情報処理学会論文誌　Vol. 50　No. 9　2371–2381 (Sep. 2009)

Yataglass：攻撃の擬似実行による攻撃メッセージの振舞いの解析

嶋村　誠[†1]　河野健二[†1,†2]

バッファオーバフロー攻撃に代表されるリモートコードインジェクション攻撃が大きな問題となっている．このような攻撃を検知するため，近年ではメッセージ中に機械語命令列に相当するバイト列が含まれているかどうかを検査するネットワーク侵入検知システム（NIDS）が提案されている．しかし，これらのシステムでは検知した攻撃コードが実際にサーバ上でどのように振る舞うかは分からない．このため，NIDS が攻撃を検知すると，管理者は適切な対策をとるため，人手で攻撃コードの振舞いを調査しなければならない．本論文では攻撃メッセージを解析し，攻撃コードの振舞いを抽出するシステムである Yataglass を提案する．Yataglass では，NIDS が検知したメッセージを機械語命令列と見なして擬似的に実行し，攻撃が成功したときに実行されるシステムコール列を抽出する．実際に Intel x86 アーキテクチャの Linux および Windows に対する攻撃メッセージを対象とした Yataglass のプロトタイプを作成し，実験を行った．実験の結果，Samba を対象とする攻撃メッセージや，Metasploit Framework から生成された攻撃メッセージが実行するシステムコールを抽出することができた．

Yataglass: Network-level Attack Behavior Analysis with Emulated Execution

Makoto Shimamura[†1] and Kenji Kono[†1,†2]

Remote code injection attacks such as buffer-overflow attacks are still one of the most serious attacks on computer systems. Current researchers focus on network intrusion detection systems (NIDSs) to detect anomal byte sequences such as machine instructions in network messages. However, such systems do not analyze behaviors of detected attacks and thus the administrator must find damage on her server when her NIDS detects an attack. In this paper, we propose a network message analyzer called Yataglass which executes attack code in an emulated environment. Yataglass treats the byte stream of a detected message as machine instructions and analyzes them to reveal behaviors of the attack code (i.e., which system calls the attack issues). Experimental results show that Yataglass successfully generated a list of system calls issued by a real attack message for Samba server and polymorphic attacks generated by the Metasploit Framework.

1. はじめに

現在，バッファオーバフロー攻撃に代表されるように，攻撃者が作成した任意のコードをサーバに実行させる攻撃が問題になっている．これらの攻撃は，攻撃者が動作させたい攻撃コードを通信メッセージに含ませて，サーバの脆弱性を利用してサーバに攻撃コードを挿入するため，リモートコードインジェクション攻撃と呼ばれている．このようなリモートコードインジェクション攻撃を検知するため，ネットワーク侵入検知システム（NIDS）[1)–3)] が広く利用されている．

しかし，攻撃の検知だけでは管理者は適切な対応を行うことはできない．攻撃への対応を行うためには，攻撃が成功したときに，サーバ上で攻撃コードがどのような振舞いを起こすかを知ることが必要になる[4)]．たとえば，攻撃コードが悪意のあるプログラムをダウンロードして起動した場合は，そのプログラムを終了させ，実行ファイルを消去しなければならない．また，攻撃コードがパスワードファイルを改ざんした場合は，管理者はパスワードファイルを適切に修復しなければならない．このような攻撃コードの振舞いに関する情報は既存の NIDS では提供されない．したがって，NIDS が検知したメッセージを人手で解析したり，ツールを用いて重要なファイルの改ざんなどの攻撃の痕跡を検出したりする手法が使われている[5),6)]．しかし，実際には攻撃メッセージは複雑に暗号化や難読化されている場合が多く[7)–11)]，人手での解析は難しい．また，パスワードファイルを読み出すだけの攻撃など，攻撃の痕跡が分かりにくい攻撃もある．

そこで，本論文では，NIDS が検知した疑わしいメッセージを解析し，リモートコードインジェクション攻撃が成功したときに，攻撃メッセージ中に含まれる攻撃コードがサーバ上でどのような振舞いを起こすかを抽出するシステムである Yataglass を提案する．Yataglass

†1 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Faculty of Science and Technology, Keio University
†2 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

© 2009 Information Processing Society of Japan

では，NIDS が検知したメッセージを機械語命令列と見なして擬似的に実行し，その攻撃が成功したときに発行されるシステムコール列とその引数を抽出する．

Yataglass によって出力された API コールリストは様々な目的に使用できる．第 1 に，API コールリストを見ることで，管理者がサーバの被害を特定するための手がかりになる．たとえば，ある攻撃が `open("/etc/passwd",O_RDWR)` を発行し，これにより得られたファイルディスクリプタに対して `write()` による書き込みを行った場合，パスワードファイルを改変する振舞いと見なして，管理者はパスワードファイルを調べればよいことが分かる．第 2 に，攻撃のより深い解析のために利用することができる．たとえば，何らかのプログラムをダウンロードする振舞いが抽出された場合，API コールリスト中のダウンロードに関わる引数を見ることで，管理者はプログラムの入手元の URL が分かるので，プログラムを入手し解析を行うことができる．第 3 に，Yataglass は NIDS の誤検知の削減に応用できる．Kruegel らの Snort Alert Verification [12] では，シグネチャ方式の NIDS について，攻撃が成功した証拠を検出することで攻撃の成否を確認する．たとえば，Slapper [13] ワームによる攻撃を検知した場合，/tmp/.uubugtraq というファイルが作られたかどうかを調べ，ファイルがあれば警告を発する．しかし，彼らのシステムでは，攻撃の成否の確認方法をあらかじめシグネチャごとに記述しなければならない．このようなシステムに Yataglass を用いることにより，攻撃の成否の確認方法の作成を容易に行えるようになる．たとえば，攻撃の呼び出す API コールを検出することで，攻撃の成否を確認するようにすればよい．

Yataglass は NIDS により攻撃が検知された後の攻撃の解析に用いるため，オフラインでの使用を想定している．管理者は Yataglass に NIDS が検知した攻撃メッセージを入力し，攻撃コードの振舞いの解析を行う．**図 1** に Yataglass の動作の概要を示す．Yataglass は，NIDS が検知した攻撃メッセージを受け取り，そのメッセージを機械語の命令列だと思って

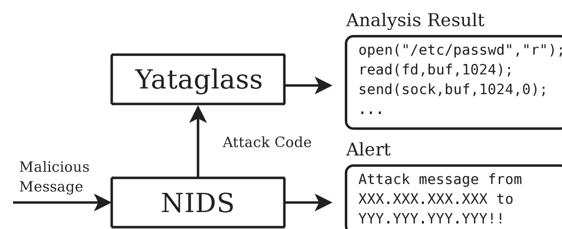

図 1　Yataglass の動作の概要
Fig. 1　Basic architecture of Yataglass.

