コンピュータセキュリティ　18−18
(2002.　7.　18)

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について

中村　豪一†　　村瀬　一郎†

† 三菱総合研究所　〒110−8141 千代田区大手町 2−3−6
E-mail: †{gon,murase}@mri.co.jp

あらまし　ネットワーク制御等高度なセキュリティが要求されるソフトウエアの開発言語としてCは依然として重要であるが，Cプログラムの最も深刻な脆弱性としてバッファオーバーフローが問題になることが多い．バッファオーバーフローとして重要なのはスタック上のリターンアドレス書換えである．このバッファオーバーフローを検出する手法も開発されているが，それらはプログラムに潜むバッファオーバーフローを系統立って検出するには至っていない．本研究では，Cプログラムを GCC によってコンパイルする過程で現れるレジスタ転送言語コード (RTL コード) を静的に解析してバッファオーバーフローを検出するという新しい手法をまとめた．この手法は，スタック上のリターンアドレス書換えが起きる条件を関数引数等を使って表現したものを算出する．また手法を実現したツールを開発し，バッファオーバーフロー脆弱性を持つCプログラムに対する手法の有効性を検証した．
キーワード　静的解析，バッファオーバーフロー，Cプログラム，RTL

About buffer_overflow detection by static analysis of C program

Goichi NAKAMURA† and Ichiro MURASE†

† Mitsubishi Research Institute　Otemachi 2−3−6, chiyoda-ku, Tokyo, 110−8141 JAPAN
E-mail: †{gon,murase}@mri.co.jp

Abstract　C language is still important as programming language of softwares such as network control software that needs high security. But the buffer_overflow problem is frequently seen in C programs, it is one of the most serious vulnerabilities about C programs. Among the buffer_overflow vulnerabilities, rewriting of return address on the stack is most important. There are several methods to detect this buffer_overflow vulnerability in C program . But these methods can not pick over　this buffer_overflow vulnerability. In this research, a new method is developed to detect the buffer_overflow vulnerability(rewriting of return address on the stack) in C program statically, that is, by static analysis of register translate language code(RTL code) which is made in C program compilation by GCC compiler. As this method's output, the conditions on which rewriting of return address on the stack occurs is expressed in function arguments and so on. And the tool which carry out this method is developed. Then, the effectiveness of this method is checked by adapting the tool to C programs to detect the buffer_overflow vulnerability.
Key words　static analysis, buffer_overflow, C program, RTL

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ーオーバーフローに関する脆弱性の報告が多い．また，攻撃者に任意のコードを実行されたり等被害の深刻さが最も目立つものバッファーオーバーフローである．

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 が最も普及している．Java については，言語の設計段階から型安全性に配慮が計られており，バッファーオーバーフローを含むプログラムの脆弱性検出の問題はバーチャルマシンの問題（実装やバイトコード検証系等）に帰着される [8]．現実に Java プログラムのバッファー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，そのスタックの構造上，バッファー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．

Cプログラムにおいて，バッファオーバーフロー検出のために使われる既存の手法・ツールが存在するが，バッファオーバー

表 1　ソフトウエア脆弱性の報告件数

ソフトウエアの種類	総報告件数	バッファオーバーフロー報告件数
UNIX(LINUX も含む)及びその上のアプリに関するもの	436	74
Windows 及びその上のアプリに関するもの	267	50
その他	90	4

フローをプログラム実行前に体系立って検出する，あるいは網羅的に検出することを目指したもので成功しているものは無い．本研究では，静的解析によるバッファオーバーフロー検出を目標とした．

以下，第 2 節で，バッファオーバーフロー検出の意義を述べる．第 3 節では，C プログラムの最も深刻な脆弱性である，スタック上のリターンアドレス書換え (stack smashing) に起因する狭義バッファオーバーフロー脆弱性の形態を明確にする．さらに，レジスタ転送言語コードを静的に解析することで狭義バッファオーバーフローを検出する手法を提示する．第 4 節では，提示した手法を実装したツールによる検証結果を述べ，第 5 節でまとめと課題を述べる．

