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

2005－OS－98 (9)
2005／2／23

プログラム注釈に基づくバッファオーバーフローの検知方法

行友 英記* 金野 晃* 中山 雄大* 竹下 敦*

従来，プログラムの外部から派生した値を格納するメモリ領域を追跡/記録し，この領域への実行遷移を検知することでバッファオーバーフローの発生を検知する方法が知られている．
しかしながら，動的にコードを生成するプログラムに対してはそのまま適用できなかった．
そこで本稿では，動的にコードを生成するプログラムにおいても適用可能なバッファオーバーフローを検知する方法として，外部から派生した値を格納することがあってはならない領域の指定をプログラム作成時に注釈として与え，実行時，この注釈が与えられた領域に外部から派生した値が格納された時に，バッファオーバーフローの発生を検知する方法と，その実現案を示す．

A Buffer Overflow Detection Method based on Program Annotation

Hideki Yukitomo* Akira Kinno* Takehiro Nakayama* Atsushi Takeshita*

Previously, a buffer overflow detection method was proposed which tracks the memory areas affected by input channels from the outside of the program, and disallows such memory addresses to be executed. However, that is not applicable to programs that dynamically generate program code.

This presentation shows a new buffer overflow detection method and implementation idea applicable to programs that dynamically generate program code. The proposed method detects buffer overflows at runtime based on preliminarily added annotations on the source code of the program. The annotation indicates the memory areas that are not supposed to be affected by input channels from the outside of the program.

1. はじめに

C言語は高級言語でありながらポインタ演算等の機能を持つことから，ハードに密接した低レベルの処理を効率よく記述可能である．そのため，パーソナルコンピュータのような汎用計算機だけでなく，携帯電話やセットトップボックス等の組み込み機器等においても，動作速度が重要視される場合，ソフトウェアの記述に広く利用されている．その反面，これら低レベルの処理を効率よく記述ができる特徴は，注意深く利用しないとバグやプログラムの脆弱性の原因となりえるという問題点も含んでいる **[1]**.

なかでもバッファオーバーフローと呼ばれるプログラムの脆弱性を利用して計算機プログラムの実行を乗っ取る方法により，Web サイトの運営を妨害したり，個人情報を盗んだりする行為が昨今問題となっている**[2,3]**.

このバッファオーバーフローの発生を検知し，攻撃者による任意コードの実行を防止するための数々の研究が行われている**[4,5,6,7,8,9,10]**.

例えば Windows XP Service Pack2 の Data Execution Prevention(DEP)**[8]**や，Linux Kernel の ExecShield**[9]**等は，データを格納するスタック領域やヒープ領域におかれた内容は，プログラムコードとして実行できないよう，仮想ページエントリの情報やセグメント情報を工夫している．

また，Sue らは，プログラムの仮想アドレス空間上の各メモリアドレスが，プログラムの外部から派生した値を格納しているか否かを追跡/記録しておき，外部から派生した値を格納するメモリアドレスに実行遷移しようとすると，攻撃者から送り込まれた侵入コードであるとして，検知する方法を提案している**[10]**.

しかしながら，JIT 搭載の JVM やスクリプト言語等，動的にコードを生成するプログラムの場合，新たに生成したコードを書き込むメモリ領域に対しても実行権限を与える必要があるため，先の DEP**[8]**や ExecShield**[9]**は利用できない．

また，Sue らの方法**[10]**を動的にコード生成するプログラムに適用する場合，動的に生成されたコード自体が外部から派生した値と認識されることから，動的に生成されたコードを格納する領域はバッファオーバーフロー検知の対象から外してやらねばならない．この時，プログラム中のポインタ変数への攻撃が可能であるなら，攻撃者はバッ

* (株)NTT ドコモ マルチメディア研究所
Multimedia Labs, NTT DoCoMo, Inc.

