2006－OS－101（8）
2006／2／17

侵入の検知を契機にサーバを安全な状態に回復する機構

櫻庭健年[†,††]　道明誠一[†]　櫻井幸一[††]

プロセスがバッファオーバフロー攻撃を受けた場合、プロセスを強制終了させるのが通常の判断であるが、可用性に不満が残る。そこで、攻撃の検知を契機に、サーバを安全な状態に迅速に回復するＯＳ機能「空蝉」を提案する。「空蝉」は、攻撃の検知のためにシステムコールアドレスを監視し、安全な状態の退避・回復には fork の機構を利用する。実装、適用、および性能について報告する。

Server Recovery at the Intrusion Detection

Taketoshi Sakuraba,[†,††] Seiichi Domyo[†] and Koichi Sakurai[††]

"Utsusemi (cicada's cell)," a recovery-based countermeasure for the buffer overflow attacks is proposed. It does not only detect the attacks but also immediately recovers the attacked process to a safe state saved before the attack. The fork mechanism is used for saving the status of the process, and the detection is done by monitoring the attributes of the location where the system was called. Implementation, application and performance evaluation are reported.

1. はじめに

バッファオーバフロー攻撃は、依然としてコンピューティング環境における最大の脅威の一つである。バッファオーバフローはプログラムの実装上のバグであり、慎重なプログラム開発が普及すれば、このような脆弱性は時とともに姿を消していくという見方もありうるが、現実はそうではない。たとえば、CERT の脆弱性ノート[1] の報告数の推移を見ると (図 1)、2005 年は、全 256 件中、バッファオーバフローに関係するものが少なくとも 73 件 (28.5%)、2004 年は、全 345 件中、同 93 件（27%）であった。警告が盛んに出ており、慎重なソフトウエア開発が行われていると想像されるにもかかわらず、比較的高い割合で常時発生し続けている様子が窺える。

そのため、バッファオーバフロー攻撃に対しては、多くの対策が提案されてきており[2]、様々なコンポーネントにおける、様々な対策アプローチと実装がある。バッファオーバフロー攻撃に対する重要な対策アプローチとして「検知」があり、検知方式にも色々なものが知られている。多くの場合、バッファオーバフローの検知後は、攻撃を受けたプロセスを強制終了させる。しかし、これではサーバが止まってしまい、サービス妨害（DoS）攻撃が成功したことになる。

そこで、攻撃を検知した際、プロセスを単に終了させるのではなく、サービスを継続させる手段が必要である。

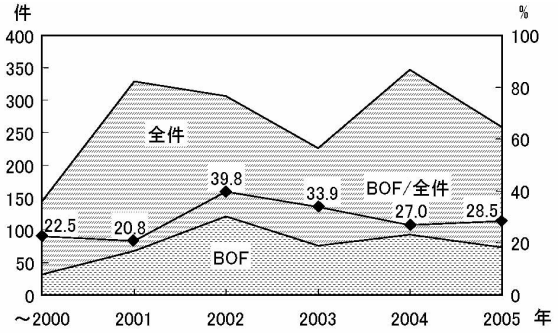

図 1　CERT 脆弱性ノート発行数の推移

一般に、可用性の維持は、秘匿性、完全性とともに、セキュリティの大きな目的の一つであり、回復はシステム可用性向上のための主要技術であるから、両者の組み合わせは自然な発想といえる。しかし、サーバの二重化のような手段では、大掛かりでコストもかかる上、同様の攻撃によってバックアップサーバもダウンしてしまう可能性が高い。もっとセキュリティに配慮した、単純で粘り強い回復手段が必要である。

本稿では、サーバへのバッファオーバフロー攻撃の検知を契機に、サーバを迅速かつ安全に回復してサービスの可用性の維持を図るＯＳ機能「空蝉」を提案する。「空蝉」は、サービス開始前のプロセスの状態を退避し、侵入を検知したならば、直ちにプロセスを退避した時点の状態に回復し、サービスを継続させる。

一般に、回復処理は、プロセスあるいはサービスが持つ状態の多さに応じて複雑になる。単純な構成のサーバならば、プログラムを改変することなく、「空蝉」の保

† 日立製作所システム開発研究所
Hitachi Systems Development Laboratory
†† 九州大学
Kyushu University

護の対象にすることができる。一方、複雑な回復処理が必要なサーバの場合は、プロセスの自動的な回復とともに、サーバ自身による回復処理が避けられない。そのため、サーバが「空蝉」の基本機能を利用して、回復処理を実行することも可能としている。

攻撃の検知を直ちに回復処理に結びつけるという「回復」ないし「検知&回復」のアプローチは、他の脅威に対する対策を検討する上でも有効であり、セキュリティ対策検討のための、主要アプローチの一つと考えられる[3]。

以下、2章では、これまでに知られているバッファオーバフロー攻撃対策をアプローチ毎に概観し、新たな対策アプローチとして、検知&回復アプローチを提案する。3章では、バッファーオーバフロー攻撃の検知方式に焦点を絞り、攻撃手法との関係から検知方式の特性と限界について検討する。4章では、検知&回復アプローチに基づくバッファオーバフロー攻撃対策機能「空蝉」を説明する。5章では実際のサーバへの「空蝉」の適用実験について報告し、6章では性能評価結果を示す。7章では関連研究について触れ、8章でまとめと今後の課題を述べる。