2. プログラムのバッファオーバーフロー検出の意義

2.1 バッファオーバーフロー脆弱性

コンピュータセキュリティの脆弱性報告例において，プログラムの脆弱性に関するものの割合は一定の割合を占める．その中で最も報告が多いのがバッファオーバーフローである．被害の深刻さが最も目立つものバッファーオーバーフローである．

2.2 脆弱性報告に見るバッファオーバーフロー

ソフトウエア脆弱性の報告が集められたサイトとしては，以下のものがメジャーである．

- CVE
  (http://cve.mitre.org/cve/downloads/full-cve.html)
- CERT
  Advisories (http://www.cert.org/advisories/)
- SecurityFocus.Com
  (http://www.securityfocus.com/)

1999 年以降 2000 年初までにこれらサイトに報告された脆弱性情報について，サイト間で重複して報告されたものを除いた (つまり「のべ」ではなく正味の) 報告件数，そのうち脆弱性がバッファオーバーフローであると明記されている報告件数を表 1 示す [4].

これらの報告では，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が多い．プログラム自体に内在する脆弱性として，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である．

2.3 バッファオーバーフローを突く攻撃と影響

バッファオーバーフローを突いた攻撃とそれによる被害を被害の深刻さでおおまかに分類すると以下の 3 種になる．被害の軽微な方から順に列挙する．尚，ここでプロセスとはプログラムの実行単位である．

- A) プロセスの実行を妨害する攻撃：プログラム中のバッファオーバーフローを攻撃者によって突かれることにより，そのプロセスがクラッシュする．そのプロセスが関連するサービスが停止する等の被害が上がるが，被害の範囲はそのプロセスが関係する部分に概ね限定される．
- B) プロセスの実行を部分的に制御する攻撃：プログラム中のバッファオーバーフローを攻撃者によって突かれることにより，そのプロセスの制御条件等が書換えられ，プロセスの実行が一部おかしくなる．そのプロセスが稼動しているサイトの資源枯渇等に繋がる可能性もあり，サイト全体に被害が広がる恐れがある．
- C) プロセスを乗っ取る攻撃：プログラム中のバッファオーバーフローを攻撃者によって突かれることにより，そのプロセスの実行制御が完全に攻撃者に奪われる．攻撃者が攻撃コードをそのプロセスの実行権限で行えるようになってしまう．特にルート権限のプロセスにおいてこれをやられると，サイト全体が攻撃者に乗っ取られることにつながりかねない．

2.4 バッファオーバーフローの種類と本研究での検出対象

バッファオーバーフローの定義を一口で言うと，開発者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），ということになる．しかし，現在では，「バッファオーバーフロー」という言葉の意味は以下の 2 種類のいずれかとして使われている．

- **狭義バッファオーバーフロー**：ソフトウエア実行中に作られるデータ領域の中にはコード実行制御に関するクリティカルな部分がある．C 言語で言えば，スタック中の RET アドレスが代表的である．他の部分への書込みがこのクリティカルな部分への書込みになるような脆弱性を狭義バッファオーバーフローとする．狭義バッファオーバーフローを持つソフトウエアに対しては攻撃者は A) から C) の攻撃を行なうことができる．この狭義バッファオーバーフローを突く攻撃は stack smashing とも言われる．
- **広義バッファオーバーフロー**：クリティカルな部分以外のデータも書き換えられるような場合を広義バッファオーバーフローとする．広義バッファオーバーフローを持つソフトウエアに対しては攻撃者は A) の攻撃を行なうことができる．また，B) の攻撃が行われる可能性もある．

広義のバッファオーバーフローが完全に検出できるのであれば，狭義のバッファオーバーフロー検出の方法を研究する意味は薄れる．しかし，広義バッファオーバーフローを検出する既存の

方法（後述）にはいずれも欠陥があること，広義に比べて狭義のバッファオーバーフローを突かれた場合の被害が深刻であり検出ツールに対する差し迫ったニーズがあること，広義よりも狭義の方が形態が明確で解析し易いことを考えると，狭義バッファオーバーフロー検出を，静的解析による検出の研究の取付きに選ぶことは妥当と考えられる．

