エミュレーションに基づくシェルコード検知手法の改善

藤井 孝好　吉岡 克成　四方 順司　松本 勉

横浜国立大学

240-8501 横浜市保土ヶ谷区常盤台 79 番 7 号

あらまし マルウェアの感染活動等に用いられるリモートエクスプロイト攻撃への対策として，ネットワーク上のデータを機械語とみなしてエミュレータ上で実行し，その動作の特徴によりシェルコードの動的検知を行う手法が提案されている．これらの手法では，暗号化されたペイロードを実行時に復号する，いわゆるポリモーフィックシェルコードのみを検知対象としているため，実用においては，ポリモーフィックシェルコード以外のシェルコード(本報告では非ポリモーフィックシェルコードとよぶ)の検知が可能な手法との併用が必要である．一方，非ポリモーフィックシェルコードに対しては，これまで複数の静的検知手法が提案されているが，いずれも難読化による検知回避への対応が十分でなかった．そこで，本報告ではポリモーフィックシェルコードと非ポリモーフィックシェルコードのいずれもエミュレーションに基づき検知可能な手法を提案する．提案方式は，静的検査と動的検査を組み合わせることで実現しているため，全ての通信データを動的に検査する従来手法に比べて効率化も期待できる．

Improvement of Emulation-based Shellcode Detection

Takayoshi Fujii　Katsunari Yoshioka　Junji Shikata　Tsutomu Matsumoto

Yokohama National University

79-7 Tokiwadai, Hodogaya-ku, Yokohama 240-8501, Japan

Abstract Several dynamic shellcode detection methods, in which network traffic is examined by being executed as machine codes on light-weighted emulator to analyze its behavior, have been proposed as countermeasures against remote exploits. These previous methods, however, focus on detecting only a polymorphic shellcode, which decrypts its encrypted payload upon its execution, and therefore, require in practice a parallel use with an alternative method that detects a non-polymorphic shellcode. Although a number of static detection methods for a non-polymorphic shellcode have also been proposed, it is said that they have limitations on detecting a shellcode crafted by a series of obfuscation techniques. In this report, we propose a novel dynamic detection method that is able to detect not only a polymorphic shellcode but also an obfuscated non-polymorphic shellcode. Since the proposed method combines both static and dynamic detection, the efficiency can also be improved compared with a previous method that examines all traffic data by dynamic detection.

1.はじめに

近年，コンピュータウイルス，ワーム，ボット等，いわゆるマルウェアによるセキュリティ被害が深刻になっている．これらマルウェアがネットワークを介してコンピュータへ感染活動を行う手段のひとつに，感染先のネットワークサービスの脆弱性を突いて権限を奪取し，シェルコードとよばれる機械語のコードを実行させるリモートエクスプロイト攻撃がある．リモートエクスプロイト攻撃を防ぐため，保護対象のコンピュータにパケットが到達する前にネットワーク上で通信データを検査し，シェルコードを検知する様々な技術が研究開発されている．

このようなネットワークベースでのシェルコード検知方式のうち最も基本的なものとしてシェルコードの特徴的なバイトパターン(シグネチャ)と通信データを照合して検知を行うパターンマッチングがある[1]．また，単なるシグネチャとのマッチングではなく，通信データを機械語とみなして，逆アセンブルを行ったり，制御フローを解析してシェルコードの検知を行う静的検知手法が提案されている[2]．

これに対して，シグネチャや静的検知手法による検知を逃れるために，シェルコード本体を暗号化し，実行時に復号を行う**ポリモーフィックシェルコード**が多く用いられるようになった[3,4]．ポリモーフィックシェルコードについても，復号ルーチンの構造に着目した静的解析による検知手法[2]が提案されているが，復号ルーチンの難読化への対応が十分ではなかった．そこで，近年これらのポリモーフィックシェルコードによる検知回避の対抗策として，通信データを機械語と解釈してエミュレータ上で擬似的に実行し，その挙動に基づきシェルコードの特定を行う動的検知手法が提案されている[3,4,5]．

これらの動的検知手法ではポリモーフィックシェルコ

ードの検知を目的としているため，実用においては，ポリモーフィックシェルコード以外のシェルコード(以降は，**非ポリモーフィックシェルコード**とよぶ)の検知が可能な，シグネチャや静的検知手法との併用が必要となる．しかし，非ポリモーフィックシェルコードにおいても，シグネチャや静的検知を難読化により回避する手法[6]が依然存在しており，既存の静的検知手法[2,7]では，これらの難読化に対して十分に対応できていなかった．