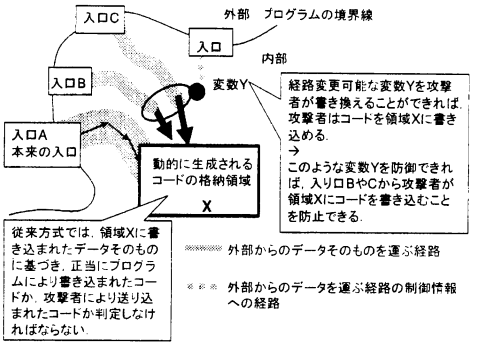

図 1　従来技術の問題点と対策

ファオーバーフロー検知の対象から外された領域に対し，任意の値を書き込むことも可能な場合があるため，なんらかの対策が必要である．

図 1 を用いて説明する．動的にコードを生成するプログラムの場合，動的に生成されたコードの格納領域（以下，領域 X とする）がプログラム中に存在し，外部から派生した値を最初に格納する領域（以下，入口と呼ぶ）から値を運ぶ経路が存在する．

Sue らの方法は，実行遷移先が外部から派生した値か否かに基づいて，バッファオーバーフローの発生を検知する．そのため，入力経路（図 1 の場合，入口 A，入口 B，入口 C のいずれか）にかかわらず，領域 X に書かれたデータが，正当にプログラムにより生成されたコードなのか，攻撃者により送り込まれたコードなのか判定するには，データそのものに頼るしかなくなり，本質的な解決は非常に困難となる．また，入り口 A からの本来の経路を経由する場合も同様である．

しかしながら，入口 B や入口 C からの経路等，本来の経路以外の経路を変更して攻撃者が領域 X にコードを書き込むためには，経路の変更に攻撃者の意図を反映する必要がある．すなわち，攻撃者の意図するコードを運ぶ経路を制御可能な変数 Y が存在し，この変数 Y 自体も他の外部から派生した値を運ぶ経路上にある必要がある．

もし，プログラムの性質上，このような変数 Y が外部より派生した値を運ぶ経路上にないことが予め分かっていれば，実行時にこのような変数 Y を監視し，外部より派生する値の格納が確認できれば，その時点でバッファオーバーフローを検知することが可能であり，領域 X に攻撃者が意図するコードを書き込むことを防止することができる．

2. 提案方式

本稿では，1 章に示す変数 Y のように経路を変更可能となるような変数にプログラマが明示的に，注釈を与えておき，実行時にこの注釈が与えられた変数を格納するメモリ領域が外部から派生した値を格納した時にバッファオーバーフローの発生を検知する方法を提案する．

提案方式の全体の処理の流れを図 2 に示す．

まず，プログラム中のデータメモリ領域を実行不可能であるか否か，外部から派生したデータを格納するか否かの 2 軸で分類し，図 3 のように分割された領域にそれぞれ領域番号を振る．

更にこの分割された領域番号 1,2,3 に相当する変数に対し，それぞれどの領域番号に相当するか注

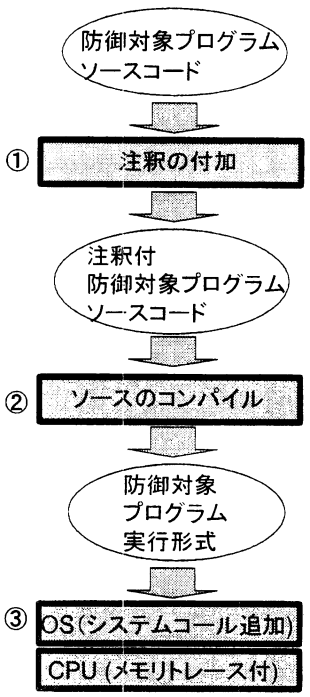

図 2　提案方式の全体の処理の流れ

	実行不可能	実行可能
外部から派生したデータを格納する．	1	3
外部から派生したデータを格納しない．	2	4

図 3　データメモリ領域の分類

釈をつける（図2中①）.

各注釈の詳細は2.1章にて説明するが，以後，本稿では，領域1を表す注釈を汚染可実行不可注釈（注釈1），領域2を表す注釈を汚染不可注釈(注釈2)，領域3を表す注釈を汚染可実行可注釈(注釈3)と呼ぶこととする.

次に，①の過程で注釈が加えられたソースコードをコンパイルし，プログラムの実行形式を生成するが，この時，注釈に関する情報を実行時に利用できるよう，通常のプログラム実行のためのインストラクションの他，システムコール呼び出しを加える（図2中②，詳細は2.2章参照）.

