動的なアクセス権限変更のためのアクセス制限の安全な解除機構

光来 健一† 千葉 滋‡

† 東京大学大学院 理学系研究科 情報科学専攻
‡ 筑波大学 電子・情報工学系, 科学技術振興事業団 さきがけ研究 21

要 旨

ウェブサーバに代表されるサーバの安全性を確保する技術の一つに、サーバのアクセス権限を動的に変更する技術がある。しかし、安全のためにアクセス権限を強い状態から弱い状態に変更することしかできず、プロセスプールの手法を用いたウェブサーバなどで利用するのが難しかった。そこで我々はプロセスを安全な状態に戻すことによって、アクセス制限を安全に解除する機構を提案する。プロセスを安全な状態に戻す作業をプロセス・クリーニングと呼び、レジスタやメモリの内容、シグナルやファイル・ソケットの状態などをあらかじめ保存しておいた状態に戻す。apache ウェブサーバにおいてプロセス・クリーニングを行うことにより、サーバの平均応答時間が 1.2 倍〜1.7 倍になり、スループットが 16%〜40%低下することが分かった。しかし、リクエスト毎に子プロセスを作って処理させる場合と比べると、サーバ性能が約 2 倍向上した。

A Safe Mechanism for Dynamically Removing Access Control Rules

Kenichi Kourai† Shigeru Chiba‡

†Department of Information Science, Graduate School of Science, University of Tokyo
‡Institute of Information Science and Electronics, University of Tsukuba,
PREST, Japan Science Technology Corp.

Abstract

To guarantee safety of the server software like web servers, several techniques for dynamically changing the access rights of servers have been proposed. However, these techniques are difficult to use for web servers using the process pool technique because they only allow to weaken the access rights. We propose a safe mechanism for removing access control rules so that the server obtains stronger access rights. The safeness is guaranteed by a new technique called *process cleaning*, which restores the states of registers, memory, signal handlers, files, and sockets to the previously saved states. With the apache web server, we observed that the server response time drops down by a factor of 1.2 to 1.7 and the server throughput is degraded by 16% to 40% if the process cleaning technique is used. However, the apache server using the process cleaning technique is two times faster than the server creating a child process for every request.

1 はじめに

ウェブサーバに代表されるような、サーバプログラムの安全性を確保する技術の一つに、処理の内容に応じてサーバのアクセス権限を動的に変更する技術がある。この技術により、クラック攻撃によりサーバが誤動作しても、重要なデータが盗まれたり、踏み台として使われるのを防ぐことができる。このような技術を使わなければ、サーバを `root` 権限で動かしておき、サーバプログラム自身でアクセス制御を行ったり、より権限の弱いユーザ ID でサーバを動かす必要がある。しかし、これらの方法で構築されたサーバはセキュリティホールが生じやすい。

サーバが処理しているリクエストに応じて、サーバ自身のアクセス権限を強めたり弱めたりするようにすれば、サーバの機能を制限せずに安全性を確保することが可能である。UNIX では `setuid/setgid` システムコールを用いて、プロセスのアクセス権限

を動的に変更することができる。これにより、ウェブサーバが CGI プログラムを実行する時にはより弱い権限で動作させ、セキュリティホールを利用した攻撃を防ぐことができる。また、我々が開発している Compacto システムのアクセス制御機構は、プロセス毎に利用可能なシステムコールを制限して、きめ細かなアクセス制御を行うことができる。

このようにサーバのアクセス権限を動的に変更できるシステムでは、安全のためにアクセス権限を強い状態から弱い状態に変更することだけを許している場合が多い。これはアクセス制限の解除を安全に行うのは一般に難しいからである。UNIX では root に setuid されたプログラムは、root 以外のユーザが root 権限で実行させることができる。これはアクセス権限を弱い状態から強い状態に変更する安易な方法であり、セキュリティホールになりやすい。

しかし、アクセス制限の解除はプロセスプールの手法を用いて実装されているウェブサーバなどでは欠かせない機能である。このようなサーバでは、あらかじめいくつかのプロセスを作っておき、それらにサーバの処理を行わせるため、リクエスト毎に子プロセスを作ってアクセス制限を追加するという方法が使えない。そのため、サーバプロセスはリクエスト処理の開始直前にアクセス制限を自身に追加し、処理が終了したら追加したアクセス制限を安全に解除して、次のリクエストに備える必要がある。

そこで我々は、プロセスを安全な状態に戻すことによって、アクセス制限を安全に解除できる機構を提案する。安全な状態とは、プロセスにアクセス制限を追加する前の状態であり、バッファオーバフロー攻撃などによってプロセスに悪影響が及んでいない状態である。プロセスからこのような悪影響を取り除いた後であれば、アクセス制限を解除しても安全である。プロセスを安全な状態に戻す作業をプロセス・クリーニングと呼ぶ。

