平成21年度　　卒業研究論文

# 複製プロセスによるロールバック機能を有したセルフヒーリングシステムの設計と実装

指導教官
齋藤　彰一　准教授

名古屋工業大学　情報工学科
平成18年度入学　18115077番

志水　広海

# 目 次

# 第1章

# はじめに

近年インターネット環境の普及に伴ない，不正アクセスによる被害が増加の一途をたどっている．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といった被害が起こりうる．これらの被害を引き起こす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ことが知られており，このセキュリティホールを修正するためにプログラムの提供元から修正パッチが提供される．しかしその修正パッチの配布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み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防止システムが注目されつつある．しかし，侵入防止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．

侵入防止後もサービスを提供できる仕組みとしてセルフヒーリングシステムが提案されている．これは不正アクセスを検知後，不正アクセスの際に改竄された箇所を修復し，プログラムの実行を継続させる．未知の攻撃に対応するためには，どこを改竄されても修復できるように，十分な情報を保持しておく必要がある．

本論文では複製プロセスによるロールバック機能を有したセルフヒーリングシステムを提案する．プロセスを修復に用いることで修復の成功率を高める．また，提案システムを Linux 上に実装し，複製プロセスの生成と複製プロセスに実行を切り替える際の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それらに対する既存の対策手法を述べ，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防止手法

本章では不正アクセスと不正アクセスによる被害，不正アクセスを行う上でよく利用さ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

不正アクセスとは，あるコンピュータに対して正規のアクセス権を持たない人が，何らかの方法によりアクセス権を奪取し，不正にコンピュータを利用すること，あるいは試みることである．アクセス権を奪取する主な方法として，ソフトウェアのセキュリティホールを悪用する方法が挙げられる．

近年インターネットの普及に伴ない不正アクセスによる被害が増加していることから，国内において 1999 年に不正アクセス行為の禁止等に関する法律が成立，施行された．しかし未だに不正アクセスによる被害が多発しているのが現状である．

## 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．ファイルを盗み見られたり削除，改変されるウェブサイトを改竄され，ウイルスに感染させるためのウェブサイトにリダイレクトされるといった被害も多発している．また，不正アクセスのための踏台にさせられ他のユーザやネットワークに被害を及ぼす可能性もある．

## 2.3 不正アクセスの種類

多くの不正アクセスはセキュリティホールに起因する．メモリの値を改竄し，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，フォーマットストリングに不正なデータを送り込むことで任意のプログラムを実行させるフォーマットストリング攻撃が有名である．

### 2.3.1 バッファオーバーフロー攻撃

本項ではバッファオーバーフロー攻撃について述べる．本攻撃は，メモリに対して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場合，書き込み先として確保した領域を超えて書き込みを行うというセキュリティホールを悪用することで成立する．このセキュリティホールは，確保された領域より大きいデータを書き込み，本来ではアクセス出来ないメモリの値を変更することができる．これにより変数やリターンアドレスの値を変更し，プログラムの動作を変更させることや，任意のコードを実行される可能性がある．

### 2.3.2 スタックオーバーフロー攻撃

スタックオーバーフロー攻撃は，スタック領域内に確保されたバッファをオーバーフローさせることで行う．スタックにはバッファと共に関数の戻りアドレス等の制御情報が配置されている．本攻撃はバッファをオーバーフローさせることでこの制御情報の書き換えを試みる．書き換えに成功するとコンピュータの制御を奪うことが出来る．

### 2.3.3 ヒープオーバーフロー攻撃

C 言語において，malloc 関数や calloc 関数を呼び出すとヒープ領域内に新たなバッファが確保される．バッファオーバーフローはこのヒープ領域でも起こりうる．セキュリティホールを利用してヒープ領域内のバッファをオーバフローさせることで本来書き込むことが出来ない領域に対して書き込みを行うことが出来る．

```
{
    recv(buf);
    printf(buf);
}
```

図 2.1: フォーマットストリング攻撃を行うことが可能なプログラム

### 2.3.4 フォーマットストリング攻撃

本項ではフォーマットストリング攻撃について述べる．本攻撃は printf 関数等におけるフォーマットストリングの引数に不正なデータを送り込むことで攻撃を行う．通常，フォーマットストリングは固定されたフォーマット文を引数にとるが，単に文字列を渡すこともできるようになっている．しかし，この文字列が入力に依存している場合，攻撃者が意図した任意のコマンドを送り込むことが可能となる．図 2.1 に，フォーマットストリング攻撃の対象となるプログラムを示す．図 2.1 は，外部から受け取ったデータを printf 関数によって出力するというプログラムである．printf 関数には外部から受け取ったデータを引数として直接渡している．この時引数として渡されたデータがフォーマット文の形式となっていた場合，buf 以外の引数は指定されていないため，printf 関数はコールスタックから値を取り出して表示する．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を知ることができる．これによって攻撃時に必要な情報を取り出すことができる可能性がある．また%n トークンを指定すると任意のアドレスに任意のデータを書き込める．

## 2.4 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削減することは出来るが，完全に除去することは困難である．そのため不正アクセスを防ぐための手法が提案されている．

### 2.4.1 Exec_Shield

バッファオーバフロー攻撃では，メモリ上に不正な命令コードを送り込みそれを実行することで侵入を試みる．これを防ぐために Linux では Exec_Shield[1] という機能が実装されている．この機能を有効にすると，データ用メモリページ上のデータを実行することが不可能になり，コード用メモリページに対して書き込みすることが出来なくなる．そのためデータ用メモリページに保存されている不正な命令コードは実行することが出来なくなる。さらにコード用メモリページの値を書き換えて処理を変更することも不可能となる．しかしこの機能では既にメモリ上にロードされている関数を呼び出す return to libc 攻撃を防ぐことは出来ない．

### 2.4.2 安全なライブラリ

スタックオーバフローやフォーマットストリング攻撃等を防ぐことを目的として作成された libsafe[2] と呼ばれるライブラリが公開されている．ダイナミックリンクライブラリ形式で動作する．C 言語においてスタックオーバフローを引き起こす可能性のある関数の動作を変更し，書き込み先アドレスのチェックを行うことでスタックオーバフローの発生を防ぐ．また，スタックのリターンアドレスやフレームポインタの正当性のチェックを行うことでフォーマットストリング攻撃を検知，防御する．しかし局所変数の改竄とヒープ領域のオーバーフローを検知することは出来ず，戻りアドレスの改竄を検知することは出来ない．

