Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

Yataglass: メモリスキャン攻撃を利用した攻撃コードの振る舞い解析

嶋村 誠[†1] 河野健二[†1,†2]

現在，攻撃者の任意のコードをサーバ上で動作させるリモートコードインジェクション攻撃がセキュリティ上大きな問題となっている．これに対し，攻撃コードを検知・解析するシステムとして，ネットワーク・コード・エミュレータが提案されている．ネットワーク・コード・エミュレータでは攻撃コードの擬似実行を行うことにより，攻撃コードを精度よく検知したり，攻撃コードの振る舞いを詳細に解析することができる．本論文では，被害プロセスのメモリ上のデータを命令列として利用する攻撃であるメモリスキャン攻撃が既存のネットワーク・コード・エミュレータを回避できることを示し，また，メモリスキャン攻撃を応用した攻撃コードの振る舞いを解析するネットワーク・コード・エミュレータである Yataglass を提案する．実際に Yataglass を作成し，実際の攻撃コードにメモリスキャン攻撃を適用し実験を行った結果，Yataglass は正しくメモリスキャン攻撃を適用した攻撃コードを解析できた．

Yataglass: Network-level Behavior Analyses of Memory Scanning Attacks

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

Remote code-injection attacks are one of the most frequently used attacking vectors in computer security. To detect and analyze injected code, some researchers have proposed network-level code emulators. A network-level code emulator emulates shellcode's behaviors and it thus can detect shellcode accurately and help analysts understand the behaviors. We demonstrated that *memory-scanning attacks* can evade current network-level code emulators, and propose Yataglass, an elaborated network-level code emulator, that enables us to analyze shellcode that incorporates memory-scanning attacks. According to our experimental results, Yataglass successfully emulated and analyzed real shellcode into which we had manually incorporated memory-scanning attacks.

1. はじめに

現在，リモートコードインジェクション攻撃がセキュリティ上の脅威となっている．リモートコードインジェクション攻撃では，攻撃者は様々なインターネットサーバの脆弱性を利用し，攻撃コードを実行させることで，サーバに被害を与える．このような攻撃の例としては，スタック溢れ攻撃[1)]，ヒープ上書き攻撃[2)]，フォーマット文字列攻撃[3)] などがある．

リモートコードインジェクション攻撃を検知するために，ネットワーク侵入検知システム (NIDS)[4)–6)] が広く使われている．しかし，攻撃者は NIDS を回避するために，暗号化や難読化を攻撃コードに適用している[7)–10)]．暗号化や難読化が行われた攻撃コードは静的な逆アセンブルができない．従って，攻撃コードをバイト列によって検知する NIDS[4),5)] や制御フローに基づいた検知を行う NIDS[6)] ではこのような攻撃コードを検知できない．

そのため，近年ではネットワーク・コード・エミュレータが提案されている．ネットワーク・コード・エミュレータでは，メッセージのバイト列を機械語命令列と見なして疑似実行を行うことでメッセージ中に含まれる攻撃コードを解析する．このため，解析結果が暗号化や難読化の影響を受けない．Polychronakis ら[11),12)] と，Zhang ら[13)] は攻撃コードの検知を行うネットワーク・コード・エミュレータを提案した．また，Spector[14)] は，攻撃コードの用いる Win32 API を抽出するネットワーク・コード・エミュレータである．

しかし，攻撃者は，既に簡単なネットワーク・コード・エミュレータを回避する方法を作成している．例えば，TAPiON エンコーダ[8)] は攻撃コードに意味のない浮動小数点命令や `rdtsc` 命令を挿入することで，浮動小数点命令や `rdtsc` 命令を実装していないエミュレータを回避する．Polychronakis ら[11)] のエミュレータは TAPiON でエンコードされた攻撃コードを正しく解析できている．しかし，攻撃者は今後もエミュレータを回避するための技術を作成する可能性がある．

本論文では，ネットワーク・コード・エミュレータをさらに有用な物にするため，ネットワーク・コード・エミュレータが持つ弱点と，その弱点を用いた回避技術について提示する．具体的には，Linn らによって示された**メモリスキャン攻撃**[15)] がネットワーク・コード・エミュレータを回避できることを示す．メモリスキャン攻撃は攻撃を受ける被害プロセスの

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

Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

メモリにある命令列を攻撃コードの一部として利用する攻撃である．既存のネットワーク・コード・エミュレータは被害プロセスの命令列を用いず，攻撃コードのみを用いて解析を行うため，このような攻撃コードに対処することができない．メモリスキャン攻撃の例として，被害プロセスのメモリから Intel x86 の `ret` 命令である `0xC3` を探して，見つかったアドレスへジャンプする攻撃が考えられる．この攻撃では，一旦攻撃コードから被害プロセスの命令列に制御が移るが，すぐに攻撃コードに制御が戻り，攻撃コードの続きを実行する．既存のネットワーク・コード・エミュレータでは，被害プロセスにある命令列が分からないため，この攻撃コードを正しく実行できない．本論文では，メモリスキャン攻撃が実際にネットワーク・コード・エミュレータを回避できることを示す．

また，本論文ではメモリスキャン攻撃を用いた攻撃コードを疑似実行するネットワーク・コード・エミュレータである Yataglass を提案する．Yataglass は，攻撃コードを疑似実行により解析し，攻撃コードが発行する Linux のシステムコールや Win32 API コールを抽出する．Yataglass では，メモリスキャン攻撃を解析するために，攻撃コードが探す命令列を攻撃コードから抽出する．その後，Yataglass は攻撃コードが探している命令列を書き込んだメモリ領域を用意し，その領域を被害プロセス上の命令列として使用させる．このようにして，Yataglass はメモリスキャン攻撃を解析し，攻撃コードがネットワーク・コード・エミュレータを回避することを防止する．なお，Yataglass は被害プロセスのメモリ上のデータを使わない．これは，攻撃コードが実際に挿入された状態を再現するには攻撃コードを挿入するための脆弱性に関する知識を要するが，既存のネットワーク・コード・エミュレータは未知の脆弱性に対する攻撃コードを解析するシステムとして使われるためである．このため，Yataglass は既存のシステムと同様に，攻撃コードのみを用いて疑似実行を行う．