プロセス・クリーニングでは、レジスタやメモリの内容、シグナルやファイル・ソケットなどの状態を元に戻す。レジスタの内容を元に戻すことで、プログラムの制御も元の状態に戻される。メモリの内容を元に戻すのは特にオーバヘッドが大きいが、copy-on-write の技術を利用することにより、変更されたページだけのコピーで済むようにしている。さらに、同じ処理が繰り返し行われる場合には、変更されるページをあらかじめ予測してコピーしておくことにより、copy-on-write に伴うページフォールトを減らすことができる。

我々は Linux 2.2.16 カーネルをベースとして Compacto システムを構築しており、その上にアクセス制限を安全に解除する機構を実装した。そして、プロセスプールの手法を使っている apache ウェブサーバでプロセス・クリーニングのオーバヘッドを調べる実験を行った。その結果、サーバの平均応答時間が 1.2 倍〜1.7 倍になり、スループットが 16%〜40%低下することが分かった。これは比較的大きなオーバヘッドであるが、プロセス・クリーニングをせずに安全にアクセス制限を解除できるようにするには、リクエスト毎に子プロセスを作らなければならない。この場合と比べると、プロセス・クリーニングではサーバの性能が約 2 倍向上した。

以下、2 章では、サーバの安全性を確保するために使われるアクセス権限の動的な変更について述べる。3 章では、プロセスプールの手法を用いたサーバなどで必要になるアクセス制限の安全な解除機構について述べる。4 章では、プロセス・クリーニングのオーバヘッドを調べるために行った実験について述べる。5 章で関連研究に触れ、6 章で本論文をまとめて、今後の課題を述べる。

2 アクセス制御機構によるクラック攻撃対策

2.1 動的なアクセス権限の変更

ウェブサーバに代表されるような、サーバプログラムをクラック攻撃から守り、その安全性を確保するために、様々な技術が提案されてきている [4, 5, 2, 7]。その中の一つに、サーバのアクセス権限を動的に変更し、万一クラック攻撃によりサーバが誤動作しても、重要なデータが盗まれたり、他のホストを攻撃する踏み台に使われないようにする、という技術がある。この技術を用いると、サーバが処理しているリクエストに応じて、サーバ自身のアクセス権限を強めたり弱めたりすることができ、サーバの機能を制限せずに安全性を確保することが可能である。例えばウェブサーバの場合、普段は root 権限で動かしておき、CGI プログラムの実行などセキュリティホールを含んでいる可能性のある処理を行う時だけ、アクセス権限を弱めるようにすればよい。また、クラック攻撃される可能性がある外部ホストからのリクエストの処理は、サーバの権限をより低い状態にして行えばよい。

このような技術を使わずに、サーバを常に root 権限で動かすと、クラック攻撃に対して非常に脆弱になってしまう。このような場合、OS の提供するアクセス制限を使うことができず、アクセス制御はサーバプログラム自身が行わなければならないので、プログラムの誤りによってセキュリティホールが生じやすい。例えば、本来見えないはずのウェブページを誤って見せてしまうかもしれない。また、バッファオーバフロー攻撃などを受け、実行中のサーバプログラムを攻撃側が送りこんだ不正コードで置き換えられてしまうと、攻撃側が望む処理を root 権限で行われてしまう。

OS によるアクセス制御を利用して上記のような問題を回避するには、現在の典型的なウェブサーバのように、アクセス権限を弱めたユーザ ID(例えば nobody) の下でサーバを動かすようにすればよい。この方法の欠点は、サーバが必要とするファイルなどの資源を全て nobody のアクセス権限で参照可能にしなければならない点である。例えば、あるウェブサーバの上に、認証に成功したクライアントからしか見えないページを置きたいとしても、そのサーバが動いているホストにログイン可能で、nobody

と同等かそれ以上のアクセス権限をもっているユーザにはそのページのファイルを読まれてしまう。

サーバのアクセス権限を動的に変更する方法として、UNIX では `setuid/setgid` システムコールが提供されている。これらのシステムコールは、サーバプロセスのユーザ ID(UID)/グループ ID(GID) を動的に変更するものである。UID/GID が変更されると、そのプロセスのアクセス権限も新しい UID/GID に対応するものに変更される。ウェブサーバの例では、CGI プログラムを実行する子プロセスの UID を、 `nobody` など弱いアクセス権限しかもたないものに変更することによって、セキュリティホールを利用した攻撃による被害を抑えることができる。

また、現在我々が開発している Compacto システムのアクセス制御機構を用いると、プロセス毎に利用可能なシステムコールをきめ細かく制限することができる。例えば、同じ UID をもつプロセスであっても、一方のプロセスには `socket` システムコールなどを使ったネットワークの利用を禁止することができる。Compacto によるアクセス制限は、UID/GID と同様、特に指定しない限り、親プロセスから子プロセスへ継承される。さらに Compacto では、同一サイト内の信頼できるホスト間の通信の際、クライアントプロセスの UID/GID やアクセス制限の内容などの情報を、サーバプロセスへ暗黙の内に伝達する。サーバプロセスは、この伝達された情報を元に、よりきめ細かくアクセス制限をかけた状態で、クライアントのリクエストを処理することができる。

