# 2-4 C プログラムの静的解析によるバッファオーバーフロー検出

## 2-4 Buffer-Overflow Detection in C Program by Static Detection

中村豪一　牧野京子　村瀬一郎
NAKAMURA Goichi, MAKINO Kyoko, and MURASE Ichiro

**要旨**

C プログラムに潜在する脆弱性の中で最も危険な脆弱性はバッファオーバーフローである。そのようなバッファオーバーフローは、プログラムへの入力に依存して、実行の際に発現する場合と発現しない場合がある。本研究では、プログラム実行時に偶々発生したバッファオーバーフローを実行状況監視によって検出するのではなく、実行前にプログラムを静的に解析し、バッファオーバーフローを発生する可能性のあるプログラムの構造を網羅的に検出する手法をまとめ、ツールに実装した。

Buffer_overflow is the most dangerous vulnerability implicit in C programs. whether a Buffer_oveflow emerges or not in the program runtime, is depend upon inputs of the C program. We developed algorithms and tools to detect buffer_overflows, not accidentally partial detection by monitoring program runtime situation, but exhaustively detection by static-analysis of C source code before the program run. The exhaustively detection is the detection of, not only the source code position where a Buffer_overflow emerges, but also the essential source code structure makes the Buffer_overflow vulnerability.

**[キーワード]**
バッファオーバーバーフロー，静的解析，脆弱性検出
Buffer_overflow, Static analysis, Vulnerability detection.

## 1 はじめに

コンピュータのセキュリティホールの要因は様々であるが、プログラム自体に潜む脆弱性が原因となる場合が少なくない。そのような脆弱性の中でもバッファオーバーバーフローという現象を起こすプログラムの構造が脆弱性の原因となっている場合が多い。また、攻撃者に任意のコードを実行される等、被害の深刻さが最も目立つもの、このバッファオーバーバーフローに由来する脆弱性である。

バッファオーバーバーフローは、一言で言えば、開発者（プログラマ）が想定したデータ領域以外の場所にデータがあふれる（データが書き込まれる）という現象であり、主に C プログラムの実行時に生起する。

ソフトウエアの記述言語としては、C（C++ を含む、以下同様）や Java が現在では最も普及しているが、Java については、言語の設計段階から型安全性に配慮が図られており、バッファオーバーフローの生起の検出や生起の防止といった課題はバーチャルマシンの設計・実装（特にバイトコード検証系）に帰着される[1]。現実に Java プログラムのバッファオーバーフローがセキュリティホールとして問題になることは少ない。これに比べて、C プログラムは、実行時におけるデータ構成上、バッファオーバーフローが起きやすく、それによるセキュリティホールの問題は深刻である。

バッファオーバーフロー（を起こし得るようなコード）が潜在した C プログラムは、プログラムへの入力に依存して、実行の際にバッファオー

バーフローを発現する場合と発現しない場合がある。本研究開発では、プログラム実行時に偶々発現したバッファオーバーフローを実行状況監視によって検出するのではなく、実行前にプログラムを静的に解析し、発生する可能性のあるバッファオーバーフローを網羅的に検出するアルゴリズムとツールを開発した。

## 2 バッファオーバーフローについて

近年、不正侵入やウィルス、情報漏えい、Web ページ改竄といった情報セキュリティ上の様々な問題が起きている。これらの問題の要因としては大きく分ければ、セキュリティポリシーやセキュリティマネージメントの欠落といった組織管理上の要因、システムやネットワークの不適切な設定や運用といったシステム管理上の要因、バッファオーバーフローやメモリーリークといった個々のソフトウエアに潜在する脆弱性という要因が挙げられる。中でも、使用されているソフトウエアに脆弱性が潜在する場合、組織管理やシステム管理において対策を施しても情報セキュリティ上の問題が発生してしまうという点及び発見が難しいという点で、ソフトウエアに潜在する脆弱性は情報セキュリティ上の問題において最も重大な要因となっている。

