システムソフトウェアと
オペレーティング・システム 90−13
（2002．6．27）

システム・コール・レベルでのRPCに対するアクセス制御の強化

新城 靖† 中田 吉法‡ 板野 肯三†

†筑波大学電子・情報工学系
‡筑波大学情報学類
URL: http://www.is.tsukuba.ac.jp/~yas/
E-Mail: <yas@is.tsukuba.ac.jp>

要旨

この論文は、システム・コールに対するリファレンス・モニタ を用いてRPC に対するアクセス制御を強化する方法を提案している。この論文で述べているリファレンス・モニタ SysGuard は、ガードと呼ばれる、組み合わせ可能なカーネル・モジュールを用いる。各ガードは、システムコールの入口と出口で付加的なアクセス制御を行う。この論文は、SunRPC に対するアクセス制御を強化するためのガードの設計と実現について述べている。従来の IP 層におけるパケット・フィルタとは異なり、それらのガードは、フィルタリングのため RPC のプログラム番号や手続き番号が利用でき、さらに、単一ホスト内のプロセスを区別することもできる。

Enhancing access control for RPC at the system call level

Yasushi Shinjo† Yoshinori Nakata‡ Kozo Itano†

†Institute of Information Sciences and Electronics, University of Tsukuba
‡Colleage of Information Sciences, University of Tsukuba
URL: http://www.is.tsukuba.ac.jp/~yas/
E-Mail: <yas@is.tsukuba.ac.jp>

Abstract

This paper proposes a method to enhance access control for RPC by using a reference monitor for system calls. The reference monitor SysGuard described in this paper uses composable kernel modules called guards. Each guard performs additional access control at the entrance and the exit of system calls. This paper presents the design and implementation of a set of guards that enhance access control for SunRPC. Unlike conventional packet filters at the IP layer, those guards can use the program numbers and the procedure numbers of RPC for filtering, and they can distinguish processes within a host.

1　はじめに

RPC (Remote Procedure Call) は、分散システムを構築するために広く使われている技術の１つである。たとえば、NFS (Network File System) や NIS (Network Information Service) は、RPC の実装の１つである SunRPC [14] 上に構築されている。RPC は、デスクトップ環境等での単一のホスト内の通信へも利用が拡大している。

RPC の利用が拡大するにつれ、RPC によりサービスを提供しているサーバの弱点を狙う不正アクセスも増加している [2, 3, 4, 5]。パッチやソース・コードが入手できない場合や、入手可能であったとしても、それらを使って弱点を修正するまでの期間に、弱点を含んだサーバを使い続けなければならないことがある。そのような弱点を含む既存のソフトウェアに対して外付けの要素を追加し弱点を補うことができれば非常に有用である。

外付けでネットワークに対するアクセス制御を強化し、弱点を補う方法の１つに、パケット・フィルタを使う方法がある。従来のパケット・フィルタにより RPC に対するアクセス制御を強化しようとすると、次のような問題があった。

1. RPC レベルの概念であるプログラム番号や手続き番号を用いてフィルタリングが行えない。
2. 同一ホスト内のプロセスを区別できない。

このような問題点を解決するために、この論文では、システム・コールに対するリファレンス・モニタを利用して RPC に対するアクセス制御の強化する方法を提案する。リファレンス・モニタとしては、我々が開発している SysGuard を用いる [6]。SysGuard の特徴は、ガードと呼ばれる、高い移植性を持ち、組み合わせ可能 (composable) なカーネル内のモジュールを用いる点にある。この論文では、SunRPC のアクセス制御を強化するためのガードの設計と実現について述べる。

SysGuard を利用して RPC に対するアクセス制御を強化する方法には、次のような利点がある。

1. ガードによりシステム・コールの引数を調べることで、RPC レベルの概念であるプログラム番号や手続き番号を用いてフィルタリングを行える。
2. SysGuard が持っているスコープ機能を利用することで、同一ホスト内のプロセスを区別してアクセス制御を強化することができる。