実際に Yataglass のプロトタイプを Intel x86 アーキテクチャ上に実装し，Yataglass の有効性を示すために実験を行った．実験では，最新のネットワーク・コード・エミュレータである Spector[14] との比較を行った．Spector はメモリスキャン攻撃に耐性を持たないため，メモリスキャン攻撃を用いた攻撃コードを解析できない．一方，Yataglass はそのような攻撃コードを解析できる．7 つの実際の攻撃コードに対しメモリスキャン攻撃を適用した攻撃コードをそれぞれのシステムに解析させたところ，Spector はメモリスキャン攻撃の解析に失敗した．一方，Yataglass は正しく解析することができた．

本論文の構成は以下の通りである．まず第 2 章でネットワーク・コード・エミュレータについて説明する．次に第 3 章でメモリスキャン攻撃について説明する．第 4 章では Yataglass の行う攻撃コードの疑似実行について説明する．第 5 章では Yataglass にメモリスキャン攻撃を解析させた実験結果について示す．第 6 章では Yataglass の制限となる点について示す．第 7 章で関連研究をまとめる．最後に第 8 章で本論文をまとめる．

表 1 Intel x86 でよく使われるレジスタ．下部のレジスタはオペランドとして直接使うことはできない．
Table 1 Freqently used registers in Intel x86. Registers in lower half are not directly accessible in instruction operands.

レジスタ名	説明
eax, ebx, ecx, edx	汎用レジスタ
esi, edi	ストリング命令に用いるレジスタ
esp	スタックポインタ
ebp	ベースポインタ
eip	命令カウンタ
eflags	特別な命令のためのフラグレジスタ (例：`jcc` の条件分岐に使用)

2. ネットワーク・コード・エミュレータ

ネットワーク・コード・エミュレータは，リモートコードインジェクション攻撃に対して，メッセージ中に含まれる攻撃コードの解析を疑似実行を用いて行う．具体的には，ネットワーク・コード・エミュレータは仮想的なレジスタとメモリを用意し，攻撃コードに従ってこれらを操作することで，攻撃コードを解析する．**図 1** に Intel x86 アーキテクチャの代表的なレジスタを示す．攻撃コードがメッセージ中のどの位置に存在するかは事前にはわからないため，ネットワーク・コード・エミュレータはメッセージ中の攻撃コードの位置を特定するために，様々なアプローチを用いる．例えば，Polychronakis ら[11] は命令列の実行をメッセージの全てのバイト位置から行うことで，攻撃コードを検索する．Spector[14] では，解析の前に NIDS を用いて攻撃コードの位置を特定する．

ネットワーク・コード・エミュレータには二つの利点がある．第一に，暗号化や難読化が施された攻撃コードを解析することができる．暗号化された攻撃コードはエミュレータで実行することにより復号化される．また，エミュレータによる解析結果は難読化の影響を受けない．第二に，無害なメッセージを攻撃コードであると誤検知する可能性が低い．これまでの研究により，無害なメッセージが十分な長さの命令列として実行できてしまうことは少ないことが示されている[11),16),17]．従って，メッセージ中のバイト列を用いた検知システム[4),5)] より誤検知の可能性が少ない．

ネットワーク・コード・エミュレータによる疑似実行の結果は以下の二つの目的で使われる．第一に，管理者が疑似実行の結果を用いて，攻撃コードの目的を判断できる．疑似実行

Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

の結果は実行された命令列とシステムコール呼び出しを含むので，管理者が攻撃コードの目的を判断できる．例えば，Spector[14] では，攻撃コードの Win32 API コールを抽出することで，攻撃コードの目的の理解を助ける．第二に，自己改変を行う暗号化された攻撃コードの検知に使うことができる．このためには，疑似実行の結果から攻撃コードを発見する手法を必要とする．例えば，Polychronakis らのエミュレータ[11] と Zhang[13] らのエミュレータでは，1) 命令カウンタを取り出す *GetPC* と呼ばれる命令列を用いるかどうか，2) 自己改変を行うためにメッセージの中身を読み出すかどうかという二点を用いて攻撃コードを探す．

2.1 エミュレーションの回避可能性

攻撃者はネットワーク・コード・エミュレータを大きく分けて二つの手法で回避できる．第一に，攻撃コードはエミュレータが実装していない機能を利用して，エミュレータを回避できる．例えば，既知のファイル (設定ファイルなど) を読み出し，書式が予想と異なる場合，それ以降の動作を停止する攻撃コードが考えられる．この場合，エミュレータは書式チェック以降の攻撃コードの振る舞いを抽出できない．この回避手法を防止するには他のエミュレータで用いられている手法[18)–20)] を使用し，チェックを成功させ実行を続ければよい．

第二に，攻撃コードは被害プロセスのメモリ上のデータを命令列として用いて，エミュレータを回避できる．この場合，エミュレータは被害プロセスのメモリ状態を持たないため，実行を続行できない．メモリスキャン攻撃は被害プロセスのメモリ上のデータを用いる回避手法である．本論文ではこの攻撃への対策を行ったエミュレータを提案する．

