UNIVERSITÉ DE MONTRÉAL

DESIGN PATTERNS FOR DISTRIBUTED APPLICATION SECURITY

YUN WANG
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2004

© Yun Wang, 2004.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DESIGN PATTERNS FOR DISTRIBUTED APPLICATION SECURITY

présenté par: WANG Yun

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. QUINTERO Alejandro, Doct., président

M. GUIBAULT François, Ph.D., membre et directeur de recherche

M. FERNANDEZ José M., Ph.D., membre

DEDICATION

To all my family members.

Without your support, this would not have been possible.

ACKNOWLEDGMENTS

I would like to thank my director, professor François Guibault for guiding me through my research to complete a Master at École Polytechnique de Montréal. I thank professor Guibault for being so patient to provide me the insight to tackle different problems, read several drafts of my thesis and make invaluable comments. I also would like to thank my committee members: professor Alejandro Quintero and professor José M. Fernandez who have accepted to evaluate this thesis.

I am grateful to all of my family members, for their support and encouragement over the years. I would not have been able to make it this far without their support in so many different endeavours and for being great friends.

I thank Dr. Suzanne Sirois and Dr. Dongqing Wei who brought me opportunities to develop my interests in research. I also thank professor Jean-Yves Trépanier who gave me invaluable advices and comments on my research related subjects. Thanks to my colleagues Amadou N'diaye, Babak Mahdavi, Bin Chen, Daojun Liu, Djamel Bouhemhem, Marie-Garbrielle Vallet, Qun Zhou, Sébastien Laflamme, and Yuanli Wang who provided me much needed support during my research activities. Thanks to all my friends and everybody in CERCA and school who supported me and made sure that I maintained my sanity.

This acknowledgement would not be complete without mentioning my best friend Qingsong Xiao Bach and some very special people who have always been with me and given me lot of encouragement and happiness.

RÉSUMÉ

Dans le développement des applications et des logiciels, la sécurité est souvent ignorée à l'étape du prototype, soit parce que la politique de sécurité n'est pas disponible, ou parce qu'il semble plus facile de remettre à plus tard les soucis de sécurité. Cette omission rend le déploiement du système final beaucoup plus difficile.

Ce mémoire présente le patron "Gestionnaire de Sécurité", un patron de conception architectural qui établit une architecture de sécurité utilisant trois niveaux de mécanismes de contrôle d'accès et six modules. L'objectif de ce patron est d'aider à intégrer la politique de sécurité liée au contrôle d'accès à n'importe quelle étape dans le cycle de développement de la plate-forme VADOR et de permettre de prendre en compte les questions de sécurité plus facilement et de façon plus flexible pour différents organismes ayant des politiques de sécurité diverses.

Le patron Gestionnaire de Sécurité est utilisé dans le cadre du développement de la plate-forme VADOR, un projet qui a été initié afin de fournir une plate-forme pour la conception optimale multidisciplinaire (MDO). VADOR a été développé au CERCA en collaboration avec Bombardier Aéronautique, qui a fourni la grande majorité des applications d'analyse, des processus à automatiser ainsi q'un environnement industriel pour la validation du système.

Pour répondre aux exigences d'une plate-forme MDO (Salas and Townsend (1998)), une architecture multicouche et client/serveur a été proposée pour la plate-forme VADOR. Le patron Agent Actif, un patron de conception architectural global, a été développé pour construire la plate-forme d'agent mobile VADOR. Il est basé sur des composants logiciels de base, des langages, des protocoles standards et les principes de conception Orienté-Objet. Il permet l'exécution automatique des processus d'analyse et le mouvement des données à travers un réseau distribué d'ordinateurs hétérogènes et comporte un GUI (In-

terface Usager Graphique) qui permet aux utilisateurs de fonctionner interactivement. Il a également la capacité de la gestion de base de données permettant aux utilisateurs d'accéder à l'information et de visualiser les résultats d'analyse intermédiaires et finaux.

Le prototype de la plate-forme VADOR a été développé et utilisé chez Bombardier Aéronautique comme une plate-forme MDO. Il passe actuellement à l'étape de deploiment pour répondre aux exigences spécifiques des clients, y compris l'intégration des politiques de sécurité, car les mesures de sécurités n'ont pas été définies dans le prototype initial.

Ce mémoire se concentre sur le développement d'une architecture orientée vers la sécurité pour la plate-forme VADOR et le développement d'un patron "Gestionnaire de Sécurité" basé sur cette architecture. Comme composant du patron Agent Actif, le patron Gestionnaire de Sécurité aidera à créer un modèle de sécurité pour VADOR afin de faciliter l'intégration des politiques de sécurité pour exigences de sécurité dans divers organismes. Le Gestionnaire de Sécurité résout les problèmes spécifiques de sécurité liés aux requêtes d'accès aux ressources du système dans la plate-forme VADOR.

ABSTRACT

In software applications' development, security is often ignored at the stage of prototype, the reason being either that the security policy is not readily available, or that it seems easier to postpone security concerns. This omission makes the deployment of the system more difficult.

This thesis presents the Security Manager pattern, an architecture design pattern that builds security architecture with three level access control mechanisms and six modules. The objective of this pattern is to help to integrate the security policy related to access control at any stage in the development cycle of VADOR framework, and makes addressing security concerns easier and more flexible for different organizations with various security policies.

The Security Manager pattern is used in the development of the VADOR framework, a project that was initiated with the objective of developing a Multi-disciplinary Design Optimization (MDO) framework. It has been developed at CERCA, in collaboration with Bombardier Aerospace, who provided actual analysis applications, design processes in need of automation and test ground for the framework.

To meet the MDO framework's requirements (Salas and Townsend (1998)), a multi-layer, client-server architecture has been proposed for the VADOR framework, the Active Agent pattern, a global architecture design pattern has been developed to construct the VADOR mobile agent framework. It is based on standard basic software components, languages, and protocols, and extensively uses object-oriented principles. It provides automatic execution of processes and the movement of data across a distributed network of heterogeneous computers, and comprises a client GUI to allow users to operate interactively, it is also linked to a database management system providing users access to information and intermediate visualization of final analysis results.

The VADOR framework prototype has been developed and used in Bombardier Aerospace as a MDO framework, it is currently moving onto the implementation stage to meet customers' specific requirements, including the integration of security policies, as the security issues were not defined in the initial prototype.

This thesis is going to focus on the development of a security architecture in the VADOR framework, then a Security Manager pattern will be defined based on the security architecture. As a component of the Active Agent pattern, the Security Manager pattern will help to create a VADOR security model, so that it can facilitate the integration of security policies for security requirements from different organizations, and solve the specific security problems related to control accesses to system resources in the VADOR framework.

CONDENSÉ EN FRANÇAIS

Ce document présente le développement du patron *Gestionnaire de Sécurité*, un patron de conception orienté vers la sécurité qui comporte six modules et définit des mécanismes de protection divisés en trois niveaux. Ce patron vise à proposer une architecture de conception pour VADOR, une application répartiee, multi-utilisateur et multi-fil. Le patron *Gestionnaire de Sécurité* vise à permettre aux concepteurs d'éviter certains défauts liées à la sécurité au moment de la conception et à résoudre des problèmes spécifiques de sécurité.

Le patron *Gestionnaire de Sécurité* est mis en application et validé dans le système VADOR en tant qu'une des composantes du patron *Agent Actif*. L'objectif principal de cette mise en application du patron est d'éviter certains défauts de sécurité et de résoudre des problèmes de sécurité liés à l'accès aux commandes et aux ressources gérées par le système VADOR. Par ailleurs, la validation et les essais d'utilisation du patron *Gestionnaire de Sécurité* au sein du système VADOR.

Les patrons *Gestionnaire de Sécurité* et *Agent Actif*

Le système VADOR est un environnement de gestion de données et d'exécution de tâches d'analyse basé sur une approche de distribution utilisant le concept des agents mobiles qui met en application le patron *Agent Actif* (Chen (2004)). Tel qu'illustré à la figure I, le prototype de VADOR était conçu de telle sorte qu'un *Gestionnaire de Sécurité* soit responsable des aspects de sécurité dans le système. Cependant, étant donné que la politique de sécurité de VADOR n'avait pas été définie lors de la conception du prototype, le *Gestionnaire de Sécurité* n'a pas été réalisé, et les responsabilités liées à la sécurité n'ont pas été prises en compte. Le système VADOR est maintenant passé à une nou-

velle phase de développement, comprenant entre autre un déploiement dans plusieurs départements techniques chez Bombardier Aérospatiale. Dans ce contexte, les questions de sécurité lors des échanges de données et d'exécution des tâches d'analyse par différents groupes d'ingénieurs au sein de la compagnie ont pris une importance grandissante, particulièrement lorsque ces échanges et ces exécutions de tâches impliquent des sites distants connectés par Internet.

Le développement d'une politique de sécurité et des outils de gestion de cette politique est devenu une enjeu important du développement de VADOR, particulièrement dans la perspective où plusieurs industries peuvent potentiellement être intéressées à déployer le système, pour lesquelles la politique de sécurité devra être spécifiquement adaptée.

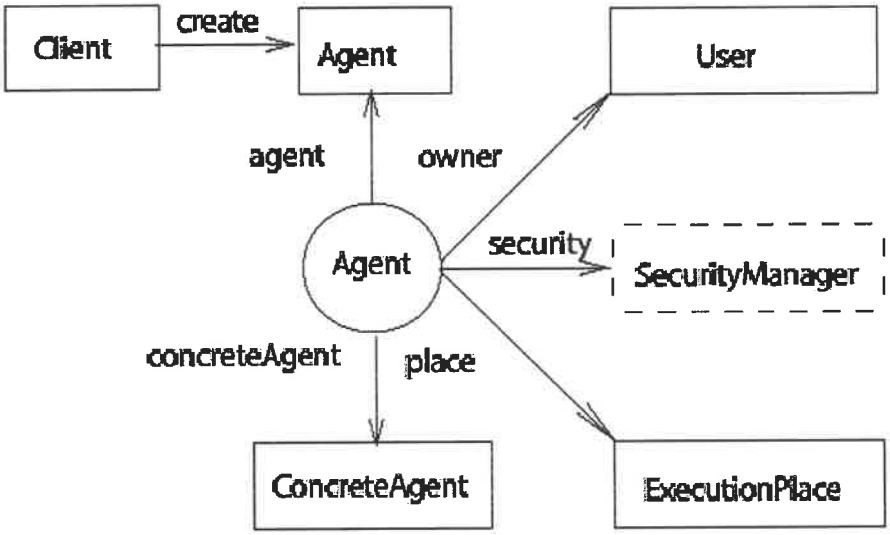


FIGURE I La Structure du Patron de *Agent Actif*

Problèmes de sécurité dans le système VADOR

Conçu dès le départ comme une application distribuée et multi-utilisateurs, le système VADOR fournit aux ingénieurs un environnement de conception efficace et bien adapté

à leurs besoins de collaboration au sein de l'entreprise. Cependant, puisque Internet est le principal média de transmission de l'information entre les applications réparties basées sur une technologie d'agents mobiles, la sécurité représente un défi central qui dérive des caractéristiques même de la plate-forme VADOR.

L'un des problèmes fondamentaux de sécurité de la plate-forme est lié à la façon dont les tâches sont déclenchées à distance sur les différents ordinateurs afin de réaliser une séquence d'analyse. Dans le prototype de VADOR, un utilisateur unique nommé VADORADM devait possèder les objets d'exécution (*Wrappers*) et les fichiers résultant des analyses, et ce, dans les répertoires de chacun des utilisateurs de façon à ce que le système VADOR puisse représenter sans restriction chaque utilisateur lors de la manipulation des dossiers d'autres utilisateurs.

Ce problème a été résolu au niveau du système d'exploitation, en utilisant SSH afin de laisser des utilisateurs de VADOR autres que VADORADM posséder leurs propres objets d'exécution de tâches et leurs dossiers dans leurs répertoires. Cependant, d'autres problèmes potentiels de sécurité existent toujours dans la plate-forme VADOR, tel que l'intégration de SSH à la plate-forme et des problèmes dans l'utilisation de processus multi-fils pour l'exécution simultanée de tâches multiples.

Ces problèmes peuvent induire des défauts de sécurité liés à une conception incorrecte de l'architecture de sécurité de VADOR. Ces défauts peurent être exploités à des fins malicieuses. Ils incluent le contrôle incorrect de point d'accès multiples , la vérification déficiente des erreurs, la mauvaise gestion des profile-utilisateurs, un contrôle déficient de l'accès à l'information, la mauvaise manipulation d'exceptions, et l'intégration déficiente des aspects de sécurité de systèmes externes.

Tous ces défauts peuvent mener, d'une façon ou d'une autre, à des problèmes de contrôle d'accès, principalement au niveau de la gestion des ressources des utilisateurs de VADOR. Par ailleurs, des mécanismes de contrôle définis incorrectement peuvent également poser

des problèmes fondamentaux de sécurité au niveau de la plate-forme. Dans ce contexte, l'architecture de sécurité de VADOR cherche principalement à fournir des mécanismes de protection et de contrôle flexibles pour l'ensemble des ressources de la plate-forme VADOR.

Structure et modules du patron *Gestionnaire de Sécurité*

Motivé par les besoins de contrôle d'accès aux ressources de VADOR, un patron *Gestionnaire de Sécurité* est développé et intégré à la plate-forme. L'objectif de ce patron de sécurité est de fournir un canevas architectural de sécurité pour la plate-forme, capable d'empêcher des défauts et des problèmes de sécurité liés au contrôle d'accès aux ressources.

Le patron *Gestionnaire de Sécurité* est basé sur des patrons de sécurité existants, et se compose de six modules prenant chacun une part de responsabilité dans la mise en place de mécanismes de protection au niveau du contrôle d'accès aux ressources. Tous ces modules travaillent ensemble pour établir la structure du patron, tel qu'illustré à la figure II.

- Le **Module d'Interface de Sécurité** fournit des interfaces à tous les modules reliés à la sécurité dans la plate-forme VADOR, favorisant l'intégration du patron dans une architecture logicielle déjà existante.
- Le **Module d'Authentification d'Agent** se compose des modules *Serveur de Sécurité* et *Signature d'Agent*. Ce module est responsable de contrôler le premier niveau de protection du système VADOR - l'authentification et la vérification des agents.
- Le **Module de Signature d'Agent** participe au premier niveau de protection en signant un agent qui est envoyé pour l'exécution d'une tâche.

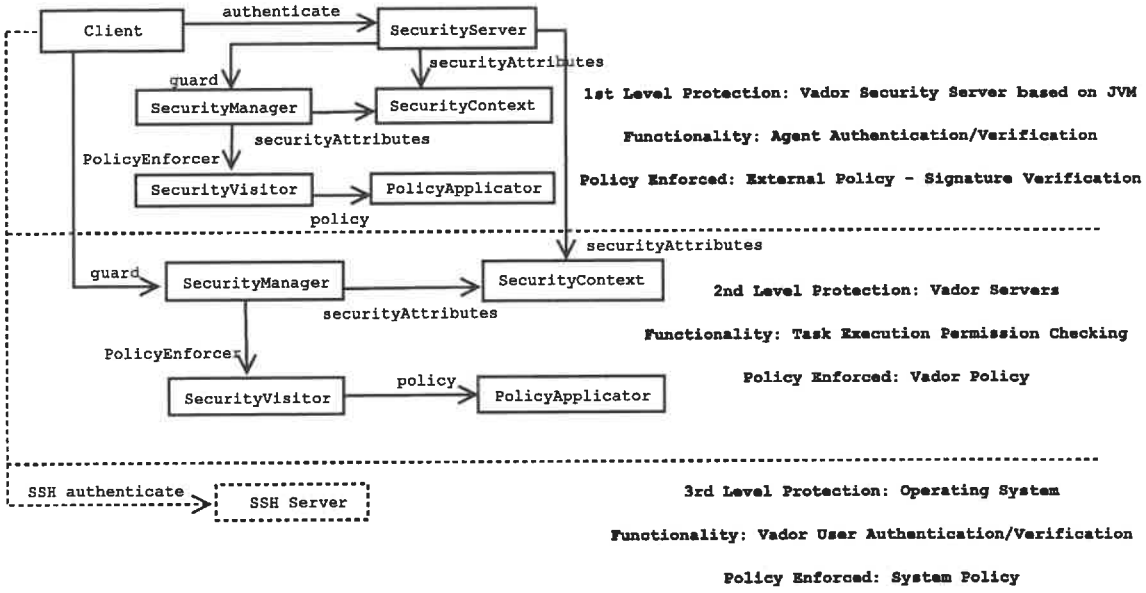


FIGURE II Structure du patron Gestionnaire de Sécurité

- Le **Module de Serveur de Sécurité** participe également au premier niveau de protection et collabore avec le module de signature d'agent, de sorte qu'il puisse vérifier la signature d'un agent et l'authentifier.

- Le **Module de Description des Attributs de Sécurité** permet d'accéder aux attributs relatifs à la sécurité d'une entité au nom de laquelle des opérations doivent être exécutées et de séparer ces attributs des autres caractéristiques des objets.

- Le **Module Gestionnaire de Sécurité** organise tous les modules afin qu'ils fonctionnent ensemble pour fournir aux autres serveurs de VADOR les fonctionnalités spécifiques liées à la sécurité. À titre d'exemples, le Gestionnaire de Sécurité, en réponse à une requête du *Serveur de Sécurité*, fournit des services de contrôle d'accès, pour le premier niveau des mécanismes de protection, soit la vérification et l'authentification des agents. Par ailleurs, le Gestionnaire de Sécurité, en réponse à une requête de l'*Executive Server*, fournit des services de contrôle d'accès pour le deuxième niveau des mécanismes de protection, soit l'autorisation d'un agent.

Trois niveaux de mécanismes de protection

Tel que mentionné ci-dessus, les problèmes fondamentaux de sécurité dans le système de VADOR ont été résolus en utilisant SSH au niveau du système d'exploitation. Cependant, d'autres problèmes potentiels de sécurité pourraient apparaître, en raison de possibles défauts dans les mécanismes de sécurité des différentes applications qui composent le système. Spécifiquement, les défauts dans le contrôle d'accès au système VADOR pourraient permettre l'accès non autorisé aux ressources du système, tels que les fichiers de données des utilisateurs. L'utilisation de SSH ne permet pas de se prévenir ce type de problème.

Afin de protéger les ressources tant contre les défauts au niveau de VADOR qu'au niveau du système d'exploitation, des mécanismes structurés en trois niveaux de protection sont définis dans le patron *Gestionnaire de Sécurité*, tel qu'illustré à la figure III :

- Le **Premier Niveau** est contrôlé par le Gestionnaire de Sécurité à travers le Serveur de Sécurité. Ce niveau est basé sur les mécanismes de sécurité fournis par la *Java Virtual Machine (JVM)*, et permet de respecter une politique de sécurité externe, dans laquelle des clefs de sécurité sont définies et initialisées par l'administrateur de VADOR. L'objectif de ce niveau est de contrôler les accès aux ressources du système VADOR, tels que les objets de VADOR. Les mécanismes associés à ce niveau comprennent la vérification et l'authentification des agents.

