Vol.2009-OS-112 No.11
2009/8/6

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

白井宏憲[†1]　齋藤彰一[†1]　松尾啓志[†1]

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，近年ではゼロデイ攻撃が多く発生している．これら未知の攻撃を防ぐ手法として異常検知型侵入防止システムがある．しかし，侵入防止システムでは，侵入を検知後に，管理者に侵入を通知したり，サービスを停止させるに留まる．このため，セキュリティパッチなどにより脆弱性が修正されるまでは，安定してサービスを提供できない．これらの理由により，プロセスの状態を修復する機能を持つ self-healing システムに関する研究が盛んに行われている．しかし，既存の self-healing システムでは処理時間や検知精度の点で問題がある．本論文では，プログラムを過去に攻撃を受けた付近にある脆弱性を含む可能性の高い範囲と，脆弱性を含む可能性の低い範囲の 2 種類に分類し，オーバーヘッドの低い self-healing システムを提案する．

Implementation of Self-Healing System Based on Eliminating Malicious Input Data and Protecting Return Address

Hironori Shirai,[†1] Shoichi Saito[†1]
and Hiroshi Matsuo [†1]

The damege by illegal access is more increase, as the internet spreads. In recentry, zero-day attacks become one of the most conspicuous attacks. There are Intrusion Prevention System(IPS) for protecting a system from the unknown attacks. However, IPS only notifies an administrator of it or stops the computer, when the system is attacked. Thus service couldn't be kept steady until the bug is fixed. That is why self-healing system is actually necessary. Self-healing system can recover the tampered system. But existing self-healing system have a problem in processing time. We argue that an execution divides into an unsafe part that may have vulnerability and a safe part. Our system is lightweight.

1. はじめに

近年，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．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．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できる．しかし，パッチが配布される前に行われるゼロデイ攻撃が多発している．さらに，パッチの配布自体が遅れる場合も多いため，パッチが配布されるまでの間も安全にサービスを運用するための仕組みが必要である．このため，未知の攻撃を検出可能な侵入防止システムが注目されている．しかし，それらのシステム[1)]では，侵入を検知後に，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．よって，セキュリティパッチを適用してセキュリティホールを修正するまでは，攻撃を受ける度に，サービスの停止もしくは再起動を繰り返すことになり，安定してサービスを提供できない．

現在，プロセスの状態を自動的に修復する self-healig システム[2)–7)]も研究が進んでいる．しかし，既存の手法[2)–7)]ではオーバーヘッドが高いという問題や検知精度が低いという問題がある．本提案システムでは，過去に受けた攻撃情報から脆弱性を含むコードの位置を推測する．推測した情報を利用し負荷の高い処理を行う範囲を制限して，オーバーヘッドの低い self-healing システムを提案する．推測範囲は複数回攻撃を受け最適化する．初め数回の攻撃ではサービスの再起動を余技なくされる場合もあるが，最適化された後は攻撃を受けても再起動なくサービスを続行できる．

脆弱性を含む可能性の低い範囲では，既存の侵入防止システムと関数の戻りアドレスのバックアップ処理のみ行う．脆弱性を含む可能性のある範囲では不正入力データのチェックと，ライブラリ関数内でのバッファオーバーフローの検出を追加で行う．侵入防止システムは本提案システムとの親和性の高い実行バイナリファイルを静的に解析した動作規則と，システムコールや関数の呼出し場所を比較するシステム[1)]を用いる．

本論文では第 2 章では既存の侵入防止システムと self-healing システムの特徴と問題点について述べ，第 3 章で我々の提案手法について述べる．第 4 章で実験結果とその考察を述べる．第 5 章でまとめと今後の課題を述べる．