擬似的に実行する．実行中にシステムコールやシステム API のコール（API コール）を検出した場合には，そのときのレジスタおよびスタックの内容を利用して，API の名前や引数を記録する．その後，解析結果として，記録された呼び出しのリストを出力する．なお，本論文では NIDS が検知した攻撃メッセージのうち，機械語命令列を含むものを対象とする．

本論文では，Intel x86 アーキテクチャ上で動作する Linux，Windows に対する攻撃コードを対象とした Yataglass のプロトタイプを実装し，動作確認を行った．実験の結果，Samba を対象とする実際の攻撃スクリプトから発行された攻撃メッセージや，Metasploit Framework から生成された攻撃メッセージについて，NIDS 上で攻撃が発行するシステムコールを判断することができた．また，Metasploit Framework で用いられる各種エンコーダや TAPiON エンコーダについて実験を行い，これらのエンコーダによって暗号化，難読化された攻撃メッセージが正しく復号され実行されることを確認した．388,220 命令を使用する攻撃コードの解析に要する時間は 153 ミリ秒程度であった．

本論文の構成は以下のとおりである．2 章では Yataglass の対象とする攻撃コードの特徴について説明する．3 章では Yataglass の行う攻撃コードの解析について述べ，4 章では Yataglass の動作確認を行った結果を述べる．5 章では Yataglass の制限となる点について議論する．6 章では関連研究をまとめる．そして 7 章で本論文をまとめる．

2. 攻撃コードの特徴

本論文では，NIDS において，攻撃コードの解析を行い，攻撃が成功したときに攻撃コードがサーバ上でどのような振舞いを起こすかを抽出することを目的とする．また，本論文では Intel x86 アーキテクチャ上で動作する Linux および Windows 上で動作する攻撃コードを対象とする．これは，現在では Intel x86 アーキテクチャ上の Linux と Windows が広く使われており，実際に適用できる攻撃コードが多いためである．

本論文が対象とするリモートコードインジェクション攻撃では，攻撃者は攻撃対象であるサーバプログラムの脆弱性を利用した攻撃メッセージを用いて，攻撃メッセージ中の攻撃コードをサーバプログラムに挿入する．この攻撃コードはプログラムとして実行可能な機械語命令列になっている．次に攻撃者はサーバプログラムの実行制御にかかわるデータを書き換え，攻撃コードを実行し被害を引き起こす．リモートコードインジェクション攻撃の中でも特にバッファ溢れ攻撃がよく知られている [14]．そのほかにも，フォーマット文字列攻撃 [15),16]，ヒープの二重解放を利用した攻撃 [17]，return-into-libc 攻撃 [18] など，様々なリモートコードインジェクションの方法が存在している．

© 2009 Information Processing Society of Japan

リモートコードインジェクション攻撃で用いられる攻撃コードは多くの場合 2 つの特徴を持つ．第 1 に攻撃コードは計算機資源へのアクセスを行う．第 2 に攻撃コードは攻撃コード単体で動作する．以下ではそれらの特徴について説明する．

2.1　計算機資源へのアクセスの必要性

攻撃コードは攻撃の目的を達するために，攻撃対象のサーバ上でシステムコールを実行することが多い．これは，システムコールを実行しない限り，サーバ上のファイルなどの計算機資源を用いた様々な攻撃が行えないためである．たとえば，シェルを実行するには，`execve` のようなシステムコールを用いる必要がある．一方，攻撃者がサーバ上でシステムコールを実行しない場合，行える攻撃の種類がきわめて限定されてしまい，無限ループを用いたサービス拒否攻撃程度しか行うことができない．

ただし，実際に攻撃コードが計算機資源へのアクセスを行う方法はシステムコールだけに限られない．たとえば，Windows を対象とした攻撃コードでは，計算機資源の操作を行うためにシステムコールだけでなく，Win32 API を用いることが多い．Win32 API はシステムコールを抽象化するソフトウェアレイヤであり，1 つの Win32 API は多くのシステムコールを実行する．以下ではシステムコールと Win32 API コールを総称して API コールと呼ぶ．本論文では，API コールを実行する攻撃コードに注目する．

振舞いの解析を行うときには，攻撃コードに従った API レベルで解析結果を出力する必要がある．たとえば，Win32 API を用いるような攻撃コードに対しては，Win32 API が実行するシステムコール列ではなく，Win32 API コール列として解析結果を出力することが望ましい．これは，Win32 API をシステムコール列に分解すると，非常に大量の解析結果が出力され，かえって管理者の理解を妨げてしまうためである．一方，Linux を対象とした攻撃コードはシステムコールを直接実行することが多いため，システムコール列を出力すればよい．

2.2　攻撃コード単体での動作の必要性

攻撃メッセージに埋め込まれた攻撃コードは，サーバプロセスの持つデータやプログラムにかかわらず単体で実行可能なことが多い．これは，攻撃メッセージが Metasploit Framework [11] のようなツールを用いて，脆弱性情報と攻撃コードを組み合わせて生成されるためである．たとえば，Metasploit では，Linux を用いたサーバを攻撃するためによく使われる攻撃コードとして “Linux Execute Command” という任意のコマンドを実行する攻撃コードがある．これに，“samba_trans2open” [19] という Samba に存在する脆弱性の情報を組み合わせると，Linux 上の Samba において任意のコマンドを実行する攻撃コードが作成できる．このような手法で攻撃メッセージを生成することにより，攻撃者は新しい脆弱性を発見するとすぐに攻撃メッセージを生成することが可能になる．

したがって，本論文ではメッセージ中の攻撃コードのみを解析し，その振舞いを抽出する．しかし，このようにすると，攻撃者がサーバ上のメモリ内容を利用する攻撃コードを用いた場合には，そのメモリ内容が攻撃メッセージ中に存在しないため，攻撃の振舞いを解析できない．しかし，現在の攻撃コードの多くは単体で実行可能であるため，メッセージ中の攻撃コードのみで解析可能である．これについては，5 章で詳しく議論する．

また，ここでいう攻撃コードとは，サーバの脆弱性を利用し攻撃メッセージによってサーバに挿入され実行される命令列のことである．近年の攻撃では，ボットなどのように，攻撃コードを実行して必要なプログラムをダウンロードし，実行することが多い．しかし，このような場合でも攻撃コードを解析することで，次の攻撃を構成するプログラムを入手し解析できる [4]．Yataglass では，必要なプログラムをダウンロードし起動するまでの振舞いを解析することを目的としており，ダウンロードされたプログラムがどのような振舞いをするかということは対象としていない．このようなプログラムの振舞いの解析には，既存のマルウェア解析技術 [20),21] を用いることができる．

3.　Yataglass

3.1　Yataglass による攻撃コードの解析

Yataglass では，NIDS に検知された攻撃メッセージ中の攻撃コードに従って，Yataglass の持つ仮想的なレジスタとメモリを操作することで攻撃コードを解析する．**表 1** に，Intel x86 アーキテクチャに用意されていて Yataglass が持つレジスタを示す．Yataglass はこれらのレジスタに対応するメモリ領域を用意して，仮想的に使用する．また，Yataglass はメ

表 1　Intel x86 アーキテクチャで使われる代表的なレジスタ
Table 1　General registers in the Intel x86 architecture.

