのため，引き続き継続して実験を行い，stolen bytes を利用しているマルウェアの調査を行う予定である．

8 今後の課題

本提案の実装は Argos をベースとしており，VMM の物理メモリ 1 バイトに対して，1 バイトのブレークポイント用メモリを割り当てており，255 個の識別子しか設定できない．これに対して，プレフィックスを置き，その後ろ 2 バイトにブレークポイントの種類を入力するようにした．しかしこの解決方法では，関数の先頭 1byte をコピーするような stolen bytes があった場合，正しく検知することができない．そのため，仮想物理メモリ 1byte に対して，2byte もしくは 4byte のブレークポイント用メモリの割り当てを可能にする必要がある．

参考文献

[1] 畑田 充弘，中津留 勇，秋山 満昭，三輪 信介，"マルウェア対策のための研究用データセット ～ MWS 2010 Datasets ～",2010

[2] Georgios Portokalidis and Asia Slowinska and Herbert Bos, "Argos: an Emulator for Fingerprinting Zero-Day Attacks",ACM SIGOPS EUROSYS'2006, 2006,

[3] 川古谷 裕平, 岩村 誠, 伊藤 光恭, "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",MWS2009, 2009

[4] 川古谷 裕平, 岩村 誠, 伊藤 光恭, "OEP 自動検出によるマルウェアアンパックの手法",信学技報, vol. 110, no. 79, ICSS2010-3, pp. 13-18, 2010 年 6 月．

[5] Intel; 64 and IA-32 Architectures Software Developer's Manuals, "http://www.intel.com/products/processor/manuals/" (2010 年 8 月確認)