### 2.4.3 専用ハードウェアを用いたロールバック

Sathre らの提案するシステム [3] では，監視対象プロセスの実行中に一定間隔でチェックポイントログを作成し，作成したチェックポイントログを専用の記憶装置に保存する．各チェックポイントにはタイムスタンプ，レジスタの状態や I/O マップ等のプロセッサの状態とメモリの状態で構成される．チェックポイントログの作成には専用のハードウェアを利用する．不正アクセスを検知した際は，保存しておいたチェックポイントを使ってロールバックを行う．これにより不正アクセスが行われる前の状態に

戻し，改竄された箇所を修復する．専用ハードウェアを用いることで負荷は軽減されている．しかし既存のシステムに対してこのシステムの導入を考えた場合，専用ハードウェアを追加しなければならない．そのため導入は容易ではないと考えられる．

### 2.4.4 DIRA

Smirnov らの提案する手法 [4] では，GCC コンパイラを拡張する．監視対象プロセスのソースコードをコンパイルする際，拡張済み GCC コンパイラを用いることで，不正アクセスの検知や攻撃パケットの特定等の機能を付加した実行バイナリを生成する．GCC が自動でこれらの機能を持つようにソースコードを変換するので，プログラマがこれらの機能を意識する必要はない．しかしこれを利用するためには監視対象プロセスのソースコードが必要となるため，ソースコードが非公開となっているプログラムに対しては適用することが出来ない．

### 2.4.5 侵入防止システム

侵入防止システム (IPS:Intrusion Prevention System) とは，ネットワークとコンピュータへの不正侵入を防止するシステムである．不正アクセスを検知しても通知するのみであった侵入検知システム（IDS:Intrusion Detection System）を基に，不正アクセスに対して自動的に防御動作をとるよう発展させたものが侵入防止システムである．侵入防止システムには大きくわけてネットワーク型とホスト型の 2 つが存在する．

ネットワーク型の侵入防止システムはネットワークの境界に設置される．コンピュータウイルスや DoS 攻撃などのパターンを予め記憶しておき，侵入検知時には通信の遮断などの防御動作をリアルタイムに行う．また侵入があったことを管理者に通知したり，ログを記録しておく等の機能も備えている．

ホスト型の侵入防止システムは，サーバにインストールするソフトウェアとして使用される．バッファオーバフローを利用した不正侵入の OS レベルでの阻止や，アクセスログの改竄防止，不正アクセス検知時にサーバをシャットダウンさせる等の機能を持つ．

**Belem**

侵入防止システムの既存手法として槙本らの提案する Belem[5] と呼ばれるシステムがある．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履歴と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アクセスを検知するホスト型の侵入防止システムである．予め監視対象プロセスの実行バイナリを解析し動作規則を作成する．監視対象プロセス実行中において，システムコールが発行される毎に実行中に得られる情報と動作規則とを比較し，正常に実行されているかどうかをチェックする．比較項目は，ライブラリ関数の呼出アドレス，ライブラリ関数の呼出順序と呼出元ライブラリ関数のシステムコール群である．

## 2.5 セルフヒーリングシステム

侵入防止システムは，不正アクセス検知後の動作として管理者に通知後，監視対象プロセスを停止させるに留まる．そのため監視対象プロセスのセキュリティホールが修正されるまではサービスを安定して提供できないという問題がある．近年ではこの問題を解決するためにセルフヒーリングシステムと呼ばれるシステムが提案されている．これは侵入を検知後，監視対象プロセスの改竄された箇所を自動的に修復し，実行を継続させるシステムである．

## 2.6 セルフヒーリングシステムの既存手法

既存のセルフヒーリングシステムとして白井らの提案する手法 [6] を説明する．プロセスを停止させずに実行を継続させる実行継続処理と，オーバーヘッドを軽減するために脆弱性の位置を推測して実行継続処理の実行範囲を制限する脆弱性推測処理からなる．この実行継続処理は，不正入力データの除去とライブラリ関数内でのオーバーフローの検出と戻りアドレスの保存処理からなる．これらの処理を常に行うとオーバーヘッドが高い．そのため，脆弱性の存在する位置を推測して，実行継続処理の一部を脆弱性を含む可能性が高い範囲でのみ行う．また，脆弱性の位置を何度か攻撃を受け

ることで最適化する．

### 2.6.1 実行継続処理

#### 不正データの除去

不正データは外部からのデータを受け取るシステムコールによってプログラム内に入る．これを利用し，システムコール内でデータの検査を行う．通常 read システムコールと recv システムコールが発行されると，OS は引数で渡された領域に直接データを書き込もうとする．ユーザ領域に不正データが書き込まれるのを防ぐために，カーネル内に外部から読み込むデータを保存しておくための一時領域を確保し， 外部から読み込むデータはまずその一時領域に保存する．その後，不正データかどうかのチェックをし，不正データでなければユーザ領域に書き込む．

#### 戻りアドレスの保護

libsafe[2] と同様の手法で，環境変数 LD_PRELOAD を用いてライブラリ関数のフックを行う．そして，ライブラリ関数が呼び出された際にコールスタックに含まれる戻りアドレスと退避済み ebp の値を別領域に保存する．呼び出されたライブラリ関数がリターンする際，コールスタックに含まれる戻りアドレスと保存しておいた戻りアドレスを比較する．もし一致しなければ保存しておいた値でコールスタック上の戻りアドレスを書き換える．

### 2.6.2 脆弱性範囲推測処理

#### 脆弱性範囲の設定

まず脆弱性範囲が設定されていない状態でプロセスを起動する．もし実行中に攻撃が行われた場合，攻撃を検知した場所付近に脆弱性を含むコードが存在する可能性が高いため，異常を検知した場所から遡って脆弱性範囲の推測を行う．設定後，プロセスの再起動を行う．この後，推測を行った範囲を実行する時には不正入力データの除去とライブラリ関数内でのオーバーフローの検出を追加で行う．

**最適化**

前項で設定した脆弱性範囲の推測は間違う可能性もある．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竄された場合には，推測範囲を広げて対応する．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．これにより，システム負荷を再び軽減する．絞り込みが完了するまでは，実行継続処理により一時的に高負荷がかかる．しかし，同じ攻撃を繰り返し受けて推測範囲を最適化し，最終的には脆弱性が含む場所のみで実行継続処理を有効にする．