レジスタ名	レジスタの役割
eax，ebx，ecx，edx	汎用レジスタ
eip	命令カウンタを格納するレジスタ
esi，edi	ストリング命令用のレジスタ
esp	スタックポインタを格納するレジスタ
ebp	ベースポインタを格納するレジスタ
eflags	条件分岐のための各種フラグを持つレジスタ
cs，ds,，es，fs，gs，ss	セグメントレジスタ

© 2009 Information Processing Society of Japan

モリ領域としてコード領域とスタック領域を管理する．ここで，コード領域は攻撃メッセージを配置する領域であり，ここに配置されたメッセージが機械語命令列と見なされ実行される．また，スタック領域は実行した命令列が esp レジスタなどを用いてスタック操作を行う際に用いる領域である．攻撃コードとして実行される命令列は，基本的にはこれらの領域しかアクセスすることはない．これは，2.2 節でも述べたとおり，攻撃コードがサーバプロセスの持つデータやプログラムにかかわらず単体で実行可能であるためである．

ただし，攻撃コードがコード領域やスタック領域の範囲外に書き込みを行う場合がある．これは，攻撃コード自体が新たに命令列を生成しその命令列を実行する場合があるからである．このような場合も攻撃コードの解析ができるように，攻撃コードが範囲外のメモリ領域に書き込みを行った場合，それ以降はその領域への読み出しおよび実行ができるようにする．なお，攻撃コード自体が新たな命令列を生成するのは珍しいことではなく，たとえば，暗号化された攻撃コードを復号し，平文の攻撃コードが生成されることは多い．

3.1.1　攻撃コードの開始アドレスの決定

x86 アーキテクチャは可変長の命令を使用しているため，どのバイト位置からでも命令列の実行を開始できる．したがって，Yataglass は NIDS が検知したメッセージの初めのバイト位置から実行を開始し，実行が終了すると，次のバイト位置からの実行を開始する．これを，メッセージの最後のバイト位置からの実行が行われるまで繰り返し，得られたすべての API コールを出力する．正しくないバイト位置から実行を行った場合，出力される API コール列が意味のある API コール列になることは非常に少ないため，管理者は API コール列を見ることで正しいバイト位置からの実行結果が分かる．なお，SigFree [3] などのような，NIDS がどのバイト位置から攻撃コードが開始されるかを判定できる NIDS を用いている場合は，その情報を用いることにより解析の所要時間を削減できる．

3.1.2　API コールの検出

攻撃コードが API コールを発行する方法は攻撃対象の OS によって決まっている．具体的には，システムコールの発行は Linux 上では int 0x80 命令，もしくは sysenter 命令を用いる．また，Windows 上では int 0x2E 命令と sysenter 命令を用いる．また，Win32 API の発行は API のエントリアドレスに対する制御移行命令（jmp 命令，jcc 命令，call 命令，ret 命令）を用いる．したがって，攻撃コードの振舞いを解析するには，これらの命令が実行されたときのスタックやレジスタの内容を用いて，どのような API コールが発行されたのかを判断すればよい．なお，ret 命令も含めているのは，攻撃コードがスタック上の戻りアドレスを書き換えて ret 命令を用いることで API コールを発行することがあるためである．

Yataglass は API コールの実行を検知した場合，API コールの本体であるコードの実行は行わず，Yataglass 中に存在するスタブを実行する．スタブは Yataglass が検出した API コールの種類に応じて，Yataglass が都合のよいエミュレーションを行うために呼び出す関数である．これにより Yataglass 自体への攻撃や，Yataglass を踏み台とした他のホストへの攻撃を防ぐことができる．たとえば，`fork` のような子プロセスを生成する API コールは，呼び出されると，実際に Yataglass を fork して，攻撃コードの実行を親プロセスと子プロセスに分けて解析を続行する必要がある．これは攻撃コードにおいて親プロセスと子プロセスで振舞いが異なる可能性が高いためである．しかし，このようにすると無限に子プロセスを生成することによる Yataglass への攻撃が可能になる．これを防ぐため，Yataglass ではスタブを用いて，プロセスの数に上限を定めるようにする．

また，`send` のような外部と通信を行う API コールは，攻撃コードによって他のホストに攻撃を行うために用いられる可能性がある．したがって，Yataglass はこのような API コールに対して，スタブを用いて，実際には通信を行わずに，通信に成功したことを意味する値を返す．

スタブが用意されていない API コールでは，実行を行う代わりに，返り値として eax レジスタに API コールの成功を意味する値を設定する．具体的には，Linux のシステムコールの場合は 0，Win32 API コール，および Windows のシステムコールの場合は 1 を設定する．また，API コールの返り値が 0 や 1 以外のときを成功と見なすような場合は，その API の仕様に応じた成功を意味する値を返すスタブを用意する必要がある．

3.1.3　Win32 API を用いる攻撃への対応

Win32 API を用いる攻撃コードは単純にメッセージ中の命令列を解析しスタックやレジスタの内容を調べるだけでは正しく解析できない．これは，このような攻撃コードが攻撃メッセージの外のプロセスメモリ上に存在する Process Environment Block（PEB）と，動的リンクライブラリ（DLL）中の API を利用するためである．PEB は，プロセスの実行状態に関する情報を保存している構造体であり，プロセスがロードした DLL に関する情報が保存されている．攻撃コードは PEB 上に存在するロード済みの DLL の情報を用いて，DLL に存在する API を用いることができる．たとえば，`ws2_32.dll` はネットワークソケットを扱う DLL であり，サーバプロセスにロードされていることが多い．このため，攻撃コードは `ws2_32.dll` 内の API を用いて，新たにネットワーク接続を作成することができる．さらに Windows 上のプロセスはつねに `kernel32.dll` をリンクしているため，攻撃コードは `kernel32.dll` で定義されている `LoadLibrary` と `GetProcAddress` を用いることが