2. 対策のアプローチ

2.1 Attack-based Countermeasure Analysis

筆者らは、具体的な脅威に対する対策を探索する手法として、「Attack-based Countermeasure Analysis」、すなわち脅威を実現する「攻撃の瞬間」を中心とした時間経過に沿って「各時点で実施可能な対策群」を対策アプローチとして特定し、各アプローチに従ってセキュリティ対策を探索する手法、を提案している[3]。そこでは、攻撃の前になしうる対策アプローチとして (1) 予防 (2) 排除 (3) 阻止 を、攻撃直後の対策アプローチとして (4) 検知 (5) 回復 (6) 耐忍 (7) 限定 を、さらに、攻撃後、一定の時間を経てからの対策アプローチとして (8) 監査 (9) 対処 を、また、以上の時系列的整理とは独立したアプローチとして (10) 保全 (11) 強化 (12) 運用 を、それぞれ特定している。

既存の対策の多くは上記で整理することができる。対策の検討は、これらのアプローチに従って対策を考えるのであるが、状況、前提条件と課題が限定されているので考えやすい。また、他の脅威に対するそのアプローチの既存の対策を参考にすることができ、それによって新しい対策技術を「発見」することが重要である。

2.2 バッファオーバフロー対策

上記をバッファオーバフロー攻撃対策について試みたものが表1である。バッファオーバフロー攻撃における「攻撃の瞬間」を、バッファ以外のメモリを受信したメッセージによって上書きする時とする。以下、上書きの瞬間を中心とした時間経過に従って、上に述べた各アプローチごとに実施可能な対策を概観する。

バッファオーバフロー攻撃に対する「予防」には、バグのあるプログラムを作らないこと、作ってしまったバグは配布後も含めて修正すること、などが該当する。

「排除」とは危険なものは近づけないことであり、相手認証や、危険な通信の遮断[4] などが該当する。また、メッセージと内部データの分離が考えられる[5]。これは両者の完全性（integrity）レベルの違いからも支持される対策である。

「阻止」とは、攻撃を成功させないことであり、バッファ以外のメモリの上書きを、ハードウエアのメモリ保護機構などによって阻止すること[6] などが該当する。

「検知」とは、攻撃を受けたことを、攻撃成功後、攻撃されたプロセスがまだ存在し、進行中の攻撃に対する直接的対策の可能性があるうちに検知することである。各種の検知方式について、次章で詳述する。

「回復」とは、攻撃を検知したときに、サーバを安全な状態に迅速に回復して、サービスの継続を図ることである。我々はこれを新たに独立して取り上げるべきアプローチとして提案している[3]。本稿で提案するＯＳ機能、「空蝉」のアプローチである。

「耐忍」とは、攻撃者の真の目的を達成させないことである。これも、新たに独立して取り上げるべきアプローチとして提案している[3]。たとえば、バッファオーバフローでメモリの上書きがなされても、メモリ構造を攻撃者の想定するものと異なるものにしておけば、真の目的であるサーバの制御の奪取は回避することができる[7]。

表 1 バッファオーバフロー対策のアプローチ

アプローチ	対策	対策例
予防	セキュアプログラミング	コンパイル時に脆弱性のある危険なコードをチェック
	セキュリティホール修正	セキュリティパッチを適用
排除	IP フィルタリング	ファイアウォールなど
	入力のフィルタリング	文字以外を含むメッセージ、長すぎるメッセージなどを遮断
	バッファとスタックの分離	バッファ専用の領域を設け、隔離する
		戻りアドレス用の領域を設ける
阻止	メモリ保護	戻りアドレスの領域の書き込み保護
検知	テキスト領域外実行の監視	スタック、ヒープでの命令実行、システムコール呼び出し、ライブラリ呼び出しを検出
	振舞い監視	侵入後のプログラムの振舞いの変化の検出
	自己監視コードの挿入	サブルーチンコールの前後で「カナリア」をチェック
回復	機能回復	侵入検知後、プロセスを安全な状態に回復
耐忍	スタック構造の非標準化	スタックの成長方向の変更
		戻りアドレスの退避場所の変更
限定	権限奪取防止、最少特権	侵入しても、root の権限を限定し、被害範囲を最小化
	メモリ保護、sandbox	侵入しても、アクセス可能範囲を限定
監査	Audit 機能、IDS	アクセスを記録し、不審な動きをチェックする
保全	対策用データの保護	検知用データを別セグメントへ退避

「限定」とは、検知ができなかった場合の被害を限定することである。事前にサーバの権限を最小に絞っておくことが行われる。いわゆる最少特権の原理であり、これをきめ細かく実施可能としたものが「セキュアＯＳ」の特徴技術となっている[8]。

「保全」とは、セキュリティ対策そのものを保護することである。たとえば、バッファオーバフロー攻撃検知のための比較対象データの保護がある[9]。

2.3 検知＆回復アプローチ

我々はセキュリティと可用性を直接を結びつける技術分野として「検知＆回復」アプローチを提案している。可用性は、秘匿性、完全性とともにセキュリティの目的とされ、関係が深いが、可用性は信頼性のコンテキストで議論されることが多く、また、それぞれが独立した技術分野として確立していることもあり、両者を密接に関連付け、同時に議論することは少ないように思われる。

