情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 2　166–177 (July 2009)

実行時のフェーズを用いたセキュリティポリシ記述の簡略化

品川高廣[†1]　忠鉢洋輔[†2]
河野健二[†3]　加藤和彦[†1]

近年，インターネットサーバに対する不正アクセスが頻繁に行われている．不正アクセスの被害を抑える仕組みとしてはサンドボックスと呼ばれる方式が広く研究されているが，被害を最小限に抑えるためにはサーバの資源に対して細かい粒度で保護を行う必要があり，セキュリティポリシの記述が複雑化することが問題となっている．本研究では，サーバプログラムにおける実行時のフェーズを用いることで，セキュリティポリシの記述を簡略化できることを示す．インターネットからメッセージを受け取る前後でプログラムを 2 つのフェーズに分け，フェーズ間でセキュリティポリシを切り替えることにより，ポリシ記述を大幅に簡略化することが可能になる．本研究では，HTTP，SMTP，POP のそれぞれのサーバに対して実際にポリシ記述を行い，ポリシ記述が実際に簡略化されることを確認した．

Simplifying Security Policies by Exploiting Execution Phases

Takahiro Shinagawa,[†1] Yousuke Chubachi,[†2]
Kenji Kono[†3] and Kazuhiko Kato[†1]

Internet servers are often attacked by crackers to gain unauthorize access. To mitigate the damage of unauthorized access, many researches have been conducted on developing sandbox systems. Unfortunately, minimizing damage of unauthorized access requires fine-grained protection of server resources, complicating the description of security policies in sandbox systems. This paper shows a scheme to simplify security policy descriptions by exploiting execution phases of server programs. This scheme divides program execution into two phases: the initial phase and the protocol processing phase. By switching security policies between the two phases, the description of security policies can be significantly simplified. This paper shows experimental results that describe security policies of HTTP, SMTP, and POP servers, and shows that the description of security policies is actually simplified.

1. はじめに

近年のインターネットでは，Web サーバやメールサーバなどのインターネットサーバに対する不正アクセスが頻繁に行われている．不正アクセスを行う攻撃者は，サーバプログラムに対して細工を施した不正なメッセージを送りつけて脆弱性を攻撃することにより，サーバの制御を乗っ取ったり権限のないファイルを読み書きしたりすることが可能である．サーバプログラムはますます高機能化・複雑化しており，プログラムの脆弱性を完全に取り除くことは難しい．

サーバへの不正アクセスを防ぐ手法の 1 つとして，サンドボックスという仕組みが広く研究されている[1)]．サンドボックスとは，ファイルなどの資源へのアクセスを通常より制限した環境でプログラムを動作させる仕組みのことで，仮にプログラムの制御が乗っ取られても，アクセス可能な資源はサンドボックスの中に限定され，サンドボックスの外に被害が及ぶことを防止できる．サンドボックスの中でアクセス可能な資源を定める**セキュリティポリシ**を適切に記述することで，不正アクセスの被害を最小限に抑えることができる．

サンドボックスは様々な実装方式が提案されているが，なかでもシステムコールレベルでアクセスを監視する方式は，実現の容易性，既存のプログラムに対する透過性，多くのシステムで使える汎用性などの利点から数多くのシステムで用いられている[2)–6)]．この方式では，サンドボックス内のプログラムが発行したシステムコールを別のプログラムで横取りし，セキュリティポリシに反するシステムコールの実行を禁止する．たとえば Web サーバでは，HTML ファイルへの読み出しは許可するが，パスワードファイルへのアクセスは禁止するなどの細かい粒度のポリシを実現できる．

†1 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba
†2 筑波大学情報学群情報学類
College of Information Science, School of Informatics, University of Tsukuba
†3 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Faculty of Science and Technology, Keio University

© 2009 Information Processing Society of Japan

サンドボックスのように細粒度での保護を実現するシステムでは，セキュリティポリシの記述が複雑になることが問題となっている[3),7),8)]．保護の粒度を細かくすると個々の資源に対して逐一アクセスの可否を記述しなければならず，ポリシの記述量が多くなってしまう．また，アクセスの可否を正しく判断するためには個々の資源に対する専門的な知識が必要となり，一般のユーザやプログラマにとってポリシを適切に記述すること自体が難しくなる．たとえば Web サーバの Apache では実行環境によっては数百個ものファイルにアクセスするため，これらのファイルに対して過不足なく適切なポリシを記述しなければならない．ポリシ記述が複雑になると，ポリシ自体の正当性を保証することが難しくなる．

本論文では，インターネットサーバにおけるセキュリティポリシ記述を簡略化する手法として，実行時のフェーズを用いたポリシ記述方式について述べる．本手法では，インターネットからのリモート攻撃を想定した場合，通信相手からのメッセージを受け取った後でなければサーバに対する攻撃を仕掛けられない点に着目し，メッセージを受け取る前後でセキュリティポリシを切り替える．たとえば，メッセージを受け取る前の**初期化フェーズ**では，攻撃を受ける危険性がないため，サーバプログラムのアクセスを制限しない．一方，メッセージを受け取った後の**プロトコル処理フェーズ**では，バッファオーバフロー攻撃をはじめとした様々な攻撃を受ける危険性があるため，サーバの実行に最低限必要な資源のみアクセスを許可する．

提案手法は，セキュリティポリシの記述に関して以下のような利点がある．

記述量の削減　初期化フェーズでアクセスする資源に対するポリシ記述が不要なため，ポリシ全体の記述量を大幅に削減できる．たとえば Web サーバの Apache では，ファイルアクセス全体の 99%が初期化フェーズでアクセスされるため，これらのファイルに対するポリシ記述を削減できる．

難易度の低下　初期化フェーズでアクセスするライブラリや独自のシステムコールなど実行環境に依存した資源に対する専門的かつ詳細な知識が不要になる．したがって，サーバ処理に本質的な部分のポリシだけを記述すればよくなり，直感的に理解しやすくなる．たとえば Apache では，HTTP によって送信することを許可する HTML ファイルに関するポリシだけを記述すればよい．

汎用性の向上　システム固有の初期化処理など実行環境への依存性が低下するため，OS がバージョンアップアップした場合や他の OS に適用する場合などにポリシの書き換えが少なくて済む．また，Web サーバ向けポリシなど同じ機能を持つサーバ間でのポリシの共有や再利用も容易になる．

ポリシ記述を簡略化するためのアプローチとしては，GUI を用いたポリシ記述過程の支援機構や，インクリメンタルにポリシを生成する手法が提案されている[9)–11)]．しかしこれらの手法では，最終的には初期化フェーズでアクセスする資源に対するポリシ記述も必要であり，適切にポリシを記述するためには，実行環境に対する詳細な専門知識が必要となる．また，セキュリティポリシを動的に切り替える機構[11)–14)]も提案されているが，これらの手法ではポリシ記述の簡略化に対する有効性は明らかにされていない．