2.5 C プログラムにおけるバッファオーバーフロー

ソフトウエアの記述言語としては java の伸長も見られるが，依然として C が最も重要である．

C プログラムは，そのスタックの構造が単純であるため，バッファーオーバーフローが起きやすい．C は安全性よりも処理速度に重点を置いた言語であり，実行時のデータ構造をなるべく単純にして，それに対する安全性チェックを行わないことが処理系設計の基本思想になっており，バッファオーバーフローに対する安全性はこの基本思想と言わばトレードオフの関係にある．

2.6 バッファオーバーフロー検出のための既存の手法

C プログラムにおいて，その現在バッファオーバーフロー検出のために使われる既存の手法・ツールが存在する．主なものをその欠点と共に列挙する．

- バッファーオーバーフローを内在していることが判明している標準ライブラリ中の関数 (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 ReliableSoftwareTechnologies 社の ITS4 や AVAYA LABS 社の LibSafe [3] を始め幾つか存在する．しかし，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，プログラマの作成したプログラム部分に起因するバッファーオーバーフローを見つけるものではない．従って，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検出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は必要条件であるとは言える．

- プログラム実行時にスタック中の RET アドレス（後述）の回りにダミーデータを埋め込んでおき，そのダミーデータが書き換わるかどうかを常時監視して，書き換わった場合は RET アドレスも書き換わった可能性があるとして，ユーザに知らせるツールが StackGuard を始めとして幾つかある．また IBM 社の ProPolice [1] もこの系統のツールである．プログラム実行時に動的にバッファーオーバーフローを検出するものであるが，以下の欠点がある．尚，このダミーデータを StackGuard では「カナリア」と呼んでいるので，以降，このダミーデータをカナリアと呼び，ダミーデータを埋め込むというこの方式をカナリア方式と呼ぶことにする．
  - ダミーデータが書き換わるかどうかを常時監視することによる実行速度の低下が大きく，安全性よりも実行時の処理速度を指向した C の設計指針に正面から矛盾するものである．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には問題がある．
  - 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フローが起きたことを事後的に検出することであり，バッファオーバーフローが起き得る場所と（プログラムへの入力を含む）起きる条件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使ってバッファオーバーフローを取り切ることは原理的に不可能であり，また，デバッグに時間を費やして（ツールを繰り返して適用して）バッファオーバーフローを一つづつ取っていくにしても，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根拠は無い．
  - 後述するが，狭義バッファオーバーフロー検出の場合，スタック上での書き換えがスタック上の RET アドレスの位置に来るかどうかが重要である．カナリアを埋め込むことは RET アドレスの位置を，埋め込まない通常の場合に比べて，狂わせてしまう．カナリアを埋め込まない場合に発現する可能性のある狭義バッファオーバーフローが検出出来ない可能性があるという問題がある．

- 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検出する方法として LCLint [6] がある．このような，ソースコード以外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れば効果を発揮するが，そのような適切な付加的情報を与えることのプログラマに対する負荷は特に大規模なプログラムの場合には無視できない．LCLint 自体は，解析対象を特定の関数に限定しているため，それが検出できるバッファオーバーフローは限られている．

- 機械語に操作的意味を与え機械語コードを抽象解釈することによってバッファオーバーフローを検出する方法が提案されている [7]．元のソースコードがどの言語で記述されているかに関係なく検出が可能な方法であるが，実際にバッファオーバーフローが問題となっているソフトウエアは C が多く，C の場合は後述するように機械語ではなく RTL を解析対象とすれば良い．異なる機械語セットごとに操作的意味を与える必要もあり，また，検出できたとしてもソースコードのどういう構造が原因でバッファオーバーフローが発現したかを特定するには困難が伴うと思われる．

尚，メモリ関連エラー検出のツールとして Purify [2] というツールが使われている．このツールの適用が一部の広義バッファオーバーフローを検出する場合がある．

2.7 ANSI C と GCC

C 言語の言語仕様のオリジナルは ANSI C である．日本でも JIS X3010 として標準化されている． 現在，C のプログラミング環境・実行環境としては，パソコンの性能の向上によりパソコン上での開発が現実的になり，またパソコンの OS

