コンピュータセキュリティ　14−25
（2001. 7. 25）

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

日本アイ・ビー・エム株式会社 東京基礎研究所

神奈川県大和市下鶴間 1623 番 14
{etoh,yoda}@jp.ibm.com

あらまし　C 言語で構築されたアプリケーションでは、バッファーオーバーフローに起因するスタック破壊によりプログラム制御が奪われてしまうことがある。本論分ではバッファーオーバー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ー変数がスタック破壊で操作されないように、スタック上の場所を変更すること、（2）関数の引数に現れるポインター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）パフォーマンスオーバーヘッドを減らすために、ある条件の関数ではプロテクションコードの生成を止めること、の三点である。

キーワード　セキュリティ スタック コンパイラ オーバーフロー 防御

propolice: Improved stack-smashing attack detection

Hiroaki Etoh, Kunikazu Yoda

Tokyo Research Laboratory, IBM Japan

1623-14, Shimotsuruma, Yamato-shi, Kanagawa, 242, Japan
{etoh,yoda}@jp.ibm.com

Abstract　This paper presents some new 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 further corrupt arbitrary memory locations, (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.

Key words　Security Stack Protection Overflow Compiler

1 はじめに

バッファーオーバーフロー脆弱性問題は、1998 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、これまでの間さまざまなアプリケーションで問題が明らかにされてきた。今年 (2001 年) も次のような良く使われているアプリケーションで問題が明らかになった。

- 01-29: ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability
- 04-04: Ntpd Remote Buffer Overflow Vulnerability
- 05-01: Microsoft Windows 2000 IIS 5.0 IPP ISAPI 'Host:' Buffer Overflow Vulnerability

これらの問題はいずれも、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入力が与えられた際に発生している。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、実際の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ある。破壊対象には、リターンアドレスであったり、関数ポインタが含まれる。

本論文ではバッファーオーバー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のコンパイル時に自動的にバッファーオーバーフローを検知する機構を挿入する。この機構は StackGuard[3] で導入された手法に加えて、次の三点を改良したものである。

1. バッファーオーバーフローによるローカル変数の汚染から逃れるために、ポインター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. バッファーオーバーフローによる関数引数の汚染から逃れるために、ポインター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. 保護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った本システムを propolice と呼ぶ。

節 2 では保護手法がどのように攻撃を防ぐのかを説明するために、スタックスマッシング攻撃の攻撃対象について分類する。節 3 では関連研究について説明し、節 4 でスタックの完全性をチェックする手法について説明し、節 5 で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。節 6 で実験結果を示し、他の保護手法との比較をする。最後に結論と実装状況について述べる。

2 攻撃の手順と攻撃対象の分類

バッファーオーバーフローの脆弱性とは、アプリケーションが外部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。多くの場合ローカル変数領域 (バッファー) が与えられた情報 (文字) サイズよりも小さく、アプリケーションでそのサイズをチェックしていないために発生する。

スタックの構造を示して説明する．図 1 は C 言語における関数呼び出し後のスタックの内容を示している．図中の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割り当てられている。スタックポインタはスタックトップ (図では下側) を指し示す。スタックトップ側が低いアドレスになる。スタックトップから順にローカル変数領域，関数の戻り先の変数領域を示すフレームポインタ (PFP)，リターンアドレスそして関数引数 が配置されている．これらのデータを関数のフレーム領域と呼び、実行中の関数の状態を表す。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す。PFP は呼び出し側関数のフレーム領域を指し示している。

図 2 中の関数 foo はバファーオーバーフローに関して脆弱である。この関数は環境変数 HOME を 128 バイトの領域を持つ文字配列 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、buffer に 128 文字以上のデータを書き込むかもしれない。ここで関数 foo のローカル変数 lvar および buffer がこの順番でスタック上に割り当てられているものとして説明する。例えば、環境変数 HOME が 128 バイトの値 41 (アスキーコードで'A')、4 バイトの値 1、4 バイトの値 2、4 バイトの値 3 からなる 140 文字が設定されているとする。関数 strcpy の終了後 buffer には 128 文字の'A' が、ポインター変数 lvar には 0x01010101 が、PFP には 0x02020202 が、リターンアドレスには 0x03030303 が設定される。 関数 foo が処理を終え、このスタック状態で呼び出し側への戻り処理を進めると、プログラムはアドレス 0x03030303 へと処理を進める。このアドレスは正しいリターンアドレスではない、つまり攻撃者は適当なアドレスへとプログラムの処理を導ける。