ソフトウエアに潜在する脆弱性の中で近年、最も問題になっているのがバッファオーバーフローである。バッファオーバーフローが潜在するソフトウエアによる実行プロセスは最悪の場合、プロセス乗っ取りを攻撃者に許してしまい、最悪まで至らずとも、プロセスの正常な動作が阻害される等、その被害は深刻である。報告例が最も多いのもバッファオーバーフローである。

### 2.1 脆弱性報告に見るバッファオーバーフロー

CVE (http://cve.mitre.org/cve/downloads/full-cve.html) や ICAT (http://icat.nist.gov/icat.cfm) といったソフトウエア脆弱性の報告が集められた主なサイトにおいては、全体の報告例のうち、原因がバッファオーバーフローであると明示されているものが少なくない。例えば、次表は、2004 年 5 月の時点で、ICAT 及び CVE において“public”となっている脆弱性の総報告件数と、そのうちで原因がバッファオーバーフローであると明示されている件数である。

表1 バッファオーバーフロー報告件数

| 機関 | 総報告件数 | バッファオーバーフロー報告件数 |
|---|---|---|
| ICAT | 1131 | 386 |
| CVE | 6449 | 1482 |

これら脆弱性の報告における傾向は次のようになる。

- 報告例は、バッファオーバーフロー等のプログラム自体に内在する問題が原因となる脆弱性と、ソフトウエアの不適切な設定や運用により発現する脆弱性に 2 分される。
- プログラム自体に内在する問題として、突出して報告例が多いのはバッファオーバーフローであり、総報告件数の約 3 割程度を占める。この傾向はここ数年来同じである。
- プログラム自体に内在する問題として、バッファオーバーフロー以外に目立つのはフォーマットストリングバグやメモリリーク、クロスサイトスクリプティングであるが、これらもバッファオーバーフローに比べれば報告件数が少ない。
- 2000 年以降の傾向としては、総報告例が多くなったこと、バッファオーバーフローの割合も多くなっていること、クロスサイトスクリプティングなど新たな形の脆弱性の報告例が現れてきていることである。

このような脆弱性の報告の傾向から、プログラム自体に内在する問題の中で、バッファオーバーフローが非常に重要な問題であることが分かる。

### 2.2 バッファオーバーフローの種類

バッファオーバーフローという言葉はおおむね 2 種類の意味で使われる。本研究開発ではスタックオーバーフローと (広義) バッファオーバーフローと呼称で両者を区別している。

- スタックオーバーフロー：いわゆる、スタックスマッシングと呼ばれる攻撃を許すようなバッファオーバーフローである。プロ

グラム実行時、メモリ上には、ユーザプログラムが書き込みをしてはいけないクリティカルなデータが生成されることが一般的である。しかし、ユーザプログラムにおいて通常のデータ領域に対する書き込みを意図した命令がこのクリティカルデータが置かれた領域への書き込みになり、さらに、そのクリティカルデータを攻撃者が任意のデータに書き直すことができる場合、攻撃者がプロセス乗っ取りを行うという影響が出る可能性がある。このようなことを許すプログラムの構造をスタックオーバーフロー脆弱性と呼ぶことにする。

・（広義）バッファオーバーフロー：より広い意味で、ユーザプログラムにおいてある通常のデータ領域に対する書き込みや参照を意図した命令が、そのデータ領域をはみ出した場所への書き込みや参照になる現象を言う。影響は、プログラムの実行が妨げられるという程度から、プロセス乗っ取りを許すという程度まで様々である。

以下、単にバッファオーバーフローという場合は後者を指すものとする。

### 2.3 言語について

プログラム記述言語は多種多様であるが、最近では、C言語やJava、各種スクリプト言語が使われることが多い。攻撃対象になることの多いネットワーク関係ソフトウエアに特にその傾向が見られる。この中で、C言語は、次のような特徴を持っている。

・歴史も古く、世の中に蓄積されているプログラム中の大きな割合をCプログラムが占める。これら既存のプログラムに潜在されているバッファオーバーフローは大きな脅威である。

・現在でも主要なプログラミング言語の一つであり、多数のCプログラムが開発されている。当然、これらプログラムにバッファオーバーフローが潜在している場合、それは大きな脅威である。

