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

2005－DPS－122（15）
2005－CSEC－28（15）
2005／3／22

データ改ざん検出によるバッファオーバフロー検知システムの提案

長野 文昭 ∗　　鑪 講平 †　　田端 利宏 ‡　　櫻井 幸一 ‡

∗ 九州大学工学部電気情報工学科　　† 九州大学大学院システム情報科学府
‡ 九州大学大学院システム情報科学研究院　　812-8581 福岡市東区箱崎 6-10-1
{nagano,tatara}@itslab.csce.kyushu-u.ac.jp
{tabata,sakurai}@csce.kyushu-u.ac.jp

あらまし 今までにさまざまなバッファオーバフロー防御システムが提案されている．それらの方法の多くは，バッファオーバフロー発生の有無の検知を行っており，変数の完全性の検証を行わない．しかしバッファオーバフロー発生の有無の検知だけでは防ぎきれず，変数の完全性の検証が必要となる攻撃が存在する．そこで本論文では，変数の完全性を保証するシステムを提案する．既存のバッファオーバフロー防御システムでは，ユーザメモリ上から変数を読み取られると，防御システムを回避される危険があるものもあるが，本システムでは，そのような危険性はない．また，本方式はバッファオーバフローにより影響を受けた変数を，影響を受ける前に復元することもできる．

A Proposal of a System for Detecting Buffer Overflow with Detecting Alteration of Data

Fumiaki NAGANO∗　　Kohei TATARA†　　Toshihiro TABATA‡
Kouichi SAKURAI‡

∗ Department of Electrical Engineering and Computer Science,School of Engineering, Kyushu University
‡Faculty of Information Science and Electrical Engineering, Khushu University

†Graduate School of Information Science and Electrical Engineering, Khushu University
6-10-1 Hakozaki, Higashiku Fukuoka,812-8581
{nagano,tatara}@itslab.csce.kyushu-u.ac.jp
{tabata,sakurai}@csce.kyushu-u.ac.jp

Abstract Numerous security technologies which detect buffer overflow have already proposed. Almost these technologies detect if buffer overflows happen or not, but don't detect alteration of variable integrity. But there are attacks which are not be able to be defenced unless the technology detect alteration of variable integrity. So in this paper, we propose a system which detect alteration of variable integrity. Some exinting technologies could be bypassed if the attacker can see the user memory, but our proposed system can't be bypassed even if the attacker can see the user memory. And our proposed system can restore data which is altered by attackers using buffer overflow.

1　はじめに

近年，バッファオーバフローの脆弱性を利用した攻撃が蔓延している．ICAT の調査報告によると，ここ 4 年間で発見された脆弱性のうち 20%がバッファオーバフローに関する脆弱性である．バッファオーバフローを利用すると，プログラム内の関数の戻りアドレスやポインタ変数を書き換えることで悪意のある攻撃者が任意のコードを実行することが可能である．

バッファオーバフローの脆弱性は C や C++によって書かれた実行コードに多く存在する．しかし，C や C++によって書かれた実行コードはすばやく動作

し，C や C++の知識が広く普及しているため，今後も幅広く使用されることが予想される．よって，バッファオーバフローを利用した侵入行為の検出手法が必須である．

2　既存のシステムの問題点

バッファオーバフロー防御手法はこれまでにもさまざまな手法が提案されている．既存のバッファオーバフロー防御手法の一つに，カナリアを使用してバッファオーバフローの発生を検出する手法がある [1]．カナリアとは保護したい値の前書き込まれる数値のことで，保護したい値が書き換えられる場合にはカナリアも変更されることを利用してバッファオーバフロー発生の有無を検出する方法である．カナリアを用いた方法では，ユーザメモリ上に存在する変数を読み取られると，攻撃者の意図した攻撃が可能となる．これは，バッファオーバフロー時に，ユーザメモリ上から読み取った変数を用いてカナリアの値を解読し，そのカナリアの値を用いてカナリアを上書きすることにより可能となる [3]．他にも，ポインタ変数を使用するとカナリアを壊すことなく攻撃者による任意の攻撃が可能になってしまう [3]．これらは，被検証データの完全性を保証しないことに起因する．