そこで，本稿では，ポリモーフィックシェルコードと，難読化された非ポリモーフィックシェルコードの両方を，エミュレーションで検知できる方式を提案する．この手法では，まず検査対象の通信データに対する静的検査を行い，シェルコードの候補を検出する．これらの候補について，それぞれ，エミュレーションによる詳細な動的検査を行い，最終的なシェルコードの有無を判定する．動的検査は，静的検査による検出箇所の周辺のみで行うことで，効率的に検知を行う．

提案手法を評価するため，脆弱性検証ツールであるMetasploit[8]を用いたシェルコードの検知実験，ランダムなバイト列を用いた実験，および，Web サーバの通信データとCCC DATAset 2010[9]の攻撃通信データを用いたシェルコードの検知実験を行った．

本稿では，まず 2 章で提案方式が対象とするシェルコードについて説明し，3 章で提案方式，4 章で評価実験，5 章で関連研究，6 章でまとめと今後の課題を述べる．

2.検知対象のシェルコード

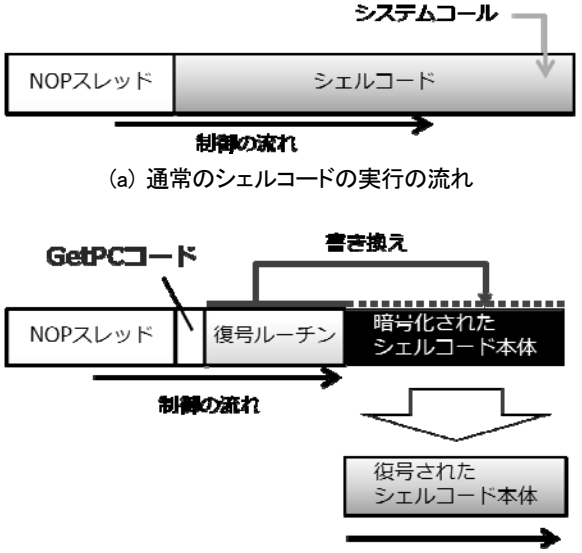
本研究では，x86(IA-32)系の機械語によって記述されたシェルコードを検知対象とする．また，提案方式はネットワークベースの検知手法であるため，攻撃対象の内部状態に依存した，汎用性の低いシェルコード(Non self-contained shellcode)は検知対象外とする[1]．以下では，提案方式が検知対象とするシェルコードについて説明する．

2.1 難読化されていないシェルコード

難読化されていないシェルコードの実行の流れを図1(a)に示す．バッファオーバーフロー等の脆弱性を突いた攻撃により攻撃対象のプログラムの権限が奪取される際，シェルコードの先頭アドレスに正確に制御を移すことは一般に困難である．そこで，**NOP スレッド**と呼ばれるシェルコードの実行上無意味な命令列を用意し，そのいずれのバイトに制御が移った場合にもシェルコードの実質的な先頭に制御が到達するようにする．最も基本的なNOPスレッドはNOP命令のみによって構成されるが，NOP命令以外の命令を用いて同様の効果を得る方法がある[10]．シェルコードに制御が移ると，レジスタやスタックの値を設定した後，**システムコール**を用いて OS またはカーネルの関数・機能を呼び出す．本稿では機械語の割り込み命令(INT 命令)および標準ライブラリの関数呼び出しを併せてシステムコールとよぶ．

2.2 ポリモーフィックシェルコード

ポリモーフィックシェルコードの実行の流れを図 1(b)に示す．ポリモーフィックシェルコードでは，メモリ上のデータ書き換えを行うため，絶対アドレスが必要となる．そこで，**GetPC コード**[11]と呼ばれる命令により，現在実行されている命令が配置されている絶対アドレス(プログラムカウンタ, PC) を取得する．GetPC コードとして利用される命令として，CALL 命令，および FPU 環境を保存する命令(FSAVE 等)がある．復号対象のデータのメモリ上の位置が特定できると，次に**復号ルーチン**において，暗号化されたペイロードの復号を行う．シェルコードには長さの制限があり複雑な復号処理を実現することは難しいため，ループ命令により 1 バイトから数バイト単位で書き換えが行われることが多い．復号処理後，復号されたシェルコードに制御が移り，実行が続けられる．なお，復号ルーチンに対して自己書き換え処理を施す方法も存在する[3,5].



