2007－CSEC－39 (11)
2007／12／14

バイナリプログラムの書換えによるバッファオーバーフロー検出の一手法

山川 高明† 真野 芳久‡
†南山大学大学院数理情報研究科 ‡南山大学数理情報学部
〒489-0863 愛知県瀬戸市せいれい町27
E-mail: †m06mm034@nanzan-u.ac.jp, ‡ymano@nanzan-u.ac.jp

概要 現在でも数多く発見され攻撃に利用されているバッファオーバーフロー脆弱性への対策として、バイナリプログラムの書換えによってバッファオーバーフローの発生を検知し攻撃を防ぐ一つの方法について述べる。そこでは、関数の入口及び出口で戻り番地を退避・検査できるようにするために、呼出し規約及びコンパイラ出力コードパターン情報を利用している。x86 システム上の Windows 環境を対象として議論し、実験とその結果についても述べる。

キーワード バッファオーバーフロー、バイナリ書換え、x86、スタックオーバーフロー検出

A Method for Buffer Overflow Detection by Binary Rewriting

Takaaki Yamakawa† Yoshihisa Mano‡
† Graduate School of Mathematical Sciences and Information Engineering, Nanzan University
‡ Faculty of Mathematical Sciences and Information Engineering, Nanzan University
27 Seireicho, Seto-shi, Aichi, 489-0863 Japan
E-mail: †m06mm034@nanzan-u.ac.jp, ‡ymano@nanzan-u.ac.jp

Abstract Buffer overflow vulnerabilities are abused in dangerous attacks, and new vulnerabilities are still found. In this paper, we describe a method for detecting and preventing some of buffer overflow attacks, which uses rewriting of binary executables. Entry and exit parts of each function are modified to save and check the return address. In order to escape limitations of modifying binary executables, we use the calling conventions and code patterns generated by compilers. Windows environments on x86 systems are assumed in this discussion. Some experiments and the results are also described.

Keyword Buffer Overflow, Binary Rewriting, x86, Stack Overflow Detection

1 はじめに

バッファオーバーフローはアプリケーションソフトウェアにおける代表的な脆弱性の一つであるが、現在でも数多くのバッファオーバーフロー脆弱性が発見され攻撃に利用されている。近年はセキュリティ意識の向上に伴い、脆弱性の発見、パッチの配布が迅速に行われるようになってきた。しかし攻撃者側の技術も向上しており、脆弱性の修正が行われる前に攻撃が行われるゼロデイアタックが問題となっている。このため、パッチの配布による既知脆弱性対策のみでなく、未知脆弱性にも対応可能な対策手法が必要となる。

本研究ではバッファオーバーフローの中でも特に危険度の高いスタックオーバーフローを検出すべく、スタック上の戻り番地の保護を行う。従来の対策手法はソースコードを利用するものが多くソフトウェア開発者向けであるが、本研究では対象をバイナリプログラムとすることでソースコードにアクセスできないソフトウェア利用者でも適用可能にする。バイナリプログラムはアーキテクチャに依存するため、本稿ここでは対象を x86 アーキテクチャとする。また、実行時にスタックオーバーフローの検出を行うため、速度低下などのコスト増大が懸念される。このため、コストを低く抑えることも重要である。

2 バッファオーバーフロー

2.1 バッファオーバーフローの概要

バッファオーバーフローとは、確保したバッファサイズ以上のデータが入力され、データがあふれることである。あふれたデータはバッファ以外の領域のデータを上書きしてしまうため、多くの場合プログラムは暴走する。しかし、入力データを巧妙に細工することでバッファオーバーフロー発生時に任意のコードを実行させることができる。このため、バッファオーバーフローは非常に危険な脆弱性である。

バッファオーバーフロー脆弱性はこれまでに数多く発見され、実際の攻撃に使用されている。**図 1** に NVD(National Vulnerability Database) [*1]による 2000 年から 2006 年までのバッファオーバーフロー脆弱性報告件数の推移を示す。概ね、近年に向けて増加傾向にあると言えよう。