- Le **Deuxièmes Niveau** de mécanismes de protection est contrôlé par les Gestionnaires de Sécurité qui agissent au nom des autres serveurs de VADOR (endroits d'exécution). Ce niveau permet de faire respecter la politique de sécurité de VADOR, dans laquelle les privilèges des utilisateurs sont définis et stockés dans la base de données de VADOR. Cette politique peut être initialisée dynamiquement. Elle est nécessaire au Gestionnaire de Sécurité pour la vérification des permissions d'exécution des processus. L'objectif de ce niveau est de contrôler les

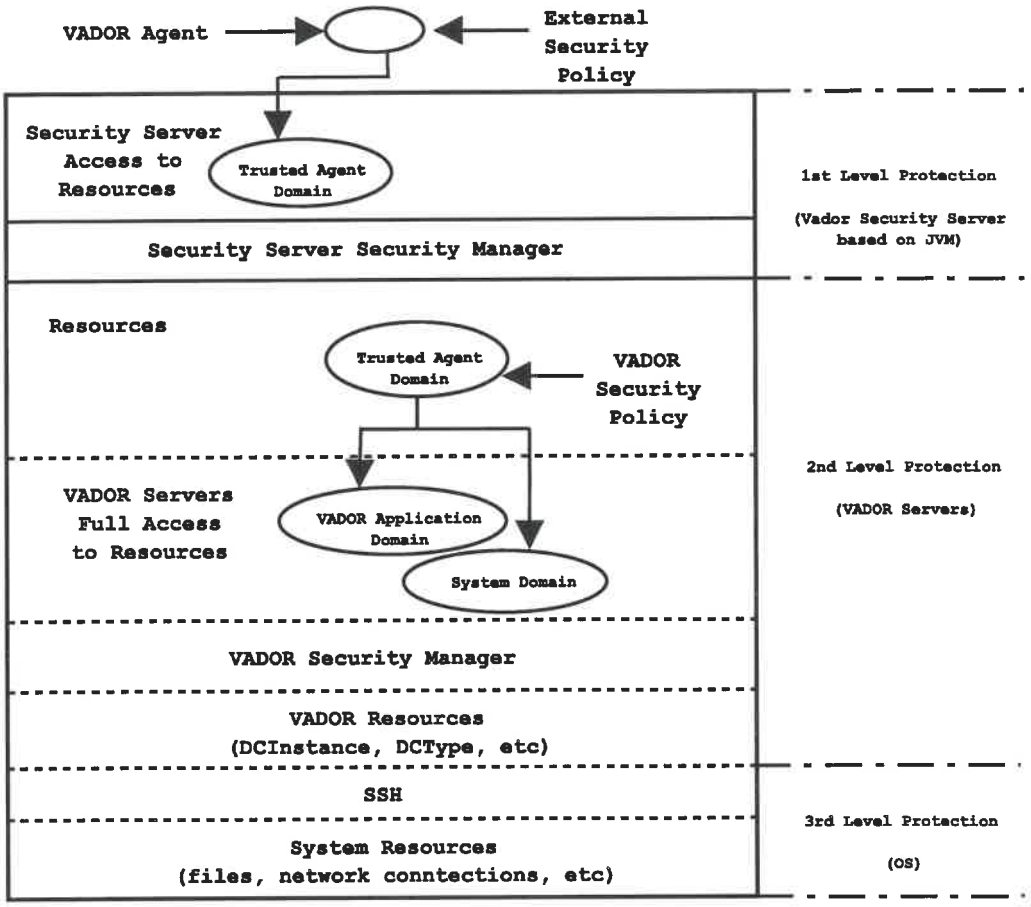


FIGURE III Modèle de Sécurité de VADOR

accès aux ressources et aux données de VADOR, tels que les fichiers de données des utilisateurs. Les mécanismes comprennent l'initialisation et la manipulation dynamique des politiques de sécurité associées aux données stockées, et l'autorisation de l'exécution d'agents basés sur cette politique.

- Le **Troisième Niveau** est la protection contrôlée par SSH au niveau du système d'exploitation. L'objectif derrière l'implantation de ce niveau est de permettre d'associer la propriété d'un agent à l'usager qui l'utilise et d'éliminer le concept du propriétaire d'agents VADORADM. De cette façon, les serveurs d'exécution (Wrapper) et les fichiers de données appartiennent uniquement à l'utilisateur qui les a créés, et les accès non autorisés sont automatiquement rejetés.

Patrons de sécurité

Plusieurs patrons de conception liés à la sécurité ont été utilisés dans le patron *Gestionnaire de Sécurité*:

- L'*Authentification d'Agent* et la *Signature d'Agent* implantent la classe *Codifier* fournie par le patron *Cryptographic Meta* pattern, de sorte qu'un agent puisse être signé avant qu'il soit envoyé pour des exécutions de tâche. Tout deux implantent également le patron *Sender Authentication* pour faciliter les procédés de vérification et d'authentification des expéditeurs d'agents.
- Le Serveur de Sécurité est une spécialisation du patron *Single Access Point*. Il contrôle les accès aux autres serveurs de VADOR et ne peut pas être évité.
- Le patron *Policy* fournit la structure de définition du serveur de sécurité pour les procédés de vérification et d'authentification. Il collabore également avec le patron *Security Manager* dans les processus d'autorisation.
- Les patrons *Subject Descriptor* et *Session* aident le Gestionnaire de Sécurité à définir le module Descripteur d'Attributs de Sécurité, de sorte que des attributs relatifs à la sécurité ne soient pas mêlés aux autres types d'attributs et puissent être partagés par plusieurs objets dans un même fil d'exécution du serveur.
- Les patrons *Protected System*, *Partitioned Application* et *Policy* servent à établir la structure du patron Gestionnaire de Sécurité.

Plusieurs autres patrons de conception participent également à la définition du patron *Gestionnaire de Sécurité*, tels que *Proxy*, *Template Method*, *Strategy*, et *Visitor*. Tous ces patrons collaborent avec les patrons de sécurité pour fournir aux concepteurs de systèmes VADOR une approche de conception architecturale basée sur des agents présentant des caractéristiques de sécurité évoluées.

Validation et tests du système VADOR

Le patron *Gestionnaire de Sécurité* est validé sur la base d'un plan de test élaboré selon les recommandations du *Principles of Software Validation* Soft-Solutions-International (2002). Cette approche de validation a été directement intégrée dans une démarche de vérification de la fonctionnalité du système VADOR puisque cet environment, développé à l'aide du patron *Gestionnaire de Sécurité*, correspond en tous points au type d'applications vers lesquelles le patron est orienté. Les tests visent principalement à vérifier le comportement correct du système et la possibilité de configurer le contrôle d'accès aux ressources en fonction des besoins spécifiques des usagers et adminstrateurs du système.

Les résultats des tests menés sur le système VADOR tendent à montrer le fonctionnement correct du système pour différents types de requêtes permises ou non, en fonction des configurations faites au niveau de la politique de sécurité. Ces tests permettent donc d'inférer que l'utilisation du patron *Gestionnaire de Sécurité* permet de concevoir une architecture logicielle réutilisable, qui permet de résoudre concrètement les problèmes de gestion des accès à un ensemble complexe de ressources distribuées.

Conclusion

L'implantation du patron *Gestionnaire de Sécurité* permet d'aider les développeurs de VADOR à contrôler les accès aux ressources en fournissant une architecture de sécurité composée de trois niveaux. Cette architecture logicielle, basée sur une approche par agent, permet également de contrôler les processus d'exécution de tâches associées à chaque type d'agent à l'aide de mécanismes d'identification des usagers et d'authentification des agents dans le système.

Dans sa version actuelle, le patron ne permet cependant pas de protéger l'agent contre des attaques visant les canaux de transmission. Si l'agent était attaqué et devenait malveillant, une exécution de tâche exigée par son expéditeur pourrait ne pas pouvoir être accomplie. Éventuellement, des mécanismes de protection des agents basés sur SSH et utilisant la redirection de ports pourraient être implantés au niveau du Gestionnaire de Sécurité afin de prévenir ce type d'attaques.

TABLE OF CONTENTS

DEDICATION iv

ACKNOWLEDGMENTS v

RÉSUMÉ vi

ABSTRACT viii

CONDENSÉ EN FRANÇAIS x

TABLE OF CONTENTS xx

LIST OF TABLES xxvi

LIST OF FIGURES xxvii

LIST OF ABBREVIATIONS AND SYMBOLS xxx

LIST OF APPENDICES xxxi

CHAPTER 1 INTRODUCTION 1

1.1 An Overview of the VADOR Framework 1

1.2 VADOR Architecture Design 4

1.2.1 Global Architecture 5

1.2.2 Application Layer Architecture - The Active Agent Pattern 7

1.2.2.1 Active Agent Components 7

1.2.2.2 Active Agent Structure and Participants 7

1.2.2.3 Active Agent Collaboration and Dynamic Behavior . . 9

1.2.2.4 The Sub-Components in the Agent Component 11

1.3 Explore Security Challenges 12

1.4 Enhance Security Functionalities Using Security Patterns 13
1.5 Organization of This Work 13

CHAPTER 2 REVIEW OF LITERATURE 15
2.1 Security Defects Classes in Distributed Applications 16
2.1.1 Landwher's Classification 16
2.1.2 Bishop's Classification 17
2.1.3 The Top Ten Web Application Security Vulnerabilities 18
2.1.4 Security Defects Related to Design 19
2.2 Review of Security Design Patterns 20
2.2.1 Background of Design Patterns 20
2.2.2 Template for Security Design Patterns 21
2.2.2.1 Security Pattern Definitions 21
2.2.2.2 Security Pattern Description 22
2.2.3 Design Patterns for Distributed Applications Security 24
2.2.3.1 Yoder and Barcalow 25
2.2.3.2 Eduardo B. Fernandez 25
2.2.3.3 Sasha Romanosky 26
2.2.3.4 Cryptographic Meta-pattern 27
2.2.3.5 Open Group 28
2.2.4 Summary 28
2.2.4.1 Prevent Structural Defects using Security Patterns . . . 28

CHAPTER 3 SECURITY DEFECTS IN THE VADOR FRAMEWORK . . . 34
3.1 Security Problems in the VADOR Framework 34
3.1.1 Threats to Agents 35
3.1.2 Threats to Data Files 36
3.1.2.1 The Current Situation of VADOR 36
3.1.2.2 The Fundamental Security Problem of VADOR 37

3.1.2.3 Other Potential VADOR Security Problems 38
3.1.2.4 Origin of VADOR Security Problems 38
3.1.2.5 Possible Solutions and Drawbacks 39
3.1.2.5.1 Problems of System Integration 39
3.1.2.5.2 Problems of the Multi-threaded Processes . . 40
3.2 Preview of Security Defects in the VADOR Framework 41
3.2.1 Improper Use of Multiple Access Points Control 41
3.2.2 Improper Error Checking 42
3.2.3 Improper Multi-User Profiles Management 43
3.2.4 Improper Global Information Access Control 43
3.2.5 Improper Exception Handling 44
3.2.6 Improperly Integrated External Security System 44
3.3 Summary 45

CHAPTER 4 SECURITY MODEL - FROM JAVA TO THE VADOR FRAMEWORK 47
4.1 Security in Java 2 SDK 47
4.1.1 J2SDK Security Features Overview 47
4.1.2 J2SDK Security Models 49
4.1.2.1 The Original Sandbox Model 49
4.1.2.1.1 JDK 1.0 Security Model 49
4.1.2.1.2 JDK 1.1 Security Model 50
4.1.2.2 The Current Security Model 51
4.1.3 J2SDK Protection Mechanisms 53
4.2 Security in the VADOR Framework 54
4.2.1 The VADOR Security Features Overview 54
4.2.2 The VADOR Security Model 55
4.2.2.1 1st Level Protection: Security Server 56

4.2.2.2 2nd Level Protection: VADOR Servers 57
4.2.2.3 3rd Level Protection: Operating System 58
4.2.3 The VADOR Protection Mechanisms 59
4.3 Summary 61

CHAPTER 5 THE SECURITY MANAGER PATTERN 62
5.1 Active Agent Pattern with Security Manager 62
5.1.1 The Extended Interaction Between Participants 63
5.1.2 The Extended Active Agent Dynamic Behavior 63
5.2 Security Manager Pattern 65
5.2.1 Name 66
5.2.2 Context 66
5.2.3 Problem 66
5.2.4 Solution 66
5.2.5 Structure 67
5.2.6 Participants 67
5.2.7 Interaction 69
5.3 Security Manager Pattern Modules 72
5.3.1 Security Interface Module 73
5.3.1.1 Components 74
5.3.2 Agent Authentication Module 75
5.3.2.1 Structure 75
5.3.2.2 Participants 75
5.3.2.3 Interaction 77
5.3.2.4 Related Patterns 79
5.3.3 Agent Signature Module 79
5.3.3.1 Structure 79
5.3.3.2 Participants 79

5.3.3.3 Interaction 81
5.3.3.4 Related Patterns 82
5.3.4 Security Server Module 84
5.3.4.1 Structure 84
5.3.4.2 Participants 85
5.3.4.3 Interaction 86
5.3.4.4 Related Patterns 87
5.3.5 Security Attributes Descriptor Module 87
5.3.5.1 Structure 88
5.3.5.2 Participants 88
5.3.5.3 Related Patterns 90
5.3.6 Security Manager Module 90
5.3.6.1 Structure 91
5.3.6.2 Participants 91
5.3.6.3 Interaction 95
5.3.6.4 Related Patterns 97
5.3.7 Consequences 97
5.3.8 Related Patterns 98
5.4 Summary 99

CHAPTER 6 VALIDATION AND TESTS 102
6.1 Objectives 102
6.1.1 Why Validate the Pattern 103
6.1.2 Why Test the Pattern in the VADOR Framework 103
6.2 Management of the Validation and Tests 103
6.2.1 Plan 104
6.2.2 Procedures and Expected Results 105
6.2.3 Test Cases and Results 107

6.2.3.1 Test Cases 107
6.2.3.2 Results of Validation and Tests 109
6.3 Limitations on the Tests 110
6.4 Possible Applicability to Other Systems 111
6.4.1 Relevance of Other Systems 111
6.4.2 Extension Points to Other Systems 112
6.4.2.1 Extension points for Agent Signature 112
6.4.2.2 Extension points for Security Server 112
6.4.2.3 Extension points for Security Attributes Descriptor . . . 113
6.4.2.4 Extension points for Security Manager 113
6.5 Summary 113

CONCLUSION 117

REFERENCES 122

APPENDICES 125

LIST OF TABLES

TABLE 6.1 Security Manager Pattern Validation Plan 104

TABLE 6.2 Keys Generation Test Cases and Results 108

TABLE 6.3 Certificates Exporting, Importing Test Cases and Results 108

TABLE 6.4 Agent Signature Test Cases and Results 108

TABLE 6.5 Agent Authentication Test Cases and Results 109

TABLE 6.6 VADOR Policy Database Table and Value Example 109

TABLE 6.7 Agent Authorization Test Cases and Results 110

TABLE 6.8 Comparison of Expected and Test Cases Results - 1 115

TABLE 6.9 Comparison of Expected and Test Cases Results - 2 116

LIST OF FIGURES

FIGURE I La Structure du Patron de *Agent Actif* xi

FIGURE II Structure du patron Gestionnaire de Sécurité xiv

FIGURE III Modèle de Sécurité de VADOR xvi

FIGURE 1.1 The VADOR Architecture Design 5

FIGURE 1.2 The Active Agent Pattern Components 8

FIGURE 1.3 The Active Agent Pattern Structure 8

FIGURE 1.4 Active Agent Collaboration 10

FIGURE 1.5 Sub-Components in the Agent Component 11

FIGURE 2.1 MVC 21

FIGURE 3.1 The Fundamental Security Problem of VADOR 37

FIGURE 4.1 JDK 1.0 Security Model 49

FIGURE 4.2 JDK 1.1 Security Model 51

FIGURE 4.3 The Current J2SDK Security Model 52

FIGURE 4.4 The Domain Composition of a Java Application Environment . . 53

FIGURE 4.5 The J2SDK Protection Mechanisms 54

FIGURE 4.6 VADOR Security Model 56

FIGURE 4.7 The Domain Composition of the VADOR Framework 59

FIGURE 4.8 The VADOR Protection Mechanisms 60

FIGURE 5.1 Extended Active Agent Pattern Interaction Diagram 64

FIGURE 5.2 Structure of the Security Manager Pattern 67

FIGURE 5.3 Security Manager Pattern Interaction 70

FIGURE 5.4 Signed Agent Verification Algorithms 71

FIGURE 5.5 Security Manager Pattern Modules Structure and Relationship . . 73

FIGURE 5.6 Security Interface Module Components 74

FIGURE 5.7 Agent Authentication Module Class Diagram 76

FIGURE 5.8 Agent Authentication Module Interaction Diagram 78

FIGURE 5.9 Agent Signature Module Class Diagram 80

FIGURE 5.10 Agent Signature Module Interaction Diagram 82

FIGURE 5.11 Agent Signature Algorithms 83

FIGURE 5.12 Security Server Module Class Diagram 84

FIGURE 5.13 Security Server Module Interaction Diagram 86

FIGURE 5.14 Security Attributes Descriptor Module Class Diagram 88

FIGURE 5.15 Security Manager Module Class Diagram 91

FIGURE 5.16 Security Manager Module Interaction Diagram 95

FIGURE 5.17 Extended Active Agent Pattern Sequence Diagram 101

LIST OF ABBREVIATIONS AND SYMBOLS

CERCA	CEntre de Calcul en Recherche Appliqué
VADOR	Virtual Aircraft Design and Optimization fRamework
MDO	Multidisciplinary Design Optimization
JDK	Java Development Kit
SDK	Standard Development Kit
JVM	Java Virtual Machine
SSH	Secure Shell
SecurP	Secure design Pattern project

LIST OF APPENDICES

APPENDIX I SECURITY DEFECTS CLASSES IN DISTRIBUTED APPLICATIONS 125

I.1 Landwher's Classification 125

I.2 Bishop's Classification 126

I.3 The Top Ten Web Application Security Vulnerabilities 127

I.4 Security Defects Related to Design 129

APPENDIX II SECURITY DESIGN PATTERNS 133

II.1 Yoder and Barcalow 133

II.2 Eduardo B. Fernandez 134

II.3 Sasha Romanosky 135

II.4 Cryptographic Meta-pattern 137

II.5 Open Group 139

CHAPTER 1

INTRODUCTION

The Virtual Aircraft Design Optimization fRamework (VADOR) is a Multidisciplinary Design Optimization (MDO) framework which has been developed at CERCA (CEntre de Calcul en Recherche Appliqué) and École Polytechnique in collaborating with Bombardier Aerospace. Based on a client-server architecture developed using object-oriented design patterns and the Java programming language, VADOR is built as a mobile agent environment that meets the requirements of a MDO software framework for aeronautical applications.

This introduction presents an overview of the VADOR framework, introduces the motivation and objectives of this research, and outlines the organization of this document.

1.1 An Overview of the VADOR Framework

MDO is a emerging discipline which provides methodologies and tools to tackle the formidable challenges of integrating high-fidelity physical models in a computation based design environment and to allow the synergism of mutually interacting disciplines to be fully exploited (Trépanier (1999)). It is now a vast field of research which finds application in all areas of engineering. For example, in aeronautics, coupled disciplines need to drive MDO research, as each design department is responsible for specific aspects of the engineering work required to design an airplane, but is also required to account for needs from other department in a search for overall acceptable designs.

A framework is one of the components that are involved in the deployment of an MDO

methodology. It is also one of the sources of increase in efficiency of disciplinary optimizations and sensitivity computations and of the development of specific MDO methodologies and strategies (Sobieszczansk-Sobieski and Haftka (1997)).

In the implementation of a design cycle based on MDO methodology, integration is a major weakness that precludes application development and automatic execution of analysis processes. The integration of various softwares in a software framework is a favored solution. The frameworks are ranging from engineering design frameworks to computer resources management frameworks, most often in a heterogeneous and parallel computing environment which requires available distributed computing technologies.

Through the use of an MDO framework that supports the integration of components of MDO applications, designers would be able to concentrate more on the application than the programming details. In addition, a common working environment would be provided by the framework, which would increase the productivity of multidisciplinary projects, thus reducing the time and the cost.

As a MDO framework, the objective of VADOR is to enable the seamless integration of commercial and in-house analysis applications in a heterogeneous, distributed computing environment, and to allow the management and sharing of data by the various departments of an aerospace organization.

In order to meet the MDO requirements and the needs of Bombardier Aerospace, five key characteristics have been identified for the VADOR framework:

- **Distributed system** which is developed using Object-Oriented methodologies with implementation in the Java programming language.

  This characteristic provides the VADOR Framework with the capacity to seamlessly integrate commercial and in-house analysis applications in a heterogeneous, distributed computing environment, and to allow the deployment of automatic de-

sign optimization algorithms based on the framework.

Distribution also improves efficiency and scalability of the VADOR framework and provides users with a flexible and configurable data model, in which the evolving requirements of engineers can adequately be satisfied using computational-based design-and-analysis programs. This distributed system also provides capabilities for the automation and integration of various processes used by engineers, supports and promotes collaboration and data sharing.

- **Manipulation of the user data in its native format**

  The VADOR Framework treats all the data as objects. Design-and-analysis data is encapsulated in objects, named DataComponents, that refer to the actual data stored in files. The DataComponents contain an appropriate set of attributes required for data management, but leave the data itself in the files that are being encapsulated.

- **Encapsulation of engineering applications**

  Engineering applications are treated as distinct objects in the VADOR framework. These components, named StrategyComponents, encapsulate the design-and-analysis methodologies or processes. The StrategyComponents represent the basic methods and the data flows required to transform data in a given process. The StrategyComponent can include user programs which can create the date files encapsulated in the DataComponents. The programs are usually executable legacy programs to be executed on a specific set of machines on the network.

- **Graphical user interface**

  The VADOR Framework offers a graphical user interface which is the visual part of the Java program and will be running on users' machines, the users create and manipulate interactively their own DataComponents and StrategyComponents in the graphical user interface applications.

- **Saving data in the database**

The DataComponent and StrategyComponent objects are saved in a database. The present architectural design supports the separation of the basic data, usually contained in files and potentially rather large, from the descriptive information. Only the descriptive information is stored in the database.

In order to reduce the risks related to architectural issues, and based on the above components, the VADOR framework applies a very recent approach to architectural design which involves heavy reliance upon design patterns and pattern languages to realize the distributed framework and improve its performance.

In the VADOR project, numerous previously published design patterns have been used to solve fundamental maintenance, evolution, distribution and concurrency problems encountered in the design and realization of the VADOR framework. The use of design patterns in the context of distributed software architectures is still a relatively recent topic for which research is very active. In the case of implementing the VADOR framework, great care has been taken to propose an architectural design of the framework which is both scalable and extensible. This results in a reusable framework architecture.