†1 名古屋工業大学
Nagoya Institute of Technology

2. 既存システム

本章では既存の self-healing システムと侵入防止システムについて，それぞれの特徴と問題点について述べる．

2.1 既存の self-healing システム

本節では既存の self-healing のシステムについて述べる．Jun Xu ら[2] のシステムは，監視対象プロセスのメモリ配置を変更しながら繰り返し攻撃を受けることで脆弱性の位置を調べる．最終的には，ハードウェアモニタを使用してバッファオーバーフローの原因となった命令とその領域を検出し，シグネチャを作成する．この手法の問題は，相対アドレスによる攻撃などメモリ配置に依存しない攻撃に対応できない点とハードウェアによるサポートを必要とする点があげられる．Liang ら[3] の手法は，パケット長が正常実行時よりも長いパケットを異常と判断し，実行時にこれを遮断することで実行を継続させる．この手法では，確保されたデータ領域より大きなデータを受信した結果として発生する典型的なバッファオーバーフローは防げるが，データ長に依存しない攻撃は防ぐことができない．reactive immune system[4] では，全メモリとファイルの変更データを記録し，異常を検知した場合にはメモリをリストアする．その後，関数をエラーで返し，関数の実行を無効化する．このシステムでは，本論文と同様に脆弱性を含む命令の推測により負荷の軽量化を行う．しかし，推測の方法はプログラム中で固定長配列が使用されている部分を候補にするという静的な方式である．このため，一度推測が外れた場合は常に外れるという点，固定長配列と無関係な脆弱性の場合は推測できない点が問題である．Rx[5] では，一定時間ごとにメモリの状態を記録し，異常を検知した場合，記録したメモリ情報を使用し正常な時の状態までロールバックする．その後，reactive immune system の様に関数の実行を無効にするのではなく，環境を変更して該当関数を再実行する．プロセス自体を再起動すると一定時間サービスが停止してしまうのに対し，Rx では関数単位で再実行を行うため，修復に必要な時間が短くて済むという利点がある．しかし，動作の起点と言うべき異常の検知は例外シグナルや，CCured[8]，StackGuard[9] といった比較的単純な物を想定している．そのため，攻撃を受けても異常を検知できない場合がある．

2.2 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[1] の手法を述べる．このシステムは，システムコールとライブラリ関数の呼び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検査し異常を検知するホスト型の侵入防止システムである．

槙本らのシステムでは静的に監視対象となるプログラムのバイナリ実行ファイルを解析して動作規則を作成する．この動作規則には，ライブラリ関数の呼び出しアドレス規則，ライブラリ関数の呼び出し順序規則及びライブラリ関数から呼び出されるシステムコールの種類の 3 項目が含まれる．ライブラリ関数の呼び出しアドレス規則では，バイナリ実行ファイルを解析することで，ライブラリ関数を call しているアドレスを調べる．よって，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する．次に，ライブラリ関数の呼び出し順序では，バイナリ実行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る．これにより，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群とその時のコールスタックに含まれる戻りアドレスリストが一意に定まる．実際に呼び出されたライブラリ関数が規則に無い場合と，規則に存在するライブラリ関数であるがコールスタックに含まれる戻りアドレス群が一致しない場合は異常と検知する．最後に，ライブラリファイルを解析して，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群を得る．これにより，システムコールが発行された際に呼び出し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．以上の 3 つ項目をシステムコールが発行されるごとに確認する．

この方式では静的な解析を用いているため，false positive が発生しないという利点がある．また，該当プロセスからのすべてのシステムコールをチェックするため，攻撃者はシステムコールを用いずに攻撃を行うか，動作規則の範囲内で改竄を行う必要があるため，攻撃は容易でない．

3. 提 案 手 法

本稿では，プロセスを停止させずに処理を継続させる手法として，不正入力データの除去と，関数の戻りアドレスの保護を提案する．以後，これらの処理を**実行継続処理**と呼ぶ．また，これらの処理による負荷を軽減させるために，脆弱性を含む可能性の高い危険範囲の推測を併せて提案する．以後，この処理を**脆弱性推測処理**と呼ぶ．

3.1 システムの処理の流れ

本提案システムシステム全体構成について述べる．本提案システムは，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．しかし，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，

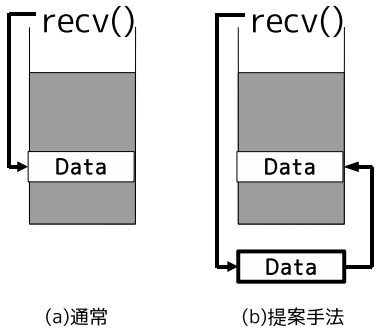

図 1 外部データのチェック

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．これによりオーバーヘッドの削減を図る．脆弱性の位置の推測には，槙本らの侵入防止システムを利用する．脆弱性の位置は，1 度の攻撃では正確に特定できない可能性があるが，何度か攻撃を受けて最適化する．以後，脆弱性を含む可能性が高いと判断されたプログラムコードを**危険範囲**と呼び，含む可能性が低いと判断された範囲を**安全範囲**と呼ぶ．以下，これらの各処理について詳細に述べる．

3.2 不正入力データの除去

本節では，シェルコードなどの悪意のあるコードがプログラムに入力され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，recv() や read() などの外部から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外部からのデータの検査を行う．

本システムにおけるこれらのシステムコールの処理では，データにシェルコードが含まれていた場合にはエラーを返す．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するなど，どのような処理を行うかは定かではないが，システムコール呼び出しに対する適切なエラー処理が行われていれば正常な処理に戻る．

