推薦論文

propolice：スタックスマッシング攻撃検出手法の改良

江藤博明† 依田邦和†

本論文ではバッファオーバフローの検出について，従来から提案されていた手法の問題点を述べ，その改良手法を提案する．改良点は，(1) ローカル変数中に現れるポインタ変数がスタック破壊で操作されないように，スタック上の配置を変更すること，(2) 関数の引数に現れるポインタ変数がスタック破壊で操作されないように，ローカル変数に複製し，複製した変数を使用すること，(3) 検出機構のオーバヘッドを減らすため，ある条件の関数ではプロテクションコードの生成をやめること，の 3 点である．

propolice: Improved Stack-smashing Attack Detection

Hiroaki Etoh† and Kunikazu Yoda†

This paper presents several security problems of existing buffer overflow detection techniques and some novel ideas for improving the state of the art in buffer overflow detection. The main ideas are (1) the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to corrupt arbitrary memory locations further, (2) the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations, and the (3) omission of instrumentation code from some functions to decrease the performance overhead.

1. はじめに

バッファオーバフロー脆弱性問題は，1988 年 11 月のインターネットワーム[4] でおよそ 6000 のシステムがシャットダウンした事件を筆頭に，これまでさまざまなアプリケーションで問題となっている．

この問題は C 言語で実装されたアプリケーションで，設計者の意図した長さ以上の文字列入力が与えられた際に発生する．そのようなアプリケーションの多くは文字列演算の中間結果をスタック上に確保している．スタックスマッシング攻撃とはそのような文字配列領域に，確保された長さ以上の文字列を与えることで，領域から溢れさせ，溢れ出た文字列で文字配列の外にある領域を改竄することで，本来の動作とは異なる処理を行わせるものである．スタック上の改ざん対象で代表的なものは実行制御に関わるリターンアドレスや関数ポインタを保管している領域である．

本論文ではバッファオーバフロー問題の網羅的な解決手法を提案する．この手法では保護すべきアプリケーションをコンパイルするときに，バッファオーバーフローを検出する機構（実行コード）を挿入する．この機構は StackGuard[3] で導入された手法に加えて，次の 3 点を追加・改良したものである．

(1) バッファオーバフローによるローカル変数の汚染から逃れるために，ポインタなどのローカル変数のスタック位置を文字配列より低いアドレスに移動する．

(2) バッファオーバフローによる関数引数の汚染から逃れるために，ポインタなどの引数を文字配列より低いアドレスにローカル変数として複製する．プログラム内での引数の使用は新たに複製された変数を利用するように変更する．

(3) 検出用コードによる性能低下を抑えるためにある種の関数では保護用コードの発生を停止する．

これらの保護手法を持つ本システムを propolice と呼ぶ．

2 章では本システムの攻撃検出・保護機構を説明す

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan

本論文の内容は 2001 年 7 月のコンピュータセキュリティ研究会にて報告され，CSEC 研究会主査により情報処理学会論文誌への掲載が推薦された論文である．

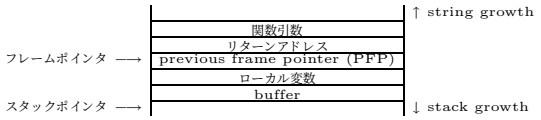


図 1 スタック構造
Fig. 1 Stack layout.

るために，スタックスマッシング攻撃を攻撃対象ごとに分類し，それぞれ説明する．3 章では関連研究について説明する．4 章ではスタックの完全性をチェックする手法について説明する．5 章では我々の保護手法を示し，どのようにして性能低下を抑えたのか説明する．6 章では実装について説明する．7 章では実験結果を示し，他の保護手法と比較する．最後に結論を述べ，本手法の実装状況について触れる．

2. 攻撃の手順と攻撃対象の分類

バッファオーバフローの脆弱性とは，外部からの情報を一時保管場所として C 言語のローカル変数に保管する際に発生する．問題となるのはローカル変数領域（バッファ）よりも与えられた情報量（文字数）が大きく，アプリケーションでその大きさをチェックしていない場合である．