3. メモリスキャン攻撃

メモリスキャン攻撃では，被害プロセスのメモリ上のデータを命令列として用いる．攻撃者は以下の 2 つの方法で被害プロセスの持つ命令列を使用できる．

- **既知のアドレスにある命令列の利用**　攻撃コードは，被害プロセス中の命令列のアドレスを指定し制御を移すことで，命令列を用いることができる．このためには，攻撃者は，あらかじめ被害プロセスのプログラムと同じプログラムを自分の計算機で動作させ，命令列のアドレスを確定しなければならない．Polychronakis ら[11] は，この攻撃は被害プロセスのメモリ配置に対し強く依存するため，大きな問題にはならないと述べた．
- **プロセス中から発見したメモリの利用 (メモリスキャン攻撃)**　攻撃コードは，被害プロセスの中から有用な命令列を探し，発見した命令列を用いることができる．Linn ら[15] はこのような攻撃をホスト侵入検知システムを回避するための攻撃として示した．しかし，この攻撃はネットワーク・コード・エミュレータを回避するために使われる可能性がある．この攻撃は被害プロセスのメモリ配置に依存せず命令列を探すことができるため，固定アドレスにある命令列を利用する攻撃より容易に使うことができる．本論文ではこのタイプの攻撃に着目する．

メモリスキャン攻撃では，攻撃コードは 3 つの機能を持つ．第一に，攻撃コードは命令列の検索を開始するアドレスを探す．第二に，攻撃コードは**スキャニング・ループ**によって，被害プロセスのメモリから使用可能な命令列を探す．第三に，攻撃コードは制御移行命令によって発見した命令列に制御を移す．

スキャンを開始するアドレスとして，スタックに存在するリターンアドレスを用いることができる．これは，リターンアドレスはプロセスのコード領域のアドレスであることが多く，攻撃コードが実行可能な命令列を探せるためである．攻撃者は，被害プロセスのメモリ配置を全て把握することは難しい．しかし，攻撃コードに制御が移ったときのスタックの状態やリターンアドレスの位置は比較的容易に把握できる．これは，リターンアドレスの位置が呼出規約によって決定されているためである．このため，リターンアドレスの位置はスタックポインタからのオフセットで決定すればよい．もし，リターンアドレスの位置が被害プロセスの状況によって異なる場合には，スタックからリターンアドレスを探せばよい．例えば，ELF フォーマットの規約[21] では，コード領域は `0x8048000` より上位のアドレスにあることが定められている．このため，攻撃コードは，`0x8048000` と `0x8060000` の間にある数値をスタックから探せば，高い確率でコード領域のアドレスを見つけることができる．

メモリスキャン攻撃の例を**図 1** に示す．メモリスキャン攻撃は 5 つの手続きから成る．まず，GetPC と呼ばれる手続きにより，攻撃コードは自分自身のアドレスを得る．これは x86 アーキテクチャでは命令カウンタ相対でのメモリアクセスができないため，攻撃コード中の暗号化されたデータのアドレスを得なければならないためである．例えば，攻撃コードは，`call` 命令を用いて `eip` の値をスタックに書き込むことで，自身のアドレスを得ることができる．次に，攻撃コードはリターンアドレスを上記の方法を使ってスタックから取り出す．その後，入手したリターンアドレスからコード領域の中の `ret` 命令を探す．`ret` 命令を見つけた場合，攻撃コードは制御を `ret` 命令に移す．その後，`ret` 命令が実行され，すぐに制御は攻撃コードへ戻る．最後に，攻撃コードは復号ループを用いて，攻撃コードの本体を暗号化されたデータから復号し，実行する．

この攻撃は既存のネットワーク・コード・エミュレータによる疑似実行を中断させる．既存のエミュレータは，被害プロセスのメモリ領域を持たないため，攻撃コード以外のメモリ領域へアクセスするとエラーを発生し停止する．もし，エミュレータがこのエラーを無視

Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

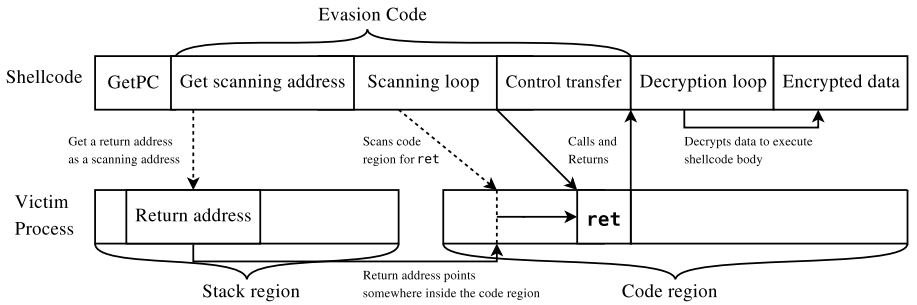

図 1 メモリスキャン攻撃の例.
Fig. 1 Example of memory-scanning attack.