# 第3章

# 提案手法

白井らの手法では監視対象プログラムのスタック上の戻りアドレスのみ複製を作成し，リターン時にスタック上の戻りアドレスと複製とを比較することで改竄のチェックを行っている．この時改竄されたデータが戻りアドレスのみである場合は，複製をスタック上に書き戻すことで正常なリターンを保証することができる．しかし戻りアドレス以外のデータも同時に改竄された場合，既存手法では戻りアドレス以外の改竄をチェックしていないためこの改竄を検知することが出来ない．よって正常にリターンを行えたとしても，監視対象プログラムは改竄された値をもとに実行を続け，予期せぬ問題を引き起こす可能性がある．

また，脆弱性範囲推測処理について，脆弱性範囲の幅は決められたライブラリ関数の個数分単純にさかのぼるだけである．この方法で決められた脆弱性範囲内に脆弱性箇所が含まれていれば実行継続処理によって脆弱性箇所を特定することが出来る．しかし脆弱性箇所が含まれていない場合，プロセス再起動をしなければならず，そのオーバヘッドは無視できない．よってこのオーバヘッドを少しでも削減するためにより効果的な脆弱性範囲推測処理を行う必要がある．

戻りアドレスも含む改竄されたデータを正常な状態に復元する手法として，複製プロセスによるロールバック機能を有したセルフヒーリングシステムを提案する．本提案システムは，提案版実行継続処理と提案版脆弱性範囲推測処理から成る．改良版脆弱性範囲推測処理によって脆弱性範囲が含まれると思われる箇所を推測し，その範囲の処理を実行する直前でプロセスの複製を作成する．脆弱性範囲内では提案版実行継

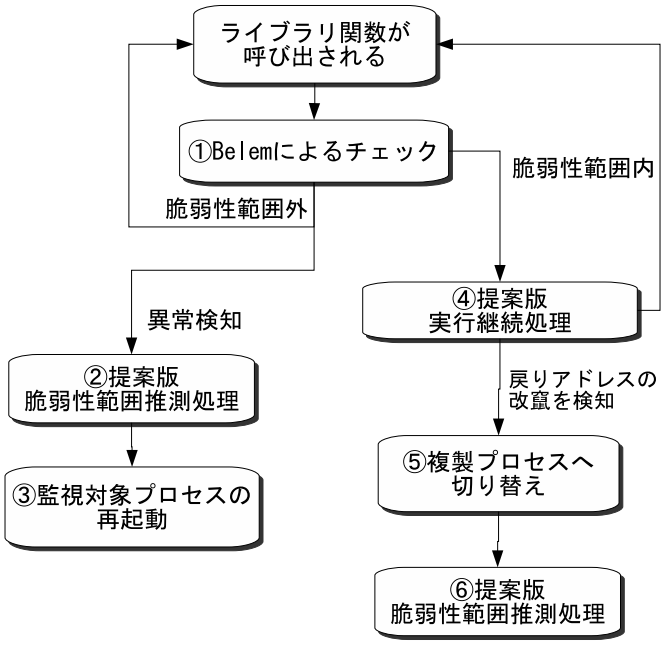


図 3.1: 提案システムの全体の流れ

続処理として不正データの除去，戻りアドレスの改竄チェック，バックアッププロセスの作成と複製プロセスへの切り替えを行う．

## 3.1 システムの流れ

本章ではシステムの全体の流れについて，図 3.1 に示すフローチャート図と図 3.2 に示すサンプルプログラムを用いて説明する． まずユーザが図 3.2 に示すサンプルプログラムを起動する．サンプルプログラムがライブラリ関数を呼び出すと，Belem によって正常な呼出かどうかチェックを行う (図 3.1 の 1)．サンプルプログラム実行中に Belem によって，funcB 関数において不正アクセスを検知したとする．この時提案システムは，提案版脆弱性範囲推測処理として不正アクセスを検知したポイントから遡って脆弱性範囲の設定を行う (図 3.1 の 2)．この時，脆弱性範囲の始点が read 関数，recv 関数，recvfrom 関数，getenv 関数のいずれかとなるように範囲を設定する．以降これら 4 つの関数を攻撃の受ける可能性のある関数とする．図 3.2 の場合，プログラムの先頭

```
{
    recv(buf);
    funcA();
    funcB(buf);
}
```

図 3.2: サンプルプログラムのソースコードと脆弱性範囲

に recv 関数が存在しているため，funcB 関数から遡って recv 関数までが脆弱性範囲として設定される．範囲を設定後，監視対象プロセスを再起動する (図 3.1 の 3)．提案版実行継続処理により，脆弱性範囲の始点となっている recv 関数を実行する直前にプロセスの複製を作成する (図 3.1 の 4)．作成後，脆弱性範囲内で Belem によるチェックに加えて提案版実行継続処理を行う．もし提案版実行継続処理中に戻りアドレスの改竄を検知した場合，実行中の監視対象プロセスは実行を終了し，複製プロセスへ実行を切り替える (図 3.1 の 5)．切り替えの際，攻撃を受けたポイントと攻撃の種類をカーネル内に確保した一時領域に記憶しておく．以降この 2 つの情報を攻撃に関する情報とする．複製プロセスの実行を再開する際，カーネル内に保存した攻撃を受けたポイントと攻撃の種類に基づいて脆弱性範囲の絞り込みを行う (図 3.1 の 6)．もし改竄を検知せずに脆弱性範囲内の処理を終了した場合，生成された複製プロセスは破棄する．