すでに指摘したように、サーバへの攻撃を検知したときに、サーバプロセスを強制終了してはサービスの可用性の喪失につながる。もちろん、セキュリティの観点から見れば、これは攻撃の進行を直ちに停止させ、また、まだメモリ上に潜んでいるかもしれない、脅威を取り除くためには一番確実な方法である。しかし、実際には、サービスの維持という別のニーズがあり、セキュリティに配慮しつつ、サービスの回復を試みることは意味のあることである。

一般に、攻撃を検知した時点では、攻撃はまだ継続しており、また再攻撃の可能性も高い。そのため、攻撃を確実に止め、かつシステムを安全な状態に確実に修復しなければならない。このようなセキュリティ上のリスクを押さえ込みつつ、サービスを安全に継続させるのには、特別な配慮が必要である。実際、セキュリティ上の危険がある状況での安全な回復は単純ではない。

たとえば、ホットスタンバイ方式によってサーバを二重化しておき、攻撃の検知の際は、攻撃されたサーバを強制終了し、バックアップサーバによってサービスを継続することが考えられる。しかし、バックアップサーバはセキュリティホールを抱えたまま稼動を続けることになるため、同様の攻撃によって、今度はバックアップサーバが強制終了を余儀なくされることになる。これから、ホットスタンバイ方式は、障害が再発する可能性が極めて小さい場合には有効であるが、バッファオーバフロー攻撃のように再現性の高い場合には適さないことが分かる。また、サーバのセキュアな回復は、セキュリティと可用性の、それぞれの技術の単純な組み合わせでは達成されないことが分かる。

3. バッファオーバフロー検知方式

バッファオーバフロー攻撃に対する直接的な対策の第一は、攻撃の検知である。様々な方法が知られているが、攻撃手法も巧妙化している。

3.1 攻撃手法

バッファオーバフローバグがあれば、長過ぎるメッセージを送りつけるだけで、サーバは誤動作する。多くの場合、バスエラーなどを起こしてサーバは異常終了するのみである。

メッセージに悪性コードを潜ませておき、バッファオーバフローによって、サーバプログラムが悪性コードに分岐するようにメモリを上書きすると、悪性コードを実行させることができる。このようにサーバプロセスの中で制御を得たコードを侵入コードと呼ぶことにする。侵入コードは侵入したプロセスのデータを破壊・改竄することができる。さらに侵入コードがシステムコールやライブラリを呼び出せば、被害は侵入されたプロセス内にとどまらず、システムの資源におよぶ。

侵入コードの実行が伴う場合、コードの実行そのものを検知することが考えられる[10]。しかし、これに対抗して、侵入コードの実行を伴わずにサーバに侵入する手段が知られている。return-to-libc 攻撃では、戻りアドレスをライブラリ関数のアドレスに書き換え、リターンのときにシステムコールを実行することができる。さらに連鎖型 return-to-libc 攻撃[9] になると、複数のシステムコールを任意の順序で実行することができるようになる。

バッファオーバフロー攻撃では、戻りアドレスを改竄してコンテキストを奪うことが多く、このような場合には、戻りアドレスの保護が有効である。しかし、バッファオーバフローでは、戻りアドレス以外のデータの上書きも発生しており、これらのデータの改竄によっても、サーバを操ることができる。たとえば、書き込みアドレスを改竄すれば、任意の場所のデータを更新することができる（ポインタ書き換え攻撃[9]）。

3.2 検知方法

監視する対象によって検知方法を分類する。

バッファ、あるいは保護すべきデータの周辺のデータの破壊を監視し、バッファオーバフローによって上書きされたことを検出する方法がある。戻りアドレスとフレームポインタが第一の監視対象であるが、ポインタ書き換え攻撃を考慮すると、スタック上、ヒープ上の区別なく、更新可能なデータはすべて監視の対象となりうる。監視の方法としては、バッファ周辺のデータを、あらかじめ退避しておいたデータのコピーと比較するのが一方法である。しかし、コピーをともに改竄するなど、隠蔽手段があり、これに対抗する手段も研究されている[11]。

また、命令やシステムコールなどを実行しているコードのアドレス領域を監視する方法がある。スタックやヒープなど、テキスト領域でない領域にあるコードの実行を検知したならば、侵入コードを疑ってよい（一部の例外に対する対応が別途必要である）。これには、ハードウエアによるメモリの命令実行保護機能を利用することができる。また、OS によるシステムコールアドレスの監視によって代用することもできる。