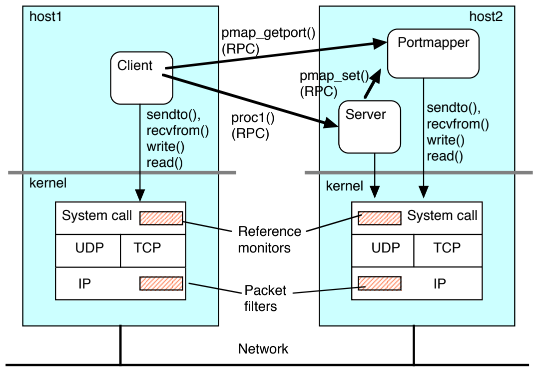

図 1: SunRPC の仕組みの概観とフィルタリング可能な場所

また、RPC 用のガードを他の汎用のガードと協調させて利用することもできる。

この論文は、次のように構成されている。2 章では、本論文が対象としている SunRPC の概要を示し、外付けでアクセス制御が強化可能な場所を示す。3 章では、SysGuard を用いてシステム・コール・レベルで RPC に対するアクセス制御を強化する方法について述べる。4 章では、RPC 要求メッセージに対するガードの実現について述べる。5 章では、実現したガードのオーバヘッドを調べる実験の結果を示す。6 章では、関連研究について述べる。最後に 8 章で以上についてまとめ、結論を述べる。

2　SunRPC とアクセス制御

この章では、RPC の１つである SunRPC の仕組みの概観を示し、外付けでアクセス制御が可能な場所を示す。図１に、SunRPC の概観を示す。これは、host1 上で動作しているクライアントが、host2 上で動作しているサーバに対して、手続き proc1() を実行している様子を示している。SunRPC では、サーバが動作しているホストでは、ポートマッパ[1] と呼ばれるプロセスが動作している。これは、SunRPC でプログラムを識別するために使われている情報であるプログラム番号とバージョン番号を、TCP/IP、または、 UDP/IP のポート番号に関連付けるプログラムである。

SunRPC では、次の手順で RPC が行われる。

[1]プログラム名は、portmap、または、rpcbind。

(1) サーバ・プロセスは、起動すると、ポートマッパの手続き pmap_set() を RPC により実行し、自分自身が使うポート番号を登録する。ポートマッパ自身のポート番号は、111 に固定されている。

(2) クライアント・プロセスは、ポートマッパの手続き pmap_getport() を RPC により実行し、TCP/IP、または、UDP/IP の番号をポート得る。

(3) クライアント・プロセスは、(2) で得られたポート番号を用いて個々の手続きを呼び出す。

SunRPC の当初の設計では、アクセス制御は、主にサーバ側の個々の手続きで行われる。個々の手続きでは、次のような種類のクライアントの証明 (credential) を調べることができる。

- AUTH_NONE: 証明がない。
- AUTH_SYS: Unix の UID (User Identfier)、GID (Group Identifier)、および、groups が含まれる。偽造が容易である。
- AUTH_DH, AUTH_KERB: DH (Diffie-Hellman)、Kerberos 等の技術を使ったディジタル署名が付加されたもの。

ディジタル署名技術を利用するには、クライアントとサーバの両方を対応させる必要があり、さらに、付加的なサーバが必要になる場合もある。したがって、これらの仕組みを利用していないプログラムが数多く存在している。また、たとえディジタル署名技術が使われていたとしても、バッファ・オーバーフロー攻撃などの攻撃を防ぐことはできない。

本来の RPC サーバに対して外付けでアクセス制御を強化する方法としては、次のような方法が考えられる (図 1)。

(1) ポートマッパにおいてアクセス制御を強化し、悪意を持つサーバ、および、クライアントからの要求を拒否する。ポートマッパを経由しないで直接 RPC サーバを利用された場合には、効果がない。

(2) クライアント側、および、サーバ側の RPC ライブラリを強化する。ライブラリ関数をバイパスされると効果がない。

(3) クライアント側、および、サーバ側の IP 層にあるパケット・フィルタを設定する。１章で述べたように、RPC レベルの概念を使ったフィルタリングができない、および、同一ホスト内のプロセスを区別することができないという問題がある。