## 3.2 提案版実行継続処理

提案版実行継続処理について説明する．提案版実行継続処理は不正データの除去と戻りアドレスの改竄チェック，複製プロセスの作成と複製プロセスへの切り替えから成る．

### 3.2.1 複製プロセス

本提案システムでは修復に用いるための複製としてプロセスを用いる．以降複製用に生成されたプロセスを複製プロセスとする．複製プロセスを生成する手法として Linux

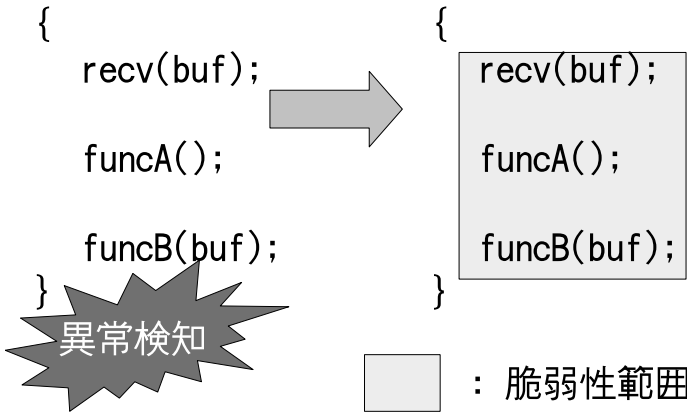


図 3.3: 脆弱性範囲の設定

のカーネルに実装されている fork システムコールを基に，複製プロセスを生成するシステムコールを実装することで実現する．

### 3.2.2 複製プロセスの作成

Linux のプロセスは，カーネルがプロセスを識別するためのプロセス ID，プロセス自身の状態，使用しているメモリー情報やファイル，ソケット，シグナル等により構成される．fork は子プロセス用の領域を新たに確保し，これらの情報を確保された領域に全てコピーする．また，メモリ空間についても，fork システムコールを呼び出した時点の親プロセスのメモリ空間全体を子プロセスにコピーする．このメモリ空間にはスタック領域やヒープ領域を含む．そのため子プロセスを複製プロセスとして用いることにより幅広い情報の修復が可能となる．

#### コピーオンライトの適用

前項で述べたプロセスを構成する情報全てをコピーするため，複製プロセス生成時のオーバヘッドは大きい．また複製プロセスの生成時に，直ちに新たな空き領域を探して割り当て，コピーを実行したとする．もし複製プロセスに実行が切り替わらずに破

棄されることになった場合，複製プロセスのために確保された領域は無駄になる．このオーバヘッドを削減するために，Linux に実装されている fork システムコールでは，コピーオンライトによる最適化が行われている．書き換えることのないメモリページは，元のプロセスと生成されたプロセスで共用し，書き換える可能性のあるメモリページは，新たなメモリページを割り当ててコピーを作成する必要がある． ここでコピーオンライトが使用される．一方のプロセスがメモリを更新すると，カーネルがその操作を横取りし，メモリのコピーを作成してメモリ内容の更新が他方のプロセスから見えないようにする．

このコピーオンライトの機能を複製プロセスの生成に適用する．これによりバックアッププロセス生成に要する時間を削減する．また，もし脆弱性範囲内で戻りアドレスの改竄が検知されず複製プロセスを破棄することになった時，図 3.4 に示すように，複製プロセス用に確保されたメモリ領域は各種管理情報と変更があったメモリページのみであるため，メモリの節約も実現される．

### 3.2.3 戻りアドレスの改竄チェック

本システムでは白井らのシステムと同様の手法で戻りアドレスのチェックを行っている．しかし白井らのシステムと異なり，戻りアドレスをスタック上に書き戻すことは行っていない．これは複製プロセス内に戻りアドレスを含むメモリ空間全体が保存されているためである．

### 3.2.4 不正データの除去

不正データの除去として，白井らのシステムで行われている read システムコール，recv システムコールの入力データのチェックに加えて，getenv 関数の入力のチェックを行っている．攻撃を受ける可能性のある関数の直前で複製プロセスを作成後，異常を検知して複製プロセスへ実行を切り替える場合を考える．脆弱性範囲の始点が例えば recv 関数であった場合，複製プロセスに切り替えることで外部からデータを受け取る直前まで復元することができる．切り替え後不正なデータが送られてこなければ以

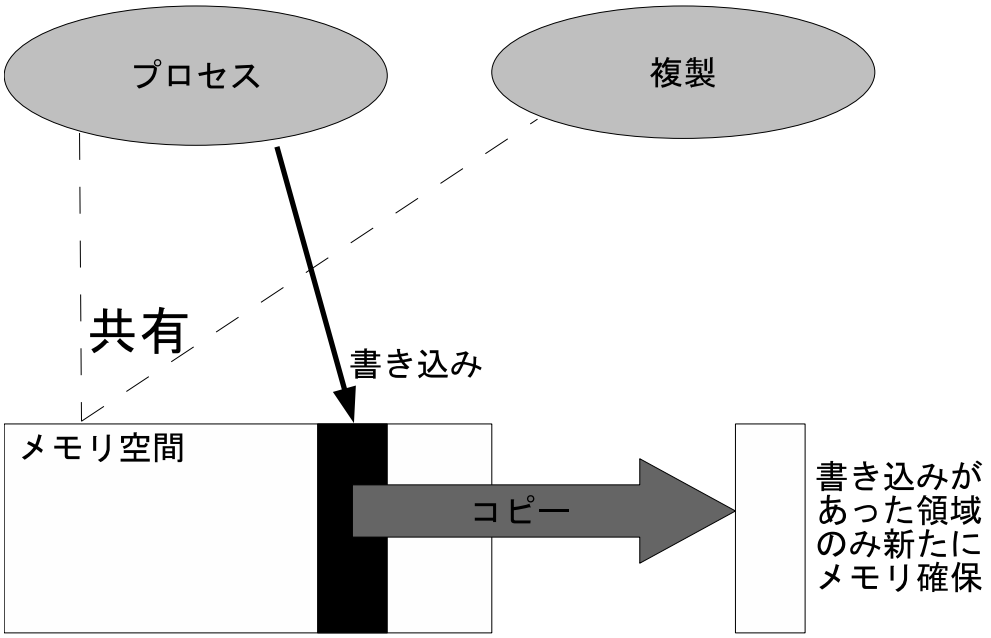


図 3.4: コピーオンライトによるメモリコピーの負荷軽減

降正常に実行することができる．しかし getenv 関数のように切り替え後も同じ入力が渡される可能性の高い関数の場合，もしこの入力が異常検知の原因となっていたとすると，複製プロセスに切り替え後も同じエラーを繰り返し検知する．

これを解決するために，getenv 関数が実行された際，その入力をカーネル内に確保した一時領域に保存する．複製プロセスに切り替え後再び getenv が実行された際にカーネル内に保存されている値と実行しようとしている値の比較を行う．もし同じであった場合は異常を引き起こす可能性があるのでエラーを返して getenv 関数の実行を終了する．現在は単純に比較しているだけであり，エラーの原因となっているかどうかを判別しているわけではない．今後改善が必要である．