具体的な動作を説明する．図 1 は，通常のシステムコールの動作と，本提案手法により変更されたシステムコールの動作を示している．通常 recv() システムコールが発行された場合 (図 1(a) 参照)，引数として指定されたバッファに受信したデータが書き込まれる．この時，受信データがシェルコードを含んでいる場合，これを取り除く．なお，シェルコードの判定は，現在は単に連続して NOP が含まれた場合としている．シェルコードは暗号化や難読化が行われている場合が多く，これらを検知するのは容易では無い．今後はより高度なシェルコードの検出が可能なシステムを実装していく予定である．

しかし，シェルコードを取り除く時，受信した data が書き込まれた領域に元々存在していたデータはシェルコードにより上書きされているため復元できない．これを解決するためのシステムコールの動作を図 1(b) に示す．recv() システムコールが発行されたとき，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を一時領域の中に書き込む．その後，受信データの中にシェルコードが含まれるか否かの検査を行う．受信データにシェルコードが含まれないことが確認された後で本来の受信領域にデータを書き込む．これによりユーザー領域内へのシェルコード混入を未然に防止する．

3.3 戻りアドレスの保護

本節では戻りアドレスの改竄を防ぐための処理を述べる．本システムで使用する，ライブラリ関数の書き込み先領域を確認して戻りアドレスの改竄を防ぐ手法と，改竄された場合に正常な値に戻す処理の二つの手法について述べる．

3.3.1 ライブラリ内でのオーバーフロー検出

libsafe[10] と同様の手法で，環境変数 LD_PRELOAD を用いて，本システムはライブラリ関数呼び出しのフックを行う．そして，ライブラリ関数の書き込み先領域に戻りアドレスと退避済み ebp が含まれていないか調べる．ライブラリ関数の書き込み先領域に，それらの値が含まれていない場合は，本システムは本来の libc のライブラリ関数を呼び出す．含まれていた場合は，不正と判断し，本来のライブラリ関数を呼び出さずにエラーを返す．この手法により，ライブラリ関数を使用して戻りアドレスの改竄を行うバッファオーバーフローを未然に防止する．

3.3.2 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と退避済み ebp の値を別領域に保存する．その後，関数が return される時に保存しておいた戻りアドレスの値と比較し，改竄を発見する．もし，値が一致しない場合には保存しておいた値を書き戻す．以上により戻りアドレスを改竄され，意図しない場所へ return されるのを防止する．この後，親関数において子関数のエラーチェックが適切に行われていれば，正常なフローへ戻ると予測される．本手法により，3.3.1 の手法では検出できないライブラリを用いない戻りアドレスの改竄に対応する．

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．そこで今後の改良として Michael ら[6] の手法によって推測，生成された戻り値を使用し関数の動作を再現したり，Sidiroglou ら[7] の手法を使い，その関数がエラー時に返す値を生成し，正常な動作に復帰できる可能性を高めていく予定である．

3.4 安全範囲と危険範囲

安全範囲では，比較的軽量な槙本らの侵入防止システムを利用し，さらに戻りアドレスの保存処理を行う．危険範囲では安全範囲での処理に加えて，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行う．この範囲分けにより，最初の攻撃では，軽量な修復処理である戻りアドレスの保存処理のみによって修復を試みる．ここで正常に修復に失敗した場合には侵入防止システムによって異常を検知し，プロセスの再起動を行う．そして，次に同じ攻撃を受ける時には，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行う．

3.5 危険範囲の推測

3.2 で示した不正入力データの除去と 3.3 で示した戻りアドレスの保護の処理を常時行う場合，オーバーヘッドが非常に高いと分かっている．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検査することになる．また，再帰や小さなライブラリ関数が連続して呼び出される場合も，頻繁にライブラリ関数の書き込み先領域を検査することになり大きなオーバーヘッドになる．そこで，これらの処理を脆弱性を含む可能性が高い場所に限定して行い，オーバーヘッドの削減を図る．

3.5.1 脆弱性の場所の推測

本項では脆弱性を含む可能性の高い場所を推測する流れについて説明する．まず，危険範囲がない状態でプロセスを実行する．もし，攻撃が行われた場合，侵入防止システムもしくは，戻りアドレス比較により異常を検知する．この時，異常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を基に危険範囲の推測を行う．なお，ここでの場所とは監視対象プログラムの実行アドレスとコールスタックに含まれる戻りアドレス群を指す．具体的な推測方法については 3.5.3 で述べる．侵入防止システムにより異常を検知した場合は，推測を行った後，本システムは対象プロセスを再起動させる．戻りアドレスのバックアップ処理により異常を検知した場合は，推測を行った後，対象プロセスの実行を再開する．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．