図 1 は C 言語の関数呼び出し後のスタックレイアウトである．このレイアウトはプロセッサに依存し，オペレーティングシステムに依存しないものである．ここでは多くのプロセッサで使用されているレイアウトで説明する．領域を指し示すスタックポインタおよびフレームポインタはプロセッサのレジスタに保管されている．スタックポインタはスタックトップ（図では下側）を指し示し，スタックは低いアドレスの方に伸びる．スタックトップから順にローカル変数領域，直前のフレームポインタ（PFP），リターンアドレスそして関数引数が配置される．これらを関数のフレーム領域と呼び，実行中の関数の状態を保持する．フレームポインタは現在実行中のフレーム領域を指し示し，PFP は呼び出し側関数のフレーム領域を指し示す．

図 2 の関数 foo はバッファオーバフローに関して脆弱である．この関数は環境変数 HOME の値を 128 バイトの文字配列 buffer に読み込む．関数 strcpy は出力配列の大きさを確かめないため，128 文字以上のデータで buffer は溢れる．ここで関数 foo のローカル変数 lvar が buffer より高位アドレスに割り当てられているとする．環境変数 HOME に値 41（アスキーコードで ‘A’）が 128 バイト，値 1 が 4 バイト，値 2 が 4 バイト，値 3 が 4 バイトの 140 文字が設定されていると，関数 strcpy の終了後 buffer は 128 文字の

```
void foo()
{
   char *lvar;
   char buffer[128];
   .......
   strcpy (buffer, getenv ("HOME"));
   *lvar = 0;
   .......
}
```

図 2 バッファオーバフローの脆弱性を持つ関数例
Fig. 2 An example of buffer overflow.

‘A’ が，ポインタ変数 lvar は 16 進数で 0x01010101 が，PFP は 16 進数で 0x02020202 が，リターンアドレスは 16 進数で 0x03030303 になる．このスタック状態で，関数 foo の戻り処理の後で，プログラムはアドレス 0x03030303 にあるコードを処理する．このようにバッファオーバフローにより攻撃者は実行させたいコードを指定できる．

攻撃対象はリターンアドレスだけではない．ここでスタック上の攻撃対象を列挙し，その攻撃手順について説明する．

- リターンアドレス

多くのスタックスマッシング攻撃の攻撃対象である．攻撃用のプログラムコードをスタック上に埋め込むと同時に，埋め込んだコードのアドレスをリターンアドレス領域に設定することで攻撃コードへと導く．

- ローカル変数と関数引数

ローカル変数や関数引数として宣言された関数ポインタは明らかな攻撃対象である．攻撃の成立する条件は対象関数内にその関数ポインタによる関数呼び出しがある場合である．バッファオーバフローにより攻撃用のコードをスタック上に埋め込むと同時に，埋め込んだコードのアドレスを関数ポインタに設定することで攻撃コードへと導く．

関数ポインタの値を直接書き換えなくても，関数ポインタ呼び出しにより攻撃コードへと導ける例を示そう．ローカル変数に構造体へのポインタがあり，その構造体からたどれる要素に関数ポインタがあり，その関数ポインタを使用した関数呼び出しがある場合に生ずる．攻撃ではバッファオーバフローにより偽の構造体を埋め込み，構造体へのポインタが偽の構造体を指すように設定する．偽の構造体の関数ポインタの領域にはあらかじめ攻撃コードのアドレスを設定しておくことで，攻撃コードへと導ける．

関数ポインタ以外のポインタ変数も攻撃対象となる．図 2 では，関数 strcpy によるバッファオーバフローで lvar の値を改ざんした後にポインタ lvar の指す領域を 0 に設定している．つまり攻撃により任意のアド

レスの 1 バイトを 0 に設定できるということである．たとえばリターンアドレスや PFP の 1 バイトを 0 にすることで，本来のアドレスよりも下位のアドレスを指定できる．そのアドレスがバッファよりも高位にあるとき，攻撃が成立する．