表 2 バッファオーバフロー検知方式

<table>
<tr><th rowspan="2">攻撃</th><th rowspan="2">被害・症状</th><th colspan="4">侵入検知方式とその有効性</th></tr>
<tr><th>データ破壊監視</th><th>振舞い監視</th><th colspan="2">バッファ実行監視</th></tr>
<tr><th></th><th>AP ベース</th><th colspan="2">OS ベース（システムコール監視）</th><th>H/W ベース</th></tr>
<tr><td>単純メモリ破壊</td><td>アドレス例外
サーバダウン</td><td>△：ソースコード要
or バイナリパッチ</td><td colspan="2" rowspan="2">×：システムコール実行前は対象外</td><td>×：命令実行前は対象外</td></tr>
<tr><td>侵入コード実行</td><td>メモリ破壊・改竄
誤動作</td><td rowspan="5">データ破壊を検出できれば動作に無関係に有効</td><td rowspan="3">◎：侵入コード実行無し
高性能</td></tr>
<tr><td>システムコール実行</td><td>状態変更・侵入</td><td rowspan="4">△：誤検知、見逃し
性能に課題</td><td>○：ソースコード不要</td></tr>
<tr><td>ライブラリコール</td><td>同</td><td>△：スタックチェック難</td></tr>
<tr><td>return-to-libc</td><td>同（自由度：小）</td><td colspan="2" rowspan="3">×：侵入コードの実行がないので無効</td></tr>
<tr><td>連鎖型
return-to-libc</td><td>同（自由度：大）</td></tr>
<tr><td>ポインタ更新</td><td>同（自由度：中）</td></tr>
</table>

さらに、サーバの動作、振舞いを監視し、その異常を検知する方法がある。たとえば、ＯＳから見た場合、プロセスが要求するシステムコールの列をそのプロセスの「振舞い」と考えることができる。監視対象のサーバが実行しないはずのシステムコール列を実行時に検知する手法がある[12)]。この手法の課題は、検知の精度と実行時の性能である。誤検知（false positive）と見逃し（false negative）の低減、および実行時の性能向上のための研究がある[13)]。

3.3 検知方式の有効性

表 2 に、検知方法と攻撃手法の関係を示した。

データの破壊の監視コードはコンパイラなどによって、アプリケーションプログラムの中におかれることが多い。保護対象のサーバのソースコードが必要であるような方式は適用が制限される。そこで、バイナリパッチなどの形で提供されることもある。

バッファ上での侵入コードの実行の監視機能は、侵入コードが動き出して初めて働き出す。特にオーバフローによるデータ改ざんそのものは検知できない。

ハードウエアで侵入コードの実行を検知する方式のメリットは侵入コードを一切実行しないことと、検出処理のオーバヘッドがないことである。それでも、バッファオーバフローを起こした関数が処理を終え、リターンするまでのタイムラグがある。この間、データの改竄によってサーバプログラムが操られる可能性がある。

システムコールの発行アドレスをチェックする方式では、侵入コードが実行され、それが最初にシステムコールを発行するまでチェックすることができない。侵入コードが既にある程度動いた後であり、検知までのタイムラグが大きい。システムコール、およびライブラリコールには対応することができる。メリットはサーバのソースコードはなくても監視が可能であることである。直接システムコールを発行さずに、ライブラリをコールするような場合は、ユーザスタックを調べなければならないが、完全な解析は難しい。

以上では、return-to-libc 攻撃など、侵入コードの実行がない攻撃には対応できない。このような高度の攻撃に対しては、ここに挙げた検知手法の中では、サーバの振舞いの異常を検知するしか方法がない。

4.「 空 蝉 」

本章では、検知＆回復アプローチに基づく、バッファオーバフロー攻撃に対するＯＳによる対策である「空蝉」について述べる。

4.1 コンセプト

「空蝉」はバッファオーバフロー攻撃を検知したとき、攻撃されたプロセスを無条件に強制終了させて、サービスの喪失に甘んじるのではなく、プロセスを攻撃前の状態に安全に自動回復して、サービスの継続を図る。

攻撃されたプロセスは、オーバフローしたデータによって破壊、ないし汚染されているから、攻撃前の状態に復旧しなければならない。そのために、「空蝉」はその名の通り、攻撃前の状態のプロセスへの変わり身を試みる。その結果、侵入を阻むと同時に、サービスを継続させる。

既存のサーバプログラムを「空蝉」の対象とすることができるよう、サーバのソースプログラムを改変することなく適用可能としたい。そのため、「空蝉」は攻撃前のプロセスの状態を自動的に退避する。そして、バッファオーバフロー攻撃を検知するとその状態に自動的に回復する。サーバプログラムはそれとは気づかずに回復点からの処理を繰り返す。

ただし、サーバプログラムによっては、プロセスを回復するだけでは矛盾なくサービスを継続できない場合があり、サーバ特有の回復処理が必要となる。そこで、サーバプログラムから直接、「空蝉」の機能を利用し、制御することも可能としたい。

サーバ計算機上では、「空蝉」の保護の対象にしなくてもよいプログラム、対象とするのは不適当なプログラムが同時に稼動している。たとえば、自動的な保護の対象にできるプログラムもあれば、そうでないものもある。「空蝉」の検知処理によっては、それがうまく働かないようなものがあるかもしれない。そこで、「空蝉」は選択されたプロセスのみを監視・保護の対象とする。

4.2 サーバモデル

「空蝉」の第一の適用対象は、UDP を利用したサーバとする。それは以下の考察に基づいている。

バッファオーバフロー攻撃を受けるプログラムは、メッセージを受けてサービスを行うサーバプログラムである。サーバは socket による TCP/IP 通信を行うものとする。サーバプログラムはサービスの要求メッセージを待ちうけ、要求があればそれを処理して、再び待ち受け状態に戻る、というサービスループを持っている。要求を受け付けるプロトコルが TCP か UDP かで、サーバの構造が異なる。