本論文では，フェーズを用いる方式の有効性を示すために，サーバプログラムが発行するシステムコールの分析を行い，ポリシの記述量や難易度，汎用性の観点から比較を行う．また，実際にポリシ記述が簡略化されることを示すために，サンドボックスの一種である LIDS[15)] をベースとしてフェーズを用いたポリシ切替え機構を実装し，HTTP，SMTP，IMAP の各サーバプログラムに対して実際にポリシ記述を行った結果について述べる．

本論文の構成は，以下のとおりである．まず 2 章では，サンドボックスの概念とサンドボックスにおけるポリシ記述の困難さについて述べる．3 章では，提案するフェーズを用いたポリシ記述とその有効性について述べる．4 章では，LIDS をベースとしたフェーズに基づくポリシ切替え機構の実装について述べる．5 章では，システムコールの分析および実際のポリシ記述を行って，提案方式の有効性を評価する実験の結果について述べる．6 章で関連研究について述べ，7 章で本論文をまとめる．

2. サンドボックスとセキュリティポリシ

本章では，まずサンドボックスの基本的な概念とその実現手法を説明し，サンドボックスをインターネットサーバに適用することの有効性について述べる．次にサンドボックスにおけるセキュリティポリシ記述について述べ，その後，適切なセキュリティポリシの記述が難しい理由について説明する．

2.1 サンドボックスの概念

サンドボックスとは，資源へのアクセスを制限した実行環境でプログラムを動作させる手法を表した概念である（**図 1** 参照）．信頼できないプログラムや攻撃で乗っ取られる可能性のあるプログラムを実行する際に，サンドボックスと呼ばれる強固な「箱」の中で動作させることで，「箱」の内部の資源には自由にアクセスできるが，「箱」の外部の資源には決してアクセスできないという保護モデルを表している．

サンドボックスに基づく保護モデルは，プログラムを箱の中に閉じ込めることにより，不正アクセスによる被害を一定の範囲に限定することを目的としている．一方，サンドボック

© 2009 Information Processing Society of Japan

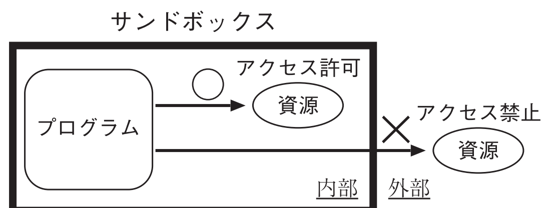

図 1　サンドボックスの概念
Fig. 1　The concept of sandbox.

スは攻撃自体を防ぐものではないため，いったん攻撃を受けるとサンドボックスの内部の資源には自由にアクセスされてしまう．したがって，不正アクセスの被害を最小限に抑えるためには，最小特権の原則[16] に基づいて，サンドボックスの内部に入れる資源を可能な限り少なくする必要がある．

2.2　サンドボックスの実現手法

サンドボックスの概念を実現したものとしては，Java におけるサンドボックスが有名である[17]．一方，特定の言語に依存しない汎用的な仕組みとしては，システムコールを監視する手法が広く知られている[2)–6]．この手法では，プログラムが発行するシステムコールを参照モニタと呼ばれる監視プログラムで検査して，アクセスする資源がサンドボックスの内部か外部かを，セキュリティポリシに基づいて判断する．

システムコール監視により実現するサンドボックスの実装方法としては，デバッグ用の機能を応用して別プロセスで監視する手法[2)–4] や，カーネルの内部に参照モニタを組み込む手法[5]，細粒度保護ドメインを応用した手法[6] などがある．いずれの手法でも，プログラムが資源にアクセスする際は必ずシステムコールを発行する必要があるため，プログラムが発行するすべてのシステムコールを漏れなく確実にチェックすることで，強固なサンドボックスを実現することができる．

2.3　インターネットサーバに対するサンドボックス

インターネットサーバに用いられるプログラムでは，しばしばセキュリティ上の脆弱性が指摘されており，これらの脆弱性を攻撃することでインターネットからサーバに対する不正アクセスが行われてしまう．たとえば，Web サーバの Apache では，過去にバッファオーバフロー攻撃に対する脆弱性[18] が指摘されており，インターネットからこの脆弱性を攻撃してサーバプロセスを乗っ取り，任意のコードを実行することが可能であった．Web サーバプロセスが乗っ取られると，そのプロセスの権限でアクセス可能なすべてのファイルが危険にさらされてしまう．たとえば，ユーザのホームディレクトリやログファイル，パスワードファイルなどに対して不正アクセスが行われたり，他のマシンに対する攻撃の踏み台にされたりする危険性がある．

そこで，インターネットサーバをサンドボックス内で動作させることにより，不正アクセスの被害を小さくできる．たとえば，Apache をサンドボックス内で動作させて，HTML ファイルへのリードオンリーアクセス以外を禁止することにより，仮にサーバプロセスが乗っ取られたとしても，Apache の動作に関係ないファイルへの不正アクセスなどを防止することができる．

サンドボックス方式の利点としては，特定の攻撃手法に限定せずに保護を実現できる点があげられる．たとえばサーバに対する攻撃としては，バッファオーバフロー攻撃のほかにフォーマット文字列攻撃，OS コマンド挿入攻撃，パス乗り越え攻撃といった様々な攻撃が知られているが，どのような攻撃手法であっても被害の範囲はサンドボックス内の資源に限定される．

2.4　セキュリティポリシ

サンドボックス内部でアクセス可能な資源を定めたものをセキュリティポリシと呼ぶ．たとえばシステムコールを監視する方式のサンドボックスでは，`open()` を許可するファイルや `connect()` を許可するマシンなど，システムコールの種類や引数によってアクセスの可否を定めたものである．セキュリティポリシは，通常はサーバプログラムの管理者などが，どの資源を守りたいかという要求に基づいて決定する．

セキュリティポリシを適切に記述することは，プログラムの安全性を確保するうえで重要な事柄である．サンドボックス内に必要以上の資源が存在すると，不正アクセスの被害が拡大してしまう危険性がある．一方，必要な資源がサンドボックス内に存在しないと，プログラムが正常に動作しなくなってしまう．したがって，セキュリティポリシを記述する際には，過不足なく適切にアクセス可能な資源を選択する必要がある．

たとえば Apache の場合，不必要なファイルなどへのアクセスは禁止しつつ，ライブラリなど必要なシステム資源や HTML ファイルの読み込みは許可するといったポリシを記述する必要がある．すなわち，抽象的な言語で記述すると，下記のようなポリシになる．

```
DEFAULT DENY
ALLOW ACCESS 必要なシステム資源
ALLOW READ-ONLY HTML ファイル
```

