Software Engineering Institute

JPCERT CC®

セキュアデザインパターン

Chad Dougherty
Kirk Sayre
Robert C. Seacord
David Svoboda
Kazuya Togashi（富樫 一哉）

2009 年 3 月

TECHNICAL REPORT
CMU/SEI-2009-TR-010
ESC-TR-2009-010

CERT Program
Unlimited distribution subject to the copyright.

http://www.cert.org/

Carnegie Mellon

JPCERT CC®

Notifications and Disclaimers.

The following notices and disclaimers must be contained on the unofficial SEI-sanctioned translation in both English and Japanese:

- *This translation of Carnegie Mellon University copyrighted material is not an official SEI-sanctioned translation.*
- *This non-SEI-sanctioned translation of "Secure Design Patterns," CMU/SEI-2009-TR-010, ESC-TR-2009-010, Copyright 2009 by Carnegie Mellon University was prepared by JPCERT/CC with special permission from the Software Engineering Institute.*
- *Neither Carnegie Mellon University nor the Software Engineering Institute directly or indirectly endorse this non-SEI-sanctioned translation. Accuracy and interpretation of this translation are the responsibility of JPCERT/CC. The SEI has not participated in this translation.*
- *CERT is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.*
- *Copyright 2009 Carnegie Mellon University.*

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Copyright ©2009 JPCERT/CC All rights reserved.

JPCERT CC®

目次

謝辞 **iv**

要約 **v**

1 序文 **1**

1.1 セキュアデザインパターンについて 1

1.2 本書の目的 2

1.3 対象範囲 3

1.4 形式と規則 **4**

2 アーキテクチャレベルのパターン **5**

2.1 Distrustful Decomposition（信頼なしの分解） 5

2.2 PrivSep（Privilege Separation：特権の分離） 8

2.3 Defer to Kernel（カーネルに委任） 16

3 設計レベルのパターン **25**

3.1 Secure State Machine（セキュアステートマシン） 25

3.2 Secure Visitor（セキュアビジター） 36

4 実装レベルのパターン **42**

4.1 Secure Directory（セキュアディレクトリ） 42

4.2 Pathname Canonicalization（パス名正規化） 46

4.3 Input Validation（入力検査） 48

4.4 Resource Acquisition Is Initialization（RAII：リソースの確保は初期化時に行う） 54

5 結論と今後の作業 **59**

5.1 結論 59

5.2 今後の作業 59

参考文献 **60**

Copyright ©2009 JPCERT/CC All rights reserved.

図の目次

図 1	Distrustful Decomposition デザインパターンの概略構造	6
図 2	qmail メールシステムの構造	7
図 3	脆弱な `ftpd` プログラム	9
図 4:	OpenSSH における PrivSep の実装	10
図 5:	Defer to Kernel パターンの概略構造	17
図 6:	Defer to Kernel パターンの構造例	18
図 7:	Secure State Machine パターンの構造	26
図 8:	Secure State Machine サンプルコードのコラボレーション図	28
図 9:	Secure Visitor パターンの構造	37
図 10:	Secure Visitor サンプルコードのコラボレーション図	39
図 11:	Input Validation パターンの構造	49

Copyright ©2009 JPCERT/CC All rights reserved.

表の目次

表 1　　パターンの要素　　4

Copyright ©2009 JPCERT/CC All rights reserved.

謝辞

後援者である JPCERT/CC に感謝する。本報告書の制作を可能にした SEI の編集者である Pamela Curtis と Amanda Parente、ならびに伊藤友里恵氏と Bob Rosenstein に感謝の意を表する。

Copyright ©2009 JPCERT/CC All rights reserved.

要約

システムのデプロイメント後に脆弱性を修正するコストおよび脆弱性に関連するリスクは、開発者とエンドユーザの双方にとって高いものとなる。ソフトウェアのセキュリティ上の脆弱性に対応するためのベストプラクティスは数多く存在するが、それらは実装ごとに固有であり、多くの場合、異なる状況下での再利用が困難である。さらに、セキュリティ上の欠陥の根本原因に対する理解が深まるにつれ、実装とデプロイメントのフェーズに限らず、ソフトウェア開発ライフサイクルのすべてのフェーズにわたってセキュリティを考慮することの重要性に対する理解が深まってきている。本報告書では、さまざまな状況で適用が可能な、セキュリティ問題に対する一般的な解決策を示した記述もしくはテンプレートである、一連の「*セキュアデザインパターン*」について解説する。本報告書で詳説するセキュアデザインパターンは、具体的なセキュリティ機能の実装を中心としたものではなく、コードへの脆弱性の偶発的な挿入を排除すること、また脆弱性に起因する結果を緩和することを意図している。これらのパターンは、既存のセキュリティ設計のベストプラクティスを一般化することと、既存のデザインパターンにセキュリティ固有の機能性を加えて拡張することで導き出された。パターンは、その抽象化のレベルに応じて、アーキテクチャレベル、設計レベル、実装レベルの 3 つに分類される。

Copyright ©2009 JPCERT/CC All rights reserved.

1 序文

1.1 セキュアデザインパターンについて

パターンとは、ソフトウェア開発において、多くの場合に直面する共通的な問題に対する一般化された再利用可能な解決策である。デザインパターンは、コードに直接変換が可能な完成された設計ではないことに注意されたい。デザインパターンは、設計上の問題を解決する方法の解説あるいはテンプレートであり、様々な場面で利用することができる。アルゴリズムは、設計上の問題ではなく計算上の問題を解決するものであり、デザインパターンではない。

セキュアデザインパターンの目的は、コードへの脆弱性の偶発的な挿入を排除すること、またこれらの脆弱性に起因する結果を緩和することである。いわゆる「四人組（Gang of Four）」による[Gamma 1995] によって広まった設計レベルのパターンとは対照的に、セキュアデザインパターンは、システムの上位レベルの設計にかかわるアーキテクチャレベルのパターンから、関数やメソッドの部品を実装する方法のガイダンスを提供する実装レベルのパターンまで、特徴の異なる幅広いレベルのセキュリティ問題に対応する。

1.1.1 パターンの歴史

1977/79 ― 建築家 Christopher Alexander が建物や街の設計に関してデザインパターンの概念を紹介 [Alexander 1977]。

1987 ― Beck と Cunningham がプログラミングにパターンを適用する実験を行い、OOPSLA で発表 [Beck 1987]。

1994/95 ― 「四人組（Gang of Four：GoF）」（Erich Gamma、Richard Helm、Ralph Johnson および John M. Vlissides）が、オブジェクト指向プログラミング言語を対象とした多数の設計レベルのパターンを含んだ書籍を発行 [Gamma 1995]。

1997 ― Yoder と Baraclow がいくつかのセキュリティパターンの概要を示した論文を発行 [Yoder 1997]。

1.1.2 参考文献

繰り返し利用可能なセキュアなプラクティス（セキュリティパターン）については、すでに数多くの研究がなされている。本項では、この分野に大きく貢献した文献を示すとともに、それぞれについて簡単に説明する。

- *Security Design Patterns, Part 1* [Romanosky 2001] ： この報告書で取り上げられたパターンは、抽象度の高いレベルでセキュリティ問題を記述している。信頼できない第三者のシステムとの通信をどのように扱うか、および複数階層にわたるセキュリティ対策の重要性などに加えて、ホワイトハット侵入テストの使用や、システム開発およびシステム構成検討プロセスの早い段階での単純かつ影響の大きいセキュリティ上の課題への取り組みなどについて言及している。
- *Core Security Patterns Book* [Steel 2005] ： J2SE、J2EE、J2ME、および Java Card の各プラットフォームのアプリケーションを対象としたセキュリティパターンを中心に取り扱っ

Copyright ©2009 JPCERT/CC All rights reserved.

ており、主に Java Web アプリケーションに適用可能な設計レベルのパターンが含まれている。

- *Security Patterns: Integrating Security and Systems Engineering* [Schumacher 2006] ： この本には、さまざまな抽象化レベルの数多くのパターンが含まれており、セキュアなシステム開発に使用される開発プロセスなどのパターンから、異なるアクセス特権を有するオブジェクトを作成する方法など設計レベルのパターンまで取り扱っている。
- *Open Group Guide to Security Patterns* [Blakely 2004] ： この報告書では、システムの可用性および特権的な資源（Privileged Resources）の保護に焦点を合わせた、アーキテクチャレベルのパターンおよび設計レベルのパターンを扱っている。これらのパターンは、一般化されており、様々な言語で開発されたシステムに適用することができる。
- *Security Patterns Repository* [Kienzle 2003] ― この報告書には、セキュアなアプリケーションの設計と構築に適用可能な設計上のパターンと、セキュアなアプリケーションの設計、構築、設定プロセスとして適用可能な手続き上のパターンの両方が含まれている。

1.2 本書の目的

1.2.1 解決すべき問題

システムのデプロイメント後に脆弱性を修正するコストおよび脆弱性に関連するリスクは、開発者とエンドユーザの双方にとって高いものとなる。ソフトウェア開発組織は、システム保守のコストおよび脆弱性のリスクを低減するための対策を採用する必要がある。ソフトウェアのセキュリティ上の脆弱性に対応するためのベストプラクティスは数多く存在するが、それらは実装固有であり、多くの場合、異なる状況下での再利用が困難である。さらに、セキュリティ上の欠陥の根本原因に対する理解が深まるにつれ、実装とデプロイメントのフェーズに限らず、ソフトウェア開発ライフサイクルのすべてのフェーズにわたってセキュリティを考慮することの重要性に対する理解が深まってきている。ベストセキュリティプラクティスの多くは、実装とデプロイメントフェーズの課題に重点をおいており、開発プロセスの早い段階で入り込むセキュリティ上の欠陥について触れていない。

本報告書で詳説する各種のセキュアデザインパターンは、ソフトウェア開発ライフサイクルのアーキテクチャ設計、詳細設計、および実装フェーズにおける、セキュリティ上の課題を解決するためのものである。いくつかのパターンは、実績ある既存のベストプラクティスを分析し、一般化することによって作成された。また、本報告書では、いくつかの新しいセキュアデザインパターンの候補も提案しているが、これらはセキュリティを考慮するように既存のデザインパターンを拡張することで作成した。

既存のベストプラクティスの一般化と分類、ならびに既存のデザインパターンを拡張することにより作成されたセキュアデザインパターンは、セキュアなソフトウェア製品の開発者にとって有用である。再利用可能なセキャアデザインパターンを使用することで、開発者はセキュアな製品の生産コストを削減するとともに、セキュリティ脆弱性にかかわる開発者とエンドユーザの双方のコストとリスクを低減することができる。

1.2.2 セキュアデザインパターンの定義手法

本文書でパターンを定義するために採用した手法は以下のとおりである。

- 既存のセキュアなソフトウェアの設計手法から、他のシステムにおいても再現可能かつ再現すべき、有効性が実証されている技法を収集する

Copyright ©2009 JPCERT/CC All rights reserved.

- これらの技法をセキュアデザインパターンとして抽出し、文書化する

加えて、本報告書では、有効性が実証されていない新規のパターンもいくつか提案している。これらのパターンは、[Gamma 1995] で解説されている既知のデザインパターンをセキュリティの観点から拡張したものである。

本文書内のパターンは、以下などから着想を得ている。

- OpenBSD から派生したプロジェクト
- qmail および Postfix メールシステムの設計
- Kernighan と Pike による『*The Practice of Programming*』[Kernighan 1999] の推奨事項のうちセキュアデザインパターンに関係のあるもの
- [Gamma 1995] において解説されているデザインパターン

1.2.3 対象読者

本報告書の対象読者は、アーキテクチャ、設計、実装を含むさまざまな抽象レベルのソフトウェア開発における成果物を作成するソフトウェアエンジニアである。

本報告書のセキュアデザインパターンは、この抽象レベルに応じて分類されている。

1.3 対象範囲

セキュアデザインパターンは、コードに脆弱性が入り込むことを排除したり、脆弱性によって引き起こされる結果を緩和したりするための一般化された設計のガイダンスを提供する。セキュアデザインパターンは、オブジェクト指向設計の手法に限定されておらず、多くの場合、手続き型言語にも適用が可能である。これらのパターンは、セキュアコーディングガイドラインよりも抽象化のレベルが高い。

また、セキュアデザインパターンは、具体的なセキュリティ機能（アクセス制御、認証および許可（access control、authentication、authorization：AAA）およびログ記録）を記述したり、セキュアな開発プロセスを定義したり、既存のセキュアなシステムの設定に関するガイダンスを提供したりしないという点で、これらをカバーする「*セキュリティパターン*」とは異なる。

本報告書で提示するパターンは、大きく分けて以下の 3 つに分類される。

- **アーキテクチャレベルのパターン** ： アーキテクチャレベルのパターンは、システムの異なる構成要素間の責任配分を抽象度の高いレベルで扱い、構成要素間のやり取りを定義する。本報告書で定義するアーキテクチャレベルのパターンは以下のとおりである。
  - Distrustful Decomposition（相互信頼を前提としないシステム構成要素の分解）
  - PrivSep（Privilege Separation：権限分離）
  - Defer to Kernel（カーネルに委任）
- **設計レベルのパターン** ： 設計レベルのパターンは、システム構成要素をいかに設計し実装するかを記述する。つまり、単一のシステム構成要素の内部設計における問題を対象とする。本報告書で定義する設計レベルのパターンは以下のとおりである。
  - Secure State Machine（セキュアステートマシン）
  - Secure Visitor（セキュアビジター）

Copyright ©2009 JPCERT/CC All rights reserved.

- **実装レベルのパターン** ： 実装レベルのパターンは、抽象度の低い実装寄りのセキュリティ上の課題を対象とする。このクラスにおけるパターンは、一般にシステム内の特定の関数やメソッドの実装に適用できる。実装レベルのパターンは、CERT Secure Coding Standards [CERT 2009a] が対象とする問題と同種の問題を取り上げ、多くの場合、対応するセキュアコーディングのガイドラインに関連付けられている。本報告書で定義する実装レベルのパターンは以下のとおりである。
  - Secure Directory（セキュアディレクトリ）
  - Pathname Canonicalization（パス名正規化）
  - Input Validation（入力検査）
  - Resource Acquisition Is Initialization（リソースの確保は初期化時に行う）

なお、本報告書は、網羅的なセキュアデザインパターンのカタログを提供するものではない。本報告書の作成にあたっては、セキュアなソフトウェアの作成に使用されたベストプラクティスのいくつかを分析し、一般化した。セキュアデザインパターンのカタログは、将来拡充される予定である。

1.4 形式と規則

本報告書ではセキュアデザインパターンを記述するために、[Gamma 1995] で使用されたデザインパターンの記述テンプレートを使用した。テンプレート内の各項を表 1 に示す。斜体の項名は、その項が省略可能であることを示す。

表 1　　パターンの要素

要素	**説明**
目的	デザインパターンによって解決される問題と、その用途および目的。
別名	パターンの別名
例	問題が存在し、パターンが必要であることを示す実例。パターンの説明全体を通して、問題の解決法や実装面を説明する上で必要あるいは有用な場合には、実例に言及する。
動機	パターンが適用可能な状況の説明、およびパターンが解決を意図している問題のより詳細な説明。
適用可能性	プログラムの設計または実装においてパターンが有用であるためにプログラムが備えていなければならない特性の概要。
構造	パターンに関与するさまざまな要素間の関係を示す文章または図。適切な表記を使用し、パターンの構造面について詳細な仕様を提供する。
関連要素	パターンに関与する要素。
結果	パターンが提供するメリットと、発生し得るデメリット。
実装	パターンを実装するためのガイドライン。あくまで提案であり、不変の規則ではない。ガイドラインを参考に、ニーズに応じて実装方法は改良すべきである。可能な場合には、UML を提示することで実装例を示し、問題の詳細を説明する。
サンプルコード	パターンの実装例を示すコード。
解決例	セキュアデザインパターンを使用することで、「*例*」の項で説明した現実の問題をどのように解決できるのかを示す例。
既知の使用例	既存のシステムに見られるパターンの使用例。

Copyright ©2009 JPCERT/CC All rights reserved.

2 アーキテクチャレベルのパターン

2.1 Distrustful Decomposition（相互信頼を前提としないシステム構成要素の分解）

2.1.1 目的

Distrustful Decomposition セキュアデザインパターンの目的は、個々の機能を互いに信頼しない複数のプログラムに分離することで、以下を低減すること。

- システム全体を構成する個々のプログラムの攻撃対象範囲
- 相互に信頼しないプログラムのいずれかが侵害された場合に攻撃者に晒される機能やデータ

2.1.2 別名

Privilege reduction（権限降格）

2.1.3 動機

攻撃の多くは、昇格した権限で実行されている脆弱なアプリケーションを狙って行われる。攻撃者は、アプリケーションのセキュリティホールを悪用した後、アプリケーションがより限定的な権限で実行されている場合よりも多くの情報にアクセスしたり、より多くの損害を与えたりすることができる。この手の攻撃の実例としては、以下がある。

- 管理者権限で実行されている Internet Explorer が侵害されるさまざまな攻撃
- 管理者特権のもとで実行されているときに、攻撃者により任意の VB スクリプトの実行が可能となる Norton AntiVirus 2005 のセキュリティ上の欠陥
- root 権限で任意のコード実行を可能にする、BSD 由来の telnet デーモンにおけるバッファオーバーフローの脆弱性

これらの攻撃はすべて、昇格した特権（UNIX では root、Windows では管理者）で実行されているアプリケーションのセキュリティ上の欠陥を利用してアプリケーションを侵害した後、そのアプリケーションの特権と基本機能を使用して、対象コンピュータ上の他のアプリケーションを侵害したり、機密データにアクセスしたりする。Distrustful Decomposition パターンは、セキュリティ上の脆弱性をシステムの小さなサブセットに隔離することで、システムの 1 つの構成要素が侵害された場合でも、それがシステム全体の侵害につながらないようにする。攻撃者が悪用できるのは、アプリケーション全体の機能とデータではなく、侵害した 1 つの構成要素の機能とデータだけである。

2.1.4 適用可能性

さまざまな特権および責任のもとで実行されるプログラム群が、ファイルやユーザの提供するデータを様々な方法で扱う必要のあるシステムにおいて、このパターンを適用することができる。そのようなシステムをよく考えずに実装すると、本質的に異なる多数の機能を一つのプログラムに割り当ててしまい、それらの機能のうち最も高い特権レベルを必要とする機能を実行するために必要な権限でそのプログラムを実行せざるをえない可能性がある。その結果、広範な攻撃範囲が攻撃者に晒らされ、システムが侵害された場合には高い特権レベルでのアクセスを許すことになる。

Copyright ©2009 JPCERT/CC All rights reserved.