TCP はコネクション指向のプロトコルであり、ステートフルである。TCP を利用した TCP サーバの基本的な構造は、メインプロセスがセッション開始要求を受け付けると、以後の処理は子プロセスにまかせる、というものである。TCP サーバの場合、メッセージは子プロセスが処理するため、バッファオーバフローは子プロセスにおいて発生することが多いと考えられる。この場合、侵入を検知して子プロセスを回復しても可用性上のメリットはない。攻撃された子プロセスを単に棄てるのが正しい処理である。

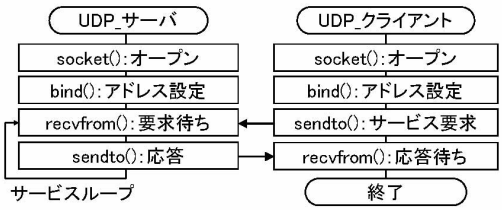

図 2 UDP サーバモデル

UDP はコネクションレスのプロトコルであり、ステートレスである。UDP を利用した UDP サーバの基本的な構造は一問一答型のサービスをサービスループの中で実行するというものである（図 2）。UDP サーバの場合、バッファオーバフローはサーバプロセス内で発生するため、バッファオーバフロー攻撃の検知を契機にプロセスを終了させると、サービスが停止する。従ってこのようなサーバの回復は、サービスの可用性の観点から有効性が認められる。また、要求待ちの時点に戻れば、新たな要求を待つのみであるから、回復による副作用は少ないと考えられる。

4.3 設　　計

Linux 2.6.X をベースに行った「空蝉」の実装について述べる。「空蝉」は、侵入を検知したならば、プロセスあらかじめ退避しておいた状態に回復して、処理を続行させる。このとき、以下のような課題がある：(1) どの時点に回復するか、(2) どのようにプロセスの状態を退避するか、(3) 攻撃の検知をどのように行うか。

4.3.1 チェックポイントのタイミング

ある時点のプロセスの状態を退避しておき、後に、退避した時点の状態から処理を継続する手法はチェックポインティングとして知られている。ここでも回復点のことをチェックポイントと呼ぶ。

受信したメッセージによってバッファオーバフローが起こるとすると、チェックポイントはオーバフローを起こしたメッセージの受信より前でなければならない。従ってメッセージ受信の直前がチェックポイントの第一候補である。

UDP サーバの場合、それは、recvfrom() システムコール発行の時点である。recvfrom() システムコールでは、外部からのサービス要求メッセージを待ち受ける。その時点でメッセージが到来していれば、それを受け取る。到来していなければ、プロセスはシステムコールの中でブロックする。メッセージが到来すると、ブロックが解け、そのメッセージを受け取り、システムコールから返って、メッセージの処理を開始することができる。回復処理では、プロセスが要求メッセージを待ち受けている状態に戻す。すると、サーバプログラムは、メッセージの受け取りから処理を再開し、新たな要求をはじめから処理することになる。

4.3.2 プロセス状態の退避方式

プロセスの状態の退避は、recvfrom() 処理を実行する前に fork() システムコールと同等の処理を行うことによって実現する。実装も、fork() の実行関数を呼び出して実現している。これにより、状態退避に伴うオーバヘッドは、Linux が備えている copy-on-write 機能によって自然に最適化される。退避された「プロセスの状態」は実質的に子プロセスと変わらないが、ディスパッチせずに待機させておく。

回復処理では、攻撃されたプロセスを廃棄するとともに、待機させておいたプロセスをディスパッチ可能にし、そのプロセスの状態を退避するところから処理を再開する。

4.3.3 バッファオーバフローの検知方式

バッファオーバフロー検知方式は、「空蝉」の実効性にとって最重要の技術項目である。ただし、検知方式の実装は「空蝉」の全体構成とは独立であり、有効な検知手段があれば、それを取り入れることができる。

現在の実装では、侵入コードによるライブラリコールを、ＯＳによって検知することを前提に、「書き込み可能な領域からシステムコールが呼び出されたならば、侵入発生と判断する」という方式を採用している。書き込みが可能か否かは、領域の属性情報から判る。この方式は、バッファがスタック領域にあるか、ヒープ領域にあるかに依存しない。システムコールの実行を伴わない攻撃には対応できないが、単純な構成の侵入コードによる攻撃は検出することはできる。

ＯＳはシステムコールアドレス、およびライブラリコールアドレスを知るために、ユーザスタックを解析する必要がある。IA32 上の Linux では、システムコールのために、割り込み命令「int 0x80」（以下 int80 と略記）と、

int80 命令のハードウエアオーバヘッドの削減を目的に導入された高速システムコール命令「sysenter」の２つが使用されている。両者で、システムコールアドレスの探索方法、およびユーザスタックの構造が異なる。

int80 命令では、システムコールアドレス（厳密にはその次の命令のアドレス）はカーネルスタック上に退避されて、ＯＳに知らされる。

一方、sysenter 命令では、ハードウエアは実行アドレスを退避しない。そのため、Linux では、ライブラリとカーネルとの間のインタフェースとして sysenter スタブを用意し、スタブが sysenter 命令を実行する。従って、syenter 実行アドレスは固定値になるため、ＯＳはアプリケーションに返ることができるが、攻撃を検知するには、スタブを呼び出したアドレスを知る必要がある。また、侵入コードがライブラリを呼び出す場合、システムコールはライブラリから出るので、攻撃を検知するためには、ライブラリを呼び出したアドレスを知る必要がある。いずれの場合もユーザスタックを調べなければならない。