過不足のない適切なポリシを記述することにより，サーバプログラムを正常に動作させつ

© 2009 Information Processing Society of Japan

つ，乗っ取られた場合の被害を最小限に抑えることが可能になる．

2.5　ポリシ記述の困難さ

セキュリティポリシを適切に設定するためには，サーバプログラムが発行する大量のシステムコールに対して過不足なくポリシを記述しなければならない．たとえば，Web サーバの Apache が発行するシステムコールの種類は 66 種類，`open()` システムコールでアクセスするファイル数は 202 個に及ぶ．セキュリティポリシは個々のサーバプログラムごとに記述しなければならないため，複数のサーバを運用する場合には，ポリシの記述量はさらに膨大な量となる．

また，最小特権の原則に従って最低限必要なアクセスだけを許可するセキュリティポリシを記述するためには，システムに関する詳細かつ専門的な知識が必要となる．たとえば，UNIX 環境でサーバが発行するシステムコールには，`setuid()` や `getpgrp()` などのプロセス管理関係，`semop()` や `sendmsg()` などの IPC 関係，`sigaction()` などのシグナル関係など UNIX 固有のものが数多く存在し，UNIX に関する高度で詳細な知識が必要になる．`setuid()` は挙動がきわめて複雑なことが知られているほか[19]，公開されている API と実際のシステムコールの実装が異なる環境もあり，これらのシステムコールに関するポリシを漏れなく適切に記述することは困難である．

さらに，サーバプログラムが発行するシステムコールには，OS など実行環境に依存するものも多い．たとえば，Linux 環境では `set_thread_area()`，Solaris 環境では `door_call()` などシステム固有のシステムコールが発行される．また，OS が同じでも設定によって発行されるシステムコールが異なる場合がある．たとえば，`gethostbyname()` などのネームサービスでは，NIS や DNS，LDAP など使用するサービスの違いや，nscd（name service cache daemon）が有効かどうかによって，発行するシステムコールの種類が変化する．したがって，同じサーバプログラムであっても，実行環境ごとに個別にポリシを記述しなければならない．

このように，セキュリティポリシの記述にあたっては，記述量が多い，記述が難しい，実行環境に依存するなどの点で困難をともなう．ポリシの記述が難しいと，記述に膨大な労力をともなうほか，ポリシを正確に記述することが難しくなり，ポリシ自体にセキュリティホールが生じる可能性が高くなる．

3.　フェーズを用いたセキュリティポリシ

本章では，**実行時のフェーズ**を用いたセキュリティポリシ記述の簡略化について述べる．まずインターネットサーバにおけるフェーズの定義について述べ，次にフェーズ遷移のタイミングについて述べる．最後にポリシの記述量と安全性に対する考察を行う．なお，本論文ではインターネットからのリモート攻撃を対象としており，サーバの稼働しているマシンからのローカル攻撃は想定していない．

3.1　実行時のフェーズ

インターネットからのリモート攻撃を想定した場合，サーバがインターネットからの通信メッセージを読み取る前は，攻撃を受けていない「きれいな状態」にある．したがって，不正アクセスが行われる可能性はなく，サンドボックスによるアクセス制御は不要である．しかし，いったん通信メッセージを読み取った後は，サーバは「汚染された状態」（tainted）にある可能性があり，サンドボックスによるアクセス制御が必要である．

すなわち，インターネットサーバの実行状態は大きく 2 つのフェーズに分けることができる．メッセージを読み取る前の**初期化フェーズ**と，メッセージを読み取った後の**プロトコル処理フェーズ**である．5 章で述べる分析結果が示すように，初期化フェーズではライブラリのリンクや実行環境自体の初期化に用いる設定ファイルの読み込みなどを行うため，サーバの本来の処理からは想像しにくいファイルへのアクセスや，OS 固有のシステムコールを実行することが多い．一方，プロトコル処理フェーズではサーバ本来の処理だけが行われるため，アクセスするファイルや実行するシステムコールが比較的想定しやすい傾向にある．

そのため，従来はプロセス単位で記述していたセキュリティポリシを，フェーズ単位で記述できるようにすることにより，初期化フェーズでの複雑な処理を許可するためのポリシを記述する必要がなくなり，ポリシ記述の簡略化が期待される．一方，プロトコル処理フェーズではサンドボックスにより従来と同等の安全性を確保されるが，プロトコル処理フェーズのポリシはサーバ本来の性質から容易に予想されうるものが多く，記述は比較的容易になることが期待される．

3.2　フェーズの遷移

実行時のフェーズはプロセスごとに設定されており，必ず初期化フェーズかプロトコル処理フェーズのいずれかにある．起動時は初期化フェーズであり，特定のシステムコールの発行を契機としてフェーズの遷移が行われる．提案方式では，初期化フェーズからプロトコル処理フェーズに遷移するタイミングは，インターネットからのメッセージを受け取ったときである．メッセージを受け取る際には必ずシステムコールの発行が必要となるため，サンドボックスのシステムによってシステムコールを監視することにより，フェーズ遷移すべきタイミングを確実にとらえることができる．

© 2009 Information Processing Society of Japan

UNIX 系 OS では，インターネットからメッセージを受け取るためのシステムコールには，recv()，recvfrom()，recvmsg() などの recv() 系や read() がある．これらのシステムコールでは，読み込み先を指定するために，引数としてファイルディスクリプタを指定する．読み込み先がインターネットの場合には，ファイルディスクリプタとしてインターネットに接続されたソケットを指定する．接続済みのソケットは，インターネットからの接続要求を受け付ける場合には accept()，内部からインターネットに接続する場合には connect() によって取得する．

したがって，インターネットからメッセージを読み込むタイミングをとらえるためには，まず accept() や connect() システムコールを監視して，ソケットのファイルディスクリプタ番号と接続先の IP アドレスとの対応表を記録しておく．次に，recv() 系や read() システムコールが発行されたときに引数のファイルディスクリプタを調べて，インターネットに接続されたソケットだった場合には，対象のプロセスをプロトコル処理フェーズに移行させる．このような方式の実装は対象となるシステム依存とはなるが，(1) UNIX 系では処理の流れがほぼ共通していること，(2) 関連するシステムコールの種類はそれほど多くなく容易に網羅可能であること，などの点で実現はそれほど難しくない．

上記で述べたフェーズ遷移のタイミングは，サーバの種類にかかわらず同じ手法でとらえることができる．また，多くのサーバプログラムでは，初期化作業はインターネットからの接続を受け付ける前に済ませ，その後はクライアントからの要求を受け付けては結果を返すという作業を繰り返すという構造になっていることが多い．したがって，このような典型的なサーバの構造を持つ多くのサーバプログラムにおいては，もともと構造的に初期化フェーズとプロトコル処理フェーズを持っており，提案方式が有効に機能すると考えられる．