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

図 2: バッファーオーバーフローの脆弱性を持つ関数例

攻撃対象としてリターンアドレスを例にあげてきた。次にスタック上の攻撃対象を列挙し、その攻撃手順につ

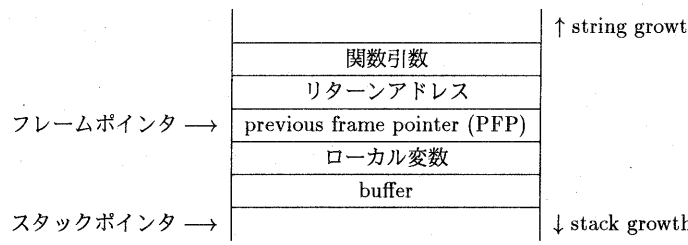

図 1: スタック構造

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はリターンアドレスを対象にしている。攻撃用のコードをスタック上に埋め込むと同時に、埋め込んだコードへ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攻撃対象となる。攻撃が成立する条件は対象関数内で関数ポインタを利用した関数呼び出しがある場合である。バッファーオーバー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、埋め込んだコードへ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。ローカル変数として構造体へのポインタが宣言されていて、その構造体から辿れるデータとして関数ポインタがあり、その関数ポインタを利用した関数呼び出しがある場合である。攻撃ではバッファーオーバー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場所には予め攻撃コードへのアドレスが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 の変数 lvar を例にして説明する。関数 strcpy によるバッファーオーバーフローで lvar の値が変更された後で、ポインタ lvar の指す領域に値を設定している場合、PFP の一部を変更することで攻撃を行う。PFP はスタック上を指しており、PFP の最下位バイトを変更した値も多くの場合、スタック上を指している。PFP を変更することで攻撃コードへと導く方法は次項に述べる。

- 関数の戻り先の変数領域を示すフレームポインタ (PFP)

  バッファーオーバーフローにより 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によって決定される。
  - 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。

  この依存関係を利用することで攻撃が成立する。準備として偽の関数フレームを作成する。この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。バッファーオーバー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、埋め込んだ関数フレームの位置を PFP に設定する。関数のリターン処理により偽の関数フレームに導くことができ、その後もう一度関数リターン処理することで攻撃コードへと導ける。

3 関連研究

ここではバッファーオーバーフロー問題に取り組む研究について、その手法と利点・欠点を述べる。

一つのアプローチとして、実行前にバッファーオーバーフロー問題の対処する手法がある。C 言語のソースコードレベルでバッファーオーバーフローを起こす可能性をチェックし、実行時にバッファーオーバーフローが発生しないようにプログラマに修正をさせるという方法 [11] である。例えば、バッファーオーバーフローを発生させる恐れのある関数 (strcpy, gets など) の使用を指摘するというものである。しかし実際にバッファーオーバーフローを発生させる原因は、それらの関数だけでなく単なる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。

もう一つのアプローチは実行時にバッファーオーバーフロー問題に対処する手法である。これは防御方法で 4 つに分けられる。

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。その領域がスタック上であれ、ヒープ上であれ、グローバルデータ領域であっても機能する。それゆえ防御手法としては最も安全な手法であるが、防御のためのオーバーヘッドが大きい。チェック用コードはソースコードからマシンコードにコンパイルする際に自動的に埋め込まれ、ポインタ操作のたびに境界チェックすることでバッファーオーバーフローを防止する。そのためソフトウェアによる領域チェックでは最適化されたプログラムに対しておよそ二倍程度遅くなる。