4.3.4 適 用 方 式

「空蝉」では、監視の対象とするサーバプロセスをコマンドによって指定する。このとき、検知モードと回復モードをそれぞれ指定することができる。図 3 に、「空蝉」の制御の概要を示す。

回復モードとして、recvfrom() を実行する毎にチェックポイントを設定しなおす CP_TEMPORARY と、最初に設定したチェックポイントを更新せずに保持する CP_PERMANENT を用意する。前者は、チェックポイントをなるべく近い過去にすることを目的としている。後者では、チェックポイント生成のオーバヘッドを最小にすることができる。

検知モードとして、チェックポイントが存在するときだけ、検知処理を行う ID_SOMETIMES と、チェックポイントの有無に関係なく、検知処理をシステムコールに対して毎回実施し、チェックポイントが存在すれば回復を図り、存在しなければプロセスを強制終了する ID_ALWAYS がある。前者は、回復できることを前提に検知を行う、という考え方に基づいている。後者は、回復機能なしで、検知のみを行う場合にも適用できる。

上記で、recvfrom() の時にチェックポイントをＯＳが自動的に設定するのは、アプリケーションを改変することなく適用可能とするためであるが、これに加えて、アプリケーションが積極的に検知&回復機能を利用するためのインタフェースを提供する。サーバプログラムが自分自身に対してチェックポイントを直接設定し、監視と回復のモードを指定する CP_CREATE と、回復処理を起動する CP_RECOVER を用意する。これらにより、サーバプログラムが自ら検知処理を実施し、プロセスの回復を指示することができる。回復後は CP_CREATE の出口にリターンする。このとき、サーバプログラムはリターンコードによって、チェックポイントの設定をしたところなのか、攻撃を受けて回復処理が起動されたのか、あるいは自ら回復を指示したのかなどが判別できる。回復後の場合は、サーバプログラムは固有の回復処理を実行することができる。

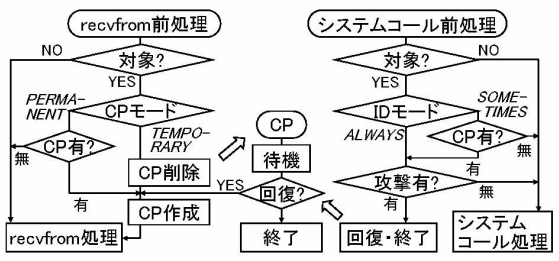

図 3 空蝉の制御方式

5. 適 用 実 験

「空蝉」を DNS サーバ（BIND）に適用し、実在の脆弱性（VU#325431、VU#196945[1)]）を使って、適用実験を行った。DNS は sdl.hitachi.co.jp のようなドメイン名で問い合わせると、ネットワークアドレスを応答する、典型的な問い合わせ応答型のサーバである。適用実験では、CP_TEMPORARY での適用では回復に失敗することがあったが、CP_PERMANENT で安定して監視、回復を実行することができた。

DNS のような広く利用されているサーバに適用可能であることは、「空蝉」の有効性の一端を示すものと考えている。

6. 性 能 評 価

「空蝉」の ID_ALWAYS、および CP_TEMPORARY ではシステムコールの度に検知処理やチェックポイント作成処理を行うため、性能への影響が懸念される。そこでこれらについて性能評価を行った。測定に使用した計算機環境は、計算機：HITACHI Flora 370 TS7（プロセッサ：Pentium4 2.4GHz, メモリ：1GB）、ＯＳ：Linux 2.6.11.11 の「空蝉」拡張、環境：Fedora Core 3 である。

6.1 検知処理の性能評価

6.1.1 システムコール呼び出し性能の測定

呼び出された回数を返す以外、何もしないシステムコールを実装し、これを 10,000 回呼び出すのにかかる時間を gettimeofday() によって測定し、これを 10,000 回繰り返して、実行時間の平均、標準偏差、最小、最大を測定した。

システムコールのアプリケーション側のインタフェースとして syscall() 関数を使用した。また比較のため、sysenter スタブの呼び出し、および int 0x80 命令による呼び出しも測定した。syscall() 関数はライブラリ関数であり、sysenter スタブを呼び出す。一般のシステムコールの呼び出しオーバヘッドとしては、syscall() 関数に準ず

表 3 システムコール呼び出し性能

Linux	2.6.11.11	2.6.11.11+空蝉			
測定条件	syscall()	syscall()	sysenter	int80	syscall() +検知
平均	1784	1878	1767	4689	2540
標準偏差	63	57	53	95	85
最小	1743	1835	1741	4651	2458
最大	5025	5094	3565	7809	5677