システムが以下の条件に当てはまる場合に Distrustful Decomposition パターンを使用できる。

- システムが複数の抽象度の高い機能を持つ
- それらの機能が異なる特権レベルを必要とする

2.1.5 構造

このパターンの全般的な構造は、システムを別々のプロセスとして実行される複数のプログラムに分割するというものである。それぞれのプロセスは異なる特権を持てる。各プロセスは、小規模の厳密に定義されたシステム機能のサブセットを扱う。分割されたプロセス間の通信は、RPC、ソケット、SOAP、共有ファイルなどのプロセス間通信メカニズムを使用し、一つのシステムとして機能する。

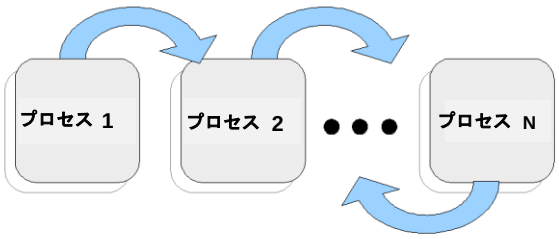

図 1　Distrustful Decomposition デザインパターンの概略構造

2.1.6 関連要素

Distrustful Decomposition パターンに含まれる関連要素は以下のとおりである。

- いくつかの別々のプログラム。それぞれが個別のプロセスとして実行される。より厳密な分離を実現するために、特権を他のユーザ ID と共有しない一意なユーザ ID を各プロセスに持たせることもできる。
- ローカルユーザまたはネットワーク経由で接続するリモートシステム
- 場合によっては、システムのファイルシステム
- 場合によっては、UNIX ドメインソケット、RPC、SOAP などのプロセス間通信メカニズム

2.1.7 結果

Distrustful Decomposition は、攻撃によってシステムの構成要素の一つが侵害されたとしても、システムを構成する他のプログラム群が侵害されたプログラムの実行結果を信頼していないため、システム全体が侵害されることを防ぐ。

2.1.8 実装

このパターンは、オペレーティングシステムに備わっている標準のプロセス／特権モデル以外は何も利用しない。各プログラムは、それぞれ独自のプロセス空間の中で、別々のユーザ特権のもとで実行されることもある。個別のプログラム間の通信は、単方向または双方向である。

- 単方向：単に `fork()/exec()`（UNIX、Linux、その他）、`CreateProcess()`（Windows Vista）、またはプロセスを作成する OS 固有のその他の何らかの方法を使用して制御を

Copyright ©2009 JPCERT/CC All rights reserved.

引き渡す。単方向通信は、プロセス間の結び付きを弱くするため、侵害されたある特定のシステム構成要素から別の構成要素を攻撃者が侵害するのを困難にする。

- 双方向：TCP または SOAP のような双方向のプロセス間通信メカニズムを使用する。双方向通信に関与する 1 つのプロセスが侵害され、攻撃者の支配下におかれる可能性があるため、双方向通信メカニズムを使用する場合には特に慎重を期す必要がある。ファイルシステムと同様に、双方向通信は本質的に信頼すべきではない。

ファイルシステムはやり取りの手段として利用されることも有るが、システムの各構成要素はファイルの内容を本質的に信頼すべきではない。

2.1.9 サンプルコード

このパターンが適用されている優れたシステムの例に、qmail メールシステムがある。このシステムは、システム、ユーザおよびソフトウェア構成要素の間で膨大な数の組み合わせでやり取りが行われる複雑なシステムである。

qmail システムの全体構造を図 2 [Oppermann 1998] に示す。

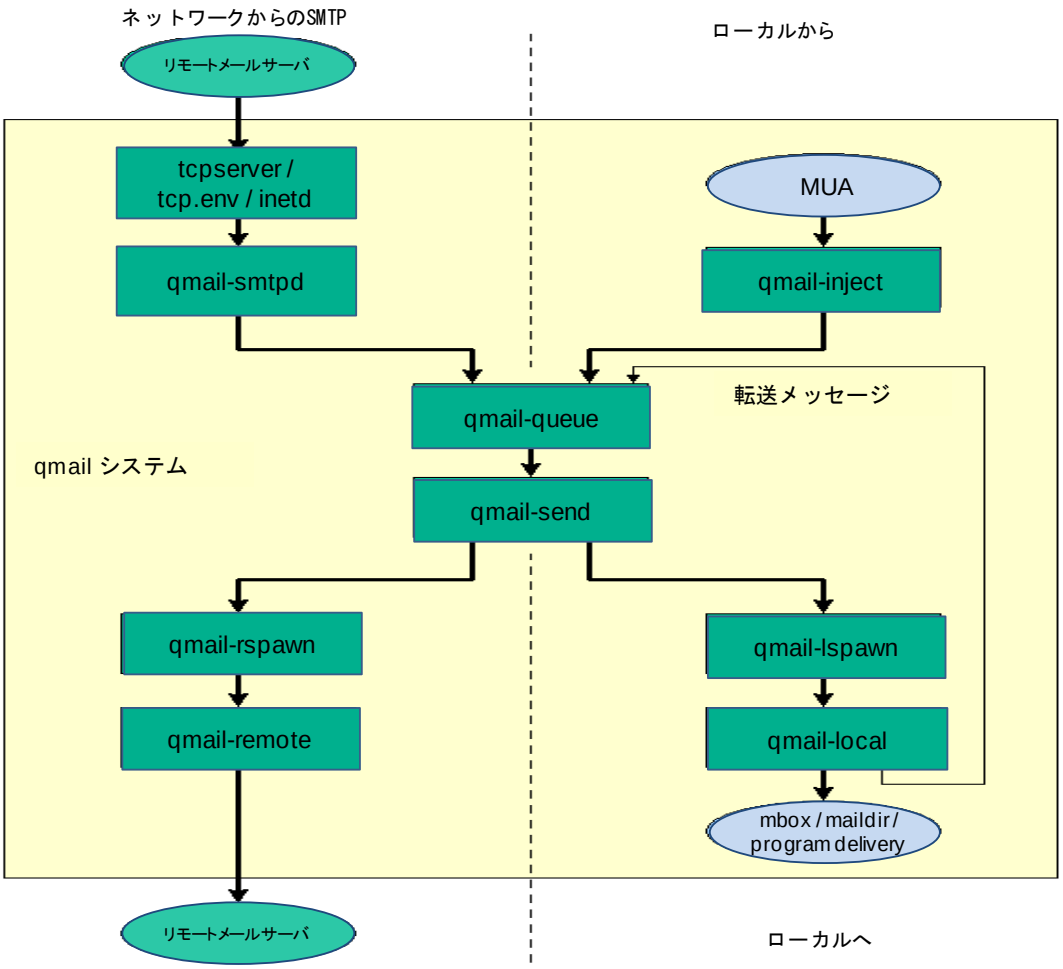

図 2　　qmail メールシステムの構造[1]

[1] 出展: http://www.nrg4u.com/qmail/the-big-qmail-picture-103-p1.gif. 作成者の許可に基づき使用。

Copyright ©2009 JPCERT/CC All rights reserved.

qmail システムの実際のソースコードはここでは省略する。コード例は、qmail の Web サイト [Bernstein 2008] を参照されたい。

2.1.10 既知の使用例

- qmail メールシステム [Bernstein 2008]。
- Postfix メールシステムも類似のパターンを使用する [Postfix]。
- Microsoft は、管理者権限でアプリケーションを実行する方法を議論する中でこのパターンに言及している [MSDN 2009b]。

ユーザアカウント制御（UAC：User Account Control）を使用した Windows Vista アプリケーションのための Distrustful Decomposition は [Massa 2008] に言及されている。

2.2 PrivSep（Privilege Separation：権限分離）

2.2.1 目的

PrivSep パターンの目的は、プログラムの機能性に影響を与えたり制限を加えたりせずに、特権で実行されるコードの量を削減することにある。PrivSep パターンは、Distrustful Decomposition パターンのより具体的な事例である。

2.2.2 動機

多くのアプリケーションにおいては、高い特権を必要とする操作は少数であり、それよりもはるかに多くの複雑でセキュリティ上の間違いが生じる可能性の高い操作は、特権を必要としない一般ユーザ権限で実行できる。このパターンを使用する動機のさらに詳しい議論については、より汎用的な Distrustful Decomposition パターンの動機を参照されたい。

PrivSep パターンを適用できるシステムの詳細と、PrivSep パターンを適用しない場合に生じ得るセキュリティ問題を図 3 に示す [Provos 2003]。ここでは例として、`ftpd` の実装を使用している。

Copyright ©2009 JPCERT/CC All rights reserved.

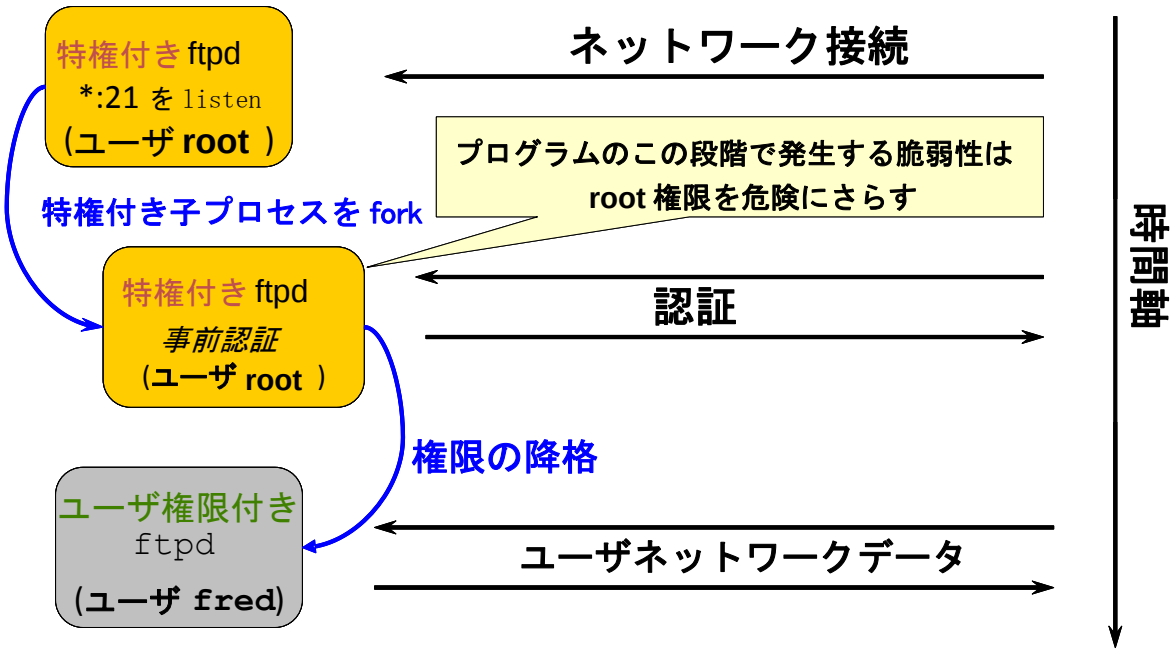

図3　　脆弱な ftpd プログラム

このセキュリティ上の欠陥は、特権を持つサーバプロセスが、その時点で信頼されていないシステムユーザとの接続を確立し、サーバプロセスと同じ特権を有する子プロセスを使用してユーザの認証を試みるときに生じる。悪意のあるユーザがこの時点で、特権を有する子プロセスのセキュリティホールを悪用できる場合、特権を持つプロセスが支配下に置かれることを許してしまう。

2.2.3　適用可能性

一般にこのパターンは、システムが以下の条件に合致する場合に適用可能である。

- 特権を必要としない一連の機能を実装しているシステム
- システムの機能が以下の特徴を有することから、攻撃される領域が比較的広範である
  - 信頼できない通信元と大量の通信を行う
  - 複雑で、潜在的に誤りの生じやすいアルゴリズムを使用する

このパターンは特に、ユーザ認証が必要で、認証後に一般ユーザ権限での対話型プログラムの実行を許可するシステムサービスには有用である。また、他の認証を行うサービスにも有用である可能性がある。

2.2.4　構造

PrivSep パターンの構造と振る舞いを図 4 に示す。この図では、子プロセスを作成するために UNIX の `fork()` 関数を使用している。非 UNIX 系の OS に PrivSep パターンを実装する場合は、`fork()` の代わりにその OS 固有の関数を使用する。たとえば、Windows の各種バージョンでは、`CreateProcess()` 関数を使用して子プロセスを作成する。

Copyright ©2009 JPCERT/CC All rights reserved.

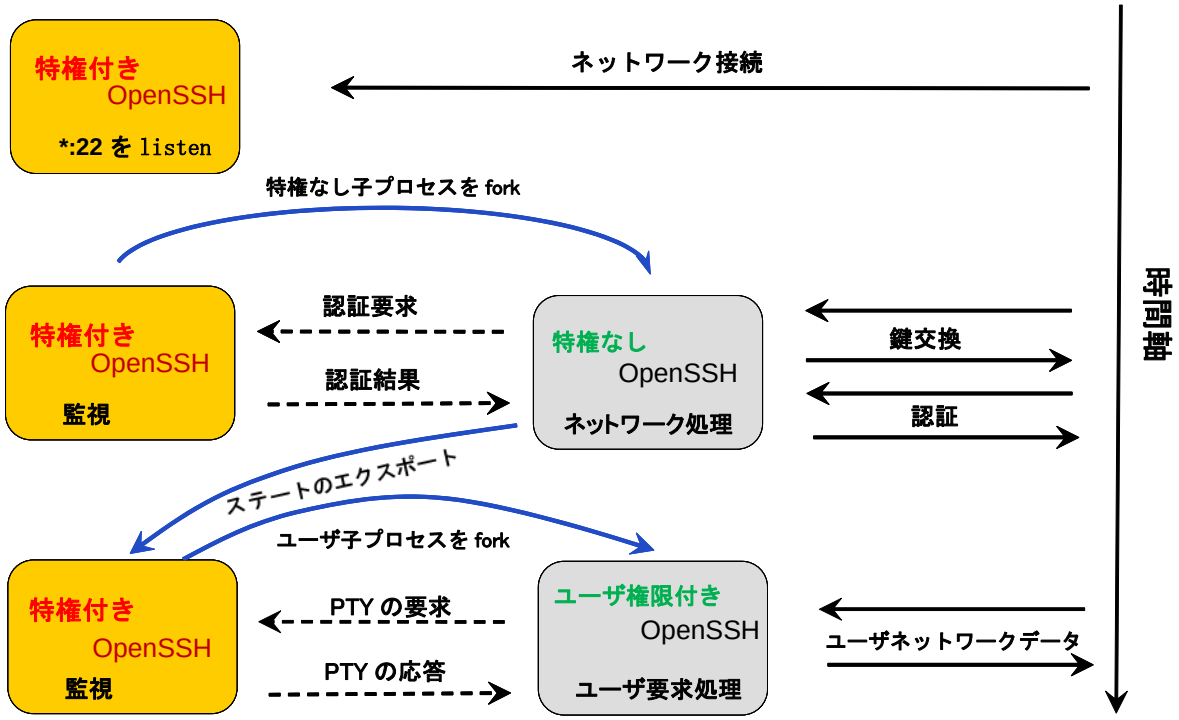

図 4:　　OpenSSH における PrivSep の実装[2]

2.2.5　関連要素

特権付きサーバプロセス

特権付きサーバプロセスは、権限昇格していない子プロセスにより最終的に処理される機能に対する初期要求の振り分けを行う。特権付きサーバには、特権付きユーザ ID が関連付けられている（多くの場合、UNIX 系 OS では `root`、また Windows の各種バージョンでは `administrator`）。

システムユーザ

システムユーザは、システムに対し何らかのアクションの実行を依頼する。機能に対する初期要求は、システムユーザから特権付きサーバに送られる。システムユーザは、ローカルユーザ、遠隔のユーザのどちらの場合もあり得る。システムユーザは、ソケットや SOAP などのプロセス間通信メカニズムを利用して特権付きサーバと通信する。

特権なしクライアントプロセス

特権なしクライアントは、システムユーザの要求の認証を取り扱う。要求が信頼できるシステムユーザからの有効な要求であることはまだ確認されていないので、認証を取り扱う子プロセスの権限は、以下のように制限されている。

- 子プロセスは、ホスト OS によって許可される最低限の特権のセットを与えられる。UNIX の権限モデルのもとでは、これはプロセスのユーザ ID(UID)を特権なしユーザ ID にすることで実現している。

[2] 出展: http://www.citi.umich.edu/u/provos/ssh/priv.jpg.

Copyright ©2009 JPCERT/CC All rights reserved.