最後に②で生成されたプログラムを実行するが，この時，オペレーティングシステムには②の過程で加えられたシステムコールが実装されているものとする．また，CPUは①の過程で加えられた注釈のうち，注釈1の情報をメモリ中に伝播させる．すなわち，汚染されているメモリ領域の値を他のメモリ領域に複製したり，汚染されているメモリ領域の値を用いた演算結果を他のメモリ領域に格納したりする場合，格納先のメモリ領域も汚染されているとする（2.3章参照）.

このようにプログラムを実行し，保護されているメモリ領域が汚染された時，これを検知してプログラムの実行を停止する図4参照).

以下，①～③のそれぞれの過程について詳細に述べる.

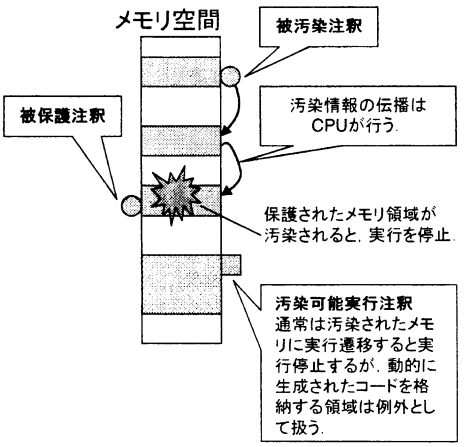

図4　プログラム実行の様子

```
/* spurious */ char b[4096];
main(){

/* protected */ int x;
/* except */ char buf[4096];
(void *) (run)()=(void *)buf;
funcA(buf,4096);

run();

}

funcA(char* buf,int size){

Char* ptr2;
/* protected */ char* ptr3;
char buf2[4096];
char buf3[4096];
ptr3=buf2;
ptr2=buf3;

fulfill(buf);
funcB(buf2);
for(i=0;i<size;i++,ptr2++,ptr3++){
 *ptr3++=*ptr2++;
 }

}
```

図5　サンプルソースコード

2.1. ステップ①　注釈の追加

図5に提案する注釈を加えたソースコードの例を示す．各注釈はC言語のコメントの形式で与えるものとし，以下，3種類の注釈について説明する.

2.1.1　注釈1：汚染可実行不可注釈

汚染可実行不可注釈(注釈1)は“外部から派生した値を格納する変数”に付加するものとし，図5中，コメント文で”spurious”と記述している．これは，例えば外部ネットワークやファイル等，プログラム外部から取得したデータを格納する変数を表す.

なお，本稿ではユーザ空間のメモリ領域の汚染状況の追跡を対象とするため，このような注釈1を明示的に与えているが，外部ファイルやネットワークからの入力を格納するカーネル内のバッファに同様の指定ができる場合は，注釈1を明示的に指定する必要はない.

注釈1が付加された変数を格納するメモリ領域は汚染されているとするが，注釈1が指定された変数を格納するメモリ領域から値を複製したり，演算に利用した結果を格納したりするメモリ領域も同様に汚染されているとする．この汚染情報の伝播は文献[10]と同様，メモリアクセスの際にCPUが行うものとする.

2.1.2　注釈2：汚染不可注釈

汚染不可注釈(注釈2)は，ソースコードの記述者

が，“外部から派生した値を格納することはあってはならない”と判断した変数に付加するものとし，図5 ではコメント文で protected と記述している．

提案方式では，この注釈 2 が指定された変数を格納するメモリ領域が汚染されると，プログラムの実行を停止するものとする．なお，注釈 2 が加えられた変数の格納領域は，実行は禁止されているものとする．

2.1.3 注釈 3: 汚染可実行可注釈

汚染可実行可注釈（注釈 3）は，外部から派生した値を格納するが，格納されたデータをコードとして実行を許可しなければならない時に指定するものとし，5 ではコメント文で except と記述している．注釈 3 は，例えば，JIT により動的に生成されたコードを格納する領域やトランポリンコードの格納領域等を指定する．