単位：ナノ秒

表 4 ApacheBench の測定

Linux	2.6.11.11		2.6.11.11+空蝉			
検知処理	—		なし		あり	
クライアント	1	10	1	10	1	10
平均	720	763	713	776	728	797
標準偏差	3.2	14.1	2.5	2.9	2.2	2.4
最小	710	737	707	765	724	791
最大	727	801	720	787	734	803

単位：ミリ秒

るオーバヘッドを想定してよいだろう。侵入検知処理は、syscall() からのスタブコールアドレスと syscall() 呼び出しアドレスをチェックして完了する。

表 3 に測定結果を示す。

int80 を sysenter 化すると、1システムコールについて、約 290 ナノ秒 (4689 − 1767) のゲインがあり、システムコール呼び出し・リターン部分について約 62% (1 − (1767/4689)) の性能向上となる。

syscall() ライブラリによるオーバヘッドは 1 システムコールについて、約 11 ナノ秒 (1878 − 1767) であり、それは syscall() の場合の所要時間の、約 5.9% ((1878 − 1767)/1878) に相当する。

「空蝉」の実装により、監視対象でないプロセスにおけるオーバヘッドは 1 システムコールについて、約 9 ナノ秒 (1878 − 1784) であり、syscall() の呼び出しリターンの、約 5.3% ((1878 − 1767)/1784) の性能劣化となる。

侵入検知処理のオーバヘッドは 1 システムコールについて、約 66 ナノ秒 (2540 − 1878) であり、「空蝉」の侵入処理オーバヘッドは、約 76 ナノ秒 (2540 − 1784) である。性能劣化率としては、それぞれ約、35% (662/1878)、42% (756/1784) である。

これらは int 0x80 によるシステムコールオーバヘッド 290 ナノ秒よりははるかに小さく、1システムコールについて、なお約 210 ナノ秒 (4689 − 2540) の余裕があり、システムコール呼び出しリターン時間の、約 46% (1 − (2540/4689)) の性能向上　を確保している。

6.1.2 ApacheBench による測定

一般のシステムコールはそれ自体の処理時間が加わるため、性能劣化の割合としては上記より小さくなる。実システムのシステムコールミックスとして「ab（apache benchmark)」による測定を試みた。

測定対象は、Http サーバ：Apache 2.0.54。デフォールトオプションで make したもの。keepAlive をサポートしている。prefork 機能によって 5〜10 個のサーバプロセスが起動されている。

ベンチマークは、ApacheBench 2.0.41 を上記とともに make したものを用いた。余分のメッセージを抑止し、keepAlive でアクセスするようにした。このとき、試行1回あたり、通信と I/O を中心とした 61 個のシステムコールを発行する。クライアント数は 1 および 10 とした。1000 リクエストの実行時間を 100 回測定し、「Time taken for tests」 の平均を求めた。

表 4 に ApacheBench における 1000 リクエストの所要時間の測定結果を示す。クライアント数を増やすとオーバヘッドが増大する。以下では、クライアント数 1 の場合を比較する。

監視対象でないプロセスにおける「空蝉」の実装によるオーバヘッドは、測定によると予想に反し、逆転している。1 リクエストについて、約 7 マイクロ秒減少 ((713 − 720)/1000) しており、「空蝉」のない場合の、約 1.0% (7/713) に相当する。

検知処理のオーバヘッドは 1 リクエストについて、約 15 マイクロ秒 ((728 − 713)/1000) であり、劣化率は、約 2.1% (15/713) である。

6.2 チェックポイント設定処理の性能評価

「空蝉」はチェックポイントの設定に fork() 処理を利用している。fork() はオーバヘッドの大きいシステムコールとしてよく知られている。そこで、fork()、vfork() と比較しつつ、チェックポイントの設定処理＋解除処理の性能評価を試みる。

「空蝉」のチェックポイント設定は、通常、recvfrom() システムコールのときに行うが、繰り返し測定が難しい。ここではダミーシステムコールを設け、チェックポイントの設定解除の対象とし、これを呼び出して測定することにする。この場合、カーネル内では do_fork() 処理の後、スケジュール処理や、プロセスの終了処理が行われる。そこで fork()、あるいは vfork()、および、exit()、wait() を用いてこの処理をシミュレートし、処理性能を比較する。すなわち、子プロセスは起動後直ちに exit() し、親プロセスは wait() して SIGCHLD を受け取ってから、次の fork へ進む。

表 5 に測定結果を示す。処理が同じではないので直接の比較は無意味だが、オーバヘッドのオーダを把握することができる。統計処理プログラムを同時に稼動させて

表 5 チェックポイント設定・解除処理の測定

OS	Linux 2.6.11.11+空蝉			
測定対象	Nop	vfork()	fork()	空蝉 CP
平均	21	990	12616	9030
標準偏差	5	44	363	267
最小	20	900	12153	8502
最大	361	2593	19385	13665

単位：μ 秒

いるので、測定データのばらつきが大きくなっている。

「空蝉」のチェックポイント設定・削除処理のオーバヘッドは fork() のオーバヘッドの約 72% (9030/12612) である。この差は主として wait() システムコールの有無によるものと考えられる。