- 子プロセスのルートディレクトリは、重要でない空のディレクトリまたは jail に設定される [Seacord 2008]。これにより、信頼できない子プロセスを実行しているマシン上の任意のファイルに対し、その子プロセスによるアクセスが防止される。

ユーザ権限付きクライアントプロセス

システムユーザとその要求がひとたび認証されれば、適切なユーザレベル権限を有する子プロセスが特権付きサーバによって作成される。システムユーザの要求を実際に処理するのは、ユーザ権限付き子プロセスである。ユーザ権限付き子プロセスの UID は、ローカルユーザの ID に設定される。

2.2.6 結果

攻撃者が子プロセスを支配下に治めることに成功した場合、以下の状態となる。

- 子プロセスの保護ドメイン内に封じ込められ、親プロセスの制御を得られない
- 特権を持つプロセスを制御できないため、攻撃者が及ぼし得る損害が限定される

特権で実行されるコードに的を絞ってコードレビュー、追加テスト、および形式的検証手法などの検証を追加で実施することで、不正な権限昇格の発生をさらに減らせるだろう。

新規の特権なしユーザ ID の管理を行うため、システム管理上のオーバーヘッドが増加する。

2.2.7 実装

PrivSep パターンは、認証前と認証後の 2 つのフェーズで構成される。

- 認証前 ： ユーザがシステムサービスに接触したが、まだ認証されていない。特権なし子プロセスはプロセスの特権も、ファイルシステムにアクセスする権限もない。

  認証前のフェーズは、監視役の役割を担う特権付き親プロセスと、従属的な役割を担う（スレーブ）特権なし子プロセスの 2 つのエンティティによって実現される。特権付き親プロセスは、特権なし子プロセスの進行を監視する有限状態マシン（FSM：Finite-State Machine）としてモデル化できる。
- 認証後 ： ユーザがシステムに認証された。子プロセスは、ファイルシステムへのアクセスを含め、ユーザの権限を有するが、それ以外の特権は持たない。

PrivSep パターンでは一般的に以下のような処理が実装される。

1. 特権付きサーバを作成する。ユーザによる初期要求はこのサーバに送られる。
2. ユーザの要求がサーバに到着すると、サーバは信頼のない特権なし子プロセスを作成し、認証処理の実行中に必要となるユーザとのやり取りを処理させる。
3. ユーザ認証後、サーバは適切な UID を持つ別の子プロセスを作成し、そのプロセスにユーザの要求を実際に処理させる。

特権なし子プロセスは、その UID またはグループ ID（GID）を、他に使われていない ID に変更することで作成される。これは、特権を持つ監視プロセスをまず開始し、監視プロセスに従属（スレーブ）プロセスを fork させることで実現する。ファイルシステムへのアクセスを防止するために、信頼のない子プロセスは、ファイルシステムのルートをファイルを書き込めない空のディレクトリに変更する。信頼のない子プロセスは、自身の UID または GID を、特権なしユーザの UID に変更することでその特権を失う。

Copyright ©2009 JPCERT/CC All rights reserved.

スレーブから監視プロセスへの要求は、標準のプロセス間通信メカニズムを使用して行われる。

2.2.8 サンプルコード

fork()、chroot()および setuid()を使用する PrivSep パターンの Linux における簡単な実装例を以下に示す。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>

// 未使用の UID を定義する
#define UNPRIVILEGED_UID 123456789

// ユーザの UID（実際には、ハードコード
// すべきではない。）
#define USER_UID 1000

// 信頼のない子プロセスのルートディレクトリとして使用する
// 空のディレクトリの場所
#define EMPTY_ROOT_DIR "/home/sayre/empty_dir"

/**
 * 以下は、作成した、特権なし子プロセスおよびユーザ権限付き子プロセスの
 * 両方の振る舞いを定義する。
 *
 * パラメータは、以下のとおり：
 *
 * childUid – 作成した子プロセスに割り当てる UID。
 *
 * sock – 子プロセスが、特権のある親プロセスとの通信に使用するソケット。
 *
 */
void handleChild(uid_t childUid, int sock) {

  // ソケットからメッセージを読み取るためのバッファ
  char buffer[100];

  // 自分が、信頼のない子プロセスならば、信頼のない子プロセスのファイルシステムのルートを
  // 空のディレクトリに変更する。
  if (childUid != USER_UID) {
    if (chroot(EMPTY_ROOT_DIR) != 0) {
      printf("Cannot change root directory to %s.\n", EMPTY_ROOT_DIR);
      exit(7);
    }
  }
  // 子プロセスの UID を即座にユーザまたは特権なし
  // UID に設定する。
  if (setuid(childUid) < 0) {
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  printf("Cannot set UID to %d\n", childUid);
  exit(6);
}

// この時点で、子プロセスは、特権のある親プロセスの全部の特権は
// 持っていない状態となる。

// 自身が、権限のチェックの対象となる特権なし子プロセスであるかどうかを
// 調べる。
if (childUid != USER_UID) {

  // 特権なし子プロセスである。

  // 子プロセスの資格情報を確認するよう、特権付き親プロセスに依頼する。
  // このサンプルコードの目的のために、「資格情報」は非常に簡単に
  // 表してる。PrivSep パターンの実際の適用においては、資格情報は、
  // より厳密な方法で取り扱われる。
  send(sock, "VERIFY: MY_CREDS", 17, 0) ;

  // 親プロセスから、資格情報の確認結果を読み取る。
  int size = recv(sock, buffer, sizeof(buffer), 0) ;

  // 親プロセスから確認結果を読み取るときにエラーは生じたか。
  if (size < 0) {
    printf("Read error in parent.\n");
    exit(2);
  }

  // 送られてきた結果文字列の末尾が null で終了していることを
  // 確認する。
  buffer[sizeof(buffer)-1] = '\0';

  // 資格情報は、「認証済み」か。
  if (strcmp("yes", buffer) != 0) {

    //認証失敗。子プロセスを適切なエラーコードを与えて
    // 強制終了する。
    printf("Authorization denied.\n");
    exit(5);
  }

  // 資格情報が認証された。
  printf("Authorization approved.\n");

  // 特権なし子プロセスがここで終了する。特権付き親プロセスは、
  // ユーザの権限を持つ子プロセスを作成する。
  exit(0);
}

// 自分は、ユーザの UID を持つ子プロセスである。認証はすでに
// 済んでいる。
else {
  // 認証済みの子プロセスにおける実際の作業をここで行う...
  // ...
  // ...
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
    // ...
  }
}

int main(int argc, const char* argv[]) {

  // 親プロセスと子プロセスが通信のために使用するソケットのペアを
  // 作成する。
  int sockets[2];
  if (socketpair(PF_UNIX, SOCK_STREAM, AF_LOCAL, sockets) != 0) {

    // ソケットのペア作成に失敗した。プロセスを終了する。
    exit(1);
  }

  // ソケットからメッセージを読み取るためのバッファ
  char buffer[100];

  // 作成された子プロセスは最初に、自身の UID を、特権の ID に変更しなければ
  // ならない。
  uid_t childUid = UNPRIVILEGED_UID;

  // 親プロセスを fork し、特権なし子プロセスを作成する。
  pid_t pID = fork();

  // 自身は子プロセスか。
  if (pID == 0) {
    // 特権なし子プロセスを認証処理に使用する。
    handleChild(childUid, sockets[0]);
  }

  // fork は失敗したか。
  else if (pID < 0) {
    printf("Fork failed\n");
    exit(3);
  }

  // 自身は親プロセスである。
  else {

    // この時点で、親プロセスは信頼のない子プロセスが認証を得ようとすることを
    // 期待する。

    // 親プロセスのためのソケットを取得する。
    int sock = sockets[1];

    // 子プロセスから認証要求を受け取る。
    int size = recv(sock, buffer, sizeof(buffer), 0) ;

    // 認証要求メッセージの読取り時にエラーは生じなかったか。
    if (size < 0) {
      printf("Read error in parent.\n");
      exit(4);
    }
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
    // 送られてきた文字列の末尾が null で終了していることを確認する。
    buffer[sizeof(buffer)-1] = '\0';

    // 子プロセスの「認証」を行う。この簡単なサンプルにおいては、
    // 認証処理を簡単なものにしていることに留意してほしい。PrivSep の
    // 実際の適用においては、資格情報は、より厳密な
    // 認証プロセスを使用する。
    if (strcmp("VERIFY: MY_CREDS", buffer) == 0) {

      // 認証に成功した。子プロセスにそれを伝える。
      send(sock, "yes", 4, 0);

      // 「認証」は成功したので、実際の作業を行うユーザの UID を持つ
      // 新しい子プロセスを作成する。
      childUid = USER_UID;

      // 親プロセスを fork し、特権なし子プロセスを作成する。
      pid_t pID = fork();

      // 自分は子プロセスか。
      if (pID == 0) {
        handleChild(childUid, sockets[0]);
      }

      // fork は失敗したか。
      else if (pID < 0) {
        printf("Fork failed\n");
        exit(3);
      }

      // 自分は親プロセスである。
      else {

        // 必要に応じて、親プロセスとしてのその他の操作を行う...
        // ...
        // ...
        // ...
      }
    }
    else {
      // 認証が失敗した。子プロセスにそれを伝える。
      send(sock, "no", 4, 0) ;
    }
  }
}
```

2.2.9 既知の使用例

OpenBSD：sshd、bgpd/ospfd/ripd/rtadvd、X window server、snmpd、ntpd、dhclient、tcpdump、他。

Copyright ©2009 JPCERT/CC All rights reserved.

2.3 Defer to Kernel（カーネルに委任）

2.3.1 目的

このパターンの目的は、昇格した権限を必要とする機能を、昇格した権限を必要としない機能から明確に分離し、カーネルレベルで利用できる既存のユーザ確認機能を利用することにある。カーネルが持つ既存のユーザ確認機能を使用することで、ユーザレベルでセキュリティ判定の手段を改めて作成するのではなく、セキュリティ判定を実施する確立済みのカーネルの役割を活用する。

Defer to Kernel パターンは以下のパターンを特化させたものである。

- 本報告書で定義する Distrustful Decomposition セキュアデザインパターン
- Schumacher らが作成した Reference Monitor セキュリティパターン

  Reference Monitor は、資源に対する要求をすべて傍受し、それらの要求が許可された権限に基づくことを確認する、抽象的な処理の定義方法を記述する汎用的なパターンである [Schumacher 2006]。

Defer to Kernel パターンと Reference Monitor パターンの主たる違いは、Defer to Kernel パターンでは OS のカーネルが提供するユーザ確認機能を使用することを明示しているのに対し、Reference Monitor パターンでは具体的なユーザ確認方法を指定していないことである。

2.3.2 動機

このパターンを実装する主たる動機は、昇格した権限で動作し、その結果として権限昇格の攻撃を受ける可能性のあるユーザプログラムの必要性を低減または回避することにある。つまり、UNIX 系のシステムでは `setuid` を使うプログラムの削減もしくは回避を、Windows では、管理者権限で動作するユーザプログラムを回避することを意味する。

加えて、このパターンは、OS のカーネルが提供するユーザ確認機能を利用することに重点をおいている。ユーザの確認にカーネルの既存の機能を利用することの利点は以下のとおりである。

- 開発者が独自にユーザの識別と確認を行う機能を作成する必要がない。
- カーネルの既存のユーザ識別、ユーザ確認機能のテストおよび検証はすでに済んでいる。
- それぞれの OS で、それぞれのプラットフォームに合った方法でユーザを確認できるため、可搬性のある解決策である。

このパターンを使用する動機として、より汎用的な Distrustful Decomposition パターンの動機も参照されたい。

2.3.3 適用可能性

Defer to Kernel パターンは、システムに以下の特性がある場合に適用可能である。

- システムが、昇格した権限を持っていないユーザによって実行されている。
- システムの一部（場合によっては全部）の機能が、昇格した権限を必要とする。
- 昇格した権限を必要とする機能を実行する前に、ユーザがその機能を実行することを許可されているかをシステムが確認しなければならない。

特に、UNIX 系の OS 上で動作するシステムでは、以下の特性がある場合に Defer to Kernel パターンが適用可能である。

Copyright ©2009 JPCERT/CC All rights reserved.

- タスクの一部または全部を行うためには、プログラムは特別な UID で動作する必要がある。
- プログラムはユーザから送信されたファイル要求またはジョブ要求を受け取る。
- ユーザがローカルユーザである場合、アクセス制御または課金管理のために、どの UID または GID が、それぞれのファイル要求またはジョブ要求を送信したかをプログラムは知る必要がある。
- ユーザがローカルユーザではない場合、プログラムは他のユーザ認証メカニズムとログ記録メカニズムを使用する。

2.3.4 構造

Defer to Kernel パターンは、基本的なクライアント／サーバ型のアーキテクチャを実現する。サーバは、昇格した権限で実行され、クライアントからユーザジョブ要求を受け付け、可能であれば既存のカーネル機能を使用してユーザを確認する。

Defer to Kernel パターンの概略構造を図 5 に示す。

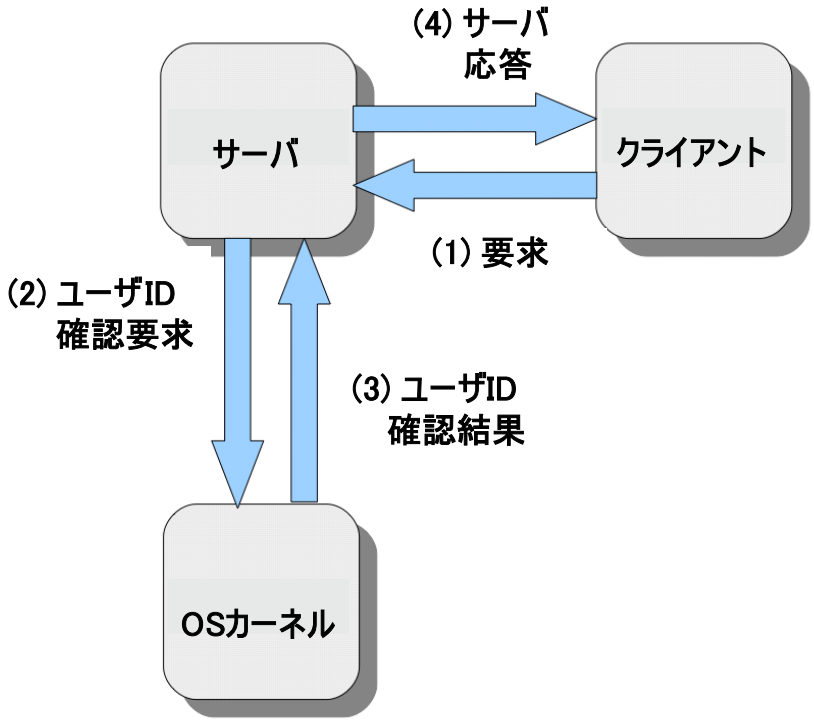

図 5: Defer to Kernel パターンの概略構造

システムが以下の特性を持つ場合の Defer to Kernel パターンの構造を図 6 に示す。

- システムが UNIX 系 OS に実装されている。
- システムがユーザ確認に `getpeereid()` を使用する。
- サーバはファイル要求またはジョブ要求をローカルユーザからのみ受け付ける。

Copyright ©2009 JPCERT/CC All rights reserved.

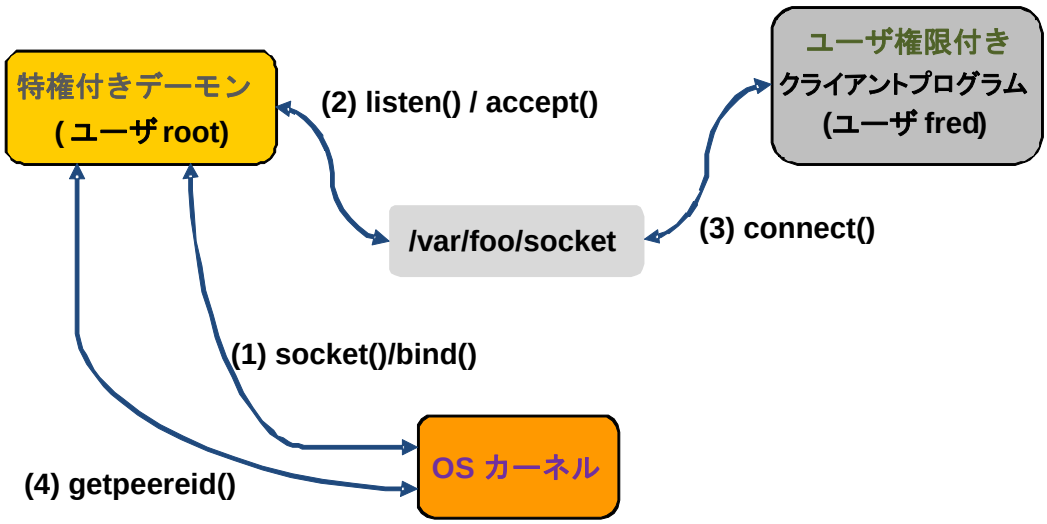

図 6: Defer to Kernel パターンの構造例

2.3.5 関連要素

Defer to Kernel パターンの関連要素は以下のとおりである。

- クライアントプログラム ： クライアントプログラムは、標準のユーザレベル権限で実行される。プログラムは、サーバにジョブ要求を送信し、クライアントが実行権限を持たない作業の実行を依頼する。
- システムカーネル ： システムカーネルは以下を提供する。
  - クライアント／サーバ間の通信に使用するプロセス間通信メカニズム
  - ユーザ ID の取得と確認機能。カーネルに実装されている既存の機能を使用して、サーバに接続しているユーザ ID を取得し、そのユーザから送信されたジョブ要求がサーバ上での実行を許可されているかを確認する。
- サーバプログラム ： サーバプログラムは、割り当て済みの IPC メカニズムの監視、クライアントから受け取るジョブ要求の読取り、クライアントから送信されたジョブをサーバで実行すべきかどうかの確認を行う。

2.3.6 結果

これまで単一の実行ファイル（UNIX 系 OS では `setuid` を使用する実行ファイル、Windows では管理者権限で実行される実行ファイル）に依存していたアプリケーションを、クライアント／サーバシステムとしてアーキテクチャを変更する必要がある。

クライアント／サーバ間の通信が加わるため、システムの複雑さは増す。

2.3.7 実装

Defer to Kernel パターンの実装の概要は以下のとおりである。

1. サーバが起動する。サーバは、何らかの既知のメカニズムを使用してクライアント要求を受け付ける。

Copyright ©2009 JPCERT/CC All rights reserved.

2. クライアントは、サーバに要求を送信する。要求には、クライアントを識別する情報が含まれている。この情報は、エンコードされ、OS のカーネルに備わっているユーザ識別メカニズムを使用して送信される。
3. サーバは、ユーザ要求を受け取り、何らかのカーネルレベルのメカニズムを使用してユーザの要求に応えるか、要求を拒否するかを判定する。

UNIX 系 OS に適した、より具体的な実装は以下のとおりである。

1. サーバ（cron ジョブ、プリントジョブなど）が UNIX ドメインソケットを既知のパス上にオープンする。クライアント要求はすべてこの UNIX ドメインソケットに送信される。
2. クライアントは、このソケットに接続して要求を送信する。UNIX ドメインソケットを使用しているため、クライアントユーザの UID および GUID に関する情報はメッセージに自動的に含まれる。これは、基盤のソケットコードによって実施され、クライアントに明示的にプログラミングする必要がない。
3. サーバは、`connect()`要求を受け取ると、その要求を行っているユーザを識別するために、`getpeereid()`などのシステムコールを呼び出す。
4. サーバは直前のステップで得たユーザ識別情報を使用して、ユーザの要求に応えるべきかを決める。

このシステムは、ローカルユーザのみを対象としており、ネットワーク経由で遠隔から接続するユーザを対象としないことに注意すること。

Linux は `getpeereid()`関数を含まないが、`getpeereid()`は以下のように簡単に実装できる。

