プログラム変換によるスタックオーバーフロー攻撃対策に関する研究

M2006MM012 河合 秀哉
指導教員　真野 芳久

1　はじめに

これまでインターネットワームをはじめとする悪性ソフトウェアによって、様々なコンピュータが被害を受けてきた。近年ではコンピュータシステムに対する攻撃の多くを、バッファオーバーフローによる攻撃が占める。このようにバッファオーバーフローの脆弱性は日常的に存在する脅威と言える。バッファオーバーフロー攻撃の中でも一般的なものとして、スタック構造の内部を改ざんするスタックオーバーフロー攻撃が挙げられる。本研究ではスタック上のリターンアドレスの保護、バッファの隔離という 2 つの観点から、未知の攻撃に対応でき、既存のリソースを生かすことができる開発環境でのスタックオーバーフローに対する攻撃耐性を高める方法を提案する。

2　スタックオーバーフロー

スタックオーバーフロー攻撃は、他のバッファオーバーフロー攻撃に比べ、比較的簡単に攻撃を行うことができるため、バッファオーバーフロー攻撃の中でもスタックオーバーフロー攻撃がよく攻撃方法として用いられる。スタックオーバーフローの脆弱性は、主に C 言語で作られたプログラムで発生する。このようなプログラムでは、スタック上に文字配列領域を確保する。スタックオーバーフロー攻撃は、この文字配列領域に確保された長さよりも多くの文字列を与えることにより、その領域から溢れ出た文字列で、他の領域が改ざんされ、本来の動作とは異なる処理を行わせるものである。

スタックオーバーフロー攻撃により、攻撃者がスタック上のリターンアドレスを任意の値に改ざんした場合、関数の実行が終了して呼出し元の関数へ戻る際に、本来の呼出し元の代わりに改ざんされた値が指すリターンアドレスにジャンプして、そこから命令の実行を再開してしまう。改ざんするリターンアドレスの値を工夫して、溢れたバッファの内容を指すことで、溢れたバッファの内容を命令列として実行することも可能である。したがって、攻撃者はバッファの内容として攻撃コードを与えることにより、任意のコードを実行することが可能である。また、スタック上で攻撃対象となるのは、リターンアドレスだけではない。ローカル変数や関数引数、フレームポインタなどを改ざんすることで、攻撃コードに導くことも可能である。

図 1 は、関数の開始直後のスタックフレームの構造である。領域を指しているスタックポインタとフレームポインタはプロセッサのレジスタに保存されている。スタックポインタはスタックトップを指し、スタックは低いアドレスへ成長する。スタックボトムから順に、関数引数、リターンアドレス、直前のフレームポインタ、ローカル変数領域が配置される。フレームポインタは現在実行中のスタックフレームを指し、直前のフレームポインタは呼出し側の関数のスタックフレームを指す。関数の呼出しごとに、スタックに図 1 のような関数の環境を保存するための領域が割り当てられる。

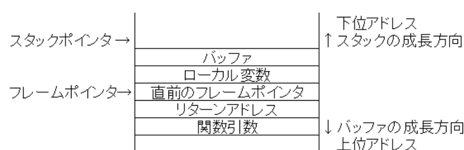

図 1　スタックフレームの構造

3　関連研究

これまでに提案された本研究に関連するスタックオーバーフローの対策方法について述べる。

3.1　コンパイラレベルによる対策

StackGuard [2] に代表されるカナリア法という方法があり、マイクロソフトの C コンパイラなどにも採用されている。その中でも無作為カナリア法は、関数に入る際に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困難な値を挿入する。関数の出口で領域が改ざんされていないかをチェックする。改ざんされていたらスタックオーバーフローとして検出する。

Propolice [4] は、StackGuard に加えて次の点を追加し、改良したものである。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。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。検出コードによる性能低下を抑えるために、文字配列を持たない関数では保護用コードの発生を停止する。

3.2　アセンブリレベルによる対策

Stack Shield [5] は、関数処理の開始時にリターンアドレスをスタックとは別の安全な場所に保存する。そして関数処理を終える直前に、スタック上にリターンアドレスを復帰させることで攻撃を防止する。