The design of the VADOR framework also relies on the development and evolution of a new design pattern, named the Active Agent pattern, which is based on the Active object, Command, Proxy, Visitor and Strategy patterns. This pattern tries to resolve concurrency problems in the distributed framework, and works as a mobile agent.

1.2 VADOR Architecture Design

In order to increase flexibility, scalability, reusability, and robustness, the VADOR framework design decouples the architecture into numerous autonomous modules, which are represented as the classical three layers: *Presentation Layer*, *Application Domain Layer*, and persistent *Data Layer*. As illustrated in figure 1.1, the architecture is expressed as a

set of components. Through the layering of different services, the proposed framework architecture should allow for easy evolution of the framework as the needs evolve. (Chen (2004))

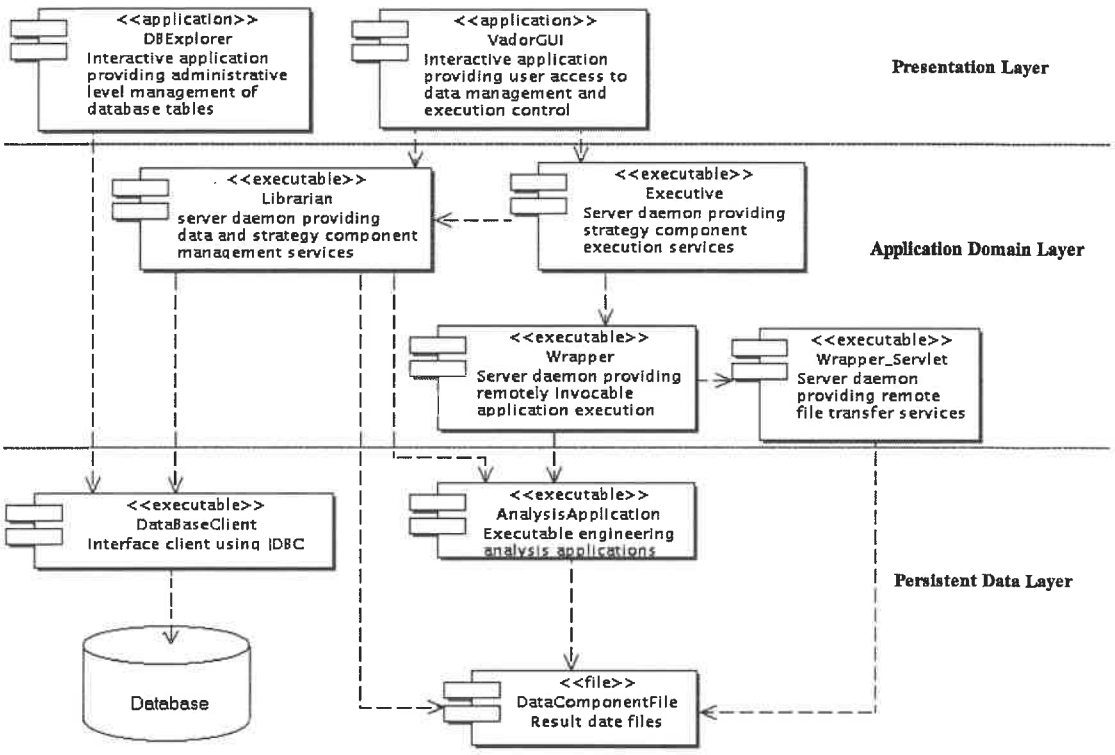


FIGURE 1.1 The VADOR Architecture Design

1.2.1 Global Architecture

The following components are included in the global VADOR modules:

1. **The Presentation Layer**

   - *The VadorGUI* provides a graphical user interface that lets users create and manipulate interactively their own Data and Strategy Components; these components form the basis of data and process information

- *The DBExplorer* is a client-side program that provides a graphical user interface that allows communications with the DataBase Client to directly manipulate the database, where the components created in the framework are stored. The *DBExplorer* is a system administration tool, as opposed to the *VadorGUI*, which is an interface targeted toward engineering users

2. **The Application Domain Layer**

   - *The Librarian Server* is responsible for the management of DataComponents and StrategyComponents, and of the interaction with the Database
   - *The Executive Server* is responsible for the execution of the commands issued by users through the VadorGUI, and for sending back execution results to the Librarian when an analysis step has completed
   - *The Wrapper and the Wrapper Servlet* are the remote CPU Servers interfaces, which are called by the Executive Server and that create the DataComponents and start the execution of the analysis applications
   - *The Analysis Application* programs are the legacy applications that are encapsulated in the framework

3. **The Data Layer**

   - *The DataComponents Files* are the files that store results of all engineering applications
   - *The DBMS* stores the descriptions of all components directly managed by the framework, including the description of DataComponents and StrategyComponents

1.2.2 Application Layer Architecture - The Active Agent Pattern

The application domain layer constitutes the core of VADOR System and comprises two main servers: *Executive Server* and *Librarian Server.* Most of the functionalities to process data and tasks in the VADOR framework reside in this layer, which is implemented using specialization of the *Active Agent* pattern.

The Active Agent pattern is based on the Active Object, Command, Proxy, Visitor, and Strategy patterns. It decouples the method executions from method invocation, so that it can enhance concurrency and simplify synchronized access to objects that reside in their own threads of control. It also decouples the method execution from the execution platform by encapsulating method executions in mobile agents. The objective is to solve problems related to the concurrency, scalability and flexibility of the framework. (Chen (2004))

1.2.2.1 Active Agent Components

The *VADOR System* is an agent based system that represents a specialized *Active Agent* pattern. It is implemented on top of the *Java Virtual Machine* (JVM). Three main components compose the Active Agent pattern: *Server*, *Client*, and *Agent.* As illustrated in figure 1.2, both server and client run on top of the JVM. They may run in the same or different machines. Agents run on the VADOR Server, they interact with their end-user via the VadorGUI.

1.2.2.2 Active Agent Structure and Participants

The Active Agent pattern consists of several internal participants, that cooperate to provide services to an external client. The pattern structure is illustrated in figure 1.3.

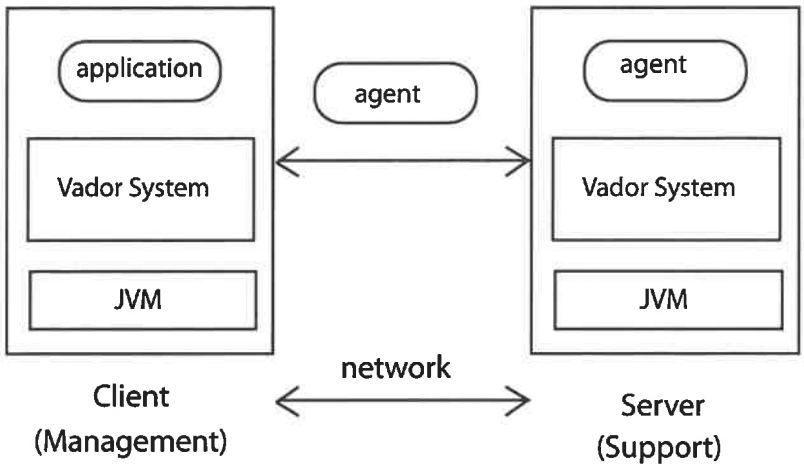


FIGURE 1.2 The Active Agent Pattern Components

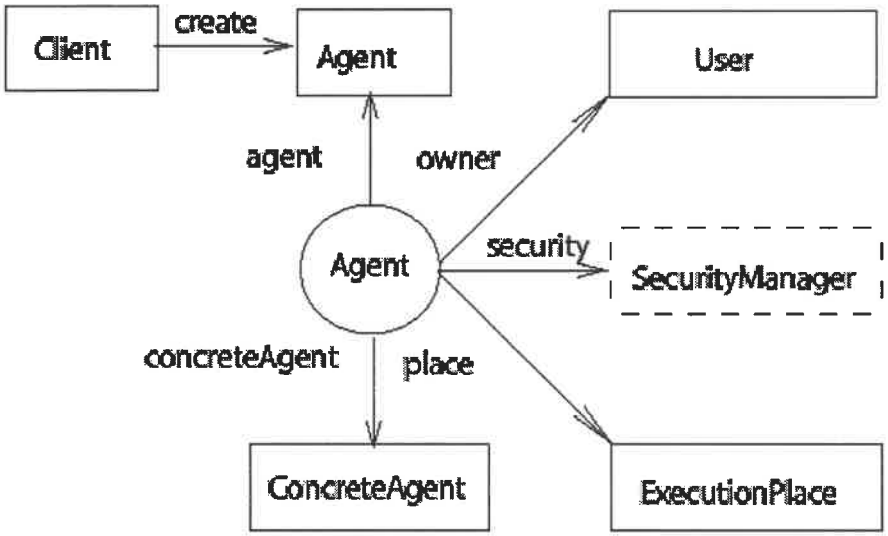


FIGURE 1.3 The Active Agent Pattern Structure

- **Client:** The client creates and manipulates the agents using the standard interfaces provided by the pattern. In the context of the VADOR framework, Clients represent applications that directly perform tasks on behalf of the framework users, such as the VadorGUI.

- **User:** Agent sender that is identified through a unique identifier (id) in the system. When a user creates an agent through one of the VADOR applications, his id is included in the agent, and this agent becomes his delegate.

- **Agent:** The Agent abstract class is the visible and extensible part of the Active Agent pattern, it defines the abstract behavior of the agent, which includes, in the case of the VADOR framework, the *call* function and the *can_run* function.

- **ConcreteAgent:** ConcreteAgent classes are subclasses of Agent. They implement the behavior related functions to execute the concrete tasks. For instance, the Open-Strategy Agent loads the StrategyComponent object from the Database, and the SaveStrategy Agent saves the StrategyComponent object in the database.

- **SecurityManager:** This class specifies the Agent access control security policy, it contains all the operations made available on the agent components. It was planned, but not implemented in the VADOR prototype.

- **ExecutionPlace:** This class specifies the agent's computational environment, which corresponds to the place where it was created as well as where it currently resides. In the VADOR framework, the Execution Places are the VADOR Servers, such as the Executive Server and the Librarian Server.

1.2.2.3 Active Agent Collaboration and Dynamic Behavior

In order to accomplish tasks using the VADOR framework, the Active Agent components need to collaborate with each other. As illustrated in figure 1.4, clients, such as VadorGUI, create agents; the agents then migrate to an execution place (the VADOR Server), which calls the standard agent operation (call function). Depending on the agent's security policy that is enforced by a Security Server in the system, and the Security Manager

on behalf of the execution place, the operation is executed, or not, on the related agent instance.

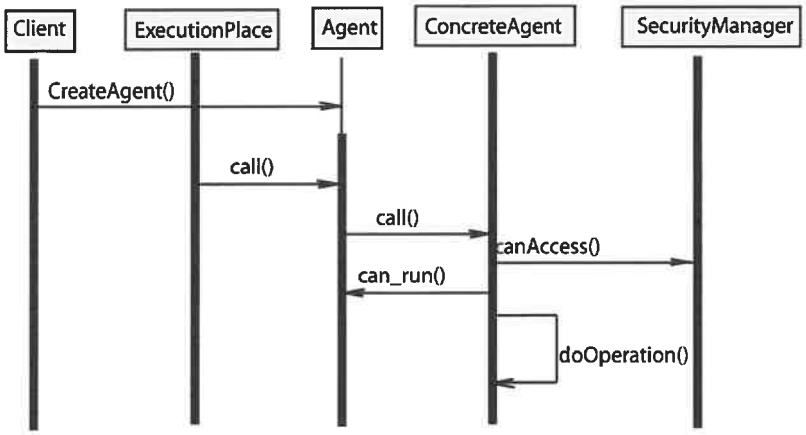


FIGURE 1.4 Active Agent Collaboration

Three phases are involved in the dynamic behavior of the Active Agent pattern:

1. **Agent construction and sending**

   A client creates an Agent object that includes the VADOR User, concrete VADOR Object, Vador Visitor interface and Vador Proxy objects. The Agent object then uses the Vador Proxy to send itself to the next Vador Server.

2. **Agent execution**

   After receiving the Agent object, the Vador Server calls its *call* function to start the execution. A runnable Agent object then dynamically loads the concrete Vador Visitor and uses it to execute the task. Before the agent does any operation, the Vador Server should request the Security Manager to enforce security policy to check the Agent's permission. Since the security policy is not available, and the Security Manager was not implemented, so that the security policy enforcement process was not developed in the VADOR prototype.

3. **Completion**

   The execution result is sent back by the Vador Server to the client.

1.2.2.4 The Sub-Components in the Agent Component

Four sub-components compose the Agent component in the Active Agent pattern. They are illustrated in figure 1.5.

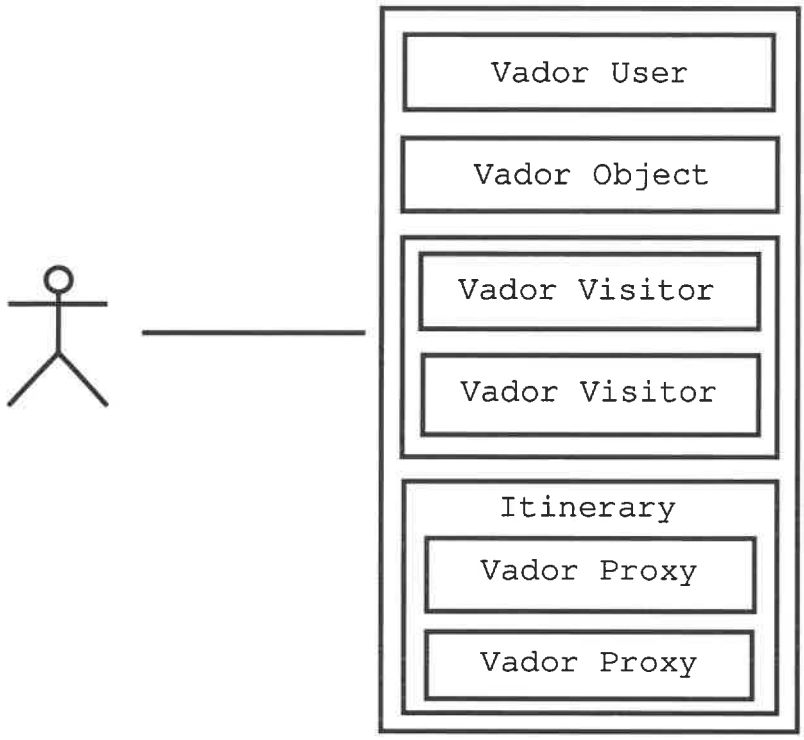


FIGURE 1.5 Sub-Components in the Agent Component

- **Vador User:** The Vador User object carries information on the agent sender (User). for example, user id and user name, etc.
- **Vador Object:** This is an abstract component which carries actual data on which the agent needs to operate. Examples of the Vador Object include StrategyComponent, DataComponent, etc.
- **Vador Visitor:** This is an execution class that includes information on how to execute a task.

- **Itinerary:** The Vador Itinerary contains agents' mobility information and their navigation among multiple destinations. It includes several Vador Proxy objects. Each Vador Proxy represents a destination server that the agent wants to go to.

1.3 Explore Security Challenges

As a MDO framework and distributed application that provides a mobile agent based architecture, the VADOR framework benefits from data transmission and sharing between designers and engineers over a heterogeneous, distributed computing environment. The Internet is thus a the major component of the transmission media.

However, security is a central challenge that derives from the characteristics of the VADOR framework, and this research is mainly motivated by this challenge, which consists of two aspects:

1. Virus, hackers, human defaults, the risk always exists when information is being transmitted via the Internet. These risks require to develop security mechanisms in the VADOR framework, so that they can protect the system and user data from threats or attackers.

2. Security policies usually are not available at the time when a system is in its building phase, but need to be defined and/or updated, and enforced to secure the system in later stages. The security policy was not initially defined in the VADOR prototype, but the functionality to flexibly specify a security policy was required for protecting the agent based system. As the development is moving onto the stage of releasing the VADOR framework for operational use at Bombardier Aerospace, an Access Control Policy needs to be defined. The objective of this security policy is to control the access to users' data files, the scope of this control is the security attributes of the data files, including their subjects, objects, and operations. With enforcement

of the access control policy, unauthorized access to the data files are not generally permitted as this would be in violation of the policy.

1.4 Enhance Security Functionalities Using Security Patterns

Motivated by the challenge of defining and implementing a valid security policy for the VADOR framework, this research conducts efforts on the enhancement of security functionalities using security design pattern concepts. The objective is to identify the main security issues involved in VADOR framework operations, preview security defects related to the functional design of the framework that may be exploited by threats and cause problems, and identify solutions to these problems using security design patterns. These security patterns will then be integrated into the Active Agent component - The Security Manager, which implements the Protected System pattern, and in which the VADOR access control security policy could be specified. This protection mechanism should control all operations made available by the agent components through each agent proxy.

1.5 Organization of This Work

This work is organized in the following parts:

- chapter 2 reviews some of the existing security defect classifications for distributed applications, it also reviews security design patterns that could be used to prevent the security defects and to solve security problems introduced by the defects.
- chapter 3 identifies security problems of the VADOR framework, previews the security defects that may cause problems from a design point of view, and then identifies approaches using security design patterns to prevent the defects.

- chapter 4 will introduce security model and mechanisms in the VADOR framework, that are used to prevent security defects introduced in chapter 3 and protect the *VADOR System* from threats.
- chapter 5 presents the Security Manager pattern that consists of the security design patterns identified in chapter 2.
- chapter 6 gathers and analyzes validation and testing information of the *Security Manager pattern* using the VADOR framework as an experimental example.
- Finally, the last chapter concludes on this research and discusses future works.

CHAPTER 2

REVIEW OF LITERATURE

"Traditionally, defects represent the undesirable aspects of a software's quality." (IBM (2002)). A software defect can cause a system to fail in its operation. A security defect of a software can be exploited that may result in unauthorized modifications of data, or disclosure of information which affects the system's reliability and security, or robustness.

The omission of security issues during a software development is the cause of security defects, the reason is either that the security policy is not generally available, or because it just seems easier to postpone security concerns.

Knowledge of security defects and appropriate approaches that could prevent them is important to ensure reliable operation and to preserve the integrity of stored information. These topics were the focus of defect classification studies that were conducted to make distributed systems secure and to improve the reliability of software, and of security design patterns studies that were conducted to prevent the defects from design and to improve the flexibility and extensibility of the software.

This chapter will review the available literature on security design patterns and defect classifications, specify the defect categories that may exist in distributed applications and could be prevented using secure and reliable design patterns, then summarizes and specifies the security patterns that can solve security problems in the design and prevent the defects introduced.

2.1 Security Defects Classes in Distributed Applications

Since the early 1970s, many researchers have been working on classifying software defects. They have published several reports concerning the methods of defect analysis, detection, correction, and categorized discovered defects into classes, or organized them into databases. The objectives of their classifications were to provide defect information to software developers, help them find approaches to prevent, detect, and correct the defects to build more robust systems.

This section reviews some of the recently published software defects or security flaws classification schemes in relation with distributed applications, and concludes on each of them by analyzing the possibilities of finding approaches that can prevent defects during the design phase using design pattern concepts. Details of the defect categories and limitations are outlined in Appendix I.

2.1.1 Landwher's Classification

Landwehr *et al.* (1994) provided a taxonomy for computer program security flaws together with an appendix that carefully documents 50 actual security flaws. His classification scheme categorized security flaws using three attributes: *By Genesis, By Time of Introduction*, and *By Location.* The goal was to help developers detect or correct the flaws. (See Appendix I.1 for details).

By analyzing the categories and focusing on the flaws which were introduced during software (by location) design (by time of introduction) and caused by design errors (genesis), this classification may help to define defects related to design. However, it is primarily related to security flaws in operating systems that have been built and released to operational use, so that it is difficult to find appropriate approaches related to security design patterns for avoiding the security flaws introduced.

2.1.2 Bishop's Classification

Bishop (Bishop (1995)) presented a taxonomy for security vulnerabilities, examined through vulnerabilities in the UNIX operating system. The objective of his work was to improve security of existing systems, and to help developers in writing programs with minimal exploitable security flaws. It is a guide for maintainers and software implementers to improve the security of these flawed systems or softwares. He categorized security faults that exist in UNIX operating systems and networks into four classes: *Improper protection*, *Improper Validation*, *Improper Synchronization*, and *Improper Choice of Operand or Operation*. See Appendix I.2 for details).

Bishop's taxonomy of security flaws (2.1.2) focused on application-level and programming-level problems based on six axis:

- The nature (cause) of a flaw based on PA (BISBEY II and HOLLINGWORTH (1978)) categories.
- The time of introduction based on conclusions by Landwehr (Landwehr *et al.* (1994)) "Time of Introduction" category into "System Problem" and "Procedure Problem".
- The exploitation domain that describes direct impact of a security flaw.
- The effect domain that describes the indirect impact of a security flaw.
- The minimum number of components to exploit the vulnerability that analyzes the conditions of introducing a flaw.
- The source of the identification of the vulnerability provides the information on identifying a security flaw.