3.5.2 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設定すると述べた．しかし，この推測が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再起動を許すが，危険範囲の推測が成功した後は，プロセスを停止させずに実行を続ける．

```
1: usr_funcA()
2: {
3:  usr_funcC();
4:  usr_funcB();
5:  lib_func1();
6:  return;
7: }
```

```
 8: usr_funcB()
 9: {
10:   lib_func2();
11:   usr_funcC();
12:   return;
13: }
```

```
14: usr_funcC()
15: {
16:   if()
17:     lib_func3();
18:   lib_func4();
19:   return;
20: }
```

図 2 テストプログラム

一方，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．その結果，軽量かつ安全にプロセスを保護できる．

3.5.3 危険範囲の探索

3.5.1 および 3.5.2 で述べた脆弱性を含む範囲の推測方法について，サンプルコードを用いて説明する．侵入検知防止システムが異常を検知した場合，異常を検知した場所の付近で攻撃が発生している可能性が高い．そのため，本システムでは，異常を検知した場所から遡って n 個のライブラリ関数を呼び出すまでを，脆弱性を含む可能性がある危険範囲とする．このライブラリの数 n によって表される異常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．例えば距離が 10 であれば，異常を検知した場所から，ライブラリ関数呼び出し 10 回以内の範囲を危険範囲とする．なお，危険範囲に含まれるライブラリ関数は最小で距離と同じ値になるが，プログラム内に分岐が含まれているとその分だけ危険範囲に含まれるライブラリ関数は増える．

図 2 のプログラムを例として具体例な動作を説明する．usr_funcA()，usr_funcB()，

usr_funcC() はユーザー定義関数であり，lib_func1()〜lib_func4() はライブラリ関数である．また，各行の左側にある数字は行番号である．なお，ここでは探索する距離は 3 とする．まず，usr_funcA() の 4 行目から usr_funcB() が呼び出され，さらに 11 行目から user_funcC() が呼び出された後，19 行目の return で異常を検知したとする．異常検知後，本システムは，19 行目から逆向きに探索を開始する．まず 19 行目の return の直前にある 18 行目の lib_func4() を危険範囲に加える．なお，この危険範囲である lib_func4() は，コールスタックに usr_funcA() の 4 行目と usr_funcB() の 11 行目が入っている場合のみ危険範囲とする．この時，探索の起点である 19 行目からの距離は 1 となり指定距離 3 より小さいため探索を続行する．同様に 17 行目の lib_func3() を危険範囲に加える．この時，17 行目の lib_func3() は if 文の条件により呼び出されない場合があるので，距離は 1 のまま増加させない．ここで関数の先頭に達したため，関数の呼び出し元の 11 行目に戻って探索を続行する．次に，11 行目の前の命令である 10 行目の lib_func2() を危険範囲に追加する．lib_func2() はコールスタックに 4 行目が入っている場合にのみ危険範囲とする．この時，探索の起点からの距離は 2 となり探索を続行する．再び関数の先頭に達し，呼び出し元である usr_funcA() の 4 行目に戻って探索を続行する．4 行目の usr_funcB() の前の命令は 3 行目の usr_funcC() であるので usr_funcC() の内部を探索する．19 行目の return から探索を開始する．18 行目の lib_func4() を危険範囲に加える．この時の lib_func4() はコールスタックに 3 行目が入っている場合にのみ危険範囲とする．ここで起点からの距離が 3 に達したので探索を終了する．最終的な結果は，11 行目の usr_funcC() をコールスタックに 4 行目が含まれている時，17 行目の lib_func3() をコールスタックに 4 行目と 11 行目が含まれている時，18 行目の lib_func4() をコールスタックに 4 行目と 11 行目が含まれている時と，3 行目が含まれている時の計 4 つの状態が危険範囲となる．

この処理により，異常を検知した地点からライブラリ関数の呼出し n 個以内で到達出来るライブラリ関数を危険範囲として抽出する．また，同一の関数であっても呼び出し元のアドレスに応じて危険範囲か否を決定する．監視プログラムの実行中は，ライブラリ関数の呼び出しごとに，呼び出されたライブラリ関数が危険範囲に含まれているかを調べ，シェルコードの除去とライブラリ関数内でのオーバーフロー検査を行うかどうかを判断する．なお，実際の探索は図 2 に示されるようなソースコードではなくバイナリ実行ファイルを基に作成した動作規則を基に行う．

```
 1: main(){
 2:   char buf[];
 3:   int num;
 4:   while(){
 5:     //負荷 1
 6:     recv(buf); funcA(buf);
 7:     //負荷 2
 8:     recv(buf); funcB(buf);
 9:     recv(num); funcC(num);
10:   }
11: }
```

```
12: funcA(char str[]){
13:   printf(str);//脆弱性 A
14: }
15: funcC(char str[]){
16:   char out[8];
17:   memcpy(out,str);//脆弱性 B
18: }
19: funcB(int num){
20:   int data[5];
21:   data[num]=10;//脆弱性 C
22: }
```