2.2 アクセス制限の解除

UNIX や我々の Compacto システムを始め、サーバのアクセス権限を動的に変える機構を提供しているシステムは多いが、それらは原則としてアクセス権限を強い状態から弱い状態に変更することだけを許している。UNIX の setuid 機構の場合、UID を `root` から `nobody` に変更することはできるが、その逆はできない。Compacto の場合、新たなアクセス制限をプロセスに追加することはできるが、追加されたアクセス制限を後から解除することはできない。

これは一般に、アクセス制限の解除を安全に行うことは容易ではないからである。安易に設計されたアクセス制限の解除機構は、クラック攻撃によって不正に利用され、それ自身がセキュリティホールになる危険性がある。UNIX には、プログラムファイル毎に setuid/setgid ビットという属性が用意されており、プログラムの所有者・グループの権限でプログラムを実行することができる。例えば、`root` に setuid されたプログラムを実行すると、そのプロセスの元の UID にかかわらず、`root` 権限がそのプロセスに与えられる。しかしこの機構は、誤った利用によって容易にセキュリティホールとなってしまう。

アクセス制限の解除に、そのアクセス制限を追加した時に得られる秘密鍵を用いる方法は、クラック攻撃によってプロセスのメモリが盗み見られる可能性がある場合には、必ずしも安全ではない。Compacto ではプロセス間でアクセス制限を共有できるように、アクセス制限に対してランダムな値からなる ID を割り当てている。この ID を知っているプロセスだけにアクセス制限の解除を許すことにすれば、ランダム ID が盗まれない限り、確率的に安全である。しかし、バッファオーバフロー攻撃によって不正コードが送り込まれた場合、ランダム ID が盗まれる可能性は決して低くない。プログラムのソースコードが公開されていれば、ランダム ID が保存されているメモリの位置を推測するのは比較的容易だからである。

しかしながら、アクセス制限の解除機構は、実用的なサーバでアクセス権限を動的に変更して安全性を確保するためには、欠かせない機能の一つである。例えば、処理性能を向上させるためにプロセスプールの手法を用いて実装されているウェブサーバは、あらかじめサーバプロセスをいくつか生成しておき、それらのプロセスにリクエスト処理を分担させる。このためリクエスト毎に子プロセスを作って、アクセス制限を追加するという方法が使えない。この場合には、サーバプロセスはリクエスト処理の開始直前にアクセス制限を自身に追加し、処理の終了後、追加したアクセス制限を安全に解除し、次のリクエストに備えられるようにする必要がある。リクエストを送ってくるクライアントに合わせて、アクセス制限を毎回変更しなければならないからである。

3 アクセス制限の安全な解除

3.1 アクセス制限の安全な解除機構

我々はプロセスを安全な状態に戻すことによって、プロセスのアクセス制限を安全に解除できる機構を提案する。安全な状態とは、アクセス制限を追加する前の、プロセスに悪影響が及んでいる可能性がない状態である。例えば、サーバプロセスの場合、リクエストを処理するとバッファオーバフロー攻撃によって不正コードが送り込まれる可能性があるが、そうなる前の状態である。プロセスを安全な状態に戻すことは、プロセスに対するこのような悪影響を取り除くということであり、その後でのみ、アクセス制限を解除しても安全である。解除することができるアクセス制限は、システムコールの利用の制限、`setuid/setgid` システムコールによるユーザ・グループの制限、`chroot` システムコールによるディレクトリの制限などである。

プロセスを安全な状態の戻すことによって、トロイの木馬のような巧妙な攻撃を無効にすることができる。例えば、バッファオーバフロー攻撃はバッファ領域の周辺のメモリを書き換えることで、後で悪影響を及ぼそうとする。関数ポインタや文字列ポインタを書き換えれば、実行される関数や exec されるプログラムを変更することができる。`PATH` 環境変数などを書き換えれば、exec されるプログラムを変更できるかもしれない。また、バッファオーバフ

ローで送り込まれた不正コードがシグナルハンドラの登録を変更すれば、後でシグナルが発生した時に不正な関数を実行させることができる。このような攻撃の影響はすぐには現れない場合が多く、検出するのは難しい。

3.2 プロセス・クリーニング

プロセスを安全な状態に戻す作業を、我々はプロセス・クリーニングと呼んでいる。Compacto ではプロセスを安全な状態の戻せるようにするために、2 つのシステムコールを提供している。一つはプロセスの状態を保存するシステムコールであり、もう一つはプロセスの状態を保存しておいた状態に戻し、状態保存後に追加されたアクセス制限を解除するシステムコールである。ここでいう状態とは、レジスタやメモリの内容、シグナルやファイル・ソケットの状態などである。この節では、それぞれの状態の保存と復元に関する実装について説明する。