```
/**
 * 指定された UNIX ドメインソケットに接続しているユーザのユーザ ID とグループ ID を
 * 取得する。
 *
 * @param sd UNIX のドメインソケット
 * @param uid 指定された UNIX ドメインソケットに接続しているユーザのユーザ ID を
 * 格納する場所。uid のためのメモリ領域は呼び出し元が割り当てなければならない。
 * @param gid 指定された UNIX ドメインソケットに接続しているユーザのグループ ID を
 * 格納する場所。gid のためのメモリ領域は呼び出し元が割り当てなければならない。
 *
 * @returns -1 を失敗の場合に返し、0 を成功の場合に返す。
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  int len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;

  return 0;
}
```

Windows ではそのセキュリティの基盤設計ゆえ Defer to Kernel パターンを簡単に実装できる。Windows では、すべてのプロセスおよびスレッドが、そのプロセスのアカウントを所有するユーザのセキュリティ識別子（SID : Security Identifier）およびユーザのグループの SID など

Copyright ©2009 JPCERT/CC All rights reserved.

を含んでいる「*アクセストークン（access token）*」を持つ。サーバを Windows サービスとして用意することもできる。Windows サービスは、サービスを「*セキュリティ保護可能オブジェクト（securable object）*」にすることでセキュリティを強化することができる。セキュリティ保護可能オブジェクトを作成するときに、それに「*アクセス制御リスト（access control list）*」を関連付けることが可能である。アクセス制御リストには、サーバに接続を許可されるクライアントプロセス、つまり Windows サービスの SID が含まれている。

Windows のセキュリティ保護可能オブジェクトの詳細については、オンラインチュートリアル『Access Control Story: Part I』 [Tenouk 2009] を参照されたい。

2.3.8 サンプルコード

Linux では、Defer to Kernel デザインパターンにおけるサーバ部分のサンプルコードは以下のようになる。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define QUEUE_SIZE          5
#define SOCKET_PATH     "/tmp/myserver"

/**
 * 指定された UNIX ドメインソケットに接続しているユーザのユーザ ID とグループ ID を
 * 取得する。
 *
 * @param sd UNIX のドメインソケット
 * @param uid 指定された UNIX ドメインソケットに接続しているユーザのユーザ ID を
 * 格納する場所。uid のためのメモリ領域は
 * 呼び出し元が割り当てなければならない。
 * @param gid 指定された UNIX ドメインソケットに接続している
 * ユーザのグループ ID を格納する場所。gid のためのメモリ領域は
 * 呼び出し元が割り当てなければならない。
 *
 * @returns -1 を失敗の場合に返し、0 を成功の場合に返す。
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  socklen_t len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  return 0;
}

/* 以下は、ユーザ確認関数を表している。宣言のみでサンプルでは実装されない。

   この関数の目的は、接続しているユーザの要求を受け付けるべきかを
   確認すること、つまり、接続しているユーザが要求（任意の要求）を
   サーバに送信することを許可されているかどうか確認
   することである。validateUser()は、ユーザのユーザ ID およびグループ ID を
   使用することに留意すること。いずれもカーネルレベルの getpeereid()関数を
   使用して取得される。

   ユーザの実際の要求は、validateRequest()関数を使用して
   その有効性が確認される。
*/
extern int validateUser(uid_t uid, gid_t gid);

/*
   この関数の目的は、接続しているユーザの要求を処理すべきかを
   調べることである。validateUser()の場合と同様に、
   validateRequest()はユーザのユーザ ID とグループ ID を使用して、
   サーバに要求を処理させる権限を接続しているユーザが
   持っているかを調べる。

   宣言のみでサンプルでは実装しない。
*/
extern int validateRequest(uid_t uid, gid_t gid, char *request);

int main(int argc, char* argv[]) {
  int hSocket,hServerSocket;  /* ソケットのハンドル */
  struct hostent* pHostInfo;   /* マシンに関する情報を保持 */
  struct sockaddr_un Address; /* Internet ソケットアドレス構造体 */
  int nAddressSize=sizeof(struct sockaddr_in);
  char pBuffer[BUFFER_SIZE];

  /* 受信クライアント要求のための UNIX ドメインソケットを作成する。*/
  hServerSocket=socket(AF_UNIX,SOCK_STREAM,0);

  if (hServerSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }

  /* アドレス構造体を埋めることで、UNIX ドメインソケットの作成方法を
      定義する。 */
  Address.sun_family = AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);
  unlink(Address.sun_path);

  /* 受信要求ソケットを「既知」のパスにバインドする。 */
  /* この簡単なサンプルでは、「既知」のパスをハードコードしている。 */
  if(bind(hServerSocket,(struct sockaddr*)&Address,sizeof(Address))
     == SOCKET_ERROR) {
    puts("\nCould not connect to host\n");
    return 0;
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
}

/*  ポート番号を取得する */
getsockname(hServerSocket,
            (struct sockaddr *) &Address,
            (socklen_t *)&nAddressSize);

/* 受信要求ソケットのためのリッスンキューを確立する。 */
if (listen(hServerSocket,QUEUE_SIZE) == SOCKET_ERROR) {
  puts("\nCould not listen\n");
  return 0;
}

/* クライアント要求を取得し、処理する。 */
for(;;) {

  /* 受信接続を通じてユーザ要求を取得する。 */
  hSocket=accept(hServerSocket,(struct sockaddr*)&Address,
                 (socklen_t *)&nAddressSize);

   /* 誰が接続してきたかを調べる。 */
  uid_t connectedUID;
  gid_t connectedGID;
  if (getpeereid(hSocket, &connectedUID, &connectedGID) != 0) {

   /* 誰が接続してきたかが分からない。接続を拒否する。 */
    puts("Cannot figure out who connected. Booting them.");
    if(close(hSocket) == SOCKET_ERROR) {
      puts("ERROR: Could not close socket\n");
      return 0;
    }

    /* 受信接続をさらに取得する。 */
    continue;
  }

  /* 要求を行うユーザの有効性を確認する。 */
  if (!validateUser(connectedUID, connectedGID)) {

    puts("User not validated. Booting them.");
    if(close(hSocket) == SOCKET_ERROR) {
      puts("ERROR: Could not close socket\n");
      return 0;
    }

    /* 受信接続をさらに取得する。 */
    continue;
  }

/* ユーザの要求を取得する。 */
  char *currRequest;
  /* .
     .  ( ユーザの要求は、currRequest に設定されるとする。 )
     . */
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  /* 接続しているユーザの要求の有効性を確認する。 */
    if (!validateRequest(connectedUID, connectedGID, currRequest)) {

      puts("User issued invalid request. Booting them.");
      if(close(hSocket) == SOCKET_ERROR) {
        puts("ERROR: Could not close socket\n");
        return 0;
      }

      /* 受信接続をさらに取得する。 */
      continue;
    }


  /* ユーザの要求を処理する。 */
    /* .
       .
       . */

  /* 現在のユーザに接続しているソケットを閉じる。 */
    if (close(hSocket) == SOCKET_ERROR) {
      puts("ERROR: Could not close socket\n");
      return 0;
    }
  }
```

Linux では、Defer to Kernel デザインパターンにおけるクライアント部分の概要は以下のようになる。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>
#include <stdlib.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define SOCKET_PATH     "/tmp/myserver"

int  main(int argc, char* argv[]) {
  int hSocket;                  /* ソケットのハンドル */
  struct sockaddr_un Address;  /* Internet ソケットアドレス構造体 */
  char pBuffer[BUFFER_SIZE];
  unsigned nReadAmount;

  /* サーバとのやり取りに使用する UNIX ドメインソケットを作成する。 */
  hSocket=socket(AF_UNIX,SOCK_STREAM,0);
  if (hSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  /* アドレス構造体を埋めることで、UNIX ドメインソケットの作成方法を
      定義する。 */
  Address.sun_family=AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);

  /* 「既知」のパスを経由してホストに接続する。 */
  /* この簡単なサンプルでは、「既知」のパスをハードコードしている。 */
  if (connect(hSocket,(struct sockaddr*)&Address,sizeof(Address))
      == SOCKET_ERROR) {
      puts("\nCould not connect to host\n");
      return 0;
  }

  /* サーバに要求を通信する。 */
  /* .
     .
     . */

  /* サーバとの通信に使用したソケットを閉じる。 */
  if (close(hSocket) == SOCKET_ERROR) {
    puts("\nCould not close socket\n");
    return 0;
  }
}
```

Windows において Defer to Kernel パターンを実装する方法の詳細は「Securable Objects」[MSDN 2009a] を参照されたい。

2.3.9 既知の使用例

ucspi-unix

Windows セキュリティ保護可能オブジェクト（Securable Objects）

Copyright ©2009 JPCERT/CC All rights reserved.

3 設計レベルのパターン

3.1 Secure State Machine（セキュアステートマシン）

3.1.1 目的

Secure State Machine パターンの目的は、セキュリティ機能とユーザレベルの機能を 2 つの別々のステートマシンとして実装することにより、セキュリティとユーザレベルの機能の明確な分離を実現することにある。

3.1.2 別名

Secure State

3.1.3 動機

セキュアなシステムの実装において、セキュリティ機能と典型的なユーザレベル機能を混在させると、両者の複雑さが増す可能性がある。複雑さが増すと、実装されたセキュリティ特性のテスト、レビュー、および検証が難しくなり、脆弱性が作り込まれる可能性が高くなる。

また、セキュリティ機能とユーザレベルの機能の結び付きが強いと、システムのセキュリティメカニズムの変更および修正が難しくなる。

3.1.4 適用可能性

このパターンは、以下の場合に適用できる。

- ユーザレベルの機能が、Gang of Four による State パターン [Gamma 1995] を使用した実装に適している場合。つまり、ユーザレベルの機能を、有限状態マシンとして明確に表現できる。
- ユーザレベル機能のステートマシンにおける状態遷移に繋がる操作のアクセス制御モデルをステートマシンとして表現できる。単純なケースでは、ユーザレベル機能のステートマシンの状態遷移に繋がる操作に対するアクセス制御モデルを、状態が 1 つだけのステートマシンとして表せる。

3.1.5 構造

Secure State Machine パターンの構造を図 7 に示す。

Copyright ©2009 JPCERT/CC All rights reserved.

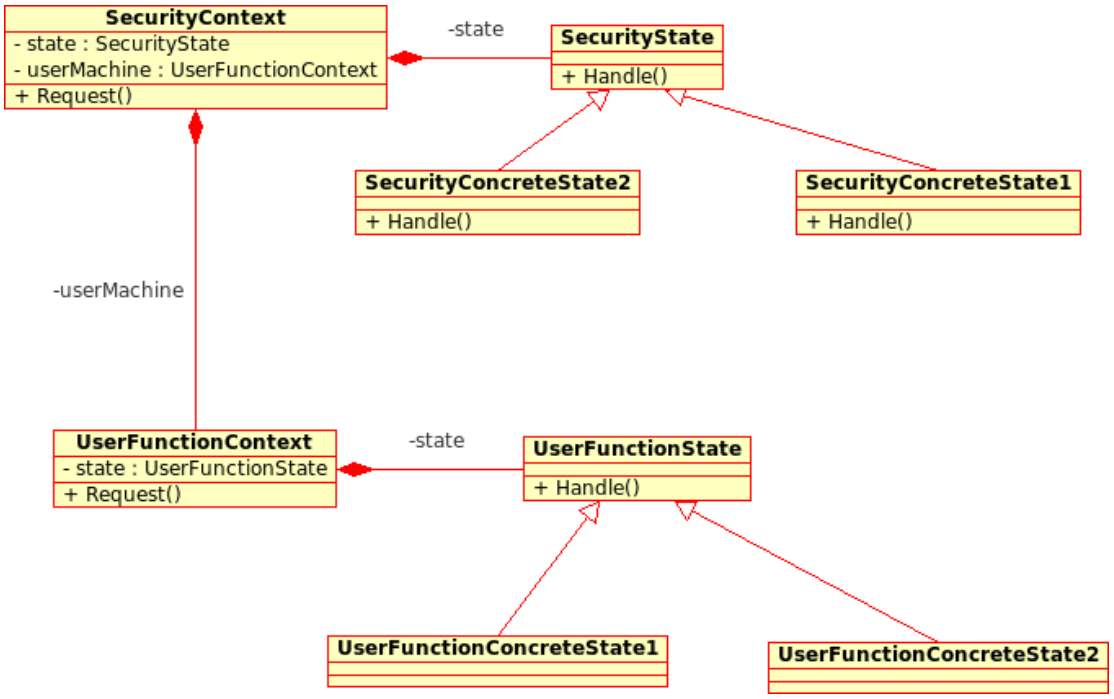

図 7:　　Secure State Machine パターンの構造

3.1.6　関連要素

Secure State Machine パターンの関連要素は以下のとおりである（サンプルコードとの対応を、関連要素に続けて括弧で囲んで示す）。

- **SecurityContext**（ExampleSystem）
  - クライアント向けのインターフェースを定義する。すべてのクライアントの操作は、はじめに SecurityContext のインスタンスによって処理される。
  - Gang of Four によるオリジナルの State パターン [Gamma 1995] と同様に、SecurityContext は、現在の状態をセキュリティの観点から定義する SecurityState サブクラスのインスタンスを維持する。
  - UserFunctionContext は、セキュリティ以外のユーザレベル機能を実装するステートマシンのインスタンスを維持する。
  - UserFunctionContext のインスタンスのプロキシとしての機能を果たす。
- **SecurityState**（SecurityState）
  - セキュリティステートマシンが取り扱う操作インターフェースを定義する。UserFunctionState は同じインターフェースを共有しなければならない点に注意すること。つまり、UserFunctionState は SecurityState と同じ操作を取り扱わなければならない。
- **SecurityConcreteState**（NotLoggedIn、LoggedInAdmin、LoggedInClerk、Locked）
  - SecurityState のサブクラスであり、それぞれ各操作のセキュリティ状態に依存する振る舞いを実装する。

ユーザレベル機能のステートマシンの構成要素は、Gang of Four による State パターンの構成要素とまったく同じである。

Copyright ©2009 JPCERT/CC All rights reserved.

- **UserFunctionContext**（UserFunctionsMachine）
  - セキュリティステートマシンの構成要素が必要に応じてユーザレベル機能のステートマシンに処理を転送できるように、SecurityContext と同じ操作をすべて定義する。
  - ユーザレベル機能への外部からのアクセスを防止するために、private コンストラクタを有する。SecurityContext だけが新規の UserFunctionContext を作成できる。
- **UserFunctionState**（UserFunctionState）
  - SecurityState と同じインターフェースを持つ。
- **UserFunctionConcreteState**（UserFunctionConcreteState1、UserFunctionConcreteState2）
  - SecurityConcreteState の場合と同様に、SecurityState のサブクラスはそれぞれ各操作のユーザレベルの状態に依存する振る舞いを実装する。

3.1.7 結果

一般的な State パターンに関連付けられている一連の結果に加えて、Secure State Machine パターンは、以下の追加の結果をもたらす。

- *セキュリティ機能をユーザレベル機能と明確に分離する。*このパターンを使用するには、セキュリティ機能をセキュリティステートマシンに明示的に実装し、システムのユーザ機能をユーザレベル機能ステートマシンに明示的に実装する必要がある。これにより、以下の効果が期待できる。
  - セキュリティ機能をユーザレベル機能とは別にテストしたり検証し易い。セキュリティ機能がユーザレベル機能と別々に実装されるため、ユーザレベル機能ステートマシンよりも厳格なテストと検証の技法をセキュリティステートマシンに適用できる。
  - セキュリティ機能を変更したり置き換えたりし易い。セキュリティ機能がユーザレベル機能と分離しているため、既存のセキュリティ機能がユーザレベル機能と混在している場合に比べて、より少ない労力で新しいセキュリティ機能を実装できる。
- *ユーザレベル機能へのセキュリティを回避したアクセスを防止する。*ユーザレベル機能ステートマシンのインスタンスを作成できるのはセキュリティステートマシンだけであるため、ユーザレベル機能ステートマシンとのやり取りのすべては、まずセキュリティステートマシンを経由しなければならず、結果としてユーザレベル機能を直接利用するような攻撃を防ぐことができる。

3.1.8 実装

Gang of Four による State パターン [Gamma 1995] で挙げられている実装上の考慮事項に加えて、Secure State Machine パターンは、以下に挙げる実装上の事項を考慮している。

ユーザレベル機能のステートマシンに誰が操作を転送するのか？ セキュリティステートマシンによって取り扱われる操作要求は、SecurityContext インスタンス、または SecurityConcreteState インスタンスによってユーザレベル機能のステートマシンへ転送することが許される。

- SecurityContext インスタンス ：SecurityContext インスタンスは、SecurityState 内に定義された操作を転送すべきかどうかを示すブール値を返す操作メソッドを用いて、ユーザレベル機能のステートマシンへの操作転送を処理する。SecurityContext インスタンスは、この戻り値を使用して、操作を転送するかどうかを決める。この方法は、サンプルコー

Copyright ©2009 JPCERT/CC All rights reserved.

ドの中で使用している。また、この方法はユーザレベル機能のステートマシンを SecurityContext インスタンス内に完全に隠蔽できるので、次に示す SecurityConcreteState インスタンスにより転送を実施する方法よりも推奨される方法である。

- SecurityConcreteState インスタンス ： SecurityConcreteState が SecurityContext がユーザレベル機能のステートマシンにアクセスできるメソッドを提供する場合、SecurityConcreteState は操作を直接転送できる。

3.1.9 サンプルコード

ここで紹介する Secure State Machine パターンを使用するサンプルは、以下のように振る舞うシステムを実装するためのコードのスケルトンを提供する。

- ユーザがシステムを使用するには、まずログインする必要がある。
- ログインに 5 回失敗すると、ユーザのアカウントはロックされる。
- 各ユーザは個別のステートマシンによって取り扱われる。ステートマシンへのユーザの割り当ては、システムの他の部分が行う。
- ユーザレベル機能は、`op1`、`op2`、`op3`、`login` および `log-out` として抽象的に表している。
- セキュリティ上の理由により、op3 は 1 つのセッションで 50 回だけ実行できる。`op3` の実行が 50 回を上回ると、ユーザは自動的にログアウトさせられる。
- `op2` を実行するには、ユーザが管理者（administrator）の役割を有していなければならない。それ以外のユーザは事務員（clerk）としての役割を有する。

サンプルコードの基本的な振る舞いを図 8 のコラボレーション図に示す。

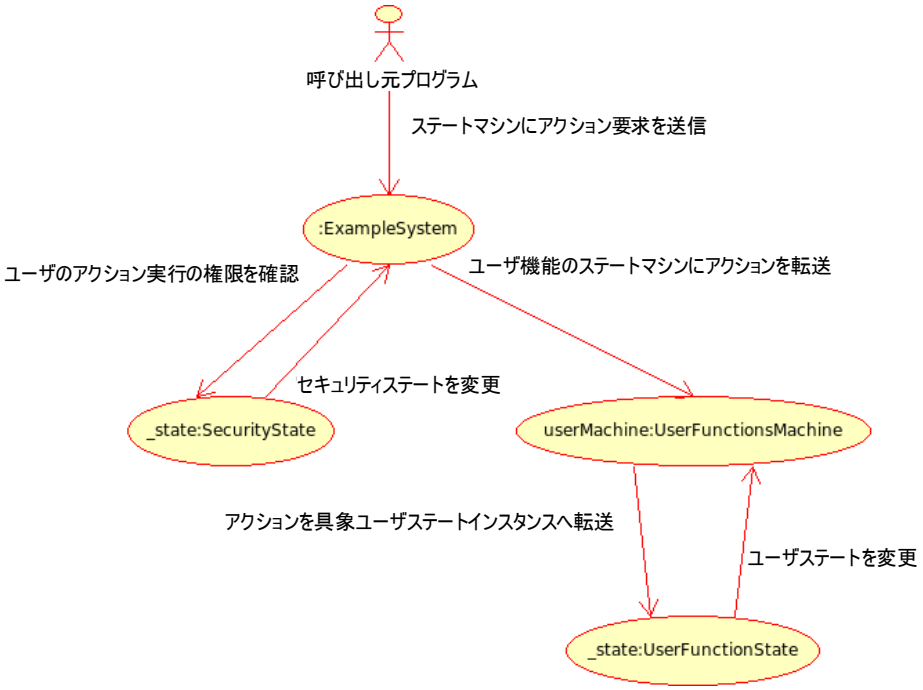

図 8: Secure State Machine サンプルコードのコラボレーション図

このクラスは、ステートマシンに関連付けられているユーザの資格情報を表す。このクラスのサンプルは示していない。

Copyright ©2009 JPCERT/CC All rights reserved.