2. スタック上のプログラム実行を禁止

   "Solar Designer" は Linux 用の修正として、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。この手法のメリット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ーヘッドがないことの二点である。欠点としてスタックを実行不可とするための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、節 2 に述べた領域の破壊を防ぐことは出来ないことが挙げられる。しかもこれらの破壊により攻撃用コードへ導ける。リターンアドレスの改ざんにより任意のアドレスに制御を移せることおよびその際のスタック状態を制御できることから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。例えばライブラリの 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。

   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）の起動を禁止するというデザインをした。この手法では OS のデバッグ情報を提供する strace などの機能の利用が前提となる。このシステムでは典型的な攻撃用コードに関して防御できるが、システムコールを組み合わせた攻撃については防御できない。

3. バッファーオーバーフローの検出

   バッファーオーバーフローを検出し、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幾つか開発されている。Snarskii はスタック上の完全性をチェックすることでバッファーオーバーフローを堅守する FreeBSD 用の修正 [9] を開発した。これは libc ライブラリ専用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、StackGuard [3], StackShield [10], そして libsafe [2] がある。

   libsafe はアプリケーションプログラムとライブラリの中間に実装され、バッファーオーバーフローを起こしやすい関数（strcpy, gets など）の呼び出しを捉まえる。この際に入力引数の内容をチェックし、PFP やリターンアドレス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、バッファーオーバーフローを防御する。

   StackGuard はリターンアドレスの上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く。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ーオーバーフローの検出を行う。

   StackShield はリターンアドレスをバッファーオーバーフローの影響を受けない領域にコピーしておき、関数の出口でリターンアドレスが保存されていることをチェックすることでバッファーオーバー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。次節で我々の手法を説明した後に、上記 4 つの手法の比較を節 5.5 で行う。

4 リターンアドレスの防御

この節では StackGuard [3] により導入された、guard 変数を使用したリターンアドレスの改ざん検出手法を説明する。我々も改ざん検出手法としてこれを採用している。StackGuard ではリターンアドレスの改ざん検出のために guard 変数という領域がリターンアドレスの直後に置かれたが、我々の手法で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 と配列の間に置いた。実際には、わずかなバッファーオーバーフローも検出できるように配列の直前に 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。

guard 変数の役割を説明するために、C 言語のソースコードレベルの変更に置き換えて説明する。一つの関数が与えられたとき、バッファーオーバーフローを検出するための幾つかのコードが、関数内のそれぞれの場所に挿入されていく。それは変数宣言部、関数の入口および関数の出口である。

関数 foo の変更後のソースプログラムは図 3 のようになる。重要なことは guard 変数は文字配列の直前に宣言することである。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に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。

```
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

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には、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できてしまう。この場合、guard 変数のチェックは成功することになり、攻撃用コードへと導かれてしまう。

そこで guard 変数には乱数を利用する。アプリケーションの初期化時に乱数を作成しておき、アプリケーションが動作中はその値を利用する。メモリ内にあるそ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。

5 スタック防御の手法

節 2 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の 4 領域である。この節ではその 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、その安全性を証明する。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 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

このモデルは次のような特徴を持つ。

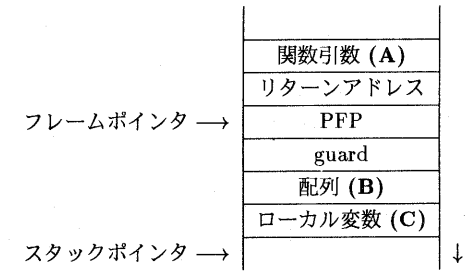

図 4: 安全なフレーム構造

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)** はバッファーオーバー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

   そのような攻撃が成立するとすれば、次の二つの条件が成立しなくてはならない。(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、二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)** はバッファーオーバーフローの基点となる唯一の領域である。破壊は領域 **(C)** から遠ざかる方向に進んでいく。それゆえ破壊 **(C)** は攻撃にさらされない。

5.2 ポインタの防御

図 5 はスタックスマッシング攻撃に対して脆弱である。関数ポインタを変更することでプログラムの制御を奪える。

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

図 5: 関数ポインタへの攻撃

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。C 言語ではローカル変数の配置に関して制約を設けていない。そこでローカル変数上の関数ポインタ func2 をモデルに従い文字配列の後ろに配置する。一方、C 言語では関数引数の配置を変更することは許されない。その制約を解消するために関数ポインタ 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。