・Cは実行速度を最優先するという思想を基本として開発された言語である。そのため、安全性に対する設計は足りない所が多く、特に実行時のメモリ構成や実行制御方法の都合上、非常にバッファオーバーフローを起こしやすい、言い換えれば、プログラミングの過程でバッファオーバーフローを潜在させやすい言語である。

Cと対象的なのはJavaである。言語の設計段階から型安全性に配慮が図られており、バッファオーバーフローの問題はバーチャルマシンでのバイトコードの型検証の問題に帰着される[1]。現実にJavaプログラムのバッファオーバーフローがセキュリティホールとして問題になることは少ない。また、既存プログラムの蓄積そのものがCに比べはるかに小さく、そこに脆弱性が潜在していたとしても、その脅威は既存Cプログラムに潜在する脆弱性の脅威に比べれば微小である。

### 2.4 本研究開発の対象

以上、述べてきたことにより、ソフトウエアとしてはCプログラム、セキュリティホールとしてはバッファオーバーフロー、これが最も重要であることが分かる。そこに対象を絞る。

スタックオーバーフローについてはそれを静的に検出する手法を既に発表している[2][3]。ここでは（広義）バッファオーバーフローを静的に検出する手法について述べる。

## 3 バッファオーバーフローの検出に関する既存の手法

### 3.1 標準ライブラリ関数を置き換える方法

バッファオーバーフローが発生することが判明している標準ライブラリ中の標準関数（strcpy等）があり、ソースコードを探索してこれら関数を見つけ出し、より安全な関数に書き換えるというツールはAVAYA LABS社のLibSafe[4]をはじめ幾つか存在する。この方式には次のような問題がある。

・ユーザプログラミング部分への対処不能：これらのツールはバッファオーバーフロー常習犯的な特定の標準ライブラリ関数を見つけるだけのものであり、プログラマが自分でプログラミングしたプログラム部分に起因するバッファオーバーフローを見つけ

るものではない。

したがって、ユーザプログラミングを含むプログラムのバッファオーバーフローに対するツールとしては全く不十分である。ただ、このようなツールを使って、標準ライブラリ中の危険な関数を置き換えておくことは、プログラムの脆弱性をなくす上で最低限の必要事項ではある。

### 3.2 動的検出

プログラム実行時に確保される領域の近傍等メモリ上の要所に特定値のダミーデータが配置されるように処理系又はコンパイルされたプログラムに手を加え、プログラム実行時にそのダミーデータが書き換わるか否かを常時監視して、書き換わった場合にはバッファオーバーフローが発生した可能性があるものとしてユーザに知らせるという動的(プログラム実行時)方法・ツールが提案されており、代表的なものとしてはスタックガード(Stack Guard)系統のツール[5]などがある。バッファオーバーフロー対策として最もポピュラーなのはこれら動的検出のツールである。某最大手のプログラム開発環境の次版にもこの動的検出機能が盛り込まれると発表されている。

なお、このダミーデータは一般に「カナリア」と呼ばれるので、以下、このダミーデータのことをカナリアと呼び、ダミーデータを埋め込むこの方式をカナリア方式と呼ぶことにする。動的検出は「ユーザプログラミング部分への対処不能」のような問題点はないが、一方で、以下のような問題点がある。

- ・実行速度低下：ダミーデータが書き換えられるか否かを常時監視することにより、プログラムの実行速度の低下が大きくなる。これは、プログラムの安全性よりも実行時の処理速度の高速化を志向するC言語の思想とは矛盾することになる。また、デバッグ段階ではなく実用段階でこのツールを使用することは実行速度の低下の点から問題がある。
- ・検出の非網羅性：バッファオーバーフローを動的に検出するということは、プログラムへの特定の入力時にバッファオーバーフローが発生したことをプログラムの実行時に事後的に検出することであり、バッファオーバーフローの発生箇所や、プログラムへの入力を含むバッファオーバーフローの発生条件などを網羅的に検出するものではない。カナリア方式のツールを用いてバッファオーバーフローを完全に防止することは動的解析にあっては原理的に不可能である。また、ツールを繰り返し適用してデバッグに時間を費やしてバッファオーバーフローを一つずつ取り除いていくとしても、それでプログラムがどの程度安全になったのかを保証する理論的な根拠はない。
- ・検出で得られる情報の過小性：バッファオーバーフローが発生したことは検出できても、プログラム上のどういう構造が原因となってバッファオーバーフローが発生したかという、バッファオーバーフロー構造を検出することはできない。バッファオーバーフロー構造はプログラムを修正する際に極めて重要な情報になるので、それが検出できないことは、プログラム修正時にプログラマに大きな負担を強いる。
- ・カナリア挿入自体の問題：バッファオーバーフロー(後述する狭義のバッファオーバーフロー)を検出する場合、メモリ上での書き換えがクリティカル領域の位置に来るか否かが重要である。ダミーデータであるカナリアを埋め込む場合は、カナリアを埋め込まない通常の場合と比較するとクリティカル領域の位置を狂わせてしまうことになる。したがって、カナリアを埋め込まない場合(本来の実行形態で実行する場合)に発生するバッファオーバーフローの検出を逃す可能性があるという問題もある。