図 3 評価プログラムの疑似コード

4. 評価と考察

脆弱性を持つクライアントサーバー型の評価プログラムを作成し，本提案手法の監視の下，攻撃された際の動作の評価を行った．評価は，戻りアドレスの改竄を行う攻撃に対してプロセスの実行を継続できることと，危険範囲が適切に設定されて，脆弱性を含む関数を特定できることである．さらに，危険範囲を設定したことによるオーバーヘッドを評価する．

評価プログラムの疑似コードを図 3 に示す．図 3 中の脆弱性 A では，printf() のフォーマットを指定する引数に外部から入力された文字列を使用している．このため，特定のメモリ領域を改竄可能な脆弱性が存在する．脆弱性 B は配列長の検査を行わずに配列のコピーを行っているためメモリ内容を改竄可能な脆弱性が存在する．脆弱性 C は配列の添字となる変数の値を外部から受信しているため，配列長を越えてメモリ内容を改竄可能な脆弱性が存在する．また，図 3 の負荷 1 では，サーバーは 15MB のファイルをクライアントへ 1 回転送を行い，負荷 2 では 10 回転送を行う．なおサーバーとクライアントは同一 LAN 上に存在しており 100Mbps のネットワークで接続されている．なお評価環境は，Fedora Core 5, Pentium 4 2.4GHz(HT 無効)，512MByte memory，Kernel バージョン 2.6.17.8 である．

また，Fedora Core 5 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ため，この機能を無効化にした状態で評価を行った．

表 1 ライブラリ関数一個当たりの実行時間

関数名	監視無し (usec)	提案システム	
		安全範囲 (usec)	危険範囲 (usec)
memcpy()	0.054	0.175	0.187
read()	1.95	3.89	101.7
write()	6.33	8.88	10.5
gettimeofday()	1.74	4.94	4.90
send()	87.2	87.1	87.1
recv()	87.3	87.4	187

表 2 脆弱性 A による被攻撃時の実行時間の推移

	被攻撃回数 (監視ライブラリ数)	実行時間 (sec)	倍率
監視無し	-(-)	14.75	1.00
提案手法	0 回 (0)	14.75	1.00
	1 回 (35)	17.41	1.18
	2 回以上 (1)	14.77	1.00

表 3 脆弱性 C による被攻撃時の実行時間推移

	被攻撃回数 (監視ライブラリ数)	実行時間 (sec)	倍率
監視無し	-(-)	14.75	1.00
提案手法	0 回 (0)	14.75	1.00
	1 回 (29)	17.39	1.17
	2 回以上 (2)	14.78	1.00

4.1 ライブラリ関数の実行時間

本システムでは，危険範囲をライブラリ関数の呼び出し個数で設定する．しかし，ライブラリ関数の種類によってオーバーヘッドが異なるため，監視するライブラリ関数の数が同じでも，含まれるライブラリ関数の種類と割合により発生するオーバーヘッドが異なる．これを示すために，予備評価としてライブラリ関数一個当たりのオーバーヘッドを測定した．主なライブラリごとのオーバーヘッドを表 1 に示す．表 1 内の “監視無し” は本提案手法を使用しなかった場合の実行時間である．また，“提案システム－安全範囲” は安全範囲に存在するライブラリ関数の実行時間である．この時間には，ライブラリ関数をフックする時間，戻りアドレスをバックアップする時間，ライブラリ関数が呼び出す使用するシステムコールの種類を調べる時間が含まれる．“提案システム－危険箇所” は危険範囲にに存在するライブラリ関数の実行時間である．この時間には安全範囲にあるライブラリ関数で行った処理に加えて不正入力データを除去する処理とライブラリ関数内でバッファオーバーフロー発生の検査を行う時間が含まれる．各結果はライブラリ関数を 10 万回実行した時の 1 回辺りの平均実行時間である．また，gettimeofday() を除く 5 つのライブラリでは 1024 バイトのバッファを扱った．

“提案システムの－安全範囲” の測定結果によると，システムコールを呼び出さない memcpy() のオーバーヘッドは，ライブラリ関数のフックと戻りアドレスのバックアップのみであるため，0.1usec 程度であった．一方，read()，write()，gettimeofday() はそれぞれ一回づつシステムコールを呼び出しており，それらの正当性を調べるための時間が約 2usec がオーバーヘッドとして表れている．また send() 及び recv() は，通信によるオーバーヘッドが大きいため本システムにおけるオーバーヘッドはあまり表れていない．