し，攻撃コードの実行を続けたとしても，攻撃コードは被害プロセス中の命令列を実行するため，攻撃コードが正しく実行された時の状態とエミュレータの状態が異なってしまう．従って，エミュレータは被害プロセスから制御が戻った後の命令を正しく実行できない．

4. Yataglass

本論文では，メモリスキャン攻撃の影響を受けないネットワーク・コード・エミュレータである Yataglass を提案する．Yataglass は Spector[14] と同様に攻撃コードを疑似的に実行し，解析結果として，実行した命令列と，攻撃コードの用いるシステムコールを抽出する．Yataglass は Spector とは異なり，メモリスキャン攻撃を解析できる．

メモリスキャン攻撃による Yataglass の回避を防止するため，Yataglass は実行中にスキャニング・ループを見つけると，そのスキャニング・ループが検索する命令列を推測する．そして推測した命令列を書き込んだメモリ領域を用意し，攻撃コードにこの領域を使用させることで，あたかもスキャニング・ループが命令列を発見できたかのようにする．このようにして，Yataglass はメモリスキャン攻撃に影響を受けないようにする．

Yataglass は初期化時にレジスタとスタックとして用いるメモリ領域を初期化し，攻撃コードをメモリに読み込む．その後，Yataglass は攻撃コードの実行を，`exit()` などの被害プロセスを終了させるシステムコールや，`execve()` などの他のプロセスへ制御を移すシステムコールが発行されるまで行う．Yataglass が攻撃コードを実行している途中でシステムコールの呼び出しを発見した場合には，システムコールの名前と引数を記録する．

4.1 Yataglass の擬似実行エンジン

4.1.1 記号的実行

Yataglass は，攻撃コードの擬似実行を行う時に，全てのレジスタとメモリの内容を *Value* という記号として扱い，Value に対して計算を行う．Value には，*Number*, *Symbol*, *Expression*, *Unknown* の 4 種類がある．Number は数値を表す Value であり，具体的な確定している値である．例えば 32 ビット整数は Number である．Expression は演算子と Value2 つから成る 3 つ組で，2 つの Value の計算結果を表す．例えば X と Y に対して，(ADD X Y) は $X+Y$ を表す．Expression の値は両方の Value が確定している場合にのみ確定する．Unknown と Symbol は両方共に不定の値を表す．しかし，Symbol は攻撃コードの実行に影響を及ぼす可能性がある値である．一方，Unknown は攻撃コードの実行に影響を与えない．例えば，Yataglass は初期化時に *STACK_PTR* という Symbol を `esp` に割り当てる．これは，`esp` の初期状態はわからないが，攻撃コードは `esp` 相対でメモリアクセスを行うことがあるためである．対して，未初期化のメモリ領域から読み出した値は Unknown になる．

Yataglass はメモリスキャン攻撃を解析するために，被害プロセスのメモリの読み出しをメモリスキャン攻撃の一部と見なす．これは，メモリスキャン攻撃が被害プロセスのコード領域のアドレスを探すためである．メモリスキャン攻撃による被害プロセスのメモリへのアクセスを検出するため，Yataglass では初期化時に，被害プロセスのコード領域へのポインタとして扱われる可能性のある値に $CODE\_PTR_n$ という symbol を割り当てる．なお，ここで n は自然数である．例えば，スタックの初期状態では，スタックが $CODE\_PTR_n$ で埋まっているものとして扱う．また，Yataglass は，被害プロセスのコード領域がアクセスされた際には，その領域の各バイトに $CODE_n$ という Symbol を割り当てることで，被害プロセスのコード領域を扱う．

Yataglass は，Value に対して制約条件を持たせることができる．この制約条件は，Value が持つことができる値の範囲として表す．例えば，X が 10 以上 20 以下の範囲の値を持ちうるとき，X は `[10, 20]` という制約条件を持つ．また，Y が 120 以外の 8 ビットの整数値のとき，`[0,119]` と `[121,255]` が制約条件となる．第 4.1.2 節では，この制約条件を使って，Yataglass がどのようにスキャニング・ループを解析するかを示す．

4.1.2 条件分岐を用いた命令列の推測

Yataglass では，スキャニング・ループが探している被害プロセス中の命令列をループの脱出条件から推測する．攻撃コード中のループは多くの場合条件分岐命令で脱出する．このとき，フラグレジスタ（`eflags`）には，条件分岐に用いられるフラグが保持されており，

Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

このフラグの内容は直前の演算命令の結果で決まる．Yataglass は条件分岐を発見すると，eflags の内容が確定しているかどうかを判断する．確定した値が条件分岐に使われている場合は，Yataglass はそのまま条件に従い実行を続ける．Borders ら[14] も述べているとおり，通常，攻撃コードはコードサイズが小さいため，確定する条件分岐を用いて決定的に動作する．しかし，メモリスキャン攻撃は被害プロセスのメモリ状態がわからないため，不定な値が条件分岐に使われた場合，非決定的な動作をしなければならない．

図 2 にスキャニング・ループの例を示す．このループは 0xC3（ret 命令）を *ADDR* で示されるコード領域から検索する．4 行目の cmp 命令は *ADDR* に由来する不定の値を比較する．このため，5 行目の条件分岐は Yataglass 上では非決定的な条件分岐である．

この非決定性を解決するために，Yataglass は不定の値による条件分岐を発見すると，Yataglass のインスタンスを fork() によってもう一つ生成し，実行の状態をコピーする．その後，片方のインスタンスは条件が成立したと仮定した実行を行い，もう片方は条件が不成立であると仮定した実行を行う．インスタンスの数が爆発することを防ぐため，Yataglass は一度実行した条件分岐に再び達したとき，実行を終了する．図 2 では，Yataglass は 5 行目で実行を二つに分け，条件分岐が成立した実行とそうでない実行を行う．5 行目で分岐したインスタンスは，2 行目に戻り，再び 5 行目に達したときに実行を終了する．もう一方のインスタンスはループを脱出し実行を続ける．

スキャニング・ループが探す命令列を推測するために，Yataglass は条件分岐に不定の値が使われた場合，その値に制約条件をつける．制約条件を求めるために，Yataglass は，不定の値を使う命令が条件分岐に影響を与えたかどうかを調べる．図 2 の例では，4 行目の cmp 命令が ZF (ゼロフラグ)， SF (符号フラグ)， OF（桁溢れフラグ），PF（パリティフラグ）を設定する．このとき，Yataglass は，(CMP $CODE_1$ 0xC3) をこれらのフラグと関連づける．ここで，$CODE_1$ は，被害プロセスのコード領域のアドレスである *CODE_PTR+1* を参照することで作られた新しい Value である．