(4) クライアント側、および、サーバ側のシステム・コールを受け付ける層にあるフィルタ（リファレンス・モニタ）を利用する。

(5) クライアントとサーバの間にプロキシを動作させる。要求の送り先がプロキシになってしまう場合、個々のサーバの手続きにおけるアクセス制御が無効になってしまう。

この論文では、(1)-(3) の問題点を、(4) で補う方法について述べる。

3 SysGuard による SunRPC に対するアクセス制御の強化

この章では、システム・コールに対するリファレンス・モニタ SysGuard を用いて SunRPC に対するアクセス制御を強化する方法について述べる。

3.1 SysGuard の概要

SysGuard は、システム・コールに対するリファレンス・モニタ／ラッパである [6]。SysGuard は、ガードと呼ばれる、組み合わせ可能 (composable) なカーネル内のモジュールを用いる。各ガードは、システム・コール処理の実行の前後で、付加的にアクセス権をチェックする。全てのガードが実行を許可した場合、システム・コールは、通常と同じように動作する（通上のアクセス制御も働く）。どれか１つのガードでも許可しなかった場合、システム・コールはエラーになる。システム・コールの実行後に働くガードの場合は、システム・コールの効果がキャンセルされ、システム・コールが再実行されることもある。

SysGuard の特徴は、各ガードのスコープを柔軟に設定できる点にある。たとえば、UID、GID、プロセスの親子関係等を用いて、あるガードが働くプロセスの集合を指定することができる。これにより、個々のガードが単純化され、再利用性が高まっている。また、デバイス・ドライバや Linux Security Module [9] のセキュリティ・モジュールとは異なり、標準のシステム・コールの引数と SysGuard の支援関数のみを利用しているガードは、他の同じ標準に従っているカーネルへ移植可能になっている。

１章で述べた２つの問題のうち、同一ホスト内のプロセスを区別できないという問題は、 SysGuard を利用することにより解決される。すなわち、SysGuard のスコープ機能を利用することにより、プロ

セスを認識して個別に RPC に対するアクセス制御を強化することができる。SysGuard が提供している標準のスコープでは記述できない場合、各ガードの内部で独自にプロセスを識別することもできる。

3.2 SunRPC に対するアクセス制御を強化するために有用な汎用のガード

SunRPC に対するアクセス制御を強化するために、UDP/IP や TCP/IP に対するアクセス制御を強化するガードも利用可能である。現在利用可能になっているネットワークに対する汎用のガードの一覧を、表 1 に示す。たとえば、(1) は、RPC 用のガードにおいて TCP/IP や UDP/IP のソケットだけに絞るため必要である。また、名前に Filter を含むものは、指定されたプロセスの集合に働くようなパケット・フィルタとして動作する。これらのガードには、RPC のメッセージを識別する機能はない。

3.3 SunRPC のアクセス制御を強化するガードの設計

SunRPC のアクセス制御を強化するためのガードを、次のような方針の元に設計した。

- サーバ側だけでなくクライアント側でもガードを動作させる。悪意のあるサーバ [13] からクライアントを保護することも重要である。同一ホスト内や保護された LAN ではクライアント側を制限することも安全性を高めるために有効である。
- できるだけ応答メッセージではなく要求メッセージの送信を止める。要求メッセージを止めた場合、通常の要求メッセージの紛失（送信に失敗）に対するエラー処理を行うだけでよく、RPC のセマンティクス上、対処が容易である。

この方針に基づき、具体的に次のようなガードを設計した。

(1) **RPCClientRequestFilter**
RPC のクライアント側で動作する。要求メッセージに含まれている RPC のプログラム番号、バージョン番号、および、手続き番号を用いてフィルタリングを行う。ルールに従い、問題があれば、sendto() や write() などメッセージを送信するシステム・コールをエラーにして要求メッセージを送らせない。

(2) **RPCServerRequestFilter**
ガード (1) と類似の機能を、RPC のサーバ側で動作する。 recvfrom() や read() システム・コールをエラーにすることで、要求メッセージがサーバ・プロセスで処理されないようにする。

(3) **RPCClientGetportFilter**

(4) **RPCServerGetportFilter**
クライアント側、または、サーバ側でポートマッパの機能を補う。手続き pmap_getport() の引数を調べ、特定のサービスへのアクセスを拒否する。

(5) **RPCClientForceGetport**
RPC のクライアントにおいて、RPC を行う前に必ずそのポート番号を ポートマッパの手続き pmap_getport() により入手していることを強制する。ガード (3) と (4) を効果のあるものにする。