## 3.3 提案版脆弱性範囲推測処理

本章ではまず白井らの脆弱性範囲設定方法と提案版実行継続処理を組み合わせた場合に発生する問題について説明し，その後提案する設定方法について説明する．

まず，白井らの脆弱性範囲設定方法と提案版実行継続処理を組み合わせた場合に発生する問題について，図 3.2 に示すサンプルプログラムを用いて説明する．図 3.2 のサンプルプログラムにおいて，funcA 関数と funcB 関数が脆弱性範囲に設定されていたとする．サンプルプログラムの実行を開始し，recv 関数で不正なデータを受信したとする．funcA を実行する直前に提案版実行継続処理によりプロセスの複製が生成される．その後実行を続け，recv 関数で受信した不正なデータが原因で funcB で異常を検知したとする．すると複製プロセスに実行を切りかえるが，複製プロセスを作成したのは recv 関数を実行した直後であり，不正を引き起こしたデータはメモリ上に残ったままとなっている．よって複製プロセスに切り替え後も同じポイントで繰り返し異常を検知する．

これを解決するために，攻撃を受ける可能性のある関数が脆弱性範囲の始点となるように範囲の設定方法を変更する．こうすることで攻撃を受ける直前で複製プロセスを作成することができるようになり，提案版実行継続処理によって除去できなかった不正データの除去も可能となる．

# 第4章

# 実装

本システムは，白井らのセルフヒーリングシステムを基に，カーネルへの追加実装と，白井らのシステムの動作を一部変更することで実装した．

## 4.1 複製プロセス

表 4.1 に示す 2 つのシステムコールを新たに実装した．以下，それぞれのシステムコールについて述べる．

### 4.1.1 create_back_up システムコール

create_back_up システムコールは Linux の fork システムコールを基に実装した，複製プロセスを生成を実現するためのシステムコールである．以下に Linux の fork システムコールにおける主要な関数を示す．

1. alloc_pidmap()：新たに PID を確保し，確保した PID を返す．
2. copy_process()：各種ディスクリプタ等のコピーと，生成した子プロセスの親プロセスへの接続を行う．
3. wake_up_new_task()：生成した子プロセスを起床させる．

生成されたプロセスを複製として用いるために，以下 2 つの処理について変更を行った．

表 4.1: 各機能を実現するシステムコール

| | |
|---|---|
| create_back_up() | 複製プロセスの生成 |
| start_back_up() | 複製プロセスへ切り替え |

**copy_process 関数**

通常 fork で生成されるプロセスは，fork を呼び出したプロセスの子プロセスとして管理される．この子プロセスをそのまま複製プロセスとして使用すると，複製プロセスへの切り替え前後でプロセスの親子関係に不整合が生じる．そこで親プロセスのプロセス情報を書き換える処理を変更する．fork では呼び出し元プロセスのプロセス情報内で管理されている子プロセスリストに生成されたプロセスを登録するが，create_back_up システムコールでは呼び出し元プロセスの子プロセスリストではなく，呼び出し元プロセスの親プロセスで管理されている子プロセスリストに登録する．

**wake_up_new_task 関数**

fork で生成された子プロセスは，生成処理が終了後，ランキューに子プロセスを登録し，スケジューラに対してスケジューリング要求を行う．これにより子プロセスは実行を開始することができる．複製として用いる場合は，生成した時点での状態を保持し続けて置く必要がある．そのため，create_back_up システムコールでは fork のようにプロセス生成後スケジューリング処理を行うことはせず，プロセスの状態を待機状態 (TASK_UNINTERRUPTIBLE) に変更後，create_back_up システムコールを終了する．複製プロセスのスケジューリング要求は start_back_up システムコール内で行う．

### 4.1.2 start_back_up システムコール

start_back_up システムコールは，呼び出し元プロセスの終了と複製プロセスの起床を行うシステムコールである．まず呼び出し元プロセスの子プロセスを複製プロセスの子プロセスとして登録する．次に複製プロセスの状態を実行待ち状態 (TASK_RUNNING) に変更しスケジューリング要求を行う．最後に do_exit 関数を呼び出して呼び出し元プ

ロセスの実行を終了する．

## 4.2 脆弱性範囲設定

### 4.2.1 設定方法変更

脆弱性範囲の設定処理において，今チェックしているライブラリ関数が攻撃を受ける可能性のある関数であれば設定処理を終了するという終了条件を追加する．

白井らのシステムでは，各ライブラリ関数をロードされているメモリ上のアドレスで管理している．よって追加する終了条件の判定は、今チェックしているライブラリ関数のアドレスが攻撃を受ける可能性のある関数のアドレスであるかどうかで行う必要がある．攻撃を受ける可能性のある関数がロードされているアドレスを管理するために，専用のテーブルを用意する．攻撃を受ける可能性のある関数のアドレスを取得するために，攻撃を受ける可能性のある関数がフックされた際の処理に変更を加える．白井らのシステムでは，ある関数をフックした際，フックされた関数がロードされているアドレスを取得する．そして各種チェックの終了後，取得したアドレスを使ってフックされた関数を呼び出す．提案手法ではこの処理を利用し，攻撃を受ける可能性のある関数がフックされた場合，フック処理中に取得したアドレスをテーブルに登録する．

脆弱性範囲設定の処理として，まずテーブルを参照し，もしテーブルに 1 つでもアドレスが登録されていれば，異常を検知したポイント以前に攻撃を受ける可能性のある関数が呼び出されていることになる．その場合は，異常を検知したポイントから遡り，テーブルに登録されたアドレスと一致するライブラリ関数を発見した場合はそこで脆弱性範囲設定処理を終了する．テーブルに 1 つもアドレスが登録されていなかった場合は異常を検知したポイント以前に攻撃を受ける可能性のある関数が存在しないことになる．この場合は白井らの設定方法を用いる．

### 4.2.2 複製プロセスにおける脆弱性範囲の絞り込み

白井らのシステムで脆弱性範囲の絞り込みは実装されているため，それを利用する．絞り込みを行うためには攻撃に関する情報が必要となるが，create_back_up システムコールにより生成された複製プロセスと呼び出し元プロセスではメモリ空間が異なる．そのため，呼び出し元プロセスから複製プロセスへ攻撃に関する情報を直接渡すことは出来ない．そこで専用のシステムコールを実装した．