Yataglass は，条件分岐が成立する場合と成立しない場合で，条件フラグを設定した命令の集合を用いて，Value に制約条件をつける．図 2 の例では，Yataglass は，5 行目で $CODE_1$([edi]) が，0xC3 であるとする場合とそうしない場合の実行をそれぞれ行う．このようにして，Yataglass は攻撃コードに被害プロセス中から探し出す命令を見つけたと思わせ，実行を続ける．このため，8 行目では，$CODE_1$([edi]) の値は 0xC3 になる．

x86 アーキテクチャでは，ストリング命令である scas，cmps によって，ループなしに命令列を検索できる．例えば，repne scasb 命令は，edi が指すメモリ領域から，eax の下位

```
# ADDR はコード領域のアドレス
1:   mov edi, ADDR        # edi = ADDR (CODE_PTR)
2: loop:
3:   inc edi              # [edi] = CODE1
4:   cmp byte [edi], 0xC3 # 'ret' と比較
5:   je  loopout          # if(*edi=='ret') goto 8;
6:   jmp loop             # else goto 2
7: loopout:               # CONT をスタックへ積み,
8:   call edi             # 'ret' へジャンプ
9: CONT:
```

図 2 スキャニング・ループの例.
Fig. 2 Example of scanning loop.

1 バイトと等しいデータを探す．この命令はデータが見つかるか，ecx が 0 になると終了する．Yataglass はこの場合，[edi] に eax の下位 1 バイトを書き込んだ実行と，ecx を 0 にした実行を行う．また，repe cmpsb 命令は，esi が指すバイト列と edi が指すバイト列が同じかどうかを調べる．Yataglass はこの場合，esi もしくは edi で示される確定しているバイト列を不定なほうのメモリに設定した実行と，ecx をデクリメントした実行を行う．

4.2 実　　装

Yataglass のプロトタイプを実装した．X86 命令のデコードには，libdasm[22] を用いた．エミュレータは 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，特権命令は実行しない．しかし，全ての命令を正しくデコードできる．

Windows に対する攻撃コードを解析するために，Process Environment Block (PEB) のスタブを用意した．Yataglass は初期化時によく使用される DLL である kernel32.dll, user32.dll, ws2_32.dll をロードし，攻撃コードがこのデータを使い API のアドレスを探す場合を扱えるようにした．さらに，Win32 API のスタブとして，LoadLibrary() や GetProcAddress() など，いくつかの API を実装した．これにより，LoadLibrary() や GetProcAddress() を用いる攻撃コードを解析できる．

5. 実　　験

5.1 メモリスキャン攻撃を利用する攻撃コードの作成

本章では，実際に Yataglass がメモリスキャン攻撃を使う攻撃コードを解析できることを示す．また，比較として，Spector[14] を独自に実装したものを用いた．Spector がシステムコールを抽出する能力は Yataglass と同じだが，Spector はメモリスキャン攻撃を用いた攻

表 2 実験で用いた攻撃コード
Table 2 Shellcodes used in the experiments.

ファイル名	攻撃対象	入手元	CVE 番号	目的	エンコード
tsig.c	bind <= 8.2.2	SecurityFocus	2001-0010	シェルを起動	なし
7350wurm.c	wu-ftpd <= 2.6.1	milw0rm	2001-0550	シェルを起動	なし
rsync-expl.c	rsync <= 2.5.1	SecurityFocus	2002-0048	バックドア作成	なし
7350owex.c	wu-imap 2000.287	milw0rm	2002-0379	シェルを起動	ToUpper 回避
OpenFuck.c	Apache with OpenSSL <=0.9.6d	SecurityFocus	2002-0656	シェルを起動	なし
sambal.c	Samba 2.2.8	SecurityFocus	2003-0201	バックドア作成	なし
cyruspop3d.c	cyrus-pop3d 2.3.2	milw0rm	2006-2502	バックドア作成	なし

撃コードを解析できない．以下では，我々の Spector の実装を Spector と呼ぶ．

メモリスキャン攻撃を実際の攻撃コードに適用した．使用した攻撃コードは，named[23]，wu-ftpd[24]，rsync[25]，wu-imap[26]，Apache Web Server[27]，samba[28]，cyrus-pop3d[29] に対する攻撃コードである．**表 2** に，攻撃コードのソースファイル名，攻撃の対象であるサーバ名とバージョン，攻撃コードの入手元，攻撃コードの利用する脆弱性の CVE 番号，攻撃コードの目的，使用されているエンコード方法について示した．

用いた攻撃コードの多くは `esp` レジスタを破壊するものであった．このため，被害プロセスのコード領域のアドレスを `ebp` から得るメモリスキャン攻撃を作成した．しかし，`ebp` を破壊する攻撃コードもある．このような攻撃コードでは，`esp` からコード領域のアドレスを得ればよい．攻撃者は用いる攻撃コードがどのレジスタを破壊するのかを知っているため，用いるメモリスキャン攻撃のコードを選べる．実験では，`cyruspop3d.c` が `ebp` を破壊する攻撃であったため，被害プロセスのコード領域のアドレスを `esp` から得るようにした．

また．メモリスキャン攻撃で用いるコードは NULL バイトを含まないように作成した．これは，多くの脆弱性が，挿入されるコードを C の文字列として扱っており，NULL バイトが文字列の終端と見なされてしまうためである．また，wu-imap に対する脆弱性では，wu-imap が `ToUpper()` をフィルタとして使うため，小文字になる値を含む命令列を用いることができない．このため，攻撃コード中の小文字を動的に生成するコードを作成した．

5.2 メモリスキャン攻撃を利用する攻撃コードの解析

図 3 に Yataglass が `rsync-expl.c` から生成されたメモリスキャン攻撃を利用する攻撃コードを実行した結果を示す．各行には命令番号 (16 進)，実行したアドレス (16 進)，命令とニーモニック，コメントを示す．この結果によると，Yataglass はメモリスキャン攻撃を正しく扱えていることがわかる．(命令番号 `004d` から `0067`)．初めに，攻撃コードは `eax` にスタック上のリターンアドレスを代入する．このとき，Yataglass は `eax` を *`CODE_PTR`* に

表 3 攻撃コードに関する実行結果
Table 3 Summary of emulation results.