2.2. ステップ② ソースのコンパイル

2.1 章で説明したようなメモリ領域が汚染されている/保護されている，といった情報はオペレーティングシステムで管理するが，プログラムの実行時に格納アドレスが決定される変数の場合，その格納アドレスの指定方法が問題となる．

提案方式では，システムコールを用いてこの指定を行い，オペレーティングシステム内に記憶することで，この問題を解決する．例えば，スタックに配置される関数引数や局所変数の場合，変数格納領域を確保する際にスタックポインタを移動させるが，その前後においてシステムコール X,Y をそれぞれ呼ぶ図6 参照）．

また，関数呼び出しから復帰する場合も，別途定めたシステムコール Z を呼び，このシステムコールが呼ばれた時には，現在のスタックフレームに含まれるメモリ領域の指定をクリアすることとする．

システムコール X,Y,Z は各注釈に応じてそれぞれ用意するものとし，コンパイラは注釈に応じたシステムコール呼び出しをプログラム中に埋め込む．

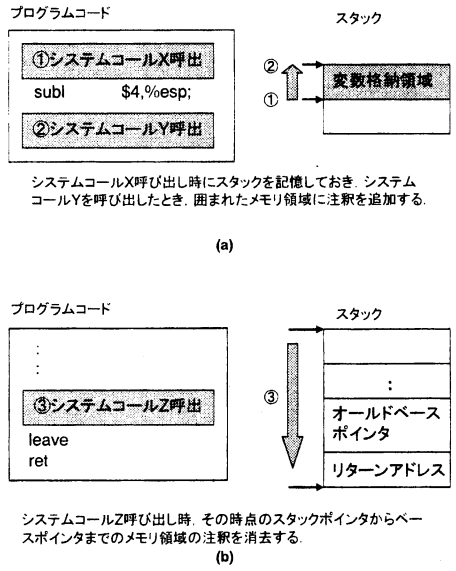

図 6 注釈情報のメモリへの反映

2.3. ステップ③ プログラムの実行

提案方式ではプログラム実行時，注釈 1 が指定されたメモリ領域の値を複製したり，演算に利用し演算結果を他のメモリ領域に格納したりする度

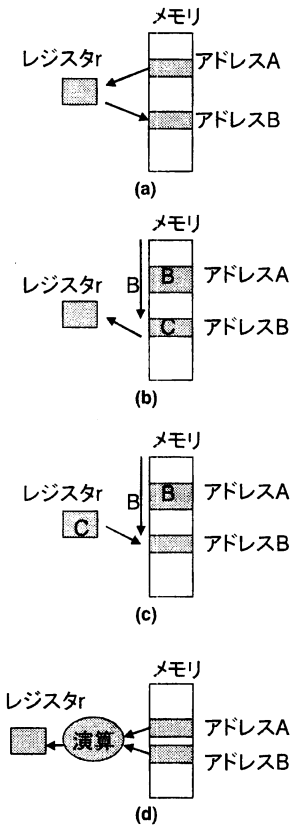

図 7 汚染情報の伝播

に，CPU によりメモリ領域の汚染情報，すなわち各メモリ領域が外部から派生した値を格納しているか否かの情報を伝播していく．これは，文献[10]が示す CPU と同様，レジスタやメモリキャッシュまで含め，各メモリ領域が汚染されているか記録可能な構成をとる CPU により行えば，これらの汚染情報の伝播処理に必要な処理時間の増加はほぼなくすことができる．

汚染情報の伝播のさせ方については，文献[10]にも示されている通り，以下に示す 4 通りの伝播依存規則が考えられる(図 7 参照)．文献[10]ではこれらの組み合わせとして，i 複製伝播依存規則/ロードアドレス伝播依存規則/ストアアドレス伝播依存規則を組み合わせたもの，及び，ii 前記の組み合わせに演算伝播依存規則を組み合わせたものの 2 通りについて，それぞれの防御性能，メモリ使用量，動作速度について考察している．