**export_pos_info**

攻撃に関する情報をカーネル内にコピーする．kmalloc 関数を呼び出しカーネル空間内に一時領域を作成し，そこに攻撃に関する情報をコピーする．

**import_pos_info**

カーネル内にコピーされた攻撃に関する情報をユーザ空間にコピーする．

# 第5章

# 実験

提案システムを実装し，そのオーバーヘッド，攻撃されたときの動作とそれに対する考察を述べる．なお評価に使用した環境を表5.1に示す．Pentium4 3.0GHzのCPUと1GBのメモリを搭載した計算機に，CentOS5.4とLinux kernel 2.6.17.8をインストールし評価を行った．

## 5.1 攻撃評価

脆弱性をもつ実験用サーバプログラムを作成し，本提案システムによる監視の元，実験用サーバプログラムを実行した．なお，評価に使用したセキュリティホールの存在する実験用サーバプログラムと攻撃を行うためのクライアントプログラムは付録として添付する．またCentOS5.4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を以下に示す．

echo 0 > /proc/sys/kernel/randomize_va_space

### 5.1.1 攻撃手法

実験用サーバプログラムにはフォーマットストリング攻撃を行う箇所が可能なセキュリティホールがprint_message関数に存在している．外部から受け取ったデータが格納されている領域の先頭アドレスをprintf関数にそのまま渡している．よって2章で説

表 5.1: 実験環境

| | |
|---|---|
| OS | CentOS 5.4 |
| kernel | linux 2.6.17.8 |
| CPU | Pentium4 3.0GHz |
| memory | 1GB |

明したフォーマットストリング攻撃を行うことが出来る．このセキュリティホールを利用するためにクライアントプログラムでは，1 回目の send 関数で print_messages 関数が呼ばれた時にスタックに積まれる戻りアドレスと，print_messages 関数内のローカル変数 retval を改竄するための不正データを送信し，2 回目の send 関数で ffff という意味のない文字列を送信する．戻りアドレスを改竄することで，実験用サーバプログラム中に宣言されている not_called 関数の呼び出しを試みる．not_called 関数は実験用サーバプログラム実行時に呼び出されることはないため，この関数が実行されたかどうかで戻りアドレスの改竄が成功したかどうかを判断する．なお，not_called 関数は「Hello World!!」という文字列を出力するのみである．

### 5.1.2 評価実験

まず，監視せずにこの実験用サーバプログラムを実行した場合の結果を図 5.1 に示す．最終行に出力されているセグメンテーション違反のエラーメッセージの直前に「Hello World!!」という文字列が出力されていることがわかる．このことから戻りアドレスが改竄され，本来呼ばれるはずのない not_called 関数が呼び出されていることがわかる．次に白井らのシステムによる監視の元，実験用サーバプログラムを実行した結果を図 5.2 に示す．なお，recv 関数を呼び出してから print_messages 関数内のセキュリティホールまでを脆弱性範囲として設定している．print_messages 関数内でフォーマットストリング攻撃が行われ戻りアドレスが改竄されるが，実行継続処理によって改竄を検知し戻りアドレスを修復する．図 5.2 の下部において，print_messages 関数終了間際に出力される「print_messages finish」という文字列の直後に「print_messages's retval : 2060」という文字列が出力されている．「print_messages' retval : 2060」という出力

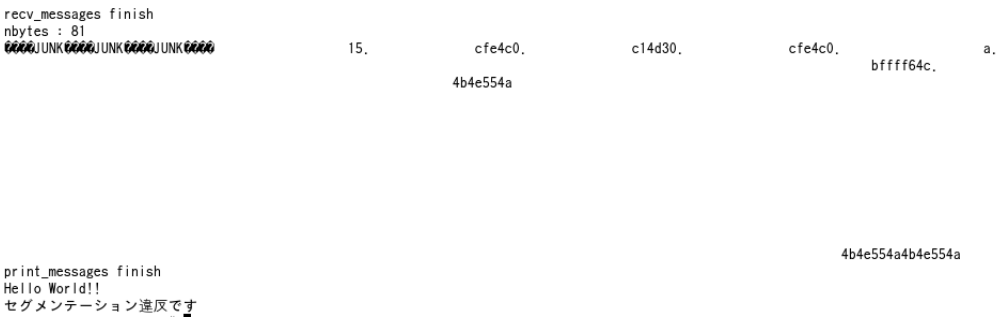


図 5.1: 監視を行わなかった場合の実行結果

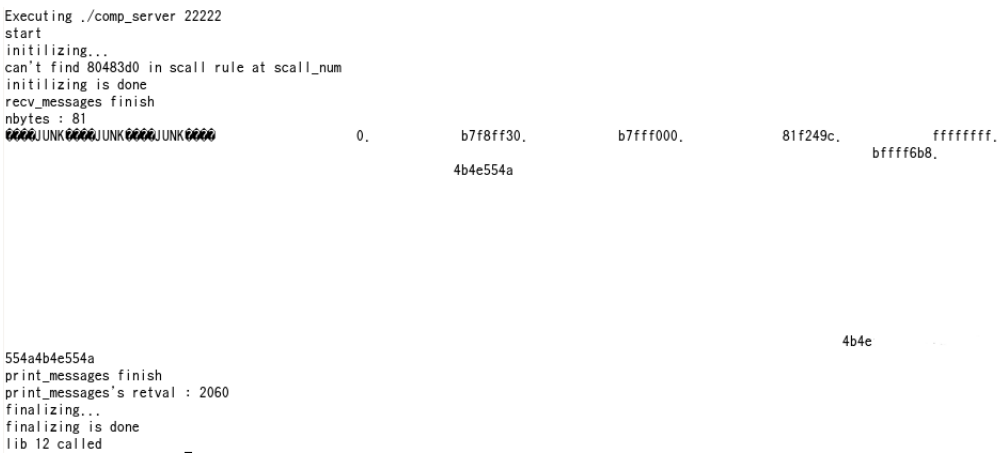


図 5.2: 白井らのシステムで監視した場合の実行結果

は，main 関数内において print_messages 関数の終了後に呼び出される printf 関数により出力され，直前の print_messages 関数の返り値を引数として受け取る．このことから，print_messages 関数を終了し正しくリターンしているが，print_messages 関数が 0 でない値を返していることが分かる．よって白井らのシステムでは戻りアドレスの改竄以外は対処できないことが確認できる．

最後に本提案システムの監視の元実験用サーバプログラムを実行した結果を図 5.3 に示す．セキュリティホールが存在する箇所は脆弱性範囲に指定されているため提案版実行継続処理によって戻りアドレスの改竄を検知し，複製プロセスへ実行を切り替えている．切り替えの発生は次の状況から判断できる．本来この実験用サーバプログラムが実行する recv 関数の回数は 1 回であるため，2 回目の send 関数で送られてくる ffff