3.3　プリプロセッサレベルによる対策

Gemini [3] は、スタック領域に配置されているバッファをヒープ領域に隔離する。スタックオーバーフロー

が発生しても、スタックに配置されているリターンアドレスが改ざんされることはないが、スタックオーバーフローのバグそのものの対策をしているわけではないため、ヒープ領域に隔離されているバッファが溢れ出すことになる。しかし、スタック領域に配置されているリターンアドレスには直接影響はないため、リターンアドレスが改ざんされることはない。また、攻撃者にとってヒープオーバーフローを利用しての攻撃は非常に難しく、多くの場合不可能である。よって、スタック領域にバッファが配置されている場合よりも、スタックオーバーフローの脆弱性は低くなるため安全性が増す。

ASR [1] は、メモリを使用する方法にランダム性を導入することで、メモリエラーを突く攻撃を無力化する。スタック領域に置かれたデータだけでなく、ヒープ領域や静的領域など様々なメモリ上のデータとコードを、あらかじめ確保した領域にランダムに配置する。スタックオーバーフロー攻撃だけでなく、ヒープオーバーフロー攻撃など、他のメモリ攻撃にも対応する。

4 リターンアドレスの保護による対策方法

4.1 Stack Shield の制限

Stack Shield は、関数処理の開始時にリターンアドレスをスタックとは別の安全な場所に保護する。スタックオーバーフローの発生により、リターンアドレスが改ざんされたとしても、保護されたリターンアドレスをスタックに復帰させることで、プログラムを安全に実行することができる。しかし Stack Shield では、再帰処理等でリターンアドレスがたくさんスタックに積まれた場合に、全てのリターンアドレスを保護することができない。よって、スタックオーバーフロー攻撃から完全にリターンアドレスを守ることができない場合がある。さらに Stack Shield では、リターンアドレスのみを攻撃からの保護対象としている。フレームポインタを攻撃から保護していないので、フレームポインタの改ざんがリターンアドレスの改ざんにつながる場合も考えられる。

4.2 本提案でのアプローチ

本提案では次の点に関して検討することで、Stack Shield よりもスタックオーバーフロー攻撃に対する安全性を高める。

1. 全てのリターンアドレスの保護
2. フレームポインタの保護

本提案では次の手順で攻撃に対する対策を行う。全てのリターンアドレスを保護できるよう、保護領域の自動的な拡張ができるようにする。Stack Shield で使用する保護領域は、大域変数を利用している。しかし、本提案ではヒープ領域を利用して動的に確保することで、保護領域の拡張を可能にする。

リターンアドレスの保護領域を拡張するための手順は次のようになる。保護領域に空き場所がない場合、malloc() 等の動的にメモリの確保ができる関数を利用して、元の保護領域よりも大きな領域を確保する。元の領域から新しい領域へ保護しているリターンアドレスを複製する。元の領域を参照するために必要なポインタを新しい領域が参照できるように変更する。これにより保護領域の自動的な拡張が可能になり、リターンアドレスを全て保護することができる。

さらに、フレームポインタもリターンアドレスと同様に危険性があるので保護する必要がある。リターンアドレスの保護方法と同様の方法を利用し、フレームポインタを保護する。

4.3 提案方法実現の検討

プログラムのアセンブリコードでは、スタックやレジスタの操作ができるので、スタックに積まれているリターンアドレスを参照することができる。そのため本提案は、アセンブリレベルでコードの挿入を行う。

本提案では次のコードを挿入する。

1. 保護領域へのポインタと保護領域の拡張を行う関数
2. プロローグコード
3. エピローグコード

保護領域へのポインタを利用してリターンアドレスの参照と保存を行う。また、保護領域に空き場所があるかの確認も行う。保護領域の拡張を行う関数は、保護領域の空き場所が無くなると、リターンアドレスとフレームポインタの保護ができなくなるので、その保護領域の拡張を行う関数である。元の領域よりも大きな領域を用意し、元の領域から新しい領域へメモリの複製を行う。その後は、新しい領域を保護領域として利用する。また、元の領域から新しい領域へポインタを付け替え、元の領域のメモリは解放する。

さらに、各関数にはプロローグコードとエピローグコードが挿入される。各関数の始めに挿入されるプロローグコードは、リターンアドレスとフレームポインタを保護領域に保存する処理である。もし、保護領域に空き場所が無い場合は、保護領域の拡張を行う関数を実行して、新しい領域に保存する。また、main 関数にのみ、保護領域の新規作成と作成された領域に対するポインタの設定をする処理も追加する。プログラムはこの領域をリターンアドレスとフレームポインタを保護するための領域として使用する。また、この領域は保護領域の拡張を行う関数が実行され、新しい領域が確保された場合には解放される。各関数の最後に挿入されるエピローグコードは、プロローグコードで保護したリターンアドレスとフレームポインタをスタックに復帰する処理である。