```
class UserCredentials;
```

このクラスは、暗号化された文字列を表す。このクラスのサンプルは示していない。

```
class EncryptedString;
```

これは、セキュリティステートマシンの各状態を表す前方宣言抽象クラスである。このクラスは、サンプルで概略を示す。

```
class SecurityState;
```

このクラスは、システムのセキュリティステートマシンを実装し、実際にユーザレベル機能を実装するステートマシンのプロキシとしての機能を果たす。

```
class ExampleSystem {

public:

  // 指定されたユーザのために、サンプルシステムのステートマシンを新規作成する。
  ExampleSystem(UserCredentials user);

  // 誰かが現在のユーザとしてシステムにログオンを試みている。
  void login(EncryptedString password);

  // ユーザはログアウトしている。
  void logout();

  // ユーザが 3 つのユーザレベルシステム操作のいずれか 1 つを
  // 実行しようとしている。
  void op1();
  void op2();
  void op3();

private:

  // セキュリティステートマシンの現在の状態を維持する。
  SecurityState* _state;

  // 制御元の現在の状態を変更する。
  void changeState(SecurityState*);

  // セキュリティステートマシンに、制御元の現在の状態を
  // 変更させる。
  friend class SecurityState;

  // セキュリティステートマシンに関連付けられているユーザを維持する。
  UserCredentials _user;

  // セキュリティステートマシンに関連付けられているユーザを取得する。
  const UserCredentials getUser();

  // ユーザ機能を実際に実装するステートマシンを維持する。
  // セキュリティステートマシンは、ユーザレベル機能のステートマシンのプロキシとしての
  // 機能を果たす。
  UserFunctionsMachine userMachine;
};
```


Copyright ©2009 JPCERT/CC All rights reserved.

SecurityContext のメソッドは、以下のように定義されている。

```
ExampleSystem::ExampleSystem(UserCredentials user) {

  // 初期状態は、ユーザはログアウトしている。
  _state = NotLoggedIn::instance(user);

  //ユーザを保存しておく必要がある。
  _user = user;

  // プロキシとして機能を果たす対象となるユーザレベル機能のステートマシンを作成する。
  userSystem = UserFunctionsMachine(user);
}

void ExampleSystem::login(EncryptedString password) {
  // 適切であれば、操作を転送する。
  if (_state->login(this, password)) {
    userMachine->login(password);
  }
}

void ExampleSystem::logout() {
  // 適切であれば、操作を転送する。
  if (_state->logout(this)) {
    userMachine->logout();
  }
}

void ExampleSystem::op1() {
  // 適切であれば、操作を転送する。
  if (_state->op1(this)) {
    userMachine->op1();
  }
}

void ExampleSystem::op2() {
  // 適切であれば、操作を転送する。
  if (_state->op2(this)) {
    userMachine->op2();
  }
}

void ExampleSystem::op3() {
  // 適切であれば、操作を転送する。
  if (_state->op3(this)) {
    userMachine->op3();
  }
}
```

これは、セキュリティステートマシンの各状態を定義する状態クラスのためのインターフェースを定義する抽象クラスの宣言である。

```
class SecurityState {

public:
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  virtual bool login(ExampleSystem* controller, EncryptedString password);
  virtual bool logout(ExampleSystem* controller);
  virtual bool op1(ExampleSystem* controller);
  virtual bool op2(ExampleSystem* controller);
  virtual bool op3(ExampleSystem* controller);

protected:

  void changeState(ExampleSystem* controller, SecurityState* newState);
};
```

このサンプルのセキュリティモデルには、4 つの状態がある。

- NotLoggedIn： ユーザがログインしていない。
- LoggedInAdmin： ユーザが管理者としてログインしている。
- LoggedInClerk： ユーザが事務員としてログインしている。
- Locked： ユーザのアカウントがロックされている。

セキュリティステートのメソッド群のデフォルトの実装は以下のとおりである。これらのメソッド定義は、すべて具象ステートクラスで定義しなおさなければならない。以下のデフォルトの実装では、操作はユーザレベルステートマシンに転送されることがない。

```
bool SecurityState::login(ExampleSystem* controller,
                          EncryptedString password) { return false; }
bool SecurityState::logout(ExampleSystem* controller) { return false; }
bool SecurityState::op1(ExampleSystem* controller) { return false; }
bool SecurityState::op2(ExampleSystem* controller) { return false; }
bool SecurityState::op3(ExampleSystem* controller) { return false; }
```

`changeState()`は、すべての具象ステートクラスに共通である。

```
void SecurityState::changeState(ExampleSystem* controller,
                                SecurityState* newState) {
  controller->changeState(newState);
}
```

以下は、`NotLoggedIn` 具象ステートクラスの定義である。

```
class NotLoggedIn : public SecurityState {

public:

  // 現在のユーザのためにこの状態のインスタンスを取得する。ユーザはそれぞれ
  // 自身に関連付けられた各セキュリティ状態のインスタンスを 1 つだけ
  // 持つ。これにより、各ユーザがそれぞれただ一つのセキュリティステートマシンに
  // 関連付けられることが保証される。
  static SecurityState* instance(UserCredentials user);

  // ユーザは、ログインしていない場合、できる唯一のことはログインを試みることである。
  virtual bool login(ExampleSystem* controller, EncryptedString password);

private:

  // ログインの試みが失敗した回数を追跡する。
  unsigned int numFailedLogins;
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
};
```

以下は、NotLoggedIn ステートクラスのメソッドの本体である。

NotLoggedIn ステートを作成する。ログインの試みが失敗した回数を初期化する。

```
NotLoggedIn::NotLoggedIn() {
  numFailedLogins = 0;
}
```

ユーザログインを取り扱う。

```
bool NotLoggedIn::login(ExampleSystem* controller, EncryptedString password) {

  // パスワードを使ってユーザの認証を行う。
  if (controller->getUser().validate(password)) {

    // 現在のユーザが自身のパスワードを正しく入力した。

    // 不正パスワードの数をクリアする。
    numFailedLogins = 0;

    // ユーザはログインした状態である。ユーザの役割に応じて適切なログイン状態を
    // 選択する。
    if (controller->getUser().isAdministrator()) {
      changeState(controller, LoggedInAdmin::instance());
    }
    else {
      changeState(controller, LoggedInClerk::instance());
    }

    // ユーザはログインした状態である。ログイン操作をユーザレベル機能のステートマシンに
    // 引き渡すことでログインに関連付けられているユーザ機能を取り扱う。
    return true;
  }

  else {
    // 現在のユーザが正しくないパスワードを入力した。

    // 失敗したログインを追跡する。
    numFailedLogins++;

    // ユーザによるログインの失敗回数は多すぎるか。
    if (numFailedLogins >= 5) {

      // 失敗ログインの回数をリセットする。
      numFailedLogins = 0;

      // ユーザのアカウントをロックする。
      changeState(controller, Locked::instance());

      // セキュリティステートマシンが、セキュリティ要件が満たされていないと
      // 判断したため、ログイン操作はユーザレベル機能のステートマシンに引き渡されない。
      return false;
    }
  }
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
}
```

以下は、Locked 具象ステートクラスの定義である。

```
class Locked : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);

  // この単純なサンプルでは、ユーザのアカウントはひとたびロックされると
  // ロックは解除できない。ユーザのアカウントがロックされると、ユーザは
  // 何もできない。ユーザレベル機能のステートマシンには、いかなる操作も転送されない。
};
```

以下は、LoggedInAdmin 具象ステートクラスの定義である。

```
class LoggedInAdmin : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInAdmin();

  // ログイン済みの管理者は、再度ログインすること以外のあらゆる操作を
  // 実行できる。
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op2(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // ユーザが op3 を実行した回数を追跡する。
  unsigned int op3Count;
};
```

以下は、LoggedInAdmin ステートクラスのメソッドの本体である。

```
// NotLoggedIn ステートを作成する。これによって、op3 が実行された回数のカウントが
// 初期化される。
LoggedInAdmin::LoggedInAdmin() {
  op3Count = 0;
}

bool LoggedInAdmin::logout(ExampleSystem* controller) {

  // ログアウトステートに移行する。
  changeState(controller, NotLoggedIn::instance());

  // ログアウト操作のためのユーザ機能アクションを処理する。
  return true;
}

bool LoggedInAdmin::op1(ExampleSystem* controller) {
  // セキュリティステートマシンの現在の状態に基づき、この操作が有効である
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  // ことは分かっている。操作をユーザレベル機能のステートマシンに転送する。
  return true;
}

bool LoggedInAdmin::op2(ExampleSystem* controller) {
  // セキュリティステートマシンの現在の状態に基づき、この操作が有効である
  // ことは分かっている。操作をユーザレベル機能のステートマシンに転送する。
  return true;
}

bool LoggedInAdmin::op3(ExampleSystem* controller) {

  // ユーザが op3 を再度実行した。これを追跡する。
  op3Count++;

  // ユーザは、op3 を実行できる上限の回数を超過したか。
  if (op3Count > 50) {

    // op3 をこのログインセッションで実行した回数のカウントをリセットする。
    op3Count = 0;

    // ユーザをログアウトさせる。これは、制御元の logout メソッドを呼び出す
    // ことに注意すること。結果として、セキュリティステートマシンとユーザレベル機能
    // ステートマシンの両方がログアウト操作を処理することになる。
    controller->logout();

    // op3 操作の処理をやめる。
    return false;
  }

  // ここに到達した場合には、op3 のセキュリティ条件を満たしたという
  // ことである。op3 をユーザ機能ステートマシンに転送する。
  return true;
}
```

以下は、LoggedInClerk 具象ステートクラスの定義である。

```
class LoggedInClerk : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInClerk();

  // ログイン済みの事務員は、再度ログインすることと op2 以外のあらゆる
  // 操作を実行できる。
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // ユーザが op3 を実行した回数を追跡する。
  unsigned int op3Count;
};
```


Copyright ©2009 JPCERT/CC All rights reserved.

以下は、LoggedInClerk ステートクラスのメソッドの本体である。

```
// LoggedInClerk 状態を作成する。これによって、op3 が実行された
// 回数のカウントが初期化される。
LoggedInClerk::LoggedInClerk() {
  op3Count = 0;
}

bool LoggedInClerk::logout(ExampleSystem* controller) {

  // 単にログアウト状態に移行する。
  changeState(controller, NotLoggedIn::instance());

  // ログアウト操作のためのユーザ機能アクションを処理する。
  return true;
}

bool LoggedInClerk::op1(ExampleSystem* controller) {
  // セキュリティステートマシンの現在の状態に基づき、この操作が有効である
  // ことは分かっている。操作をユーザレベル機能のステートマシンに転送する。
  return true;
}

bool LoggedInClerk::op3(ExampleSystem* controller) {

  // ユーザが op3 を再度実行した。これを追跡する。
  op3Count++;

  // ユーザは、op3 を実行できる上限の回数を超過したか。
  if (op3Count > 50) {

    // op3 をこのログインセッションで実行した回数のカウントを
    // リセットする。
    op3Count = 0;

    // ユーザをログアウトさせる。これは、制御元の logout メソッドを呼び出すことに
    // 注意すること。結果として、セキュリティステートマシンとユーザレベル機能
    // ステートマシンの両方がログアウト操作を処理することになる。
    controller->logout();

    // op3 の処理をやめる。
    return false;
  }

  // ここに到達した場合には、op3 のセキュリティ条件を満たしたという
  // ことである。op3 をユーザレベル機能のステートマシンに転送する。
  return true;
}
```

以下は、ユーザレベル機能のステートマシンのクラスである。ユーザレベル機能のステートマシンのインスタンスを作成できるのは、セキュリティステートマシンだけである点に注意する。

```
class UserFunctionsMachine {
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
public:

  // 誰かが現在のユーザとしてシステムにログオンを試みている。
  void login(EncryptedString password);

  // ユーザはログアウトしている。
  void logout();

  // ユーザが 3 つのユーザレベルシステム操作のいずれか 1 つを
  // 実行しようとしている。
  void op1();
  void op2();
  void op3();

private:

  // ユーザレベル機能ステートマシンのインスタンスを作成できるのは、
  // セキュリティステートマシンだけである。これは、ユーザレベル機能ステートマシンへの
  // 直接のアクセスを防止するのに役立つ。
  friend class ExampleSystem;