いったん初期化フェーズからプロトコル処理フェーズへ遷移したのちは，外部から読み込んだ通信メッセージに施された細工によって，サーバプログラムが乗っ取られるなど「汚染」された状態になっている可能性があるため，プロトコル処理フェーズから初期化フェーズに戻ることはできない．ただし exec() を実行したときは，メモリ上のデータがいったんすべて消去されるため，再び「きれいな」状態に戻ったと判断して，初期化フェーズに戻ることもできる．ただしこの場合，任意のプログラムの実行を防ぐために，exec() で実行可能なプログラムをポリシで制限しておく必要がある．また，実行する子プロセスも同様にサンドボックス内で動作させることが望ましい．

実装を簡略化するために，メッセージを読み込むシステムコールではなく，接続済みのソケットを取得するシステムコールのみを監視する方法がある．多くのプログラムでは，accept() でソケットを取得した直後に通信メッセージを読み出すため，実用上はほぼ同じタイミングをとらえられる．これにより，ファイルディスクリプタの状態を記憶する手間を省くことができる．ただし，この場合は exec() によって初期化フェーズに戻ることはできない．これはオープン済みのファイルディスクリプタに接続済みのソケットが含まれていた場合（かつ close-on-exec フラグが指定されていない場合），そのソケットからの読み出しを外部からのメッセージ読み出しと識別できないためである．

3.3 IPC によるフェーズの遷移

サーバプログラムの中には，安全性向上[20] や機能拡張[21] などのために，複数のプロセスが IPC（Inter-Process Communication）で連携してサーバの機能を実現するものがある．この場合，通常のサンドボックスシステムでは，それぞれのプログラムのプロセスをサンドボックス内に入れて，独立してポリシを記述することになる．この場合でも，それぞれのプロセス単位でフェーズ遷移を行うことにより，初期化フェーズにおける複雑なポリシ記述が不要になり，ポリシ記述の簡略化への効果が期待できる．ただし，この場合，インターネットからメッセージ受け取りだけでなく，プロトコル処理フェーズに移行したプロセスとの IPC をフェーズ遷移の契機とするかどうかが問題となる．この問題を以下の 3 つの場合で検証する．

まず第 1 に，サーバが独立した別のサーバと通信する場合である．たとえば，起動時に DNS サーバにアクセスする場合などが該当する．厳密には，これらのサーバもインターネット接続後は乗っ取られている可能性があるため，これらのサーバからメッセージを読み込んだ場合は，プロトコル処理フェーズに遷移すべきである．しかしこの場合，サーバ間の独立性が高いため，通信プロトコルが安全に設計されている場合には，実際には攻撃の難易度は高い．すなわち，攻撃者は別のサーバをいったん乗っ取った後，さらに該当するサーバに IPC 経由で攻撃するという 2 段階の攻撃が必要であるほか，2 つのサーバの脆弱性を同時に攻略する必要があり，攻撃が成功する可能性は比較的低い．したがって，安全性をそれほど低下させることなく，これらのサーバとの IPC を契機としたフェーズ遷移を回避してポリシ記述の簡略化の効果を高めることができる．

第 2 に，サーバ自体が複数のプログラムで構成されていて，IPC で通信を行いながらサーバの機能を実現する場合である．このようなプログラムには，openssh や qmail などがある．この場合，サーバ間の独立性が低いため，2 段階攻撃の技術的難易度がそれほど高くない可能性がある．たとえば，IPC でファイルのパスを指定して読み込むといったインタフェースであった場合，1 つのプロセスを乗っ取るだけで，IPC 経由で任意のファイルの読み書き

© 2009 Information Processing Society of Japan

ができてしまう可能性がある．このような場合には，IPC でプロトコル処理フェーズにあるプロセスからメッセージを受け取った時点で，該当プロセスもプロトコル処理フェーズに遷移するべきである．だたしこの場合，複数のプログラムのいずれかがすでにインターネットからメッセージを受け取った状態にあるため，サーバ全体の各プロセスもライブラリや設定ファイルの読み込みなど初期化フェーズで行うべき処理はすでに完了している状態にあると考えられる．したがって，ポリシ記述の簡略化への効果は依然として高いと期待される．

第 3 に，プロセス間に親子関係がある場合である．これには，Web サーバからの CGI 呼び出しや，inetd からのサーバ実行などがある．この場合，子プロセスが親プロセスから引数や環境変数といった「メッセージ」を受け取る場合がある．この場合，これらのメッセージによって子プロセスの制御が乗っ取られる可能性がある場合は，`exec()` を契機として子プロセスを初期化フェーズに戻すことは危険である．一方，サンドボックスで `exec()` の引数をチェックするなどにより，引数や環境変数による攻撃がないと想定できる場合は，子プロセスは初期化フェーズに戻って初期化処理を行い，親プロセスとの IPC やインターネットからのメッセージを契機にプロトコル処理フェーズに移行することで，ポリシ記述の簡略化が期待できる．

このように，IPC によるフェーズ遷移を行わない場合の安全性への影響は，各プロセス間の独立性およびインタフェースの設計に依存する．したがって，ポリシをを記述する際には，安全性と記述性のバランスを考慮することになる．なお，IPC によりフェーズ遷移を行った場合でも，従来と同程度の安全性は確保できるほか，少なくとも `main()` 関数が呼ばれる前に libc などによって行われる初期化処理のポリシ記述は不要になるため，ポリシ記述の簡略化には依然として一定の効果が見込める．

3.4　ポリシ記述量と安全性

本方式において，プログラム全体，初期化フェーズ，プロトコル処理フェーズで発行するシステムコールの種類には，$S_a = S_i \cup S_p$ の関係がある．ただし S_a，S_i，S_p は，それぞれプログラム全体，初期化フェーズ，プロトコル処理フェーズで発行するシステムコールの集合である．従来のサンドボックスでは，S_a に対してポリシを記述する必要があるのに対し，本方式では S_p に対してのみポリシ記述をすればよく，S_i に対するポリシ記述が不要である．一般に $|S_p| \ll |S_i|$ であることが多いため，ポリシ記述量の削減が期待できる．一方，$S_p \subseteq S_a$ なので，本方式によって従来のサンドボックスよりプロトコル処理フェーズにおける安全性が低下することはない．初期化フェーズでは，従来のサンドボックスよりアクセスできる資源が増えるが，初期化フェーズでは不正アクセスの危険性がないため，安全性には問題がない．

