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

M2006MM034 山川 高明
指導教員　真野 芳久

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。**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。

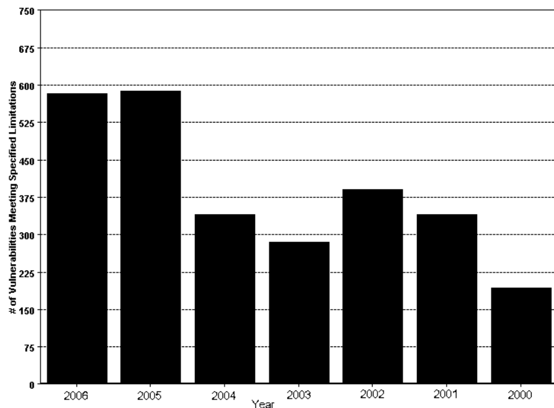

図 1　バッファオーバーフロー脆弱性報告件数の推移 [1]

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、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。

*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

2　関連研究

本章では、過去に提案された戻り番地の一貫性検査によるスタックオーバーフロー検出手法を取り上げる。

戻り番地の一貫性検査は関数の戻り番地に着目し、関数の開始直後と終了直前の戻り番地が変化していないことを検査することでスタックオーバーフローの検出を行う手法である。スタックオーバーフロー自体を防ぐことはできないが、攻撃コードが実行される直前に検出できるため、攻撃を未然に防ぐことが可能である。戻り番地の一貫性検査は、検査対象によって戻り番地の直接検査法とカナリア法の 2 種類に大別できる。

戻り番地の直接検査法は、関数の開始直後に戻り番地を安全な場所に待避し、終了直前にスタック上の戻り番地と待避された戻り番地の比較を行うことで戻り番地の改ざんを検出する。

カナリア法は、戻り番地を直接検査する代わりに、バッファと戻り番地の間にカナリアとよばれる値を置き、カナリア値の変化によってスタックオーバーフローの検出を行う。

2.1　コンパイラを用いた手法

コンパイル時に戻り番地の一貫性検査を行うコードを追加する。ソースコードが必要となるが、自由にコードを生成できるという利点がある。

StackShield[2] は、戻り番地の直接検査法に近いが、戻り番地の検査は行わず、関数の終了直前に待避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。

Cowan[3] はカナリア法の提案を行い、gcc に対するパッチの形で StackGuard を実装した。カナリアはフレームポインタと戻り番地の間に 1 ワードの値として置かれる。

2.2　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換えるため、ソースコードを必要としない。反面、複雑な改変を行うことは困難である。検査コード追加の詳細は 3.1 節で述べる。

Prasad らは Windows のバイナリプログラムを対象として、戻り番地の直接検査法を実現している [4]。

Nebenzahl らも Prasad らと同様に戻り番地の直接検査法を実現している [5]。また、TLS(Thread Local Storage) を利用することで 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。

3 提案手法

3.1 バイナリプログラムにおける戻り番地の一貫性検査

バイナリプログラムを対象として戻り番地の一貫性検査を行う場合は、コードの単純挿入が困難で、基本的なスタック構造を変更することができないため、コンパイラを用いた手法に比べて制約が強い。このため従来手法では、命令列置き換え法、関数コピー法などが用いられている。

命令列置き換え法では、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一貫性検査を行う。CISC アーキテクチャでは、命令ごとに命令長が異なるため、命令列を置き換える際、置き換え対象命令列中に他の命令からの分岐先となる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場合がある。

関数コピー法は、各関数を別の場所にコピーし、コピーする際に関数の先頭及び終端にコードの追加を行う。また、本来の関数の先頭部分にはコピー先関数への分岐命令を配置する。この手法では、関数の先頭アドレスは変動しないため関数呼び出しに影響を与えることはない。しかし、関数内のアドレスはずれるため、関数中に間接参照を用いた分岐命令が存在する場合は適用不能である。

3.2 従来手法の制限

Prasad ら [4] は、命令列置き換え法を用いている。また、局所変数を使用する関数のみに着目し、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く置き換えが困難な場合がある。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 [5] は関数コピー法を用いている。この手法では前述した関数内のアドレスがずれる問題が存在するため、間接参照を用いた分岐命令が存在する関数には適用を行わない。

両手法とも、すべての戻り番地の一貫性検査を行うわけではないので保護されていない戻り番地が存在する。このため、保護されていない戻り番地を狙った攻撃が行われる可能性がある。一般的に攻撃者は脆弱性を持つバッファと同一のスタックフレーム中の戻り番地を改ざんして攻撃を行う。しかし、戻り番地の一貫性検査が行われている場合には戻り番地の改ざんが検出されるため、攻撃は不発に終わる。この時、戻り番地を同じ値で上書きすることで、改ざんの検出を回避し、スタックフレームを跨いで保護されていない戻り番地を攻撃することが可能となる。