本稿では，これら 2 つの組み合わせのうち，より詳細な解析を行うことで未知の攻撃にも攻撃耐性が高いと考えられる②の組み合わせを想定する．

複製伝播依存規則

あるメモリ領域 A が汚染されている時，それをレジスタ r に複製したり，別なアドレス B に複製したりするとき，複製先のレジスタ r やアドレス B のメモリも同様汚染されているとする．

ロードアドレス伝播依存規則

あるメモリ領域 A が汚染されている時，メモリ領域 A に格納されている値 B をアドレスとして読み込んだ値を格納する先のレジスタやメモリ領域も汚染されているとする．

ストアアドレス伝播依存規則

あるメモリ領域 A またはレジスタ r が汚染されており，格納値が B の時，値 B をアドレスとして値を書き込んだ時，メモリ領域 B も汚染されているとする．

演算伝播依存規則

メモリ領域 A とメモリ領域 B に格納される値いずれかが汚染されている時，演算結果を格納する先も汚染されているとする．

3. 従来技術では防御できない攻撃例と提案方式による防御例

図5，8 を用いて従来技術では防御できないが，提案技術により防御できるプログラムの実行例を示す．図 5 は，このようなプログラムの具体例のソースコードを一部省略して表しており，関数 funcB，関数 fulfill 等の実装は省略している．また，図 8 はプログラムを実行したときのメモリの様子を表したものである．

図 5 のコード中，関数 main の局所変数 buf は動的に生成されたコードを格納する領域であり，関数 main の中で関数 funcA によりコードが生成されて書き込まれた後，関数ポインタ run 経由で実行遷移されるものとする．

関数 funcA の中では，動的なコード生成を模した関数 fulfill を呼び出す他，コード生成とは関連のない処理として関数 funcB の呼び出し，関数 funcA 中の局所変数 buf2，buf3 間の領域内容の複製を行うものとする．また，関数 funcB 内部において buf2 は汚染されることとし，buf2 の取り扱いにおいて，バッファオーバーフローが発生し，ポインタ ptr3 の値は攻撃者により変更可能であるとする．

図 8 は，ともに関数 funcA 呼び出し終了直前のメモリの様子を表し，図 8 (a)は通常の実行時を，図 8 (b)は関数 funcB 内においてバッファオーバーフローが発生し，ptr3 の値が改竄され buf を示すようになった場合を表す．

3.1. 通常時

b から buf，buf2 から buf3 へと値がコピーされ，汚染情報も b から buf，buf2 から buf3 へと伝播する．バッファオーバーフローも発生しないため，従来技術，提案技術ともに異常を検知せず，実行を終了する．

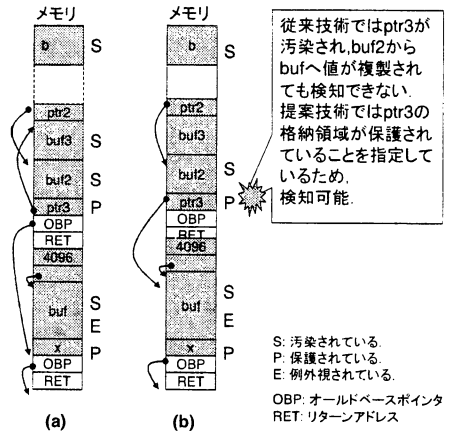

図 8　実行時のメモリの様子

3.2. バッファオーバーフロー発生時

関数 funcB 内部においてバッファオーバーフローにより ptr3 が改竄されて buf を示すようになっている時，関数 funcA の for ループは buf2 から buf へと値を複製する．この時，従来技術により防御している場合でも，汚染情報が buf3 から buf へと伝播するが，buf は動的に生成されたコードを保存する領域として例外視されているため，実行が継続されてしまう．

これに対し，提案方式では，変数 ptr3 を保護する指定がされていることから，バッファオーバーフローが発生した時点でこれを検知し，実行を停止することができる．

4. 防御性能に関する考察

図 3 のデータメモリ領域の分類方法を利用し，①最初にどの領域のデータを改竄するか，②最終的にどの領域に格納した攻撃コードを実行するか，による攻撃方法の分類を図 9 に示す．最初に改竄を行う領域は図 3 の領域 1～領域 4 の 4 通り，最終的に攻撃コードを配置する領域は実行可能である場合に限られため，領域 3，領域 4 のいずれかであることを考えると，全部で 4x2=8 通りの攻撃方法が考えられる．このうち，領域 4 を改竄して領域 3 に最終的に攻撃コードを配置する場合や，最終的に攻撃コードを領域 4 に配置する場合の攻撃方法は，従来技術，提案技術ともに検知可能であるため，図 9 からは省略する．