[*1] 国立標準技術研究所 (National Institute of Standards and Technology, NIST) が構築する脆弱性データベース

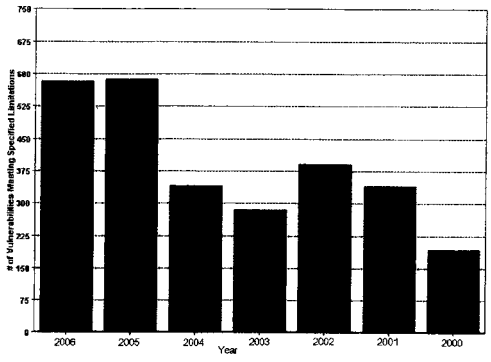

図 1　バッファオーバーフロー脆弱性報告件数の推移 [1]

2.2　バッファオーバーフローの例

バッファオーバーフローには幾つかの種類が存在するが、ここでは基本的なスタックオーバーフローの例を挙げる。**図 2** にスタックオーバーフロー発生前後のスタックの状態を示す。図の左部がスタックオーバーフロー発生前、右部が発生後のスタックである。

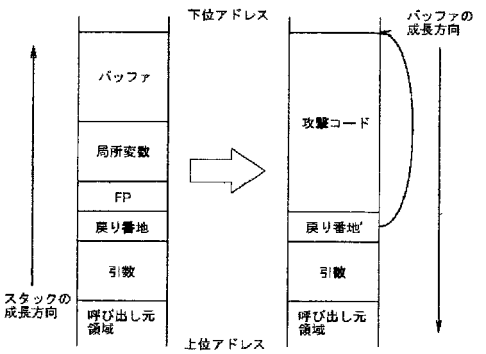

図 2　スタックオーバーフロー発生前後のスタックの状態

図 2 左部に示すように、スタックには引数、戻り番地、フレームポインタ (FP)、局所変数などが保持される。スタックの成長方向は上位アドレスから下位アドレス方向であるが、バッファは下位アドレスから上位アドレスに成長する。このため、確保したバッファサイズ以上のデータをバッファに書き込む場合、局所変数、フレームポインタ、戻り番地などが上書きされる。攻撃者はバッファに攻撃コードを書き込み、戻り番地を攻撃コードの先頭アドレスで上書きする。戻り番地が改ざんされているため、関数終了後には呼び出し元関数ではなく、攻撃コードに制御が移る。

3　関連研究

これまでに多くのバッファオーバーフロー対策手法が提案されてきた。その中の幾つかは広く用いられている。本章では、主なバッファオーバーフロー対策手法を挙げる。

3.1　ハードウェアを利用する手法

現在販売されている多くの CPU には NX ビット (No eXecute bit) という機能が搭載されている。これは、メモリをスタックなどのデータ領域とコード領域に分離し、データ領域でのコード実行を制限する機能である。この機能が有効であればスタック上の攻撃コードを実行することはできないため、バッファオーバーフローが発生した場合でも攻撃は不発に終わる。しかし、より巧妙な return-into-libc 攻撃などには有効でない。

3.2　ライブラリによる手法

Libsafe[2] は、ライブラリを置き換えることにより、脆弱性のある関数の代わりに安全な関数を実行させる手法である。ライブラリのみを置き換えるため、プログラムの再コンパイルを必要としない。Libsafe により防ぐことができる攻撃は置き換えを行った標準関数に起因するバッファオーバーフローのみである。また、標準関数を静的にリンクしたプログラムには有効でない。

3.3　コンパイラを用いた手法

コンパイル時にバッファオーバーフロー対策機能の付与を行う。ソースコードの再コンパイルが必要となる。

StackShield[3] は、関数処理の開始時に戻り番地を安全な場所に複製し、終了直前に複製された戻り番地をスタック上に書き戻す。これにより、スタックオーバーフローが発生して戻り番地が改ざんされた場合でも再度正当な値に上書きされるため、攻撃コードが実行されることはない。