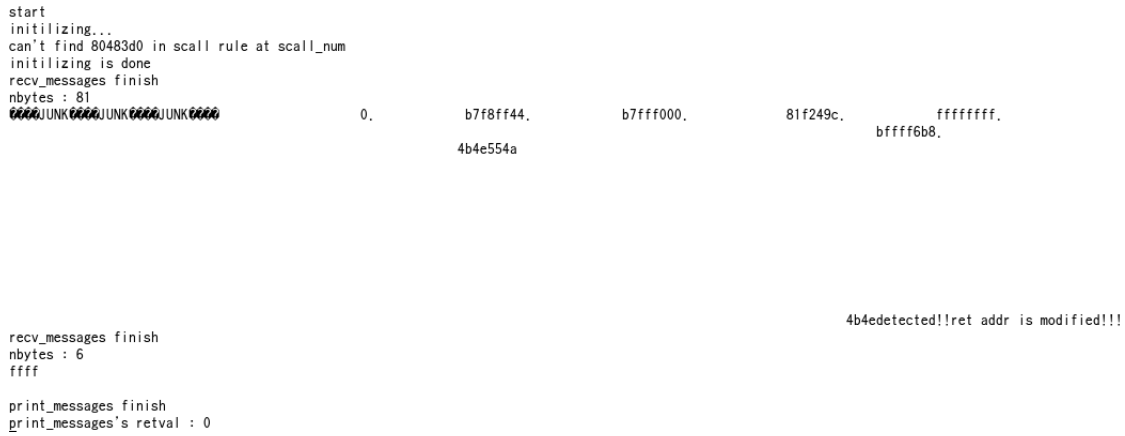


図 5.3: 提案システムで監視した場合の実行結果

表 5.2: オーバーヘッドの測定結果

| 生成 | 切り替え | 破棄 |
|---|---|---|
| 0.000125sec | 0.000263sec | 0.000007sec |

を受信することはない．しかし recv 関数の直前に複製プロセスを作成したため，複製プロセスにおいて再び recv 関数が呼び出され，ffff を受信する．図 5.3 では ffff を受信し出力していることから複製プロセスに切り替わり正常に実行されたことが分かる．

## 5.2 オーバーヘッド評価

複製プロセスの生成，破棄と複製プロセスへの切り替えに要する時間を計測した．複製プロセスの生成と破棄に要する時間はそれぞれのシステムコールを呼び出す前後に gettimeofday 関数を呼出し時刻を取得し，その差を計測している．複製プロセスへの切り替えに要する時間は，export_pos_info システムコールを呼び出してから import_pos_info システムコールの実行が終了するまでの時間を計測している．表 5.2 に計測結果を示す．本提案システムにおける提案版実行継続処理は，白井らのシステムにおける実行継続処理にプロセスの複製機能等を追加実装することで実現している．よって（新実行継続処理）のオーバーヘッドは白井らのシステムにおける実行継続処理に表 5.2 に示されるプロセスの複製機能等のオーバーヘッドを上乗せしたものとな

```
int main(){
    int i = 0;

    for(i = 0; i < 500; i++)
        puts("abc");
    return 0;
}
```

図 5.4: オーバーヘッドの測定に用いたプログラム

表 5.3: 図 5.4 のプログラムを実行した結果

| 監視せずに実行 | 白井らのシステムで監視 | 提案システムで監視 |
|---|---|---|
| 0.003sec | 0.098sec | 0.208sec |

る．次に，図 3.2 に示すプログラムを用いてオーバーヘッドの測定を行った．puts 関数を脆弱性範囲に指定し，提案版実行継続処理を 500 回繰り返した場合のプログラムの実行時間を計測した．表 5.3 に測定結果を示す． 提案システムの場合，puts 関数の実行直前で複製プロセスが生成され実行終了後に複製プロセスを破棄するという処理が 500 回繰り返されることになる．この処理の追加により，白井らのシステムと比べて約 2 倍のオーバーヘッドが発生していることが表 5.3 からわかる．また，表 5.2 から，提案システムにおける全体のオーバーヘッドは複製プロセスの生成時のオーバーヘッドが大半を占めていると考えられる．現在の create_back_up システムコールでは，復元に不必要であると考えられる情報も複製プロセスに含まれている．よって，必要となる可能性の高い情報のみを複製プロセスに含ませるよう create_back_up システムコールを改善することで，全体のオーバーヘッドを削減出来ると考える．

# 第6章

# まとめ

本研究では複製プロセスによるロールバック機能を有したセルフヒーリングシステムを提案した．fork システムコールを基にした複製プロセスを作成するためのシステムコールを新たに実装し，脆弱性となっている箇所が含まれていると推測される範囲の処理を実行する直前に複製プロセスを生成する．不正アクセスを検知した際はその複製プロセスに実行を切り替えることで監視対象プロセスの修復を行う．

この提案システムを Linux カーネル 2.6.17.8 に上に実装し，動作の検証とオーバーヘッドの測定を行った．その結果，白井らのシステムと比べてオーバーヘッドは大きくなったが，正しく実行を継続させられることが確認された．

今後の課題として，複製の作成用として最適化された複製作成システムコールの実装が挙げられる．現在は必要でないと考えられる情報も複製に含まれており，そのため複製作成のオーバヘッドは無視できない数値となっている．オーバヘッドを削減するために，これらを排除しオーバヘッドを削減した複製作成システムコールを実装する必要がある．

# 謝辞

本研究のために，多大な御尽力を頂き，御指導を賜った名古屋工業大学の齋藤彰一准教授，松尾啓志教授，津邑公暁准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた齋藤研究室並びに松尾・津邑研究室の方々に深く感謝致します．

# 参考文献

[1] redhat: Limiting buffer overflows with ExecShield, http://www.redhat.com/magazine/009jul05/features/execshield/.

[2] A. Baratloo, N. Singh, T. Tsai: Protecting critical elements of stacks, http://www.research.avayalabs.com/.

[3] Jesse Sathre, Joseph Zambreno: Automated software attack recovery using rollback and huddle, Springer Science+Business Media, LLC 2008.

[4] Alexey Smirnov, Tzi-cker Chiueh, DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks, Computer Science Department.

[5] 槙本裕司, 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, pp. 3–9 (2009).