- 直前のフレームポインタ（PFP）

バッファオーバフローにより PFP を変更し，攻撃コードへと導く方法を述べる．PFP とリターンアドレスは次の依存関係がある．

– リターンアドレスの位置はフレームポインタの相対位置である．
– 関数からの戻り処理でフレームポインタは PFP の値に設定される．

この依存関係から，PFP の改ざんがリターンアドレスの改ざんにつながる．準備として偽の関数フレームを作成する．偽フレーム内のリターンアドレスの位置には攻撃コードのアドレスを設定しておく．バッファオーバフローにより攻撃用のコードおよび偽の関数フレームをスタック上に埋め込むと同時に，埋め込んだ偽の関数フレームの位置を図 1 の PFP に設定する．関数のリターン処理で偽の関数フレームに導くことができ，その後もう一度リターン処理すると攻撃コードへ導ける．

3. 関 連 研 究

ここではバッファオーバフロー問題に取り組んでいる研究について，それぞれの利点と欠点について述べる．

研究には 2 つのアプローチがある．1 つは，アプリケーション開発時にバッファオーバフロー問題に対処する手法である．C 言語のソースコードレベルでバッファオーバフローの可能性をチェックし，危険性のあるコードをプログラマに修正させるという方法[11] である．たとえば，バッファオーバーフローの恐れのある関数（strcpy，gets など）の使用を指摘するというものである．しかしバッファオーバフローの原因は，ポインタ操作で境界チェックが誤っている場合にも発生する．残念ながらポインタの境界チェックをソースコードレベルで行うには限界がある．

もう 1 つのアプローチは実行時にバッファオーバフロー問題に対処する手法である．この防御方法は 3 つに分類される．

(1) 配列境界チェック

C 言語用の配列境界チェック[7] やメモリアクセスチェック[6] は配列用に割り当てられた領域の外へのアクセスを禁止する手法である．これはスタック以外の領域に対しても機能する．それゆえ防御手法としては最も安全な手法である．欠点は，防御のためのオーバヘッドが大きいことである．チェック用コードはソースコードからマシンコードへのコンパイル時に埋め込まれ，ポインタ操作のたびに境界をチェックする．そのためソフトウェアだけの実装では最適化されたプログラムに対して 2 倍程度の実行時間になる．

(2) スタック上のプログラム実行を禁止

Janus[5] は攻撃用コードの多くが/bin/sh を起動していることに着目して，プリビレッジモードにあるプログラムからは特別なプログラム（/bin/sh など）の起動を禁止した．欠点は，OS のデバッグ情報を提供する strace などの機能が前提となることから適用範囲が狭いこと，システムコールからなる攻撃は防御できないことである．

“Solar Designer” は Linux を元にして，スタック上でのプログラム実行を禁止する機能を開発[1] した．この手法の利点はソースコードを必要としないこと，実行時の防御に関わるオーバヘッドがないことである．欠点はスタックを実行不可領域とするためにオペレーティングシステムやプロセッサの支援を必要とすることとスタックフレーム領域の破壊を防ぐことができないことである．後者の欠点により，リターンアドレス，PFP，ローカル変数などを改ざんして，プログラムのコード領域に存在する適当な関数を，適当な引数で呼び出せる．たとえばライブラリの execute 関数に “/bin/sh” という引数を渡すことでシェルを起動できる．

(3) バッファオーバフローの検出

バッファオーバフローを検出し，攻撃用コードへ制御を移す前にプログラム実行を停止させるという手法がいくつか開発されている．Snarskii はスタック上に改ざん検出用の領域を確保し，その領域の完全性をチェックすることでバッファオーバフローを検出する FreeBSD 用の修正[9] を開発した．これは libc ライブラリ実行時のオーバフローを検出するために作成されたものであった．

より汎用性のある防御手法として，libsafe[2]，StackGuard[3] そして StackShield[10] がある．

libsafe はダイナミックリンクライブラリとして実装され，バッファオーバフローを起こしやすい関数（strcpy, gets など）の呼び出しをとらえる．このときに入力引数の内容をチェックし，フレームポインタより上位の領域を破壊しないことを確かめてから本来の関数を呼び出すことで，バッファオーバフローを防御する．