These axis may be useful for helping analysts look at characteristics of a security flaw, and analyze problems in the detection and elimination of vulnerabilities, they may also help to identify defects related to design.

He mentioned "prevention of flaws using 'abstraction' to collect small parts and operations lumped together with well defined interfaces providing the only access to the internal representation and implement the abstraction properly", this approach could be realized by using security design pattern concepts. However, there was no detailed approaches introduced to prevent specified security flaws, and it is also difficult to identify security design patterns to prevent the security flaws introduced.

Because this taxonomy was defined based on the existing security defects classifications, such as PA (BISBEY II and HOLLINGWORTH (1978)) and Landwehr (Landwehr *et al.* (1994)), it overlapped the previous works in someway.

2.1.3 The Top Ten Web Application Security Vulnerabilities

The Open Web Application Security Project OWASP(OWASP (2003)) conducts research on web application security. The Top Ten Documentation Project published a list of the most critical web application security flaws, that is becoming a de-facto standard for web application security, and that has been used by commercial and educational organizations for projects planning and execution.

OWASP's list of top ten web application security vulnerabilities (Appendix I.3) represents the most probable flaws in web application with detailed description on each of them, including the environments affected, examples and references on how to determine if you are vulnerable.

The most useful part of this list is the information on how to protect yourself. Although the approaches for protection are too general from a design point of view, they provide

some key elements that are needed for prevention strategies. For example, to protect a web application from broken access control, it mentions that "the most important step is to think through an application's access control and capture it in a web application security policy ". This can help to design strategies to prevent broken access control by defining a valid security policy, and by applying it to the access control mechanism. However, it didn't specify the defects that may be specifically introduced during the design process, and the defect classes are too general for hiring security design patterns to prevent their introduction.

2.1.4 Security Defects Related to Design

The Secure design Patterns (SecurP) project (Guibault *et al.* (2004)) focused its efforts on classifying security defects related to the design phase in the development of distributed applications that could breach security functionalities as required by the Common Criteria (CSRC (1999)). That work also focuses on flaws that could be prevented using security design patterns. The defects are categorized into *Structural Defects* and *Functional Defects* according to the security problems addressed by the defects. (See Appendix I.4 for details).

The increasing need of data communications between organizations requires the common software defects concerning security and reliability to be classified, so that they can be prevented in the design phase using design pattern concepts, and the classification can be applied and customized by different organizations to fulfil their own needs.

Motivated by the above context, the first objective of this work is to help developers in building more secure and reliable software systems. Based on the first objective, the project focused efforts on the categorization of defects that are related to the design phase in the development process. These defects could be introduced by an absent, ambiguous or improper design, exploited by threats during operations and lead to risks to assets of

a system. They could be prevented by applying security design patterns, so that it can enhance the security functionalities of the system.

Using this defect classification scheme, it is easier to identify security problems introduced by the defects and apply security design patterns to prevent them.

2.2 Review of Security Design Patterns

2.2.1 Background of Design Patterns

In Object-Oriented Design, the design patterns solve problems in similar context to the patterns applied in buildings and towns, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander *et al.* (1977)). Design patterns in software engineering are expressed in terms of objects and interfaces instead of walls and doors.

One of the most famous framework was the Model-View-Controller (MVC) framework for Smalltalk (Krasner and S.T. (1988)), it is an example of a powerful reusable framework that uses design patterns.

The MVC framework divided the user interface design problems into three parts: *Data Model*, *View*, and *Controller*. The Data Model layer contains the computational aspects of the program, the View presents the user interface, and the Controller contains control aspects of the application, which interacted between the user and the view.

The purpose of this structure is to separate objects among the different parts of an application, with each part having its own rules for managing data. The proposed structure also controls the communication between the user, the GUI and the data, and it care-

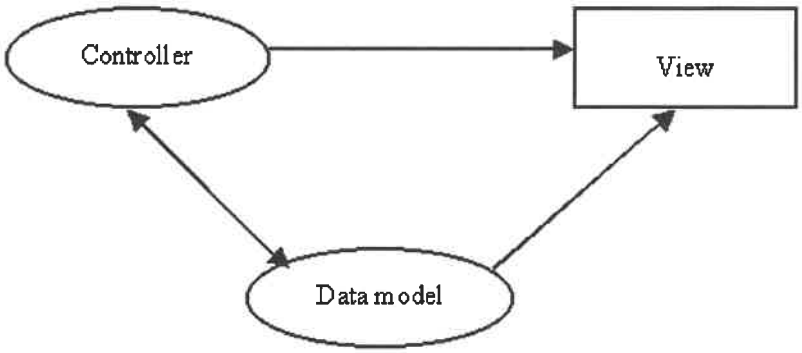


FIGURE 2.1 MVC

fully separates the responsibilities among the parts. The objects in the three parts talk to each other using a restrained set of connections, which are implemented as a set of a few predetermined communication channels.

In other words, design patterns describe how to establish communication between the objects while hiding their data models and methods from each other. Keeping this separation has always been an objective of good object oriented programming.

2.2.2 Template for Security Design Patterns

2.2.2.1 Security Pattern Definitions

Cooper (1998) cited some useful definitions of design patterns that have emerged as the literature in the field has expanded:

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development."(Pree, 1994)
- "Design patterns focus more on reuse of recurring architectural design themes,

while frameworks focus on detailed designed... and implementation."(Coplien & Schmidt, 1995).

- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it"(Buschmann, et.al.1996)
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components."(Gamma, et al.1993)

Based on the above given definitions, Schumacher and Roedig (2001) presented definitions of *Security Patterns* and *Security Pattern System*:

- **Security Patterns:** A security pattern describes a particular recurring security problems that arises in specific contexts and presents a well-proven generic scheme for its solution.
- **Security Pattern System:** A security pattern system is a collection of security patterns, together with guidelines for their implementation, combination and practical use in security engineering.

2.2.2.2 Security Pattern Description

With the additional aspects that turn a pattern into a security pattern, Schumacher and Roedig (2001) introduced the key elements of *security patterns* by following the *Mandatory Elements Present* pattern (Meszaros and Doble (1996)) and using the terminology given in the Common Criteria (CSRC (1999)):

- **Name:** Certainly *security patterns* are not different from normal patterns with regard to their *name*. The *name* of the pattern becomes a part of the vocabulary of

the community. It should be easy to remember and refer to. A good name should be evocative and give an image of what the pattern might be about.

- **Aliases (Optional):** The *aliases* section lists other names by which this security pattern might be known.

- **Context (and Related Patterns):** Based on a *scenario* the *context* of the *security pattern* is illustrated. The general conditions under which the *problem* does occur and which *forces* do emerge are described. It is also useful to list *context* setting *security patterns*. As some *countermeasures* may introduce other *vulnerabilities*, additional *security patterns* should be considered in the *related patterns* section. The same is also true of problems that are solved partly or could not be considered within the given *security pattern*. The way a pattern hierarchy will be formed in the *related patterns* section.

- **Problem:** The *Problem* statement defines the problem that will be solved by the security pattern. The major aspects of the problem are elaborated by the viewpoint of the *Forces* to be solved. In the field of security, a problem occurs whenever a system component is protected in an insufficient way against abuse. Generally speaking, we have to deal with generic *Threats*, i.e. a potential for the violation of security. A threat is a possible danger that exploits *vulnerabilities*.

- **Solution:** This section describes the *Solution* to the *Problem*. Appropriate solutions are determined by the *Context* of the pattern. According to certain *Security Objectives* (that may be written down in *Security Policies*), *Countermeasures* have to be applied in order to reduce the *Risk*. It is useful to warn from pitfalls (how does this pattern becomes an *Anti-Pattern*) and refer to variants of the pattern.

- **Structure (Optional):** Using certain diagrams to illustrate the *Structure* of a security pattern.

- **Interaction (Optional):**Using certain diagrams to illustrate *Interactions* between the participants of a security pattern.
- **Consequences (Optional):** Security has impacts on many other requirements such as performance and usability. Thus it could be helpful to enlist the *Consequences* of the application of a security pattern. The benefits and drawbacks of a security pattern can be discussed.
- **Examples (Optional):** In order to illustrate the application of a security pattern, concrete *Examples* could be provided. Useful are code or configuration samples as well as some sketches.

2.2.3 Design Patterns for Distributed Applications Security

As the Internet services have been growing dramatically in recent years, more and more distributed applications have been developed for providing data communication services, security issues are parts of the measurement for qualifying such applications.

Because threats and attacks are constantly evolving in distributed applications to exploit the systems' defects, and put systems at risk, designers are faced with challenges of incorporating into their designs specific mechanisms to detect and prevent such security breaches.

In most cases, effective security features are learned from experiences, and are not always shared among designers, so that the novice designers have to deal more and more with security even though they are not experts in the domain. Security design patterns have a great importance here because they can effortlessly bring novice designers to a higher security level. Defects can then be prevented by automatically applying the knowledge of wiser designers.

This section reviews some of the security design patterns related to the design of distributed applications' security functionalities. These patterns can be referred to the cause of design to prevent the security defects that may be exploited. Appendix II describes these patterns and the problems that they can solve in details.

2.2.3.1 Yoder and Barcalow

Yoder and Barcalow (Yoder and Barcalow (1998)) were among the first to adapt design pattern concepts to information security, they introduced seven patterns that represent an architecture patterns scheme for enabling application security: *Single Access Point* provides a security module and a way to log into the system, *Check Point* organizes security checks and their repercussions, *Roles* organizes users with similar security privileges, *Session* localizes global information in a multi-user environment, *Full View With Errors* provides a full view to users and shows exceptions when needed, *Limited View* allows users to only see what they have access to, and *Secure Access Layer* integrates application security with low level security.

These patterns are a good start for information security, but they are insufficient to cope with the issues that arise when securing a distributed application. In addition, the introduction of these patterns are too general to be applied directly to solve security problems. (See Appendix II.1 for detailed description).

2.2.3.2 Eduardo B. Fernandez

Fernandez introduced several design patterns for secure distributed applications. They are outlined in the following: *Object Filter* and *Access Control Framework* combine the functions of authentication, access control, and object filtering to constrain a client to access objects in specified ways defined by the client rights, the *Authenticator Pattern*

describes a general mechanism for providing identification and authentication to a server from a client. *Authorization*, *Role-Based Access Control*, and *Multilevel Security* correspond to the most common models for security in a newly built system. The last three patterns can be applied at all levels of the system.

Most of the security patterns introduced by Fernandez are concerned with building security models or frameworks for distributed applications, and may be used to solve security problems for these issues, but the author didn't present the collaboration between these patterns, therefore, before applying them to build security models or frameworks in distributed applications, they must be reorganized and specified. (See Appendix II.2 for detailed description).

2.2.3.3 Sasha Romanosky

Sasha Romanosky (Romanosky (2001)) presented eight patterns in a template format that was adapted from the Object Oriented design pattern template developed by the Gang of Four (Gamma *et al.* (1994)). The intent was to fulfil the gap of security patterns for distributed systems introduced by Yoder and Barcalow (Yoder and Barcalow (1998)), and to supplement Security Principles, Security Policies, and Security Procedures. The patterns are: *Authoritative Source of Data*, which recognizes the correct source of data, *Layered Security*, which configures multiple security checkpoints, *Risk Assessment and Management*, which helps to understand the relative value of information and protecting it accordingly, *3rd Party Communication*, which helps to understand the risks of third party relationships, *The Security Provider*, which leverages the power of a common security service across multiple applications, *White Hats, Hack Thyself*, which tests your own security by trying to defeat it, *Fail Securely*, which designs systems to fail in a secure manner, and *Low Hanging Fruit*, which takes care of the "quick wins".

According to Romanosky, these patterns are essentially security best practices that can

assist the reader in identifying and understanding existing patterns, and enable the rapid development and documentation of new best practices. In reality, they cannot be directly applied to solve security problems addressed by defects, because they need to hire other security patterns for the practice. (See Appendix II.3 for detailed description).

2.2.3.4 Cryptographic Meta-pattern

Braga *et al.* (1998) presented a set of nine cryptographic design patterns according to four fundamental objectives of cryptography: confidentially, integrity, authentication, and non-repudiation: *Information Secrecy* keeps the secrecy of information, *Message Integrity* avoids corruption of a message, *Message Authentication* authenticates the origin of a message, *Sender Authentication* avoids refusal of a message, *Secrecy with Authentication* proves the authenticity of a secret, *Secrecy with Signature* proves the authorship of a secret, *Secrecy with Integrity* keeps the integrity of a secret, *Signature with Appendix* separates message from signature, and *Secrecy with Signature with Appendix* separates secret from signature.

The nine patterns were then abstracted into a generic object-oriented Cryptographic Meta-pattern to define a generic software architecture to cryptography, and instantiated into a meta-pattern structure and dynamics.

These patterns describe methodologies for using cryptographic techniques. They are useful for solving problems and prevent security defects concerning user data protection, communication protection, and management of cryptography. (See Appendix II.4 for detailed description).

2.2.3.5 Open Group

In the draft of "Guide to Security Patterns" by The Open Group (OpenGroup (2002)), a set of security patterns were defined and can be used to provide a security framework for building a secure system. They were categorized into *Entity Patterns*, *Structural Patterns*, *Interaction Patterns*, *Behavior Patterns*, and *Available System Patterns* according to their scope.

Most of the introduced security patterns in this draft deal with systems' availability, rather than reliability and security. However, the *Entity Patterns* and the *Interaction Patterns* represent the basic strategies for protecting systems and communication channels. (See Appendix II.5 for detailed description).

2.2.4 Summary

Based on the surveys of security defects (Section 2.1) and security design patterns (Section 2.2) in distributed systems, this section summarizes and specifies the patterns that could be used in the design of distributed applications' security functionalities to solve security problems and prevent the defects introduced in the SecurP defect classification (Section 2.1.4).

2.2.4.1 Prevent Structural Defects using Security Patterns

1. **Group 1:**

   - *Security Design Patterns:*
     Protected System, Policy Enforcement Point, Single Access Point, Security Context.

- *Security Problems Solved:*
  Protect system resources against unauthorized access to or/and illegal operations on data.
- *Security Defects Prevented:*
  Untrusted Interface.

2. **Group 2:**

- *Security Design Patterns:*
  Security Context.
- *Security Problems Solved:*
  Manage and access to contextual properties to restrict dangerous privilege and verify the security concerns.
- *Security Defects Prevented:*
  Monolithic Application.

3. **Group 3:**

- *Security Design Patterns:*
  Risk Assessment and Management.
- *Security Problems Solved:*
  Keep track of the security relevant correctly, understanding the relative value of information and protecting it accordingly.
- *Security Defects Prevented:*
  Improper Security Auditing.

4. **Group 4:**

- *Security Design Patterns:*
  Cryptographic Meta-pattern.

- *Security Problems Solved:*

  Protect communications and properly use cryptography, so that the parties that were involved in the communications cannot deny their participants, and user data or system security functions can be protected.

- *Security Defects Prevented:*

  Improper Communication Protection, Insecure Use Cryptography.

5. **Group 5:**

   - *Security Design Patterns:*

     Limited View, Object Filter and Access Control Framework, The Authenticator, Authorization, Authoritative Source of Data, Recoverable Component, Checkpointed System, Cold Standby, Journaled Component.

   - *Security Problems Solved:*

     Protect user data from interception, interruption, modification, and deletion.

   - *Security Defects Prevented:*

     Improper User Data Protection.

6. **Group 6:**

   - *Security Design Patterns:*

     Check Point, Object Filter and Access Control Framework, The Authenticator, Authorization Roles, Role-Based Access Control, Multilevel Security.

   - *Security Problems Solved:*

     Enforce Security Policy properly, so that a system can be protected from deny of services or data disclosure.

   - *Security Defects Prevented:*

     Authentication/Identification Inadequate.

7. **Group 7:**

- *Security Design Patterns:*

  Roles, Role-Based Access Control, Multilevel Security.

- *Security Problems Solved:*

  Users can only gain access to data which they have right to access to, so that the data can be protected from unauthorized access.

- *Security Defects Prevented:*

  Improper Security Management

8. **Group 8:**

   - *Security Design Patterns:*

     Limited View, Object Filter and Access Control Framework, The Authenticator, Authorization.

   - *Security Problems Solved:*

     Protect a user's identity, so that it will not be discovered or misused by the others.

   - *Security Defects Prevented:*

     Improper Protection of Privacy.

9. **Group 9:**

   - *Security Design Patterns:*

     Full View With Errors, Limited View, Object Filter and Access Control Framework, The Authenticator, Authorization, Subject Descriptor, Recoverable Component, Checkpointed System, Cold Standby.

   - *Security Problems Solved:*

     Protect system security functions, so that they will not violate the system's security policy or disclose data.

   - *Security Defects Prevented:*

     Improper Protection of System Security Functions.

10. **Group 10:**

    - *Security Design Patterns:*

      Recoverable Component, Checkpointed System, Cold Standby, Comparator-Checked Fault-Tolerant System, Journaled Component, Hot Standby, External Storage, Replicated System, Error Detection/Correction.

    - *Security Problems Solved:*

      Prevent monopolizing the resources by users and provide availability of capacities caused by failure of the system.

    - *Security Defects Prevented:*

      Improper Utilization of Resource.

11. **Group 11:**

    - *Security Design Patterns:*

      Check Point, Session, Full View with Errors, Limited View, Object Filter and Access Control Framework, The Authenticator, Authorization.

    - *Security Problems Solved:*

      Protect systems from breaking access attempts.

    - *Security Defects Prevented:*

      Improper System Access Control.

12. **Group 12:**

    - *Security Design Patterns:*

      Secure Access Layer, Layered Security, 3rd Party Communication, The Security Provider, Secure Communication, Secure Association.

    - *Security Problems Solved:*

      Build trusted path/channels, so that they can provide assurance that the com-

munications between the users and the security functions, or/and between the security functions and the other systems are correct and secure.

- *Security Defects Prevented:*
  Untrusted Path/Channels.

CHAPTER 3

SECURITY DEFECTS IN THE VADOR FRAMEWORK

The VADOR framework is a distributed, multi-threaded, and multi-user application, it uses Internet technology to make data communications available to users from different locations and on different machines. To secure the communications from attacks, approaches to prevent security defects should be considered in the early stage of its development.

The subject of this chapter is security defects in the VADOR framework. Section 3.1 is an overview of the VADOR security problems, then section 3.2 previews the security defects that may cause problems.

3.1 Security Problems in the VADOR Framework

In the VADOR framework, there are four servers or hosts that cooperate using the Active Agent. They are the VadorGUI server, the Librarian server, the Executive server, and the Wrapper server. When a user runs the VADOR application, the VadorGUI server starts. It runs on the host where the user requires to execute a design process and retrieves the execution results. The request is sent to the Executive server, which is responsible for managing the execution of the StrategyComponenents and creating DataComponents according to the request. The executive server dispatches an Active Agent to the Librarian server, which is the manager of DataComponents and StrategyComponents, to fetch the necessary data from the database for the process. Then the agent passes the task and data to the Wrapper server for starting the analysis application and create the DataComponents.

The VADOR framework is an instance of a mobile agent environment. It benefits from the implementation of the Active Agent Pattern (Chen (2004)), which can help the system to be flexible, extensible and easier to maintain. However, it has to face the security threats to the agent and user data:

- *Threats to agent:* threats that can affect the agent during its migration, and may affect servers.
- *Threats to user data:* threats that can affect a specific user data file when there is a command that needs to be executed on it.

This section analyzes these threats and their cause in the context of the VADOR framework, and specifies the security problems that ensue.

3.1.1 Threats to Agents

In the VADOR framework, the Executive server communicates with the Librarian server through Active Agent's migration. When the Executive server receives a new process execution request from a user, it dispatches an agent to the Librarian server, to get data from the database, then the agent migrates back to the Executive server and sets the data to continue the process.

The threats to agent's migration could come from a malicious agent that attacks the servers, or from the servers that attack an agent.

In the case of the agent that attacks the servers, it is assumed that the agent has been attacked by a malicious third party when it was passing over the network, or sent by a malicious Executive server. In that case, the agent's code or state may have been modified. For example, when the agent migrates on the Librarian server, instead of sending

a "select" data to the database, it may "delete" data. When the agent moves back to the Executive server, it may report the incorrect data and cause the Executive server to pursue the process in a wrong way, or it may modify the server's code to cause the server to malfunction.

In the case where the servers attack the agent, it may be assumed that, for instance, the Librarian server has been attacked and has become a malicious server. When the agent migrates on it, it may modify the agent's state to cause the agent to report incorrect information to the Executive server, or it may modify the agent's code and cause the agent to be harmful to the Executive server when it returns back.

Both cases can result in system malfunctions and process failures.

3.1.2 Threats to Data Files

Because data files are the places where the analyzed results are stored, protecting the data files is the fundamental security concern of the VADOR framework.