[6] 白井宏憲, 齋藤彰一, 松尾啓志, 不正入力データ除去と関数戻りアドレス保護による self-healing システムの実現, 情報処理学会 研究報告, Vol. 2009-OS-112, No. 11.

# 付録

実験で使用したサーバプログラムとクライアントプログラムのソースコードを示す.

## サーバプログラム

```
/* TCP のサーバプログラム */

/* 以下のヘッダファイルは/usr/include にある */
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#define BUFFSIZE 128              /* メッセージの最大長 */

int recv_messages(int sockfd, char *buff){
  int retval;
  if((retval = recv(sockfd, buff, BUFFSIZE, 0)) < 0){
    puts("error on print_messages");
    exit(-1);
  }
  puts("recv_messages finish");
  return retval;
}
void not_called(){
  printf("Hello World!!\n");
}



int print_messages(char *buff, int nbytes){
  int retval=0;
  char tmp[BUFFSIZE];
  memcpy(tmp,buff,nbytes);
  printf(tmp);
  puts("\nprint_messages finish");
  return retval;
}
main(argc, argv)
  int     argc;
  char    *argv[];
{
  int i;
  int     port;                 /* 自ポート番号*/
  int     sockfd, acc_sockfd;   /* ソケット記述子 */
  struct sockaddr_in      addr, my_addr;
```

```
  /* インタネットソケットアドレス構造体 */
  int     addrlen;
  char    buff[BUFFSIZE];       /* 受信バッファ */
  int     nbytes;               /* 受信メッセージ長 */
  char tmp[BUFFSIZE];
  char *env_tmp;
  if (argc == 2){
    port = atoi(argv[1]);     /* 自ポート番号の設定 */
  } else {
    exit(1);
  }

  /* ソケットの生成 */
  if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    exit(1);
  }

  /* 自アドレスと自ポート番号の設定 */
  bzero((char *) &my_addr, sizeof(my_addr));      /* 0 クリア */
  my_addr.sin_family = AF_INET;                   /* アドレスファミリ */
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* アドレス */
  my_addr.sin_port = htons(port);                 /* ポート番号 */
  if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0) {
    exit(1);
  }

  /* ソケットを接続待ちの状態にする */
  listen(sockfd, 5);

  addrlen = sizeof (addr);
  bzero(buff, sizeof(buff));    /* 受信バッファの 0 クリア */

  /* クライアントからの接続待ち */
  if ((acc_sockfd = accept(sockfd,(struct sockaddr *) &addr,&addrlen)) < 0) {
    exit(1);
  }
  nbytes = recv_messages(acc_sockfd,buff);
  printf("nbytes : %d\n",nbytes);
  nbytes = print_messages(buff,nbytes);
  printf("print_messages's retval : %d\n",nbytes);
  close(acc_sockfd);  /* ソケット記述子の削除 */

  close(sockfd);               /* ソケットの終了 */
  return 0;
}
```

# クライアントプログラム

```
/* TCP のクライアントプログラム */

/* 以下のヘッダファイルは/usr/include にある */
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>
#include <strings.h>
#include <string.h>
#include <stdlib.h>
#define BUFFSIZE 1024             /* メッセージの最大長 */
```

```
int main(int argc, char *argv[])
{
  char    *host;                /* 相手ホスト名 */
  int     port;                 /* 相手ポート番号 */
  int     sockfd;               /* ソケット記述子 */
  struct sockaddr_in      addr, my_addr;
  /* インタネットソケットアドレス構造体 */
  int     addrlen;
  char    buff[BUFFSIZE];       /* 送信バッファ */
  int     nbytes;               /* 送信メッセージ長 */
  struct hostent  *hp;          /* 相手ホストエントリ */

  if (argc == 4){
    host = argv[1];          /* 相手ホスト名の設定 */
    port = atoi(argv[2]);    /* 相手ポート番号の設定 */
  } else {
    fprintf(stderr,"usage: %s host port\n",argv[0]);
    exit(1);
  }

  /* 相手ホストエントリの取得 */
  if ((hp = gethostbyname(host)) == NULL) {
    perror("gethostbyname");  /* 相手ホストエントリ取得失敗 */
    exit(1);
  }

  /* 相手アドレスと相手ポート番号の設定 */
  bzero((char *)&addr, sizeof (addr));            /* 0 クリア */
  addr.sin_family = AF_INET;                      /* アドレスファミリ */
  bcopy(hp->h_addr, (char *)&addr.sin_addr, hp->h_length);
  /* アドレス */
  addr.sin_port = htons(port);                    /* ポート番号 */

  /* ソケットの生成 */
  if ((sockfd = socket(PF_INET,SOCK_STREAM,0)) < 0) {
    perror("ソケット生成失敗");         /* ソケットの生成失敗 */
    exit(1);
  }

  /* 自アドレスと自ポート番号の設定 */
  bzero((char *) &my_addr, sizeof(my_addr));      /* 0 クリア */
  my_addr.sin_family = AF_INET;                   /* アドレスファミリ */
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);    /* アドレス */
  my_addr.sin_port = htons(0);                    /* ポート番号 */
  if (bind(sockfd,(struct sockaddr *)&my_addr,sizeof(my_addr)) < 0) {
    perror("設定失敗");         /* 自アドレスと自ポート番号の設定失敗 */
    exit(1);
  }

  addrlen = sizeof (addr);

  /* サーバとのコネクション確立 */
  if (connect(sockfd,(struct sockaddr *)&addr,addrlen) < 0) {
    perror("コネクション確立");
    exit(1);
  }

  nbytes = strlen(argv[3]);        /* 送信メッセージ長の設定 */

  argv[3][nbytes] = '\0';
  printf("%s\n",argv[3]);
  /* 送信 */
  if (send(sockfd,argv[3],nbytes+1,0) != nbytes+1) {
    perror("送信失敗");         /* 送信失敗 */
```

```
    exit(1);
  }
  printf("Send message: ");
  fgets(buff, BUFFSIZE, stdin); /* 送信メッセージの取得 */

  nbytes = strlen(buff);        /* 送信メッセージ長の設定 */

  printf("%s\n",buff);
  /* 送信 */
  if (send(sockfd,buff,nbytes+1,0) != nbytes+1) {
    perror("送信失敗");         /* 送信失敗 */
    exit(1);
  }

  close(sockfd);                /* ソケットの終了 */
}
```