提案技術は，領域 2 を改竄し領域 3 に配置された攻撃コードへと実行遷移するような攻撃方法(図 9　NO1)を検知可能であり，この点で従来技術より防御性能が高いといえる．一方，3 章に示したように，図 9 NO2 や図 9 NO3 のような攻撃方法は，コードそのものから攻撃コードなのか正常に動的に生成されたコードなのかセマンティクスに踏み込まない限り検知することができない．バッファオーバーフローにより攻撃される脆弱性が存在するのは，図 9NO1，図 9NO2，図 9NO3 の場合，同じ確率であることを考慮すると，提案技術は既存技術で防御できない攻撃方法のうち 3 分の 1 を新たに防御可能である．

また，図 1 におけるデータを運ぶ経路と図 9 の各攻撃方法を対比すると，図 1 において領域 X を直接改竄する攻撃は図 9 NO2 にあたり，本来の入口 A を経由して領域 X にコードを書き込む場合は図 9 NO 3 にあたる．

入口 B や入口 C からの経路を改竄し，領域 X にコードを書き込む場合は，図 9 NO 1 と図 9 NO2 の両方あるが，提案方式では，外部から派生した値を格納しえない変数に予め注釈記述し，図 9 NO1 に該当する場合は，入口 B や入口 C からのデータの経路変更を困難にすることで，防御性能を高めている．

5. まとめ

本稿では，プログラムのソースコードに注釈を追加し，この注釈に基づいてバッファオーバーフローの検知を行い，攻撃者による任意コード実行を防止する方法を提案し，従来技術では防御できなかった攻撃方法を防御できることを，例を用いて示した．

今後の検討課題としては，CPU シミュレータを用いた実際の防御性能，注釈を手動で与えたときの誤検出/非検出率，パフォーマンスの評価を行うことが挙げられる．

NO	初期改竄位置	攻撃コード格納位置	従来技術	提案技術
1	1 3 / 2 4	1 3 / 2 4	×	○
2	1 3 / 2 4	1 3 / 2 4	×	×
3	1 3 / 2 4	1 3 / 2 4	×	×

○ : 検出可能　　× : 検出不可

図 9　従来技術/提案技術の各攻撃方法に対する防御性能の比較

参考文献

[1] Wikipedia, http://ja.wikipedia.org/wiki/C%E8%A8%80%E8%AA%9E.

[2] G. Hoglund, G. Mcgraw, Exploiting Software: How to Break Code,Addison-Wesley, 2004.

[3] 大山,王,加藤,静的解析に基づく侵入検知システムの最適化, 情報処理学会：コンピューティングシステム, Vol. 45, No. SIG 3 (ACS 5), pp. 11-20, 2004 年 5 月

[4] C. Cowan et al., Buffer overflows: Attack and defenses for vulnerability of the decade, DARPA Information Survivability Conf. & Exp., 2000.

[5] 江藤,依田,propolice:スタックスマッシング攻撃検出手法の改良,情報処理学会論文誌, Vol.43 No.12-053,2002.

[6] Stackshield, _ http://www.angelfire.com/sk/stackshield.

[7] Baratloo et al., Transparent runtime defense against stack smashing attacks, USENIX Annual Technical Conf., 2000.

[8] Microsoft, MSDN Security Developer Center, Execution Protection, http://msdn.microsoft.com/security/productinfo/XPSP2/memoryprotection/execprotection.aspx.

[9] I. Molnar, Exec Shield, http://lkml.org/lkml/2003/5/2/96, 2003.

[10] G. E. Sue, J. Lee, S. Devadas, Secure Program Execution via Dynamic Information Flow Tracking, Proc. of 7th Intl. Conf. On Architectural Support for Programming Language and Operating System, ACM, 2004.