3.2.1 レジスタ

プロセスの状態の保存を行うシステムコールでは、命令レジスタやスタックレジスタを含めた全てのレジスタの内容を保存する。状態の復元を行うシステムコールでは、保存しておいたレジスタの内容を元の内容に戻す。命令レジスタを元に戻すことにより、プログラムの制御を状態保存時の地点、つまり、状態を保存するシステムコールを呼んだ直後の地点に戻すことができる。もちろん、スタックレジスタを元に戻しても、状態の復元を行うシステムコールが、状態の保存を行うシステムコールを呼んだ関数またはそこから呼ばれた関数内で呼ばれなければ、そのスタックレジスタが指すスタックの内容は破壊されてしまっている。これは 3.2.2 で述べるメモリの状態の復元を行うことで解決される。

レジスタの内容を元に戻すことで、不正コードがアクセス制限の解除をした上で、さらにその実行を継続することはできなくなる。状態を保存した地点にプログラムの制御が必ず戻るからである。不正コードの実行を継続できるように不正コードの中で状態を保存したとしても、プログラムの制御とともにその時点でのアクセス制限も保存されるので、状態を復元しても不正コードは望みの権限を手に入れることはできない。

3.2.2 メモリ

状態を保存する時には、ユーザアドレス空間のメモリイメージと仮想メモリ管理情報を保存する。メモリイメージとは、そのアドレス空間に対する物理メモリのマッピング情報と、マッピングされている物理ページの内容である。このメモリイメージには、コード領域やデータ領域、ヒープ領域、スタック領域、環境変数の領域などが含まれる。一方、仮想メモリ管理情報とは、アドレス空間に対する仮想メモリのマッピング情報やメモリ保護の情報、ヒープ領域の上限を表すデータセグメント・サイズなどである。

状態を復元する時には、保存しておいたメモリイメージと仮想メモリ管理情報を元の状態に戻す。状態保存前に存在していたページに関しては、保存しておいたメモリの内容をコピーすることでその内容を元に戻す。もしそのページが存在しなくなっていれば、新しい物理ページを割り当ててから元の内容に戻す。状態保存前に存在していなかったページに関しては、そのページを破棄する。例外として、他のプロセスと共有しているページに関しては、内容を元に戻すと他のプロセスとの整合性が取れなくなる可能性があるのでそのままにしておく。一方、仮想メモリ管理情報に関しては、`mmap`/`munmap` システムコールによって変更された仮想メモリのマッピングを元の状態に戻す。また、`mprotect` システムコールによって変更されたメモリ保護も元の状態に戻す。さらに、`brk` システムコールによって変更されたデータセグメント・サイズも元に戻す。

メモリイメージを元に戻す際に、全てのページをコピーして元に戻す必要はない。コード領域など、書き込み可能でないページに関しては、内容が変更されることはないので、そのままにしておいてよい。一方、データ領域など、書き込み可能なページに関しては、内容が変更されている可能性があるので、元に戻すために保存しておいた内容をコピーする必要がある。しかしながら、書き込み可能なページの内、実際に内容が変更されるページは一部にすぎず、全ての書き込み可能なページの内容をコピーするのは無駄が多い。

そこで、copy-on-write の技術 [6] を用いて、実際に内容が変更されたページだけを元に戻せるようにする。状態を保存する時に、全ての書き込み可能なページを書き込み禁止にしておく。そして、そのページとそこにマッピングされている物理ページの対応を保存し、物理ページの参照回数を 1 増加させる。状態保存後にそれらのページに書き込みが行われると、書き込み禁止になっているのでページフォールトが発生する。ページフォールト・ハンドラの中で、システムは新しい物理ページを用意し、ページフォールトが発生したページの内容をコピーする。次に、ページフォールトが起きたページに対する物理ページのマッピングを、その新しい物理ページに変更し、元の物理ページの参照回数を 1 減少させる。この新しいページは書き込み可能になっており、以後の書き込みではページフォールトは発生しない。状態を復元する時には、ページフォールトによって新しく割り当てられた物理ページを破棄し、保存しておいた元の物理ページの参照回数を 1 増加させてから、対応するページにマッピングする。そのページは次の書き込みを検出するために、再び書き込み禁止にされる (図 1 のヒープ領域のページ)。このようにして、ページの内容がコピーされるのは実際に書き込まれたページだけで済む。また、状態保存後に書き込み可能に変更されたページに関して

