社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－DPS－122（14）
2005－CSEC－28（14）
2005／3／22

能動的な制御フローの変更による異常検知手法の提案

鑪　講平 †　　田端　利宏 ‡　　櫻井　幸一 ‡

† 九州大学大学院システム情報科学府
〒 812-8581 福岡市東区箱崎 6 丁目 10 番 1 号
tatara@itslab.csce.kyushu-u.ac.jp

‡ 九州大学大学院システム情報科学研究院
{tabata，sakurai}@csce.kyushu-u.ac.jp

あらまし　バッファオーバフロー脆弱性を利用した計算機の不正利用を防ぐためには，プログラマの注意を喚起するだけでなく，コンパイラや OS 側での対策技術が重要である．一方で，計算機に関する知識が限られているユーザには，対策技術の導入や運用が容易でなければならない．本論文では，プログラムの制御フローを能動的に変更することにより，バッファオーバフローに基づく異常を検知する手法を提案する．提案手法は，上記の要件を満足しつつ誤検知が発生しないという特徴を持つ．

Active Modifier of Control Flow for Detecting Anomalous Program Behavior

Kohei Tatara†　　Toshihiro Tabata‡　　Kouichi Sakurai‡

†Graduate School of Information Science and Electrical Engineering, Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka, 812-8581 Japan
tatara@itslab.csce.kyushu-u.ac.jp

‡Faculty of Information Science and Electrical Engineering, Kyushu University, Japan
{tabata, sakurai}@csce.kyushu-u.ac.jp

Abstract In order to prevent malicious use of the computer using buffer overflow vulnerabilities, a corrective action by not only calling a programmer's attention but expansion of compiler or OS is important. On the other hand, introduction and employment of an intrusion detection system must be easy for the user by whom the knowledge about a computer is restricted. In this paper, we can detect an anomly program behavior by actively modifying some control flows of a program. Our method satisfies these requirements and gives no false positives.

1　はじめに

バッファオーバフロー脆弱性は，プログラムに内在するバグに起因する [1]．そのため，以下のような対策方法が考えられる．

プログラマ側での対策 プログラミングのフレームワークを設計する．フレームワークは，バッファオーバフロー脆弱性を内包しないコードを生成するためのプログラミングルールの集まりである．

コンパイラ側での対策 プログラムのコンパイル時に，パターンマッチングや構文解析を行う．実行コードが生成される前に，脆弱性を持つ関数や部位を特定することができる．

オペレーティングシステム（OS）側での対策 全てのアプリケーションプログラムは OS により管理される．OS 側でバッファオーバフローの発生を検知して，計算機の不正利用を防ぐ．これには，動的ライブラリの脆弱性を取り払うことも含まれる．

バッファオーバフロー脆弱性の発生は，プログラマの注意を喚起することによって防ぐことができる．

しかし，プログラマ側で脆弱性を見逃す恐れがある限り，ユーザ側で利用可能な対策技術の重要性は高い．コンパイラ側での対策技術には，コンパイラやOS，アプリケーションプログラムの再コンパイルが伴う場合がある．また，ユーザの脆弱性に対する知識を必要とするものもある．

一方で，OS側での対策技術では，システムに対する影響を最小限に留めることができる．そうした背景を踏まえた上で，本論文ではOS側での対策技術に焦点をあてる．一般に，計算機の不正利用を検知するシステムは侵入検知システムと呼ばれ，侵入行為のシグネチャに基づいて検知を行う不正検知と，正常な動作とは異なる振る舞いを検知する異常検知とに分類される．異常検知では未知の侵入行為を検知することができる一方，システムが複雑になり，オーバヘッドの増加を招く恐れがある．本論文で提案する手法は異常検知の部類に入り，下記のような特徴を持つ．

1. バッファオーバフローの発生を，ほぼリアルタイムで検知することができ，攻撃者が計算機を不正利用することを防ぐことができる．
2. バッファオーバフローの発生を検知するためのオーバヘッドを低く抑えることができる．これにより，システムのパフォーマンスを損なうことなく導入が容易となる．
3. アプリケーションプログラムの再コンパイルが必要ない．既存のシステムからの移行が用意であり，導入や運用に際して必要とするユーザの知識を抑えることができる．

実用的な対策技術として受け入れられるためには，バッファオーバフローの発生を検知し，かつ導入するための敷居を低くすることが肝要である．そのため1，2の要件を同時に満足することは重要である．また，プログラマとは違い，OSやプログラムに関するユーザの知識は限られている．検知システムの導入や運用にかかるコストはできるだけ少ない方が望ましいため，3の要件がさらなるユーザの利用を促進する．

本論文の構成は以下の通りである．2章で，バッファオーバフローが発生する背景と仕組みについて述べた後，3章で提案手法を説明する．4章で具体例を交えた実装方法について触れ，5章でセキュリティについて分析する．最後に，6章でまとめとする．