```
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

5.3 防御オーバーヘッドの軽減

ここでは guard 変数を実装する際のオーバーヘッドを軽減するための手法について議論する。次に示す二つの仮定を受け入れるとすると、guard 変数の実装を省略できる場合がある。

仮定 1 アプリケーションの外部からのデータで、バッファーオーバーフローを発生するのは文字列処理である

バッファーオーバー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 ソースコードでは正しくタイプ宣言されていて、デフォルトの型変換ルールに従っている

言い換えると、integer 変数には必ず整数が入り、決して文字列が格納されることはないということ。

もし二つの仮定を受け入れると、関数で文字配列をローカル変数や関数引数で持っていない場合には防御コードを省略できる。この手法の効果は節 6 で議論する。

5.4 制約

スタック防御手法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挙げる。

- 一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。
- 実行時にサイズが決定する文字配列を使用した場合のポインタ変数

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対応状況および運用上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、文字列関数に関して制限がないかということで示せる。なぜなら足りない部分を利用した攻撃の可能性について節 2 で示した。さらに実際の攻撃コードについて示す。

X ウインドウシステムに付属のディスプレイシステム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ている。この関数ポインタへの攻撃を StackGuard および StackShield は防御できない。

Libsafe は文字列関数のみを対象にしているため、文字列操作にバッファーオーバーフロー問題がある場合には防御できない。例えば 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。

我々の手法の実装上の特徴として、オペレーティングシステムやプロセッサを選ばないという点が挙げられる。これまでは、ソースコードの変換を使って説明したが、実際の防御コードの挿入はソースコードレベルでは実装困難である。例えば文字配列の直前に guard 変数を配置したいが、ソース上の位置とスタック上の位置は必ずしも関係はない。この問題を解消するために、我々は GCC の中間言語レベルで実装した。中間言語で実装したことでプロセッサに依存しない防御手法を提供できた。

6 実験結果

バッファーオーバー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、最後の一つは関数ポインタを持つ構造体へのポインタへの攻撃である。

この表は包括的なリストではないが、防御方法が働くことを示すには充分である。残念ながらローカル変数を対象にした攻撃プログラムは存在しなかった。推測するにローカル変数への攻撃が可能ならば、リターンアドレスへの攻撃も可能だろう。そのためより実装の簡単なリターンアドレスの攻撃が一般的なのだろう。

6.1 オーバーヘッド

防御手法はプログラム実行という点で余分なオーバーヘッドとなる。Crispin[3] はオーバーヘッドを次のよう

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
文字列操作への対応	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					
オーバーヘッド	None	Very low	Very low	Low	Low
ソースコードの必要性	–	Yes	No	Yes	Yes

1 節 5.3 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

表 1: Summary of Detection Technique Characteristics

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

表 2: Penetration Resistance

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ー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ムや線形計画法のプログラムなどのオーバーヘッドは 0 である。

防御コード挿入に伴うオーバーヘッドの運用上の上界を調べるために、プログラム 7 を用意した。

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

図 7: 防御コードのオーバーヘッド調査用

実験に使用したプロセッサは Pentium III 600MHz で 512k のレベル 2 キャッシュを持ち,256M 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ーヘッドは約８％ということになる。

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

我々の手法では、全体の関数呼び出しに対する文字配列を使用している関数の呼び出しの割合だけオーバーヘッドは軽減される。Linux システム全体で、文字配列を使用している関数の数に対して全体の関数の数の割合を調べたが、もっとも多い割合を示したものはライブラリの glibc で 14.7%, ほとんどの割合は 10%以下であった。この比は実行時の比率ではないが多少の相関があるとすると、オーバーヘッドの軽減に役立っていると考えられる。