  // 指定されたユーザのために、ユーザ機能ステートマシンを新規作成する。
  UserFunctionsMachine(UserCredentials user);
};
```

3.1.10 既知の使用例

「Method and apparatus for secure context switching in a system including a processor and cached virtual memory（プロセッサおよびキャッシュされた仮想記憶を含んだシステムにおけるセキュアなコンテキストスイッチのための方法と装置　―　米国特許出願番号 20070260838）

3.2 Secure Visitor（セキュアビジター）

3.2.1 目的

セキュアなシステムでは、各ノードのアクセス制限が異なる可能性のある階層型の構造化データを対象に、さまざまな操作を行う必要が生じる可能性がある。つまり、異なるノードへのアクセスが、データにアクセスするユーザの役割／資格情報に依存する可能性がある。Secure Visitor パターンは、訪問者（ビジター）がノードの*ロックを解除*するための適切な資格情報を提示しない限り、読み取られないようにノードが自身を*ロック*する。Secure Visitor パターンは、ロックされているノードにアクセスする唯一の方法が、ビジターとしてアクセスするように定義されており、データ構造内のノードへの不正なアクセスの防止に寄与する。

3.2.2 動機

Secure State Machine パターンと同様に、Secure Visitor パターンを実装する主たる動機は、セキュリティ上考慮すべき事項とユーザレベルの機能とを明確に切り分けることである。Secure Visitor パターンは、セキュリティ上考慮すべきすべての事項をデータ階層内のノードに割り当てることで、ユーザレベルの機能のみを考慮するビジターを開発者が自由に作成できるようにする。

Copyright ©2009 JPCERT/CC All rights reserved.

データ階層内のノードにセキュリティ機能のすべての責任を負わせることで、セキュリティ機能をより厳格にテストしたり検証することが可能になる。また、ユーザ機能の開発者が、新しいビジターが開発されるたびにセキュリティ機能を実装する必要性から解放され、新しいセキュリティホールが作り込まれることを回避する。

3.2.3 適用可能性

このパターンは、以下の場合に適用できる。

- システムが、Gang of Four による Visitor パターン[Gamma 1995]を使って処理できる階層型データを保有している
- 階層型データ内のさまざまなノードが異なるアクセス制限を持っている

3.2.4 構造

Secure Visitor パターンの構造を図 9 に示す。

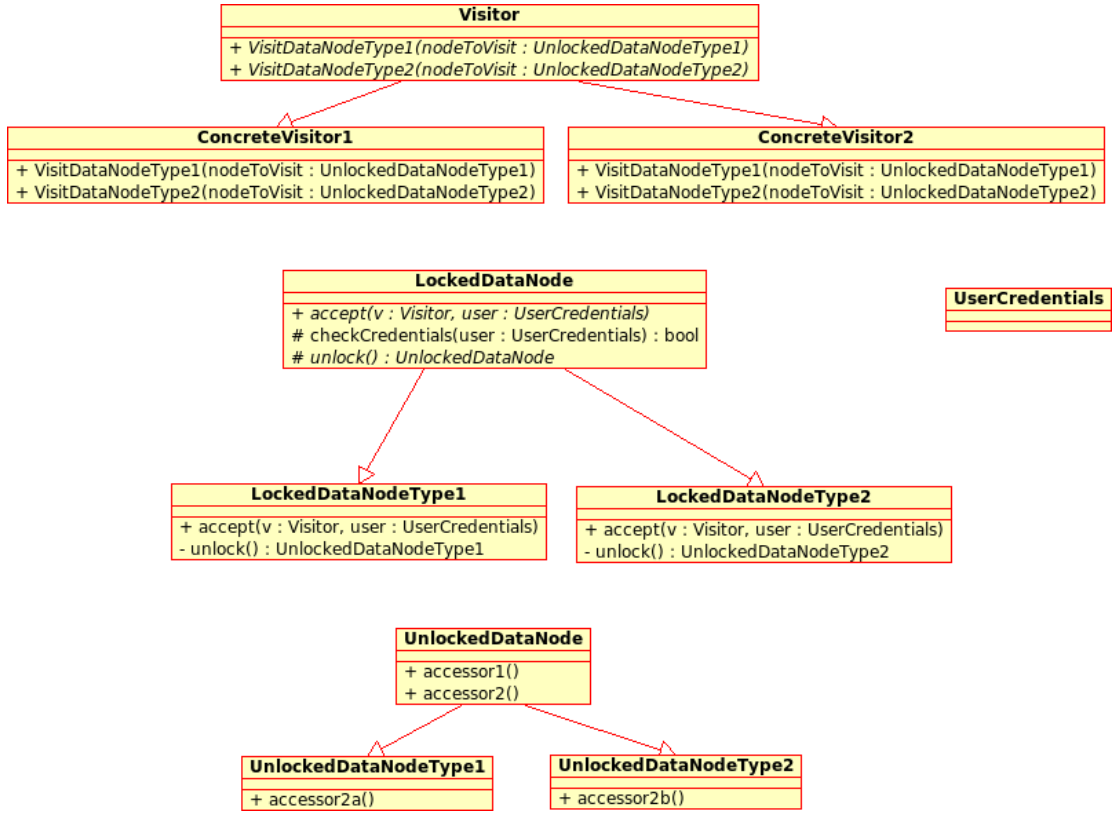

図 9: Secure Visitor パターンの構造

3.2.5 関連要素

Secure Visitor パターンの関連要素は以下のとおりである（サンプルコードとの対応を、関連要素に続けて括弧で囲んで示す）。

- **Visitor**（HierarchicalDataVisitor）：Secure Visitor パターンにおける Visitor は、標準の Visitor パターンにおける Visitor とほぼ同じである。パターン間の主な違いは、Secure Visitor パターンの各種の visit メソッドがロックされていないノードを受け取るのに対

Copyright ©2009 JPCERT/CC All rights reserved.

し、標準の Visitor パターンの visit メソッドが単純にノードオブジェクトを受け取る点である（標準の Visitor パターンには、ロックされたデータノードおよびロックされていないデータノードという概念がない）。

- **ConcreteVisitor**：標準の Visitor パターンと同様に、ConcreteVisitor クラスは、Visitor 抽象クラスに定義されている操作を実装する。
- **LockedDataNode**（LockedDataNode）：LockedDataNode クラスは、ビジターを受け付ける `accept()` 操作を定義する。さらに、LockedDataNode クラスは、ユーザの資格情報を検査する操作および現在ロックされているノードのロックを解除する操作も定義する。ロックされているノードは、ノード内のデータを閲覧したり変更したりするための public な操作を提示しない。ノードへのアクセスはすべて、ノードの `accept()` 操作を通じて行う必要がある。`accept()` 操作は、ユーザの資格情報を検査する。資格情報が、現在のノードのデータをそのユーザが閲覧できる有効な資格情報であれば、ノードは `unlock()` 操作を使用して自身のロックを解除し、ロック解除したバージョンのノードをビジターの visit メソッドに渡す。
- **LockedDataNodeTypeN**（LockedNodeType1）：LockedDataNodeTypeN クラスは、LockedDataNode 抽象クラスに定義されている操作を実装する。これには、各種のロックされているノードオブジェクトのロックを解除し、ロック解除したバージョンのノードを返す `unlock()` 操作が含まれる。
- **UnlockedDataNode**（UnlockedDataNode）：このクラスは、ロックされているデータノードのロック解除したバージョンを表す。ロック解除されたノードには、いくつかの重要な特性がある。
  - 親および子ノードにアクセスできない。ロック解除されたノード固有のデータのみ、つまり、ユーザが閲覧することを許可されたデータのみが含まれている。
  - `accept()` 操作を持たない。セキュアビジターによる階層型データ構造の横断は、ロック解除されたノードではなく、ロックされているノードを対象に行われる。
- **UnlockedDataNodeTypeN**（UnlockedNodeType1）：UnlockedDataNode の一連の具象実装は、抽象クラスに定義されている操作を実装する。
- **UserCredentials**： UserCredentials は、システムの現在のユーザおよびそのユーザに割り当てられている操作権限許可のいずれかまたは両方を表す。Secure Visitor パターンでは、ユーザの資格情報の具体的な実装に対して多くの制約は課さない。唯一の要求事項は、ノードが資格情報を使用してノードのデータへのアクセスを制御できることである。

3.2.6 結果

標準の Visitor パターンに関連付けられている一連の結果に加えて、Secure Visitor パターンは、以下の結果をもたらす。

- *セキュリティ機能をユーザレベル機能と明確に分離する。*このパターンを使用すると、ビジター自身ではなく、データ階層内のノードがセキュリティの実装を要求される。以下の効果が期待できる。
  - セキュリティの側面をユーザレベル機能とは別々にテストおよび検証がし易い。セキュリティ機能がユーザレベル機能と別々に実装されるため、ユーザレベル機能ステートマシンよりも厳格なテストと検証の技法をセキュリティステートマシンに適用できる。

Copyright ©2009 JPCERT/CC All rights reserved.

  - セキュリティ機能を変更したり置き換えたりし易い。セキュリティ機能は、各種のビジターではなく、データ階層内のノードに実装されるため、ビジターに変更を加えずにセキュリティ機能の変更が可能である。
- *ユーザレベル機能へのセキュリティを回避したアクセスを防止する。*データ階層内のロックされているノードにアクセスする唯一の方法が Visitor パターンの `accept()` メソッドを通じた方法であり、ロック解除されたバージョンのノードの作成が許されている唯一のクラスが、ロックされているノードであるため、そのノードに対して有効な資格情報を渡さずにノードのデータにプログラムからアクセスするのは困難もしくは不可能である。

3.2.7 実装

標準の Visitor パターンで挙げられている実装上の考慮事項に加えて、Secure Visitor パターンは、以下に挙げる実装上の事項を考慮している。

ロックされているノード内のデータをどのように保護するか？ Secure Visitor デザインパターンの目的は、ロックされているノード内の情報を、ノードに対する適切な資格情報を提供せずに読み取ることを困難にすることである。パターンそのものによって、ロックされているノードのデータを適切な資格情報なしでプログラムから読み取ることは困難になるが、それでもロックされているノードを構成する各バイトデータが直に読み取られ、結果として、ロックされているノード内のデータがアクセスされる可能性はある。これは、ロックされているノード内のデータが、何らかの方法で実際に「ロック（錠）を掛けられる」必要があることを示唆する。ロックされているノード内のデータは、暗号化またはオフラインのストレージを使用することでロックできる。

- **暗号化**：ロックされているノードのデータを暗号化しておき、ビジターの資格情報を受け入れた後に、ロック解除されたバージョンのノードを作成する過程でのみ復号することを許す。
- **オフラインストレージ**：ロックされているノード内の実際のデータを、データベースのような、何らかの保護された外部のデータ管理システムに保管する。ノードの実際のデータは、ビジターの資格情報を受け入れた後にのみ、外部ソースから読み込まれる。

3.2.8 サンプルコード

以下のコラボレーション図に、提示するサンプルコードの基本的な振る舞いを表す。

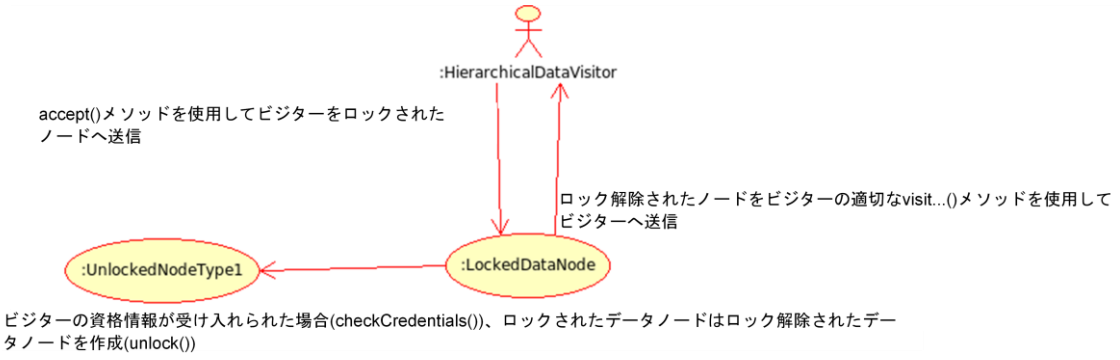

図 10: Secure Visitor サンプルコードのコラボレーション図

Copyright ©2009 JPCERT/CC All rights reserved.

このクラスは、ビジターに関連付けられているユーザの資格情報を表す。つまり、このビジターは、渡された資格情報が示すユーザが要求した何らかのアクションへの応答として、データにアクセスしているのである。このクラスのサンプルは示されない。

```
class UserCredentials;
```

これは、ロックされているノードのロック解除したバージョンの前方宣言である。このクラスは、サンプルの後半で定義する。

```
class UnlockedDataNode;
```

これは、階層型データ内のロックされているノードのビジターの前方宣言である。このクラスは、サンプルの後半で定義する。

```
class HierarchicalDataVisitor;
```

これは、Secure Visitor パターンにおけるロックされたノードのインターフェースを定義する。標準の Visitor パターンにおけるインターフェースによく似ている。

```
class LockedDataNode {

public:
  virtual void accept(HierarchicalDataVisitor& visitor,
                      UserCredentials user);

private:

  // それぞれのタイプのノードには、ユーザにノードへのアクセス権があるかどうか、
  // ビジターの資格情報を確認するための何らかの方法がある。
   virtual bool checkCredentials(UserCredentials user);
};
```

Secure Visitor におけるビジターインターフェースは、標準の Visitor パターンにおけるビジターインターフェースによく似ている。唯一の違いは、各種の visit...() メソッドが、ロックされているノードではなく、ロック解除されたバージョンのノードを受け付ける点である。

```
class HierarchicalDataVisitor {

public:

  virtual ~HierarchicalDataVisitor()
  virtual void visitNodeType1(UnlockedNodeType1 *node);

protected:

  HierarchicalDataVisitor();
};
```

データ階層内の各具象ノードは、ロックされているバージョンとロック解除されているバージョンの両方を持つ。ロックされたノードのみがロック解除されているノードを作成できる。

```
class LockedNodeType1 : LockedDataNode {

public:
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  // Secure Visitor パターンにおいてデータ階層内のデータにアクセスする
  // ための唯一の方法は、ノードへのビジターと現在のユーザの資格情報を
  // 受け取る accept()メソッドを使用することである。
  void accept(HierarchicalDataVisitor& visitor,
              UserCredentials user);

private:

  // ロックされているノードは、ビジターの資格情報を受け付けると、
  // ビジターに処理のために渡す、自身のロック解除されたバージョンを作成する。
  // UnlockedDataNode を作成できるのは、LockedDataNode だけである。
  UnlockedNodeType1 unlock();

  // それぞれのタイプのノードには、ユーザにノードへのアクセス権があるかどうか、
  // ビジターの資格情報を確認するための何らかの方法がある。
  bool checkCredentials(UserCredentials user);

  // 何らかの方法でノードの子を追跡する...
  // ...
};
```

データ階層内のロックされているノードの accept メソッドはユーザの資格情報を確認し、その資格情報がノードに対して有効であればノードのロックを解除し、ビジターに渡す。

```
void LockedNodeType1::accept(HierarchicalDataVisitor& visitor,
                             UserCredentials user) {

  // このノードに対して資格情報は有効か？
  if (checkCredentials(user)) {

    // ユーザは、このノードに対するアクセス権を持っている。
    // ノードのロックを解除して、それをビジターに渡す。
    visitor.visitNodeType1(unlock());
  }

  // ノードの子にアクセスする...
  // ...
}
```

ロック解除されたノードのコンストラクタは private であり、対応するロックされたクラスはその friend クラスである点に注意すること。これは、ロック解除されたノードが、ロックされているノードによってのみ作成できることを意味する。

```
class UnlockedNodeType1 {

public:

  . . . データアクセスメソッド、その他 . . .

private:
  UnlockedNodeType1();
  friend class LockedNodeType1;
}
```


Copyright ©2009 JPCERT/CC All rights reserved.

4 実装レベルのパターン

4.1 Secure Directory（セキュアディレクトリ）

4.1.1 目的

Secure Directory パターンの目的は、プログラムの実行中に、そのプログラムが使用しているファイルを攻撃者が操作できないようにすることである。この問題に関する追加情報については、『*The CERT C Secure Coding Standard*』[Seacord 2008]の「FIO15-C. Ensure that file operations are performed in a secure directory（ファイル操作は必ず安全なディレクトリで実行すること）」を参照されたい。

4.1.2 動機

プログラムはその実行中のかなりの時間、ファイルに依存する可能性がある。プログラム開発者は通常、プログラムが実行中に使用するファイルは、プログラムが実行されている間は外部ユーザに操作されることはないと想定する。しかし、この前提が間違っていた場合、ファイルは複数のユーザに変更される可能性があり、プログラムがファイルに変更がないことを前提とした重要な処理タイミングで、悪意を持ったユーザによるファイルの変更または削除操作の影響を受け、競合状態を引き起こす。

Secure Directory パターンは、プログラムが使用するファイルの保存ディレクトリを、プログラムのユーザのみが書き込み（場合によって、読み取り）できるように保証する。

4.1.3 適用可能性

Secure Directory パターンは以下の条件を満たす場合に適用できる。

- プログラムが安全でない環境で実行される。つまり、プログラムが使用するファイルシステムに悪意を持ったユーザがアクセス可能な環境である。
- プログラムがファイルの読み取りまたは書き込み、あるいはその両方を行う。
- プログラムの実行中にプログラムが読み取り、または書き込みを行うファイルが外部ユーザに変更された場合、プログラムの実行に悪影響を及ぼす可能性がある。

4.1.4 構造

プログラミングとしては、Secure Directory パターンの構造は比較的シンプルである。読み取り、または書き込みのためにファイルを開く前に、Secure Directory パターンはプログラムに次のことを実行させる。

1. ファイルのディレクトリパス名を正規化する（4.2「Pathname Canonicalization（パス名正規化）」を参照）。
2. 正規のパス名で参照したディレクトリがセキュアであることを確認する。

セキュアディレクトリの構造では、ディレクトリの書き込み許可がプログラムの実行ユーザとスーパーユーザに限定される。他のどのユーザもセキュアディレクトリのファイルは変更できない。さらに、対象のディレクトリの前に現れるすべてのディレクトリも、他のユーザがセキュアディレクトリの名前の変更や削除を行えないように防がなければならない。

Copyright ©2009 JPCERT/CC All rights reserved.

4.1.5 関連要素

Secure Directory パターンの関連要素は以下のとおりである。

- ファイルの読み書きを行うプログラム
- ファイルシステム

4.1.6 結果

Secure Directory は、異なるユーザにより制御されるプログラム間で発生し得る競合状態の可能性を減少させる。セキュアディレクトリ内で発生し得る競合状態は、ある特定のユーザの制御のもとで実行される複数のプログラム間でのみ発生する可能性がある。

パス名の正規化とセキュアディレクトリの確認のため、プログラムの速度は低下する。セキュアディレクトリの確認のためのオーバーヘッドを減らすために、特定のディレクトリのセキュリティの確認結果をキャッシュすることが可能である。なお、セキュアディレクトリの確認結果のキャッシュは、プログラムで使用するディレクトリへのアクセス許可がプログラム実行中には変更されないことが前提である。

4.1.7 実装

プログラムは、root 権限で実行されない限り、セキュアディレクトリを作成することはできない。したがって、プログラムは提供されるディレクトリがセキュアであることを確認し、そうでない場合は使用を拒否すべきである。構造の項で説明したように、Secure Directory パターンの基本実装には次の手順を実行する。

1. 読み取りまたは書き込みを行うファイルのディレクトリの正規パス名を取得する。（4.2「Pathname Canonicalization（パス名正規化）」を参照）。
2. 正規のパス名で参照したディレクトリがセキュア（プログラムの実行ユーザのみが書き込み、あるいは、読み込み可能）であることを確認する。
   - ディレクトリがセキュアであれば、ファイルの読み取りまたは書き込みを行う。
   - ディレクトリがセキュアでない場合、エラーを発行し、ファイルの読み書きは行わない。

4.1.8 サンプルコード

ここで提供するサンプルコードは、『*The CERT C Secure Coding Standard*』[Seacord 2008]の「FIO15-C. Ensure that file operations are performed in a secure directory（ファイル操作は必ず安全なディレクトリで実行すること）」から直接抜粋したものである。

POSIX 準拠の OS では、ディレクトリがセキュアであるかどうかを確認する関数は以下のように実装できる。

```
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <libgen.h>
#include <sys/stat.h>
#include <string.h>
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
/* ディレクトリがセキュアな場合、非 0 を返す、そうでない場合 0 を返す */
int secure_dir(const char *fullpath) {
  char *path_copy = NULL;
  char *dirname_res = NULL;
  char **dirs = NULL;
  int num_of_dirs = 0;
  int secure = 1;
  int i;
  struct stat buf;
  uid_t my_uid = geteuid();

  if (!(path_copy = strdup(fullpath))) {
    /* エラー処理を行う */
  }

  dirname_res = path_copy;
  /* ルートディレクトリまでどのくらい遠いかを計算する */
  while (1) {
    dirname_res = dirname(dirname_res);

    num_of_dirs++;

    if ((strcmp(dirname_res, "/") == 0) ||
   (strcmp(dirname_res, "//") == 0)) {
      break;
    }
  }
  free(path_copy);
  path_copy = NULL;

  /* dirs 配列を割り当て、値を埋める */
  if (!(dirs = (char **)malloc(num_of_dirs*sizeof(*dirs)))) {
    /* エラー処理を行う */
  }
  if (!(dirs[num_of_dirs - 1] = strdup(fullpath))) {
    /* エラー処理を行う */
  }

  if (!(path_copy = strdup(fullpath))) {
    /* エラー処理を行う */
  }

  dirname_res = path_copy;
  for (i = 1; i < num_of_dirs; i++) {
    dirname_res = dirname(dirname_res);

    dirs[num_of_dirs - i - 1] = strdup(dirname_res);