としてはWindowsが最も普及していることにより，マイクロソフト社のプログラミング環境・実行環境が普及してきている．しかし，ワークステーション上での開発もまだまだ一般的であること，パソコン上でもLINUXの最近の進展があること，ANSI　Cに忠実であること，これらの理由によりGCCが依然として最も重要である．　以上により，本研究では，C言語としてANSI Cを，プログラミング環境・実行環境としてはGCCを対象とする．すなわち，ANSI Cに従って書かれ，GCC上で稼動するCプログラムを対象とする．

3. 狭義バッファオーバーフロー

3.1 スタック上のクリティカルなデータ

プログラム実行時には，メモリ上に実行コードが置かれ，そのコードのアドレスデータがスタック上に置かれる．狭義バッファオーバーフローは，そのようなアドレスのデータが書き換えられることを発現の必要条件とする脆弱性である．そのような書換えられると問題となるクリティカルなデータは主に，スタック上の関数ポインタ型変数，および，RETアドレスである．スタック上の関数ポインタ型変数の書換え検出はRETアドレス書換え検出の手法を転用出来る．よって，当初の定義通り，狭義バッファオーバーフローとはRETアドレスが書き換えられる脆弱性を言うこととする．

3.2 Cプログラム実行時にスタック上に現れるデータ構造

GCCでCプログラム（Cソースコードを GCCでコンパイルして生成した実行コード）を実行する際にスタック上には一定のパターンのデータ構造が現れる．親関数から子関数を呼んでフレームを形成し終えた段階においてスタック上に積まれているデータ構造を以下に示す．ESP，EBPというのはレジスタである（後述）．

- **関数スタック領域**：ここより上は子関数の実行時スタック領域．子関数が孫関数を呼ぶ場合はここに引数を積む．
- **ローカル変数領域**：但し下から詰められて割り当てられている訳ではなく空白がある．配列は通常indexが小さい方が上．
- **EBP(1ワード)**：子関数呼出時の親関数のフレームポインタ値（EBPレジスタ値）．
- **RETアドレス(1ワード)**：親関数のコードのアドレス．
- **引数領域**：子関数に渡す引数の並び．親関数の関数スタック領域中．ソース上で引数列の前にある引数が，スタック上では上にくる．

これらをまとめてフレームと呼ぶ．関数呼出毎にその関数に対するフレームがスタック上に積まれる．　例えば，プログラム1を実行した場合，@1の時点での子関数sub_procのフレームは図1のようになる．

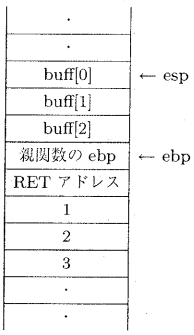

図1　プログラム1実行時のスタックの状態図

```
int sub_proc(int a, int b, int c)
{
    int buf[3];   @1
    buf[0]=a;
    buf[1]=b;
    buf[2]=c;
    buf[4]=0;
    return(1);
}
int testproc()
{
    sub_proc(1,2,3);
     .
     .
}
```

プログラム1

3.3 狭義バッファーオーバーフローとなるコードとそこを突いた攻撃 (stack smashing)

Cには，スタック上に実行コードを積んでそのコードを実行させることが可能であるという特徴がある．攻撃者が攻撃用のコードを（入力がスタック上のローカル変数に格納されること等を利用することにより）scanf等の入力関数を介してスタック上に積み，さらに，狭義バッファーオーバーフローがあればそれを利用して，スタック上のあるRETアドレスをその攻撃用のコードにセットすることが可能である．これが，狭義バッファーオーバーフローを突いたプロセス乗っ取り攻撃の過程である．ローカルな配列aについてa[b]=c; というコードがあるとする．この時，以下のような条件が揃っていると狭義バッファーオーバーフローを突いたプロセス乗っ取り攻撃を受け得る．

（1） 外部から一定の大きさのデータを取り込んでメモリ上に書き込む個所があること．

（2） 配列の要素を書き換える時，オフセット項，つまりa[b]=cとなっている場合のインデックスbで書き換える要素を指定するが，そのbがスタック上のRETアドレスに対応するインデクス値になり得ること．これは，攻撃者がそのよ