(6) **PMapServerSetFilter**
ポートマッパの手続き pmap_set() と pmap_unset() のアクセス制御を強化する。同一ホスト内の通信[2]であることを利用し、カーネル内の情報で RPC のメッセージ中に含まれている証明を検証する。さらに、プログラムの名前などを用いてアクセス制御を行う。

(7) **NISClientPasswdScrambler**
NIS 手続きの結果で、パスワードのハッシュ値の部分をダミーの値で上書きした上で許可する。ただし、`/bin/login` など、特定のプログラムについては、変更しないでそのまま渡す。

これらのガードは、(7) を除いて、要求メッセージがサーバへ渡される前にブロックするものである。(7) は、応答メッセージを検査するもであるが、エラーを返さないので、エラー回復の問題は生じない。ただし、関数性が失われているので、他のガードとの組み合わせて利用する場合には注意を要する。

4 要求メッセージに対する基本的なフィルタの実現

3.3 節で述べたガードのうち、RPCClientRequest-Filter、および、RPCServerRequestFilter の実現を完了した。この章では、これらのガードにおける方針の記述、および、内部動作について述べる。その特徴は、仮想計算機を用いている点、および、ユーザ空間からのコピー回数を減らしている点にある。

[2]遠隔からの要求を拒否する機能は、すでに既存のポートマッパの実現に含まれている。

表 1: ネットワーク・アクセスを制限するガード

名前	説明
(1) NoRawSocket	SOCK_RAW について socket() を止める。
(2) TCPClientOnly	システム・コール accept() を完全に止める。
(3) UDPClientOnly	UDP/IP による通信のうち、要求の送信と応答の受信というパタンしか許さない。
(4) TCPServerPeerFilter	システム・コール accept() の結果を調べ、特定のクライアントしか接続させない。
(5) UDPServerPeerFilter	TCPServerPeerFilter の UDP 版。recvfrom() や recvmsg() の結果を調べる。
(6) TCPClientPeerFilter	TCPServerPeerFilter のクライアント用。connect() に働く。
(7) UDPClientPeerFilter	TCPClientPeerFilter の UDP 版。
(8) TCPServerRateFilter	accept() の頻度を調べ、特定のクライアントからの過大な頻度の要求を拒否する。
(11) UDPServerRateFilter	TCPServerRateFilter の UDP 版。

4.1 アクセス制御方針の記述

ガード RPCClientRequestFilter、および、RPCServerRequestFilter は、SunRPC の要求メッセージの内容を調べ、ルールに基づき許可、または、拒否を判定する。これらのガードを、任意の SunRPC サービスに対して動作させることを目標にする。このため、フィルタリングのルールを記述するパラメタとして、次の整数を用いることにした。

- 通信相手のホストの IP アドレス
- 通信相手のホストのポート番号
- SunRPC のプログラム番号
- SunRPC のバージョン番号
- SunRPC の手続き番号

これらの整数について、次のような条件判定を行うことができるようにした。

- 等しい
- 等しくない
- 指定された範囲に入っている
- 指定された範囲に入っていない

アクセス制御の方針を、ルールのリストとして記述する。各ルールは、次のいずれかを返す。

- 許可
- 拒否
- 次のルールを評価する (条件に適合しなかった場合)

あるルールが、許可、または、拒否を返した場合、ガード全体としては、その結果を返す。それ以外の場合は、次のルールの評価に進む。いずれのルールも許可、または、拒否を返さなかった場合、ガード全体としては許可を返す。

```
1:  progeq 100004
2:  proceq 8
3:  deny
```

図 2: アクセス制御の記述例 (NIS ypall() の禁止)

図 2 に NIS の手続き ypall() [3]の実行を禁止するためのルールの記述を示す。1 行目は、SunRPC のプログラム番号が 100004 (ypserv) であることを意味する。2 行目は、手続き番号が 8 であることを意味している。これら 2 つの条件に適合した場合、全体として拒否を返す。

このようにして記述されたアクセス制御の方針は、仮想計算機の機械語に翻訳してカーネル内のガード・モジュールに送る。図 2 は、その仮想計算機のアセンブリ言語で記述されたプログラムである。この仮想計算機は、BPF (BSD Packet Filter)[10] に習い、スタックを持たない簡単なものになっている。