  }
  free(path_copy);
  path_copy = NULL;

  /* ルートディレクトリからリーフまで移動し、
   * その過程で許可を確認する */
  for (i = 0; i < num_of_dirs; i++) {
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
    if (stat(dirs[i], &buf) != 0) {
       /* エラー処理を行う */
    }
    if ((buf.st_uid != my_uid) && (buf.st_uid != 0)) {
      /* ディレクトリはユーザおよび root 以外が所有している */
      secure = 0;
    } else if ((buf.st_mode & (S_IWGRP | S_IWOTH))
      && ((i == num_of_dirs - 1) || !(buf.st_mode & S_ISVTX))) {
        /* 他のユーザがリーフディレクトリに対する許可を持っている
         * または途中のファイルの削除、または名前の変更が可能である */
        secure = 0;
     }

    free(dirs[i]);
    dirs[i] = NULL;
  }

  free(dirs);
  dirs = NULL;
  return secure;
}
```

secure_dir()関数を使用し、C では Secure Directory パターンは次のように実装できる。

```
char *dir_name;
char *canonical_dir_name;
const char *file_name = "passwd"; /* セキュアディレクトリ内のファイル名 */
FILE *fp;

/* dir_name の初期化 */
canonical_dir_name = realpath(dir_name, NULL);
if (canonical_dir_name == NULL) {
  /* エラー処理を行う */
}

if (!secure_dir(canonical_dir_name)) {
  /* エラー処理を行う */
}

if (chdir(canonical_dir_name) == -1) {
  /* エラー処理を行う */
}

fp = fopen(file_name, "w");
if (fp == NULL) {
  /* エラー処理を行う */
}

/*... ファイルの処理    */

if (fclose(fp) != 0) {
  /* エラー処理を行う */
}

if (remove(file_name) != 0) {
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  /* エラー処理を行う */
}
```

4.2 Pathname Canonicalization（パス名正規化）

4.2.1 目的

Pathname Canonicalization パターンの目的は、プログラムによって読み取り、書き込みが行われるすべてのファイルが、シンボリックリンクやショートカットを含まない正しいパス、つまり正規化されたパス名で参照されることを確実にすることである。

4.2.2 動機

シンボリックリンクや他のファイルシステム機能により、ファイルはパスが示すディレクトリに実際には存在しない可能性がある。したがって、文字列に基づいたパス名の確認の実行では間違った結果が生じる可能性がある。ディレクトリがセキュアであるかどうかを確認する際には、本当の正規パス名を使用することが特に重要になる。

4.2.3 適用可能性

次の条件をすべて満たす場合、Pathname Canonicalization パターンを適用できる。

- プログラムが信用できないソースからパス名を受け取る。
- アクセスすべきでないディレクトリやファイルへのパス名を、攻撃者が何らかの手段で隠ぺいし、システムに渡すことが可能である。
- プログラムが利用する各ファイルが一意の正規パス名を持つ。

4.2.4 構造

プログラム上、Pathname Canonicalization パターンの構造は、ファイルを開く前に、指定されたパス名を対象に OS 固有のパス名正規化関数を呼び出すことになる。正規化されたパス名はファイルを操作する際に使用される。

正規化されたパス名は、最後のファイル名を示す部分を除き、正規化されたパスのすべての要素が本物のディレクトリであり、リンクやショートカットではない。最後の要素は実ファイル名であり、リンクやショートカットではない。

4.2.5 関連要素

Pathname Canonicalization パターンの関連要素は以下のとおりである。

- ファイルを操作するプログラム
- ファイルシステム（OS 固有のパス名正規化機能の実装に依存）

4.2.6 結果

パス名を正規化することで、正規化したパス名の文字面を分析すれば正しい結果が得られるため、ファイルアクセスの正確性とセキュリティが向上する。

パス名の正規化により、プログラムの処理速度は低下する。正規化のためのオーバーヘッドを減らすために、正規化したパス名をキャッシュすることが可能である。キャッシュを行う

Copyright ©2009 JPCERT/CC All rights reserved.

には、プログラムがアクセスするディレクトリの構造がプログラム実行中には変更されないことが前提である。

4.2.7 実装

このパターンの実装の中核を成すのは、パス名の正規化を実行する OS 固有の関数である。正規化関数は、パス名に含まれるすべてのディレクトリが本当のディレクトリであって、リンクやショートカットではないことを確認する。正規化関数の結果は、文字ベースのパスの検証が必ず有効な結果をもたらす、正規化されたパスである。たとえば、ユーザのホームディレクトリのパス名で始まる正規化されたパスは、そのパス上のファイルが必ずユーザのホームディレクトリまたはその下のサブディレクトリ内に存在することを保証する。

構造の項で説明したように、正規化関数があれば Pathname Canonicalization パターンはきわめてシンプルである。

1. プログラムはファイルを開く前に、指定されたパス名を対象に OS 固有のパス名正規化関数を呼び出す。
2. 正規化されたパス名はファイルの操作時に使用される。

正規化関数はプラットフォームによって提供されるべきである。プログラムはパス名の文字面を分析する前に、単純にプラットフォームの正規化関数を呼び出せばよい。以下はいくつかの OS 固有の正規化関数である。

- POSIX 準拠 OS： `realpath()`
- `glibc` を持つシステム： `canonicalize_file_name()`（`glibc` で提供される GNU 拡張）

実装の詳細については、『*The CERT C Secure Coding Standard*』[Seacord 2008]の FIO02-C、「Canonicalize path names originating from untrusted sources（信頼できないソースからのパス名は正規化すること）」を参照されたい。

4.2.8 サンプルコード

以下のサンプルコードは、ファイルを検証して開く前に、ユーザの指定したパス名を正規化する。

```
/* argv[1] が渡されているか確認する */

char *canonical_filename = canonicalize_file_name(argv[1]);
if (canonical_filename == NULL) {
  /* エラー処理を行う */
}

/* ファイル名の検証 */

if (fopen(canonical_filename, "w") == NULL) {
  /* エラー処理を行う */
}

/* ... */

free(canonical_filename);
canonical_filename = NULL;
```


Copyright ©2009 JPCERT/CC All rights reserved.

4.2.9 既知の使用例

xarchive-0.2.8-6

4.3 Input Validation（入力検査）

4.3.1 目的

入力データの有効性が正しく確認されていることを確実にすることによって、多くの脆弱性を防ぐことができる。入力検査においては、開発者が信頼できないデータソースから取得したすべての外部入力を正しく識別し、検証することが求められる。

4.3.2 動機

有効性を確認せずユーザからの入力値を使用することで、バッファオーバーフロー攻撃、SQL インジェクション攻撃、クロスサイトスクリプティング攻撃など、数多くの深刻なセキュリティ上の攻撃が引き起こされる。

クライアント／サーバ型アーキテクチャのアプリケーションが普及し、システム設計者が直面する問題の 1 つは、クライアント側またはサーバ側のどちらで入力値の有効性確認を行うかということである。入力検査の問題は、クライアント側のみで実施される場合に発生する。

クライアント側での入力検査は本質的に安全でない。Web ページの送信を偽装し、オリジナルのページのスクリプト処理を迂回することは簡単である。これは多かれ少なかれ、すべての種類のクライアント／サーバ型アーキテクチャに当てはまる。ただし、クライアント側の入力検査は、それのみに頼ることはできないにしても、依然として有用ではある。ユーザへの即座のフィードバックにより、サーバへの通信往復を避けることができ、応答時間とネットワーク帯域を節約できる。

4.3.3 例

ある大学で、大学職員がタイムシート情報の入力、口座への購入請求、および各種の資金源のステータスの追跡ができる、Web ベースのインターフェースを持つ ERP（Enterprise Resource Planning）アプリケーションを開発している。大学はセキュリティ上の検討課題の 1 つとして、悪意のある、または不正なユーザ入力値が、意図しない ERP データの変更、データの整合性の違反、またはユーザによる禁止されたデータへのアクセスへとつながらないことを保証したいと考えている。

4.3.4 適用可能性

このパターンは、信頼できないソースからデータを受け取るすべてのアプリケーションに適用可能である。セキュリティ境界を超えてプログラムに渡されるすべてのデータは入力検査が必要である。そのようなデータの例として、`argv`、環境変数、ソケット、パイプ、ファイル、シグナル、共有メモリ、およびデバイスなどがある。Web アプリケーションに固有の入力ソースとしては、HTTP フォームから送信される `GET` および `POST` のパラメータなどがある。他のアプリケーションにおいては、他の入力ソースが存在する可能性がある。

Copyright ©2009 JPCERT/CC All rights reserved.

4.3.5 構造

Input Validation パターンの構造はきわめてシンプルで、図 11 に示すように、信頼できない各入力値の識別と有効性確認のみが必要である。

図 11: Input Validation パターンの構造

4.3.6 関連要素

Input Validation パターンの関連要素を以下に示す。

- データを受け取るシステム。このパターンの主な関連要素はデータを受け取り、有効性を確認するシステムである。
- データを提供する外部エンティティ。システムに渡されるデータは何らかの外部ソースから来る。以下のようなデータソースが考えられる。
  - 利用者
  - ファイル
  - ネットワーク接続
  - 他のプロセスと共有するメモリ
  - データベースシステム

4.3.7 結果

すべてのシステム入力の有効性を確認することのメリットは、システムセキュリティの向上（不正な入力値に対する不適切な処理を悪用した攻撃を防ぐ）、および信頼性の向上（不正な入力値が渡された場合でも、システムが予測可能な適切な動作をする）である。入力値の有効性確認に伴うコストは、システムパフォーマンスの低下と、不正な入力が行われる可能性のあるすべての個所で行う識別と有効性の確認などに必要な処理の追加である。

4.3.8 実装

Input Validation セキュアデザインパターンの実装には次の 2 つの一般的な設計作業が伴う。

- **データの仕様定義および有効性確認：**すべての信頼できないソースから入力されるデータをすべて特定し、データの有効性を確認する。有効なデータであれば、いかなる範囲／組み合わせであっても処理できるよう設計し、実装しなければならない。この場合の有効なデータとは、システムの設計および実装上予期されるデータであり、これによりシステムが予期できない状態になることがないデータをいう。たとえば、システムが 2 個の整数を入力値として受け取り、その 2 個の値を乗算する場合、システムは次のいずれかを実行する必要がある。(a)演算の結果、オーバーフローまたは他の例外状態が発生

Copyright ©2009 JPCERT/CC All rights reserved.

しないことを保証するために入力値の有効性を確認する、または(b)演算の結果として発生したオーバーフローまたは他の例外状態イベントを処理する準備をしておく。仕様には、入力データサイズおよび数などの限界値、最小値および最大値、最小および最大の長さ、有効とする内容（コンテンツ）、初期化および再初期化要件、保存および転送のための暗号化の要件が指定されていなければならない。

- **仕様に対するすべての入力値の適合性の確認：**データカプセル化（たとえば、クラス）を使用して入力値の定義とカプセル化を行う。たとえば、ユーザ名の入力が正しい文字であることを確認するのに文字を 1 つずつチェックするのではなく、同様の検査仕様を共有する入力値に対するすべての操作をカプセル化したクラスを定義する。入力値は可能な限り早い段階でその有効性を確認する。正しくない入力値が常に悪意のあるものであるとは限らず、たいていは偶発的である。エラーを可能な限り早い段階で報告することは、多くの場合問題の修正に役立つ。例外がコードの奥深くで発生した場合、原因が不正な入力値で、どの入力値が境界を超えていたのかということは常に明確とは限らない。

データディクショナリや類似の仕組みを使用して、すべてのプログラム入力値の仕様定義を行える。入力値は通常、変数に格納され、一部の入力値はその後永続データとして保存される。入力値の有効性を確認するためには、有効な入力が何であるかについて仕様を策定しなければならない。ユーザ入力を保持するすべての変数だけでなく、永続データの保管場所からのデータを保持するすべての変数も含めてデータおよび変数の仕様を定義することが推奨される。ユーザ入力の有効性確認の必要性は明白だが、永続データの保管場所から読み込まれたデータの有効性を確認するのは、永続データ保管場所が改ざんされる可能性に対する防御となる。

一般的な実装ステップ

Input Validation パターンの実装ステップは、具体的には以下の手順で構成される。

1. **すべての入力ソースを特定する：**システムに対する入力のすべてのソースを特定しなければならない。入力ソースとは、システムにデータを供給する任意のエンティティまたはリソースであり、供給するデータは不確定なものである。つまり、システムの内部状態、およびシステムが過去に実行したデータの操作を伴うアクションによってデータの値が完全には確定することのないデータの供給源である。すでに述べたように、入力ソースとして考えられるのは、ファイルシステム、データベースシステム、ソケット経由で読み込むネットワークトラフィック、パイプからの入力、キーボードなどである。
2. **入力ソースのすべての読み取り箇所を特定する：**各入力ソースに対して、入力ソースからのデータが最初に読み取られるシステム内のすべてのポイントを特定する。システムと入力ソースが疎結合となるよう設計されており、ソースコードの構成設計上入力ソースとのやり取りを行う場所が限定されている場合、入力ソースからの読み取りの特定は比較的簡単である。しかし、データソースとのやり取りがソースコード全体にちらばるようにシステムが設計されている場合、入力ソースからのすべての読み取りの特定は難しくなる。
3. **有効なデータの仕様を定義する：**前のステップで特定した各読み取りデータに対して、読み取ったデータが有効であるとは何を意味するのかを定義する。有効性の定義は、読み取るデータの種類や、そのデータが何に使用されるかに依存する。たとえば、次のような例が考えられる。
    a. **数値データ**：数値データは、それが何らかの決まった範囲内に収まっていることを確認しなければならない。またデータがそれ以降の計算でオーバーフローやアンダ

Copyright ©2009 JPCERT/CC All rights reserved.

ーフローを発生させないことを確認する必要がある。C 言語における数値データのチェックに関する詳細は、『The CERT C Secure Coding Standard』[Seacord 2008]のルールとリコメンデーションに記載されている。

b. **文字列データ**： 文字列データを Web ページに表示する場合、HTML およびクライアント側スクリプトコードが含まれていないことを保証するためにサニタイズする必要がある。文字列データをデータベースクエリに使用する場合は、SQL インジェクション攻撃を防ぐためにサニタイズする必要がある。

4. **不正なデータの処理方法を決定する**：ステップ 2 で特定したすべてのデータの読み取りに対し、有効でないデータを受け取った場合のシステムの動作を明確に定義しなければならない。有効でない入力に対する反応としては、警告を発してデフォルトデータを使用して処理を継続したり、入力ソースにデータを再要求したりするなど、さまざま考えられる。有効でないデータに対する正しい処理方法は、アプリケーション固有の問題である。

5. **不正なデータの確認および処理のコードを追加する**：ステップ 2 で特定したすべてのデータの読み取りに対し、読み取りデータの有効性を確認するコードを記述し、有効でないデータの場合は処理されなければならない。

   有効でないデータを特定する方法には一般的に 2 つのアプローチがある。*ブラックリストおよびホワイトリストの使用*の 2 つである。ブラックリストの使用では、一般的にブラックリストと呼ばれる、有効でないことが判明している入力値の集合と、入力データを比較する。入力値がブラックリストにない場合、入力値は有効であると判断される。ホワイトリストの使用では、一般的にホワイトリストと呼ばれる、有効であることが判明している入力値の集合と、入力データを比較する。入力値がホワイトリストにない場合、入力値は有効でないと判断される。ブラックリストおよびホワイトリストの使用は双方とも実装がシンプルであり、入力値をブラックリストまたはホワイトリストと比較する。主な作業は、ホワイトリストまたはブラックリストの保守である。どちらのアプローチも適用可能な場合、ホワイトリストの方がより安全な選択だと考えられる。なぜなら、ブラックリストでは、新たな攻撃手法で使用されうる値を登録する必要があるのに対し、ホワイトリストでは新たに有効でない入力値が発見されても何も変更する必要がないからである。

実装に関する補足情報

構造化された方法で入力検査を行う具体的な実装方法が以下から入手できる。

- 『Input Validation Using the Strategy Pattern』[Gervasio 2007] ：このソリューションは、Gang of Four による Strategy パターン[Gamma 1995] を使用してさまざまな入力クラスに対する入力値の有効性確認を処理する。紹介されているソリューションは、PHP でプログラミングされている。
- 『Client/Server Input Validation Using MS ATL Server Libraries』[MSDN 2009c]：ATL ライブラリで提供される入力値の有効性確認ルーチンを使用し、Windows 環境でクライアントサーバ入力値の有効性確認を行うサンプル（C++）が提供されている。
- 『*Secure Programming Cookbook*』、Viega および Messier 著[Viega 2003] ：C++で入力値の有効性確認を実行するための、関数およびプログラミングの考え方が提供されている。
- 『Input Validation in Apache Struts Framework』[You 2009]：この論文は、Apache Struts フレームワークを使用して Java でプログラミングする場合の入力値の有効性確認を行う方法について、優れたチュートリアルを提供する。チュートリアルにおいて一般の注目に値するのは、有効なシステム入力値の詳細な仕様である。

Copyright ©2009 JPCERT/CC All rights reserved.

4.3.9 サンプルコード

以下のサンプルコードは、C++で記述した入力値の有効性確認の方法論を示す例である。Input Validation パターンを実装する方法は他にも多数存在する。

Input Validation パターンのサンプル実装の設計方針は、有効性確認基準を validator クラスとして示すことである。validator クラスは、有効性を確認する入力値を 1 つ受け取る静的な validate()メソッドを 1 つだけ持つクラスで、入力値が有効であれば true、入力値が有効でなければ false を返す。

次の validator クラスは、整数が定義されている範囲内に収まっているかどうかを確認する。

```
template <int lower, int upper> class InRange {

public:

  static bool validate(int item) {
    return ((item >= lower) && (item <= upper));
  }
};
```

次の validator クラスは、2 つの整数が乗算されたときに、*オーバーフローしない*ことを確認する[Seacord 2008]。

```
class NoOverflowOnMult {

public:

  static bool validate(int o1, int o2) {

    // この有効性確認メソッドは、long long のサイズが整数のサイズの 2 倍
    // よりも大きい場合にのみ使用可能。
    assert(sizeof(long long) >= 2 * sizeof(int));

    signed long long tmp = (signed long long)o1 * (signed long long)o2;