2　背景と関連研究

2.1　バッファオーバフローを利用した侵入行為

プログラムが実行されると，スタックと呼ばれる逐次入出力データを一時的に貯えるための記憶領域が確保される．スタックは，データの追加と取出しを一方の端だけで行うFirst In Last Out（FILO）形式のデータ構造を持ち，サブルーチンや関数を呼び出す際に，処理中のデータや戻りアドレスなどを一時的に退避する場合に使うことが多い．これらの値は，関数が呼び出されるとスタックに積まれ，関数から処理が戻るとスタックから破棄される．

一方，C，C++等のプログラミング言語では，ローカルバッファの確保と管理はプログラマに委ねられている．そのため，プログラマは，自らが決めたサイズのローカルバッファが適切に使用されるようにプログラミングを行う責任がある．ここで，予め確保したローカルバッファ（local buffer）のサイズを超えて値を書き込むことを許してしまうと，フレームポインタ（fp）やリターンアドレス（ret）の値が書き換えられてしまう．これをバッファオーバフローと呼ぶ．侵入行為の過程では，このバッファオーバフローを故意に引き起こして，リターンアドレスや関数ポインタの値をシェルコードやライブラリのアドレス（attack code）で置き換える．図1に，バッファオーバフローにより改ざんされたスタックの内部状態を示す．攻撃者は，シェルを起動するように制御フローを変更をすることで，任意のコマンドを実行することができる．本論文では，これを侵入行為と呼ぶ．

2.2　侵入行為を防止するための技術

侵入行為に対するOS側での取り組みを紹介する．OS側での対策技術は，コンパイラを用いる場合とは異なり，個々のアプリケーションプログラムをコンパイルし直す手間が必要ない．また，導入時にOSの再起動を要する場合，OSが提供するサービスを一時的に停止しなければならない．侵入検知システムをアドインツールとして提供するためには，カーネルの再コンパイルは必要でないことが望ましい．

Openwall [2] は，ユーザメモリ領域においてスタック内の実行コードを実行できないようにする機能を持つ．この機能によってシェルコードをスタックに

保存することを前提とした計算機の不正利用を防ぐことが可能である．近年，64 ビット CPU の中には，NX（Non-eXcution）機能を提供するものが現れた．この機能が有効にされた OS 上では，Openwall と同様にトロイの木馬や不正なコードの実行が不可能になると考えられる．しかし，スタックの非実行化機能では検出できない手法も報告されているため，侵入行為を完全に防ぐとは言えない [3]．

StackGuard [4] は，ローカルバッファの領域を超えて値が書き込まれたことを検知する．しかし，バッファオーバフローにより書き換えられた変数の値が正常であるかどうかの判断はできない．そのため，関数ポインタが書き換えられ，不正なコードが実行を検知できない恐れがある．

他にも，システムコールの発行履歴に基づく異常検知の研究がある．システムコールとは，OS の機能を利用するために提供されている関数群である．異常検知では，予め記録しておいたシステムコールの発行パターンに基づいて，プログラムが正常に動作しているかどうかをチェックする．しかし，プログラムは複数の分岐処理やループ処理を内包することがある．システムコールの発行と制御フローとの関連付けが非決定的であると，誤検知が発生する恐れがある．そこで，システムコールの発行履歴に，スタックの情報を加えることで，決定性を付加する研究もある [5]．提案手法では，検査対象を自ら作成して，プログラムの制御フローに挿入する．すなわち，能動的に決定性を付加する方式と同様に，誤検知は発生しない．

3 提案手法

3.1 制御フローの変更

プログラムがメモリに読み込まれる際に，制御フローを変更する．関数呼び出しの始まりと終わりにおいて，正当な呼び出しであるかどうかを検証するための処理を挿入する．これを検証関数と呼ぶ．提案手法を適用した関数は以下に示す通りに動作する．

1. 関数呼び出しの直後，スタックにフレームポインタの値が積まれる．この時点におけるスタックポインタの値は新しいフレームポインタの値として用いられる．

2. 次に，アドレスのサイズ分だけスタックを減じる．この領域には，関数呼び出しの最後に検証関数のアドレスが書き込まれる．その後，ローカルバッファが確保され，本来の関数の処理が開始される．

3. 関数の処理が終了すると，フレームポインタの値を用いて予め確保しておいた領域に検証関数のアドレスを書き込む．このため，関数呼び出しが終了する際にプロセスの制御は一時的に検証関数へと移る．