5 バッファの隔離による対策方法

5.1 従来方法の制限

Gemini はスタック領域に配置されているバッファをヒープ領域に隔離する。しかし、配列だけを隔離の対象としており、構造体を対象としていない。構造体が配列を含む場合や、構造体を要素とする配列である場合も考えられ、もしこれらがスタックに存在しており、バッ

ファオーバーフローが発生した場合、リターンアドレスが改ざんされる可能性が非常に高い。

ASR はメモリを使用する方法にランダム性を導入することで、メモリエラーを突く攻撃を無力化する。スタックだけでなく、様々なメモリ上のデータやコードの位置をランダム化しており、保護するデータの対象範囲が広い。また、プログラムの変換により、配列の参照や演算などで問題が発生する。Gemini は sizeof 演算子に対する変換だけで解決しているが、ASR は配列に対する参照を全て変換している。

5.2 本提案のアプローチ

Gemini のように、配列をスタック領域から隔離することで、スタックオーバーフロー攻撃の対策を行う方法は多く存在するが、スタックオーバーフローは配列以外の場合も発生する。本提案では配列だけでなく、構造体も隔離するバッファの対象とするので、構造体のスタックオーバーフローによるリターンアドレスの改ざんも防止することができる。

本提案では、図 2 のようなプログラムを図 3 の形へ変換する。本提案の変換により発生する問題点を、次の 2 つの単純な場合を挙げて考える。

1. 構造体を要素とする配列
2. 配列を含む構造体

その他の場合として、これらの構造体が組み合わさっている場合や、構造体が入れ子構造になっている場合等が考えられるが、構造体の定義の内部を変更するわけではないので、変換の方法は基本的に同じとなる。また、C 言語では共用体と呼ばれるものもあるが、宣言やアクセス方法等は構造体とまったく同じである。そのため共用体に関しては、構造体に対して提案方法の適用を行う場合と同様の方法で実現できる。

5.2.1 構造体を要素とする配列の変換

構造体を要素とする配列を、配列へのポインタに変換する。そして、ポインタに対してメモリを割り当てる。この変換以降、構造体のメンバを参照する場合や、演算子を使用する場合は、ポインタに対しての処理になる。そのため、次の問題が発生する。ポインタから構造体の配列のサイズの判断ができない。もし、そのままの状態で sizeof 演算子を使用すると、ポインタのサイズを返してしまう。sizeof 演算子の問題を解決するため、使用する構造体の配列と同じサイズの構造体の配列を代わりに用意する。しかし、他の演算子や配列の参照等で問題は発生しないため、構造体を要素とする配列の変換は、sizeof 演算子に対する変換だけで良い。

5.2.2 配列を含む構造体の変換

配列を含む構造体を、構造体へのポインタに変換する。そして、ポインタに対してメモリを割り当てる。この変換以降、構造体のメンバを参照する場合や、演算子を使用する場合は、ポインタに対しての処理になる。そのため、次の問題が発生する。構造体を要素とする配列の変換と同様で、ポインタに対して sizeof 演算子を使用した

```
struct test{
  char str;
};

struct test2{
  char str[5];
};

main(){
  struct test a[5];       /*構造体の配列*/
  struct test2 b={"xxxx"};/*配列を含む構造体*/

  …a[3].str…sizeof(a)…
  …b.str…sizeof(b)…
  return;
}
```

図 2　変換前のプログラム

```
struct test{
  char str;
};

struct test2{
  char str[5];
};

struct test *init(int n){
  struct test *tmp;
  tmp=(struct test*)
        malloc(sizeof(struct test)*n);
  return tmp;
}

struct test2 *init2(){
  struct test2 *tmp;
  struct test2 tmp2={"xxxx"};
  tmp=(struct test2*)
        malloc(sizeof(struct test2));
  memcpy(tmp,&tmp2,sizeof(tmp2));
  return tmp;
}

main(){
  struct test (*a)=init(5);
  struct test a2[5];
  struct test2 (*b)=init2();

  …a[3].str…sizeof(a2)…
  …(*b).str…sizeof((*b))…
  free(a);
  free(b);
  return;
}
```