“提案システムの－危険範囲” の測定結果では，read() 及び recv() において，読み出されたデータにシェルコードが含まれるかどうかの文字列比較の処理を行うため，100usec 程度のオーバーヘッドが発生している．また，memcpy() はオーバーフローを検出する処理を追加で行うため，若干のオーバーヘッドが発生している．これらの結果により，ライブラリ関数 1 個あたりのオーバーヘッドはライブラリ関数の種類に依存すると言える．

なお，今回は評価を行っていないが，ループ内に存在する呼び出し回数の多いライブラリを危険範囲に含めた時もオーバーヘッドが大きくなる．これは，静的な解析ではループの繰り返し回数が分からないため，0 回として計上していることが原因である．

4.2 実行継続性の評価

本提案システムの危険範囲の設定方法と，プロセス保護方法と，実行継続処理の評価について述べる．評価プログラムを用いて，本提案手法での監視の元で実行を行った．なお，評価プログラムは，フォーマットストリング脆弱性とバッファオーバーフロー脆弱性の 2 つのライブラリ関数を使用する脆弱性と，ライブラリ関数を使用しない脆弱性 1 つを持ち，それぞれに対して検証を行う．これらを，戻りアドレスのバックアップ処理によって正常に修復できて再起動を必要としない場合と，正常に修復できず再起動が必要な場合の二つに分けて説明する．

なお，現在の実装ではローカル変数やヒープ領域のデータを保護できない．このため，関数ポインタ変数を書き換える等の攻撃には対応できない．

4.2.1 再起動なし

脆弱性 A を利用した攻撃が行われた場合，フォーマットストリング脆弱性の特性上，戻りアドレスや退避済み ebp などが格納されているメモリ領域をピンポイントで改竄される場合が多い．そのため，本実験でも同様の攻撃データを使用した．その結果，3.3.2 で示した戻りアドレスのバックアップにより，戻りアドレスと退避済み ebp を正常に修復して 3.5 で示した脆弱性推測処理を行った．その後，プロセスの実行を再開し，その後も正常に処理を続行できた．この時の，実行時間と監視範囲の遷移を表 2 に示す．まず，表 2 の “監視無

し” は本提案手法を使用しなかった場合の実行時間を示す．次に “提案手法− 0 回” は本提案手法で監視中するが 1 度も攻撃を受けていない状態であり，侵入防止システムと 3.3.2 で示した戻りアドレスのバックアップ処理のみが動作している．次に “1 回” では脆弱性 A を対象とした攻撃が行われ，一部の区間で 3.2 で示した不正入力データの除去と 3.3.1 で示したライブラリ関数内でのバッファオーバーフロー検査が実行されている場合である．そのため，“0 回” に比べ実行時間が増加していることが分かる．なお，危険範囲を示す距離はライブラリ関数 20 個分とした．また，この距離 20 に含まれるライブラリ関数の数を括弧内に示す．次に “2 回以上” では，監視ライブラリ数が 1 になっており，またオーバーヘッドが減少しているのが分かる．これは，本提案システムが，脆弱性の位置を特定し，危険範囲を脆弱性の存在する printf() 関数のみに限定したことを示す．

また，ライブラリを使用しない脆弱性 C の場合も，num の値によりメモリ領域をピンポイントで改竄する攻撃データを使用した．この結果，戻りアドレスと退避済み ebp を正常に修復し，その後も正常に処理を続行できた．

4.2.2 再起動あり

脆弱性 B を利用した攻撃が行われた場合，配列 out の領域を越えて書き込むことで，戻りアドレスや退避済み ebp，またスタック内に含まれるローカル変数などが改竄される恐れがある．本実験では，バッファオーバーフローにより funcB() の戻りアドレスを書き換えるコードとリモートからシェルを起動するシェルコードを入力データとして与えた．攻撃は，シェルを起動するシェルコードを事前に送り，脆弱性 B でバッファオーバーフローを発生させて戻りアドレスを書き換え，シェルコードを実行させる．本システムで監視の元で攻撃を行った結果，3.3.2 で示した戻りアドレスのバックアップにより戻りアドレスと退避済み ebp を修復し，シェルコードの実行を防いだ．本システムは 3.5 で示した脆弱性推測処理を行った後，サーバープログラムの実行を再開した．しかし，ローカル変数が改竄されていたため，サーバープログラムは異常終了した．そのため，本システムは対象プロセスの再起動を行った．次に攻撃を受けた時には，3.2 で示した不正入力データの除去および 3.3.1 で示したライブラリ関数内でのオーバーフロー検査が有効となり，recv() 内でシェルコードを除去し，memcpy() 内でのバッファオーバーフローを事前に防いだ．この時の実行時間と監視範囲の遷移を表 3 に示す．“2 回以上” は，監視ライブラリ数が 2 になっている．これは，本提案システムが，脆弱性の位置を特定し，危険範囲をシェルコードが挿入された recv() とバッファオーバーフローが発生した memcpy() の二つのみに限定したことを示す．