StackGuard[5] は、スタック上の戻り番地の前にカナリアと呼ばれる値を置き、関数の終了時にカナリアの値が変化していないことを確認する。戻り番地を改ざんするにはカナリアも上書きする必要があるため、スタックオーバーフローの検出が可能となる。

3.4　バイナリプログラムに対する手法

コンパイラを用いた手法と異なり、バイナリプログラムを直接書き換えるため、ソースコードを必要としないという利点がある。反面、複雑な改変を行うことは困難である。

Prasad らは Windows のバイナリプログラムを対象として、戻り番地を保護する手法を提案している [6]。StackShield と同様に、関数処理の前後で戻り番地の改ざんを検出することでスタックオーバーフローの検出を行う。

Nebenzahl らも Prasad らと同様の戻り番地改ざん検出手法を提案している [7]。DLL、マルチスレッドプログラムでも使用可能な設計となっている。

4 提案手法

4.1 スタックオーバーフロー防止のアプローチ

スタックオーバーフローを防止する代表的な手法の一つとして、戻り番地の保護が挙げられる。これは、関数処理の前後で戻り番地あるいは周辺の値の改ざんを検出することで実現される。コンパイラを用いた手法では、コンパイル時のコード追加が容易であるため、多くの手法が提案されている。これに対しバイナリプログラムに対する手法ではコードの追加が困難であるため、各関数の先頭及び終端の命令列を分岐命令で置き換え、分岐先で戻り番地の保護を行う。ただし、命令列を置き換える際、対象命令列中に分岐先命令が存在する場合には置き換えを行うことができない。このため、1バイトの命令で置き換えるなどの対策が必要となる。

4.2 従来手法の制限

Prasad らはローカル変数を使用する関数のみに着目し、定型的なプロローグコード及びエピローグコードを置き換え対象としている。エピローグコードに関しては分岐命令よりもサイズが小さい場合がある。これは、エピローグコードの前の命令も含めて置き換えることで解決可能な場合もあるが、分岐先命令が含まれている場合は置き換えが困難である。

通常の分岐命令による置き換えができない場合には、INT 3 命令を用いて置き換えを行う。INT 3 命令はデバッガへのブレークポイントとして用いられ、命令サイズが 1 バイトであるため、任意の命令をブレークポイントで置き換えることが可能である。INT 3 命令を用いる欠点としては、制約が強くパフォーマンスの低下を招く点が挙げられる。

Nebenzahl らは関数をコピーし、コピーした関数の先頭と終端にコードを追加する方法を用いている。この方法では関数内のアドレスがずれるため、間接参照を用いた分岐命令が存在する場合はこの方法を用いることはできない。

4.3 本研究でのアプローチ

Prasad らの手法ではエピローグコードの置き換えに難があった。そこで本手法では、RETURN 命令の戻り先を意図的に変更することで、関数終了後に追加処理を実行させる。

RETURN 命令はスタックトップの値に制御を移すため、RETURN 命令の直前の命令を PUSH 命令に置き換えることでプッシュしたアドレスに制御を移すことが可能である。この時、PUSH 命令のオペランドにレジスタを使用することで PUSH 命令のサイズを 1 バイトに抑えることができる。

図 3 に本手法適用前後の関数のコード列を示す。図の左部が適用前の関数、右部が適用後の関数及び追加するプロローグ関数、エピローグ関数[*2]である。プロローグ関数中でレジスタにエピローグ関数のアドレスを格納しておくことで、適用後の関数の RETURN 命令は本来の戻り番地ではなくエピローグ関数に制御を移し、エピローグ関数の RETURN 命令から本来の戻り番地に制御が移る。