3.3 提案手法の概要

本手法はバイナリプログラムにスタックオーバーフロー検出機能を付加することを目的とする。そのために、バイナリプログラム中の実行コードを書換え、戻り番地の一貫性検査を行う。対象は広く用いられており攻撃の対象になりやすい x86 システム上の Windows 環境とする。

対象プログラムには、戻り番地の待避などを行うプロローグ関数及び、改ざんの検出などを行うエピローグ関数の追加を行う。そして、局所変数を使用する関数の開始直後にプロローグ関数を、終了時にエピローグ関数を呼び出すことで戻り番地の一貫性検査を行う。**図 2** に本手法適用前後の関数のコード列と処理の流れを示す。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。プロローグ関数では、戻り番地の待避を行い、エピローグ関数では改ざんの検出を行う。

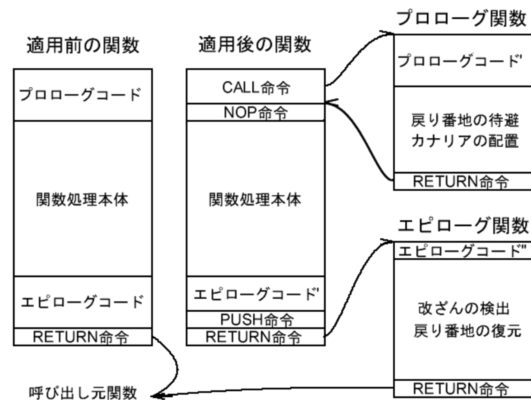

図 2　本手法適用前後の関数のコード列

バイナリプログラムには関数情報が残されていないため、逆アセンブルを行って関数情報の復元を行う。アセンブリコード上で関数の開始地点と終了地点を探索し、対応するバイナリプログラム上のバイナリコードを必要に応じて書換える。書換え対象は、局所変数をスタック上に確保する関数の先頭命令列及び、RETURN 命令の直前の命令である。局所変数を用いない関数あるいは、局所変数をスタック上に確保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、書換え対象としない。対象関数の先頭命令列はプロローグ関数の CALL 命令に置き換えを行う。RETURN 命令の直前の命令の置き換えについては、3.3.1 節で述べる。

上記の書換えによって対象関数の実行前にプロローグ関数が実行され、対象関数の終了後にエピローグ関数が実行される。エピローグ関数中で行う改ざんの検出法については 3.3.2 節で述べる。

*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

3.3.1　追加関数への制御の移動

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時にエピローグ関数に制御を移動させる。

RETURN 命令は呼び出し元関数に戻る命令であるが、実際の動作はスタックトップのデータを戻り番地と認識して取り出し、そのアドレスにプログラム制御を移す。スタックオーバーフロー攻撃が行われるとスタック上の戻り番地を改ざんされるため、RETURN 命令は改ざんされたアドレスにプログラム制御を移すことになる。このため、スタックオーバーフロー攻撃を検出するには、戻り番地が使用される前に戻り番地の改ざんを検出する必要がある。

本手法では、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への制御の移動を実現する。プッシュする値をエピローグ関数のアドレスにすることで、戻り番地の改ざんの有無に関わらずエピローグ関数に制御を移し、改ざんの検出を行うことが可能である。エピローグ関数のアドレスは予めプロローグ関数中でレジスタに格納しておく。オペランドにレジスタをとる PUSH 命令のサイズは 1 バイトであるため、任意の命令と置き換えてエピローグ関数への制御の移動を実現することが可能である。使用するレジスタの確保に関しては、3.4 節で述べる。

3.3.2　耐性の強化

従来手法ではスタックフレームを跨いだ攻撃が可能であった。そこで本手法では、カナリア法を導入する事でスタックフレームを跨いだ攻撃を防ぐ。

バイナリプログラム上でカナリアを置く領域を新たに追加するのは困難であるため、カナリア法は通常、コンパイラを用いた手法で実現される。本手法では、戻り番地を待避し、代わりにカナリアを置くことで、スタックフレームに新領域を追加することなくカナリア法を実現する。戻り番地は関数呼び出し時にスタックに積まれた後、関数終了時まで使用されることはない。このため、プロローグ関数中で戻り番地の待避を行った後、戻り番地の代わりにカナリアを置き、エピローグ関数中でカナリアの改ざんを検査することでスタックオーバーフローの検出を行うことができる。また、カナリアの値を攻撃者が予測するのは困難であるため、カナリア値を同じ値で上書きして保護されていない戻り番地を攻撃することは困難になる。

カナリア法には、乱数値を用いる無作為カナリア法と、固定値を用いる終端カナリア法の 2 種類がある。終端カナリア法は共有ライブラリの保護に使用できるという利点があるが、本手法ではプログラム単体のみに適用するため無作為カナリア法を用いる。無作為カナリア法ではカナリア値として乱数を用いるため、攻撃者は予測が困難である。