This section introduces the actual situation of the VADOR framework, the fundamental and potential security problems that may exist in VADOR, and then it analyzes the causes of the problems, possible solutions and drawbacks of the solutions.

3.1.2.1 The Current Situation of VADOR

The VADOR application requires to remotely execute commands on different hosts on behalf of VADOR users. In the current implementation, a data directory, named VADOR, is created initially within the user's file system and the owner of this directory is the user. However, the user needs to open this directory's permission to other users in a group, because when a task is to be remotely executed by the VADOR system, it is executed by a

Wrapper on any machine and the Wrapper belongs to a user named VADORADM, which needs to read or write files in the user's VADOR directory. As a result, the files created by VADORADM in the user's VADOR directory are owned by VADORADM.

3.1.2.2 The Fundamental Security Problem of VADOR

The fundamental problem of the VADOR system is that VADORADM owns the files inside the user's VADOR directory.

Actually, when the VADOR system remotely executes a task to create a file, it is a Wrapper belonging to VADORADM which creates the file, and writes it into the user's VADOR directory. In this way, VADORADM is the owner of the file. And as the owner, VADORADM can manipulate the files without the permission of the user who owns the VADOR directory.

Figure 3.1 illustrates a potential security problem with an example of executing the "rm" command on a remote machine through the VADOR system.

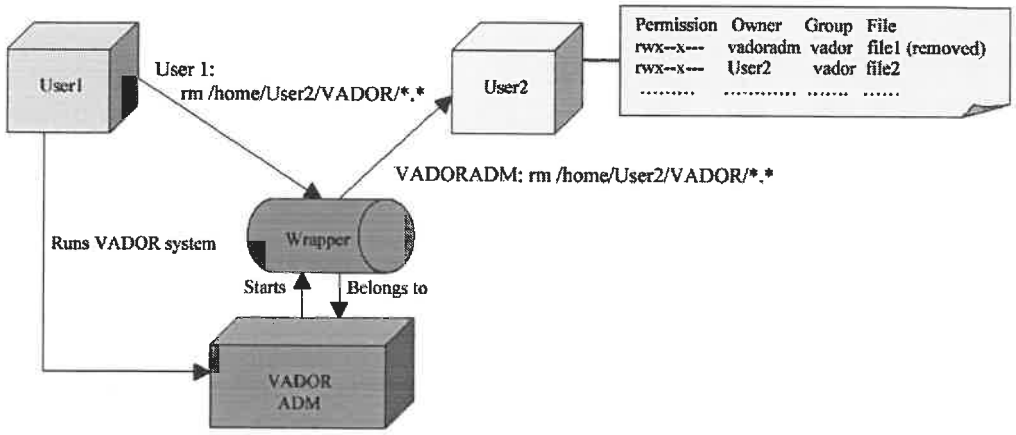


FIGURE 3.1 The Fundamental Security Problem of VADOR

This example shows that when User1 wants to remove all files in User2's VADOR directory via the VADOR system, it is a Wrapper that belongs to VADORADM which executes

the "rm" command in the User2's VADOR directory. In this case, because file1 was created by VADORADM or has been changed owner to VADORADM, it can be removed. File2 is owned by the user, and members of the group don't have the permission to read or write to it, thus, it won't be removed.

As a result, the fact that VADORADM owns all the files created by the system, allows any user to use the system to remove files in other user's VADOR directories. How to protect users' data files is the first challenge in securing the VADOR framework.

3.1.2.3 Other Potential VADOR Security Problems

Section 3.1.2.2 discussed some fundamental security problems of the VADOR system: the files that are created by the VADOR system are owned by the user VADORADM, and can be remotely manipulated by VADOR users because Wrappers belong to VADORADM and perform the manipulation.

However, other problems may also exist. For example, in figure 3.1, when User2 himself needs to modify file2 using the VADOR system, he has to grant read and write permissions to VADOR group members. Indeed, correct execution of VADORADM which is the owner of the Wrapper server that executes the commands on behalf of users is based on group permissions of the files that need to be manipulated. This can cause file2 to be modified by other VADOR group users without notifying user User2.

3.1.2.4 Origin of VADOR Security Problems

The original idea of having a VADOR user VADORADM who owns the Wrappers and the files in VADOR user's directories came from two considerations:

- Having only one Executive server for the system and one Wrapper server on every machine;
- Maintain the consistency of the database by permitting only the user VADORADM to manipulate the files on disk.

However, in a multi-user environment with multiple access points, to realize this idea is not easy. It requires well designed mechanisms for controlling access from the multiple access points, or dealing with the multi-threaded task execution.

3.1.2.5 Possible Solutions and Drawbacks

Ideally, the solution to these problems in the VADOR system would be to secure the files by giving the ownership of the files to the users, and at the same time, to keep having only one Executive server for the system, one Wrapper server on every machine, and of course, maintaining the consistency of the database. These requirements have been met through the use of SSH2 (SSH (2004)) and Expect (Libes (1995)) scripts in the VADOR system. The drawbacks to this solution are new problems that need to be solved:

1. *Problem of the system integration:* Instead of using the SSH2 and Expect application, how to integrate SSH2 to the system.
2. *Problem of file access control for multi-threaded processes:* How to get permission from a user when there is a multi-threaded process requiring access to his file.

3.1.2.5.1 Problems of System Integration

In the current VADOR system, SSH2 and Expect scripts are working together to implement file access control.

The reason of using SSH2 is that the SSH2 uses host key (public key and private key) identification to secure the communication over the network, when a remote user requires to logon a local machine, there is no password needed for authentication. However, a passphrase will be asked either when the user logs on to his machine (using SSH-Agent) or when he is trying to remotely logon to another machine (without SSH-Agent). This passphrase will not be transferred over the network; it is used for encrypting the user's private key for the authentication purposes.

Because the Wrapper server is the final executive of the SSH commands, to type the passphrase for every command is impossible (especially when the commands are written in a script file). In practice, the passphrase will be asked to the user by the VadorGUI at logon and will be transmitted by the VadorGUI to the Executive and Wrappers when needed during a VADOR session. The Expect script solved this problem by allowing to pass the passphrase as an answer to the SSH command in a simulated terminal.

Even though SSH and Expect solved the file access problem for the VADOR application, they have not been integrated into the VADOR framework, and can only be applied on UNIX operating systems. Another problem is that the passphrase has to be entered from the VadorGUI by the user, and passed through the Executive server on to the Wrapper server by the active agent. How to protect the passphrase during the transfer and how to integrate the SSH and Expect into the VADOR architecture are the main design challenges relating to file access control in the VADOR system.

3.1.2.5.2 Problems of the Multi-threaded Processes

The previous section discussed using SSH2 and Expect scripts for file access control in the VADOR system, but they can only control file access for a single request execution on a specified file. VADOR users usually require multi-threaded processes in the course of design, which needs to synchronously or recursively execute more than one command

on different files by different Wrappers. In this case, in order to access the files, the user needs to ask for the file owners' passphrases, and passes them with the commands to the Wrappers for the authorization.

Since a multi-threaded process may be a long procedure, for example, it may take several days to complete the tasks, it is difficult for the user to be aware of when he has to ask for a passphrase from which file's owner. This is another problem that arises with the current file access control in the VADOR framework.

3.2 Preview of Security Defects in the VADOR Framework

As any other distributed application, such as the Unix operating system, security defects may also exist in the VADOR framework. If these defects cannot be prevented by appropriate approaches, they may cause serious security problems. In the VADOR framework, the primary security problem is the user data file protection discussed in section 3.1.

This section previews some of the security defects that may exist in the VADOR framework regarding user data protection, and analyzes the causes of the defects and their effects.

3.2.1 Improper Use of Multiple Access Points Control

The first security defect of the VADOR framework is that the VADOR framework has multiple access points.

The main reason lies in the fact that VADOR consists of many servers. These servers are the components of the framework, they may reside on different machines, or the same machine on different ports, and each of them provides specific services to the VADOR system.

When a user requires to execute a task using the VADOR system, the task will be passed through every server that provides access to a separate service, so that the user can get access to each of the points on the entire system. For example, a task first gets to the Executive server to be executed, and than to the Librarian Server to obtain stored data component information, and back to the Executive for further processing, and onto a Wrapper server for final execution.

These multiple access points may possibly allow users to get through a back door and allow them to view or edit sensitive data. It is thus difficult to control information flow and secure the system.

Allowing only the user VADORADM to gain access to data files is the original idea for controlling access to data files from the multiple access points, but because of the improper control mechanism, it caused the fundamental and other potential security problems introduced in section 3.1.2.2 and 3.1.2.3.

3.2.2 Improper Error Checking

This can be considered as a combination of authentication and validation defects, such as Unvalidated Parameters and Broken Access Control vulnerabilities defined by the OWASP project (OWASP (2003)).

The reason is that the security policy was not available when the VADOR framework was initially designed. However, the system needs to be secured from break-in attempts, and needs to take actions depending on the severity of a mistake that the user could make, so that lots of checking code may be needed to authenticate or validate a user in order to protect user data from unauthorized access.

This error checking code can make it difficult to debug and maintain the system, in the sense that a security policy may be defined at a later point and changed over the life of

the system.

3.2.3 Improper Multi-User Profiles Management

The management of multi-user profiles is another potential security defect of the VADOR framework.

The VADOR framework is a multi-user application, its users may share similar (e.g., groups of users) or have individual security profiles in order to access shared or individual data files, and their profiles may overlap or change over time.

Because a system administrator needs to manage security permissions for users depending on their profiles, when the number of users is large, it is hard to customize and manage security profiles for each person.

3.2.4 Improper Global Information Access Control

This defect occurs when there are many objects that need to access shared values, but these values are not unique through the system.

This is a problem related to VADOR's multi-user and multi-threaded characteristics. In order to secure the system, the VADOR system keeps track of global information for each thread or process, such as username, or their respective privileges, the information is then stored in a private, single global location. But when the threads or processes share a common global address space for task execution, their private global information cannot be shared. The defect may also make code and APIs very complex because of the passing around of many objects.

This defect causes the problems of the multi-threaded processes discussed in section

3.1.2.5.2.

3.2.5 Improper Exception Handling

Improper exception handling, or error handling may occur during normal operation, but the source lies in an improper design.

Because VADOR users may have different privileges to access data files, they should be prevented from viewing information that they do not have permission for. In that context, an attacker may attempt to perform some illegal operation to gain access to protected information, while at the same time, an authorized user may also perform improper operations. It then becomes difficult to distinguish between the two conditions, so that exception handling designed to protect from an attacker may be improper for a legal user.

The condition code to determine which operation is legal could be very complex and difficult to test. If an exception cannot be handled properly, users may get confused on what is available to them, it may also disclose information to attackers, or crash a server.

This defect maybe prevented using *Full View with Errors* pattern or *Limited View* pattern. The *Full View with Errors* may notify users with error message when they perform illegal operations without unnecessary revealing internal details. The *Limited View* only lets users see what they have access to (See section 2.2.3.1 for detailed description).

3.2.6 Improperly Integrated External Security System

The VADOR application needs to communicate with other pre-existing systems. If the integration of security mechanisms with the systems is improperly planed, it may become the weakest security point and the most susceptible to break-ins. For example, VADOR uses the MySQL database system on a remote server to store meta data, and the user data

files that correspond to the meta data are stored on hosts on behalf of VADOR. If break-in attacks exploit defects in the integration with the database, this may lead to risks to the user data files.

The reason of this improper integration generally lies in the interfacing problems with the external security systems which is sometimes difficult and often not well documented with respect to security. In addition, external systems may not have sufficient security. If the developers put checks in the application wherever it communicates with other systems, code will become very difficult to maintain.

The *Secure Access Layer* pattern may help to prevent this defect with building application security arround existing operating system, networking, and database security mechanisms, or building own low-level security mechanism on top of the low-level security, then build a secure access layer for communicating in and out of the program (See section 2.2.3.1 for detailed description).

3.3 Summary

The VADOR framework is a multi-user, multi-threaded, mobile agent based distributed application with multiple access points. As for other distributed applications, such as the UNIX operating system, security defects need to be prevented using appropriate approaches during early stage of the development.

In order to maintain the two aspects of the VADOR framework: one Executive server for the system and one Wrapper server on each machine, and the data consistency, the fundamental and other potential VADOR security problems may be introduced by exploring security defects existing in the VADOR system.

This chapter previewed several security defects that exist in the VADOR framework, in-

cluding *Improper Multiple Access Points Control*, *Improper Error Checking*, *Improper Multi-User Profiles Management*, *Improper Global Information Access Control*, *Improper Exception Handling*, and *Improper Integrated External Security System*.

As mentioned, the *Improper Multiple Access Points Control* defect may be exploited and lead to risks to the agent or user data files, and is considered to constitute the most fundamental security problem of VADOR (See section 3.2.1 for details). Furthermore, the objective of VADOR security is to first solve fundamental security problems rather than other potential problems, thus, this work will mainly focus on the prevention of *Improper Multiple Access Points Control* rather than the other defects.

Although the SSH and Expect scripts have been used in the system for resolving the data file access problems, they make it difficult to integrate SSH into VADOR, and to control multi-threaded processes.

To solve the security problems and protect the VADOR system from threats, first, the security defects should be addressed, and then they should be prevented using appropriate design approaches. Earlier prevention of the security defects, less cost and better results in building the system, so that it will be very effective to develop defects prevention design approaches using design pattern concepts in earlier stage of the system development.

A *Security Manager* pattern is proposed based on a study of actual security defects existing in the distributed system, that uses security design patterns. The objective of developing the Security Manager is to prevent security defects in the VADOR framework, protect the system from threats, and solve the security problems regarding data file access in VADOR.

Chapter 5 will introduce the Security Manager pattern in details.

CHAPTER 4

SECURITY MODEL - FROM JAVA TO THE VADOR FRAMEWORK

As introduced in section 1.2, the VADOR System is an agent based distributed system that represents a specialized *Active Agent* pattern. It is implemented on top of the JVM, using the Internet as its major transmission media component.

In this context, risk exists that information being transmitted via the Internet, agents or data files could be threatened. The main reason for this lack of security is that improper system design may lead to security defects, that threats could exploit to attack the system.

Security defects in the VADOR framework have been studied in chapter 3. This chapter will introduce a security model and mechanisms in the VADOR framework, that are used to prevent security defects and protect the VADOR System from threats.

Section 4.1 will introduce the Java 2 SDK security architecture, that provides low level protection mechanisms to the VADOR System. The *VADOR Security Model* will be introduced in section 4.2, it constitutes the *Security Manager* in the *Active Agent* pattern (Section 1.2.2), as well as the *Security Manager* pattern, that will be developed in chapter 5.

4.1 Security in Java 2 SDK

4.1.1 J2SDK Security Features Overview

The Java 2 SDK (J2SDK - Java 2 Standard Development Kit) security architecture is policy-based, and allows for fine-grained access control. When code is loaded, it is as-

signed "permissions" based on the security policy currently in effect. Each permission specifies a permitted access to a particular resource, such as "read" and "write" access to a specified file or directory, or "connect" access to a given host and port. The policy, specifying which permissions are available for code from various signers/locations, can be initialized from an external configurable policy file. Unless a permission is explicitly granted to code, it cannot access the resource that is guarded by that permission. These new concepts of permission and policy enable the SDK to offer fine-grain, highly configurable, flexible, and extensible access control. Such access control can be specified for applets and all other Java code, including applications, beans, and servlets.

The Java Security API is a Java core API, built around the *java.security* package (and its subpackages). The first release of Java Security in JDK (Java Development Kit) 1.1 contains a subset of cryptographic functionality, including APIs for digital signatures and message digests. In addition, there are abstract interfaces for key management and certificate management.

JDK 1.2 contains substantial security features enhancements based on the JDK 1.1: policy-based, easily-configurable, fine-grained access control; new cryptographic services, new certificate and key management classes and interfaces; three new tools (*keytool*, *jarsigner*, and *policytool*) have been added for key management, signature generation and verification, and security policy management.

In JDK 1.3, several security enhancements have been made to the cryptographic services and the security tools.

Security enhancements for JDK 1.4 include providing support for dynamic policies, adding several packages and APIs for Certification, Authentication, and Cryptographic services, and three new tools (*kinit*, *klist*, and *ktab*) have been added for obtaining, listing, and managing Kerberos tickets. Instead of loading security policies through a class loader and binding them to the class loader's lifetime, the support for dynamic policies al-

lows dynamically querying of security policies when they are needed by security checks.

4.1.2 J2SDK Security Models

Access control has evolved to be far more fine-grained than in earlier versions of the Java platform since JDK 1.2. This section introduces security models in the Java *Security Architecture Extensions.*

4.1.2.1 The Original Sandbox Model

4.1.2.1.1 JDK 1.0 Security Model

The original security model provided by the Java platform is known as the *sandbox model,* which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. This model is illustrated in figure 4.1.

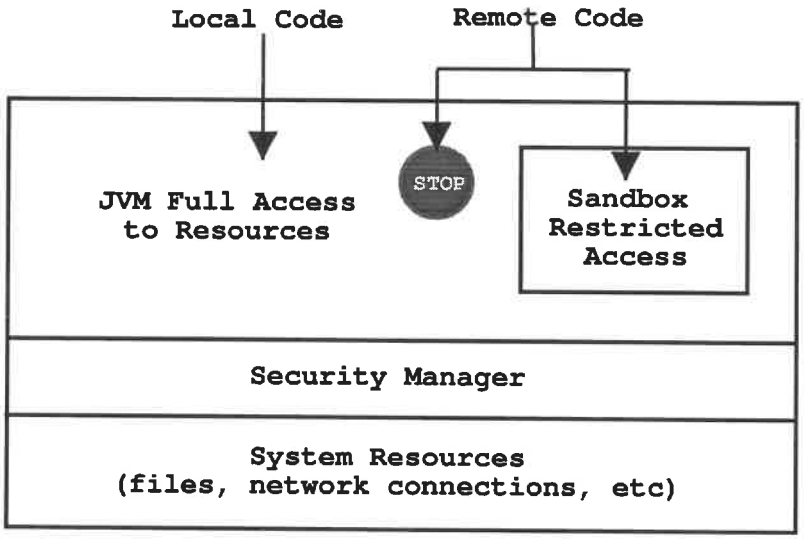


FIGURE 4.1 JDK 1.0 Security Model

Participants in this model are listed as the following:

- **Local Code:** is trusted code that has full access to vital system resources, such as the file system.
- **Remote Code:** is downloaded code (an applet) that is not trusted and can access only the limited resources provided inside the sandbox.
- **JVM:** Java run time system that organizes trusted code to access resources and untrusted code that is limited to the sandbox.
- **Security Manager:** is a class that is responsible for determining which resource accesses are allowed in the security model and subsequent platforms.
- **System Resources:** are vital system resources that include files, network connections, etc. Access to crucial system resources is mediated by the JVM and is checked in advance by the Security Manager class that restricts the actions of a piece of untrusted code to the bare minimum.

4.1.2.1.2 JDK 1.1 Security Model

JDK 1.1 introduced the concept of "signed applet". In this model, signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. As illustrated in figure 4.2, a digitally signed applet is treated as local code, with full access to system resources, if the public key used to verify the signature is trusted by the end system that receives the signed applet. Unsigned applets still run in the sandbox.

A new participant has been added into this model:

- **Trusted Signed Code:** is remote code that was signed with the sender's private key and verified using a trusted public key by its receiver. It is treated as local code that has full access to system resources.

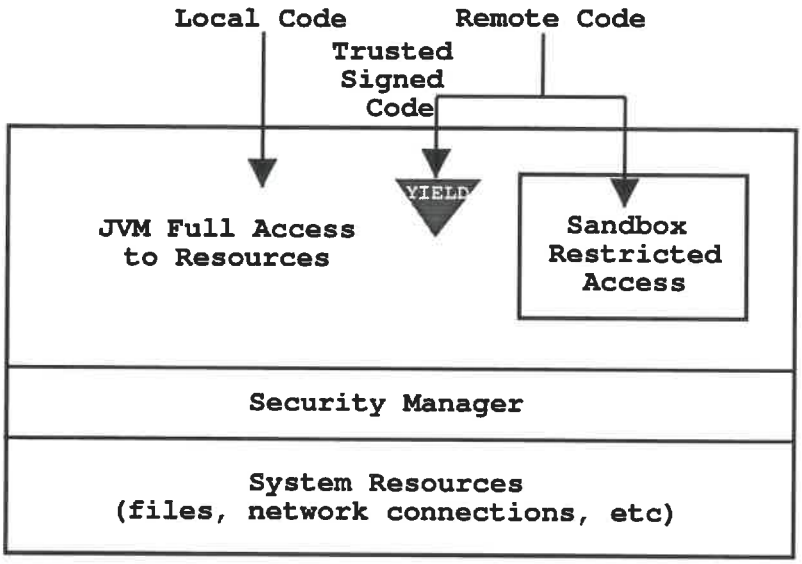


FIGURE 4.2 JDK 1.1 Security Model

4.1.2.2 The Current Security Model

JDK 1.2 introduced a number of improvements over JDK 1.1, and the later versions have done many enhancements to the new security architecture introduced in JDK 1.2. The current J2SDK security model is illustrated in figure 4.3. This model is introduced for the following purposes:

- Fine-grained access control.
- Easily configurable security policy.
- Easily extensible access control structure.
- Extension of security checks to all Java programs, including applications as well as applets.
- Make internal adjustment to the design of security classes. (including the SecurityManager and ClassLoader classes) to reduce the risks of creating subtle security holes in future programming.