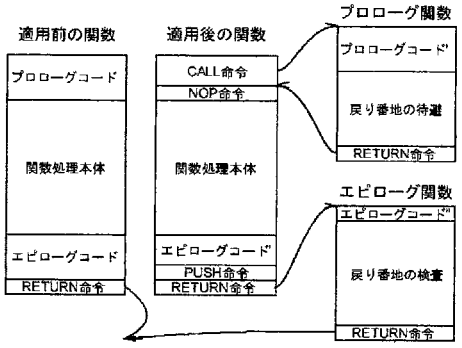

図 3 本手法適用前後の関数のコード列

提案手法を適用する上で解決すべき課題を以下にまとめる。

- 定型的なプロローグコード及びエピローグコードの調査
- レジスタの確保と安全性
- スタックの整合性

まず、Windows で主に用いられるコンパイラの出力コードを調べ、定型的なプロローグコード及びエピローグコードを列挙する必要がある。調査結果については、5.1.1 節で述べる。さらに、本手法ではプッシュにレジスタを使用するため、レジスタの確保に関する考察が必要となる。詳細は 4.4 節で述べる。また、プロローグ関数、エピローグ関数を呼び出す際にスタックの状態がずれるため、調整を加える必要がある。詳細は 4.5 節で述べる。

4.4 レジスタの確保と安全性

x86 系の CPU は 8 個の汎用レジスタ[*3]を保有しており、その内 2 個はスタック操作に用いられるため、6 個のレジスタをアキュムレータとして用いることが可能である。関数中でこれらのレジスタを使用する場合、自由に使用可能 (元の値を壊して良い) なレジスタと、制限付きで使用可能なレジスタがある [4]。前者は呼び出し元関数が待避を行う (caller save)。後者を使用する場合には呼び出された関数が元の値を待避し (callee save)、呼び出し元に戻る前に復元する必要がある。

本手法では、RETURN 命令の直前でエピローグ関数のアドレスをプッシュするために、レジスタを一つ使用する。関数中で使用しないレジスタが存在

[*2] CALL 命令による呼び出しでないため、エピローグ関数は変則的な関数となる

[*3] eax、ebx、ecx、edx、esi、edi、ebp、及び esp この内、ebx、ebp、esi、edi が callee save である

する場合は占有しても問題ないが、全てのレジスタを使用する可能性もあるため、対策が必要である。以下に、レジスタ ebx を使用する場合のプロローグコード、エピローグコードの例を示す。

```
push ebp        ; p1 呼び出し元関数の FP を待避
mov  ebp, esp   ; p2 フレームポインタの設定
sub  esp, +08   ; p3 局所変数の確保
push ebx        ; p4 ebx の待避
【関数処理本体】
pop  ebx        ; e1 ebx の復元
mov  esp, ebp   ; e2 局所変数の解放
pop  ebp        ; e3 呼び出し元関数の FP を復元
ret             ; e4 リターン
```

上記の例では、p1-p3 をプロローグ関数への CALL 命令に置き換え、プロローグ関数で ebx にエピローグ関数のアドレスを格納する。この値は p4 で待避された後、e1 で復元されるため、e3 を push ebx に置き換えることでエピローグ関数に制御を移すことが可能である。また、戻り番地の待避と同時に ebx も待避し、検査時に復元することで callee save の規約も守られる。使用するレジスタは ebx でなくとも良いが、他関数によるレジスタ破壊を防ぐため、callee save レジスタを用いる。

エピローグ関数のアドレスは p4 でスタックにプッシュされるため、この値を変更することで任意のコードが実行可能となる。しかし、大半のコンパイラは局所変数の確保後にレジスタの待避を行うコードを出力する (5.1.2 節参照) ため、エピローグ関数のアドレスがバッファ溢れによって汚染されることはない。レジスタの待避後に局所変数の確保を行うコードを出力するコンパイラも存在するが、この場合エピローグコードのサイズが 5 バイト以上となるため、通常の CALL 命令を用いて置き換えが可能である。