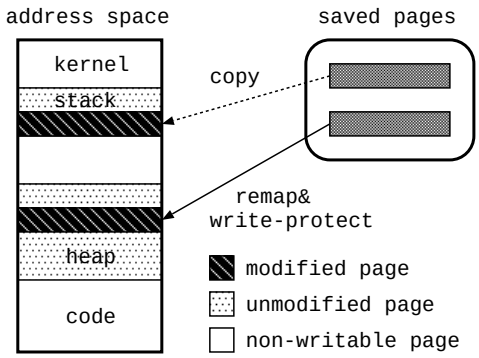

図 1: メモリの状態の復元

は、その時点で状態保存時の処理を行って書き込み禁止にすることにより、書き込みを検出することができる。

ページフォールトを発生させて、新しい物理ページを割り当てるという処理は比較的重い処理であるが、このオーバヘッドを軽減できる場合がある。サーバプロセスが行うリクエスト処理では、毎回似たようなコードが実行されることが多い。そのため、データの書き込みパターンも毎回似たものになると予測される。この性質を利用して、状態復元時に、書き込みが行われたページを全て破棄して書き込み禁止にする代わりに、再び書き込まれることが予測されるページについては、あらかじめ元のページの内容をコピーしてやる (図 1 のスタック領域のページ)。これを予測ページコピーと呼ぶ。これにより、次のリクエスト処理において同じページに書き込みが起こってもページフォールトは起こらない。その結果、ページフォールトが起こるのは予測が外れたページだけとなり、ページフォールトの回数を減らすことができる。ただし、予測が外れると先行して行ったコピーが無駄になるので、システムの性能を低下させる可能性もある。

状態保存前にマップされていた仮想メモリ領域に関しては、状態保存後にページインされた物理ページであっても、状態復元時に破棄する必要はない。このような状況は、デマンド・ページング機構によって、仮想メモリ領域への物理ページの割り当てが、必要になるまで遅らされるために生じる。サーバプロセスの性質上、このようなページは再びページインされる可能性が高い。それに、このようなページを状態保存前から存在していたものとして扱っても、セキュリティ的には問題はない。ただし、書き込み可能なページであれば、ページインされた時に上で述べたような手法を用いて元の状態に戻せるようにする必要がある。

このようにメモリの状態を元に戻すことにより、メモリに影響を残すトロイの木馬のような巧妙な攻撃を防ぐことができる。このような影響を放置しておいても、クラック攻撃をしてくるような信頼できないクライアントからのリクエストを処理している間は、強いアクセス制限が適用されるので悪影響は出ないかもしれない。しかしその後で、信頼できるユーザからのリクエストを弱いアクセス制限の下で処理する時には、悪影響が出る可能性がある。他の利点として、クラック攻撃によってメモリが破壊されたサーバプロセスのメモリを元の状態に戻すことによって、サーバを安定な状態に保つことができる。たとえセグメンテーション・フォールトなどが起こっても、シグナルハンドラを設定しておけば、その中から状態を復元するシステムコールを呼ぶことができる。これにより、サーバの不正利用の試みの結果としてのサービス拒否攻撃を防ぐことができる。

3.2.3 シグナル

状態を保存する時には、シグナルハンドラやブロックされているシグナルの一覧などを保存する。状態を復元する時には、それらが変更されていれば元の状態に戻す。ただし、シグナルハンドラを元に戻した後で未処理のシグナルが処理されると、意味的におかしくなるので、元の状態に戻す前に、キューに溜まっているシグナルを処理する。ブロックされていないシグナルについては、状態を復元する前にシグナルハンドラを実行する。ブロックされているシグナルについては、その時点ではシグナルハンドラを実行することはできないので、元の状態でもブロックされていればそのままにしておく。元の状態でブロックされていなければ、そのシグナルは破棄する。

シグナルハンドラを元の状態に戻すことにより、クラック攻撃によって意図的に変更されたシグナルハンドラが、アクセス制限を解除した後で呼ばれる、トロイの木馬攻撃を防ぐことができる。シグナルハンドラの不正な変更そのものを、アクセス制限によって禁止することもできるが、SIGALRM を利用している場合のように制限できないこともある。

3.2.4 ファイル・ソケット

状態を保存する時には、オープンされているファイル・ソケットを記録し、その参照回数を 1 増加させる。これは仮に状態保存後にファイル・ソケットがクローズされても、OS 内部にその管理情報が残るようにするためである。状態を復元する時には、その時点でオープンされているファイル・ソケットの参照回数を 1 減少させ、状態保存時にオープンされていたファイル・ソケットの参照回数を 1 増加させる。これにより、状態保存後にオープンされたファイルだけをクローズすることができる。

ファイル・ソケットの状態を元の状態に戻すことで、状態保存後に不正にオープンされたファイル・ソケットが利用できる上限を圧迫して、ファイル・ソケットがオープンできなくなる事態を避けることができる。また、たとえ不正にファイル・ソケットがクローズされても、状態復元時に再びオープンされた状態に戻されるので、サーバの処理が妨害されることはない。

3.2.5 その他

プロセスの優先度やリソースの使用制限に関して、状態を保存する時に nice 値や rlimit 値を保存しておき、状態を復元する時に元の値に戻す。これらの値は、root 権限で動くプロセス以外にとっては元に戻すことができないが、元に戻せるようにすることで、外部からのリクエストならばプロセスの優先度を下げて処理するようにしたり、攻撃に備えてリソースの制限をきつくしたりすることができる。また、カレントディレクトリや umask 値に関しても、状態を保存しておき、状態復元時に元の状態に戻す。