© 2009 Information Processing Society of Japan

できる．攻撃コードは LoadLibrary によってサーバプロセスに新しい DLL をロードさせ，GetProcAddress によって DLL 中の API のアドレスを取得することで，サーバにインストールされた DLL の API を利用する．したがって，このような攻撃コードの解析を行うには，PEB の読み出しを行えるようにしたり，LoadLibrary などの一部の API の呼び出しについて擬似的に API を実行したりする必要がある．

したがって，Yataglass では，攻撃コードが利用できるようダミーの PEB を作成し，また，メモリ中に kernel32.dll，user32.dll，ws2_32.dll などのよく利用される DLL を展開する．こうすることにより，攻撃コードは Yataglass 上に作成された PEB を参照し，これらの展開された DLL のデータを用いるようになる．また，これらの DLL 内の API のアドレスが確定し，これらの API の呼び出しを検出できる．もし，LoadLibrary や GetProcAddress の呼び出しがあった際には，API を擬似的に実行し，DLL を展開したり，確定した API のアドレスを返したりすることで，解析を続行する．

Yataglass において，LoadLibrary や GetProcAddress 以外のあまり使われない Win32 API の擬似実行には汎用のスタブを用いる．これは，攻撃コードが LoadLibrary を用いて，任意の DLL の API を呼び出すことができるため，攻撃コードの利用するすべての Win32 API についてスタブを用意することは難しいためである．しかし，Win32 API の呼び出し規約では引数をスタックに乗せるため，API が呼び出されたときに何もしないとスタックの内容がずれてしまうため，以降の解析が続行できなくなる．Yataglass では専用のスタブが用意されていない API が呼び出されたときには，DLL の中の実行コードを API のアドレスから逆アセンブルし，初めに見つけた retn 命令の引数を用いてスタックから引数を除去することで，攻撃コードを正しく動作させる．

3.1.4　擬似実行の終了条件

Yataglass は，あるバイト位置から開始した攻撃コードの実行を続行できなくなったときに実行を終了し，次のバイト位置からの実行へ移る．攻撃コードの実行が続行できなくなる条件を以下に示す．

- **他のプログラムへの制御の移行**　他のプログラムに制御を移行するような API コールが実行された場合，それ以上の振舞いを解析できないため，実行を終了する．このような API コールには，たとえば，execve，exit，WinExec，ExitProcess などがある．
- **命令でないバイト列の実行の検知**　命令列として成立しないバイト列を実行しようとした場合，実行を終了する．実行を開始したバイト位置が攻撃コードの開始位置でない場合は，実行時に命令でないバイト列が見つかることが多い．なお，命令でないバイト列で終わるような攻撃コードである場合は，それまでに見つかった API コール列を出力する．
- **アクセス違反，および管理外のコードへの制御の移行の検知**　実行を開始したバイト位置が攻撃コードの開始位置でない場合は，攻撃コードが間違って解釈され，Yataglass の管理外のメモリ領域へアクセスすることがある．すでに 2.2 節で述べたとおり，攻撃コードは単体で実行されることが多く，Yataglass の管理外のアドレスを読み出すことはまれである．したがって，そのようなアドレスが利用された場合は実行を終了する．また，もし攻撃コードが管理外のアドレスに書き込みを行った場合は，3.1 節で述べたとおり，書き込まれたアドレスを管理下に置き，以降読み出せるようにしている．
- **無限ループの検知**　攻撃コードの実行が無限ループに入り込んでしまうことがある．本論文の実装では，無限ループを Tubella らの無限ループ検出手法[22] を用いて，ループ中にループを脱出するためのジャンプが存在するかどうか，およびループからの脱出に使うジャンプで使用する EFLAGS レジスタのビットがループ中で変化する可能性があるかどうかを利用して検出する．Yataglass がこのような無限ループを検知した場合には，実行を終了する．しかし，上記の無限ループ検出手法では検出できない無限ループが存在する可能性があるため，実行した命令数が閾値を超えた場合は無限ループに入り込んだと考え，実行を終了する．本論文の実装では閾値を 50 万命令に設定した．

3.1.5　ダミーデータの利用

攻撃コードはその実行中に，サーバ上で実行しない限り正しい値を得ることができないような API を呼び出したり，サーバ上のデータを用いたりすることがある．このようなデータには，たとえば，PEB の中のデータ，システムコールが返すデータ，rdtsc 命令で取り出す CPU のクロックカウンタの値などがある．したがって，サーバ上の情報が必要な場合にはダミーデータを用いる．ダミーデータは Yataglass 中にダミーデータであるというフラグを持つ適当なデータとして用意する．

しかし，攻撃者は，Yataglass のような擬似実行を行う解析機構で解析されているかどうかを判断するために，このようなダミーデータを検出する攻撃コードを作成する可能性がある．たとえば，攻撃者は read システムコールで得られた値が予測した値と異なる場合，それ以上の続行を中断するような攻撃コードが作成できる．この場合，解析機構を判断するコード以降の API コールは解析できなくなるため，解析された API コール列が実際のサーバ上で発行される API コールと異なってしまう．したがって，Yataglass はダミーデータを利用するすべての条件分岐について，通常の分岐結果による実行と，条件を反転させた実行を行

© 2009 Information Processing Society of Japan

う．具体的には，ダミーデータの伝搬を Newsome らの Dynamic Taint Analysis [23] と同様の手法を用いて追跡し，もしダミーデータに由来する条件分岐が発生した場合，Yataglass の実行状態を保存する．その後，正当な分岐による実行が失敗したら，保存した実行状態に戻り，条件を反転させて実行する．これにより，ダミーデータの検出を回避し実行を続けることができる．このような条件分岐の反転は実際にマルウェアの解析に利用されている [21]．

3.2　実　　装

Yataglass のプロトタイプを Intel x86 の Linux 上に実装した．x86 命令のデコードには，libdasm [24] を用いた．Yataglass は IA-32 命令セットの算術命令，ストリング命令，制御命令などを実行する．fstenv，fnstenv，fsave，fnsave を除く FPU，SIMD，特権命令は実行せず nop として扱う．これは，現在の攻撃コードがそのような命令を有効に活用せず，NOP の代わりとして使っていることが多いためである．

4．実　　験

4.1　実際の攻撃メッセージによる実験

Yataglass が実際の攻撃メッセージから API コールリストを作成できるかどうかを調べるために実験を行った．実験は，CPU が Intel Core2 6300 1.86 GHz，メモリが 2 GB のマシン上で行った．