そこで，本論文では変数の完全性検証のためのデータを攻撃者に読み取られたとしても防御システムを回避できない，変数の完全性保証手法を提案する．

3　提案手法

バッファオーバフローが発生する前に，戻りアドレスや，関数ポインタなどプログラムの制御が奪われる原因となる脆弱な変数について検証子と呼ばれるものを作成し，バッファオーバフローが発生した可能性がある後に検証子の検証を行うことでデータの改ざんの検知を行う．また，本方式は改ざんが検出された変数を改ざん前に戻し，バッファオーバフローの発生による影響を無効化することも可能である．以下，この方式において使用する検証子の構造について説明し，その後，その検証方法について説明する．

3.1　検証子

検証に使用するデータ構造として，図 1 のような検証子構造体配列 (VSA:Verifier Structure Array) を採用する [4]．

VSA は検証子構造体 (VS:Verifier Structure) を要素とする配列である．VS はカーネル空間またはユーザ空間に存在する．カーネルに存在する VS の要素は，検証データ，変更フラグ，検証対象データ長，検証対象ポインタからなる．ユーザ空間に存在する VS は，検証データ，変更フラグ，データリンクフラグ，検証対象データ長，検証対象ポインタからなる．これは，カーネルに存在する検証子構造体にデータリンクフラグを追加したものである．

以下，それぞれの要素が格納している値について説明する．

カーネル上の検証子構造体の要素

- 検証データ
  検証対象ポインタが指しているデータのハッシュ値
- 変更フラグ
  データ完全性のチェック時にデータの改ざんが検出されるとセットされるフラグ
- 検証対象データ長
  検証するデータの長さ
- 検証対象ポインタ
  検証するデータへのアドレス

ユーザメモリ上の検証子構造体の要素

- 検証データ
  検証対象ポインタが指しているデータのハッシュ値，もしくは検証対象ポインタが指している実データ
- 変更フラグ
  カーネル上の検証子構造体と同様
- データリンクフラグ
  検証対象ポインタが指しているデータが VSA であるか，ハッシュ値であるか，実データであるかを区別するフラグ
- 検証対象データ長
  カーネル上の検証子構造体と同様
- 検証対象ポインタ
  カーネル上の検証子構造体と同様

カーネルメモリ上にデータリンクフラグがないのは，カーネルの VS の検証対象ポインタは VSA を指しており，カーネルの VS の検証データは必ずハッシュ値となるからである．カーネルの VS の検証データがハッシュ値を格納している理由はユーザメモリに存在する VSA は大きいため，検証データにデータそのものは使用できないからである．