3.3 アクセス制限の解除

Compacto ではプロセスの状態を復元すると同時に、プロセスの状態を保存した後に追加されたアクセス制限を解除する。アクセス制限を解除できるようにするには、戻すべき元の状態を保存しておく必要がある。ここでいうアクセス制限の状態とは、Compacto が行うシステムコールの利用制限、ユーザ・グループ、ルートディレクトリなどである。

システムコールの利用制限に関しては、状態を保存する時にはその時点で適用されている利用制限の状態を記録しておき、状態を復元する時には状態保存後に追加された利用制限を無効にする。利用制限は追加されるのみなので、状態保存時の利用制限の内容を全て保存しておく必要はない。また、Linux にはケーパビリティ機構が実装されており、ケーパビリティを用いることによっても、root 権限を必要とするシステムコールの利用を制限することができる。そこで、状態保存時にプロセスが保持しているケーパビリティも保存しておき、状態復元時に元の状態に戻す。システムコールの利用制限を安全に元の状態に戻せるので、アクセス権限を常にできるだけ小さくしてプロセスを動かすことができる。

ユーザ・グループに関しては、状態を保存する時にはユーザ ID(UID)・グループ ID(GID) を保存し、状態を復元する時にそれらを元の値に戻す。対象とする UID/GID は、実 UID/GID、実効 UID/GID、保存 UID/GID、および、ファイルシステム UID/GID である。ファイルシステム UID/GID は Linux 特有のもので、ファイルシステムのアクセスチェックにのみ用いられる UID/GID である。プロセスの UID/GID を安全に元に戻すことができるので、必要に応じて setuid/setgid システムコールを用いて、様々なユーザ権限でプロセスを動かすことができる。

ルートディレクトリに関しては、状態を保存する時には、そのディレクトリ・エントリを保存しておき、状態を復元する時には、その時点でのディレクトリ・エントリを破棄し、保存しておいたディレクトリ・エントリに戻す。ルートディレクトリを安全に元に戻すことができるので、必要に応じて chroot システムコールを用いて、別々のディレクトリ構成をプロセスに見せることができる。

内容	時間 (μs)
システムコール	0.79
レジスタ	0.25
メモリ	3.36 (6.20)
シグナル	0.13
ファイル・ソケット	0.09
その他	0.10
計	4.62 (7.52)

表 1: プロセス・クリーニングのオーバヘッドの内訳 (カッコ内の数値は予測ページコピーを行わなかった場合)

4 実験

我々は Linux 2.2.16 カーネルをベースにして Compacto システムを開発しており、このシステム上でプロセス・クリーニングを行い、アクセス制限を安全に解除する機構を実装した。そして、プロセス・クリーニングを行うことによって、サーバがどのくらいのオーバヘッドを被るかを調べた。

4.1 マイクロベンチマーク

状態の保存に関しては最初の一回しか行われないため、そのオーバヘッドはほとんど問題にならないが、状態の復元はサーバのリクエスト処理ループで毎回行われるので、そのオーバヘッドはサーバの性能に影響を及ぼす。プロセス・クリーニングでは、プロセスを元の状態に戻すために様々な処理を行っているが、どの処理にどのくらいの時間がかかっているのかを測定した。測定に用いたマシンは Pentium II 400MHz、L1 キャッシュ16KB(命令)/16KB(データ)、L2 キャッシュ512KB の CPU を 1 つ搭載し、メインメモリは 128MB であった。

何も行わないループで状態の復元のみを繰り返し行うプログラムにおいて、プロセス・クリーニングのオーバヘッドの内訳は表 1 のようになった。メモリのカッコ内の数値は予測ページコピーを行わず、毎回 copy-on-write を行った場合の測定結果である。このベンチマークプログラムにおいて、メモリアクセスはスタックに対してのみ行われ、変更されるページは 1 ページであり、ページコピーの予測は必ず成功する。その他の資源の状態は全く変化しない。この結果から、メモリの状態を元に戻すのに大きなオーバヘッドがかかっていることが分かる。また、予測ページコピーを行うことにより、このオーバヘッドを大幅に減らすことができている。また、全くアクセス制限を追加しなかった場合に、アクセス制限を解除するためのオーバヘッドは $0.2\mu s$ であった。

プロセスの状態を保存しておき、処理が終わったら状態を復元させる代わりに、子プロセスを作って処理させ、処理が終わったら終了させる場合についても測定した。子プロセスの影響は親プロセスには及ばないので、親プロセスはプロセス・クリーニン

グと同様の効果を得ることができる。この測定の結果、子プロセスを作って終了させるのにかかるオーバヘッドは 100μs であることが分かった。これはプロセス・クリーニングに比べ、はるかに大きなオーバヘッドとなっている。

4.2 Apache ウェブサーバ

実際のサーバプログラムでプロセス・クリーニングを行うことにより、どの程度性能が低下するのかを調べるために、apache ウェブサーバ [1] を用いて実験を行った。apache ウェブサーバは広範に用いられており、プロセスプールの手法を用いて性能の向上を図っている。実際に用いられているサーバプログラムを用いて実験を行うことにより、プロセス・クリーニングの現実的な書き込みページ数、ファイル・ソケット数におけるオーバヘッドを測定することができる。