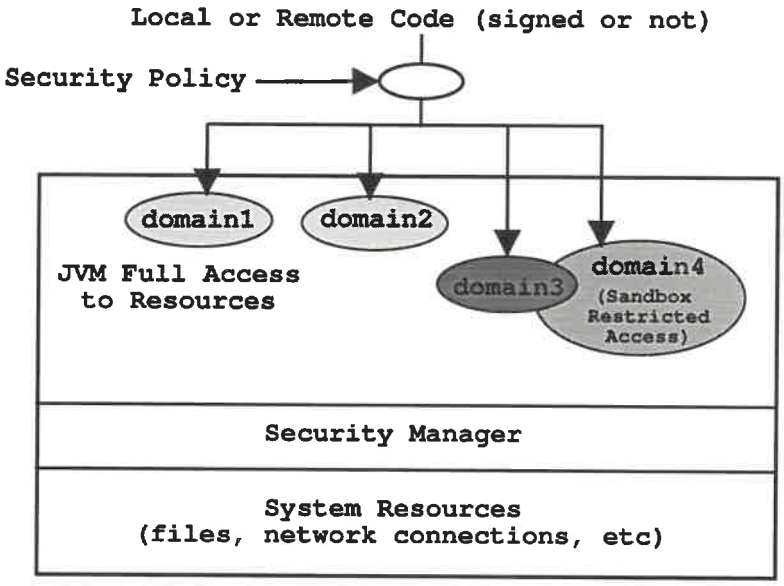


FIGURE 4.3 The Current J2SDK Security Model

In the current J2SDK security model, code runs with different permissions, and there is no built-in notion of trusted code, all code, regardless of whether it is local or remote, can be subject to a *security policy*.

Two participants have been added to this model:

- **Security Policy:** defines the set of *permissions* available for code from various signers or locations and can be configured by a user or a system administrator. Each permission specifies a permitted access to a specified file or directory or connect access to a given host and port. Security policies are queried dynamically while they are needed by security checks.

- **Domains:** contain code organized by the run time system, each of which encloses a set of classes whose instances are granted the same set of permissions. As illustrated in figure 4.3, code belonging to domain1 and domain2 is granted full access to resources. The domain4 code is restricted exactly the same as the original sandbox. The domain3 code lies in between, it has more accesses allowed than the

sandbox, but less than full access.

4.1.3 J2SDK Protection Mechanisms

The protection domain concept serves as a convenient mechanism for grouping and isolation between units of protection. A *Domain* can be scoped by the set of objects that are currently directly accessible by a *Principal*, where a *Principal* is an entity in the computer system to which permissions, and as a result, accountability, are granted. The sandbox utilized in JDK 1.0 is one example of a protection domain with a fixed boundary, existing object accessibility rules remain valid under the current security architecture.

Protection domains generally fall into two distinct categories: *system Domain* and *Application domain.* The domain composition of a Java application environment is illustrated in figure 4.4. It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, be accessible only via system domains.

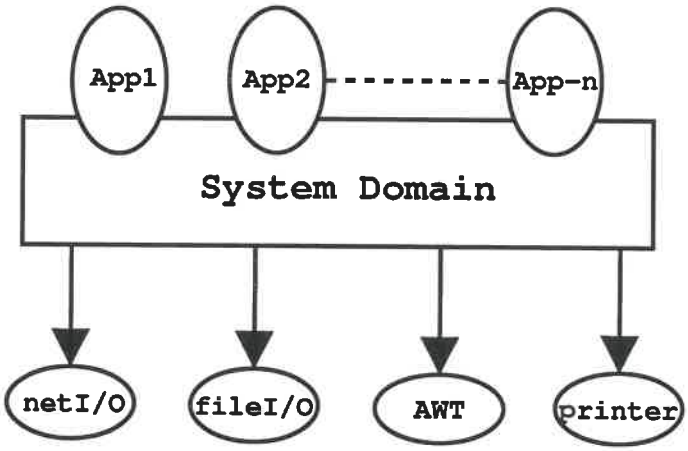


FIGURE 4.4 The Domain Composition of a Java Application Environment

The Java application environment maintains a mapping from code (classes and instances) to their protection domains and then to their permissions. The context of this mapping is

that a domain conceptually encloses a set of classes whose instances are granted the same set of permissions. Protection domains are determined by the policy currently in effect. This mapping is illustrated in figure 4.5.

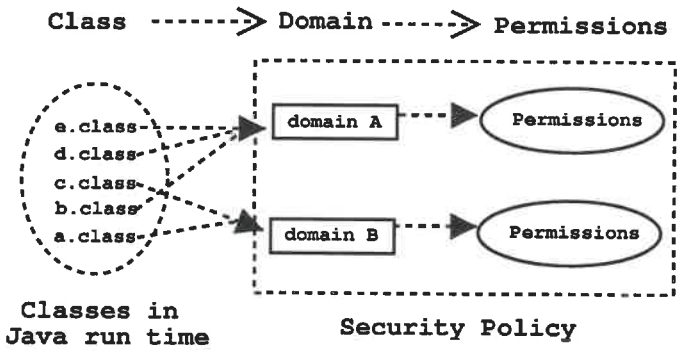


FIGURE 4.5 The J2SDK Protection Mechanisms

Each domain (system or application) may also implement additional protection of its internal resources within its own domain boundary. For example, the VADOR application needs to support and protect internal concepts such as DCInstances, creations and retrieving. Because the semantics of such protection is unlikely to be predictable or enforceable by the Java 2 SDK, the protection system at this level is left to the application developers. Nevertheless, the J2SDK provides helpful primitives to simplify the developers' tasks. One such primitive is the *SignedObject* class.

4.2 Security in the VADOR Framework

4.2.1 The VADOR Security Features Overview

In addition to the policy-based, easily-configurable, fine-grained access control derived from Java (4.1.1), the VADOR security architecture also provides Certificate and Key Management, and SSH based Data File Management.

Different from the J2SDK, the VADOR security requires multi-level security policies, each level being responsible of a VADOR Server. For example, the first level policy is responsible of the Security Server. The Executive Server and Librarian Server also have their own policies. The policies, specifying which permissions are available from various agent senders, can be defined and initialized by the VADOR administrator when the VADOR servers are started. Unless a permission is explicitly granted to an agent, it cannot access the resource that is granted by that permission. These concepts of permission and policy enable VADOR to offer easily-configurable, flexible, extensible and fine-grained access control.

The VADOR security uses *Keytool* provided by Java for Key generation and Certificate management. The VADOR administrator is responsible for generating keys and managing certificates for every VADOR user using *Keytool*.

SSH is used for system data file Management in VADOR. This means that instead of controlling accesses to data files through a VADOR external security policy, the framework controls the accesses using SSH within the operating system. This concept has been introduced in chapter 3.

4.2.2 The VADOR Security Model

Associated with the current J2SDK security model, the VADOR security model provides multilevel protections to the VADOR framework, including multilevel security policy enforcement, security managers, and protection domains. Figure 4.6 illustrates this model.

The VADOR security model has the following protection levels and participants:

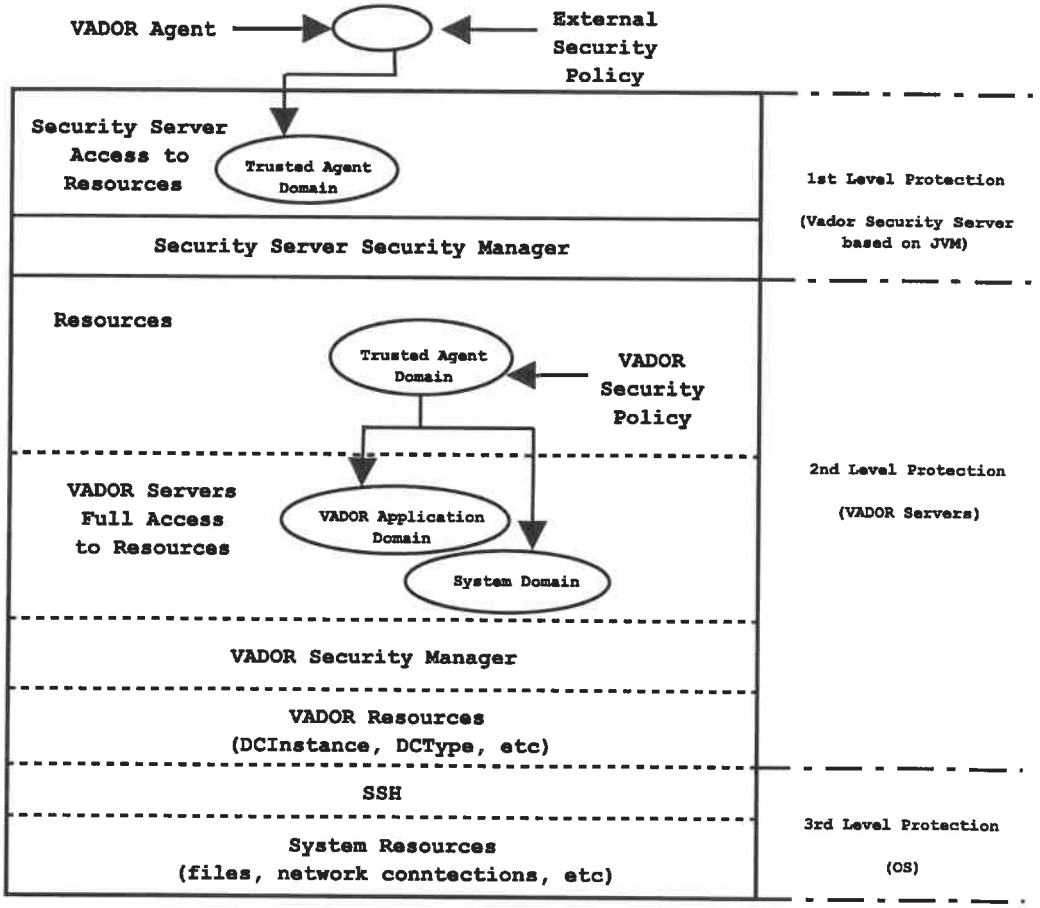


FIGURE 4.6 VADOR Security Model

4.2.2.1 1st Level Protection: Security Server

Protection domains in this level are organized by VADOR Security Server based on the JVM.

- **VADOR Agent:** is remote code that is signed by a VADOR client with its private key, and sent by the client to execute on a specified host on behalf of the VADOR system.
- **External Security Policy:** defines the *permissions* available for the signed VADOR agent and can be configured by the VADOR administrator. These permissions spec-

ify permitted connecting accesses to hosts and ports that will allow the agent to work.

- **Security Server:** is based on the Java run time system to organize trusted agent code to access resources.
- **Trusted Agent Domain:** contains the agent code organized by the JVM and has been granted the permission specified by the *External Security Policy*. A trusted agent domain has the right to access resources, but cannot be guaranteed full access, because the permissions are specified by the second level *VADOR Security Policy* for accesses to the VADOR and system resources.
- **Security Server Security Manager:** a class that is responsible of verifying and authenticating a signed agent.

4.2.2.2 2nd Level Protection: VADOR Servers

VADOR Servers organize protection domains at this level to permit or limit accesses to the Resources that include both VADOR Resources and System Resources.

- **Resources:** include both resources in the VADOR system such as meta-data of *DCInstance or DCType*, and System Resources such as *files*, or *network connections.*
- **VADOR Security Policy:** is the second level security policy in the VADOR security model. It defines the set of permissions available for the trusted agent and can be specified by the VADOR administrator. Each of these permissions specifies a permitted access to VADOR Resources associate to data files or directories stored as System Resources.

- **VADOR Servers:** organize trusted code to access the VADOR Resources and/or System Resources.
- **VADOR Application Domain:** contains trusted code organized by the VADOR Servers and has full access to the VADOR Resources.
- **System Domain:** contains trusted code organized by the VADOR Servers and has full access to the System Resources.
- **VADOR Security Manager:** is a class defined in the VADOR security model that is responsible for determining which resource accesses within the VADOR system are allowed.
- **VADOR Resources:** are internal resources in the VADOR framework. They include meta-data information stored in the database such as DCInstance and DCType. They are associated with System Resources.

4.2.2.3 3rd Level Protection: Operating System

In addition to the VADOR server protection, the System Resources have an extra level of protection enforced by the operating system using SSH. The participants in that level are:

- **System Resources:** include files, network connections, etc. The System Resources in the VADOR framework are also considered as internal resources.
- **SSH:** Secure Shell system that functions as the third level security policy and security manager. It controls accesses to System Resources by VADOR applications via the System Domain.

4.2.3 The VADOR Protection Mechanisms

As introduced in the J2SDK protection mechanisms (section 4.1.3), there are two protection domains: *Application Domain* and *System Domain*. A *Domain* is the set of objects that are currently directly accessible by a *Principal*, where a *Principal* is an entity in the computer system to which permissions are granted. *Application Domains* can only access external resources, such as the file system and network facility, via *System Domains*.

Based on the above architecture, VADOR framework security architecture organizes the protection domains into three categories: *Trusted Agent Domain*, *VADOR Application Domain*, and *System Domain*. The domain composition of the VADOR framework is illustrated in figure 4.7. The VADOR Agent can only access resources via the Trusted Agent Domain. Internal VADOR resources, such as DCInstance, DCType, are accessible only via the VADOR Application Domain, and the external system resources, such as file system and network facility are accessible only via the System Domain.

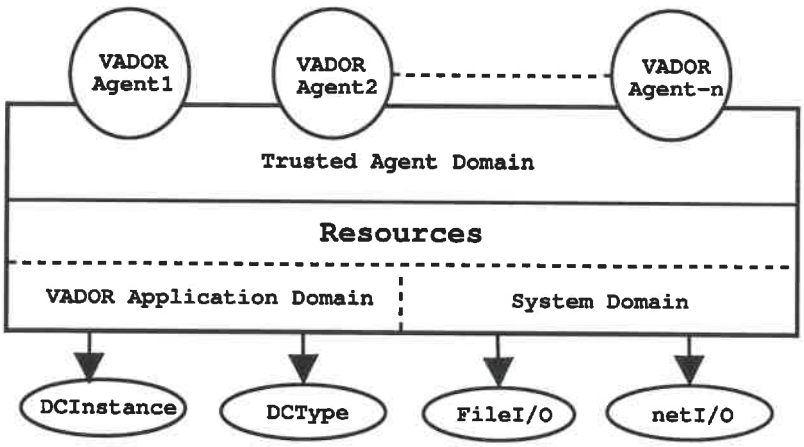


FIGURE 4.7 The Domain Composition of the VADOR Framework

As a Java application, the VADOR framework also maintains mapping from code (classes and instances) to their protection domains and then to their permissions. However, as illustrated in figure 4.8, additional protection of resources has been implemented. A multi-

level mapping has been defined, and multilevel protection domains have been introduced in the VADOR framework.

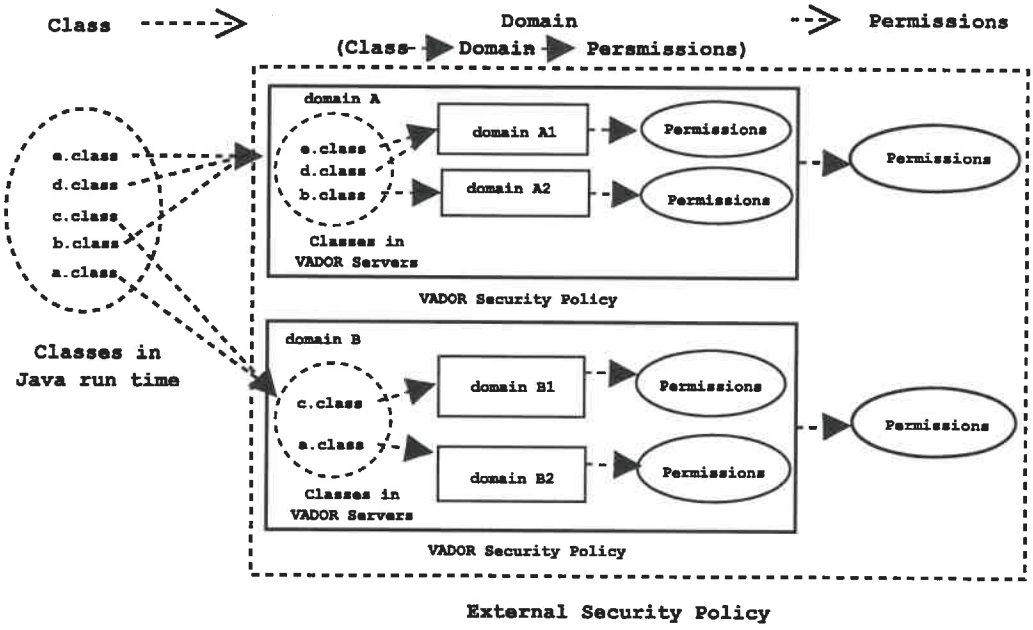


FIGURE 4.8 The VADOR Protection Mechanisms

In figure 4.8, Trusted Agent Domain A and B are determined by the *External Security Policy* currently in effect, these domains enclose the classes in Java run time, their instances are granted the same set of permissions which are specified by the external security policy.

In order to access the internal VADOR Resources or System Resources, instances of the classes in the domain A or domain B need to be granted the same set of permissions which are specified by the second level VADOR Security Policy. This security policy determines the domain A1, A2, B1, or/and domain B2, these domains enclose the classes in the VADOR servers and could be either VADOR application domains that can access VADOR resources, or System Domains that can access system resources.

4.3 Summary

This chapter has introduced the *VADOR Security Model* that represents the *Security Manager* which participates in the *Active Agent* pattern, as well as the *Security Manager* pattern which will be further developed in chapter 5.

Based on the *J2SDK Security Extension*, which provides the low level protection mechanisms, the VADOR Security Model defines multi-level protection for the VADOR framework: *Security Server* level, *VADOR Server* level, and *Operating System* level. Each level implements its own protection mechanisms to prevent security defects (Chapter 3.2) in the VADOR framework, and protect the VADOR system from threats (Section 3.1).

The multi-level protection mechanisms constitute the most significant VADOR security feature. This protection is policy-based, easily-configurable, and provides fine-grained access control, cryptographic services based on secure key and certificate, and SSH based data file management.

Chapter 5 will move on to introduce of the *Security Manager* pattern, which implements the *VADOR Security Model.*

CHAPTER 5

THE SECURITY MANAGER PATTERN

In the VADOR global architectural design, most of the functionalities to process data and tasks within the framework reside in the *Application Domain Layer* (Section 1.2). This layer constitutes the core of the VADOR System, it is implemented using a specialization of the *Active Agent* pattern.

However, in the VADOR prototype, the VADOR System focused on solving problems related to concurrency, scalability, and flexibility of the framework. Although a *Security Manager* had been planned to participate in the Active Agent pattern, it was not implemented.

As the needs related to security issues increased during the release of the VADOR framework for industrial usage at Bombardier Aerospace, security policy enforcement and system protection processes have become required.

Based on the above context, the *VADOR Security Model* (Chapter 4) is designed to fulfill the security requirements of the VADOR System. It is implemented using a specialization of the *Security Manager* pattern, that corresponds to the Security Manager in the Active Agent pattern.

5.1 Active Agent Pattern with Security Manager

When the Security Manager participates in the Active Agent pattern (see section 1.2.2 for its full description), the *Security Server* becomes an intermediate Execution Place, which cannot be by passed. In other words, no matter which Vador Server an Agent object

migrates to for its execution, at first, it must be authenticated and verified by the Security Server. Then, when the Vador Server invokes the authenticated agent's *call()* function, and before starting task execution, the Security Manager, on behalf of the server, should check that the agent's permission allows this task execution. As for the Security Server, the Security Manager cannot be by passed.

The first level and second level protection introduced in the VADOR Security Model (Section 4.2.2) are implemented as the Security Server and the Security Manager.

5.1.1 The Extended Interaction Between Participants

In the original active agent collaboration illustrated in figure 1.4, in order to accomplish tasks, clients, such as the VadorGUI, create agents and send them to an execution place (the Vador Server), which calls the standard agent operation (call() function). Depending on the agent's security policy, the operation is executed, or not, on the related agent instance (See Chen (2004) for details).

When the Security Manager pattern participates in the Active Agent pattern, an additional Execution Place (Security Server) is added to the Application Domain layer. It specifically provides authentication and verification services to the VADOR framework. Figure 5.1 illustrates the extended Active Agent pattern interaction.