(a) 通常のシェルコードの実行の流れ

(b) ポリモーフィックシェルコード

図 1 シェルコードの実行の流れ

2.3 その他の難読化手法

2.2 節で述べたポリモーフィックシェルコードとは異なる方法で難読化されたシェルコードも存在する．以下では，ポリモーフィズム以外の主な難読化方法について説明する．

間接ジャンプ　移動命令の中で，ジャンプ先アドレスを即値(定数)で直接指定せずに，命令時のレジスタに保存されている値を用いる方法を間接ジャンプという．

[1] 一部の Non self-contained shellcode に対応したネットワークベースの検知手法が提案されているが[4]，我々は攻撃対象の内部状態に依存した攻撃はホストベースの手法により対処すべきと考えるため，提案方式では Non self-contained shellcode は検知対象外とした．

例えば，jmp eax（EAX レジスタに保存されているアドレスへ移動する）のような命令が該当する．静的な逆アセンブルや制御フローグラフ（CFG）に基づく静的検知手法では，間接ジャンプ以前のコードの実行によるレジスタやメモリの状態を追跡しないため，間接ジャンプを含む処理の流れを正しく認識できない[6].

メタモーフィズム　メタモーフィズムは，命令の実行順の変更や等価な命令との置き換えにより，シェルコードの表層的表現を変化させる手法を指す[12]. これにより，シグネチャとのパターンマッチによる検知を困難にする．例えば，xor eax, eax と sub eax, eax の入れ替えなどが典型例であるが，あるレジスタに値を設定する際にあえて他のレジスタを介して値を渡すなど複雑な処理を行い，静的解析を妨害する方法もある．例えば，EAX レジスタを 0 に初期化する場合，xor eax, eax と記述する代わりに，他のレジスタを介して，mov eax, ebx → add eax, 0x1 → sub ebx, 0x1 → xor eax, ebx のように初期化するなど様々な難読化が考えられる．このような複雑な値の受け渡しに対して、静的検知方式での追跡は難しい[7].

3.提案方式

本章では通常のシェルコード，ポリモーフィックシェルコード，および難読化が施された非ポリモーフィックシェルコードのいずれも，エミュレーションに基づき検知可能な手法を提案する．提案方式では，検知対象の各シェルコードの中で，それぞれ難読化が施されない命令(本稿では**平文命令**とよぶ)が存在することに着目する．提案方式の流れは次の通りである．提案方式の概要を図 2 に示す．

(1)まず，検査対象の通信データを TCP ストリーム単位，または，UDP パケット単位に分割し，それぞれを静的に走査することで平文命令の有無を確認する．
(2)平文命令が検出された場合，検出された平文命令に応じた動的検査を検出位置周辺で行いシェルコードの有無を判断する．

提案方式では，1 度の静的走査によって，検知対象の全ての種類のシェルコードについて一括して検査できる点と，静的検査と動的検査を組み合わせることで全ての通信データを動的検査する手法[3,4]にくらべて効率化を図っている点が特長である．以下では，各処理を詳説する．

①静的検査

静的検査では，各 TCP ストリームまたは UDP パケットの先頭から順に 1 バイトずつ平文命令の有無を検査する．表 1 に検出対象の平文命令を示す．ポリモーフィックシェルコードでは，原理上，GetPC コード自体は暗号化されないためこれを平文命令として探索する．また，非ポリモーフィックシェルコードでは，シェルコードの一部は難読化されている可能性があるものの，システムコール自体は平文で現れるため，これを検出対象とする．

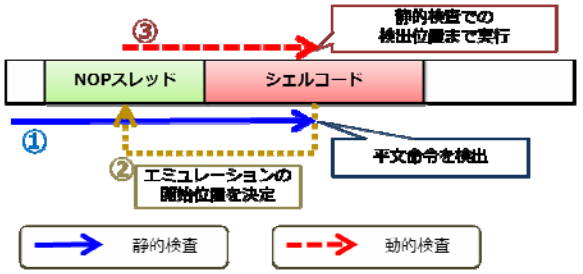


図 2 提案方式の概要図

表 1 静的検査の検出対象となる平文命令

シェルコードの種類	平文命令	対応する機械語命令
ポリモーフィック シェルコード	GetPCコード	CALL(相対ジャンプ), FSAVE, FNSAVE,FSTENV, FNSTENV
非ポリモーフィック シェルコード	システムコール	INT, CALL(間接ジャンプ), JMP, RET