ウェブサーバプログラムには apache 1.2.13 を用い、サーバの OS にはアクセス制限の安全な解除機構が実装された Linux 2.2.16 を用いた。サーバマシンは Pentium II 400MHz、L1 キャッシュ16KB(命令)/16KB(データ)、L2 キャッシュ512KB の CPU を 1 つ搭載し、メインメモリは 128MB であった。ネットワーク・インタフェース・カードには 3COM Fast EtherLink を 1 つ用いた。一方、クライアントの OS には標準の FreeBSD 3.4 を用いた。クライアントマシンは Celeron 300MHz の CPU を 1 つ搭載し、メインメモリは 64MB であった。ネットワーク・インタフェース・カードには Intel EtherExpress Pro 10/100B を 1 つ用いた。8 台のクライアントマシンは、100Mbps のスイッチングハブでサーバマシンと接続されている。

サーバ性能の測定には WebStone 2.5 [9] を用い、合計 4 種類のウェブサーバについて測定を行った。一つは予測ページコピーを使わずにプロセス・クリーニングを行うサーバ、もう一つは予測ページコピーを使ってプロセス・クリーニングを行うサーバである。比較のために、apache 標準のサーバでも測定を行った。また、プロセス・クリーニングを行う代わりに、子プロセスを作ってリクエストを処理させ、処理が終了したらその子プロセスを終了させるようにしたサーバについても測定を行った。どの場合についても、プロセスプールされるプロセスの数は 16 に固定した。

クライアント台数を 1、2、4、8 と変えて、サーバの平均応答時間とスループットを測定した結果が図 2 と図 3 である。サーバの応答時間はクライアントがサーバに接続要求を出してから、0 byte の HTML ファイルをリクエストしてそのファイルを取得し、接続を切断するまでの時間である。スループットはクライアントが 0 byte の HTML ファイルをリクエストする場合に、単位時間当たりにサーバが処理できたリクエストの数である。また、一回のリクエスト処理で書き込みが行われたメモリページ数は 15 ページ、変更されたシグナルハンドラは 1 つ、オープンされたファイル・ソケットは 2 つであった。

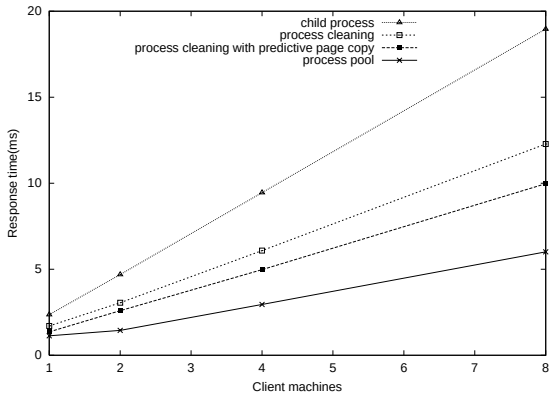

図 2: サーバの平均応答時間の変化

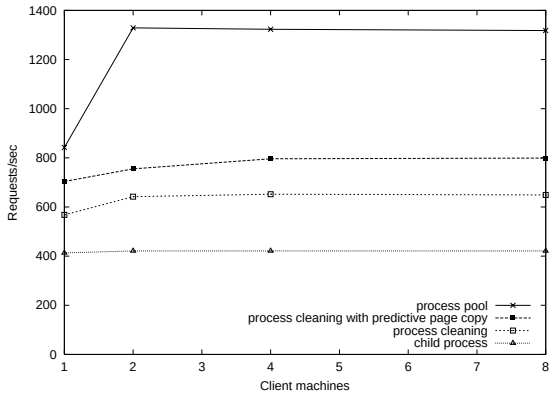

図 3: サーバのスループットの変化

クライアント台数が 1 台の時には、予測ページコピーを使ったプロセス・クリーニングを行うサーバは apache 標準のサーバと比べ、応答時間は 1.2 倍になり、スループットは 16%低下している。一方、子プロセスに処理させるサーバは apache 標準のサーバと比べ、応答時間は 2.1 倍になり、スループットは 51%低下しており、大幅に性能が落ちている。プロセス・クリーニングを行うサーバでは、予測ページコピーを使うことにより、応答時間が 0.8 倍になり、スループットが 24%向上しており、効果が現れているといえる。

クライアント台数が 8 台の時には、予測ページコピーを使ったプロセス・クリーニングを行うサーバは apache 標準のサーバと比べ、応答時間は 1.7 倍になり、スループットは 40%低下している。1 台の時に比べてかなり性能が低下しているが、これはサーバの負荷が高くなり、プロセス・クリーニングのオーバヘッドがネットワーク通信などに隠蔽されにくくなったせいだと考えられる。それでも子プロセスに処理させるサーバと比べると、応答時間は約半分で済み、スループットは 90%向上している。