これらの問題点は、動的検出というアプローチに本質的に伴うものである。これらを避けるには静的にプログラムを解析する必要がある。なお、メモリチェックツールとして代表的なPurifyも、カナリア方式ではないが、動的にバッファオーバーフローを検出することがある。

### 3.3 静的解析による検出

ソースコード中の注釈を利用して、静的解析によりバッファオーバーフローを起こす箇所を

検出するツールとしてLCLint[6]系列のツールがある。このツールの方法には次のような問題点がある。

・付加情報作成の困難性：このようなソースコード以外の部分にプログラマが付加的情報を与える方法においては、適切な付加的情報が与えられた場合には効果を発揮するが、このような適切な付加的情報を与えることは、プログラマに多大の負荷を掛けることになり、特に大規模なプログラムの場合にはその負荷は無視することができないほど甚大である。

LCLint自体は解析対象を特定の関数に限定しているために、検出することができるバッファオーバーフローの箇所は限られている。

また、機械語に操作的な意味を与えて機械語コードを抽出して解釈することにより、バッファオーバーフローを起こす箇所を検出する方法が提案されている[7]。しかし、機械語上で意味のある動作や機能を突き止めるためには非常に大域的な静的解析が必要となり、検出精度の面及び解析に必要な時間の面で非現実的な試みである。バッファオーバーフローを起こす箇所が検出できたとしてもどのような構造が原因となってバッファオーバーフローが発生したのかを特定することには非常な困難が伴うので、「検出で得られる情報の過小性」という問題もある。

### 3.4 パターンマッチングによる検出

ウィルス検出等では、対象コードをウィルスパターンとマッチングすることによりウィルスコードを検出する手法が広く用いられている。バッファオーバーフローは、それが起きる箇所は単なる代入命令や参照である。代入命令や参照がバッファオーバーフローになるかどうかは、その代入命令や参照へ至るまでのコード部分に複雑に依存するため、パターンマッチングやルールベースによる解析では、バッファオーバーフローを起こす箇所やその構造を網羅的に検出することは難しい。すなわち「検出の非網羅性」が問題となる。

## 4 Cソースコードの静的解析による検出

### 4.1 求められる検出手法の条件

以上を踏まえ、バッファオーバーフローに対する対策に関して要求される事項をまとめると次のようになる。

(1) プログラマがプログラミングした部分（ライブラリとして提供された関数以外の、プログラマが自分でプログラミングした部分）におけるバッファオーバーフローを検出対象とする。

(2) 実行時にバッファオーバーフローの発生を検出するではなく、実行前に静的にプログラムを解析して検出する。

(3) バッファオーバーフローが発生し得る箇所を網羅的に検出する、つまり検出漏れがないという検出の完全性が確保されている。

(4) 検出をプログラム修正までつなげるために、
・バッファオーバーフローが発生し得るデータ領域
・そのデータ領域におけるバッファオーバーフローを発生させ得るプログラム上の箇所（命令）
・その箇所で実際にバッファオーバーフローが発生する条件
これらから成るバッファオーバーフロー構造を求める。