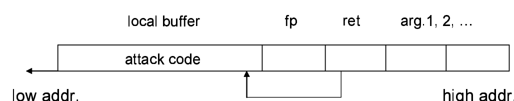

図 1: スタックの状態

3.2 関数呼び出しの変更

プログラム中に関数呼び出しが存在する場合は，下記のように変更を行う．他の関数を呼び出す場合，はじめにリターンアドレスをスタックに積んだ後，オペランドで指定したアドレスへと制御が移る．そのとき，スタックに積むリターンアドレスの値と，最後に乱数に基づいて選択された値 p との排他的論理和を計算する．この計算結果によって得られた値が，リターンアドレスとしてスタックに積まれる．

次に，関数呼び出しにおけるオペランドに対して変更を加える．このとき，オペランドで指定された値がレジスタやローカルバッファの値であると判断されるとき，この値に対して，入力を与える命令を遡って調べる．プログラムの実行中に変化しない一定の値を見つけたとき，その値と p との排他的論理和を計算して得られた値に置き換える．ここで，一定の値が発見できない場合としては，外部入力により呼び出される関数が異なるプログラムなどが考えられる．本論文では，このようなプログラムは異常検知の対象としない．

3.3 検証関数の処理

検証関数では，引数に乱数に基づいて選択された値と，前の関数へと戻るためのアドレス，フレームポインタの値を引数とする．ここでは，乱数に基づき選択された値の検証を行う．すなわち，この値が正常であるかどうかを調べることにより，バッファオーバフローを検知する．具体的には，オペランド

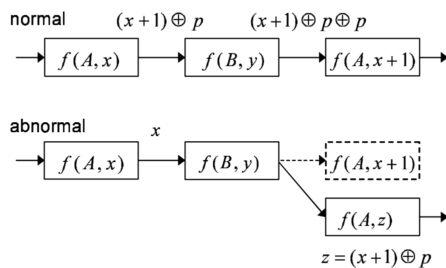

図 2: 関数呼び出しの流れ

に指定したアドレスと p との排他的論理和を取り，その計算結果が指すアドレスに制御を戻す．

3.4　異常検知の手順

提案方式が，侵入行為を検知する手順について説明を行う．図 2 は，関数 A から他の関数 B に処理が移り，再び関数 A の実行が再開する様子を示す．

$f(A, x)$ は，CPU が関数 A 内における x 番目の命令を実行することを表す．提案手法を適用することで，A が B を呼び出す際にスタックに積まれるリターンアドレスは p との排他的論理和により計算される．p は，乱数に基づく値である．

$f(A, x)$ から $f(B, y)$ に処理が移る際，リターンアドレスとして $x \oplus p$ を渡す．B から A に戻る際には，必ずリターンアドレスが書き換えられて検証関数が呼び出される．$f(A, x+1)$ から処理を再開する際には，リターンアドレスと p との排他的論理和を計算する．このように，検証関数を介することで制御フローの完全性を検証することができる．

ここで，攻撃者は関数 B におけるリターンアドレスや関数ポインタを，シェルコードもしくは共有ライブラリのアドレスに書き換えると仮定する．しかし，バッファオーバフローの事実如何にかかわらず，必ず検証関数が呼び出される．$f(A, x)$ から $f(B, y)$ に渡される値は，リターンアドレスと p との排他的論理和により計算される．すなわち，攻撃者は p を知らなければ，侵入行為は成功しない．

4　実装方法

4.1　プログラムの修正手順

本章では，提案手法の既存システムに対する適用可能性について論じる．具体的には，Intel 32-bit Architecture（IA-32）上で動く Linux OS（Kernel 2.4.19）における実装方法を述べる．コンパイラは GNU C Compiler（gcc-2.96）を用いる．

```
_func1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        incl    8(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, (%esp)
        call    _func2
        leave
        ret
```

図 3: 提案手法を適用する前のプログラム

図 3 は，提案手法を適用する前のプログラムを表す．このプログラムでは，関数 func1 の中で別の関数 func2 が呼び出される．このプログラムに提案手法を適用すると，図 4 のような制御フローが得られる．このプログラムの実行時における動作の様子を説明する．

1. 始めに，関数 func1 が呼び出されると，%ebp がスタックに積まれ，この時のスタックポインタの値%esp が%ebp として用いられる．ここで，検証関数 ver のためのアドレス領域として 4 バイトだけスタックを減じた後に，ローカルバッファが確保される．

2. 元の関数には，call 命令が存在するため，検証関数 ver に処理が移る際，スタックに積まれる値は，リターンアドレスと p との排他的論理和により計算されるように修正を行う．

3. 元の関数の処理が終わると，フレームポインタの値を用いてローカルバッファに書き込みを行う．この時，検証関数 ver のアドレスを書き込む．このため，関数呼び出しが終了する際にプロセスの制御が一時的に検証関数に移る．