5 関連研究

プロセス・クリーニングにおける状態の保存はチェックポインティング、状態の復元はロールバックであると考えることができる。このような技術は、データベースのトランザクションやプロセスの永続化などでも広く使われている。トランザクションや永続化では、必要になる度、または一定時間毎にチェックポインティングを行って状態を保存し、問題が生じた時にロールバックを行う。これらの場合には、ロールバックは例外的な処理なので、チェックポインティングの性能向上が重要になる。我々のシステムでもこのような使い方をすることは可能であるが、特にプロセスプールの手法を使ったサーバへの対応を考えている。このようなサーバでは、最初に一回だけ状態を保存し、その後何度も同じ状態に復元させる。そのため、ロールバックの性能向上の方が重要になり、予測ページコピーなどの最適化が有効となる。

OS による細粒度保護ドメイン [8] や Palladium [3] など、プロセス内でモジュールを安全に実行できるようにする研究が行われている。これらのシステムではサーバのリクエスト処理など、危険を伴なう処理をモジュールの中で行わせることにより、悪影響がプロセスの他の部分に広がらないようにすることができる。しかし、モジュール内には悪影響が蓄積するので、モジュールを使った後で安全な状態に戻す必要がある。プロセス全体を安全な状態に戻すのに比べると、オーバヘッドを小さくできるかもしれない。もしくは、リクエストの度にモジュールを作り、処理が終了したらモジュールを破棄するという方法をとる必要がある。この場合は、子プロセスを作って処理させるよりはオーバヘッドを小さくできるかもしれないが、プロセス・クリーニングより高速にできるかどうかは不明である。

サーバの安全性を確保するための技術の一つに、バッファオーバフロー攻撃を防ぐものがある。StackGuard [4] や PointGuard [5] は、専用コンパイラを用いて、関数の戻り番地や関数ポインタが格納されたメモリの隣りに特殊な値を書き込むようにし、その値が変化すればバッファオーバフローさせられたと見なしてサーバを終了させる。libsafe [2] はバッファの境界チェックを行うようにした C の標準ライブラリであり、既存のプログラムを変更せずにバッファオーバフローを検出できる。別のアプローチとして、スタックを実行禁止にすることで、スタックオーバフローで送り込まれた不正コードを実行できないようにするものもある [7]。

6 まとめと今後の課題

プロセスのアクセス権限を動的に、より柔軟に変更できるようにするために、プロセスのアクセス制限を安全に解除する機構を提案した。この機構では、プロセスを安全な状態に戻した後でのみ、アクセス制限の解除を許す。プロセスを安全な状態に戻す作業をプロセス・クリーニングと呼び、レジスタやメモリの内容、シグナルやファイル・ソケットの状態などを元の状態に戻す。この機構は、プロセスプールの手法を用いたサーバなど、プロセスに必要なアクセス権限が状況に応じて変わる場合に有用である。

プロセス・クリーニングで使われる予測ページコピーの予測の精度を上げるために、アクセスの統計を利用することが考えられる。現在の実装では、一旦書き込みが行われたページは次も書き込まれると予測しているが、ページに書き込みが行われた時にセットされるダーティビットを利用すれば、より予測の精度を上げることができると考えられる。

`printf` 関数や `write` 関数のようにライブラリの中でバッファリングを行っている場合、プロセス・クリーニングを行う前にバッファをフラッシュしてやる必要がある。バッファのフラッシュを行わなければ、一部のメッセージが出力されずに消えてしまう可能性がある。

プロセス・クリーニングはプロセスの状態を元に戻してしまうので、グローバルな情報を変更したい場合に問題が生じる。例えば、一回目の処理の際に初期化を行わなければならない場合や、統計情報を残したい場合などである。この問題への対処は、十分に安全性を考えながら行う必要がある。

参考文献

[1] http://www.apache.org/.

[2] Baratloo, A., Tsai, T. and Singh, N.: Transparent Run-Time Defense Against Stack Smashing Attacks, in *Proceedings of the USENIX Annual Technical Conference* (2000).

[3] Chiueh, T., Venkitachalam, G. and Pradhan, P.: Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions, in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 140–153 (1999).

[4] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, in *Proceedings of the 7th USENIX Security Symposium*, pp. 63–78 (1998).

[5] Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, in *Proceedings of the DARPA Information Survivability Conference and Expo* (1999).

[6] Nelson, M. and Ousterhout, J.: Copy-on-write for Sprite, in *Proceedings of USENIX Summer Conference*, pp. 187–201 (1988).

[7] http://www.openwall.com/linux/.

[8] Takahashi, M., Kono, K. and Masuda, T.: Efficient Kernel Support of Fine-grained Protection Domains for Mobile Code, in *Proceedings of the IEEE 19th International Conference on Distributed Computing Systems*, pp. 64–73 (1999).

[9] http://www.mindcraft.com/webstone/.