うな値にオフセット項を操作することが可能な場合と，攻撃者が特に設定をしなくてもそのような値にオフセット項がなる可能性がある場合の2通りの場合がある．

（3） a[b] の位置に書かれる値である c が攻撃者により操作可能であること．

操作可能についてであるが，関数引数等関数外部で定義されたもので攻撃者により値が設定されている可能性のあるものが c の値に影響を与えている場合には，そのような c は操作可能であるとする．攻撃者は以下のようにして攻撃する．

- (1) のような個所を使ってメモリ領域に攻撃用のコードを書き込む
- (2) のような，オフセット項が RET アドレスに対応する値になる可能性のある or 攻撃者が RET アドレスに対応する値にオフセット項を定義可能な個所をみつける
- そのような個所において (3) が成り立つ場合，メモリ上に書き込んだ攻撃用コードを指すように c を操作して RET アドレスの上に書く

プログラマが普段組んでいるプログラムは (1) の条件を満たすことが一般的であり，これを満たさないようにプログラムを組むようプログラマに強制させるなどということは非現実的である．また，配列の要素への書込み（同義でポインタが指す領域への書込み）も C プログラムの骨格を成す必要不可欠なコードの形である．従って，狭義バッファーオーバーフローを突いた攻撃を受けないようなプログラムを作るには，配列の要素への書込みが (2) と (3) を満たすかどうかを検出し，満たす場合はプログラマに知らせる，ということが妥当と考えられる．尚，配列の要素への書込みが (2) を満たすが (3) は満たさない場合は，広義バッファオーバーフローの脆弱性となる．

以上の配列に関する議論は $*(a+p)$ のようなポインタ＋オフセット項の形を左辺に持つ代入文にも同様に適用できるので，以下では配列で以って議論を行う．

3.4 RTL コードを解析対象にすること

GCC のコンパイル（リンクを除く）は，大きく分けて次の4段階から成る．

（1） C ソースコードを字句・構文解析して中間コードを生成する．

（2） 中間コードを書き換えていくことで最適化等を行う．

（3） レジスタ割付等をしてスタックを含むデータ構造を固定し RTL コードとする．

（4） RTL コードをアセンブラへ変換する．

従って，RET アドレスの位置は (3) において決まり，RTL コードを解析することによってそれは求められる．ちなみに，(1) で生成された中間コードも RTL コードと呼ばれるが，混乱を避けるため，ここでは，(3) で生成されたものを RTL コードと呼ぶことにする．

子関数呼び出し時に積まれるローカル変数とそのサイズは静的に決まっているので，子関数のスタック領域の構造は静的に決まる．プログラムにおいて，関数呼出関係が静的に決まる部分については，その部分中の各実行時点でのスタック領域の構造が静的に決まり，従って，スタック上の RET アドレスの配置も静的に決まる． しかし，RET アドレスの位置（のみならずフレームの構造の細部）は最適化オプションや GCC インストール時のオプションによって異なってくる場合がある．従って，C ソースコードを解析してもスタック上の各 RET アドレスの位置は厳密には求められない．RTL コードを解析することが，RET アドレスの配置を厳密に求めること，すなわち狭義バッファオーバーフロー検出には必要である．また，RTL コードが C ソースコードに比べ文法構造が単純で解析し易いことも RTL コードを解析対象にする理由である．

3.5 RTL コードの構造

一つの関数に対する RTL コードは以下のような部分から構成される．

- **プロローグ**：フレームを作り EBP，ESP をセットする．ローカル変数領域もここで確保される．
- **本体部分**：関数の処理の本体．RTL 文が並んでいる．上から下へ順に実行されるが，jump を行う jump 文と jump 先を示すラベルである label 文が含まれている．ある label 文から次の label 文直前までを一つの RTL ブロックと呼ぶことにする．すると，本体部分は RTL ブロックが並んでいる構造になる．
- **エピローグ**：フレームを撤去し，EBP と ESP を関数呼出前の状態に戻し，返り値を積む．