4.5 スタックの整合性

関数の呼び出し直後から局所変数確保までのプロローグコードにおけるスタック状態の変遷を**図 4** に示す。(a) が通常の局所変数確保処理、(b) が本手法適用後のスタック状態の変遷である。

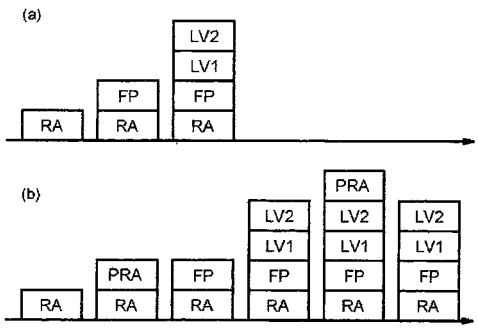

図 4　関数開始時のスタック状態の変遷

関数呼び出しによって戻り番地 (RA) がスタックに積まれる。その後、通常のプロローグコードでは以下の処理を行う。ここでは、スタック操作処理のみを列挙している。

1. フレームポインタ (FP) のプッシュ
2. 局所変数 (LV1、LV2) の確保

本手法適用後の処理の流れを以下に示す。2-5 がプロローグ関数内での処理である。プロローグ関数に渡される戻り番地を PRA とする。

1. プロローグ関数のコール (PRA のプッシュ)
2. PRA のポップ、フレームポインタのプッシュ
3. 局所変数の確保
4. PRA をプッシュ
5. リターン (PRA のポップ)

図 5 にエピローグコードにおけるスタック状態の変遷を示す。(a) が通常の局所変数確保処理、(b) が本手法適用後のスタック状態の変遷である。

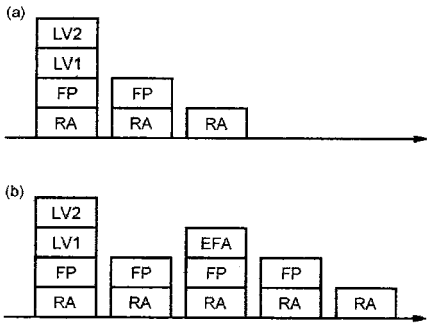

図 5　関数終了時のスタック状態の変遷

通常のエピローグコード処理の流れを以下に示す。

1. 局所変数の解放
2. フレームポインタをポップ
3. リターン (戻り番地のポップ)

本手法適用後の処理の流れを以下に示す。4-5 がエピローグ関数内での処理である。

1. 局所変数の解放
2. エピローグ関数のアドレス (EFA) をプッシュ
3. リターン (EFA のポップ)
4. フレームポインタをポップ
5. リターン (戻り番地のポップ)

双方とも、スタックの初期状態及び終了時の状態が一致するため、プログラム本来の動作に影響を与えることなく追加コードを実行させることが可能である。

4.6 バイナリプログラム書換え

以下にバイナリプログラム書換えの処理の流れを示す。

1. バイナリプログラムの逆アセンブルを行う。
2. アセンブリコードからCALL命令、RETURN命令をすべて探し出す。
3. バイナリプログラムの終端に新しいセクションを追加する。
4. 追加したコードセクションにプロローグ関数、エピローグ関数を配置する。
5. 各関数のプロローグコードをプロローグ関数へのCALL命令に置き換える。
6. RETURN命令の直前の命令をPUSH命令に置き換える。
7. PEヘッダの修正を行う。