なお，本方式を拡張してプロトコル処理フェーズを複数持たせる方式も考えられる．たとえばネットワークの接続先に応じてフェーズを変えたり，ユーザ認証を行った後にユーザごとに定義したフェーズに遷移したりするなどにより，従来よりも厳密な保護を実現することが考えられる．しかしこの場合，多数のフェーズに対してポリシを記述することになり，ポリシ記述の簡略化という観点ではあまり効果が見込めない．たとえば，プロトコル処理フェーズ i で発行するシステムコールを Sp_i とすると，$S_p = \bigcup_{i=1}^{n} Sp_i$ となり，ポリシ全体の記述量は変わらないかむしろ増える．

本手法では，あくまでもポリシ記述の簡略化を目的としており，保護のレベルとしては，従来のフェーズの概念を持たないサンドボックスで実現可能な保護と同程度のものを実現することを想定している．ただし，本手法の存在を前提として，特権分割の手法を用いてサーバプログラムを適切に分割することにより，従来よりも厳格な保護を簡単に記述できるようになる可能性はある．特権分割とフェーズ切替えを組み合わせた場合のポリシ記述の簡略化に対する効果の検証については，今後の課題である．

従来のポリシ記述と比べると，提案方式ではサーバプログラムをフェーズに分割するという新たな作業が加わっている．しかし，本方式ではフェーズの種類を 2 種類に限定しており，フェーズが切り替わるタイミングも特定のシステムコールを発行した時点に限られているため，フェーズの分割作業は比較的容易に行うことができる．たとえば，システムコールのトレースからポリシを記述する場合は，3.2 節で示した特定のシステムコールの発行パターンを分析することにより，プロトコル処理フェーズに遷移するタイミングを機械的に抽出することができる．また，ソースコードを解析してポリシを記述する場合にも，上記のシステムコールを発行している個所を検索することにより，プロトコル処理フェーズに遷移するポイントは容易に見つけることができる．これにより，プロトコル処理フェーズでのみ適用すべきポリシの抽出は比較的容易に行える．フェーズの種類や遷移のタイミングを限定しても，5 章の分析結果が示すように，ポリシ記述の簡略化（記述量の削減，難易度の低下，汎用性の向上）に対しては十分効果が見込める．

4.　フェーズに基づくサンドボックスの実装

本章では，提案した手法に基づいて我々が実装したサンドボックスシステムについて説明する．本実装は，サンドボックスの一種である LIDS（Linux Intrusion Detection System）[15] をベースとして使用した．LIDS は Linux のカーネルパッチとして実装されているセキュリ

© 2009 Information Processing Society of Japan

ティ機構であり，サンドボックスの機能を含んでいる．今回の実装では，ファイルに対するアクセス制御の部分のみを用いてサンドボックスとして利用した．

LIDS のセキュリティポリシはリスト形式で管理されている．リストの各エントリでは，サブジェクト（アクセスの主体）のプログラムのファイル名，オブジェクト（アクセスの対象）のファイル名，アクセス権の 3 つを指定する．エントリの追加は lidsconf と呼ばれる専用のコマンドによって行う．以下に LIDS におけるポリシ記述の例を示す．

```
lidsconf -A -s /usr/sbin/httpd -o /var/log/httpd  -j APPEND
```

この例では，サーバ（/usr/sbin/httpd）がログを格納するディレクトリ（/var/log/httpd）に対して，ファイルを追記する（APPEND）ことを許可するポリシを指定している．“-s” は主体となるサブジェクト，“-o” は対象となるオブジェクト，“-j” はアクセス権を表しており，“-A” はこのポリシを追加することを意味する．対象となるオブジェクトがディレクトリの場合は，そのディレクトリ以下のすべてのファイルに対して適用される．

フェーズに基づくポリシ記述を実現するために，LIDS に対して以下の 3 つの機能を追加した．第 1 の機能は，各ポリシに対して初期化フェーズとプロトコル処理フェーズのどちらで適用されるのかを指定するフラグを追加する機能である．これは LIDS のセキュリティポリシを格納するファイルのフォーマットを拡張してフラグ領域を追加し，lidsconf にオプションを追加することで実現した．具体的には，以下のようにして適用されるフェーズを指定できる．

```
lidsconf -A -p -s /usr/sbin/httpd -o /var/log/httpd  -j APPEND
```

この例では，新たに “-p” オプションが追加されている．“-p” オプションを指定したポリシはプロトコル処理フェーズでのみ適用される．“-p” オプションがない場合は，初期化フェーズでのみ適用される．

LIDS に追加した第 2 の機能は，フェーズに基づいて適用されるポリシを切り替える機能である．まず，現在のフェーズを識別するために，プロセス構造体に専用の領域を追加して，現在のフェーズを示すフラグを格納した．また，アクセス制御を行う際に，ポリシに付随するフラグと現在のフェーズを比較して，ポリシの適用の有無を判断する処理を追加した．

第 3 の機能は，システムコールの発行を契機としてフェーズを遷移する機能である．現在の実装では，3.2 節で述べた 2 つの方式のうち，accept() を契機として遷移する方式を実装している．この方式では，3.2 節で述べたように exec() を契機とした初期化フェーズへのフェーズ遷移はできないため，3.3 節で述べたようにプロトコル処理フェーズで子プロセスを起動する場合には子プロセスのポリシ記述を簡略化することは難しくなる．次章の評価で用いたサーバにはこのようなプログラムはなかったが，Web サーバから CGI を実行した場合には recv() 系や read() を契機としてフェーズ遷移する実装が必要であり，その場合の評価は今後の課題である．

5. 評　　価

本章では，フェーズを用いたセキュリティポリシ記述方式の有用性を評価する実験の結果を示す．まず，ポリシ記述の簡略化に対する効果を定量的に評価するために，HTTP，SMTP，POP のそれぞれのサーバプログラムのシステムコールのトレース結果を分析し，フェーズを用いた場合と用いない場合のポリシ記述に関して，記述量，記述の難易度，システムへの依存度などの観点から比較を行う．また実際のポリシ記述に対する効果を評価するために，LIDS をベースにしたサンドボックスの実装を用いて実際にポリシの記述を行い，簡略化に対する効果を検証する．

5.1　システムコールの分析

本節では，ポリシ記述の簡略化に対する効果を検証するために，サーバプログラムが発行するシステムコールの分析を行う．システムコールトレースの取得には strace コマンドを使用し，サーバの起動時から終了時まで子プロセスも含めて取得した．トレース内容を分析して，プログラム全体とプロトコル処理フェーズで発行したシステムコールをそれぞれ抽出し，その内容を分析した．サーバ起動中に行った処理は，HTTP では HTML ファイルの取得，SMTP ではメール送信，POP ではメール受信である．

今回の実験では，サーバの処理は最も単純なものを用いている．しかし初期化フェーズでの処理内容はサーバの処理にかかわらず一定であり，発行されるシステムコールの内容は通常の環境とあまり変わらないと考えられる．一方，プロトコル処理フェーズで発行されるシステムコールの内容は，サーバの処理内容に応じて変化する．本実験では，最も単純なケースにおけるシステムコールの内容を比較することで，ポリシ記述の簡略化の効果を浮き彫りにするほか，簡略化に対する効果の最大値を見積もるものである．

サーバプログラムには，Fedora Core 4 標準のものを使用した．HTTP サーバは Apache 2.0.54（httpd），SMTP サーバは sendmail 8.13.4，POP サーバは dovecot 0.99.14 である．OS は Linux 2.6.11 である．

5.1.1　システムコールの種類

サーバプログラムが発行したシステムコールの種類を分析した結果を**表 1** に示す．いずれのサーバにおいても，プログラム全体と比べて，プロトコル処理フェーズで発行するシステ

© 2009 Information Processing Society of Japan

表 1　サーバプログラムが発行したシステムコール
Table 1　System calls issued from server programs.

