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

2005－ＳＬＤＭ－119 (28)
2005／3／18

スタックフレームのセグメント化による バッファオーバーフロー対策法

蛭田 智則† 山名 早人‡

† 早稲田大学大学院理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1
‡ 早稲田大学理工学術院 〒169-8555 東京都新宿区大久保 3-4-1
E-mail: † tomo@yama.info.waseda.ac.jp, ‡ yamana@waseda.jp

あらまし 近年、バッファオーバーフローと呼ばれる脆弱性を利用した攻撃が増えつつある。バッファオーバーフローはプログラムの変数用に用意された領域に、領域の大きさを越えたデータが入力されることで発生する。バッファオーバーフローのなかで、最も危険なものがスタックオーバーフローである。スタックオーバーフローが発生すると、リターンアドレスが書き換えられ、悪意あるコードが実行可能になる。本論文ではスタックフレームのセグメント化によるスタックオーバーフローの解決手法を提案する。SimpleScalar ツールセット ver3.0d に本手法を実装し、SPEC CINT95 を用いて性能への影響を検証した。検証の結果、提案手法適用による性能低下率は平均 2.53%であった。

キーワード バッファオーバーフロー,スタック,スタックフレーム,セグメント

Defense against Buffer Overflow by Segmenting Stack Frame

Tomonori Hiruta† Hayato Yamana‡

† Graduate School of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjyuku-ku, Tokyo, 169-8555 Japan
‡ Science and Engineering, Waseda University 3-4-1 Okubo, Shinjyuku-ku, Tokyo, 169-8555 Japan
E-mail: † tomo@yama.info.waseda.ac.jp, ‡ yamana@waseda.jp

Abstract In recent years, Buffer overflow Attacks are increasing. Buffer overflow is caused by inputting larger data than date space which prepared for various numbers. The most danger buffer overflow is stack overflow. When Stack Overflow occurs, return address is re-written and malicious code becomes executable. This paper proposes defense technique against buffer overflow by segmenting stack frame. We implement this technique Simplescalar tool set ver3.0d and evaluate with SPEC CINT95. Evaluation result shows that this technique causes 2.3% performance degradation.

Keyword Buffer Overflow, Stack, Stack Frame, Segment

1. はじめに

近年、バッファオーバーフローと呼ばれる脆弱性を利用した攻撃が増えつつある。バッファオーバーフローを利用すると、システム全体を乗っ取ることも可能である。バッファオーバーフローを利用したウィルスの例として、2001年に発生したCode Red、2003年に発生したBlasterが挙げられる。これらのウィルスが世界的規模の被害をもたしたことは記憶に新しい。

図１にCERTにより報告されたセキュリティ警告数を示す。図１よりバッファオーバーフローを利用した攻撃は、近年では全体の約50%にのぼることがわかる。したがって、バッファオーバーフローを防ぐことはセキュリティ上、非常に重要であり、早急な対策が必要であると言える。

バッファオーバーフローの中で最も危険とされているのがスタックオーバーフローである。スタックオーバーフローが発生すると、リターンアドレスが書き換えられ、任意のコードが実行可能な状態になる。攻撃者は、この状況を意図的に作り出し、システムへの攻撃を試みる。

スタックオーバーフローに対処するには、それが既知の場合、プログラムのアップデート等をすることで解決することができる。しかし、未知のスタックオーバーフローには対応できない。このため、ソフトウェアやハードウェアを用いて、バッファオーバーフローを検出したり、リターンアドレスの改竄を防止したりする研究がなされている。しかし、ソフトウェアによる手法は、スタックオーバーフローに対して柔軟に対応することができるが、プログラムに何らかの手を加えるため、ソースコードが増大する。また、使用時の性能低下率が比較的大きい。ハードウェアによる手法では、ソフトウェアに見られる問題点は改善されてい

るが、リターンアドレスのみをスタックオーバーフロー発生検出の対象としているため、スタックオーバーフローに対して十分ではないと言える。そこで、本稿では、ハードウェアによる手法に見られる問題点を解決するために、スタックフレームのセグメント化によるバッファオーバーフロー対策法を提案する。本手法はハードウェアを用いたスタックオーバーフロー発生防止法であり、改竄防止の対象をリターンアドレス以外にも広げているため、従来手法よりも精度向上を見込むことができる。