ファイル名	Yataglass	Spector
tsig.c (未改変)	√	√
tsig.c (改変済)	√	−
7350wurm.c (未改変)	√	√
7350wurm.c (改変済)	√	−
rsync-expl.c (未改変)	√	√
rsync-expl.c (改変済)	√	−
7350owex.c (未改変)	√	√
7350owex.c (改変済)	√	−

ファイル名	Yataglass	Spector
OpenFuck.c (未改変)	√	√
OpenFuck.c (改変済)	√	−
sambal.c (未改変)	√	√
sambal.c (改変済)	√	−
cyruspop3d.c (未改変)	√	√
cyruspop3d.c (改変済)	√	−

```
Emulation start from 00000000
No.  Addr. Inst.          Mnemonic           Note
------------------------------------------------------------------
0040 0076  31db          xor ebx,ebx
0041 0078  53            push ebx
0042 0079  686e2f7368    push dword 0x68732f6e
0043 007e  682f2f6269    push dword 0x69622f2f
0044 0083  89e3          mov ebx,esp
0045 0085  8d542408      lea edx,[esp+0x8]
0046 0089  31c9          xor ecx,ecx
0047 008b  51            push ecx
0048 008c  53            push ebx           # (SUB STACK 0x50)
0049 008d  8d0c24        lea ecx,[esp]
004a 0090  31c0          xor eax,eax
004b 0092  b00b          mov al,0xb         # Syscall No. of execve
004c 0094  60            pusha              # Save registers
004d 0095  89ee          mov esi,ebp        # ebp = STACK
004e 0097  81c6fcffffff  add esi,0xfffffffc # esi = STACK - 4
004f 009d  8b06          mov eax,[esi]      # eax = CODE_PTR
0050 009f  3d01900408    cmp eax,0x8049001  # Avoids null byte
    # compared CODE_PTR and 0x8049001
    # symbol: (CODE_PTR AT 0xbffe10fc)
0051 00a4  7cf1          jl 0x97
    # conditional jump: (CMP (CODE_PTR AT 0xbffe10fc) 0x8049001)
    #### (forked and child process terminates) ####
    # symbol: (CODE_PTR AT 0xbffe10f8)
0052 00a6  3d10101008    cmp eax,0x8101010 # Avoids null byte
    compared CODE_PTR and 8101010
    symbol: (CODE_PTR AT 0xbffe10fc)
0053 00ab  7fea          jg 0x97
    conditional jump: (CMP (CODE_PTR AT 0xbffe10fc) 0x8101010)
    #### (forked and child process terminates) ####
0054 00ad  d9ee          fldz
0055 00af  d97424d0      fstenv [esp-0x30]
0056 00b3  8b7424dc      mov esi,[esp-0x24]
0057 00b7  89c7          mov edi,eax
0058 00b9  b05d          mov al,0x5d
0059 00bb  b9ffffffff    mov ecx,0xffffffff
005a 00c0  fd            std
005b 00c1  47            inc edi
005c 00c2  803f5d        cmp byte [edi],0x5d
    compared CODE_1 and 5d
005d 00c5  75fa          jnz 0xc1
    conditional jump symbol: (CMP CODE_1 0x5d)
    assign_value: CODE_1 = 0x5d
005e 00c7  47            inc edi
005f 00c8  803fc3        cmp byte [edi],0xc3
    compared CODE_2 and c3
0060 00cb  75f4          jnz 0xc1
    conditional jump symbol: (CMP CODE_2 0xc3)
    assign_value: CODE_2 = 0xc3
0061 00cd  83c628        add esi,0x28
0062 00d0  4f            dec edi
0063 00d1  56            push esi
0064 00d2  55            push ebp
0065 00d3  ffe7          jmp edi  # Jump to victim's memory
0066 ----  5d            pop ebp  # CODE_1
0067 ----  c3            ret      # CODE_2
0068 00d5  61            popa
0069 00d6  cd80          int 0x80
    Linux system call 11 (execve) detected!!
        path=//bin/sh |CONCRETE|
        argv[0]=//bin/sh |CONCRETE|
```

図 3 `rsync-expl.c` から生成された，メモリスキャン攻撃を利用する攻撃コードの Yataglass による実行結果．初めの 64 個 (0x40) の命令，および条件分岐で分岐した実行からの出力は省略した．
Fig. 3 The emulation result of shellcode generated by `rsync-expl.c`, incorporated with memory-scanning attack, with Yataglass. Logs of first 64 (0x40) instructions and outputs from forked instances are omitted.