仮想計算機を利用したことの一般的な利点としては、安全性を確保しながら拡張が容易であることがあげられる。さらに、 RPC に対するアクセス制御を強化するという観点から、次のような利点がある。

- ユーザ・インタフェース・レベルでワイルドカード (any,*) を用いる方法と比較して、ルールの記述が簡潔になる。比較する必要がない部分は、記述する必要がない。
- システム内部においてワイルドカードを含む表を使う方法と比較して、ルール評価のステップ数が少なくなる。順序の入れ替えによる最適化が可能になる。

[3]NIS のパスワード・マップを盗み出すためにしばしば用いられる。NIS の主要な機能は、手続き ypmatch() において実現されている。

4.2 仕組み

ガード RPCClientRequestFilter は、次のようなシステム・コールを監視する。

- send(), sendo(), sendmsg() 実行前
- write() の実行前
- connect() の実行後

ガード RPCServerRequestFilter は、次のようなシステム・コールを監視する。

- recv(), recvfrom(), recvmsg() の実行後
- read() の実行後
- accept() の実行後

これらのシステム・コールを監視する中で、特に問題になるのは、write() や read() システム・コールの引数に現れたユーザ空間のメモリの内容を調べている点にある。この 2 つのシステム・コールは、TCP/IP を用いた RPC だけでなくファイル入出力や、HTTP による通信などの RPC 以外の通信でも用いられる。このように多用されるシステム・コールでメモリの内容を検査すると、性能が大きく低下する恐れがある。3.2 節で述べたガードには、このようにメモリの内容を検査するものはない。

ガード RPCClientRequestFilter、および、RPCServerRequestFilter では、write() や read() におけるオーバヘッドを減らすために、次のような工夫を行っている。

(1) プロセスごとに監視すべき TCP/IP のコネクションに対応したファイル記述子のリストを設ける。まず、connect() 、または、accept() システム・コールにより作成されたものを全てリストに加える。

(2) write() 、または、read() システム・コールにおいて、(1) のリストを調べる。リストにあれば、RPC 要求メッセージとして考え、ユーザ空間からデータをコピーして検査する。ファイルのように、open() システム・コールで得られたものは、リストには含まれていないので、コピーを行わずに許可を返す。

(3) メッセージを検査した結果、TCP/IP ではあるが、 RPC の要求メッセージではない (HTTP など他の通信プロトコルである) と判定された場合、リストからはずす。

RPC の要求メッセージかどうかの判定には、次のような RPC ヘッダに含まれた冗長性を利用している。

- メッセージの種別（要求か応答か）が、0(要求) である。
- RPC プロトコルのバージョンが 2 である。

5 実験

SysGuard は、現在、Intel x86 プロセッサ版、および、DEC Alpha プロセッサ版の Linux カーネル 2.2.18 で利用可能になっている。4 章で述べたガードの動作を確認し、オーバヘッドを測定する実験を、Intel x86 プロセッサ版の SysGuard を利用して行った。実験に用いたハードウェアは、CPU が Pentium III（Coppermine）1GHz（FSB 133MHz）、メモリが 384M バイト（PC133 CL2）のパーソナル・コンピュータである。

5.1 動作確認

実現したガード RPCClientRequestFilter、および、RPCServerRequestFilter の動作を確認した。RPCClientRequestFilter を設置した状態で、手続き ypall() を実行するクライアント `yp-all` を実行すると、次のように write() システムコールによる要求メッセージの送信に失敗する。

```
% ./yp-all $host $domainname passwd.byname
yp_all: clnt_call: RPC: Unable to send;
errno = Permission denied
% _
```

ここで `yp-all` には、RPC の実行に必要な 3 つの引数 (サーバのホスト名、NIS のドメイン名、NIS のマップ名) を与えている。

次に、ガード RPCClientRequestFilter のインスタンスを、ypserv のプロセスに対して設定した。ガードが設定されていない状態でクライアント `yp-all` を実行すると、次のようにガードが設置されていない read() システムコールが失敗している。

```
% ./yp-all $host $domainname passwd.byname
yp_all: clnt_call: RPC: Unable to receive;
errno = Connection reset by peer
% _
```