StackGuard はリターンアドレスの下位アドレスに改

ざん検出用の領域を確保し，その領域に攻撃者が推測できない値を入れておき，関数の出口でその領域の完全性をチェックすることでバッファオーバフローの検出を行う．

StackShield はリターンアドレスをバッファオーバフローの影響を受けない領域に保存しておき，関数の出口で保存した値を使用してリターンアドレスの完全性をチェックすることでバッファオーバフローの検出を行う．

我々の手法は StackGuard 手法を改良したものである．

次章で我々の手法を説明した後に，上記 4 つの手法との比較を 5.5 節で行う．

4. スタックの改ざん検出手法

この章では StackGuard [3] により導入された，guard 変数を使用したスタック領域の改ざん検出手法を説明する．本システムの改ざん検出手法はこの改良版である．StackGuard ではリターンアドレスの改ざん検出のために guard 変数という領域をリターンアドレスの直後に置いた．本システムはリターンアドレスだけでなく，PFP も保護するために guard 変数を PFP とバッファの間に置いた．具体的には，わずかなバッファオーバフローも検出できるようにバッファの高位の隣接領域に guard 変数を配置した．改ざん検出という点では PFP の改ざん検出を追加したことが改良点となっている．

これ以降の説明では，guard 変数の役割を明確にするため，特殊なセマンティックスを持つ C 言語を使用する．ここでは関数に定義されたローカル変数をその出現順にスタックに積むものとし，volatile 属性の付いたローカル変数は参照の度にメモリから読み込まれるものとする．

関数 foo（図 2）に改ざん検出機構を入れたプログラムは**図 3** のようになる．guard 変数を文字配列の直前に宣言し，関数の入り口では攻撃者に分からない値を guard 変数に格納し，関数の出口でその値が保持されていることを確認している．guard 変数が改ざんされていた場合はセキュリティログに警報メッセージを書き出し，プログラムの実行を停止する．

guard 変数に格納する値は攻撃者によって推測できない値でなくてはならない．もし攻撃者がその値を知っていた場合，guard 変数にその値を書きながら PFP やリターンアドレスを変更することで guard 変数のチェックをパスし，攻撃用コードへと導ける．

一般に，メモリ内の値はプリビレッジユーザ以外は読めない，したがって容易に推測できない乱数ならば

```
void foo()
{
   volatile int guard;        /*変数宣言部*/
   char buf[128];

   guard = guard_value;       /*関数の入口*/
        .......
   strcpy (buf, getenv ("HOME"));
        .......
   if (guard != guard_value){/************/
       /* output error log */ /*関数の出口*/
       /* halt execution */   /************/
   }
}
```

図 3 防御コードの挿入
Fig. 3 Guard code insertion.

guard 変数の要件を満たす．Linux や FreeBSD にはそのような乱数がデバイスとして実装（/dev/random）されている．これらは環境ノイズ（CPU 稼動状態，ネットワーク状態など）を利用して乱数を生成しているため，推測できない乱数を提供するものとして知られている．その値をアプリケーションの初期化で取得し，アプリケーションの動作中はその値を利用する．

5. スタック防御の手法

2 章で説明したようにスタックスマッシング攻撃から守るべき領域は 4 つある．関数引数，リターンアドレス，PFP およびローカル変数である．この章ではその 4 領域を保護する理想的なスタック配置（ここでは安全なフレーム構造と呼ぶ）を定義し，その安全性を説明する．次に一般のスタック配置から安全なフレーム構造に変換する手法について説明する．

5.1 安全なフレーム構造

関数引数，リターンアドレス，PFP およびローカル変数の 4 領域について制限付配置方法（**図 4**）を安全なフレーム構造と呼ぶ．

- 領域**(A)** は配列およびポインタを含まない．
- 領域**(B)** は配列や配列を含んだ構造体の集まり．
- 領域**(C)** は配列を含まない．