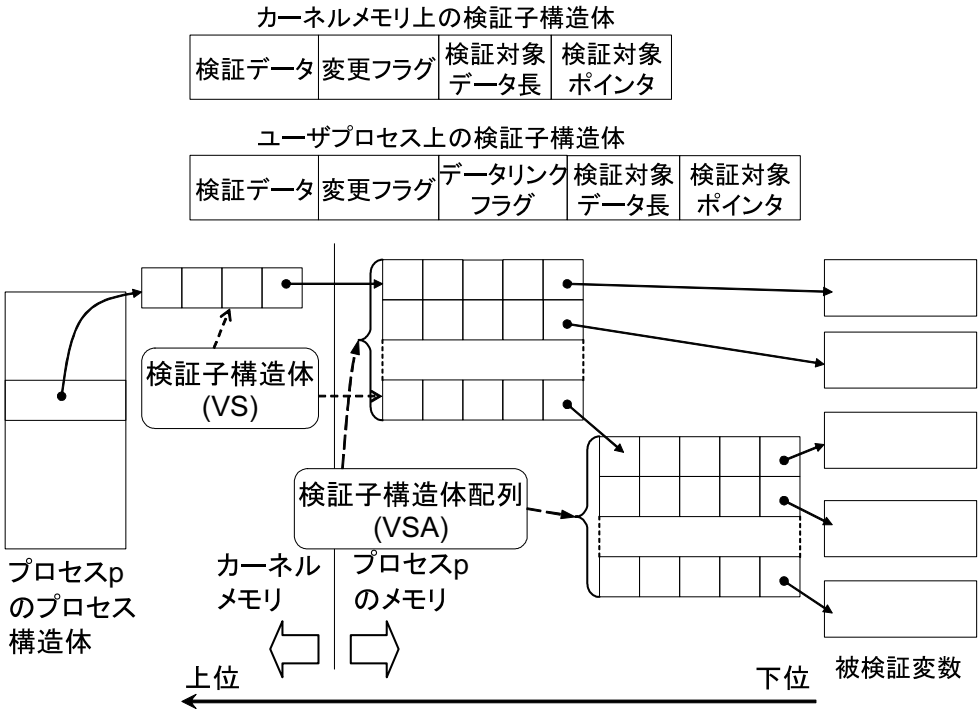

図 1: 検証子のデータ構造

3.2 検証方法

次に，実際にどのようなアルゴリズムを用いて，上記で述べたデータ構造を作成し，そのデータ構造を利用するのかを述べる．まず，データ構造の作成方法を述べ，その後そのデータ構造の利用方法を述べる．ここで，説明の都合上カーネルメモリ側を上位，ユーザメモリ側を下位と呼ぶ (図 1 では，左が上位，右が下位となる)．

検証子作成方法

(1) バッファオーバフローが発生する可能性がある前に，プログラムの制御が奪われる原因となる脆弱な変数について，最下位の VSA の要素 VS の作成を行う．すなわち，VS の要素，検証対象ポインタ，検証対象データ長，データリンクフラグ，検証データの値を作成する．

(2) 作成が終わったら，作成した VSA をユーザメモリ領域に格納する．この際，プロセスのヒープ領域に格納するものとする．

(3) (1)，(2) の動作を全ての被検証データに対して VS を作成するまで繰り返す．

(4) ユーザ領域の VSA の作成が終了したら，カーネルに存在させる最上位の VSA 作成を行う．すなわち，VS の要素，検証対象ポインタ，検証対象データ長，検証データの値を作成する．

(5) 作成が終了したら，作成した VS をカーネルメモリ空間に格納する．

検証方法

(1) バッファオーバフロー発生の検証を行いたい時に検証子のチェックを行う．まず，最上位の VSA の要素 VS 全てについて検証データの作成と VS に格納されている検証データとの比較を行う．ここでは，下位の VSA のハッシュ値と最上位の VS の検証データの要素に格納されている値を比較することとなる．

(2) 検証データの比較の際に，値が同じであったらその下位の VSA についても，その要素 VS の検証データの検証を同様に行い，最下位に至るまで同様のことを行う．検証データ比較の際に，値が異なっていたらその VS の変更フラグをセットする．

上記のように検証子の作成，検証を行うことにより，下位の VSA は上位の VSA により検証が行われるため，たとえ改ざんがあったとしても検知することができる．このことはつまり，たとえユーザメモリ上のデータを読み取られ，プロセスの変数やメタデータを書きかえられても，その改ざんを検知できることを意味する．また，最上位の VSA はカーネルメモリに置くことで，特権プロセスであっても操作不可能にすることで安全に格納している．

最上位以外の VSA については改ざんが検知できるためカーネルメモリに格納する必要はなく，またカーネルメモリの節約，システムコールの呼び出しの節約のため置くべきでない．

データ構造利用方法

上記のアルゴリズムが改ざんを検知した時にはユーザアプリケーションに対してシグナルが送られる．そのときの処理はユーザアプリケーションのポリシ次第である．シグナルが送られた際，改ざんが検知された VS のデータリンクフラグによりその VS の検証データの欄に検証対象データがハッシュ値ではなく，実データが入っている場合にはその検証データを使用することで，データを改ざん前に復元することも可能である．これにより，バッファオーバフローを利用した攻撃の影響を無効化することも可能となる．