(5) 検出に際しては、コメント等の追加的記述を一切プログラマには求めず、ソースコードをそのまま解析する。

(6) パターンマッチングやルールベースではなく、アルゴリズムで検出する。

### 4.2 検出アルゴリズムの概略

バッファオーバーフローの出現を左右するのは、配列やmalloc等領域確保命令で確保されたポインタ領域のサイズ、本文中でそれら配列やポインタ領域にアクセスする場合の指数、すなわち、a[i]や*(p+i)といった式における変数iの値及び変数aやpがどの配列やポインタ領域を示しているのか、これらの事柄であるので、通常のコンパイラにおいて用いられる定義-使用分析の手法を改良してこれらを同定する。

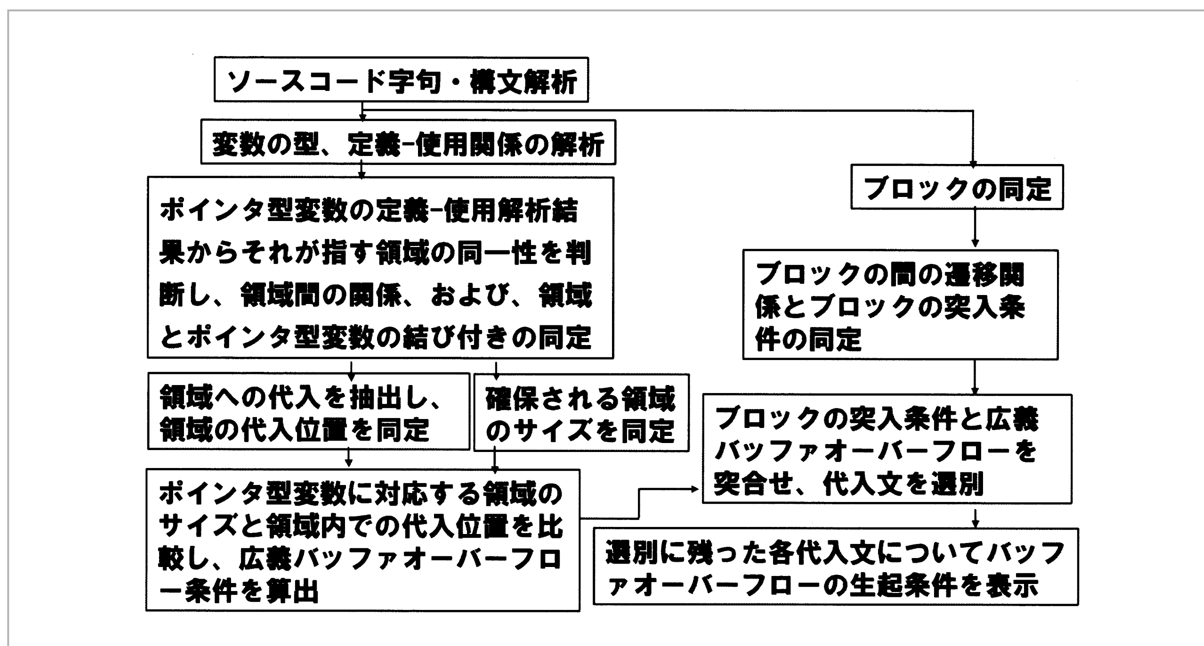


**図1** バッファオーバーフロー検出アルゴリズム概略

そして、検出において最も重要なのは、**4.1**(3)で述べた検出の完全性と同時に健全性(検出したすべての箇所で本当にバッファオーバーフローが発生し得る、つまり、誤検出がないこと)を確保することである。上の同定の結果から、バッファオーバーフローが起きる可能性のある箇所というのは基本的にすべて同定可能であるが、その多くの箇所においては、実際にバッファオーバーフローが発現するのは一定の条件が満たされた場合であり、多くの箇所は実際にはバッファオーバーフローを発現しないことが多い。すなわち、バッファオーバーフローが発現する可能性のある箇所をすべて検出するという検出の完全性は比較的容易に確保できるが、本当にバッファオーバーフローを発現する箇所及びその時の条件を求めるという検出の健全性を確保することが更に必要である。本アルゴリズムでは、ブロック間の遷移の構造を解析することによって、このバッファオーバーフローが発現する可能性のある箇所のみならず、それが発現するための条件も解析しており、これが健全性の確保に役立っている。検出アルゴリズムの概略は図1のようになる。この検出アルゴリズムは「求められる検出手法の条件」をすべて満たしている。