この構造は次のような 3 つの特徴を持つ．

(1)「関数から戻るとき，関数フレームよりスタックボトム側の領域は破壊から守られている」

領域**(B)** はバッファオーバフローの基点となる唯一の領域である．関数フレームよりスタックボトム側の領域が破壊されたときは必ず guard 変数のチェックで検出できる．また，そのときはプログラムが停止し，関数から戻ることはない．

(2)「関数フレームの外にあるポインタ変数を狙った攻撃は成立しない」

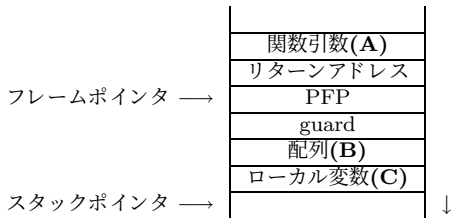


図 4 安全なフレーム構造
Fig. 4 Safe stack layout.

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

図 5 攻撃を受ける関数ポインタの例
Fig. 5 An example of vulnerable function pointer.

攻撃が成立するのは次の 2 つの条件が成立したときである．関数ポインタを例にすると，(1) 関数フレームの外にある関数ポインタを改ざんできること，(2) その関数ポインタを使用した関数呼び出しを実行すること．2 番目の条件を達成するためには関数ポインタがその関数から見えなくてはならない．前提条件から，その関数ポインタは関数スコープの外にあり，2 番目の条件は成立しない．

(3) 「関数フレーム内のポインタ変数を狙った攻撃は成功しない」

領域**(B)** はバッファオーバフローの基点となる唯一の領域である．破壊は領域**(B)** からスタックボトム方向に進んでいく．それゆえ領域**(C)** 中のポインタ変数を改ざんできない．

5.2 安全なフレーム構造への変換

図 5 は関数ポインタを改ざんすることでプログラムの制御を奪えるため，スタックスマッシング攻撃に対して脆弱である．

関数ポインタを防御するために，それぞれの変数の配置を安全なフレーム構造になるように変更する．まず関数ポインタ func2 を文字配列の次に配置する．関数引数 func1 の配置は変更できないため，func1 を新たに作成するローカル変数に複製する．そして func1 への参照はすべて新たなローカル変数への参照に変更する．**図 6** が変換結果である．