②動的検査開始位置の決定

検査対象のストリームにシェルコードが含まれている場合，静的検査によって平文命令が検出された位置の周辺がシェルコード本体であることが期待される．通常，シェルコード本体は高々数十から数百バイトであるため，平文命令が検出された位置の前後数百バイトの範囲で動的検査を行う．検査範囲で 1 バイトずつ網羅的に検査を行う方法も考えられるが，経験的に定めた**オフセット集合**を用いて検査を簡略化する方法もある．4 章の実験では後者の方法を用いる．

③動的検査

動的検査では，エミュレーションがエラーを起こすか，一定のシェルコード検知規則に適合するか，または，所定値 max 回の命令が実行されるまでエミュレータによる実行を続ける．シェルコード検知規則は以下の通りである．ここで，共通規則は全てのシェルコードが適合する規則である．よって，共通規則とポリモーフィックシェルコード検知規則の両方に適合する場合のみ，ポリモーフィックシェルコードとして検知する．同様に，共通規則と非ポリモーフィックシェルコード検知規則の両方に適合する場合のみ，非ポリモーフィックシェルコードとして検知する．

- (共通規則) エミュレーションがエラーを起こすことなく，静的検査により特定された平文命令まで至ること
- (ポリモーフィックシェルコード検知規則) GetPC コードを利用した自己書き換えが実際に行われていること．具体的には，スタックに保存されたプログラムカウンタがレジスタにコピーされており，かつ，メモリの書き換え命令の対象アドレスが検査対象データの範囲内になっていること．
- (非ポリモーフィックシェルコード検知規則)静的検査によって特定されたシステムコールが INT 命令の場合，割り込み番号が適切であること．また，システムコ

ールが標準ライブラリの関数呼び出しの場合は，静的検査で検出されたシステムコールについて，呼び出されたアドレスが，標準ライブラリの配置されるアドレス範囲であること．

なお，非ポリモーフィックシェルコード検知規則における例外として，プロセスの情報に関する構造体 PEB(Process Environment Block)の参照処理が行われた場合は，共通規則における平文命令位置までにエミュレーションが至らずとも，非ポリモーフィックシェルコードとして検知を行う．これは，Windows でライブラリ呼び出しを行うシェルコードについては，呼び出しアドレスを検索するために PEB の参照を行う方法があること，およびパフォーマンスの効率化を図るためである．

最後に提案方式の擬似コード表現を図 3 に示す．図中の変数，配列，関数を以下のように定義する．

Streams:
　TCPストリーム，UDPパケットの集合
foreach(x in X){...}:
　集合Xの各要素xに対して{...}内の処理を行うことを表す
sizeof(data):
　バイト列dataのサイズ(バイト)を返す
data.isGetPC(i):
　バイト列dataのiバイト目からGetPCコードが始まる場合にtrueを返す
data.isSysCall(i):
　バイト列dataのiバイト目からシステムコールが始まる場合にtrueを返す
Offsets:
　動的検査の開始アドレスを定めるオフセットoffsetの集合
emulate(data, i, offset, suspectShellType) :
　バイト列dataを(i + offset)バイト目からエミュレータ上で実行し，
　suspectShellTypeに関する検知規則にしたがい動的検査を行う．
　詳細は3章③を参照のこと．

```
foreach (data in Streams){
  for(i=0; i<sizeof(data); i++){
    suspectShellType = NULL;
    if(data.isGetPC(i) == true)
      suspectShellType = polyShell;
    if(data.isSysCall(i) == true)
      suspectShellType = nonPolyShell;
    if(suspectShellType != NULL){
      foreach (offset in Offsets)
        emulate(data,i,offset,suspectShellType);
    }
  }
}
```

図 3 検知動作の擬似コード

4.評価実験

本章では，提案方式の評価実験の説明を行う．

4.1 実験方法

実験 1　最初に，提案方式によって，検知対象のシェルコードを実際に検知できるかを調べるために，脆弱性検証ツール Metasploit[8]を用いて生成したシェルコード計 39 体を用いて評価を行った．実験 1 に用いたシェルコードは，表 2(a)に示すペイロード計 10 体(Linux 用 5 体，Windows 用 5 体)から生成した非ポリモーフィックシェルコード 9 体[2]，および表 2(b)に示す 3 種類のエンコーダにより生成されたポリモーフィックシェルコード 30 体の合計 39 体である．