以下、第2章でバッファオーバーフローについて述べ、第3章で関連研究について述べる。第4章で提案手法について述べ、第5章で評価実験について述べる。最後に第6章でまとめとする。

表1 CERTによる脆弱性警告数

Year	Advisories	Advisories involving buffer overflow	Percent buffer overflow
1999	17	8	47.06%
2000	22	2	9.09%
2001	37	19	51.35%
2002	37	20	54.05%
2003	28	19	67.86%

2. バッファオーバーフロー

バッファオーバーフローはプログラムの変数用に用意された領域に、領域の大きさを越えたデータが入力されることで発生する。バッファオーバーフローにはヒープオーバーフロー、スタックオーバーフロー等があるが、本論文では、最も危険とされているスタックオーバーフローを対象とする。

2.1 スタック

スタックはLIFOデータ構造のメモリ領域であり、手続き呼び出しとリターンの間の状態を保持するために用いられる。図1に示すように、スタックは高位アドレスから低位アドレスに成長する。スタックはフレームポインタ(fp)とスタックポインタ(sp)で示されるスタックフレームで構成される。fpは実行中の手続きのスタックフレームのベースを指し示し、spは実行中の手続きのスタックフレームのトップを指す。データがスタックにプッシュ、もしくはスタックからポップされた場合、spはそれに応じて変化する。手続きが新たに呼ばれた場合、現在のスタックフレームのfpは新しい手続きのスタックフレーム内に保持される。

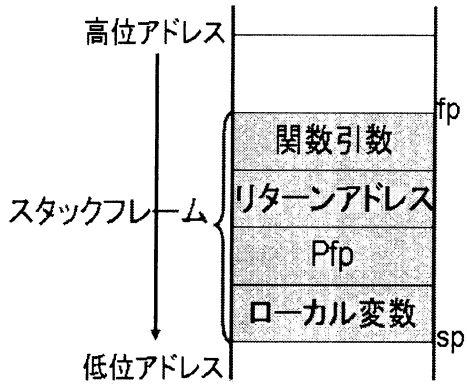

図1 スタック

2.2 スタックオーバーフロー(stack smashing)

図2に、図3のプログラムにおけるバッファオーバーフロー攻撃を示す。図2中の手続きstrcpy()は入力データの領域チェックを行なわないため、バッファオーバーフローを引き起こす可能性がある。関数gにおいて、変数s1が指す文字列が、変数bufのサイズを超える場合、手続きstrcpy()はbufの領域を超えてデータを上書きする。攻撃者はこの性質を利用する。s1が悪意あるコードと改竄したリターンアドレスを含み、bufより大きなサイズの文字列であった場合、strcpy()は悪意あるコードをスタックにコピーし、手続きgのスタックフレームにある手続きfのリターンアドレスを悪意あるコードの最初の命令のアドレスに上書する。その結果、手続きgの終了後、プログラムは手続きfに戻らず、悪意あるコードの先頭にジャンプし、悪意あるコードを実行することになる。

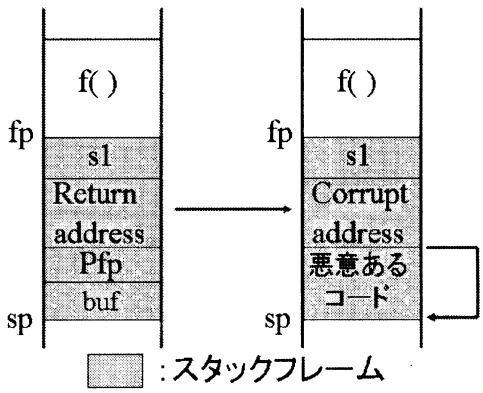

図2 スタックオーバーフロー

```
f(){
        ---
    g(s1);
        ---
}
g(char *s1){
    int a;
    char buf[100];
        ---
    strcpy(buf,s1);
        ---
}
```

図 2　サンプルプログラム

3. 関連研究

スタックオーバーフロー問題の解決手法はソフトウェアによる手法と、ハードウェアによる手法の2つに分けることができる。ソフトウェアによる手法は、さらに、静的な手法と動的な手法に2つに分類できる。従来は、ソフトウェアによる手法の提案が多かったが、近年になってハードウェアを用いた手法が提案されてきている。