	Apache		sendmail		dovecot	
	全体	HTTP 処理フェーズ	全体	SMTP 処理フェーズ	全体	POP 処理フェーズ
システムコールの種類	66	13（19.7%）	67	40（59.7%）	79	42（53.2%）
オープンするファイルの種類	202	1（0.5%）	61	13（21.3%）	64	22（34.4%）
/usr	101	0（0.0%）	24	0（0.0%）	11	1（9.1%）
/lib	0	0（0.0%）	0	0（0.0%）	0	0（0.0%）
/etc	79	0（0.0%）	16	5（31.2%）	13	8（61.5%）
/var	1	1（100.0%）	2	1（50.0%）	4	0（0.0%）
/proc	2	0（0.0%）	2	2（100.0%）	2	1（50.0%）
/dev	2	0（0.0%）	1	1（100.0%）	2	0（0.0%）
非 POSIX システムコール	13	0（0.0%）	14	6（42.9%）	17	5（29.4%）

ムコールの種類は大幅に少ない．たとえば Apache では，プロトコル処理フェーズではプログラム全体の 19.7%であった．同様に sendmail では 59.7%，dovecot では 53.2%であった．

したがって，初期化フェーズのポリシ記述を省略して，プロトコル処理フェーズにおけるポリシのみを記述することにより，ポリシ記述量の削減が期待できる．

5.1.2　オープンするファイルの種類

open() システムコールでオープンするファイルの種類を分析した結果を表 1 に示す．いずれのサーバも，プログラム全体と比べてプロトコル処理フェーズでアクセスするファイルの種類は大幅に少ない．たとえば Apache では，プログラム全体では 202 種類なのに対し，プロトコル処理フェーズでは 1 種類である．sendmail では全体の 21.3%，dovecot でも 34.4%である．

一般にプログラムの起動時には，多数の共有ライブラリファイル（/usr，/lib 以下）や設定ファイル（/etc 以下）が読み込まれる．たとえば Apache では，起動時に perl や python などのモジュールファイルを多数読み込む．一方プロトコル処理フェーズでは，サーバの本来の機能に関係するファイルが主にアクセスされるため，ファイルの種類が少なくなる．

サーバの処理内容によってアクセスするファイルの種類は増加するが，プロトコル処理フェーズではサーバ本来の処理に密接に関係したファイルへのアクセスが多く，アクセスの必要性などが理解しやすいため，初期化フェーズでアクセスされるファイルに比べるとポリシ記述は容易である．また，共有ライブラリやシステムの設定など，実行環境に依存したファイルに関するポリシ記述の多くを省略できるので，ポリシ記述の難易度の低下や汎用性の向上が期待できる．

5.1.3　非 POSIX システムコールの数

サーバプログラムが発行したシステムコールの中で，POSIX で定義されていないシステムコールの数を分析した．**表 2** に結果を示す．表中の‘-’はプログラム全体で発行された非 POSIX システムコールを表し，‘*’はプロトコル処理フェーズでも発行されたものを表す．

分析結果から，プロトコル処理フェーズでは非 POSIX システムコールが少ないことが分かる．たとえば Apache では，プロトコル処理フェーズで発行する非 POSIX システムコールは 0 である．sendmail では 6 種類（42.9%），dovecot では 5 種類（29.4%）であり，いずれもプログラム全体と比べて少ない．

特にプロトコル処理フェーズでは，Linux 固有のシステムコール（set_thread_area() など）が発行されないことが多いほか，非 POSIX システムコールでも POSIX に存在するシステムコールの拡張版が多く，動作を理解するのが比較的容易なものが多い．したがって，ポリシ記述が容易になることが期待できるほか，異なる OS で動作させた場合などでもポリシの変更が少なくてすむなど汎用性が向上することが期待できる．

5.2　LIDS におけるポリシ記述の簡略化

フェーズを用いたポリシ記述方式が，ポリシ記述の簡略化に有効であることを示すために，4 章で示した LIDS をベースとした実装を用いて，実際のポリシ記述を行った．実装および実験は Fedora 9 上で行った．フェーズに基づくポリシ切替え機構は，Linux カーネルに対して LIDS 2.2.3rc5 のパッチを適用したものをベースとして実装したものを用いた．HTTP サーバは apache 2.2.9，SMTP サーバは sendmail 8.14.2.4.fc9，POP サーバは dovecot 1.0.13 である．記述したポリシは，2.3 節および 2.4 節で述べたような適切なポ

© 2009 Information Processing Society of Japan

表 2　非 POSIX システムコール
Table 2　Non-POSIX system calls.

システムコール名	Apache	sendmail	dovecot
`_llseek()`	-		*
`_sysctl()`	-	-	-
`brk()`	-	-	-
`chroot()`			-
`clone()`	-	*	*
`exit_group()`	-	*	*
`futex()`	-	-	*
`getdents()`	-	-	-
`madvise()`			-
`rt_sigaction()`	-	*	*
`rt_sigprocmask()`	-	*	-
`sendfile()`			-
`set_thread_area()`	-	-	-
`set_tid_address()`	-	-	-
`setgroups()`	-	-	-
`setresuid()`			-
`sigreturn()`	-	*	-
`statfs()`		-	
`wait4()`		*	
計	0/13	6/14	5/17

'-'：プログラム全体で発行したもの
'*'：プロトコル処理フェーズでも発行したもの

リシになるように，ライブラリや設定ファイル，HTML ファイルの読み込みとログファイルの書き込みという必要最小限のアクセスのみを許可したものである．また，実際にサーバを動作させて正しく動作することを確認している．性能に対するオーバヘッドもほとんどないことを確認している．

表 3 に実際に記述したポリシの行数を示す．表 3 では，フェーズを用いたポリシ記述方式，従来方式（省略版），従来方式（厳密版）の 3 つの方式での記述結果を示している．従来方式（省略版）は，ディレクトリ単位でアクセス権を指定することで，一部のポリシ記述を省略したものである．従来方式（厳密版）は，最小特権の原則を厳密に厳密に適用して，個々のファイル単位でポリシを記述したものである．

実験結果から，実際にポリシの記述量が削減されていることが分かる．たとえば，apache では，従来方式（厳密版）では 199 行，従来方式（省略版）でも 17 行であったのに対し，フェーズを用いたポリシ記述方式では 3 行であった．同様に sendmail および dovecot でも大幅に記述量を削減できていることが分かる．

表 3　ポリシ記述量の比較
Table 3　Comparisons of policy description amount.