4.1.1　Samba に対する攻撃メッセージ

Linux 上の Samba 2.2.8 のバッファオーバフロー脆弱性 [19] を利用する攻撃メッセージを SecurityFocus [25] から取得し，Yataglass 上で実行した．実行結果の一部を**図 2** に示す．図の下線部は得られたシステムコールを示す．実行結果では，この攻撃コードがソケットを作成し，標準入出力を作成したソケットに接続したうえで `/bin/sh` を実行することが確かめられた．また，実際に Redhat 7 上で Samba 2.2.8 を動作させた仮想マシンを用意して攻撃を実行し，この解析結果が正しいことを確認した．

4.1.2　Linux に対する攻撃メッセージ

Linux に対する攻撃メッセージを解析できるかどうかを調べるために，Metasploit Framework に用意されている 16 種類の Linux 用の攻撃コードについて Yataglass による解析を行った．これらの攻撃コードの名称，および解析結果として得られたシステムコール列を**表 2** に示す．解析を行った攻撃コードのうち，“Linux Command Shell, Find Tag Inline” は `recv` システムコールで特定のバイト列を受け取った後に動作を続行することが分かった．また，“Bind TCP Stager”，“Find Tag Stager”，“Reverse TCP Stager” のつく攻撃コードは 2 つのメッセージに分かれており，初めのメッセージ中の攻撃コードがネットワークソケットを準備し，次のメッセージ中の攻撃コードがそのソケットを利用するという動作になっていることが分かった．これらは，外部からのデータを `recv` システムコールで受信するような動作をしているが，これは 3.1.5 項で述べたダミーデータの追跡を用いることにより，`recv` システムコールに依存する分岐の両方を実行し，正しく解析できた．

```
No.    EIP    Inst.      Mnemonic
--------------------------------------------------
000663 000708 31c0       xor eax,eax
000664 00070a 31db       xor ebx,ebx
000665 00070c 31c9       xor ecx,ecx
000666 00070e 51         push ecx
000667 00070f b106       mov cl,0x6
000668 000711 51         push ecx
000669 000712 b101       mov cl,0x1
00066a 000714 51         push ecx
00066b 000715 b102       mov cl,0x2
00066c 000717 51         push ecx
00066d 000718 89e1       mov ecx,esp
00066e 00071a b301       mov bl,0x1
00066f 00071c b066       mov al,0x66
000670 00071e cd80       int 0x80
  - socket detected. domain=2, type=1, protocol=6
  -- return dummy1
000671 000720 89c1       mov ecx,eax
000672 000722 31c0       xor eax,eax
000673 000724 31db       xor ebx,ebx
000674 000726 50         push eax
000675 000727 50         push eax
000676 000728 50         push eax
000677 000729 6668b0ef   push word 0xefb0
000678 00072d b302       mov bl,0x2
000679 00072f 6653       push bx
00067a 000731 89e2       mov edx,esp
00067b 000733 b310       mov bl,0x10
00067c 000735 53         push ebx
00067d 000736 b302       mov bl,0x2
00067e 000738 52         push edx
00067f 000739 51         push ecx
000680 00073a 89ca       mov edx,ecx
000681 00073c 89e1       mov ecx,esp
000682 00073e b066       mov al,0x66
000683 000740 cd80       int 0x80
  - bind detected. fd=dummy1,
    addr={family=2, port=61360, addr=0.0.0.0}, len=16
000684 000742 31db       xor ebx,ebx
000685 000744 39c3       cmp ebx,eax
000686 000746 7405       jz 0x74d
000687 00074d 31c0       xor eax,eax
000688 00074f 50         push eax
000689 000750 52         push edx
00068a 000751 89e1       mov ecx,esp
00068b 000753 b304       mov bl,0x4
00068c 000755 b066       mov al,0x66
00068d 000757 cd80       int 0x80
  - listen detected. fd=dummy1, backlog=0
00068e 000759 89d7       mov edi,edx
00068f 00075b 31c0       xor eax,eax
000690 00075d 31db       xor ebx,ebx
000691 00075f 31c9       xor ecx,ecx
000692 000761 b311       mov bl,0x11
000693 000763 b101       mov cl,0x1
000694 000765 b030       mov al,0x30
000695 000767 cd80       int 0x80
  - signal detected. signum=17 sighandler=1
000696 000769 31c0       xor eax,eax
000697 00076b 31db       xor ebx,ebx
000698 00076d 50         push eax
000699 00076e 50         push eax
00069a 00076f 57         push edi
00069b 000770 89e1       mov ecx,esp
00069c 000772 b305       mov bl,0x5
00069d 000774 b066       mov al,0x66
00069e 000776 cd80       int 0x80
  - accept detected. fd=dummy1, addr=0, len=0
  -- return dummy2
00069f 000778 89c6       mov esi,eax
0006a0 00077a 31c0       xor eax,eax
0006a1 00077c 31db       xor ebx,ebx
0006a2 00077e b002       mov al,0x2
0006a3 000780 cd80       int 0x80
  - fork detected. -- emulator forks
0006a4 000782 39c3       cmp ebx,eax
0006a5 000784 7540       jnz 0x7c6
0006a6 000786 31c0       xor eax,eax
0006a7 000788 89fb       mov ebx,edi
0006a8 00078a b006       mov al,0x6
0006a9 00078c cd80       int 0x80
  - close detected. fd = dummy1
0006aa 00078e 31c0       xor eax,eax
0006ab 000790 31c9       xor ecx,ecx
0006ac 000792 89f3       mov ebx,esi
0006ad 000794 b03f       mov al,0x3f
0006ae 000796 cd80       int 0x80
  - dup2 detected. oldfd=dummy2, newfd=0
0006af 000798 31c0       xor eax,eax
0006b0 00079a 41         inc ecx
0006b1 00079b b03f       mov al,0x3f
0006b2 00079d cd80       int 0x80
  - dup2 detected. oldfd=dummy2, newfd=1
0006b3 00079f 31c0       xor eax,eax
0006b4 0007a1 41         inc ecx
0006b5 0007a2 b03f       mov al,0x3f
0006b6 0007a4 cd80       int 0x80
  - dup2 detected. oldfd=dummy2, newfd=2
0006b7 0007a6 31c0       xor eax,eax
0006b8 0007a8 50         push eax
0006b9 0007a9 682f2f7368 push dword 0x68732f2f
0006ba 0007ae 682f62696e push dword 0x6e69622f
0006bb 0007b3 89e3       mov ebx,esp
0006bc 0007b5 8b542408   mov edx,[esp+0x8]
0006bd 0007b9 50         push eax
0006be 0007ba 53         push ebx
0006bf 0007bb 89e1       mov ecx,esp
0006c0 0007bd b00b       mov al,0xb
0006c1 0007bf cd80       int 0x80
  - execve detected. path=/bin//sh  argv[0]=/bin//sh
```

図 2　Samba に対する攻撃メッセージの解析結果
Fig. 2　Analysis of a real attack message for Samba server.