4.2　提案手法の適用前後における一貫性の確保

制御フローを変更する上で最も重要なことは，元のプログラムに与える影響を可能な限り小さくすることである．また，プログラムが実行される度に変

化したり，ユーザの入力に影響を受けたりする値を用いることは，提案手法の適用を自動化する上で重要である．例えば，call 命令では，スタックに格納される値としてリターンアドレスと p（$= 0x12345678$）との排他的論理和を計算するように修正を行う．提案手法では，関数 func を呼び出す．関数 func の呼び出しが終了すると，検証関数 ver に処理が移る．検証関数 ver では，リターンアドレスとして，スタックに格納される値と p との排他的論理和を計算する．こうして，提案手法の適用前後において，プログラム動作の一貫性が保たれていることがわかる．

また，

```
        call _func2
```

という命令は，

```
        call _trampoline2
        ...
    _trampoline2:
        subl $0x12345678, (%esp)
        call _func2
        movl $_ver, 4(%esp)
        ret
        ...
```

と置き換えられる．“trampoline” 関数を挿むことにより，前後の命令に対する影響を考慮する必要がなくなる．図 3，4 より，提案手法の適用前後において関数 func のサイズに変化がないことがわかる．関数 func のサイズに変化が生じると，他の関数内で用いられるアドレスの値に変更を加える必要があるためである．

5 セキュリティ

5.1 侵入行為への耐性

リターンアドレスや関数ポインタを書き換えると，制御フローを変更することができる．しかし，攻撃者が利用可能なローカルバッファは限られている．そこで，本論文では，攻撃者は OS が提供する関数やライブラリを利用するという仮定を置く．ここで，攻撃者が OS の提供する関数やライブラリを利用しない場合，提案手法は何ら効を成さないことに注意する必要がある．すなわち，攻撃者の目的が，プログラムで用いる変数の値を書き換えるだけであった場合，提案手法では攻撃を検知することはできない．

```
_func1:
        call    _trampoline1
        nop
        incl    8(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, (%esp)
        call    _trampoline2
        nop
        ret
        ...
_trampoline1:
        popl    %eax
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        subl    $8, %esp
        jmp     (%eax)
        ...
_trampoline2:
        subl    $0x12345678, (%esp)
        call    _func2
        movl    $_ver, 4(%esp)
        ret
        ...
_ver:
        xorl    $0x12345678, (%esp)
        ret
        ...
```

図 4: 提案手法を適用した後のプログラム

提案手法を適用すると，リターンアドレスをスタックに積む際に値を修正する．このため，攻撃者がシェルコードやライブラリのアドレスでリターンアドレスを置き換えたとしても，これらの関数に制御が移ることはない．

一方で，関数ポインタの書き換える侵入行為は，関数の終了時におけるリターンアドレスの検証では防ぐことができない．また，関数ポインタは関数内で変化する可能性もあるため，正しい値が入力されたかどうかを判断することはできない．提案手法では，関数呼び出しにおけるオペランドを修正することで，この攻撃に対しても耐性を持つ．

5.2 侵入行為が成功する可能性

プログラムに対する修正は，メモリ上にロードされる際に行われる．その際，関数毎に乱数に基づいて p を選択して，スタックに積むリターンアドレスの値を修正する．すなわち，侵入行為を働くためには，正しい p の値を選択する必要がある．p は限りなく大きな値（2^{32} 程度）を想定しているため，攻

撃者が p を取得することができないとき，侵入行為が成功する可能性は限りなく低い．

6 まとめ

バッファオーバフローを利用した計算機の不正利用の事例は多数報告されており，深刻な問題である．今日，不正利用の事実を検知するため，ソフトウェアやハードウェアの面から様々な研究が行われている．殆どの OS で，NX 機能が標準化されれば，このような状態も改善されるかもしれないが，先に述べた通りあらゆる不正利用を防止する技術ではないため，侵入検知システムの重要性は高い．

本論文では，プログラムの制御フローを変更して，異常検知を行う手法について提案を行った．提案手法では，オーバヘッドを抑えつつ，誤検知が発生しないという利点がある．また，OS 側での導入と運用のコストを必要とするだけで，ユーザの利用を促す効果が期待できる．今後の課題としては，異常検知システムとしてのさらなる性能評価などがある．

参考文献

[1] Beyond-Security's SecuriTeam.com. Writing Buffer Overflow Exploits - a Tutorial for Beginners. `http://www.securiteam.com/securityreviews/5OP0B006UQ.html` (accessed 2003-09-05).

[2] Openwall Project, Linux kernel patch from the Openwall project, `http://www.openwall.com/linux/` (accessed 2004-01-20).

[3] Linus Torvalds, `http://old.lwn.net/1998/0806/a/linus-noexec.html` (accessed 2004-02-13)

[4] P. Wagle and C. Cowan. StackGuard: SimpleStack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, pp. 243–255, May 2003.

[5] M. Oka, H. Abe, Y. Oyama, K. Kato. Intrusion Detection System Based on Static Analysis and Dynamic Detection. In *Proceedings of Forum on Information Technology (FIT 2003)*, Sep. 2003.