表 4 複数の脆弱性に対するオーバーヘッド 1

	状態遷移 (監視ライブラリ数)	実行時間 (sec)	倍率
監視無し	-(-)	14.75	1.00
提案手法	0(0)	14.75	1.00
	B(29)	17.39	1.17
	B(2)	14.78	1.00
	A(35)	17.39	1.17
	A(3)	14.78	1.00

表 5 複数の脆弱性に対するオーバーヘッド 2

	状態遷移 (監視ライブラリ数)	実行時間 (sec)	倍率
監視無し	-(-)	14.75	1.00
提案手法	0(0)	14.75	1.00
	B(29)	17.39	1.17
	A(44)	18.55	1.25
	A(30)	17.27	1.17
	B(3)	14.80	1.00

4.3 複数の攻撃に対する評価

本提案システムが複数の脆弱性に対する攻撃に対応可能であることを示す．脆弱性 A 及び脆弱性 B の 2 種類の攻撃が両方行われた場合の実行時間と監視ライブラリ数の遷移を，表 4 及び表 5 に示す．なお，2 種類の脆弱性が存在する場合，全体で 6 通りのパターン[⋆1]があるが本評価では表 4 及び表 5 に示す 2 つのパターンを対象とする．まず，表 4 に脆弱性 B に対する攻撃が 2 回行われた後，脆弱性 A に対する攻撃が 2 回行われた場合の実行時間と監視ライブラリ数の推移を示す．最初に脆弱性 B に対する攻撃が行われ，これによって一時的にオーバーヘッドが増加している．しかし，同じ脆弱性 B に対する攻撃が行われたため，脆弱性 B の位置を特定してオーバーヘッドは減少している．この後，脆弱性 A に対する攻撃が行われ，再びオーバーヘッドが増加するが，再度，脆弱性 A に対する攻撃を受けた後は元のオーバーヘッドに戻っている．この時，監視ライブラリ数は 3 となっており，フォーマットストリング脆弱性を持つ printf()，シェルコードを受信した recv() とバッファオーバーフローが発生した memcpy() の 3 つを特定できている．

次に，表 5 に脆弱性 B に対する攻撃が行われた後，続いて脆弱性 A に対する攻撃が 2 回行われ，最後に脆弱性 B に対する攻撃が再び行われた時の実行時間の推移を示す．脆弱性 B に対する攻撃が行われた後，続いて脆弱性 A が行われたため，危険範囲は脆弱性 A と B の両方が対象となる．したがって，片方のみの攻撃が行われた場合よりもオーバーヘッドが増加しているのが分かる．なお，脆弱性 A と B に対する攻撃が単独で行われた場合のオーバーヘッドがそれぞれ 2.5 秒程度であるのに対し，両方同時に攻撃された場合のオーバーヘッドが 4 秒程度しか増加していない理由は，脆弱性 A による危険範囲と脆弱性 B による

⋆1 攻撃の発生順が，AABB，ABAB，ABBA，BBAA，BABA，BAAB の 6 通り考えられる

表 6 危険範囲のサイズによる実行時間の違い

	距離	監視ライブラリ数	実行時間 (sec)	倍率
監視無し	-	-	14.72	1.00
提案手法	0	0	14.73	1.00
	10	14	16.28	1.11
	20	29	17.59	1.19
	30	46	20.60	1.40
	40	64	23.60	1.60
	50	79	24.77	1.68
	60	96	27.81	1.89
	70	110	29.40	2.00
	80	124	30.50	2.07
	90	129	30.48	2.07
	100	129	30.44	2.07
	全体	132	30.47	2.07

危険範囲の一部が重なっているためである．この後，再度同じ攻撃が行われたため，オーバーヘッドは再び減少している．この場合も表 4 の場合の評価と同じく正しく 3 つの脆弱性を持つライブラリを特定できている．どの様な順番で攻撃が行われたとしても，最終的には脆弱性を特定できてほぼ等しいオーバーヘッドとなる．