実験 2　次に，シェルコードが含まれない通常の通信データに対して誤検知がどの程度発生するかを評価した．今回は，暗号化通信を想定して擬似乱数ストリームを生成し，これに対する検知結果を調べた．擬似ストリーム 1 個あたりのサイズは 8KB とし，計 38400 個（合計サイズ 300MB）を Perl の rand 関数により生成した．

実験 3　最後に，実際の通信データに対する提案手法の検知結果を調べた．実際の通信データとしては，大学にて実運用中の Web サーバの通信データ（2010 年 7 月 25 日から連続する 18 日間分）および CCC DATAset 2010[9]の攻撃通信データ（7 日間分）を用いた．これらの通信データの基礎統計情報を表 3 に示す．なお，CCCDATAset の攻撃通信データはハニーポットの通信記録であるため，シェルコードが検知されることが期待される．一方，Web サーバの通信データについては，大学内の上位ネットワークにおいてフィルタリングを実施しており，多くは Web サイトへの正規アクセスであると思われる．また，上記のフィルタリングにより当該 Web サーバに届いたパケットの宛先ポートは全て 80/TCP であった．

表 2　実験 1 で用いた Metasploit によるシェルコード

(a) ペイロードの種類

ペイロード名		攻撃内容
Linux用	Windows用	
exec		任意のコマンドを実行する
bind_tcp		接続を待ち受け状態にする
reverse_tcp		攻撃元との接続を確立する
adduser		ユーザを追加する
chmod		ファイル・ディレクトリの権限を設定する
	dl_exec	ネットワーク経由でファイルをダウンロードして実行する

(b) 暗号化・復号ルーチンの種類

エンコーダ名	実行時の特徴
call4_dword_xor	GetPCコードにCALL命令を用いる
fnstenv_mov	GetPCコードにFSTENV命令を用いる
shikata_ga_nai	復号ルーチンの自己書き換えを行う

表 3　実験 3 で用いた通信データ

Webサーバへの通信データ		
	ストリーム・パケットサイズ合計	69 [MB]
	TCP(HTTP)ストリーム数	2288 [個]
CCC DATAset2010の攻撃通信データ		
	pcapファイルサイズ合計	約3.4 [GB]
	ストリーム・パケットサイズ合計	162 [MB]
	TCPストリーム数	15910 [個]
	UDPパケット数	272814 [個]

なお，提案方式の実装においては，すべて C 言語を

[2] ペイロード dl_exec の非ポリモーフィックシェルコードは作成できなかったため除外した．

用いた. また, エミュレーションについては, C ライブラリとして提供されている x86 系 CPU ミュレータ, libemu[13]を用いた. また, エミュレーション開始位置を決定するオフセット集合 Offsets は{±512, ±256, ±192, ±144, ±128, ±112, ±96, ±72, ±64, ±48, ±32, ±16}とした. なお, エミュレーションにおける命令実行回数の所定値 max は先行研究[3]を参考に 2^{17} とした. [3]

4.2 実験結果

実験 1 Metasploit により生成した 39 種類のシェルコードが全て正しく検知された. 検知されたシェルコードの内訳を表 4 に示す. なお, 非ポリモーフィックシェルコードの平文命令について, Linux 用ではすべて割り込み命令であったのに対して, Windows 用ではいずれもライブラリ呼び出しだった.

表 4 実験 1 で検知されたシェルコードの内訳

作成したシェルコードの総数	39 [個]
検知されたシェルコードの総数	39 [個]
ポリモーフィックシェルコード総数	30 [個]
静的検査で検知されたGetPCコードの内訳	
CALL命令	10 [個]
FPU環境保存命令	20 [個]
非ポリモーフィックシェルコード総数	9 [個]
静的検査で検知されたシステムコール内訳	
割り込み(INT)命令	5 [個]
標準ライブラリ呼び出し	4 [個]

実験 2 実験 2 で用意したいずれのランダムデータからも, シェルコードが誤検知されることはなかった. また, エミュレーション中に命令実行回数が max に到達した回数は 67465 回だった. これらは, リモートエクスプロイト攻撃とは関わりのない通信データが偶然ループを構成することにより発生していると推測される. なお, 当該ランダムデータに対する検査処理速度を測定したところ約 247 [kbps]であった.