サーバ側では、ガードの働きで、read() がエラーを返すので、サーバは、その TCP/IP のコネクション

を切断する。その結果、クライアントでは、read() がエラー (ECONNRESET) になっている。

5.2　オーバヘッドの測定

ガード RPCClientRequestFilter のオーバヘッドを測定するために、次の2つのプログラムを用いた。

yp-match NIS サーバについて、ypmatch() 手続きを繰り返し実行する。 UDP/IP では、sendto()、TCP/IP [4]では write() の各システム・コールに対するガードの手続きが呼ばれる。約 120 バイトのデータが送受信される。

write ファイルに対して指定されたバイト数の書き込みをくり返し実行する。write() システム・コールに対するガードの手続きが呼ばれる。

ガードには、4.1 節で述べた ypall() 手続きを全て拒否するルールを登録した。ルールの数とオーバヘッドの関係を調べるために、同一のルールを複数回登録した。各手続き (ypmatch()、または、write()) を、100 回から 10000 回実行し、全体の実行時間を測定し、実行回数で割ることで1回当たりの実行時間を得た。同じ実験を 100 回繰り替えし、その最も短い時間を測定した。ypmatch() では、ネットワーク通信の変動を排除するために、クライアントとサーバを同一のホストで実行した。

表 2 に、実験の結果を示す。ypmatch() の場合、標準カーネルと比較して、SysGuard を導入した段階で 0.0 μ 秒から 0.8 μ 秒、ルールを1個追加した状態で、3.1 μ 秒から 2.4 μ 秒、それぞれ実行時間が増加している。また、ルールを 100 個登録した場合と比較することで、ルール1個当たり、 0.1 μ 秒の割合いで実行時間が増加することがわかる。ypmatch() を 100Base-TX のハブで接続した2台のコンピュータ間で実行すると、90%の範囲が 145 μ秒から 342μ 秒であった。したがって、実際の NIS の利用では、ガードによるチェックのオーバヘッドは、ネットワーク通信の変動の中に埋もれてしまう。

ファイルに対する書き込みでは、最大 1.2 μ 秒のオーバヘッドがある。この時間は、ディスク・キャッシュに書き込む時間であり、実際の入力を伴う処理では、ディスクの回転待ち時間 (15000rpm で 0m 秒から 4m 秒) の変動に吸収される。また、ルールの数が増えても、実行時間は増大していないことから、4.2 節で述べた最適化が有効に動作していることがわかる。

次に、RPC を用いないアプリケーションに対する影響を調べるために、コンパイル作業（GNU patch 2.5.4）に要する時間を測定した。このアプリケーションを選んだ理由は、fork() システム・コールとファイルへの write() システム・コールが多用されているからである。SysGuard では、fork() システム・コールにおいてプロセスごとのジャンプ表のコピーを行う。また、実験で用いたガード RPCClientRequestFilter は、fork() システム・コールにおいて、プロセスごとに管理しているファイル記述子の表をコピーしている。コンパイル作業中に発行された fork() の回数は 64 回であった。

表 3 に、測定の結果を示す。括弧内の数値は、標準のカーネルに対する実行時間の増加の割合を示している。この実験では、RPC に対する要求のフィルタを導入しても、標準のカーネルに比べて実行時間の増加は、1% 以下であった。

6　関連研究

The Eternal System は、CORBA オブジェクトに対してフォールト・トレランスを提供するシステムである [11]。 CORBA では、interceptor と呼ばれる概念を利用して、メッセージの書き換えや機能拡張を許している [12]。 The Eternal System は、interceptor として CORBA のメッセージを横取りし、信頼性のあるマルチキャストや複製管理を実現している。具体的には、Solaris が提供しているシステム・コールのトレース機能や動的リンク・ライブラリの置き換えを利用している。 The Eternal System と比較して、この論文で述べた方法の特徴は、悪意があるプログラムに対応できる点、および、カーネル内のリファレンス・モニタを利用している

表 2: 手続きの最小実行時間（単位:μ秒）

手続き	引数	標準カーネル	SysGuard 付きカーネル ガードなし	ガード付き 1 rule	ガード付き 100 rules
ypmatch()	udp	41.0	41.0	44.1	53.4
ypmatch()	tcp	52.9	53.7	55.4	65.4
write()	0 B	0.38	0.38	1.03	1.03
write()	1 B	0.82	0.85	1.51	1.51
write()	4 KB	19.7	19.7	20.3	20.4
write()	8 KB	40.1	40.5	41.3	41.3