まず、バイナリプログラム中の関数位置を把握するために逆アセンブルを行い、CALL命令、RETURN命令をすべて探し出す必要がある。バイナリプログラムの既存のコードセクションに新関数を追加した場合、コードセクション以降のセクションに影響を与えるため、新しいコードセクションを作成して新関数を追加する。局所変数を使用しない関数ではスタックオーバーフローは発生しないため、命令の置き換えは局所変数を使用する関数にのみ行う。各関数ごとに4-6の処理を行い、最後にPEヘッダ中のイメージサイズ、セクション情報などの修正を行う。以下に命令置き換え前後のプロローグコード、エピローグコード、プロローグ関数、エピローグ関数の例を示す。

```
push  ebp                    call PROLOGUE_F
mov   ebp, esp      ⇒        nop
sub   esp, +08
【関数処理本体】
mov   esp, ebp               mov   esp, ebp
pop   ebp           ⇒        push  ebx
ret                          ret

PROLOGUE_F:
pop   ecx           ; PRAをecxに待避
push  ebp           ; フレームポインタの待避
mov   ebp, esp      ; フレームポインタの設定
sub   esp, +08      ; 局所変数の確保
push  ecx           ; PRAを復元
【ebxの待避】
mov   ebx, EPILOGUE_F ; EFAをebxに格納
【戻り番地の待避】
ret                 ; リターン

EPILOGUE_F:
【ebxの復元】
pop ebp             ; フレームポインタの復元
【戻り番地の検査】
ret                 ; リターン
```

プロローグコードは通常、PUSH命令、MOV命令、SUB命令の3命令から構成されている。この命令列のサイズは6-9バイトであるため、5バイトのCALL命令及び幾つかの1バイトNOP命令で置き換えを行う。エピローグコードには幾つかの種類が存在するが、どの場合もRETURN命令の直前の命令は1バイト命令である。この例ではPOP命令をPUSH命令に置き換える。

プロローグ関数では、局所変数確保処理、レジスタの待避、エピローグ関数のアドレスをレジスタに格納、戻り番地の待避を行う。また、プロローグ関数の呼び出しによって戻り番地(PRA)がスタックに積まれるため、局所変数確保処理中はレジスタに待避しておく必要がある。エピローグ関数では、レジスタの復元、局所変数解放処理の一部、戻り番地の検査を行う。

5 実験及び評価

5.1 コンパイラの調査

Windows環境で主に用いられるコンパイラにおいて以下の調査を行った。

1. 定型的なプロローグコード及びエピローグコードの調査
2. レジスタ待避と局所変数確保の順序に関する調査

調査対象としたコンパイラを以下に挙げる。gcc(GNU Compiler Collection)のWindows版は複数存在するため、Cygwin版、MinGW版、djgppについて調査を行った。

1. Microsoft Visual C++ 2005
2. Intel C++ Compiler 10.0
3. Borland C++ Compiler 2007
4. GNU Compiler Collection 3.3.1 (Cygwin)
5. GNU Compiler Collection 3.4.2 (MinGW)
6. GNU Compiler Collection 2.7.2 (djgpp)

5.1.1 調査1

対象コンパイラが出力するコードを分析した結果、プロローグコードは1種類、エピローグコードは3種類のコードが主に使用されていることが分かった。以下に定型プロローグコードP1、定型エピローグコードE1-E3を示す。

```
; P1 6-9B                   ; E2 4B
push ebp      ; 1B          mov esp, ebp ; 2B
mov ebp, esp  ; 2B          pop ebp      ; 1B
sub esp, x    ; 3-6B        ret          ; 1B

; E1 5-8B                   ; E3 2B
add esp, x    ; 3-6B        leave        ; 1B
pop ebp       ; 1B          ret          ; 1B
ret           ; 1B
```

エピローグコードでは、E2 の前半 2 命令をまとめて 1 命令にした LEAVE 命令が存在するため、最小で 2 バイトとなる。E1 はコンパイラ 5 のみが出力し、他のコンパイラは E2 または E3 を使用していた。

5.1.2 調査 2

今回調査を行ったコンパイラの中では、コンパイラ 5 のみが局所変数確保前にレジスタの待避を行い、他のコンパイラは局所変数確保後にレジスタの待避を行っている。また、Linux 用の gcc でも同様の調査を行ったところ、コンパイラ 5 以外と同様の結果が得られた。このため、MinGW 版の gcc が特殊であると思われる。