関連づける．次に，攻撃コードは `eax` を `0x8049001`，`0x8101010` とそれぞれ比較し，`eax` が指す領域がコード領域であることを確定する．Yataglass はこの比較ループを `eax` に制約条件をつけて抜ける．その後，攻撃コードは，被害プロセス中の命令列によってリターンアドレスとして使われるアドレスを得る．(攻撃コードの `00d5` 番地)．また，攻撃コードは

Vol.2009-ARC-183 No.9
Vol.2009-OS-111 No.9
2009/4/22

```
Emulation start from 00000000
No.  Addr. Inst.        Mnemonic            Note
-------------------------------------------------------
0040 0076  31db         xor ebx,ebx
0041 0078  53           push ebx
0042 0079  686e2f7368   push dword 0x68732f6e
0043 007e  682f2f6269   push dword 0x69622f2f
0044 0083  89e3         mov ebx,esp
0045 0085  8d542408     lea edx,[esp+0x8]
0046 0089  31c9         xor ecx,ecx
0047 008b  51           push ecx
0048 008c  53           push ebx
0049 008d  8d0c24       lea ecx,[esp]
004a 0090  31c0         xor eax,eax
004b 0092  b00b         mov al,0xb
004c 0094  60           pusha
004d 0095  89ee         mov esi,ebp         # ebp = unknown
004e 0097  81c6fcffffff add esi,0xfffffffc  # esi = unknown
004f 009d  8b06         mov eax,[esi]       # Unknown を参照した
MEMORY FAIL -- unknown address is used
```

図 4 rsync-expl.c から生成された，メモリスキャン攻撃を利用する攻撃コードの Spector による解析結果．初めの 64 個 (0x40) の命令は省略した．

Fig. 4 Emulation result of shellcode generated by rsync-expl.c, incorporated with memory-scanning attack, with Spector. First 64 (0x40) instructions are omitted.

スキャニング・ループにより，被害プロセス中の命令列から pop ebp と ret の並びを探す．Yataglass は 0x5D，0xC3 を *CODE_1*，*CODE_2* としてそれぞれ用意する．スキャニング・ループの後で，攻撃コードは発見した命令列 (*CODE_1*) に制御を移す (命令番号 0065)．すると，攻撃コードは pop ebp, ret を実行し，攻撃コードの 00d5 番地へ戻る (命令番号 0068)．最後に，攻撃コードは execve() システムコールを用いて，/bin/sh を実行する．

図 4 に Spector による同じ攻撃コードの実行結果を示す．実行結果では，Spector は攻撃コードが被害プロセスのメモリを探しに行ったときに，エラーを出して実行を停止している (004f 行目)．従って，Spector はこの攻撃コードを正しく実行できていない．

また，**表 3** に Yataglass と Spector が攻撃コードを解析した結果を示す．Spector はメモリスキャン攻撃を利用するよう改変した攻撃コードを解析できなかった．一方，Yataglass は全ての攻撃を解析できた．

6. 議　　論

現在の Yataglass の実装はメモリスキャン攻撃がスタックからスキャンの開始アドレスを得ると仮定している．従って，攻撃者は他の方法でスキャンの開始アドレスを得ることで，Yataglass を回避する可能性がある．例えば，攻撃コード中で，/proc/XXX/maps(XXX は被害プロセスのプロセス ID) からメモリマップを取得し，これを用いて，コード領域のアドレスを確定できる．しかし，この場合は Yataglass が適切に作成した/proc/XXX/maps を攻撃コードに見せることで，回避を防止できる．また，攻撃コードは GOT (Global Offset Table) や PLT (Procedure Linkage Table) のような関数テーブルからコード領域のアドレスを得ることができる．この場合は，Yataglass が偽の GOT や PLT を持たせればよい．

また，今後，Yataglass はより優れたスキャニング・ループに対処しなければならない．例えば，スキャニング・ループが同時に**複数の**命令を検索する場合，現在の Yataglass では命令列を確定できない．例えば，x86 アーキテクチャの pop 命令は 8 種類で，0x58 から 0x5F の範囲にある．攻撃者は，スキャニング・ループをこれらの 8 種類の pop 命令のどれでも良いように作成し，適当なデータをスタックに積んで，その命令に制御を移す攻撃コードを作成できる．このような攻撃コードを解析するには，Yataglass は未確定の Value が取る値の数がある閾値を切った時に，それぞれの値を仮定し実行すればよい．10 種類を超える命令を同時に受理するようスキャニング・ループを作るのは難しいと考えられる．このため，この拡張を行った場合でも，Yataglass のインスタンスの数が爆発する可能性は少ない．

最後に，Yataglass は攻撃者が被害プロセス中の制御変数を書き換えて実行を変えてしまうという攻撃には対処できない．これは，Yataglass が被害プロセスのメモリ情報を用いないことを仮定しているためである．第 3 章でも述べたように，攻撃コードは被害プロセスのメモリ情報に依存しないで作成されることが多い．これは，Address Space Randomization[30] などの技術が広く用いられているためである．従って，このような攻撃コードは Yataglass で扱えなかったとしても，成功しサーバに被害を与えることは少ない．

7. 関 連 研 究

ネットワークメッセージを静的に解析することにより攻撃メッセージを検出する手法が提案されている．SigFree[17] は，メッセージの命令列として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，メッセージの全てのバイト位置から逆アセンブルを行い，実行可能なバイト列の長さが閾値を超えた場合，メッセージを攻撃コードであるとみなす．Kruegel ら[16] 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．この手法では，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することで，命令列であるかどうかを判定する．Yataglass ではこれらのシステムとは異なり，疑似実行により攻撃コードの実行時の振る舞いを解析する．

ホスト侵入検知システム[31),32] 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することができる．しかし，これらのシステムでは，ファイルが読み出されてしまっただけの場合や，このような検知システムから逃れるプログラムを使用された場合，管理者は被害に気がつくことができない．Yataglass では，攻撃コードの振る舞いを解析するため，管理者はより正確に被害を特定することができる．