© 2009 Information Processing Society of Japan

表 2　Metasploit が生成する Linux 用の攻撃コードの解析結果
Table 2　Analyses of Linux shellcodes generated by Metasploit.

攻撃コード名	実行するシステムコール列
Linux Add User	setreuid, open, write, exit
Linux Add User, Bind TCP Stager	socket, bind, listen, accept, read, setreuid, open, write, exit
Linux Add User, Find Tag Stager	recv, setreuid, open, write, exit
Linux Add User, Reverse TCP Stager	socket, connect, setreuid, open, write, exit
Linux Command Shell, Bind TCP Inline	socket, bind, listen, accept, dup2, execve
Linux Command Shell, Bind TCP Stager	socket, bind, listen, accept, read, dup2, execve
Linux Command Shell, Find Port Inline	getpeername, dup2, execve
Linux Command Shell, Find Tag Inline	recv, dup2, execve
Linux Command Shell, Find Tag Stager	recv, dup2, execve
Linux Command Shell, Reverse TCP Inline	socket, dup2, connect, execve
Linux Command Shell, Reverse TCP Inline - Metasm demo	socket, dup2, connect, execve
Linux Command Shell, Reverse TCP Stager	socket, connect, dup2, execve
Linux Execute Command	execve
Linux Execute Command, Bind TCP Stager	socket, bind, listen, accept, read, execve
Linux Execute Command, Find Tag Stager	recv, execve
Linux Execute Command, Reverse TCP Stager	socket, connect, read, execve

4.1.3　Windows に対する攻撃メッセージ

Windows に対する攻撃メッセージが解析できるかどうかを調べるため，Metasploit Framework [11] の Web サイトに用意されている攻撃コードである，“Win32 Bind Shell” および “Win32 Add User” の解析を行った．これらを用いたのは，アセンブリ言語のソースコードが Web サイトに存在しており，容易に解析結果が正しいかどうかを確認できるためである．

“Win32 Bind Shell” を解析した結果得られた API コール列を**図 3** に示す．この結果より，“Win32 Bind Shell” が `ws2_32.dll` を用いて新しくネットワーク接続を作成することと `CreateProcess` を用いてプロセスを生成することが分かった．また，“Win32 Add User” を解析した結果得られた API コール列を**図 4** に示す．この結果より，“Win32 Add User” は `netapi32.dll` に存在する `NetUserAdd`，`NetLocalGroupAddMembers` を用いて，新しいユーザをシステムに追加することが分かった．さらに，この結果を Metasploit Framework [11] の Web サイトに用意されている各攻撃コードのソースコードと比較し，結果が正しいことを確認した．なお，これらの攻撃コードは，Windows 上のプロセスの PEB を利用して，`kernel32.dll` のベースアドレスを取得し，`LoadLibrary` を用いて必要とする DLL をロードして，DLL に存在する API を用いるような攻撃コードであった．Yataglass は 3.1.3 項で述べたダミーの PEB と，`LoadLibrary` のスタブによる擬似実行によりこれらの攻撃コードの解析を行うことができた．

```
LoadLibraryA @ kernel32.dll (offset=00029491)
WSAStartup @ ws2_32.dll (offset=0000a639)
WSASocketA @ ws2_32.dll (offset=00008fa9)
bind @ ws2_32.dll (offset=0000652f)
accept @ ws2_32.dll (offset=0001bdf6)
CreateProcessA @ kernel32.dll (offset=00001c36)
WaitForSingleObject @ kernel32.dll (offset=0004c1a0)
closesocket @ ws2_32.dll (offset=0000330c)
ExitProcess @ kernel32.dll (offset=00023b54)
```

図 3　“Win32 Bind Shell” の実行する API コール列
Fig. 3　API calls issued by “Win32 Bind Shell”.

```
LoadLibraryA @ kernel32.dll (offset=00029491)
NetUserAdd @ netapi32.dll (offset=00034604)
NetLocalGroupAddMembers @ netapi32.dll (offset=000345bc)
ExitProcess @ kernel32.dll (offset=00023b54)
```

図 4　“Win32 Add User” の実行する API コール列
Fig. 4　API calls issued by “Win32 Add User”.

© 2009 Information Processing Society of Japan

4.1.4　暗号化された攻撃メッセージ

暗号化されたメッセージについて解析できるかどうかを調べるために，Metasploit Framework[11] に用意されている，Windows 上でコマンドを実行する攻撃コードである “Windows Execute Command” を，Metasploit で使用される 14 種類の暗号化手法（JmpCallAddtive, PexFnsenvMov，PexAlphaNum，Alpha2，ShiKaTaGaNai，Countdown，Alpha-upper，Alpha-mixed，Avoid-utf8-tolower，Call4_dword_xor，Nonalpha，Nonupper，Unicode-mixed，Unicode-upper），および TAPiON エンコーダ[10] で暗号化した攻撃コードについて，それぞれ Yataglass による解析を行った．実験したすべての暗号化手法において，Yataglass 上で復号し，元の “Windows Execute Command” が実行する API コールリストを取得することができた．

4.2　解析時間

表 3 に，Samba に対するバッファオーバフロー攻撃，および Metasploit によって暗号化した攻撃について，擬似実行を行った命令数と処理時間を示す．紙面の都合上，“Windows Execute Command” を暗号化したメッセージについては，高度なエンコーダである ShiKaTaGaNai エンコーダ，TAPiON エンコーダについての結果のみ掲載する．Samba に対する攻撃では int 0x80 命令で直接 Linux のシステムコールを呼び出すのに対して，Windows に対する攻撃では PEB から DLL のベースアドレスを検索し，DLL に存在する API を用いていた．このため，命令数が大きくなっており，実行に時間がかかっている．

4.3　類似方式との比較

Borders らの Spector[4] では，Symbolic Execution[26] を攻撃コードの解析に利用し，攻撃コードが発行する Win32 API の呼び出しを抽出する．Spector は Yataglass と同様に，攻撃コードの解析を目的とした技術であり，その手法も Yataglass に近い．このため，本節では Yataglass と Spector の違いを述べる．

第 1 に，Spector ではユーザがメッセージ中の攻撃コードの開始位置をあらかじめ特定しておく必要がある．このため，ユーザが開始位置を間違えて与えると，正しい解析結果を得ることはできない．一方で，Yataglass は攻撃コードがすべてのバイト位置から始まる可能性を考慮し，メッセージ中の全バイト位置からの疑似実行を行い解析する．第 2 に，Spector はシステムコールの返り値や外部からのデータを用いた分岐が行われないと仮定している．このため，攻撃者は，Spector をシステムコールの返り値を検査することで回避することができる．たとえば，`read` システムコールで得られた値が予測された値と異なる場合，攻撃者はそれ以上の続行を中断するような攻撃コードを作成できる．一方，Yataglass は 3.1.5 項で述べたダミーデータの追跡を行うことにより，このような回避コードを無視して解析を行える．

表 3　実行命令数と所要時間
Table 3　# of instructions and processing times.