実験 3 Web サーバの通信データでは, シェルコードは検知されなかった. 一方, 攻撃通信データからは全部で 608 個のシェルコードが検知された. この結果を表 5 に示す. また, エミュレーション中に命令実行回数が max に到達した回数について, Web サーバの通信データでは 18720 回, 攻撃通信データでは 63,653 回であった. これらは, 実験 2 と同様, リモートエクスプロイト攻撃とは関わりのない通信データが偶然ループを構成しているためであると推測されるが, 動的解析を回避するために故意に構成されたループの可能性もあるため, 今後, 詳細な解析をおこなう予定である. また, 攻撃通信データにおいて, 全てのシェルコードは, いずれも TCP ストリームから検出され, UDP パケットからは検知されなかった. 図 4 に宛先ポートごとの統計を示す. 検知されたほとんどのリモートエクスプロイト攻撃が 135, 445/TCP に集中していることが分かる. 一方, ポリモーフィックシェルコードが 135, 139, 445/TCP に対して送られているのに対して, 非ポリモーフィックシェルコードは 1027/TCP 以降のポートに送られたものが多かった. シェルコードの種類と送信先ポート番号の関連性の考察については今後の課題とする.

表 5 攻撃通信データ内で検知されたシェルコードの内訳

検知されたシェルコードの総数	608 [個]
ポリモーフィックシェルコード総数	589 [個]
静的検査で検知されたGetPCコードの内訳	
CALL命令	577 [個]
FPU環境保存命令	12 [個]
非ポリモーフィックシェルコード総数	19 [個]
静的検査で検知されたシステムコール内訳	
割り込み(INT)命令	0 [個]
標準ライブラリ呼び出し	19 [個]

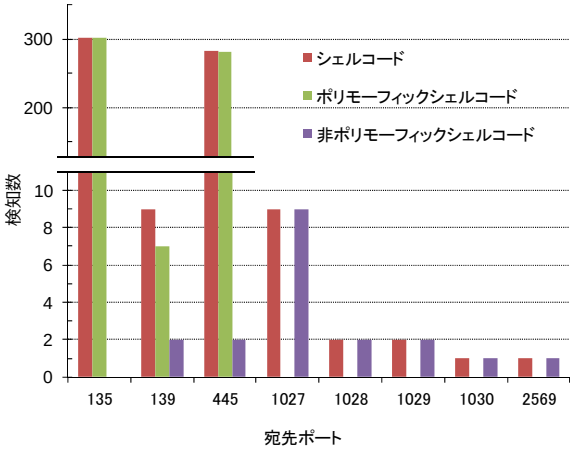


図 4 各ポートにおけるシェルコードの検知数

検知されたコードの手動解析 実験 3 で検知されたシェルコードをすべて手動で解析した. その結果, いずれも実際にリモートエクスプロイト攻撃と思われる通信データが確認でき, フォールスポジティブと思われる検知結果は確認できなかった. 特に, 検知されたシェルコードはいずれも PEB を参照する処理が行われており, Windows をターゲットとしたシェルコードであることが分かった. また, 検知されたポリモーフィックシェルコードのうち, 最も多く用いられた GetPC コードは CALL 命令を用いたものであった. 復号ルーチンについてはいずれもループを用いており, 1 バイト, または 4 バイトごとに XOR 演算による復号を行っていた. 中には, 難読化のため, 復号ルーチンに対しても自己書き換え処理が施されたシェルコードも含まれていた. 一方, 非ポリモーフィックシェルコードについては, いずれも平文命令としてライブラリの呼び出しを用いていた. 特に, 非ポリモーフィックシェルコードにおいては, (1) NOP 命令以外の 1 バイト命令を用いて難読化された NOP スレッド, および, (2)システムコール実行より前に,

[3] 実験に使用したマシンのスペックは CPU Intel(R) Xeon(R) 2.13 [GHz], 主メモリ 8[GB]である.