3.1 ソフトウェアによる手法

3.1.1 コンパイラによる静的手法

・StackGuard[2]

Cowan らは 1998 年に StackGuard を提案している。StackGuard はコンパイラベースの解決手法であり、gcc のパッチという形式で提供されている。StackGuard はスタックオーバーフローが発生しリターンアドレスが書き換えられた場合、リターンアドレスが格納されているメモリの直前のメモリに格納にされている値も書き換えられるという性質を利用している。

StackGuard を用いてコンパイルされたプログラムは、手続き呼び出し時に、図 4 に示すように、リターンアドレスの前に Canary と呼ばれる乱数を挿入し、手続き終了時に Canary の値を比較する。Canary の値が一致しなかった場合、プログラムは実行を停止する。

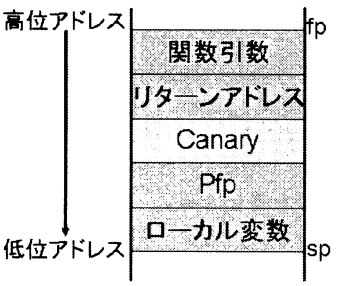

図 4 StackGuard

3.1.1 ライブラリによる動的手法

・libsafe[3]

Baratloo らは 2000 年に libsafe を提案している。libsafe はライブラリベースの解決手法である。libsafe に登録されている関数は、バッファオーバーフローを引き起こす可能性がある標準関数に、配列の境界チェックなどを施した安全な関数である。

libsafe を用いた場合、プログラムは実行開始時に、libsafe に動的にリンクされる。標準関数が呼ばれる場合、通常の標準ライブラリからではなく、安全な libsafe から呼び出される。

3.2 ハードウェアによる手法

・SRAS[4]

Lee らは 2003 年に SRAS(Secure Return Address Stack) を提案している。SRAS は HW スタックを使用する。SRAS では手続き呼び出し時に、リターンアドレスが HW スタックのトップに保存され、手続き終了時に、HW スタックのトップからリターンアドレスが取り出され、コピー元のリターンアドレスと比較される。SRAS はリターンアドレスの保存に HW スタックを用いているため、LIFO と異なる動作には対応が難しい。

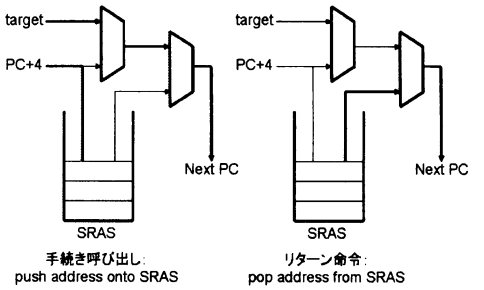

図 5 SRAS

・Scache[5][6]

井上は 2004 年に SCache を提案している。SCache はキャッシュ上のラインにリターンアドレスのコピーを保持するレプリカラインを用いてリターンアドレスの改ざんを防ぐものである。これは、手続き呼び出し時にはスタック領域にプッシュされるリターンアドレスがキャッシュに一旦ストアされるという性質、手続き終了時にはリターンアドレスがキャッシュからロードされるという性質を利用している。

SCache は手続き呼び出し時にレプリカラインにリターンアドレスのコピーを保存し、手続き終了時にコピー元のリターンアドレスとレプリカラインからのリターンアドレスを比較する。リターンアドレスが一致しなかった場合、その旨をプロセッサに通知しリターンアドレスの改ざんを防ぐ。ただし、全てのレプリカラインがキャッシュから追い出されていた場合、リターンアドレスの改ざんを検出することが出来ない。

4. 提案手法

4.1 従来手法の問題点

これまでの提案手法の多くはリターンアドレスの

みを対象としてきた。しかし、実際の攻撃では、リターンアドレスだけでなく、他の手続き情報やローカル変数も攻撃の対象となっている。したがって、これまでの手法はスタックオーバーフローに対して十分でないと言える。

4.2 提案手法

本論文では、スタックフレームのセグメント化によるバッファオーバーフロー対策法を提案する。本手法は、スタックフレームを手続き情報領域とローカル変数領域の2つセグメントに分割し、手続き情報領域への書き込みに制限を加えることでリターンアドレスの改竄を防ぐ。