4 実装方式

本方式を実装するには，カーネル領域に VSA を作成するためにカーネル領域にアクセスすることや，プログラムの実行中に検証子を作成することが必要である．よって，本方式を実装するには以下の 4 点が必要である．

(1) カーネルに VSA を登録するシステムコールの実装
(2) カーネルに登録した VSA の完全性を検証するシステムコールの実装
(3) ユーザ領域に上記の検証子を作成する関数の実装
(4) ユーザ領域の検証子を検証し削除する関数の実装

(1)，(2) は OS に新たにシステムコールを追加することで可能となる．また，(3)，(4) はコンパイラを改良することで可能となる．このように行うことにより，ユーザが望む位置で任意のデータの完全性を検証することが可能となる．なぜならユーザが望む位置で上記の検証子を作成，検証することができるためである．

また，どの変数に対して検証子を作成するかについては，システムに多大な攻撃を与える変数はポインタである [5] ことを考えて，ポインタに対して検証子を作成するようにコンパイラを改良する．ただし，ユーザがポインタ以外の変数の完全性を保証した場合は，あらたに命令を加えることでその変数に対しても検証子を作成，検証することができる．いつ，検証子を作成するかは，ポインタに対して入力があった直後に検証子を作成し，ポインタを使用するときに検証子の検証を行う．これにより，ポインタに対して入力があった時のポインタの値の完全性が保証されることとなる．

5 他方式との比較

この節では，提案手法と関連研究との比較を行う．表 1 は，提案手法と既存の手法との比較をまとめた表である．以下，それぞれの要素について見ていく．

5.1 境界チェック

表 1 の境界チェックとは，バッファの境界をチェックすることでバッファオーバフローを防ぐ既存の方式である．新しい言語を用いることでバッファオーバフローを利用した攻撃の発生を防いでいる．しかし，この方式を既存のプログラムに適用させるには，プログラムコードを一から書き直さないといけない．また，これらの方式はバッファオーバフローの発生を検知することはできるが，バッファオーバフローにより影響を受けた変数の復元は一切できない．

5.2 ヒープやスタック領域でのコードの実行禁止

表 1 の実行不可能とは，メモリ上のある領域のコードを実行不可能にすることでバッファオーバフローの発生による悪意のあるコードの実行を不可能にしようという方式である．既存の方式として，スタック上のコードを実行不可能にするという方式，ヒープ上のコードを実行不可能にするという方式がある [6]．この方式はコストは比較的少ないが，このバッファオーバフロー防御システムを回避可能であるという欠点がある．再帰的なプログラムを実行す

表 1: 提案手法と関連研究との比較

クラス	技術	範囲	回避	コスト	変数復元
境界チェック	新しい言語	全て	不可能	コード書き直し	不可能
実行不可能	スタック実行不可	スタック上	可能	0	不可能
	ヒープ実行不可	ヒープ上	可能	0.1〜0.3 倍	不可能
ポインタ保護	StackGuard	戻りアドレス	可能	〜0	不可能
	Libsafe	ライブラリ	可能	〜0	不可能
	PointGuard	ポインタ	可能	0〜0.2 倍	不可能
提案手法	完全性チェック	全て	不可能	後述	可能

る場合には適用できないという欠点もある [2]．また，バッファオーバフローにより影響を受けた変数の復元は一切できない．

5.3 ポインタ保護

表 1 のポインタ保護とは，ポインタを保護することで，バッファオーバフローによる攻撃からシステムを保護する方式である．StackGuard[7] はカナリアを使用することで，Libsafe[8] はライブラリを変更することで，PointGuard[5] はポインタを暗号化することで，それぞれポインタを保護し，バッファオーバフローによる攻撃の発生を防いでいる．オーバヘッドは比較的少ないが，これらの方式を使用することで防御する変数以外にも，バッファオーバフローを利用した攻撃時に使われる変数が存在する事があるため，このシステムは回避可能である．また，これらの方式もバッファオーバフローにより影響を受けた変数の復元はできない．