RTL コード中に現れるレジスタの種類は次の通りである (レジスタの名称は CPU によっては異なる場合がある)．

- **esp**：スタックポインタ（スタックの先頭を指す）を格納するレジスタ，本体部分では使われない．
- **ebp**：フレームポインタ（フレームの根元を指す）を格納するレジスタ，本体部分で最も多用されるレジスタである．ローカル変数や関数引数等スタック上にあるデータは ebp からのオフセットという形で指定される．
- **eax,ecx,…**：値を一時的に保管しておくテンポラリレジスタ．
- **cc0**：jump 文においてジャンプするかどうかの判断に使われるレジスタ．

RTL ブロックは次のような種類の RTL 文の並びである．

- **label 文**：RTL ブロックの先頭に来る文．jump 先を示すラベルとしての役割以外の役割は無い．

- **set 文**：どこのレジスタあるいはメモリ位置にあるデータ(of 項と呼ぶ)を，どこのレジスタあるいはメモリ位置に(at 項と呼ぶ)書き込むかを記述した文．RTL とは Register Traslation Language の略だが，その Register Traslation を記述する．
- **jump 文**：レジスタ cc0 の値が特定の値だった場合，どこの label 文にジャンプするかを記述した文．
- **call 文**：関数呼出を行う文．

set 文の説明中の「メモリ位置」の中にはローカル変数が含まれる．　プリミティブ型ローカル変数への代入，すなわち，プリミティブ型ローカル変数に対応する項を at 項とする set 文において，at 項の一般形は次の形になる．

ebp+負定数

配列型ローカル変数への代入，すなわち，ローカル変数である配列の要素に対応する項を at 項とする set 文において，at 項の一般形は次の形になる．

ebp+負定数+オフセット項

ここで，ebp+負定数が配列型ローカル変数を示し，オフセット項が要素のインデックスを示している．RTL ブロックにはこれ以外の種類の RTL 文も含まれるが，本研究ではそれらは解析の対象にならない．

3.6 遡　　及

RTL ブロック本体中の各 set 文の at 項と of 項は一般的に以下を含む項である．

- テンポラリレジスタ
- ローカル変数に対応する項
- グローバル変数に対応する項
- 関数引数に対応する項
- アドレスデータ（ポインタ型変数がポイントするヒープ領域内部）に対応する項

ある項を遡及対象項とする．遡及対象項に対して以下の手続きを実行することで遡及が行われる．

（1） 遡及対象項中にテンポラリレジスタがある場合，そのテンポラリレジスタを at 項とする set 文のうち，遡及対象項が属する set 文の直前にある set 文の of 項を新たな遡及対象項とする．

（2） 遡及対象項中にローカル変数に対応する項がある場合，そのローカル変数に対応する項を at 項とする set 文のうち，遡及対象項が属する set 文の直前にある set 文の of 項を新たな遡及対象項とする．

（3） 遡及対象項中にローカル変数に対応する項がある場合であって，そのローカル変数に対応する項を at 項とする set 文が同じ RTL ブロック内に無い場合は，そのローカル変数に対応する項に対する遡及を止める．

（4） 遡及対象項中にテンポラリレジスタやローカル変数に対応する項が無い場合は，その遡及対象項に対する遡及を止める．

この手続きにより，最初の遡及対象項中にあるテンポラリレジスタがローカル変数や関数引数等ソースプログラム中に現れる変数から構成される項に帰着される．また，最初の遡及対象項中にあるローカル変数に対応する項が関数引数等に帰着される場合もある（必ずそうなるとは限らない）．

3.7 ブロック突入条件

各 RTL ブロックの最後の部分には，他の RTL ブロックへの jump 文がある．jump を行うかの判定はレジスタ cc0 の値によって成され，jump 文の前には cc0 の値をセットする set 文がある．この set 文では at 項は cc0 であり，of 項は比較演算子とその下の二つの項という形になる．この of 項を jump 条件項と呼ぶことにする．ある RTL ブロックにプログラム実行が突入するための条件は，RTL ブロック間の遷移関係上でその RTL ブロックへの遷移の可能性のある各 RTL ブロックの jump 条件項で表現できる．