本手法では、スタックフレームを2つのセグメントに分割するための値（リミットアドレス）を保存する必要が生じるが、その保 存法が重要になる。SRAS では HW スタックを使用したために、LIFO と異なる動作への対応が困難になっている。本手法では、リミットアドレスがどのスタックフレームに対応しているか示すために、fp をリミットアドレスと供に保存する。

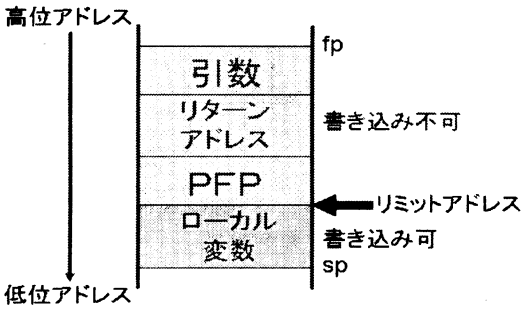

図 6 提案手法

4.3 動作

本手法の動作の概要を図4に示す。図7（1）に示すように、fpとリミットアドレスの保存に関する動作は、HW スタックと同様である。

図7の(2)は書き込み命令実行時の動作例について示したものである。書き込み命令の場合、始めに、書き込み先アドレスが属するスタッククフレームの探索を行う。探索には線形探索を用いる。アドレスがスタックフレームに属しているかどうかは、書き込み先アドレスと FP の比較で判断する。書き込み先アドレスがFPよりも小さい場合、書き込み先アドレスはスタックフレームに属していると判断し、探索を終了する。そうでない場合、書き込み先アドレスはスタックフレームに属していないと判断し、探索を継続する。図4の(2)では2回目の探索で、属するスタックフレームを発見できる。属するスタックフレームを発見できたならば、書き込み先アドレスとリミットアドレスを比較する。書き込み先アドレスがリミットアドレスより小さい場合、命令はローカル変数領域への書き込みであるため、書き込みを許可する。書き込み先アドレスがリミットアドレスより大きい場合、命令は手続き情報領域への書き込みであるため、書き込みを中止する。図7の(2)では書き込み先アドレスがリミットアドレスよりも小さいため書き込みを実行する。

4.4 オーバーヘッド

本手法では探索と比較を行うため、オーバーヘッドが生じる。探索には線形探索を用いるため、探索によるオーバーヘッドは n サイクルとする。また、比較によるオーバーヘッドは 1 サイクルとする。したがって、書き込み命令時のオーバーヘッドは n+1 サイクルである。

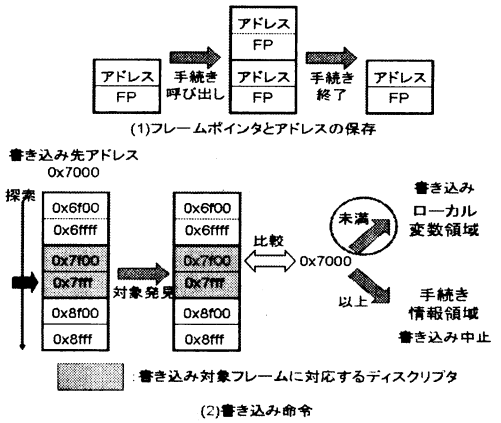

図 7 動作

5. 評価実験

5.1 実験環境

提案手法による性能低下を検証するため、SimpleScalar ツールセット ver.3.0d[7]の sim-outorder を用いて提案手法を実装した。sim-outorder はアウト・オブ・オーダー実行可能なサイクルレベルプロセッサシミュレータであり、提案手法実装時の性能への影響を計測することができる。本実験における性能低下の指標は IPC である。ベンチマークプログラムとして SPEC CINT95（ train 入力セット）を使用した。実験環境のまとめを表2に示す。

表 2 実験環境

パイプライン段数	5
フェッチ幅	4
デコード幅	4
発行幅	4
コミット幅	4
RUUサイズ	16
LSQサイズ	8

プログラム	入力セット
099.go	50 9 2stone.in
124.m88ksim	Ctl.raw
129.compress	100 q 2131
130.li	train.lsp
132.ijpeg	vigo.ppm
134.perl	primes.in