5.1.2 The Extended Active Agent Dynamic Behavior

The Active Agent dynamic behavior (see section 1.2.2.3 for details) also has been extended by the Security Manager's participant. Agent signature, authentication, and permission checking processes have been added to the three phases. The details of this extension are illustrated in figure 5.17 (See page 101).

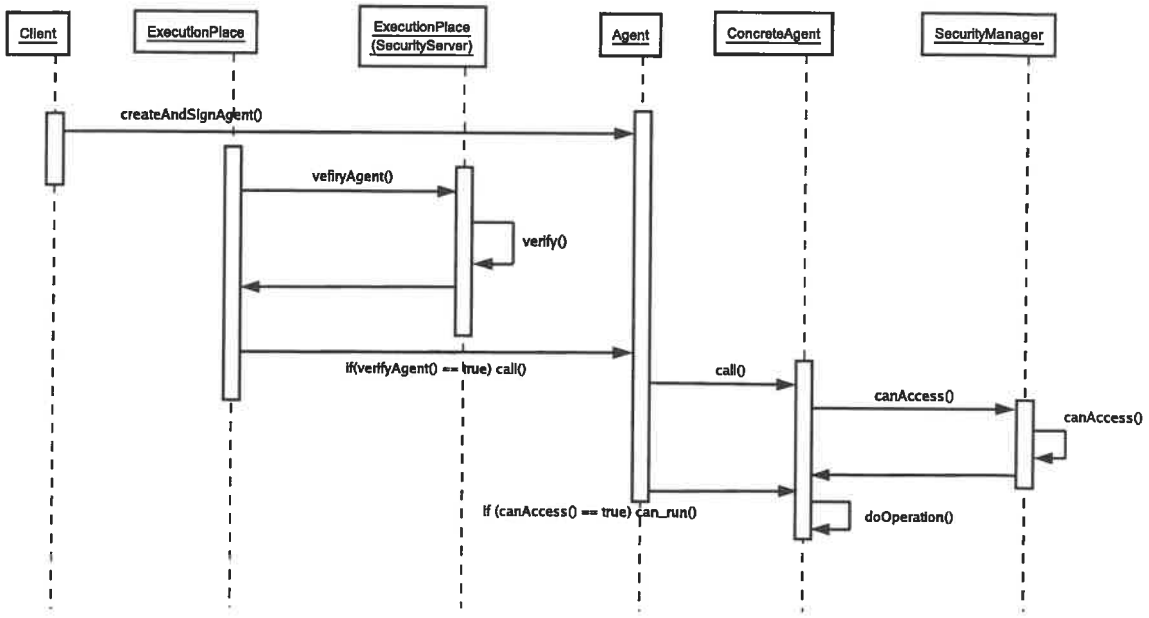


FIGURE 5.1 Extended Active Agent Pattern Interaction Diagram

1. **Agent Creation, Signature, and Sending**

   A client creates and signs an Agent (Original Agent) object, that includes a unique identifier (id) of the Original Agent and an id of the Signed Agent. The Signed Agent then migrates to an Execution Place (Vador Server) via the Vador Proxy.

2. **Agent Authentication, Permission Checking, and Execution**

   - *Agent Authentication:* Before the Vador Server invokes the Original Agent's *call()* function to start task execution, it must authenticate and verify the Signed Agent object. So it creates and signs a *Security Agent* object, and includes the Security Agent id and the Signed Agent id in it. The Security Agent object then migrates to its Execution Place (the Security Server) where a Security Manager resides, and performs two authentication processes based on the agents' ids: *Security Agent Authentication* (See section 5.3.4 for details) and *Signed Agent Authentication* (See section 5.3.6 for detailed description).

- *Agent Permission Checking:* If both the Security Agent and the Signed Agent were verified, the Vador Server can invoke the Original Agent's *call()* function. Before the task execution starts, the Security Manager, on behalf of the Vador Server, enforces *VADOR Security Policy* to check that the agent's permissions correspond to its task execution request.
- *Agent Execution:* If the request is granted, the agent will dynamically load the concrete Vador Visitor class and use it to execute the task. If the task requires access to system data, before it is executed, the SSH subsystem will perform an authentication process to verify the agent's sender (Vador User) at the operating system level. The agent then can gain access to the system data only if its sender has been authenticated.

3. **Completion**

   The execution result is sent back to the client by the Vador Server.

5.2 Security Manager Pattern

Chapter 4 introduced the VADOR security model and mechanisms to protect the system. This chapter will present the *Security Manager* pattern , that is a *Security Pattern System* represents the structure of the security model, and is implemented in the VADOR security system.

The *Security Manager* pattern is a combination of three structural security design patterns: *Protected System*, *Partitioned Application*, and *Multilevel Security*. These patterns work together to help the Security Manager build the VADOR security architecture, and divide it into several independent modules, easier to validate.

The pattern description is based on the format introduced in section 2.2.2.2.

5.2.1 Name

Security Manager

5.2.2 Context

The VADOR system is a multi-user, multi-threaded, client-server architecture, mobile agent based, distributed application. The characteristics of VADOR require that the system be capable of allowing access to many users for working on different machines, to a limited set of resources, on behalf of the system at the same time. This capability also includes protecting the system resources from attacks. Specifically, preventing security defects from system design, so that it can control accesses to resources, and stop attackers from exploiting the defects.

5.2.3 Problem

1. An Active Agent may be attacked and its status may be changed during its migration, a malicious agent can then attack data files.
2. It is difficult to validate an Active Agent on behalf of a Vador Server, because VADOR consists of many servers, and each of them represents an access point.

5.2.4 Solution

1. An Active Agent sender signs the agent using its private key before sending it. An agent receiver applies Security Server to authenticate and verify the agent using a certificate that corresponds to the public key before allowing the agent to execute tasks.

2. Each Vador Server owns a Security Manager for checking permissions that are defined for this server by VADOR Security Policy. If the agent's requests are permitted, the agent can then execute tasks on the Vador Server.

5.2.5 Structure

Figure 5.2 illustrates the structure of the Security Manager Pattern.

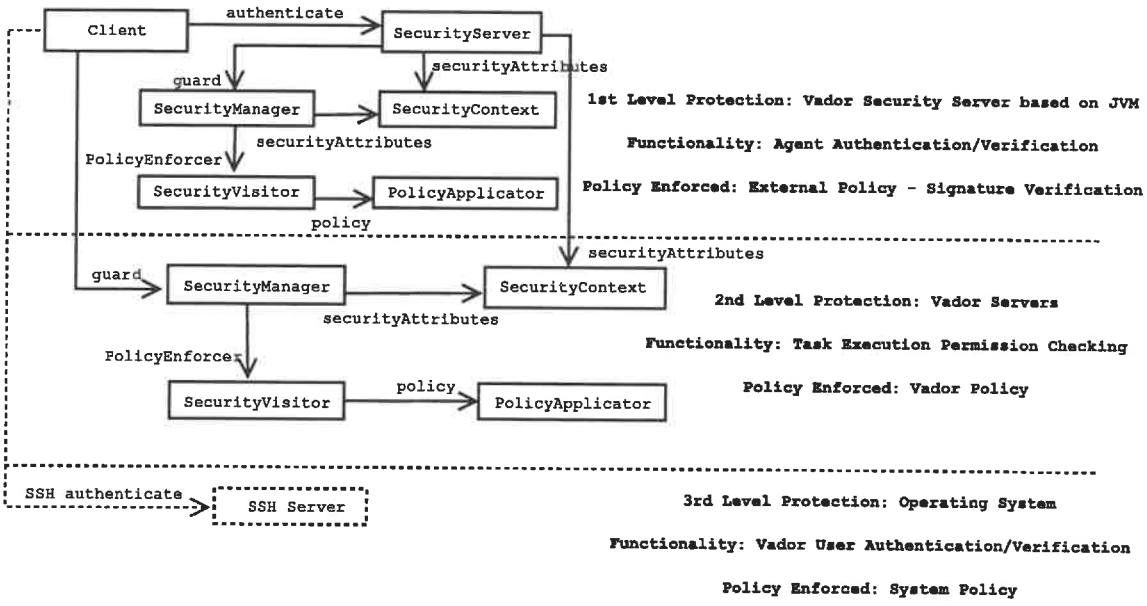


FIGURE 5.2 Structure of the Security Manager Pattern

5.2.6 Participants

- **Client**
  - Any VADOR User or Vador Server that creates, signs, and sends active agents, or any Vador Server that receives, authenticates, verifies, and checks permissions on a signed agent .

- An agent sender client sends the signed agent with a unique identifier of the original agent and an identifier of the signed agent.
- An agent receiver client verifies the signed agent using its identifier and certificate. The certificate is defined by the administrator and corresponds to the public key that is in pair with the private key, which the sender client uses to sign the agent.

- **SecurityServer**
  - It is an Execution Place that is responsible of a signed agent authentication and verification.
  - Its SecurityManager implements and extends the same interfaces and abstract classes as the security manager on the other execution places.
  - It enforces External Policy that is defined by administrator and initialized when the Security Server starts.

- **SecurityManager**
  - Every Execution Place has its own SecurityManager that implements and extends the same interfaces and abstract classes.
  - The security managers on behalf of the execution places represent their security guards, and are responsible of security issues. For example, the Security Server's Security Manager is responsible of agent authentication and verification, the other VADOR Servers' Security Managers are responsible of permission checking before allowing an agent to execute tasks.

- **SecurityContext**
  - Every Execution Place has its own Security Context that implements and extends the same interfaces and abstract classes.

- The Security Context on behalf of the execution place represents its security attributes, including user attributes, subject attributes, and object or information attributes.

- **SecurityVisitor**
  - Every Execution Place has its own Security Visitor that implements and extends the same interfaces and abstract classes.
  - The Security Visitor, on behalf of the execution place, represents its security policy enforcer, that is responsible of checking permissions according to the policy which it enforces.

- **PolicyApplicator**
  - Every Execution Place has its own PolicyApplicator that implements and extends the same interfaces and abstract classes.
  - The Policy Applicator, on behalf of the execution place, represents its security policy, that is responsible of performing actual algorithms for checking if execution requests match the permissions that are defined by the policy.

- **SSH Server**
  - A security server resides on the operating system and provides the lowest level protection of VADOR.
  - It is responsible of system policy enforcement to authenticate and verify a VadorUser that creates and sends an agent at the operating system level.

5.2.7 Interaction

Figure 5.3 illustrates interaction of the Security Manager Pattern.

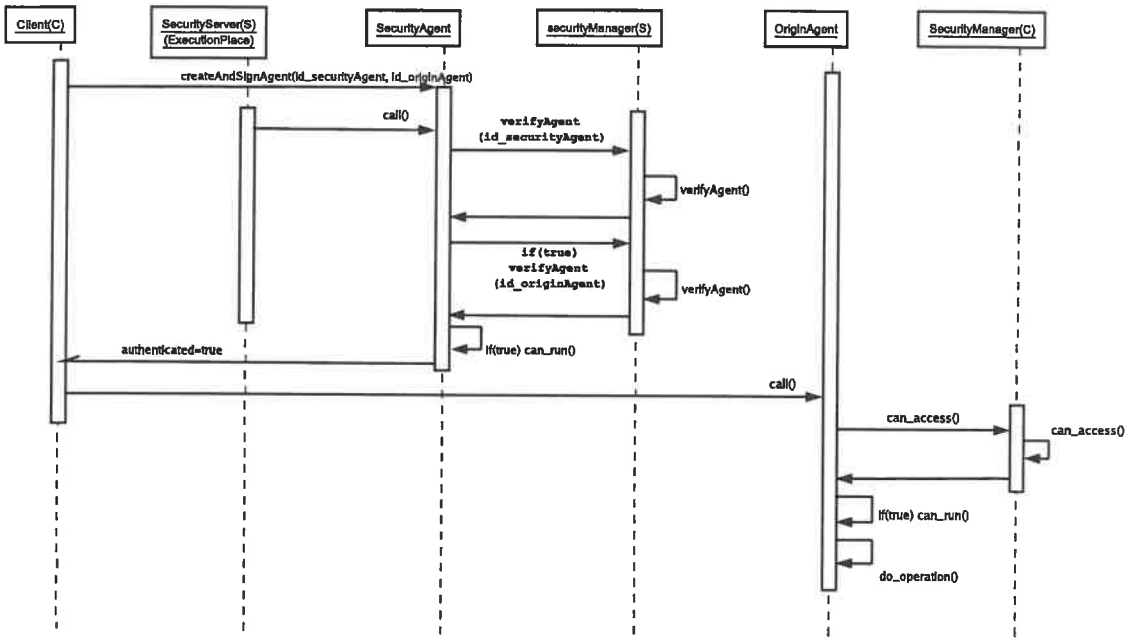


FIGURE 5.3 Security Manager Pattern Interaction

- A Client creates and signs an Active Agent using the Private Key which was defined in the External Security Policy by the administrator (a Certificate that corresponds to the Public Key exported to an Execution Place by the administrator). The client then sends the agent to the Execution Place for task executions.

- When an Execution Place (Vador Server) receives the agent (called Original Agent), before allowing it to execute tasks, it must authenticate the Original Agent and verify the agent's signature. The Vador Server (Client) creates and signs another Active Agent, called Security Agent, in which the Original Agent is included. This client then sends the Security Agent to a specified Execution Place - Security Server for the authentication and verification processes.

- When the Security Server receives the signed Security Agent, it invokes the agent's call() function, this function then calls the Security Server's Security Manager to perform the authentication and verification processes using the Certificates that was imported by the administrator that corresponds to the agent sender's Public Key.

- The Security Agent's call() function requires the Security Server's Security Manager to perform two steps for the agent's verification and authentication: the first, verifies the Security Agent, if the Security Agent could be verified, it then verifies the Original Agent. Figure 5.4 illustrates the verification algorithms.

```
public boolean verify(String certname, SignedObject so)
{
    boolean can_verify;
    try
    {
        FileInputStream certfis = new FileInputStream(certname);
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        Certificate cert = cf.generateCertificate(certfis);
        PublicKey pubKey = cert.getPublicKey();
        Signature verificationEngine =  Signature.getInstance("SHA1withDSA", "SUN");
        if (so.verify(pubKey, verificationEngine))
        {
            can_verify = true;
        }
        else
        {
            can_verify = false;
        }
    }
    catch (Exception ex)
    {
        System.err.println(ex.toString());
    }
    return can_verify;
}
```

FIGURE 5.4 Signed Agent Verification Algorithms

- If both of the Security Agent and the Original Agent can be authenticated and verified, the Security Server sends "authenticated = true" feedback to its sender to inform that the Original Agent's signature has been verified, which means that the agent has not been attacked during its migration. If either of the agents could not be authenticated or verified, they both will be discarded.

- When the Execution Place receives an agent "authenticated = true" feedback, it

invokes the Original Agent's call() function, that calls the Security Manager on behalf of the Execution Place and passes it the Security Context. The Security Manager then applies VADOR Security Policy defined for the Execution Place by the administrator.

- If the agent's Security Context matches the permissions defined by the VADOR Security Policy, the Original Agent can then perform the doOperation() on the Execution Place, otherwise, the agent will be discarded.

5.3 Security Manager Pattern Modules

The VADOR security model implements a specialization of the Security Manager pattern that consists of sets of packages and interfaces, in which the Security Manager pattern modules are represented. These modules are listed as the following:

1. Security Interface Module
2. Agent Authentication Module
3. Agent Signature Module
4. Security Server Module
5. Security Attributes Descriptor Module
6. Security Manager Module

Figure 5.5 illustrates the structure and relationship between the modules.

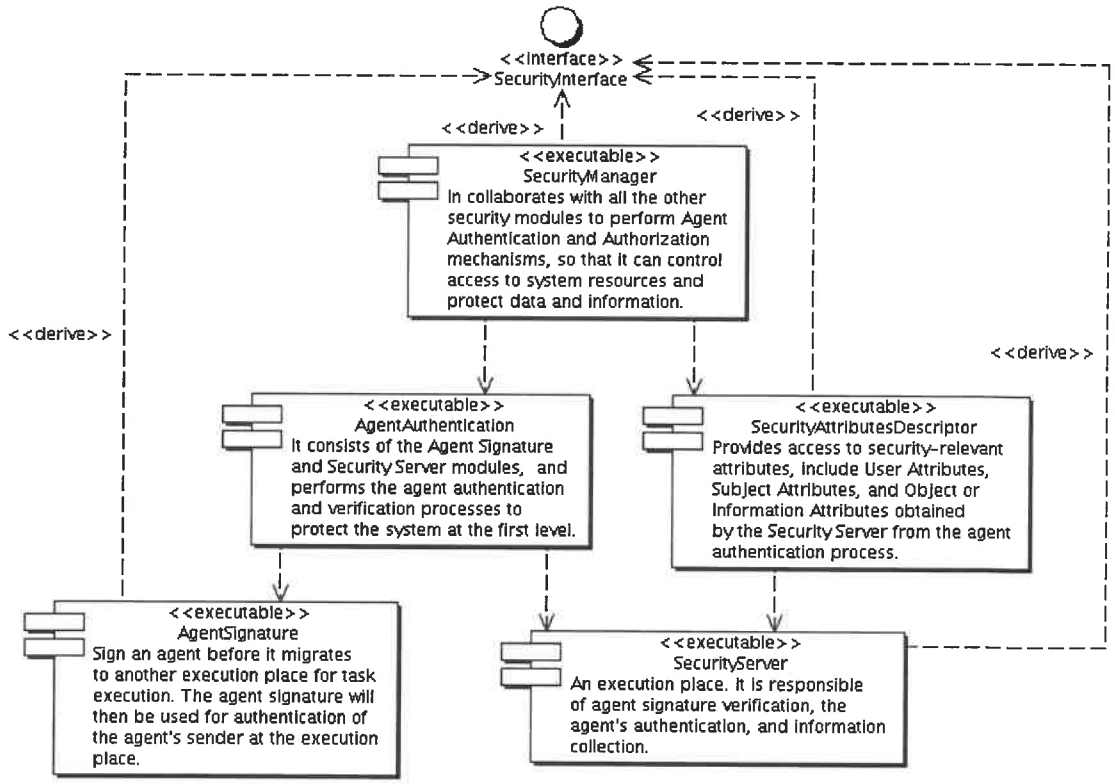


FIGURE 5.5 Security Manager Pattern Modules Structure and Relationship

5.3.1 Security Interface Module

The *Security Interface Module* provides interfaces to all the security related modules in the VADOR system, including the *Agent Authentication* that consists of the *Agent Signature*, *Security Server*, *Security Manager*, and *Security Attributes Descriptor* modules. This module has been integrated into the Active Agent's *NetworkTool Module*, in which a set of VADOR system interfaces has been defined.

Components	Intefaces or Abstract Classes	Services
Interface of Agent Signature	IVadorSigner	Provides interfaces of Agent Signature services to VadorServers for signing agents before sending them to a remote VadorServer for task execution.
	AbstractVadorSigner	
Interface of Security Server	IVadorProxy	Defined in Active Agent's NetworkTools module. Provides an interface of local representitive of the remote Security Server.
	VadorProxy	
	IServerThread	Defined in Active Agent's NetworkTools module. Provides an interface of a Security Server Thread launched by the Security Server.
	ServerThread	
Interface of Security Attributes Descriptor	IVadorSecurityAttributesDescriptor	Provides interface to concret security attributes descriptor classes (i.e, ServerSession) to keep track of security attributes.
	VadorSecurityAttributesDescriptor	
Interface of Security Manager	IVadorSecurityManager	Defines interfaces for Security Managers, SecurityVisitors, and PolicyApplicators to perform access control mechamisms in the Vador System.
	AbstractVadorSecurityManager	
	ISecurityVisitor	
	AbstractSecurityVisitor	
	IPolicyApplicator	
	AbstractPolicyApplicator	

FIGURE 5.6 Security Interface Module Components

5.3.1.1 Components

As illustrated in figure 5.6, the *Security Interface Module* consists in the following components:

- **Agent Signature Module Interface** provides services to VADOR servers to sign an agent before sending it to remote VADOR Servers for task execution.
- **Security Server Module Interface** provides an interface of a local representative of the Security Server to a VadorUser or Vador Server. It provides an abstract ServerThread class to be extended by a concrete SecurityServerThread class, so that it can be launched by the Security Server for executing a specified task.
- **Security Attributes Descriptor Module Interface** provides interface and abstract security attributes descriptor to concrete classes that may keep track of security

attributes. For example, the concrete ServerSession class.

- **Security Manager Module Interface** defines interfaces for *Security Managers*, *SecurityVisitors*, and *PolicyApplicators* that are on behalf of VADOR Servers to perform access control mechanisms to the VADOR system.

5.3.2 Agent Authentication Module

The *Agent Authentication Module* consists of the *Agent Signature* and *Security Server* modules. It is responsible of the first level protection of the VADOR system - Agent Authentication and Verification. The objective is to manage agent signature for the agent's sender, and authenticate the signed agent for the agent receiver. It uses security mechanisms provided by the JVM for agent signature and verification processes.