5.2 空間的コストの試算

本手法では、対象プログラムにプロローグ関数とエピローグ関数を追加する。プロローグ関数は 36 バイト、エピローグ関数は 34 バイトで、さらに置き換えを行う関数 1 個あたり 7 バイトが必要となる[*4]。追加コードは非常に小さいが、追加はセクション単位で行うため、アライメントの影響を受ける。追加するセクションはデータセクションとコードセクションの 2 セクションであるためアライメントが 1KB であれば、空間的コストは 2KB となる。100KB を超えるプログラムであれば空間的コストは 2% 以下になるため、実用上問題ないと思われる。

5.3 時間的コストの実験

本手法を適用した際の時間的コストを評価すべく実験プログラムを実装し、実験を行った。実験環境は Core2 Duo 2.4GHz、2GB RAM 上の Windows XP SP2 である。

対象プログラムとして、ループ中からプロローグコードとエピローグコードのみのブランク関数を呼び出すプログラムを作成し、本手法適用前後の実行時間変化を計測した。実験にはループ回数が 1 億回、1000 万回、100 万回のプログラム a、b、c を使用した。また、プログラム b、c では総ループ回数を揃えるために、ブランク関数中でそれぞれ 10 回、100 回のループを実行している。

表 1 に本手法適用前後の実行時間変化を示す。表の各列は左から順に、対象プログラム、適用前の実行時間 (ミリ秒)、適用後の実行時間 (ミリ秒)、増加割合 (%)、関数実行 100 万回ごとの増加量 (ミリ秒) を示す。

表 1　本手法適用前後の実行時間変化

	適用前	適用後	割合	増加量
a	281	2672	851%	24
b	326	455	39.6%	13
c	309	326	5.5%	17

[*4] 空間的コストを抑えるために共通部分を抜き出し、プロローグコードのみを各関数ごとに用意している

100 万回の関数実行にかかる時間的コストは 20 ミリ秒程度である。短い関数では戻り番地保護処理の比重が増えるためコストがかさむが、ある程度の処理を行う関数であれば時間的コストは数 % 程度に抑えられるため、許容範囲内であると言えよう。

5.4 評価

INT3 命令を用いずにエピローグコードの置き換えを行うことができるため、時間的コストは Prasad らの手法に勝る。また、各関数に対して命令列の置き換えのみを行うため、間接分岐命令を含む関数の保護も可能である。

6 おわりに

本稿では、スタックオーバーフローの検出を行う手法としてバイナリプログラムの書換えによって、戻り番地の改ざんを検出する手法を提案した。呼出し規約及びコンパイラ出力コードパターン情報を利用し、RETURN 命令の戻り先を変更することで、従来手法に比べて安全性及び時間的コストの面で改善が見られた。

今後は、コンパイラ出力コードのより詳細な調査を行い、特殊なプロローグコード、エピローグコードについても対応できるようにする。また、より現実的なプログラムを用いてコストの評価実験を行う。

参考文献

[1] NIST: National Vulnerability Database, `http://nvd.nist.gov/statistics.cfm`.

[2] Avaya Labs Research: Libsafe, `http://www.research.avayalabs.com/`.

[3] Vendicator: Stack Shield: A "stack smashing" technique protection tool for Linux, `http://angelfire.com/sk/stackshield/`.

[4] Agner Fog: "Calling conventions for different C++ compilers and operating systems", `http://www.agner.org/optimize/calling_conventions.pdf`.

[5] C. Cowan, C. Pu, et al.: "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", 7th USENIX Security Conference, pp.63-78 (Jan. 1998).

[6] M. Prasad and T. Chiueh: "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks", USENIX Annual Technical Conference (June 2003).

[7] D. Nebenzahl, et al.: "Install-time Vaccination of Windows Executables to Defend Against Stack Smashing Attacks", IEEE Trans. Dependable and Secure Computing, Vol.3 No.1, pp.78-90 (2006).