5.2 実験結果

5.2.1 手続きの深さ

図 8 は各プログラムの手続きの深さを示したものである。手続きの深さは 130.li で最大 634 である。その他のプログラムでは 128 に満たない。

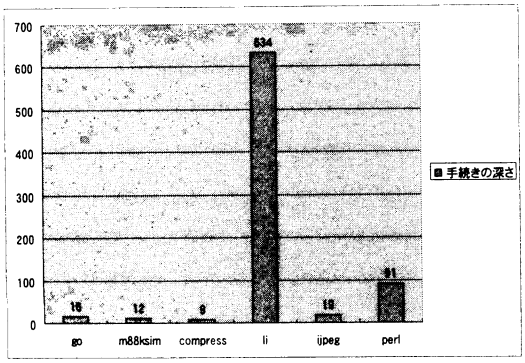

図 8　手続きの深さ

5.2.2 性能低下率

表 3 に各プログラムの性能低下率を示す。性能低下率は 132.ijpeg で最大となり、低下率は 5.3%である。

また、性能低下率は平均 2.53％である。これはソフトウェアによる手法に比べて小さな値である。

表 3　性能低下率

プログラム	適用前IPC	適用後IPC	低下率
0.99go	0.8861	0.8861	0%
124.mk88sim	1.3751	1.3517	0.4%
129.compress	1.8361	1.7518	4.6%
130.li	1.5149	1.4594	3.7%
132.ijpeg	1.9979	1.8930	5.3%
134.perl	1.0657	1.0501	1.5%

5.2.3 考察

評価実験では、手続きの深さと IPC の低下率に関連性が見られない。手続きの深さが比較的多い 134.perl の IPC 低下率は 1.5%である。このことから、探索によるオーバーヘッドは比較的小さいといえる。したがって、本手法のオーバーヘッドは、専ら、書き込み命令実行数に依存すると言える。

また、評価実験により、本手法適用による性能低下は平均 2.53%であることがわかった。この結果はソフトウェアによる手法に比べて十分に小さいと言える。しかし、ハードウェアによる手法に比べると大きなものである。本手法は書き込み命令時に、オーバーヘッドがかかるため性能低下率が大きくなる傾向がある。しかし、本手法は、リミットアドレスを調整することで、リターンアドレスだけでなく、FP 等の他の手続き情報をも保護することが可能である。これは、SRAS や SCache に対して大きなアドバンテージである。

6 終わりに

本稿では、スタックフレームのセグメント化によるスタックオーバーフロー対策法を提案した。提案手法では、スタックフレームをローカル変数領域と手続き情報領域に分離し、手続き情報領域への書き込みに制限を加えることでリターンアドレスの改竄を防ぐ。SimpleScalar ツールセット ver.3.0d の sim-outorder を用いて提案手法を実装し、評価実験を行った。評価実験の結果、本手法の適応による性能への影響は平均 2.53%であった。この結果はソフトウェアによる手法に比べて十分に小さいと言える。

今後は、提案手法のオーバーヘッドが小さくなるように提案手法の改良を行っていく予定である。

謝辞

本研究の一部は、21 世紀 COE プログラム「プロダクティブ ICT アカデミア」によるものである。

参考文献

[1]CERT Advisories:http://www.cert.org/advisories

[2]Cripson Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaran Grier, Perry Wagle and Qian Zhang: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, Proceedings of the 7th USENIX Security Symposium.pp.63-78,1998

[3]Arash Baratloo, Navjot Singh and Timothy Tsai: Transparent Run-Time Defense Against Stack Smashing Attacks, Proceedings of the USENIX Security Annual Technical Conference, 2000

[4]John P. McGregor, David K. Karig, Zhijie Shi and Ruby B. Lee: A Processor Architecture Defense against Buffer Overflow Attacks, Proceedings of IEEE International Conference on Information Technology: Research and Education, pp.243-250, 2003

[5]井上弘士:バッファ・オーバーフロー・アタックを動的に検出するセキュア・キャッシュ-安全性と消費エネルギーのトレードオフ-,先進的計算基盤システムシンポジウム SACSIS 2004 ,pp.315-323,2004

[6]井上弘士:不正プログラムの実行防止を目的とするオンチップ・キャッシュ・アーキテクチャ,情報処理学会研究報告,ARC-159,pp.121-126,2004

[7]D. Burger and T. M. Austin: The SimpleScalar Tool Set, Version 2.0, University of Wisconsin-Madison Computer Sciences Department Technical Report no.1342, 1997