攻撃コード	エンコーダ	命令数	所要時間
Samba B/O	-	1,919	12 ms
“Win32 Bind Shell”	エンコーダなし	388,220	153 ms
“Windows Execute Command”	エンコーダなし	24,136	20 ms
	ShiKaTaGaNai	24,257	19 ms
	TAPiON	26,257	20 ms

最後に，Spector は Yataglass と比較して実行速度が遅い．これは，Spector の Symbolic Execution では，XOR などのビット演算について，各ビットに対してシンボルが用いられるためである．Spector では 2 GHz の CPU を用いて，3,074 命令の攻撃コードを実行するのに 3.65 秒を要した．つまり，毎秒 842 命令を実行できる．一方，Yataglass では，1.86 GHz の CPU において，Spector と同様に実行した全命令列をコンソールに出力するようにした場合，388,220 命令の実行に 14.5 秒を要した．これは，毎秒 26,773 命令の実行に相当する．

5.　議　　論

4 章の実験で示したように Yataglass は多くの攻撃コードを正しく解析することができる．しかし，いくつかの特殊な攻撃コードを作成することによって Yataglass による解析を回避することが可能である．第 1 に，Yataglass は攻撃者がサーバ上のメモリ情報を利用する攻撃コードを用いた場合には攻撃の振舞いを解析できない．このような攻撃コードの例としては，サーバ上で ret 命令が存在することが分かっているアドレスにジャンプし，制御が戻ってきた後に攻撃コードの実行を続けるコードが考えられる．しかし，このようなサーバのメモリ内容に依存する攻撃は，攻撃の適用可能なサーバの数が少なくなる[9]．また，そのような攻撃に対応するには，解析時にサーバプロセスのメモリ内容を参照するようにすればよい．

第 2 に，Yataglass はシステムコールの抽出を目的としているため，システムコールを用いない攻撃コードを解析することはできない．このような攻撃コードの例としては，無限ループを用いることによるサービス拒否攻撃がある．しかし，Yataglass は攻撃コードの擬似実行を行うため，攻撃コードの開始位置さえ特定できれば，そのようなサービス拒否攻撃を検出することはできる．また，攻撃者はサーバプログラムの特定の位置に存在する変数を

© 2009 Information Processing Society of Japan

書き換えることにより攻撃を行うことが考えられる．しかし，そのような攻撃コードはサーバプログラムに依存するため，Metasploit[11] のようなツールでそのような攻撃を自動生成することはできない．

第 3 に，Yataglass は外部と通信する能力を持たないため，攻撃コードの実行時にデータを受信する攻撃コードを解析することはできない．このような攻撃コードの例としては，攻撃コードの一部が暗号化されており，復号のための鍵を攻撃者から受信することで復号が行われ，攻撃を続行するような攻撃コードが考えられる．このような場合，外部から受信するデータを与えなければ攻撃が成立しない．この場合は NIDS に保存されている攻撃者からの後続のパケットを与えることで攻撃コードの実行が可能であろうと考えられる．

6. 関 連 研 究

Polychronakis ら[9),27)] は，ネットワークメッセージに埋め込まれた機械語命令列を擬似的に実行し，復号化されるかどうかを調べることで，暗号化された攻撃コードを検知する手法を提案した．これにより，復号機能を持つ攻撃メッセージを検知することができる．一方で，実行時に復号動作を行わない攻撃メッセージは検知できない．Yataglass ではこの方式と同様に機械語命令列を擬似的に実行するが，これとは異なり，攻撃コードの実行時の振舞いを解析する．

ネットワークメッセージを静的に解析し，攻撃メッセージを検出する手法が数多く提案されている．SigFree[3] は，メッセージ中に含まれる意味のある命令列の長さを調べることで攻撃コードを検知する．このシステムは，メッセージの任意のバイト位置から逆アセンブルを行い，得られた実行可能なバイト列の長さが閾値を超えた場合に，メッセージを攻撃コードであると見なす．Kruegel ら[8] は，メッセージを静的に解析して，実行可能命令列の検出を行う手法を提案した．彼らの手法では，ネットワークメッセージを命令列だと仮定して逆アセンブルを行い，その結果から制御フローグラフを作成し，命令列であるかどうかを判定する．これらのシステムはエンコードされた攻撃メッセージの解析をうまく行うことはできない．たとえば，SigFree は暗号化されたメッセージを解析することはできない．Kruegel らの手法では，制御フローグラフを壊すような難読化が行われたメッセージを解析することはできない．一方，Yataglass は動的な解析を行うため，暗号化や難読化の影響を受けない．

また，STILL[28] では，ネットワークメッセージを命令列だと仮定した逆アセンブル結果を用いて自己改変コードや間接ジャンプ，システムコールの呼び出しを静的に検出することで攻撃コードの検出を行う．STILL では，自己改変コードを即座に攻撃コードと見なすため，暗号化や難読化の影響を受けることはない．しかし，STILL はシステムコールの呼び出しを静的に検出するので，若干の誤検知を発生する．Yataglass は攻撃コードの振舞いを抽出することが目的であるため，このようなシステムと共存することができる．たとえば，このような NIDS が検知した暗号化された攻撃コードについて，攻撃コードの命令列を擬似的に実行することで解析し，実際に有効なシステムコールが呼び出されるかどうかを調べることで，誤検知を削減することができる．

ホスト侵入検知システム（HIDS）では，攻撃の結果として改変されたファイル，作成されたネットワーク接続などを検出したり[5),6)]，異常なシステムコールが発行されたことを検知したりできる[29),30)]．しかし，攻撃の結果を検知するタイプの HIDS は攻撃を検知するために，あらかじめ攻撃ごとに攻撃の振舞いの内容を記述しておかなければならないため，攻撃者は記述されていないような攻撃を行うことで，検知を回避することができる．一方，異常なシステムコールの検知を行うタイプの HIDS に対しては，システムコールの発行の内容や順序を工夫することで検知を回避し攻撃を行えることが知られている[31]．また，これらの HIDS は攻撃の検知を目的とした研究であり，攻撃コードがどのような振舞いをするかを抽出することについて着目した研究ではない．