### 4.3 検出ツール

この検出アルゴリズムをツールとして実装した。検出結果としては、**4.1**(4)で述べたバッファオーバーフロー構造となる。C言語のマクロやインクルードファイルについては、これをプリプロセスで展開しないと、以後の静的解析はできないので、プリプロセス後のCソースコードを検出ツールは解析対象とする。

### 4.4 GUI及び実験

バッファオーバーフロー構造は、かなり複雑なデータである。文字列としてユーザに表示するのはユーザフレンドリ性からいって無理がある。そこで、GUIを構築し、プログラマが(バッファオーバーフロー構造を取り除く)デバッグに利用しやすいものにした。GUIでは、バッファオーバーフロー構造のうち、バッファオーバーフローが発生し得るデータ領域を「生起し得る領域」、そのデータ領域におけるバッファオーバーフローを発生させ得るプログラム上の箇所(命令)を「生起し得る箇所」、その箇所で実際にバッファオーバーフローが発生する条件を「生起する条件」と称している。

「生起し得る領域」と「生起し得る箇所」は一般的に1対多の関係にあることは自明である。「生起し得る箇所」について、そこでバッファオーバ

ーフローが実際に生起する条件は、その箇所より（プログラムの制御フロー上）前の各地点で異なってくるので、「生起し得る箇所」と「生起条件」も1対多の関係にある。GUIではユーザのクリック操作によって、これらを切り替えて見やすく表示する。実験については、C言語に関する市販の書籍や脆弱性報告でバッファオーバーフローがあると報告されたプログラムのうちソースコード入手可能なものについて、検出の実験を行っている最中であるが、完全性を含め「求められる検出手法の条件」において示した要件を満たし、健全性について問題なく、かつ現実的な解析時間で検出が行えている。

## 5　まとめ

ソフトウエアに潜在する最も危険な脆弱性であるバッファオーバーフローに関して、狭い意味のものと、広範な現象としてのものという2種に分類した。後者について、Cプログラムを対象として、実行前にプログラムを静的に解析し、発生する可能性のあるバッファオーバーフローとその原因構造を網羅的に検出するアルゴリズムとツールを開発した。

今後は、C++をはじめ、バッファオーバーフローが問題となっている他言語へアルゴリズム・ツールを拡張して適用すること及びメモリリーク等の他のソフトウエア潜在脆弱性の静的解析による検出手法を開発することを行っていく。

### 参考文献

1　萩谷昌巳，“Java 仮想機械手続きのための新しいデータフロー解析について”，第1回プログラミングおよび応用のシステムに関するワークショップ論文集，日本ソフトウェア科学会，1998.
2　中村豪一，村瀬一郎，“静的解析によるCプログラムのバッファオーバーフロー検出”，情報処理学会プログラミング研究会，June 2002.
3　中村豪一，村瀬一郎，“静的解析によるバッファオーバーフロー検出について”，情報処理学会コンピュータセキュリティ研究会，Jul. 2002.
4　AVAYAlabs LibSafe : http://avayalabs.com/project/libsafe/.
5　江藤博明ほか，“proplice-スタックスマッシング攻撃検出手法の改良”，信学技報，ISEC2001-43，pp. 181-188，2001.
6　Larochelle, D. and Evans, D., "Statically Detecting Likely Buffer Overflow Vulnerabilities", 2001 USENIX security symposium, Washington D. C., Aug. 1996.
7　Xu, Z., Miller, B. P., and Reps, T., "Safety Checking of Machine Code", proceedings of the conference on programming language design and implementation, June 2000.

**中村豪一**（なかむらごういち）
株式会社三菱総合研究所
情報セキュリティ、量子計算

**牧野京子**（まきのきょうこ）
株式会社三菱総合研究所
ソフトウェア工学

**村瀬一郎**（むらせいちろう）
株式会社三菱総合研究所
情報セキュリティ