論文の内容の要旨

論文題目　**A Study of a Dynamic Safety Net for Server Software**
（サーバソフトウェアのための動的なセーフティネットの研究）

氏　　名　光來健一

ソフトウェア技術の飛躍的な進歩により、ソフトウェアはますます肥大化・複雑化してきている。この流れの中で、ソフトウェアは安全面で2つの大きな問題を抱えている。一つはソフトウェアに対する攻撃である。インターネットを利用するソフトウェアは常にバッファオーバフロー攻撃などの攻撃を受ける危険にさらされている。一旦攻撃に成功すると、攻撃者は犠牲となったソフトウェアの持つ権限を悪用することができる。このような攻撃を防ぐために様々な手法が提案されているが、多くの手法はまだ知られていない新しい攻撃を防ぐことができない。

ソフトウェアが抱えるもう一つの問題は、製品のベータ版や一部のフリーソフトのように不安定なソフトウェアが存在することである。不安定なソフトウェアはその内部にデータのチェックの不備などのエラーの引き金を持っており、いくつかの条件が満たされるとその引き金が引かれる。ソフトウェアが内部エラーの引き金を引いてしまうと、そのソフトウェアの権限の範囲内で何が起こるか分からない。プログラミング言語やコンパイラのサポートによりこのようなエラーの発生を減らすことはできるが、全てのエラーを取り除くのは難しい。

攻撃と不安定化の主な原因はソフトウェアの不具合である。攻撃は普通のユーザが使っていると表面化しない隠れた不具合を利用する。逆に、不安定化は普通のユーザが使っていても特定の条件で表面化する。ソフトウェアの不具合には実装上の問題と設計上のミスがある。実装上の問題は比較的検出しやすいので様々な手法が提案されているが、それでもなお数多くのソフトウェアに未知の不具合が残されている。また、プログラマはよく設計上のミスを犯すが、この種の不具合を自動的に検出するのは難しい。

そのため、万一ソフトウェアが不具合のせいで異常な動作をした場合でも、被害を最小に抑えることができる機構が必要とされる。この機構を**セーフティネット**と呼ぶ。セーフ

ティネットはこのような場合に被害が外に拡がるのを防止する。たとえソフトウェアが攻撃を受けたとしても、攻撃者はセーフティネットの範囲を越えて保護されている資源にアクセスすることはできない。同様に、不安定なソフトウェアがセーフティネットの外の他のソフトウェアを不安定にすることもない。現在、既に様々なセーフティネットが提案され、実際に利用されている。

動機

本論文は特にサーバソフトウェアのためのセーフティネットに焦点を当てる。サーバソフトウェアのためのセーフティネットを構築するにあたって2つの障害がある。(1) サーバソフトウェアは常に動いていて、同時に様々なユーザからの要求を処理しなければならないので、それぞれのユーザ毎に異なるセーフティネットが必要となる。(2) サーバソフトウェアは安全性と同時に十分な性能も実現しなければならないが、その両方を満たすのは難しい。これらの2つの障害を克服するには、サーバソフトウェアが動的にそのセーフティネットの範囲を変更できる**動的なセーフティネット**が不可欠である。

ウェブサーバのようなユーザレベル・サーバはクライアントに応じてセーフティネットの範囲を変更するべきである。ユーザレベル・サーバにとっては悪意を持ったクライアントからの攻撃が最大の脅威だからである。しかしながら、サーバにこのような機能を提供すると攻撃者も悪用することができるので、セキュリティ上危険である。万一サーバが乗っ取られた場合、攻撃者はセーフティネットの範囲が最大になるように変更することにより、セーフティネットを無力化することができる。また、トロイの木馬がサーバに送り込まれ、サーバがセーフティネットの範囲を変更した後でそのコードが実行された場合、送り込まれたトロイの木馬は適切でないセーフティネットの中で実行される可能性がある。システムが正常なサーバだけにセーフティネットの変更を許すようにすればこの危険性を回避できるが、サーバが攻撃されているかどうかを判断するのは容易ではない。

一方、サーバ専用のファイルシステムのようなオペレーティング・システム（OS）のモジュールは、そのセーフティネットの範囲を安定性に応じて変更するのが望ましい。OSモジュールが直接攻撃を受けることは少ないので、誤動作への対処の方が重要になる。そのためには、OSモジュールの安定性に応じて性能と安全性の適切なトレードオフを取れるようにする必要があるが、従来のシステムのほとんどはそのようには設計されていない。適切なトレードオフが取れるシステムを設計するにあたって、トレードオフの変更はユーザに意識させずに行えることが要求される。加えて、トレードオフを変更する機構がモジュールの性能を低下させないことも重要である。モジュールを範囲の最も広いセーフティネットの中で動かせば、セーフティネットを使わないモジュールと性能がほとんど等しくなるべきである。

アプローチ

本論文ではこのような動的なセーフティネットを提供するシステムについて述べる。このシステムはユーザレベル・サーバのためのアクセス制御機構とOSモジュールのための保護機構からなる。開発したアクセス制御機構はクライアントとクライアントからの要求