防御コード発生時、関数引数に文字配列が使われているとそのサイズ分だけはローカル変数にコピーされる。防御コードのオーバーヘッドという観点からどの程度の割合でそのような関数が呼ばれるのか調べた。表 ?? で挙げたすべて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。C 言語の場合配列はポインタで渡すことが徹底されているのだろう。一方、関数引数にポインタが使われている場合、やはりローカル変数領域にコピーしなくてはならない。そのためのオーバーヘッドについて議論する。ポインタ変数のコピーは実装上の最適化により、必ずしもローカル変数にコピーされるわけではない。最適化によりレジスタに割りあたっているからである。防御用の関数引数の処理が特別なのはレジスタが足りなくなったときの処理である。普通の関数引数ではレジスタが足りなくなればレジスタを開けるだけだが、我々の処理ではローカル変数にコピーするという処理が入る。レジスタはバッファーオーバーフローに対して安全であるから、この最適化もバッファーオーバーフローに対して安全である。しかもオーバーヘッドに影響のあるほど処理の軽い関数ではレジスタも余っていると予想されるためコピーのオーバーヘッドは 0 に近いと予想される。

図 8 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ー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 は乱数を guard に使用する物で計測した。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、そしてネットワークを使用する **imapd** で 2 K 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%の信頼区間を示した。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ベーヘッドを示しており、我々の手法が最もオーバーヘッドが少ないことがわかる。

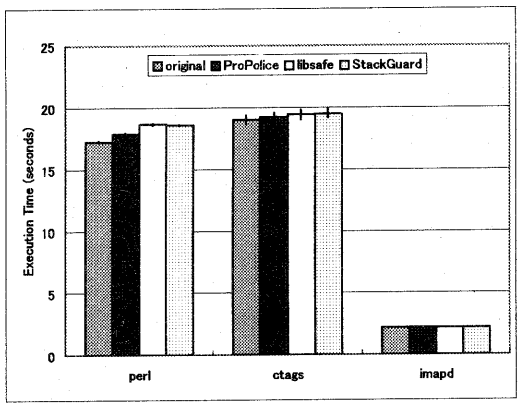

図 8: 平均実行時間

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

表 3: オーバーヘッドの比較

7 おわりに

スタックスマッシング攻撃でスタック上のどの領域が攻撃対象となるのか、またその攻撃手法について網羅的に説明した。

我々の手法が StackGuard 手法から次の点で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保護コードを挿入することでパフォーマンスオーバーヘッドを軽減した。

その結果、われわれの手法が libsafe StackGuard StackShield よりも多くの攻撃用コードを防いぐことを実験し、最も少ないオーバーヘッドで動作することを示した。文字列処理の幾つかのアプリケーションで防御コードのないものと 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は性能低下はないことを説明した。

GCC の中間言語レベルの変換プログラムとして実装しており、egcs-1.1.2, gcc-2.95.2, gcc-2.95.3, gcc-3.0 の各リリースへの対応が完了している。また、Intel X86 系, powerpc, sparc への動作確認が完了している。 OS では Linux, FreeBSD, AIX, および Solaris 上で動作確認が完了している。現在、gcc-3.1 に取り込まれるように活動中である。

参考文献

[1] "Solar Designer". non-executable user stack. http://www.false.com/security/linux/.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, A. G. Steve Beattie, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings in the 7th USENIX Security Symposium*, January 1998.

[4] T. Eisenberg. The Cornell Commission: On Morris and the Worm. *Communications of the ACM*, 32(6), 1989.

[5] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[6] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*. 1992.

[7] R. Jones and P. Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html, July 1995.

[8] Perlbench. http://www.metacard.com/perlbench.html.

[9] A. Snarskii. FreeBSD stack integrity patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[10] "Vendicator". Stack shield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/.

[11] J. Viega, J. Bloch, T. Khono, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceeginds of ACSAC 2000*, 2000. To appear.