[4] コネクションを確立するための connect() の実行時間は含まれていない。

表 3: patch プログラムのコンパイルに要する時間（平均、単位：秒）

標準カーネル	SysGuard 付きカーネル		
	ガード無し	ガード付き	
		1 rule	100 rules
5.32	5.32 (+0.0%)	5.33 (+0.2%)	5.37 (+0.9%)

点にある。

ユーザ・レベルでアクセス制御を強化する方法として、カーネル空間からのコールバックを用いる方法 [15] や、 Unix System V が提供するシステム・コールのトレース機能を用いる方法 [1, 8] がある。本論文で述べた SysGuard では、カーネル内のモジュールによりシステム・コールの実行の可否を判断する。従って、より高速な実現が可能である。

Generic Software Wrappers[7] では、システム管理者が WDL（Wrapper Description Language）と呼ばれる汎用の言語を用いて、アクセス制御モジュールを記述することができる。4 章で述べたガードでは、SunRPC に対するアクセス制御を強化するために特化したマイクロ言語により方針を記述する。したがって、記述が簡潔であり、また言語処理系や仮想計算機も簡単になっている。

7 おわりに

この論文では、RPC に対して外付けのモジュールでアクセス制御を強化する方法として、システム・コールに対するリファレンス・モニタを利用する方法を提案した。そして Unix のシステム・コールに対するリファレンス・モニタ SysGuard を利用して、いくつかのアクセス制御を強化するカーネル・モジュール (ガード) を設計し、SunRPC の要求メッセージに対して働く基本的なフィルタを実現した。実現したガードにより、SunRPC のプログラム番号や手続き番号を利用したアクセス制御やホストより細かい単位でのフィルタの設定が可能になった。実験の結果、実現したガードによる実行時間の増加は、ネットワーク通信やディスク入出力の実行時間の変動に吸収される程度であることがわかった。

今後は、設計したガードのうち、まだ実現していないガードの実現を進める。また、 SunRPC のインタフェース記述と同水準でアクセス制御を記述できる仕組みを実現したい。

参考文献

[1] Acharya, A. and Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications, *Proceedings of the 9th USENIX Security Symposium* (2000).

[2] CERT Advisory CA-1999-08: *Buffer Overflow Vulnerability in Calendar Manager Service Daemon, rpc.cmsd* (1999).

[3] CERT Advisory CA-2000-17: *Input Validation Problem in rpc.statd* (2002).

[4] CERT Advisory CA-2001-27: *Format String Vulnerability in CDE ToolTalk* (2001).

[5] CERT Advisory CA-2002-10: *Format String Vulnerability in rpc.rwalld* (2002).

[6] 榮樂恒太郎, 新城靖, 板野肯三: システム・コールに対するラッパ/リファレンス・モニタ SysGuard の設計と実現, 情報処理学会論文誌, Vol. 43, No. 6 (2002).

[7] Fraser, T., Badger, L. and Feldman, M.: Hardening COTS Software with Generic Software Wrappers, *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 2–16 (1999).

[8] Kato, K., Oyama, Y., Kanda, K. and Matsubara, K.: Software Circulation using Sandboxed File Space Previous Experience and New Approach, *Proceedings of 8th ECOOP Workshop on Mobile Object Systems* (2002).

[9] Linux Security Module: *http://lsm.immunix.org/* (2002).

[10] McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *USENIX Winter 1993 Conference*, pp. 259–270 (1993).

[11] Narasimhan, P., Moser, L. E. and Melliar-Smith, P. M.: Using Interceptors to Enhance CORBA, *IEEE Computer*, Vol. 32, No. 7, pp. 64–68 (1999).

[12] Object Management Group: *CORBA v2.4.2* (2001).

[13] OpenSSH Security Advisory (adv.channelalloc): *Off-by-one error in the channel code* (2002).

[14] RFC1831: *RPC: Remote Procedure Call Protocol Specification Version 2* (1995).

[15] 品川高廣, 河野健二, 高橋雅彦, 益田隆司: 拡張コンポーネントのためのカーネルによる細粒度軽量保護ドメインの実現, 情報処理学会論文誌, Vol. 40, No. 6, pp. 2596–2606 (1999).