7. 関連技術

プロセスクリーニング[14] でも、プロセスの状態を保存し、後に保存した時点の状態に回復する。その目的は、プロセスのセキュリティレベルを安全に緩和することであり、状態保存、および回復の契機はサーバプログラムが発行する専用のシステムコールである。想定している脅威はサーバへの侵入であり、侵入コードによる制限の緩和の防止に効果があることから、「限定」アプローチに属する技術と考えられる。

「空蝉」では、回復の契機は攻撃の検知であり、回復の目的はサーバの可用性の維持である。従って、「空蝉」では攻撃検知技術の比重が大きく、また、既存のサーバにも改変なしに適用することを目指している。これらの点で、プロセスクリーニングとは意図するものが異なる。しかし、copy-on-write をプロセスの状態の退避の最適化に援用し、また、論文にはプロセスが制御を奪われたことを検出した場合に、プロセスクリーニングを強制的に実行する可能性も述べられており、思想的に共通する部分も多い。

DYBOC[15] の問題意識はセキュリティと可用性にあり、「空蝉」と共通する部分がある。アプローチは言語ベースであり、ソースコードを解析・変換して、アプリケーションによって、チェックポイントを登録し、バッファオーバフローを検知し、スタックの回復を実現している。

8. まとめ

攻撃の検知を、直接、サーバの安全な回復に結びつける「検知&回復」アプローチを提案し、このアプローチに基づく、ＯＳによるバッファオーバフロー攻撃対策である「空蝉」について述べた。「空蝉」の DNS サーバに対する適用実験を行い、プログラムを改変することなく保護対象とするとともに、バッファオーバフロー攻撃を受けても、安全に機能回復することを確認した。通常稼動時の攻撃検知処理のオーバヘッドをシステムコール単独、および ApacheBench によって評価し、許容範囲内であることを確認した。「空蝉」を適用すると、バッファオーバフロー攻撃を受けても、侵入を阻止できると同時に、サーバの再起動の手間が不要であり、システム運用の容易化にも効果がある。

今後の課題としては、粒度の細かい実行保護ハードウエアや、振舞い監視に基づく、より精度の高い攻撃検知手法の利用が挙げられる。プロセスの退避・回復については、処理の最適化の他、より広いクラスのサーバに適用可能としていくことが挙げられる。また、セキュリティ管理、システム管理の観点からは、再発防止を含む、攻撃検出後の「対処」も課題として挙げられる。さらに、サーバプログラムからの積極的な利用を支援するために、開発環境との連携も挙げられる。

参考文献

1) CERT/CC: Vulnerability Notes Database. http://www.kb.cert.org/vuls.
2) Cowan, C. et al.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *Proceedings of DARPA Information Survivability Conference and Exposition* (1999).
3) 櫻庭健年, 櫻井幸一: 攻撃の時系列的諸局面に対応したセキュリティ対策の探索, 情報処理学会研究報告, 2005-DPS-126/2005-CSEC-32 (2006). to appear.
4) 河野健二, 品川高廣, ラハト・カビル: TCP ストリームに対するフィルタリングによるインターネット・サーバの安全性向上, 情報処理学会論文誌:コンピューティングシステム, Vol. 46, No. SIG 4(ACS 9), pp. 33–44 (2005).
5) Xu, J. et al.: Architecture Support for Defending Against Buffer Overflow Attacks, *Proceedings of the 2002 Workshop on Evaluating and Architecting System dependabilitY* (2002).
6) 安藤類央, 武藤佳恭: 分岐命令処理フィルタを用いた不正プロセス実時間防御機構の構築, 情報処理学会論文誌, Vol. 46, No. 8, pp. 1935–1946 (2005).
7) 江藤博明, 依田邦和: propolice: スタックスマッシング攻撃検出手法の改良, 情報処理学会論文誌, Vol. 43, No. 12, pp. 4034–4041 (2002).
8) Loscocco, P.A., Smalley, S.D. et al.: The inevitability of failure: The flawed assumption of security in modern computing environments, *21st National Information Systems Security Conference* (1998).
9) 品川高廣: セグメント機構を用いたバッファオーバフロー攻撃防止手法, 情報処理学会研究報告, 2005-OS-99, Vol.2005, No. 48, pp. 113–120 (2005).
10) Intel Corporation: Execute Disable Bit Functionality. http://www.intel.com/business/bss/infrastructure/security/xdbit.htm.
11) Cowan, C. et al.: StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks, *Proceedings of 7th USENIX Security Symposium*, pp. 63–78 (1998).
12) Forrest, S. et al.: A sense of self for UNIX processes, *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 120–128 (1996).
13) 阿部洋丈, 大山恵弘, 岡瑞起, 加藤和彦: 静的解析に基づく侵入検知システムの最適化, 情報処理学会論文誌:コンピューティングシステム, Vol. 45, No. SIG 3(ACS 5), pp. 11–20 (2004).
14) 光来健一, 千葉滋: サーバのアクセス制御を安全に変更するための機構, 情報処理学会論文誌, Vol. 42, No. 6, pp. 1492–1502 (2001).
15) Sidiroglou, S. et al.: A dynamic mechanism for recovering from buffer overflow attacks, *Proceedings of the 8th International Security Conference*, pp. 1–15 (2005).