3.8 狭義バッファオーバーフロー検出の手続き

以上述べてきたことをまとめて，狭義バッファオーバーフロー検出の手続きを示す．

1. RTL コード（一つの.c ファイルについて一つ生成される）の字句・構文解析．この時以下の項目について求める
   - 1.1 各関数の引数個数，呼ばれる時の引数領域のサイズ（スタック上何ワード分か）
   - 1.2 関数間の呼出関係について静的に決まる部分
   - 1.3 各関数のフレーム構造（ローカル変数領域のサイズや並び等）
2. RTL コードの各関数本体部分について以下の解析を行う
   - 2.1 RTL ブロックの同定
   - 2.2 RTL ブロック間の遷移関係の同定
3. 各 RTL ブロックについて以下の解析を行う
   - 3.1 各 set 文について at 項の遡及をブロック内で行う（3.2 における判別の準備）
   - 3.2 ローカル変数を a として，a[b]=または *(a+b)=という形をした代入に対応する set 文を判別
   - 3.3 それら set 文について，オフセット項（上の例では b）と of 項に対してブロック内で遡及しておく
   - 3.4 jump 条件項下の二つの項（前節参照）に対する遡及をしておく
4. 上の 3.3 のオフセット項と of 項の各組について以下の解析を行う
   - 4.1 RTL ブロック間の遷移関係を辿っていくこと

により，of 項が関数外部から操作可能かどうかを判別

4.2 of 項が操作可能と判別された場合はその項を持っていた（3.2 における）set 文について狭義バッファオーバーフローになる条件を（ブロック内遡及後の）オフセット項で表現（＃1）

4.3 その set 文が属するブロックのブロック突入条件（＃2）と，狭義バッファオーバーフローになる条件を比べ，前者により後者が否定される場合は，このオフセット項と of 項の組についての 4. を終える

4.4 狭義バッファオーバーフロー条件の中にローカル変数が含まれていた場合は，RTL ブロック間の遷移関係を辿って関数引数等まで遡及できる場合は遡及（＃3）して，条件の表現を改める

5. 狭義バッファオーバーフロー条件を出力（＃4）

（＃1）1.2 において関数間の呼出関係について静的に決まる部分があれば，その関数呼出関係において上位にある関数のフレームの RET アドレスについても狭義バッファオーバーフローになる条件を表現する．

（＃2）現状においては，遷移関係において一段前のブロックからの遷移が起きる条件のみ．

（＃3）現状においては，遷移元のブロックが単一である場合にのみ遡及している．

（＃4）次節で述べる．

3.9 解析結果のアウトプットの仕方

解析結果である狭義バッファオーバーフローが発現する条件をプログラムにフィードバックして，狭義バッファオーバーフローが発現しないような措置を講じることが最終的には必要となるが，その方式としては大きく次の二つが考えられる．

- アウトプット方式1：狭義バッファオーバーフローが発現する条件を，プログラマへ警告として表示する．プログラマがその警告を見てどういう対策を講じるか（プログラムをどのように修正するか）判断する．
- アウトプット方式2：狭義バッファオーバーフローが発現する条件をチェックする if 文は容易に自動生成できるので，そのような if 文をソースコードへ自動的に挿入する．

4. 検出ツールの開発と検証

前章で述べた内容の実装として，狭義バッファオーバーフロー検出ツールを開発した．本開発では，bison および flex を使用した．解析結果のアウトプットの仕方についてはアウトプット方式1に従った．

狭義バッファオーバーフローの基本的なパターンを持つプログラム，及び，以前に著者が開発したより本格的なプログラムについて，開発したツールを適用し，ツールの有効性をチェックした．その一部について以下に示す．

次のプログラム2中の一連の sub_proc は，狭義バッファオーバーフローの基本的なパターンの一部である．これを解析・検出した結果（検出ツールのシェル上への出力）のうち狭義バッファオーバーフローの起き得る場所と条件を警告している部分を図2に示す．この程度の簡単なプログラムなら検出に問題は無い．

```
int sub_proc1(int a, int b)
{
 int buf[1];
 buf[a] = b;
}

int sub_proc2(int a, int b)
{
 int buf[1];
 int c = a+1;
 buf[c] = b;
}

int out_of_func;
int sub_proc3(int a, int b)
{
 int buf[1];
 buf[a+out_of_func] = b;
}

int sub_proc4(int a, int b)
{
 int i;
 int buf[2];
 for(i=0;i<a;i++){
  buf[i] = b;
 }
}

int sub_proc5(int a, int b)
{
 int i=0;
 int buf[2];
 while(i<a){
  buf[i] = b;
  i++;
 }
}

int sub_proc6(int a, int b)
{
```

```
 int i=0;
 int buf[2];
 do{
  buf[i] = b;
  i++;
 }while(i<a);
}
```