8. 終 わ り に

リモートコードインジェクション攻撃では，サーバの脆弱性を利用し，攻撃コードと呼ば

れる機械語命令列をサーバに挿入し実行させる．攻撃コードを検知・解析するシステムとして，攻撃コードの擬似実行を行うことで攻撃コードを検知・解析するネットワーク・コード・エミュレータが提案されている．本論文では，既存のネットワーク・コード・エミュレータは被害プロセスのメモリ上のデータを命令列として利用するメモリスキャン攻撃により回避できることを示し，また，メモリスキャン攻撃を利用した攻撃コードの振る舞いを解析するネットワーク・コード・エミュレータである Yataglass を提案した．実際に Yataglass のプロトタイプを作成し，実際の攻撃コードにメモリスキャン攻撃を適用し実験を行った結果，Yataglass は正しくメモリスキャン攻撃を適用した攻撃コードを解析できた．今後の課題としては，より高度なメモリスキャン攻撃への対策を考える必要がある．

参 考 文 献

1) AlephOne: Smashing stack for fun and profit, Phrack (1996).
2) Sotirov, A.: Apache OpenSSL heap overflow exploit, `http://www.phreedom.org/research/exploits/apache-openssl/` (2002).
3) Li, W. and cher Chiueh, T.: Automated Format String Attack Prevention for Win32/X86 Binaries, *Proc. of the 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp.398–409 (2007).
4) Roesch, M.: Snort: Lightweight Intrusion Detection for Networks, *Proc. of the 13th USENIX Conference on Systems Administration* (*LISA '99*), pp.229–238 (1999).
5) Paxson, V.: Bro: a system for detecting network intruders in real-time, *Computer Networks*, Vol.31, No.23–24, pp.2435–2463 (1999).
6) Chinchani, R. and Berg, E. V.D.: A Fast Static Analysis Approach to Detect Exploit Code in Network Flows, *Proc. of the 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp.284–308 (2005).
7) The Metasploit Project: Metasploit, `http://www.metasploit.com/`.
8) P.Bania: TAPiON, `http://pb.specialised.info/all/tapion/` (2005).
9) K2: ADMMutate, `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz` (2007).
10) Sedalo, M.: JempiSCode, `http://goodfellas.shellcode.com.ar/proyectos.html` (2006).
11) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Network-Level Polymorphic Shellcode Detection Using Emulation, *Proc. of the 3rd Conference on Detection of Intrusions, Malware, and Vulnerability Assessment* (*DIMVA '06*) , pp.54–73 (2006).
12) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Emulation-Based Detection of Non-self-contained Polymorphic Shellcode, *Proc. of the 10th International Symposium on Recent Advances in Intrusion Detection* (*RAID '07*), pp.87–106 (2007).
13) Zhang, Q., Reeves, D.S., Ning, P. and lyer, S.P.: Analyzing Network Traffic To Detect Self-Decrypting Exploit Code, *Proc. of the 2nd ASIAN ACM Symbopsium on Information, Computer and Communications Security* (*ASIACCS '07*), pp.4–12 (2007).
14) Borders, K., Prakash, A. and Zielinski, M.: Spector: Automatically Analyzing Shell Code, *Proc. of the 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp.501–514 (2007).
15) Linn, C.M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S.K. and Hartman, J.: Protecting Against Unexpected System Calls, *Proc. of the 13th Usenix Security Symposium*, pp.239–254 (2005).
16) Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables, *Proc. of the 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp.207–226 (2005).
17) Wang, X., Pan, C.-C., Liu, P. and Zhu, S.: SigFree: A Signature-free Buffer Overflow Attack Blocker, *Proc. of the 15th Usenix Security Symposium*, pp.225–240 (2006).
18) Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L. and Engler, D.R.: EXE: Automatically Generating Inputs of Death, *Proc. of the 13th ACM Conference on Computer and Communications Security* (*CCS '06*), pp.322–335 (2006).
19) Moser, A., Kruegel, C. and Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis, *Proc. of the 2007 IEEE Symposium on Security and Privacy* (*S&P '07*), pp.231–245 (2007).
20) Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D. and Yin, H.: BitScope: Automatically Dissecting Malicious Binaries, Technical Report CMU-CS-07-133, Carnegie Mellon University (2007).
21) The Santa Cruz Operation, Inc.: *SystemV Application Binary Interface Intel 386 Architecture Processor Supplement*. `http://www.caldera.com/developers/devspecs/abi386-4.pdf`.
22) jt: Libdasm, `http://www.klake.org/~jt/misc/libdasm-1.5.tar.gz` (2006).
23) MITRE: ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability, `http://www.securityfocus.com/bid/2302` (2001).
24) MITRE: Wu-Ftpd File Golbbing Heap Corruption Vulnerability, `http://securityfocus.com/bid/3581` (2001).
25) MITRE: rsync Signed Array Index Remote Code Execution Vulnerability, `http://www.securityfocus.com/bid/3958` (2002).
26) MITRE: Wu-imapd Partial Mailbox Attribute Remote Buffer Overflow Vulnerability, `http://securityfocus.com/bid/4713` (2002).
27) MITRE: OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability, `http://www.securityfocus.com/bid/5363` (2002).
28) MITRE: Samba 'call_trans2open' Remote Buffer Overflow Vulnerability, `http://securityfocus.com/bid/7294` (2003).
29) MITRE: Cyrus IMAPD POP3D Remote Buffer Overflow Vulnerability, `http://www.securityfocus.com/bid/18506` (2006).
30) PaX Team: PaX address space layout randomization (ASLR), `http://pax.grsecurity.net/docs/aslr.txt`.
31) Murilo, N. and Steding-Jessen, K.: chkrootkit, `http://www.chkrootkit.org/`.
32) 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).