5.3 防御オーバヘッドの軽減

ここでは guard 変数の完全性チェックを軽減するための手法について議論する．次に示す 2 つの仮定を受け入れるとすると，guard 変数のチェックを省略できる場合がある．

```
void bar( void (*tmpfunc1)() )
{
        char buf[128];
        void (*func2)();
        void (*func1)(); func1 = tmpfunc1;
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

図 6 関数ポインタの防御
Fig. 6 Protection of function pointer.

仮定 1 スタックスマッシング攻撃を被るのは文字列処理である．

バッファオーバフローの発生する関数の多くは境界を検出するのに特別な値（終端文字）を利用している．その典型が文字列処理である．

仮定 2 文字列処理用の配列は文字列として型宣言されている

いい換えると，型宣言に従わないプログラム（たとえば integer 変数に cast 演算により文字列を格納するような処理）の防御をあきらめるということである．

この 2 つの仮定を受け入れると，文字配列をローカル変数や関数引数に持っていない関数では防御コードを省略できる．この手法の効果は 7 章で議論する．

5.4 制　　約

スタック防御はプログラム変換によって行われるが，実際にはすべての関数が安全なフレーム構造に変換できるわけではない．ここではそのような例をあげる．

- 1 つの構造体の中にポインタ変数と文字配列が使われている場合．
- 関数引数としてポインタ変数を使用し，そのポインタ変数が可変引数の一部として使用されている場合．
- スタックトップに割り当てなくてはならない，実行時に大きさが決定する文字配列を使用した場合．

5.5 防御手法の比較

表 1 は我々の手法，libsafe，StackGuard，StackShield の 4 つについて比較したものである．4 つの守るべき領域の防御状況，文字列関数の対応度，運用の観点から比較した．

防御の有効性はスタック上の保護すべき領域をどれだけ守れているか，文字列関数の対応度で評価できる．それぞれについて具体的な攻撃で比較する．

X ウインドウシステムに付属のディスプレイカードの判別プログラムである “SuperProbe 2.11” は関数引数に関数ポインタを含む構造体へのポインタを持っ

表 1 検出技術の特徴
Table 1 Summary of detection technique characteristics.

Description	None	propolice	libsafe	StackGuard	StackShield
Protection Effectiveness					
リターンアドレス	No	Yes[1]	Yes	Yes	Yes[2]
PFP	No	Yes[1]	Yes	No	Yes[2]
関数引数	No	Yes[1]	Yes	No	No
ローカル変数	No	Yes[1,3]	No	No	No
文字列関数への対応	No	All	Not all	All	All
一般配列への対応	No	No	No	All	All
Implementation characteristics					
OS 非依存	–	Yes	No[4]	No[5]	Yes
プロセッサ非依存	–	Yes	Yes	No	No
Other Characteristics					
オーバヘッド	None	Very low	Very low	Low	Low
ソースコードの必要性	–	Yes	No	Yes	Yes

1 5.3 節の仮定に従った場合
2 関数の呼び出しの深さが一定値以下
3 構造体にポインタ変数と文字配列が混在しない
4 ダイナミックライブラリが必要
5 メモリ保護機構が必要

ている．StackGuard および StackShield ではこの関数ポインタへの攻撃を防御できない．

RedHat5.2 に付属の xterm はライブラリ libtermcap 内の tgetent 関数にポインタ操作の不具合があったが，Libsafe はライブラリ関数のみを対象にしているため，独自の文字列操作にバッファオーバフロー問題がある場合には防御できない．

6. 実　　装

Gnu Compiler Collection（以下 GCC と呼ぶ）に対して行った実装について説明する．GCC は，C 言語などで記述されたソースコードからターゲットとなるマシン用の実行コードを出力するコンパイラである．その内部処理はソースコードから中間コード（以下 RTL と記述）を経由してターゲットのマシンコードに変換するもの（**図 7**）である．コンパイラの最適化処理は RTL レベルで複数の変換処理からなる．

RTL レベルでは，ソースコード上の変数はレジスタまたはメモリ領域のいずれかに割り当てられる．すなわちこの時点でローカル変数のスタック上の割当てが決定する．我々の実装は RTL レベルの変換処理として追加実装した部分 (A) と実レジスタに収まらなかった仮想レジスタをスタック上に割り当てる際のアドレス決定処理として実装した部分 (B) からなる．

(A) の変換処理は RTL で割り当てられているスタックレイアウトに対して，guard 変数の挿入，ローカル変数中のバッファとポインタ変数の配置変更，ポインタ宣言された引数をローカル変数に追加する．さらに guard 変数の初期化，完全性の確認および引数の複製処理を中間コードとして追加する．

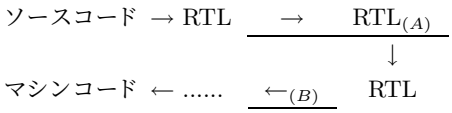


図 7 GCC での追加箇所
Fig. 7 Updated portion in GCC.

(B) の処理は，実レジスタに割り当てられなかったローカル変数（B の時点までは仮想レジスタとして扱っていた分）をスタック上のバッファよりも下位のアドレスに割り当てるという作業である．

7. 実 験 結 果

この章では propolice 実装された GCC を使用して，Redhat Linux6.2 で行った実験について議論する．

バッファオーバフロー問題を持つプログラムとその攻撃コードを集めて，防御前と防御後のプログラムに対して攻撃した．プログラムの名前およびバージョン番号，防御前の結果と防御後の結果の順（**表 2**）で示す．最初の 3 つはリターンアドレスへの攻撃であり，最後は関数ポインタを持つ構造体へのポインタが関数引数に与えられたプログラムに対する攻撃である．

いずれもバッファオーバフローを検出し，処理をやめている．

7.1 オーバヘッド

防御処理はプログラム実行の点でオーバヘッドとなる．Cowan[3] はオーバヘッドを次のように定義した： 関数あたりの防御前の CPU タイムと防御後の CPU タイムの比．我々の手法ではオーバヘッドはアプリケーションで文字配列が使われているか否かに依存する．文字配列の存在しない整数のソートプログラ

表 2 攻撃耐性の結果
Table 2 Protection result.

実験プログラム	防御前	防御後
xlockmore 3.10	root shell	terminated
Perl 5.003	root shell	terminated
elm 2.003	root shell	terminated
SuperProbe 2.11	root shell	terminated