5.4 提案手法の利点と欠点

以下，提案手法の特徴を述べる．提案手法は，変数の完全性チェックを行うことでバッファオーバフローによる攻撃を防いでいる．この手法は全てのばっふぁオーバフローを利用した攻撃を防いでいる．なぜなら，VS の検証対象データの欄にプログラムの制御が奪われる原因となる変数を指定することができるためである．また，この手法は回避不可能である．最上位の VSA をカーネルに置くことによりユーザプログラムは最上位の VSA にアクセスできず，最上位の VSA の完全性が保証できる．そして，その完全性が保証された最上位の VSA が下位の VSA の完全性を保証することができる．このようにすることにより，たとえメタデータに対する攻撃でも検出することができるためである．この方式は，VS の検証子の欄に，検証対象データの実データを置くことで，バッファオーバフローにより改ざんされた元の値を復元できる．つまり，この機構を利用することで，バッファオーバフローを利用した攻撃の影響を無効化することができる．

オーバヘッドは，検証子を一回作成し，検証するのに必要な時間は 23.1 μ秒であった．これはオーバヘッドが大きい．そこで，ユーザメモリ上の VSA を保存する場合にヒープを使用しない方式と最上位の VSA を保存する際にシステムコールを利用しない方式についてもオーバヘッドを測定を行った．(CPU Intel(R) Pentium(R) III 1GHz，メモリ 512MB の環境で実験) 結果は表 2 のようになった．h ○はヒープを利用した方式，h ×はヒープを利用しなかった方式，k ○はシステムコールを使用した方式，k ×はシステムコールを利用しなかった方式を示している．単位は 1 マイクロ秒であり，20 回実験を行ったものの平均値を記す．

表 2: 検証子の保存領域の違いによるオーバヘッドの違い

h ○ k ○	h ○ k ×	h × k ○	h × k ×
23.1	20.4	5.00	3.01

表 2 からわかるように，ヒープを利用せず，システムコールを利用しないとオーバヘッドがかなり削減できることがわかる．そこで，オーバヘッド削減のためには，ヒープ，システムコールを使用しなければいいが，ヒープを使用しないということは新たなメモリ領域を確保することができないという欠点が発生する．しかし，あらかじめ適当にメモリを確

保しておき，それでもメモリがなくなった場合に限りヒープ領域を利用することでこの問題を解決できる．また，システムコールを利用しないと，ユーザメモリ上の最上位の VSA の値を利用してこのシステムを回避されてしまうという問題が発生してしまう．しかし，ユーザメモリ上の値を読まれる心配あるときに限りシステムコールを利用することでこの問題を解決することができる．

6 まとめ

本論文では，既存のバッファオーバフロー防御システムの問題を解決するために，変数の完全性を保証することでバッファオーバフローの発生を検知するバッファオーバフロー防御システムを提案した．本システムは，バッファオーバフローにより影響をうけた変数の復元も可能である．

今後の課題としては，本方式の実装が挙げられる．また，さらなるオーバヘッドの削減も実現する必要がある．

参考文献

[1] Andre Zuquete. STACKFENCES: A RUNTIME APPROACH FOR DETECTING STACK OVERFLOWS. Proceedings of International Conference on E-Business and Telecommunication Networks(ICETE), pp. 76–84, Aug 2004.

[2] 情報処理進行事業協会セキュリティセンター. オープンソースソフトウェアのセキュリティ確保に関する調査. http://www.ipa.go.jp/security/fy14/reports/oss_security/.

[3] Greg Hoglund, Gary McGraw, トップスタジオ訳. セキュアソフトウェア. 日経 BP 社, 2004.

[4] 江頭徹, 稲村雄, 竹下敦. マルチタスク OS におけるメモリの保護-データ改ざん検出手法の提案-. 電子情報通信学会技術研究報告 (ISEC), pp. 27–34, Jun 2004.

[5] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. Proceedings of 12th USENIX Security Symposium, pp. 91–104, 2003.

[6] Openwall Project. Non-Executable User Stack. http://www.openwall.com/linux/.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proceedings of 7th USENIX Security Symposium, pp. 63–78, 1998.

[8] AVAYA. Avaya labs research. http://www.research.avayalabs.com/project/libsafe/.