7. 終 わ り に

本論文では，NIDS が検知したメッセージを機械語命令列であると見なして実行し攻撃の振舞いを解析するシステムである Yataglass を提案した．抽出された攻撃コードの振舞いは，攻撃による被害の特定の補助や，攻撃の成否の確認などに応用できる．

本論文では Yataglass のプロトタイプを作成し，実験を行った．実験の結果，実際の Samba サーバに対する脆弱性を利用した攻撃スクリプトから抽出した攻撃メッセージや，Metasploit Framework で作成した攻撃について，暗号化された攻撃メッセージを復号化し，実行を行い，攻撃コードの振舞いを抽出できた．今後の課題としては，実際の攻撃のログを用いた実験を行うことや，サーバ上のメモリ情報を用いるように拡張することが必要である．

参 考 文 献

1) Roesch, M.: Snort: Lightweight Intrusion Detection for Networks, *Proc. 13th USENIX Conference on Systems Administration* (*LISA '99*), pp.229–238 (1999).
2) Paxson, V.: Bro: A system for detecting network intruders in real-time, *Computer Networks*, Vol.31, No.23–24, pp.2435–2463 (1999).
3) Wang, X., Pan, C.-C., Liu, P. and Zhu, S.: SigFree: A Signature-free Buffer Over-

© 2009 Information Processing Society of Japan

flow Attack Blocker, *Proc. 15th Usenix Security Symposium*, pp.225–240 (2006).
4) Borders, K., Prakash, A. and Zielinski, M.: Spector: Automatically Analyzing Shell Code, *Proc. 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp.501–514 (2007).
5) Murilo, N. and Steding-Jessen, K.: chkrootkit. http://www.chkrootkit.org/
6) Kim, G.H. and Spafford, E.H.: Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection, Technical Report CSD-TR-93-071, Purdue University (1993).
7) Linn, C. and Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly, *Proc. 10th ACM Conference on Computer and Communications Security* (*CCS '03*), pp.290–299 (2003).
8) Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables, *Proc. 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp.207–226 (2005).
9) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Network-Level Polymorphic Shellcode Detection Using Emulation, *Proc. 3rd Detection of Intrusions, Malware, and Vulnerability Assessment Conference* (*DIMVA '06*), pp.54–73 (2006).
10) Bania, P.: TAPiON (2005). http://pb.specialised.info/all/tapion/
11) The Metasploit Project: Metasploit. http://www.metasploit.com/
12) Kruegel, C., Robertson, W. and Vigna, G.: Using Alert Verification to Identify Successful Intrusion Attempts, *Practive in Information Processing and Communication*, Vol.27, No.4, pp.219–227 (2004).
13) Symantec: Linux.Slapper.Worm (2002). http://www.symantec.com/security_response/writeup.jsp?docid=2002-091311-5851-99
14) AlephOne: Smashing stack for fun and profit, Phrack (1996).
15) Wu-Ftpd Remote Format String Stack Overwrite Vulnerability (2000). http://www.securityfocus.com/bid/1387
16) Wu-Ftpd Debug Mode Client Hostname Format String Vulnerability (2001). http://www.securityfocus.com/bid/2296
17) OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability (2002). http://www.securityfocus.com/bid/5363
18) Nergal: The Advanced Return-into-lib(c) Exploits, *www.Phrack.org*, Vol.58, No.4 (2001).
19) Samba 'call_trans2open' Remote Buffer Overflow Vulnerability (2003). http://www.securityfocus.com/bid/7294
20) Lanzi, A., Sharif, M. and Lee, W.: K-Tracer: A System for Extracting Kernel Malware Behavior, *Proc. 16th Annual Network and Distributed System Security Symposium* (*NDSS '09*) (2009).
21) Moser, A., Kruegel, C. and Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis, *Proc. 2007 IEEE Symposium on Security and Privacy* (*S&P '07*), pp.231–245 (2007).
22) Tubella, J. and González, A.: Control speculation in multithreaded processors through dynamic loop detection, *Proc. 4th International Symposium on High Performance Computer Architecture* (*HPCA '98*), pp.14–23 (1998).
23) Newsome, J. and Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, *Proc. 12th Annual Network and Distributed System Security Symposium* (*NDSS '05*) (2005).
24) jt: Libdasm (2006). http://www.klake.org/~jt/misc/libdasm-1.5.tar.gz
25) SecurityFocus. http://www.securityfocus.com/
26) Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L. and Engler, D.R.: EXE: Automatically Generating Inputs of Death, *Proc. 13th ACM Conference on Computer and Communications Security* (*CCS '06*), pp.322–335 (2006).
27) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Emulation-Based Detection of Non-self-contained Polymorphic Shellcode, *Proc. 10th International Symposium on Recent Advances in Intrusion Detection* (*RAID '07*), pp.87–106 (2007).
28) Wang, X., Jhi, Y.-C., Zhu, S. and Liu, P.: STILL: Exploit Code Detection via Static Taint and Initialization, *Proc. 24th Annual Computer Security Applications Conference* (*ACSAC '08*), pp.289–298 (2008).
29) Hofmeyr, S.A., Forrest, S. and Somayaji, A.: Intrusion Detection Using Sequences of System Calls, *Journal of Computer Security*, Vol.6, pp.151–180 (1998).
30) Linn, C.M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S.K. and Hartman, J.: Protecting Against Unexpected System Calls, *Proc. 13th Usenix Security Symposium*, pp.239–254 (2005).
31) Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conference on Computer and Communications Security* (*CCS '02*) (2002).

(平成 21 年 1 月 6 日受付)
(平成 21 年 6 月 4 日採録)

© 2009 Information Processing Society of Japan

嶋村　　誠（学生会員）

1982 年生．2005 年電気通信大学情報工学科卒業．2007 年慶應義塾大学大学院理工学研究科開放環境科学専攻修士課程修了．現在，同専攻博士課程在学中．オペレーティングシステム，システムソフトウェア，インターネットセキュリティに興味を持つ．IPSJ，IEEE，ACM 各学生会員．

河野　健二（正会員）

1993 年東京大学理学部情報科学科卒業．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．現在，慶應義塾大学理工学部情報工学科准教授．博士（理学）．平成 11 年度情報処理学会論文賞受賞．平成 12 年度山下記念研究賞受賞．オペレーティングシステム，システムソフトウェア，インターネットセキュリティに興味を持つ．IEEE/CS，ACM，USENIX 各会員．

© 2009 Information Processing Society of Japan