スタックの値を用いた入力コード内への間接ジャンプを行うシェルコードが確認された．間接ジャンプを用いた非ポリモーフィックシェルコードは，従来手法である静的な逆アセンブルや CFG の生成や従来の動的検知手法により検知することは困難であるため，提案手法による検知が有効に働くケースであるといえる．

5.関連研究

ネットワークベースのシェルコード検知方式のうち，最も古典的な方法がシグネチャとのパターンマッチングであり，代表的なツールとしては Snort[1]がある．事前に作成したシグネチャファイルを用いることで，既知の攻撃に対して検知を行うことができる．

静的解析を用いる検知手法では，逆アセンブルおよび制御フローグラフ（CFG）を用いた方式[2]や，Taint Analysis を用いた方式[3]が提案されている．文献[2]では，静的に CFG を生成し，非ポリモーフィックシェルコードに含まれるシステムコール，またはポリモーフィックシェルコードの復号ルーチンを構成するループ構造を検出して検知を行う．ただし，自己書き換え等の静的解析を妨害する手法には対応できないものがある．また，文献[7]では，自己書き換え等，多くの静的解析を妨害する手法に対しても検知を行うことができるように改善を図っているが，レジスタの初期化における難読化に対して，依然として問題が残る．

一方，動的検知手法では，文献[3,4]が既存研究として挙げられる．文献[3]では，まず静的解析を妨害する手法を用いたポリモーフィックシェルコードが存在することを示し，それを解決する方法として，GetPC コードの実行，および復号ルーチンの処理におけるメモリへの読み込みの挙動をエミュレーションによる動的検査により調べる手法を提案している．さらに，文献[4]では攻撃時に送信されるコード以外の環境情報に依存するポリモーフィックシェルコードについての検知方式を提案している．また，文献[5]では，事前に静的解析で GetPC コードを探してから，エミュレーションによって復号ルーチンを調べることにより，効率化を図っている．

6.まとめと今後の課題

本稿では，ポリモーフィックシェルコードと非ポリモーフィックシェルコードのいずれも検知可能な動的検知手法を提案した．そして，脆弱性検証ツールを用いた検知実験により，実際にポリモーフィックシェルコードと非ポリモーフィックシェルコードを検知できることを示した．また，ランダムデータを用いた検知実験により，暗号文のようなランダムデータがシェルコードとして誤検知されることがないことを示した．さらに，ハニーポットの通信データに対してシェルコードの検知実験を行い，実際に多数のシェルコードが検知できていることを確認した．

一方，提案方式のさらなる評価が必要である．まず，今回，脆弱性検証ツールにより生成した検証用シェルコードは 39 種類のみであるため，これ以外のシェルコードについても評価を行う必要がある．また，攻撃ではない正規通信に対する誤検知の評価についても，300MB 程度のランダムデータを用いた評価にとどまっているため，今後データ量と種類を増やして評価する必要がある．また，他の検知方式との比較実験も行う予定である．

最後に，検知処理速度については，すべての通信データを動的検査する従来方式と比べて改善が見込めるものの，定量的な評価や先行研究との比較は今後の課題としたい．

参考文献

[1] “Snort,” http://www.snort.org/ (Last Visit: 2010/08/20).

[2] R.Chinchani1 and E. van den Berg, “A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows,” *RAID 2005*, pp. 284 – 308, 2005.

[3] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis, ”Network-level polymorphic shellcode detection using emulation,” *DIMVA 2006*, pp. 54 – 73, 2006.

[4] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis, ”Emulation-based detection of non-self-contained polymorphic shellcode,” *RAID 2007*, pp. 87 – 106, 2007

[5] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer, “Analyzing network traffic to detect self-decrypting exploit code,” *AsiaCCS 2007*, pp. 4 – 12, 2007.

[6] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” *ACM CCS 2003*, pp. 290– 299, 2003.

[7] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, “STILL: Exploit Code Detection via Static Taint and Initialization,” *ACSAC 2008*, pp. 289 – 298, 2008.

[8] “The Metasploit Project,” http://www.metasploit.com/ (Last Visit: 2010/08/26).

[9] 畑田充弘，中津留勇，秋山満昭，三輪信，“マルウェア対策のための研究用データセット ～MWS 2010Datasets～,” *MWS2010*, 2010.

[10] “writing detection signatures,” http://www.usenix.org/publications/login/2005-12/pdfs/jordan.pdf (Last Visit: 2010/08/26).

[11] “GetPC code,” http://www.securityfocus.com/archive/82/327348/2006-01-03/1 (Last Visit: 2010/08/26).

[12] Peter Szor, “Hunting for metamorphic,” *Virus Bulletin Conference 2001*, pp. 123 – 144, 2001.

[13] “libemu,” http://libemu.carnivore.it/ (Last Visit: 2010/08/20).