4.4 危険範囲の大きさによるオーバーヘッド

危険範囲と安全範囲の分割によるオーバーヘッド削減の効果と，危険範囲の大きさによるオーバーヘッドを調べるために評価を行った．これを示すために，危険範囲の大きさを示す距離を 10 刻みで変更した場合のオーバヘッドを測定した．なお，評価プログラムの脆弱性 B に対する攻撃を用いた．この結果を表 6 に示す．表中の “監視ライブラリ数” とは危険範囲に含まれるライブラリ関数の呼び出し個数である．なお，監視ライブラリ数は分岐が存在しない場合は距離と等しい値になるが，分岐が存在すると距離よりも大きい値をとる．また，表中の “全体” とはプログラム全体を危険範囲とした場合，つまり，脆弱性の位置の推測を行わずに全ての場所で実行継続処理を行った場合を示す．プログラム全体を危険範囲とした場合の結果より，このプログラムの最大オーバーヘッドは 2.07 倍程度であると分かる．また，距離を小さくすればオーバーヘッドが減少することが分かる．しかし，距離を小さくすると，脆弱性の位置を特定するまでに必要な攻撃回数が増える場合もありトレードオフの関係にある．今後，主な脆弱性について異常検知場所と脆弱性の場所の距離を調べる必要がある．

なお，距離 90 から距離 100 に増えた場合では，監視ライブラリ数とオーバーヘッドが増えていない．これは，プログラムの開始点 (図 3 の 1 行目) と，メインループ全体 (図 3 の 4 行目から 10 行目) が危険範囲に含まれ，これ以上探索する範囲が存在しないためである．また，“距離 100” と “全体” で監視ライブラリ数に差があるのは，メインループの後に行われる終了処理で呼び出されるライブラリ関数が存在するためである．

5. ま と め

本稿では攻撃を受けてもプロセスの実行を継続させる手法と，この手法による負荷を減らすために脆弱性の位置を推測する二つの手法を提案し，オーバーヘッドの低い self-healing システムを実現した．プロセスの実行を継続させる手法として不正入力データの除去と，関数の戻りアドレスを保護する手法を提案した．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う．また，戻りアドレスの改竄を防止するために，ライブラリの書き込み先領域と戻りアドレスの格納領域を比較し，ライブラリ関数内でのバッファオーバーフローを防いだ．さらに，戻りアドレスのバックアップを行い，ライブラリ関数を使用しない戻りアドレスの改竄に対応した．また，これらの処理による負荷を軽減させるために，過去の攻撃情報から脆弱性の位置を推測した．この推測範囲は同じ攻撃を複数回受けて最適化を行う．

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，最初の攻撃ではプロセスの動作を変更され，プロセスの再起動をしたが，2 回目の攻撃時には，シェルコードの除去と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した．また，オーバーヘッドを測定した結果，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後は無視できる程度であった．

今後は，より一般的なプログラムでの検証や，ローカル変数など保護する対象を増やす予定である．また本提案手法ではシステムに必要な情報の一部がユーザー領域に置かれている．今後，古屋らの手法[11] を組み合わせてこれらのデータを保護を行う予定である．

参 考 文 献

1) 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 2009-OS-110, Vol.2009, No.6, pp.3–10 (2009).
2) Xu, J., Ning, P., Kil, C., Zhai, Y. and Bookholt, C.: Automatic diagnosis and response to memory corruption vulnerabilities, *Proceedings of the 12th ACM con-*

ference on Computer and communications security, pp.223–234 (2005).
3) Liang, Z., Sekar, R. and DuVarney, D.: Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems, *USENIX Annual Technical Conference*, pp.375–378 (2005).
4) Sidiroglou, S., Locasto, M., Boyd, S. and Keromytis, A.: Building a reactive immune system for software services, *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp.149–161 (2005).
5) Qin, F., Tucek, J., Zhou, Y. and Sundaresan, J.: Rx: Treating bugs as allergies–a safe method to survive software failures, *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Vol.39, No.5, pp.235–248 (2005).
6) Locasto, M., Stavrou, A., Cretu, G., Keromytis, A. and Stolfo, S.: Return Value Predictability Profiles for Self–Healing, *Proceedings of the 3rd International Workshop on Security: Advances in Information and Computer Security*, Springer, pp. 152–166 (2008).
7) Sidiroglou, S., Laadan, O., Keromytis, A. and Nieh, J.: Using Rescue Points to Navigate Software Recovery (Short Paper), *Proceedings of the IEEE Symposium on Security and Privacy*, pp.273–280 (2007).
8) Condit, J., Harren, M., McPeak, S., Necula, G. and Weimer, W.: CCured in the real world, *ACM SIGPLAN Notices*, Vol.38, No.5, pp.232–244 (2003).
9) Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, *Proceedings of the 7th USENIX Security Conference*, pp.63–78 (1998).
10) Tsai, T. and Singh, N.: Libsafe: Transparent system-wide protection against buffer overflow attacks, *roceedings of the 2002 International Conference on Dependable Systems and Networks*, p.541 (2002).
11) 古屋雄介，齋藤彰一，松尾啓志：侵入防止システムにおける動作規則保護機構の開発，情報処理学会研究報告 2009-OS-112(発表予定) (2009).