```
int test()
{
   char buf[128];
   strcpy(buf, "1234567890");
   return strncmp(buf, "1234", 4);
}
```

図 8 オーバヘッド調査関数
Fig. 8 Overhead benchmark.

original run time	our method run time	overhead (%)
4.67	5.05	8%

図 9 防御コードのオーバヘッド
Fig. 9 Instrument overhead.

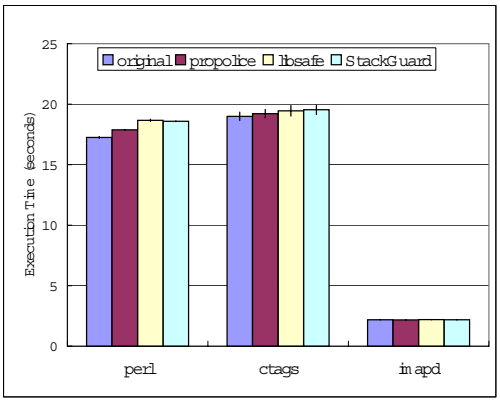


図 10 平均実行時間
Fig. 10 Average execution time.

表 3 オーバヘッドの比較
Table 3 Comparison of protection overhead.

	perl	ctags	imapd
propolice	4%	1%	0%
libsafe	8%	2%	0%
StackGuard	8%	3%	0%

ムや線形計画法のプログラムでは，オーバヘッドは 0 である．

我々の手法で，防御コード挿入にともなうオーバヘッドの運用上の上界を調べるために，**図 8** を用意した．

実験に使用したプロセッサは Pentium III 600 MHz で 512 k のレベル 2 キャッシュを持ち，256 M のメインメモリで計測した．表は 50,000,000 回あたりの秒数を示す．これによるとオーバヘッドは約 8%ということになる（**図 9**）．

防御コードで最もオーバヘッドの大きな処理は，関数引数に使用された文字配列の退避である．これは文字配列をローカル変数にコピーする処理である．実際にどの程度の頻度でそのような関数が出現するのか調べたところ，Linux システム（カーネル，ライブラリと/usr/bin 以下のプログラム）を調べたところ，どのプログラムにも関数引数に文字配列を持つ関数はなかった．それゆえ，この場合のオーバヘッドは実際上問題とならないと考えられる．

次に，関数引数にポインタが使われている場合のオーバヘッドについて議論する．我々の実装ではポインタ引数は仮想レジスタにコピーしている．さらにそのレジスタは最適化処理で元のポインタ引数に置き換えられないようにした．ローカル変数へのコピーが発生するのは，仮想レジスタが実レジスタに割り当てられなかったときである．レジスタはバッファオーバフローの影響を受けないため，この引数コピー処理はバッファオーバフローに対して安全である．しかもオーバーヘッドの影響が大きいと考えられる小さな関数ではレジスタが余っていて，ローカル変数コピーは発生しないと考えられる．以上のことからコピーのオーバヘッドは小さいと予想される．

図 10 は現実の 3 つのアプリケーションに関して実行時オーバヘッドを計測したものである．比較のために libsafe および StackGuard も計測した．StackGuard は guard に乱数を使用するバージョンで計測した．アプリケーションには CPU バウンドの`perlbench`[8]，I/O バウンドの`ctags` を egcs-1.1.2 のディレクトリに対して適用する試験，そしてネットワークを使用する`imapd` で 2K バイトのメッセージを 100 回送るという試験をした．

実行時間はそれぞれ 100 回計測し，95%の信頼区間（棒グラフの先端に付いた線分）で示した．時間はすべて/bin/time を使用した経過時間である．**表 3** はそれぞれの防御方法のオーバヘッドを示しており，我々の手法が最もオーバヘッドが少ないことが分かる．つまり文字配列を含む関数を選択的に防御する手法が実行時間の観点から効果的であることが示された．

8. おわりに

スタックスマッシング攻撃で攻撃対象となる領域について網羅的に説明した．またその領域に対する攻撃手法について説明した．

我々の手法は StackGuard 手法から次の点を改善し

たことを述べた．

(1) ローカル変数内の配置を換えることで，ポインタ変数を防御した．
(2) 関数引数をローカル変数へコピーすることで，ポインタ変数を防御した．
(3) PFP を保護するとともに，文字配列の使用時のみに防御コードを挿入することでパフォーマンスオーバヘッドを軽減した．

その結果，我々の手法が libsafe，StackGuard，StackShield よりも多くの攻撃を防ぐことを実験し，最も少ないオーバヘッドで動作することを示した．文字列処理のいくつかのアプリケーションで 4%以内の性能低下で動作することを実験で示し，また数値計算系のアプリケーションで性能低下はないことを説明した．