    // 結果を 32 ビット整数として表現できない場合、
    // オーバーフローが発生する。
    return !((tmp > INT_MAX) || (tmp < INT_MIN));
  }
};
```

次の validator クラスは、1 つの空白文字を含み、且つ、定義された文字長以下である、アルファベット文字の有効な名前を文字列データが保持しているかどうかを確認する。

```
template<int maxNameLen> class GoodName {

public:

  static bool validate(char *str) {

    // 名前には数字は使用できず、空白が 1 つだけ必要。
    unsigned int pos = 0;
    bool sawSpace = false;
    while ((pos < maxNameLen) && (str[pos] != '\0')) {
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
      // 確認対象文字列の文字は空白か？
      if (str[pos] == ' ') {
        // これは文字列内の 2 つ目の空白文字か？
        if (sawSpace) {
          // 名前に 2 つ以上の空白文字が存在する。有効な名前ではない。
          return false;
        }
        // 空白文字を 1 つ検出したことを記録する。
        sawSpace = true;
      }
      else {
        // 現在の文字はアルファベット文字か？
        if (!isalpha(str[pos])) {
          // 名前に少なくとも 1 つ非アルファベット文字が存在する。
          // 有効な名前ではない。
          return false;
        }
      }
      // 次の文字に進む。
      pos++;
    }

    // 有効な名前の文字列の文字数は、maxNameLen より少ない。
    if (str[pos] != '\0') {
      return false;
    }

    // ここまで到達したら、名前は有効である。
    return true;
  }
};
```

以下の main()プログラムは、validator クラスの使用方法の例をいくつか示す。

```
int main(int argc, const char* argv[]) {
  if (InRange<1,10>::validate(5)) {
    cout << "5 is valid input\n";
  }

  if (!InRange<1,10>::validate(15)) {
    cout << "15 is NOT valid input\n";
  }

  if (NoOverflowOnMult::validate(12, 33)) {
    cout << "12*33 will not overflow\n";
  }

  if (!(NoOverflowOnMult::validate(INT_MAX, 33))) {
    cout << "INT_MAX*33 WILL overflow\n";
  }

  if (GoodName<100>::validate("Corey Duffle")) {
    cout << "'Corey Duffle' is a valid name.\n";
  }
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  if (!GoodName<100>::validate("Sir Chumley the 5th")) {
    cout << "'Sir Chumley the 5th' is NOT a valid name.\n";
  }
}
```

4.3.10 解決例

大学は、ERP システムへの入力を行う 3 つのソースを特定した。

- データベースシステム
- HTML フォームの GET のパラメータ
- HTML フォームの POST のパラメータ

この大学は、GET／POST のパラメータを読み取るときに開発者が簡単に有効性確認ルーチンを指定できるように、GET／POST パラメータを読み取るユーティリティライブラリを作成した。開発者は、GET／POST パラメータの読み取りがユーティリティライブラリを通してのみ行われることを保証するために、ソースコードの静的解析ツールを使用している。また、入力の有効性の基準が正しく実装されていることを保証するために、入力の有効性確認ルーチンのソースコードに対してレビューを実施した。

大学は、入力ソースから得られる SQL クエリの作成に使用されるすべての文字列をサニタイジングするサードパーティ製のデータベース抽象化ライブラリを使用するほか、SQL クエリの結果を対象とする基本的なサニティチェックを提供する。

4.3.11 既知の使用例

多くの Web フレームワークやプログラミング言語、ライブラリが、入力の有効性確認およびサニタイジングを行うためのサポートを提供している。入力の有効性確認をサポートするフレームワークとしては以下が挙げられる。

- Ruby on Rails
- Java Struts
- Pylons
- Django

4.4 Resource Acquisition Is Initialization（RAII：リソースの確保は初期化時に行う）

4.4.1 目的

RAII パターンの目的は、プログラムのすべての実行パスにおいて、システムリソースの割り当てと割り当て解除が適切に行われるようにすることである。RAII は、オブジェクトのコンストラクタとデストラクタでリソースの割り当てと解除を行い、オブジェクトの外部利用者がオブジェクトのリソースの割り当てと解除を行う必要がないようにすることで、プログラムのリソースが適切に扱われることを保証する。

4.4.2 動機

一般的に、使用されるリソースはすべて適切なタイミングで解放する必要がある。これは、リソースの枯渇を防止するために必要なことである。また、使用している最中のリソースを解放してしまわないことも重要である。このような不適切なリソース管理は多くの場合、致命的な結果となるからである。たとえば、解放されているメモリを正しい割り当て手順を踏

Copyright ©2009 JPCERT/CC All rights reserved.

まずに使用することは、セキュリティ上の欠陥であると広く考えられている。なぜなら、既存の多くのメモリ管理メカニズムは、メモリの追加要求があると解放されているメモリを再利用するからである。解放されているメモリを適切な割り当て手続きなしで使用すると、メモリが解放された後に正しい手順で要求されたメモリに格納されたデータを上書きしてしまう可能性がある。

さらに、リソースを解放するタイミングを維持管理するのは、確保されているリソースが数多くある場合には難しい作業となる。リソースの寿命がソフトウェアの設計時に検討されていない場合、リソースが不要になり解放可能になるタイミングを判断するのは困難である。

4.4.3 例

RAII パターンの 1 つの使用例は、関数の先頭でメモリを割り当て、関数の終了直前にメモリを解放するプログラムである。また、制御フローが変わるタイミングでメモリを解放する場合も含む。たとえば、関数が例外をスローしたりプログラムを停止したりする場合でも、まず割り当てされたメモリを解放する。他の RAII パターンの使用例としては、オブジェクトが生成されるタイミングでネットワーク接続を開き（リソース割り当て）、オブジェクトが破棄される際にそのネットワーク接続を閉じる（リソース解放）使用方がある。

同様に、オブジェクトの生成時にファイルを開くオブジェクトも考えられる。この場合、オブジェクトはそのデストラクタの中でファイルを閉じなければならない。ファイルのオープン処理が任意で行われる場合、デストラクタは、ファイルが開いている場合にのみファイルを閉じる責任を担う。

4.4.4 適用可能性

RAII は、リソースを取得し、その後に解放しなければならない任意のシステムに適用可能である。このリソースとしては、メモリ領域、ファイル記述子、ソケットなどのネットワークリソースなどがある。

利用可能なリソースの量が限られており、取得したリソースを解放しないとリソースの枯渇やサービス運用妨害が引き起こされる場合に、本パターンは有用である。

4.4.5 構造

RAII セキュアデザインパターンの構造は比較的分かりやすい。オブジェクトの生成時に実行される共通コードの中で（一般的に、オブジェクト指向言語の場合にはオブジェクトのコンストラクタの中で）、リソースを割り当てる。オブジェクトが破棄される時に実行される共通コードの中で（一般的に、オブジェクト指向言語の場合にはオブジェクトのデストラクタの中で）、リソースの割り当てを解除する。

4.4.6 関連要素

RAII パターンの関連要素は以下のとおりである。

- システムのリソースを使用するオブジェクト
- 対象となるシステムのリソース

Copyright ©2009 JPCERT/CC All rights reserved.

4.4.7 結果

RAII により、自動的なリソース管理を実現される。つまり、リソースは、それを必要とするオブジェクトまたは関数によってのみ取得され、オブジェクトが破棄されるときに、リソースが解放されずに残ることがない。RAII を使用した場合、ガベージコレクションのような別のリソース管理方式の場合よりもプログラムの実行速度が遅くなる可能性があるが、実装に大きく依存する。

4.4.8 実装

RAII を適用することで、自動的なリソース管理が実現される。つまり、リソースは、それを必要とするオブジェクトまたは関数によってのみ取得され、解放される。オブジェクトがリソースを管理するとき、オブジェクトは一般的にそのコンストラクタの中でリソースを割り当て、そのデストラクタの中でリソースを解放する。オブジェクトのデストラクタは、オブジェクト自身が不要になった場合にも呼び出される必要がある。これも、リソースを管理するオブジェクト自身もまた別のリソースであり、別のオブジェクトまたは関数で管理する必要があるという点では、RAII の別の例である。

関数がリソースを管理するとき、関数は一般的にその実行中に（多くの場合はじめに）リソースを割り当て、戻る前にリソースを解放する。開発者は、例外など関数のあらゆる形態の異常終了を認識し、どのような終了の場合にもリソースが確実に解放されるようにしなければならない。つまり、例外をスローするような下位の関数を呼び出す関数は、その例外をキャッチして処理するか再スロー前に、リソースを解放しなければならない。

実装の詳細については、CERT による以下のセキュアコーディングガイドラインを参照されたい。

- C++の場合は、FIO42-CPP「Ensure files are properly closed when they are no longer needed（ファイルは必要がなくなったら必ず正しくクローズすること）」のファイルベースの RAII を参照[CERT 2009b]。
- C の場合は、MEM00-C「Allocate and free memory in the same module, at the same level of abstraction（メモリの割り当てと解放は、同一モジュールの同一抽象レベルで行うこと）」のメモリベースの RAII を参照[Seacord 2008]。
- Java の場合は、FIO34-J「Ensure all resources are properly closed when they are no longer needed（すべてのリソースは必要がなくなったら必ず正しくクローズすること）」のネットワークベースの RAII を参照[CERT 2009c]。

4.4.9 サンプルコード

RAII は、C++において最も普及している。C++の自動変数オブジェクトは、そのスコープが通常どおりまたは例外のスローを通じて終了するときに、そのデストラクタが呼び出されるからである。

以下に示す RAII クラスは、C の標準ライブラリのファイルシステムコールに対する軽量のラッパーである。

```
#include <cstdio>
#include <stdexcept> // std::runtime_error
class file {
public:
```


Copyright ©2009 JPCERT/CC All rights reserved.

```
  file (const char* filename) : file_(std::fopen(filename, "w+")) {
    if (!file_)
      throw std::runtime_error("file open failure");
  }

  ~file() {
    if (0 != std::fclose(file_)) { // 最終変更内容を破棄する必要はあるか？
      // 処理する
    }
  }

  void write (const char* str) {
    if (EOF == std::fputs(str, file_))
      throw std::runtime_error("file write failure");
  }

private:
  std::FILE* file_;

  // コピーと割り当てを防止する。未実装
  file (const file &);
  file & operator= (const file &);
};
```

以降、file クラスを以下のように使用できる。

```
void example_usage() {
  file logfile("logfile.txt"); // ファイルを開く（リソースを取得する）
  logfile.write("hello logfile!");
  // logfile の使用を継続する...
  //ログファイルを閉じることを気にせず、例外をスローするか関数を戻る。
  // logfile がスコープ外になると自動的に閉じられる。
}
```

この例は、file クラスが FILE*ファイルハンドルの管理をカプセル化する形で RAII を適用している。file オブジェクトのスコープが関数にローカルである場合、C++はそれを内包するスコープ（この例では関数）の外に出るときにそれらのオブジェクトが破棄されることを保証し、file のデストラクタは std::fclose(file_)を呼び出すことでファイルを解放する。さらに、file のコンストラクタにおいてファイルを開けなかった場合、例外をスローする。つまり、file インスタンスが作成されていれば、ファイルを利用できることが保証される。

4.4.10 既知の使用例

BOOST ライブラリは、新しい C++0x 標準に含められることが予定されている **boost::shared_ptr** を提供する。これは、参照先のオブジェクトの管理に参照カウント

Copyright ©2009 JPCERT/CC All rights reserved.

を使用するスマートポインタであり、**`shared_ptr`** が破棄された場合に参照先オブジェクトが破棄されることを保証する。

Copyright ©2009 JPCERT/CC All rights reserved.

5 結論と今後の作業

5.1 結論

セキュアなソフトウェア開発には、セキュアな設計が必須である。セキュアデザインパターンを活用することで、さまざまな抽象レベルでセキュリティの課題に対応することができる。有益なセキュアデザインパターンは、一見パターンには見えない既存のセキュアなソフトウェア開発におけるベストプラクティスを分析し一般化したり、あるいはセキュリティ上の課題に対応するために既存のデザインパターンを拡張したりすることで作成できる。

セキュリティ上の欠陥の少ないコード開発を支援するための情報やツールは年々充実してきているが、セキュアな設計技法に関する情報は容易に手に入らない状況である。セキュアな設計技法を再利用可能なデザインパターンとして抽出し、定義することで、それらの技法を繰り返し利用できるようになる。再利用可能なセキュアな設計手法が広く適用されるようになれば、セキュアな製品を作成するコストが下がるとともに、開発者およびエンドユーザの双方が直面する脆弱性のリスクが低減されるだろう。

5.2 今後の作業

既存のオブジェクト指向のデザインパターンを拡張することで作成したセキュアデザインパターン（設計レベルのセキュアデザインパターン）は、今後、実際のアプリケーションに実装し、テストする必要がある。これらのパターンの適用を評価するためにいくつかのプロトタイプを作成することは、パターンの利点を示す上で有益であり、また開発者が実際にパターンを利用する際の手本になる。プロトタイプ開発には Java や C++のようなオブジェクト指向言語を使用するのが適当である。

いくつかの既存のオブジェクト指向のデザインパターンを拡張し、セキュリティ上の特性を追加することができたが、同様の拡張が他の既存のデザインパターンでも可能か分析することは有用であろう。

また、既存のセキュアな製品を引き続き研究することで、有用なセキュアデザインパターンをさらに特定できる可能性がある。

セキュアデザインパターンを記述する過程で、ソフトウェアのセキュリティに弊害をもたらすいくつかの技法を明らかにすることができる。そのようなセキュアデザイン *アンチパターン*を具体的に文書化することは、ソフトウェアにおいて特定のリスクにさらされている範囲を開発者が切り分けるのに役立つであろう。

Copyright ©2009 JPCERT/CC All rights reserved.

参考文献

以下に記載の URL は、本文書の発行時点で有効であった URL である。カッコ内の日付は最終更新日、最終アクセス日、または Web ページ上の著作権登録日である。

[Alexander 1977]
Alexander, Christopher. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977 (ISBN 0195019199).

[Beck 1987]
Beck, Kent & Cunningham, Ward. "Using Pattern Languages for Object-Oriented Programs." Design Methodology for Object-Oriented Programming, Panel Session, OOPSLA, 1987. ACM, 1987.

[Bernstein 2008]
Bernstein, Daniel J. The qmail home page. http://cr.yp.to/qmail.html (2008).

[Blakely 2004]
Blakely, Bob, Heath, Craig, et al. *Security Design Patterns*. Berkshire, UK: The Open Group, 2004 (ISBN 1-931624-27-5).

[CERT 2009a]
CERT Secure Coding Standards (March 2009). https://www.securecoding.cert.org/confluence/x/BgE

[CERT 2009b]
The CERT C++ Secure Coding Standard (March 2009).
https://www.securecoding.cert.org/confluence/x/fQI

[CERT 2009c]
The CERT Sun Microsystems Secure Coding Standard for Java (March 2009).
https://www.securecoding.cert.org/confluence/x/Ux

[Kienzle 2003]
Kienzle, Darrell, Elder, Matthew, Tyree, David, & Edwards-Hewitt, James. "Secure Patterns Repository Version 1.0." November 2003.
http://www.scrypt.net/~celer/securitypatterns/repository.pdf

[Gamma 1995]
Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (ISBN 0201633612).

[Gervasio 2007]
Gervasio, Alejandro. "Validating User Input with the Strategy Pattern." Developer Shed, March 6, 2007. http://www.devshed.com/c/a/PHP/Validating-User-Input-with-the-Strategy-Pattern/

[Kernighan 1999]
Kernighan, Brian W. & Pike, Rob. *The Practice of Programming*. Addison-Wesley, 1999 (ISBN 020161586X).

Copyright ©2009 JPCERT/CC All rights reserved.

[Massa 2008]
Massa, Jacques. "Migrate an application to Windows Vista, UAC." BakTek, July 2008. http://www.baktek-web.com/en/topics/UAC.aspx

[MSDN 2009a]
Microsoft Developer Network. "Securable Objects." MSDN, January 2009. http://msdn.microsoft.com/en-us/library/aa379557.aspx

[MSDN 2009b]
Microsoft Developer Network. "Running with Administrator Privileges." MSDN, January 2009. http://msdn.microsoft.com/en-us/library/ms717801(VS.85).aspx

[MSDN 2009c]
Microsoft Developer Network. "Input Sample: Demonstrates User Input Validation on Client and Server." MSDN Visual Studio Developer Center. http://msdn.microsoft.com/en-us/library/x88c4k6b(VS.71).aspx (2009).

[Oppermann 1998]
Oppermann, André. "The big qmail picture." http://www.nrg4u.com/ (1998).

[Postfix]
The Postfix Home Page, http://www.postfix.org/ (2009).

[Provos 2003]
Provos, Niels. "Privilege Separated OpenSSH." Center for Information Technology Integration, August 2003. http://www.citi.umich.edu/u/provos/ssh/privsep.html

[Romanosky 2001]
Romanosky, Sasha. "Security Design Patterns, Part 1." http://www.cgisecurity.com/lib/securityDesignPatterns.html (November 2001).

[Schumacher 2006]
Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, & Sommerlad, Peter. *Security Patterns: Integrating Security and Systems Engineering*. West Sussex, UK: John Wiley and Sons, 2006 (ISBN-10 0-470-85884-2).

[Seacord 2008]
Seacord, Robert. *The CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[Steel 2005]
Steel, Chris, Nagappan, Ramesh, & Lai, Ray. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice-Hall, 2005 (ISBN-10 0131463071) .

[Tenouk 2009]
Tenouk. Module H, Windows OS Security, "Access Control Story: Part 1." http://www.tenouk.com/ModuleH.html (2009).

Copyright ©2009 JPCERT/CC All rights reserved.

[Viega 2003]
Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. O'Reilly Media, Inc., 2003 (ISBN-10 0596003943).

[Yoder 1997]
Yoder, Joseph & Barcalow, Jeffrey. "Architectural Patterns for Enabling Application Security." *Proceedings of the 4th Pattern Languages of Programming Conference*, 1997. http://hillside.net/plop/plop97/Workshops.html

[You 2009]
You, James (Jun B.). "Applying Design Patterns to Your Struts Validation Framework." Java Boutique. http://javaboutique.internet.com/tutorials/strutsvalid/ (2009).

Copyright ©2009 JPCERT/CC All rights reserved.