方式	apache	sendmail	dovecot
フェーズを用いた方式	3 行	9 行	3 行
従来方式（省略版）	17 行	18 行	22 行
従来方式（厳密版）	199 行	70 行	68 行

表 4　Apache のポリシ（従来方式省略版）
Table 4　Policies for apache.

```
lidsconf -A -o /usr/sbin/httpd -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /lib -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /usr/lib -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/httpd/conf -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/httpd/modules -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/httpd/conf.d -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/ld.so.cache -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/nsswitch.conf -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/passwd -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/group -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/resolv.conf -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/hosts -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/host.conf -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /etc/selinux/config -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /var/www/html -j READONLY
lidsconf -A -s /usr/sbin/httpd -o /var/log/httpd -j WRITE
lidsconf -A -s /usr/sbin/httpd -o /var/run/ -j WRITE
```

表 4 に従来方式（省略版）による Apache に対するポリシ記述例を示す．このポリシでは特に`/etc` 以下のファイルに対して詳細なポリシ記述が必要となっている．この中には，Linux 固有のファイルや UNIX に精通しなければ必要性を判断できないファイルが含まれており，ポリシ記述が難しい一因となっている．また，このポリシでは`/lib` や`/usr/lib` の記述を省略しているほか，LIDS の仕様上の都合で`/proc` や`/dev` はデフォルトで許可されており，厳密には最小特権の原則を満たしていない．従来方式（厳密版）によるポリシ記述例では，行数，難易度，依存性の点でさらに複雑になっている．

表 5 にフェーズを用いた方式でのポリシ記述例を示す．1 行目は初期化フェーズで適用されるポリシであるが，これは LIDS の仕様上の都合で，サーバプログラムに対してサンド

© 2009 Information Processing Society of Japan

表 5　Apache のポリシ（フェーズを用いた方式）
Table 5　Policies for apache.

```
lidsconf -A -o /usr/sbin/httpd -j READONLY
lidsconf -A -p -s /usr/sbin/httpd -o /var/www/html -j READONLY
lidsconf -A -p -s /usr/sbin/httpd -o /var/log/httpd -j WRITE
```

ボックスの適用を開始することを指定するためのものである．残りの 2 行は，HTML ファイルが格納されたフォルダへの読み込みアクセス，ログが格納されるディレクトリへの書き込みアクセスを許可している．このように，apache では全体で 3 行でポリシ記述を行うことができた．また，これらのアクセスは HTTP サーバの性質から容易に予測される挙動であるほか，OS 固有のファイルも含まれていない．したがって，ポリシの記述量，記述の難易度，システムへの依存度の観点から，ポリシ記述の簡略化が実現されているといえる．

6. 関連研究

セキュリティポリシを実行時に動的に切り替える機構としては，阿部らによる関数呼び出しのタイミングでポリシを切り替えられるシステムがある[11]．このシステムでは，デバッグ用のトラップ命令を用いて関数呼び出しを捕捉し，特定の関数が実行中の間だけアクセス権を追加するといったポリシを記述できる．塩谷ら[12] は，システムコールの発行時にプロセスが実行している関数を調べて，動的にポリシを切り替えられるサンドボックスを提案している．保理江ら[13] は，攻撃を受けた際のリスク低減を目的として，SELinux とカーネルベースの IDS を組み合わせて，侵入を検知した時点でより制限の厳しいアクセスポリシに動的に切り替える方式を提案している．また，SubDomain[14] では，子プロセス呼び出し時や `change_hat()` という独自システムコールの呼び出し時にセキュリティポリシを切り替えられる．

これらのシステムでは，システムコール監視のみで実現したサンドボックスよりも制約の厳しいポリシを実現するために動的切替えを用いており，適切な保護を実現するためには，従来以上にポリシ記述が難しくなる可能性がある．本論文の提案方式は，従来のサンドボックスと同程度のポリシをより簡単に記述するために動的切替えを利用している．また，これらのシステムでは，ポリシ切替えのタイミングが，関数呼び出し，システムコール発行時に実行中の関数の種類，侵入検知，独自システムコールの発行など，切替えのタイミングが固定されていないのに対し，本研究では対象をインターネットサーバに限定することで，インターネットからメッセージを読み込んだ時点という明示的なタイミングを定義して，ポリシ記述の簡略化を実現している．ただし，本論文の提案方式は特定の実装に依存した手法ではないため，上記のような動的ポリシ切替え機構を持つサンドボックスに対して提案方式を適用することにより，ポリシ記述の簡略化に利用できる可能性はある．

セキュリティポリシの動的切替えをより一般化した概念として，オートマトンを用いた状態遷移に基づくセキュリティモデルがある[22),23]．また，Chinese Wall[24] などの過去のアクセス履歴に基づいてアクセス権が変化するセキュリティモデルもある．これらの研究は，情報の流れに着目するという点で本研究と似ているが，その目的は主に情報漏洩の防止にあり，ポリシ記述の簡略化の枠組みは提供していない．本論文では，インターネットサーバの特性に着目して，フェーズを用いたセキュリティポリシの動的切替えにより，ポリシ記述を簡略化する仕組みを提案している．

MAPbox[3] では，セキュリティポリシ記述の手間を軽減するために，アプリケーションの挙動に基づいて “compiler” や “netclient” などにクラス分けしたポリシの雛形を用意している．MAPbox で適用されるポリシはプログラム全体の実行を通して同一であり，動的には切り替わらない．したがって，初期化フェーズでアクセスする資源に対するポリシ記述も必要となる．

ポリシの記述を支援する仕組みとしては，プログラムを実行させながら徐々にアクセス権を追加していく手法[9),10] が提案されている．また，GUI などによって視覚的にポリシを設定する手法[11] も提案されている．これらの手法では，最終的に生成されるポリシの量は同じであり，初期化フェーズでアクセスする資源に対するポリシ記述が必要となる．

chroot では，ルートディレクトリを変更することで，ファイルアクセスを制限することができる．chroot で動的にポリシを切り替えるにはプログラムの修正が必要になるほか，root 権限の適切な管理が必要である．