本システムの実装状況は GCC のリリース egcs-1.1.2，gcc-2.95.2，gcc-2.95.3，gcc-3.2 への対応が完了している．また，Intel x86 系，powerpc，sparc，MIPS の各プロセッサでの動作確認が完了している．また，Linux，FreeBSD，OpenBSD，NetBSD，AIX，および Solaris のオペレーティングシステムで動作確認できている．

参 考 文 献

1) “Solar Designer”, Non-Executable User Stack. http://www.false.com/security/linux/
2) Baratloo, A., Singh, N. and Tsai, T.: Transparent Run-Time Defense Against Stack Smashing Attacks, *Proc. USENIX Annual Technical Conference* (2000).
3) Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Steve Beattie, A.G., Wagle, P. and Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. 7th USENIX Security Symposium* (1998).
4) Eisenberg, T.: The Cornell Commission: On Morris and the Worm, *Comm. ACM*, Vol.32, No.6 (1989).
5) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A.: A secure environment for untrusted helper applications, *Proc. 6th USENIX Security Symposium* (1996).
6) Hastings, R. and Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors, *Proc. Winter USENIX Conference* (1992).
7) Jones, R. and Kelly, P.: Bounds Checking for C (1995). http://www-ala.doc.ic.ac.uk/p̃hjk/BoundsChecking.html
8) Perlbench. http://www.metacard.com/ perlbench.html
9) Snarskii, A.: FreeBSD Stack Integrity Patch (1997). ftp:// ftp.lucky.net/pub/unix/local/libc-letter
10) “Vendicator”, Stack Shield: A “stack smashing” technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/
11) Viega, J., Bloch, J., Khono, T. and McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code, *Proc. ACSAC 2000* (2000).

(平成 14 年 1 月 30 日受付)
(平成 14 年 9 月 5 日採録)

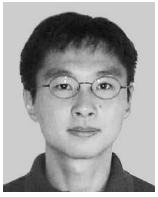
推 薦 文

プログラミング言語環境におけるセキュリティについて分かりやすく述べ，バッファオーバフローに起因する攻撃に対し，従来の対策を改良した．実装による性能評価も行い，推薦に値する内容である．

（CSEC 研究会主査 岡本 栄司）

江藤 博明（正会員）

昭和 36 年生．昭和 60 年東京工業大学大学院工学研究科情報工学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．現在東京基礎研究所主任研究員．最適化問題（プリント基板の配線問題），ネットワークセキュリティ等の研究に従事．



依田 邦和（正会員）

昭和 47 年生．平成 8 年京都大学大学院工学研究科応用システム科学専攻修士課程修了．日本アイ・ビー・エム（株）入社．現在東京基礎研究所副主任研究員データベース，ネットワークセキュリティ，スケジューリング問題の最適化等の研究に従事，ACM 会員．