プログラム 2

より本格的なプログラムでの検証として JVM 実装の一部等，著者が以前開発したプログラムにおいて同様の検証を行った．これらのプログラムでは，配列のレンジチェック等を特に注意してコーディングしたという訳ではない．その結果，検出結果は大量なものとなった，すなわち警告が多数にのぼる結果となった．これには二つの原因が考えられる．

- 検出対象のプログラムがバッファオーバーフローに対する考慮が成されず（足りず）にプログラミングされている場合，本当に多数の狭義バッファオーバーフローが存在する．
- 手続きの 4.3 や 4.4 が不完全なものに止まっているため，実際に狭義バッファオーバーフローが発生しない場所についても警告を出すという不健全性がある．

5. まとめと課題

C プログラムの最も深刻な脆弱性である，スタック上のリターンアドレス書換えに起因する狭義バッファオーバーフロー脆弱性の形態を明確にした．C プログラムを GCC によってコンパイルする過程で現れるレジスタ転送言語コードを静的に解析して，狭義バッファオーバーフローを検出する手法をまとめた．この手法は，スタック上のリターンアドレス書換えが起きる条件を関数引数等を使って表現したものを算出する．また手法を実現したツールを開発し，バッファオーバーフロー脆弱性を持つ C プログラムに対する手法の有効性を検証した．

課題としては，前章で示したように，警告が過多になる問題についての課題がある．「狭義バッファオーバーフロー検出の手続き」の 4.3 や 4.4 を refine することによって，警告が減っていくと考えられる．これには，ブロック間の遷移関係が複雑な場合においてもブロック間を跨いでの遡及をいくような手法を確立していく必要がある．

広義バッファオーバーフロー検出も課題の一つである．そのためには，スタック上のデータ構造の解析だけでなくヒープ上データ構造解析 [5] も対象にする必要がある．ただ，プログラマが配列のレンジチェック等に特別に注意を払ってプログラミングしたプログラムでない限り，広義バッファオーバーフローの可能性のある場所は多数に登り，その形態も多様を極める（重要な問題であるにも拘わらず，この問題に対する静的解析アプローチによる研究が少ないのは，このような困難があるためと思われる）．静的，動的，ソースコードレベル，RTL コードレベル，といった様々な解析を組み合せていくことが有効ではないかと考えている．

```
...
when (ARG1) == 2,
 narrowBO at line 6
...
when ((ARG1)+(1)) == 2,
 narrowBO at line 13
...
when ((ARG1)+(out_of_func)) == 2,
 narrowBO at line 20
...
when (LV6) == 5,
 under_condition (((LV6)-(ARG1))<(0))
 narrowBO at line 28
...
when (LV6) == 5,
 under_condition (((LV6)-(ARG1))<(0))
 narrowBO at line 37
...
when (LV6) == 5,
 under_condition (((LV6)-(ARG1))<(0))
 narrowBO at line 47
```

図 2　プログラム 2 に対する出力

なお，本研究は通信・放送機構の委託研究制度により委託を受けて実施されたものである．

文　献

[1] 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.
[2] Rational Purify: http://www.rational.co.jp/products/purify/.
[3] AVAYAlabs LibSafe: http://avayalabs.com/project/libsafe/.
[4] 通信・放送機構: 平成 1 2 年度委託研究成果報告書, セキュリティ共通要素技術・評価検証技術, 2002.
[5] Binky, D.W. et al: Application of the Pointer State Subgraph to Static Program Slicing, J. SYSTEMS SOFTWARE, 40, pp. 17–27, 1998.
[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.
[8] 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.