7. まとめ

本論文では，インターネットサーバ向けのサンドボックスにおけるセキュリティポリシ記述を簡略化する手法として，実行時のフェーズを用いたポリシ記述について述べた．フェーズを用いることでサーバ本来の処理に関するポリシのみを記述すればよくなり，ポリシ記述量の削減，記述の難易度の低下，汎用性の向上が実現できる．実際に提案方式を HTTP，SMTP，POP の各サーバに適用した結果，実際にポリシの記述を簡略化できることを確認した．

今後の課題としては，様々なサーバプログラムにおける分析やクライアントプログラムへ

© 2009 Information Processing Society of Japan

の応用，SecureOS などサンドボックス以外のシステムへの応用，特権分割と組み合わせてさらなるセキュリティ向上を実現するシステムへの応用などがあげられる．

参 考 文 献

1) 大山恵弘：ネイティブコードのためのサンドボックスの技術，コンピュータソフトウェア，Vol.20, No.4, pp.55–72 (2003).
2) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A.: A Secure Environment for Untrusted Helper Applications, *Proc. 6th USENIX Security Symposium* (1996).
3) Acharya, A. and Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications, *Proc. 9th USENIX Security Symposium* (2000).
4) Stein, L.D.: SBOX: Put CGI Scripts in a Box, *Proc. 1999 USENIX Annual Technical Conference* (1999).
5) Bernaschi, M., Gabrielli, E. and Mancini, L.V.: Operating System Enhancements to Prevent the Misuse of System Calls, *Proc. 7th ACM Conference on Computer and Communications Security*, pp.174–183 (2000).
6) Shinagawa, T., Kono, K. and Masuda, T.: Flexible and Efficient Sandboxing Based on Fine-Grained Protection Domains, *Proc. International Symposium on Software Security* (2002). LNCS 2609, pp.172–184, Springer-Verlag (Feb. 2003).
7) Jaeger, T., Sailer, R. and Zhang, X.: Analyzing Integrity Protection in the SELinux Example Policy, *Proc. 12th USENIX Security Symposium*, pp.59–74 (2003).
8) Zanin, G. and Mancini, L.V.: Towards a formal model for security policies specification and validation in the selinux system, *Proc. 9th ACM symposium on Access control models and technologies*, pp.136–145 (2004).
9) Peterson, D.S., Bishop, M. and Pandey, R.: A Flexible Containment Mechanism for Executing Untrusted Code, *Proc. 11th USENIX Security Symposium*, pp.207–225 (2002).
10) 大山恵弘，加藤和彦：SecurePot: システムコールフックを利用した安全なソフトウェア実行系，日本ソフトウェア科学会第 18 回大会 (2001).
11) 阿部洋丈，加藤和彦，王　維：セキュリティポリシーの動的切替機構を持つリファレンスモニタシステム，コンピュータソフトウェア，Vol.20, No.3, pp.2–16 (2003).
12) 塩谷知宏，大山恵弘，岩崎英哉：実行コンテキストに応じたポリシー指定が可能なサンドボックス，情報処理学会研究報告，2007-OS-105, pp.87–93 (2007).
13) 保理江高志，原田季栄，田中一男：Linux カーネルの動的アクセスポリシー制御，*Linux Conference 2004* (2004).
14) Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain: Parsimonious Server Security, *Proc. 14th Systems Administration Conference*, pp.355–367 (2000).
15) Huagang, X., Biondi, P., et al.: LIDS (Linux Intrusion Detection System). http://www.lids.org/
16) Saltzer, J.H. and Schroeder, M.D.: The Protection of Information in Computer Systems, *Proc. IEEE*, Vol.63, No.9, pp.1278–1308 (1975).
17) Java Team, Gosling, J., Joy, B. and Steele, G.: *The Java[tm] Language Specification*, Addison Wesley Longman (1996). ISBN 0-201-6345-1.
18) CERT/CC: CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability (2002).
19) Chen, H., Wagner, D. and Dean, D.: Setuid Demystified, *Proc. 11th USENIX Security Symposium* (2002).
20) Provos, N., Friedl, M. and Honeyman, P.: Preventing Privilege Escalation, *Proc. 11th USENIX Security Symposium*, pp.231–242 (2002).
21) Robinson, D. and Robinson, D.: The Common Gateway Interface (CGI) Version 1.1 (2004). RFC 3875.
22) Erlingsson, U. and Schneider, F.B.: SASI enforcement of security policies: A retrospective, *Proc. 1999 Workshop on New Security Paradigms* (*NSPW'99*), pp.87–95 (1999).
23) Mehta, N.V. and Sollins, K.R.: Expanding and Extending the Security Features of Java, *Proc. 7th USENIX Security Symposium*, pp.159–172 (1998).
24) Brewer, D. and Nash, M.: The Chinese Wall Security Policy, *Proc. IEEE Symposium on Security and Privacy*, pp.206–214 (1989).

(平成 20 年 10 月 3 日受付)
(平成 21 年 2 月 16 日採録)

品川　高廣（正会員）

1974 年生．2003 年東京大学大学院理学系研究科情報科学専攻博士課程修了，博士（理学）取得，東京農工大学助手就任．2007 年より筑波大学大学院システム情報工学研究科講師．オペレーティングシステムや仮想マシンモニタ等のシステムソフトウェア，システムソフトウェアによるセキュリティに興味を持つ．平成 11 年度情報処理学会論文賞，平成 14 年度山下記念研究賞受賞．ACM，IEEE/CS，USENIX，日本ソフトウェア科学会各会員．

© 2009 Information Processing Society of Japan

忠鉢　洋輔（学生会員）

2009 年筑波大学第三学群情報学類卒業．現在，筑波大学大学院システム情報工学科博士前期課程．

河野　健二（正会員）

1993 年東京大学理学部情報科学科卒業．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．現在，慶應義塾大学理学部情報工学科准教授．博士（理学）．平成 11 年度情報処理学会論文賞受賞．平成 12 年度山下記念研究賞受賞．オペレーティングシステム，システムソフトウェア，コンピュータセキュリティに興味を持つ．IEEE/CS, ACM，USENIX 各会員．

加藤　和彦（正会員）

1962 年生まれ．1985 年筑波大学第三学群情報学類卒業．1989 年東京大学大学院理学系研究科情報科学専攻中退．1992 年博士（理学）（東京大学大学院理学系研究科）．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，2004 年筑波大学大学院システム情報工学研究科教授，現在に至る．オペレーティングシステム，分散システム，仮想計算環境，セキュリティに興味を持つ．1990 年情報処理学会学術奨励賞，1992 年同研究賞，2005 年同論文賞，2004 年日本ソフトウェア科学会論文賞各受賞．

© 2009 Information Processing Society of Japan