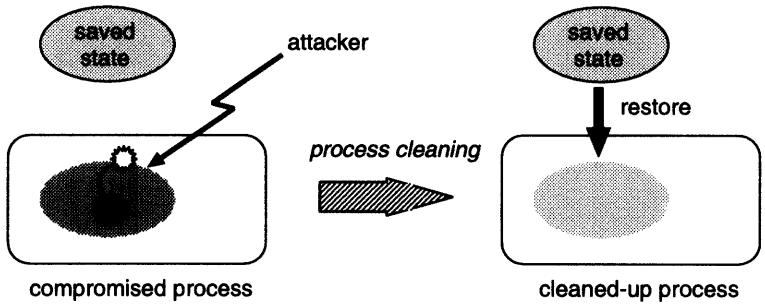

図 1: プロセスクリーニングによるプロセスの健全化

の内容に応じたアクセス制限によって、動的なセーフティネットを実現する。この機構を使って、サーバプロセスはクライアントからの要求を処理している間、そのクライアントに合ったアクセス制限を自分自身にかけることができる。

このアクセス制御機構の特徴はサーバプロセスが動的にアクセス制限を変更することを許している点である。サーバプロセスがアクセス制限を弱めるときに伴う危険を避けるため、この機構はアクセス制限を弱める前にプロセスをクリーンにし、送り込まれているかもしれない悪意あるコードを除去する（図 1)。これを**プロセスクリーニング**と呼ぶ。プロセスクリーニングを行うために、OS はサーバが攻撃される前のサーバプロセスの状態を保存しておく。そしてアクセス制限を解除する時には保存しておいた状態を復元し、攻撃者によってなされている可能性がある変更を元に戻す。復元される状態はプロセスの制御やメモリイメージなどである。

プロセスクリーニングは攻撃者が許可されていない権限を手に入れるのを防ぐ。乗っ取られたサーバが高い権限を手に入れるためにアクセス制限を解除しようとしても、プロセスクリーニングによりサーバはプロセスの制御を取り戻すことができる。攻撃者はプロセスの制御を保ったまま、アクセス制限を解除することはできない。また、アクセス制限の解除後に実行する目的でトロイの木馬としてのコードがサーバプロセスに送り込まれていても、アクセス制限を解除する時にプロセスクリーニングによってメモリから除去される。

一方、開発した保護機構はデフォルトで OS モジュールを最も狭いセーフティネットの中で動かし、モジュールと OS カーネルの間のインタフェースを厳しく制限する。それぞれの OS モジュールは OS カーネル、他の OS モジュールやユーザレベル・プロセスから切り離され、OS モジュールの異常な動作がシステムの他の部分に影響を及ぼすことはない。例えば、OS モジュールはユーザレベル・プロセスとして動かされ、その OS モジュールに必須のカーネルの機能だけを使うことが許される。

動的なセーフティネットを実現するために、開発した保護機構はユーザが OS モジュールの安定度に応じて保護レベルを変更することを可能にしている（図 2)。これを**マルチレベル・プロテクション**と呼ぶ。マルチレベル・プロテクションは OS モジュールに様々な保護レベルを提供しており、それぞれの保護レベルは検出・回復できるエラーの種類が異なる。マルチレベル・プロテクションは様々な保護レベルを実現するために、アドレス空間の切替やカーネルデータの複製などの様々な保護技術を用いる。例えば、OS モジュールを異なるアドレス空間に配置することにより、メモリアクセス違反を検出することがで

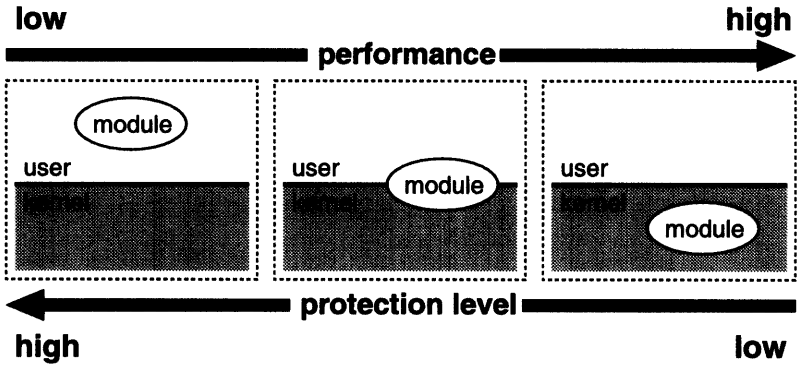

図 2: マルチレベル・プロテクションの取るトレードオフ

きる。

保護レベルは OS モジュール毎に提供されている保護マネージャを交換することによって変更することができる。保護マネージャはカーネルと OS モジュールの間の中継器の役割を果し、それらの間の相互の呼び出しを中継する。その時に様々な保護技術を用いて、カーネルを OS モジュールの誤動作から保護する。また、様々な保護レベルを実現している全ての保護マネージャが、OS モジュールに対して共通のアプリケーション・プログラミング・インタフェース（API）を提供している。そのため、保護レベルを変更する際にモジュールへの変更は全く必要ない。

実験

我々は開発したアクセス制御機構を Linux 上に実装し、Apache ウェブサーバにおけるプロセスクリーニングのオーバヘッドは最大で 35%であることを確かめた。また、我々のアプローチと同様にアクセス制限を安全に解除することができる従来の fork-join 法と比べると、平均で 45%高速化することができた。また、開発した保護機構を NetBSD 上に実装し、我々が作成したファイルシステムとネットワーク・サブシステムの OS モジュールを使って実験を行った。その結果、最大の保護レベルで動かすと現実的なアプリケーションで最大 75%程度のオーバヘッドがかかることが分かった。一方で、最小の保護レベルで動かした場合は、マイクロベンチマークでも最大で 12%程度しかオーバヘッドがかからないことが分かった。