図 3　変換後のプログラム

場合に構造体のサイズの判断ができない。さらに、構造体メンバの参照でも値が参照できなくなる。これらの問題を解決するため、図 3 のように構造体 b を (*b) で囲むことで、構造体の値を参照できるように変換する。以上のように、配列を含む構造体の変換では、sizeof 演算子だけでなく、プログラム中の全ての参照に対して変換しなければならない。

5.3 提案方法実現の検討

本提案方法は次のようにして実現を行う。まず始めにプリプロセス処理を行い、ヘッダファイルの読み込みとマクロの展開を行う。そして、プリプロセス処理後のファイルに対して構文解析を行う。そこから得られる情報を基にして変換を行うことで、提案方法が適用されたファイルが出力される。本提案方法の変換の手順を以下に示す。

1. 連続した宣言の分割
2. typedef の展開
3. 配列へのポインタに変換
4. 演算や参照に対しての変換
   (a) 配列　：sizeof() で代用する配列の宣言と変換
   (b) 非配列：配列を含む構造体の参照の変換
5. free() の追加
6. メモリの割り当て関数の用意

連続した宣言の分割では、1 つの型宣言に対して 2 つ以上の変数が宣言されている場合は、1 つの型宣言に対して 1 つの変数しか宣言されていないように変換する。このようにして、宣言を分割することで残りの変換を簡略化する。

次に、typedef の展開では、typefdef によって変更された型や型の配列などを元の名前に変換する。typedef の展開により、typedef が入れ子になっている場合などの複雑な場合を簡略化する。配列へのポインタに変換は、構造体を要素とする配列や配列を含む構造体の宣言をポインタの宣言に変換する。

演算や参照に対しての変換は、sizeof() で配列のサイズが判断できないため、代わりの配列を用意する。また、配列でない場合はポインタの値を参照するように変換する。

free() の追加では、関数の終了時にメモリの解放をする必要があるので、その処理を加える。最後に、メモリの割り当て関数の用意では、変換されたポインタに対してメモリの割り当てを行う関数を用意する。初期設定もこの関数内で行う。

6 実験及び評価

表 1 は Linux プログラムに対して提案方法による変換を試み、それぞれのプログラムの実行開始から終了までの時間を試算した結果を示す。

リターンアドレスの保護による対策方法では、関数ごとにコードを挿入しているので、実行時間の増加は挿入先プログラムの関数の実行回数に依存する。再帰処理等で、関数を何度も実行する処理を伴うプログラムでは、実行時間が大きく増加すると考えられる。

バッファの隔離による対策方法では、配列や構造体の宣言の数と、その使用頻度に実行時間が依存する。宣言の度にメモリの割り当てを行うため、メモリの割り当てによる実行時間の増加が考えられる。また、割り当てられたメモリへのアクセスはポインタを経由して行われる。そのためアクセス数が多いほど実行時間は増加する。

以上のことから提案方法によるプログラムの実行時間の増加がある程度存在する。そのため、プログラムの実行時の性能劣化を最小に抑えるための工夫についても検討しなければならないと考える。

表 1　提案方法適用による実行時間増加率

プログラム	リターンアドレスの保護	バッファの隔離
bison	4.87%	0.61%
flex	1.59%	3.98%
which	14.86%	3.15%

7 おわりに

本研究ではスタックオーバーフローのバグを悪用するコンピュータシステムの攻撃方法について述べ、その攻撃に対する対策方法を提案した。今後の課題として、本研究で提案した以外にも対策方法を提案し検討することと、実際に存在する攻撃コードをこれらの方法で防ぐことができるか調査することが挙げられる。

参考文献

[1] Bhatkar, S., “Efficient Techniques for Comprehensive Protection from Memory Error Exploits”, USENIX Security Symposium, Jul. 2005, pp.17-33

[2] Cowan, C., “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”, USENIX Security Symposium, Jan. 1998, pp.63-78

[3] Dahn, C., “Using Program Transformation to Secure C Programs Against Buffer Overflows”, Working Conference on Reverse Engineering, Apr. 2003, pp.323-332

[4] 江藤 博明, “Propolice: スタックスマッシング攻撃検出手法の改良”, コンピュータセキュリティ研究会, Jul. 2001, pp.181-188

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

[6] 脇田 建, “バッファ溢れ攻撃とその防御”, コンピュータソフトウェア, Vol.19, No.1, Jan. 2002, pp.49-63