3.4　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、元の値を壊して良いレジスタと、元の値を保持しなければならないレジスタに分けられる [6]。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。このため、レジスタの確保に関する考察が必要である。**図 3** 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 呼び出し元の FP を復元
ret             ; e4 リターン
```

図 3　プロローグコード及び、エピローグコードの例

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できる。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アドレスに制御を移すことが可能となる。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ため、同一スタックフレーム中でのスタックオーバーフローによって改ざんされることはない。また、カナリア法を用いることでスタックフレームを跨いだ攻撃を検出することができるため、異なるスタックフレーム中のエピローグ関数のアドレスも改ざんされることはない。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、この場合はエピローグコードのサイズが増加するため、通常の分岐命令を用いて置き換えが可能である。

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

4　評価実験

本手法の有効性を評価すべく、コストの試算及び実験を行う。実験環境は Core2 Duo 3GHz、4GB RAM 上の Windows XP SP2 である。実験は以下の 2 項目に関して行う。

1. 本手法適用前後のファイルサイズ変化
2. 本手法適用前後の実行時間変化

4.1　空間的コストの試算及び実験

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。プロローグ関数及びエピローグ関数はそれぞれ 48 バイトで、さらに置き換えを行う関数 1 個あたり 33 バイトと置き換える命令列のサイズ分 (合計約 40 バイト) が必要となる[*4]。同一命令列の集約も行っているため、実際の空間的コストはこれより低い値になる。また、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。アライメントが 4KB であれば、空間的コストは最低でも 8KB となる。

55 個のプログラムを対象として、空間的コストの実験を行った。**表 1** に実験結果を示す。表の各列は左から順に、対象、対象の数、適用前のファイルサイズ (B)、適用後のファイルサイズ (B)、増加割合 (%) を表す。結果は、ファイルサイズが 1MB 以上の大、100KB 以上 1MB 未満の中、100KB 未満の小及び、全体の平均に分類して示す。

表 1　本手法適用前後のファイルサイズ変化

対象	対象数	適用前	適用後	割合
大	14	2,377,660	2,395,465	0.7
中	29	435,472	443,586	1.9
小	12	17,001	17,478	2.8
平均	55	843,800	852,861	1.1

概ね、ファイルサイズの増大に伴って、増加割合は減少する傾向が窺える。

4.2　時間的コストの実験

関数呼び出し 1 回当たりの時間的コストの実験を行った。対象は、局所変数確保処理と解放処理のみから成るブランク関数を 1 億回呼び出すプログラムとした。実験を行った結果、100 万回の関数実行にかかる時間的コストは 20 ミリ秒程度であった。

また、現実に用いられているプログラムにも本手法を適用して時間的コストの実験を行ったが、適用前後での実行時間の変化は見られなかった。時間的コストが低いため、誤差に紛れて計測ができなかったものと思われる。

[*4] 空間的コストを抑えるために共通部分を抜き出している

4.3　評価

Prasad ら [4] の手法に比べて、INT 3 命令を使用しない分時間的コストの面で改善していると思われる。

Nebenzahl ら [5] の手法では関数を別の場所にコピーするため、空間的コストは 20% 程度となっている。本手法では、2% 程度であり、空間的コストの面で有利である。安全性の面でも、本手法は間接分岐命令を含む関数にも適用できるため有利と言える。

また、両手法ではスタックフレームを跨いだ攻撃が可能であったが、本手法ではカナリア値の変化によって検出できるため、安全性が向上している。

5　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を用いた戻り番地の改ざんを検出する手法の提案を行った。従来手法で可能だったスタックフレームを跨いだ攻撃を検出すべく、カナリア法を導入した。また、制御の移動方法を改善することで時間的コストの低減を図った。

本手法では、カナリア法を導入することでスタックフレームを跨いだ攻撃への耐性を確保したが、従来手法において、スタックフレームを跨いだ攻撃がどの程度有効であるかの検証は未だ不十分である。このため、脆弱性を持つプログラムに従来手法を適用した上でスタックフレームを跨いだ攻撃を行い、攻撃成功割合の調査を行うことが望ましい。

参考文献

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

[2] Vendicator: Stack Shield: A “Stack Smashing” Technique Protection Tool for Linux, `http://angelfire.com/sk/stackshield/`.

[3] Cowan, C.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *7th USENIX Security Symposium*, pp.63–78 (1998).

[4] Prasad, M.: A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks, *Proc. USENIX Annual Technical Conference* (2003).

[5] Nebenzahl, D. and Sagiv, M.: Install-Time Vaccination of Windows Executables to Defend against Stack Smashing Attacks, *IEEE Dependable and Secure Computing*, Vol.3, No.1, pp.78–90 (2006).

[6] Fog, A: Calling Conventions for Different C++ Compilers and Operating Systems, `http://www.agner.org/optimize/calling_conventions.pdf`.

[7] 山川高明, 真野芳久：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, 情報処理学会 2007-CSEC-39, pp.61–66 (2007).