5.3.2.1 Structure

Figure 5.7 presents the structure of the Agent Authentication Module.

5.3.2.2 Participants

- **Client**
  - Any VadorUser or Vador Server that may create and send an Active Agent to a remote Execution place for task execution.
  - Before sending the agent, it signs the agent by applying its VadorSigner.
- **VadorSigner**
  - On behalf of the Client, applies a Cryptographic Algorithm for agent signature.

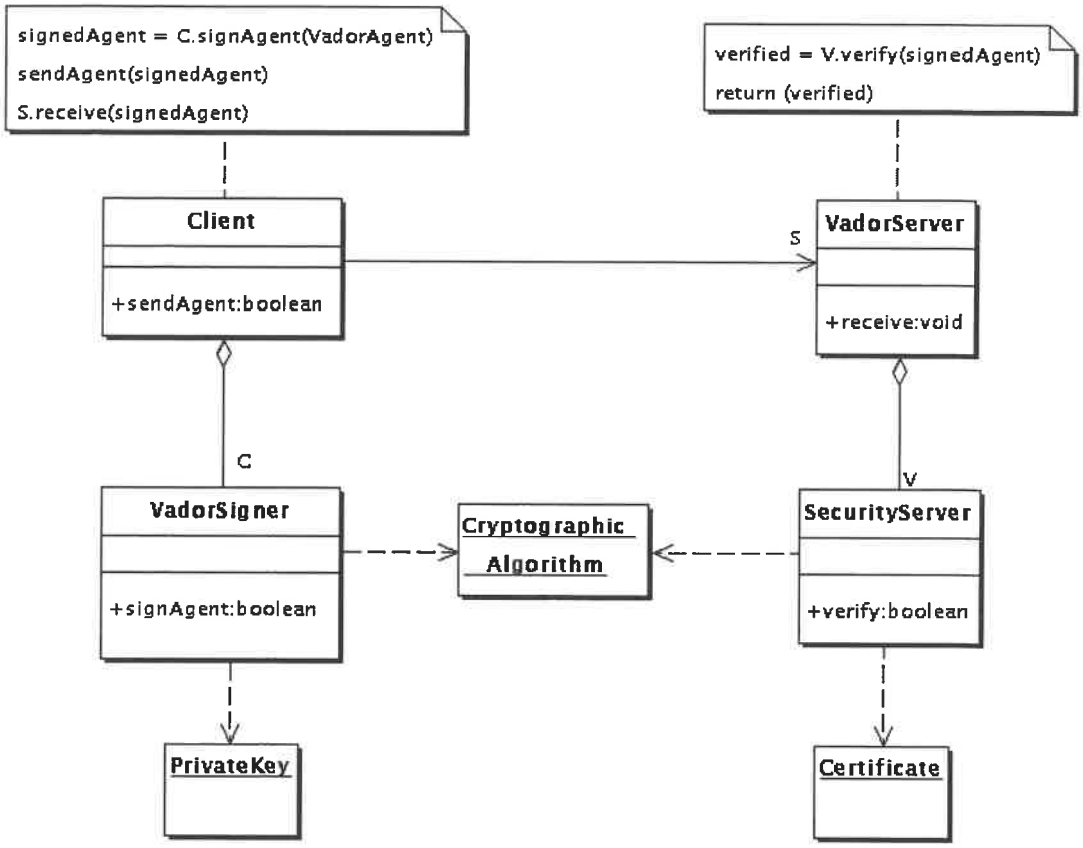


FIGURE 5.7 Agent Authentication Module Class Diagram

- Uses the PrivateKey generated by the VadorUser to sign the agent.

- **VadorServer**
  - Any Vador Server that may receive a signed Active Agent from the Client.
  - Before allowing the agent task execution on the server, it must authenticate the agent by verifying its signature.

- **SecurityServer**
  - On behalf of the VADOR System, applies the same Cryptographic Algorithm as the Client for the agent's verification and authentication.

- Uses a Certificate corresponding to the VadorUser's PublicKey to verify the agent's signature.

- **Cryptographic Algorithm**
  - Any Cryptographic Algorithm provided by cryptographic service providers for Java security applications.
  - Is the same for both agent signature and verification.

- **PrivateKey**
  - Generated for VadorUser, and stored in a keystore that belongs to the user.
  - Together with a PublicKey, forms a key pair.
  - Used for agent signature.

- **Certificate**
  - Is the PublicKey generated for VadorUser, and exported to the SecurityServer as a Certificate of the PublicKey.
  - Together with the PrivateKey, consists of a key pair.
  - Used for agent verification.

5.3.2.3 Interaction

The Agent Authentication Module interaction is illustrated in figure 5.8

- Clients create VadorAgent objects and apply VadorSigners to sign the VadorAgents.
- The VadorSigners apply Cryptographic Algorithms and use PrivateKeys to sign the agents.

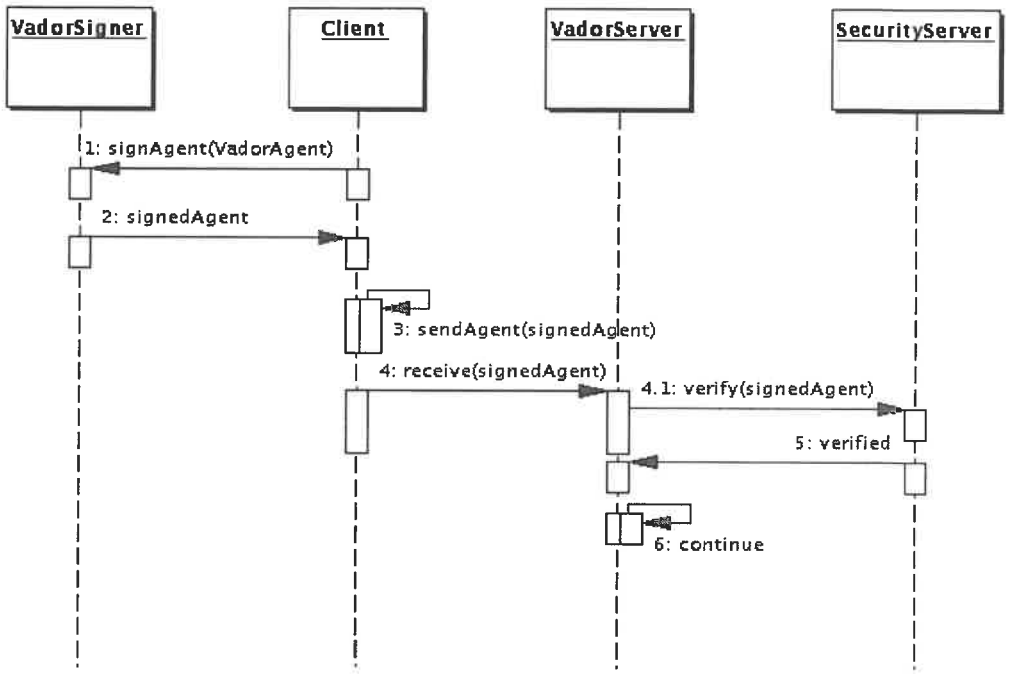


FIGURE 5.8 Agent Authentication Module Interaction Diagram

- Clients send the SignedAgents to remote VADOR Servers for task execution.
- VADOR Servers receive the SignedAgents from Clients, and then verify the agents' signatures by applying the SecurityServer's verification processes.
- According to security protocols, the SecurityServer applies the same Cryptographic Algorithms as the VadorSigner, and uses the Certificate import from the Clients to verify the agents' signatures.
- If the agents' signatures can be verified, the VADOR Servers will allow the agents to execute tasks on them. Otherwise, the agents will be discarded.

5.3.2.4 Related Patterns

- **Cryptographic Meta pattern** (Braga *et al.* (1998)): The VadorSigner represents the Codifier class, and the SecurityServer represents the Decodifier class in the pattern.
- **Sender Authentication pattern** (Braga *et al.* (1998)): It is the base of this module, because both can guarantee that information have a genuine and authentic sender, in such a way that the sender cannot repudiate the information that its receiver believes was sent by the sender.

5.3.3 Agent Signature Module

The *Agent Signature Module* is responsible of VadorAgent signature before an agent migrates to another execution place for task execution. The objective of agent signature is to authenticate the agent's sender at the execution places and protect the destination resources. This module allows the execution places to distinguish malicious agents from their original copies by verifying their signature, so that they can decline it, but accept the authenticated one to execute tasks on them.

5.3.3.1 Structure

The structure of the Agent Signature Module is illustrated in figure 5.9.

5.3.3.2 Participants

- **Clients**

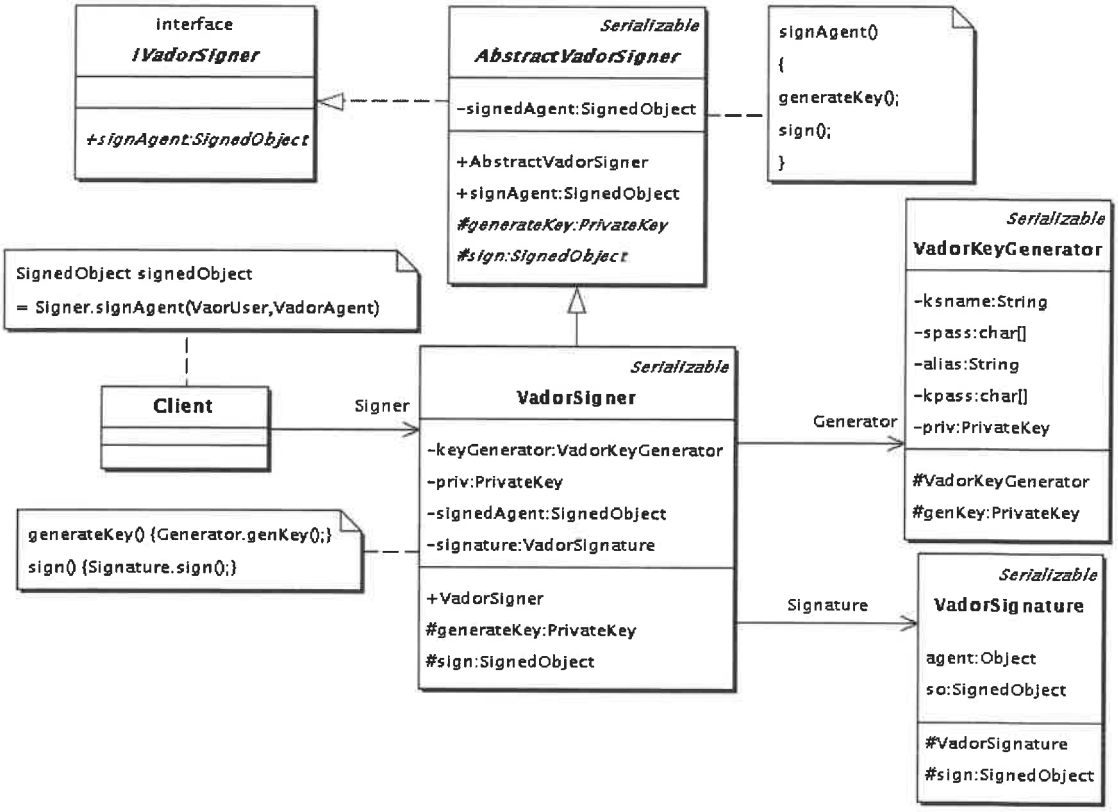


FIGURE 5.9 Agent Signature Module Class Diagram

- Any Vador Server that may apply its signers to sign the agents that have been created.
- VadorUsers (agents' owners) and agents provide information to the Vador-Signers.

- **IVadorSigner**
  - Is an interface to the VadorSigner class.
  - Provides an abstract *signAgent()* method that needs to be implemented.
- **AbstractVadorSigner**
  - Is an abstract class that implements the interface *IVadorSigner*, and its *signAgent()* method.

- Provides two abstract methods: the *generateKey()* method for key generation, and the *sign()* method for signature process.

- **VadorSigner**
  - Extends the *AbstractVadorSigner* class.
  - Provides signature services to the Clients.
  - Implements the abstract *generateKey()* method by applying the *VadorKeyGenerator* class.
  - Implements the abstract *sign()* method by applying the *VadorSignature* class.

- **VadorKeyGenerator**
  - Provides services to the *VadorSigner.*
  - Generates a PrivateKey for agent signature according to users' information provided by the *VadorSigner.*

- **VadorSignature**
  - Provides services to the *VadorSigner.*
  - Signs the agent according to the requests from *VadorSigner* using the PrivateKey provided by the *VadorSigner.*

5.3.3.3 Interaction

The Agent Signature Module Sequence Diagram illustrated in figure 5.10 presents the interaction between the participants in this module.

- A Client invokes the *signAgent()* method in the *AbstractVadorSigner* to obtain a SignedObject out of a VadorAgent object. It provides it with the information about the VadorAgent owner (VadorUser), and the agent object that needs to be signed.

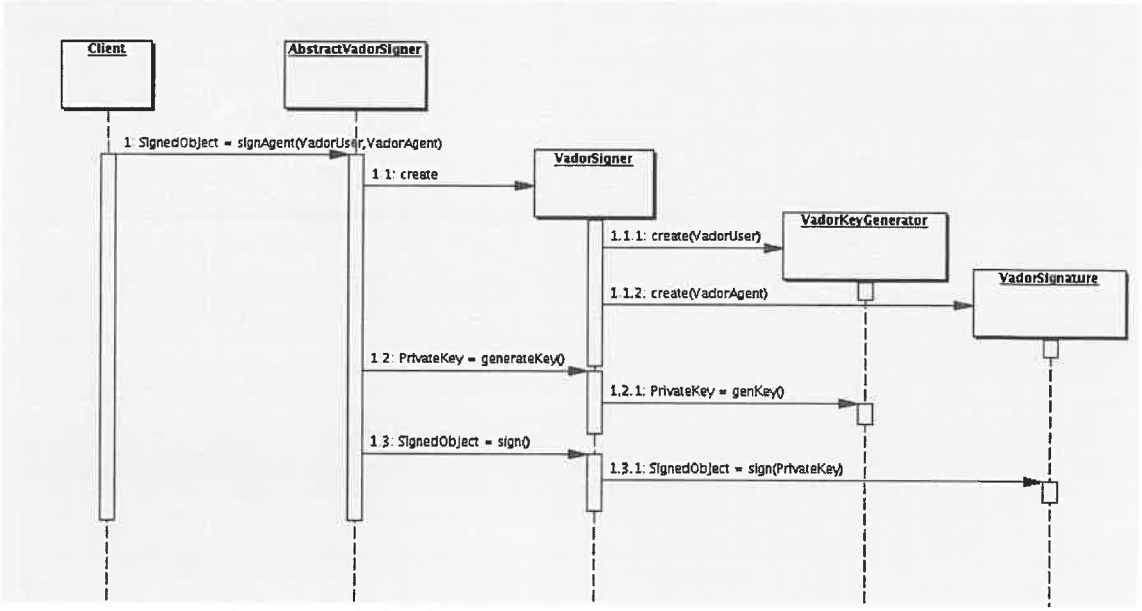


FIGURE 5.10 Agent Signature Module Interaction Diagram

- The AbstractVadorSigner creates a concrete *VadorSigner*, and requires it to perform the *generateKey()* method to obtain a PrivateKey from the agent owner's keystore. Then it applies the *sign()* method to obtain a SignedObject using the given PrivateKey.

- A concrete VadorSigner creates a *VadorKeyGenerator* object and invokes its *genKey()* method to get a PrivateKey. It also creates a *VadorSignature* object, and then invokes its *sign()* method to get the SignedObject. Figure 5.11 illustrates the agent signature algorithms in the VadorSignature class.

5.3.3.4 Related Patterns

- **Cryptographic Meta pattern**: This module represents the Codifier class in the *Cryptographic Meta* pattern, it is responsible for the signature process.

- **Sender Authentication pattern**: In *Sender Authentication* pattern, the Signer class has the same utilities as this module.

```
package Vador.NetworkTools.IVadorSecurityManager.AgentSignature;
import Vador.NetworkTools.*;
import java.security.*;
import java.io.*;

public class VadorSignature implements Serializable
{
    Object agent = null;

    protected VadorSignature (Object agent)
    {
        this.agent = agent;
    }
    protected SignedObject sign(PrivateKey priv)
    {
        try
        {
            Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
            Serializable s_agent = (Serializable)(agent);
            SignedObject so = new SignedObject(s_agent, priv, dsa);
        }
        catch (Exception e)
        {
           System.err.println(e.toString());
        }
        return so;
    }
}
```

FIGURE 5.11 Agent Signature Algorithms

- **Template Method pattern** (Gamma *et al.* (1994)): The abstract *AbstractVador-Signer* class defines the skeleton of agent signature algorithms in the *signAgent()* method, and lets its subclass-VadorSigner redefine certain steps of the algorithms without changing the structure.

5.3.4 Security Server Module

The *Security Server Module* defines a new Execution Place, that is a Vador Server responsible of VadorAgent signature verification, the agent's authentication, and information collection. It is a single access point to the other VADOR Servers and cannot be bypassed, it has its own *Security Manager* to enforce the External Security Policy for the verification and authentication processes.

5.3.4.1 Structure

The structure of the Security Server Module is illustrated in figure 5.12.

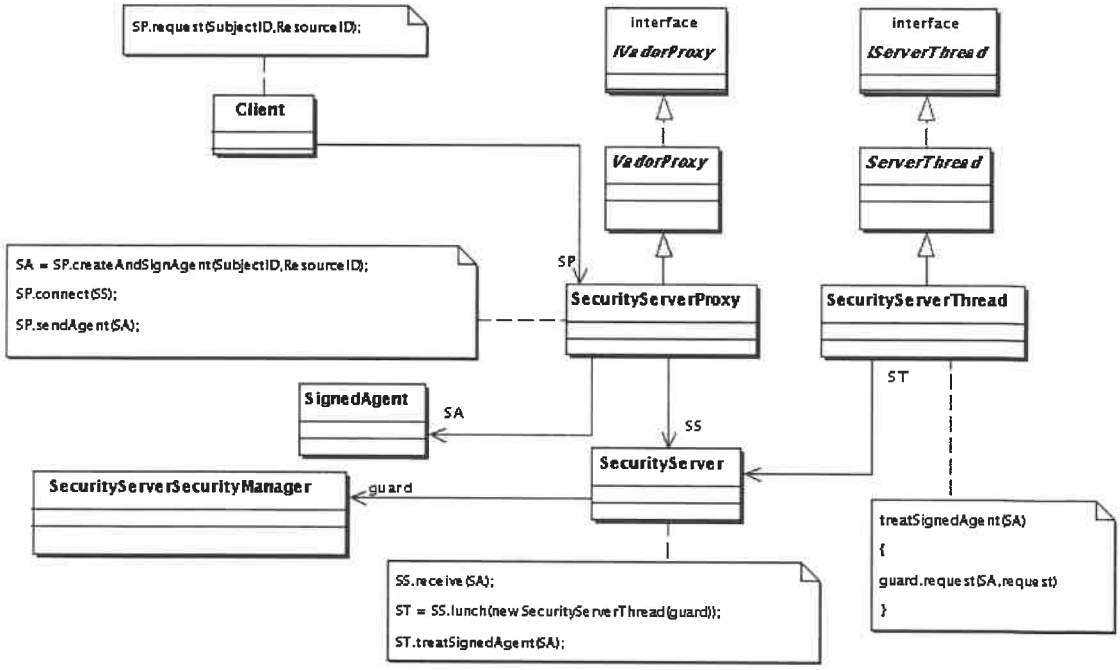


FIGURE 5.12 Security Server Module Class Diagram

5.3.4.2 Participants

- **Clients**
  - Any Vador Server that may send requests to the SecurityServer for verifying signature of its received agents.
  - Provides SubjectId and ResourceID obtained from the agents to the Security-Server via a *SecurityServerProxy* object.
- **SecurityServerProxy**
  - Extends the abstract *VadorProxy* class defined in the VADOR System.
  - Provides a local representative for the SecurityServer.
- **SignedAgent**
  - An object created by the *SecurityServerProxy*.
  - Is responsible of signing a *VadorAgent* object using security attributes provided by the SecurityServerProxy.
- **SecurityServer**
  - A Vador Server that provides agent signature verification service to its clients.
  - Applies its Security Manager for the authentication and verification processes.
- **SecurityServerThread**
  - Extends the Thread class defined in java.net package.
  - Launched by the *SecurityServer* for communicating with a specified client.
  - Applies the *SecurityServerSecurityManager* assigned by the *SecurityServer* to verify a *SignedAgent*.
- **SecurityServerSecurityManager**

- Extends the *AbstractVadorSecurityManager* class defined in the *SecurityManager* module (See section 5.3.6 for details).
- It is an instance of the *SecurityManager* module.
- Specifically deals with VadorAgent signature verification(VadorAgent authentication).

5.3.4.3 Interaction

Interaction between the Security Server Module participants is illustrated in figure 5.13.

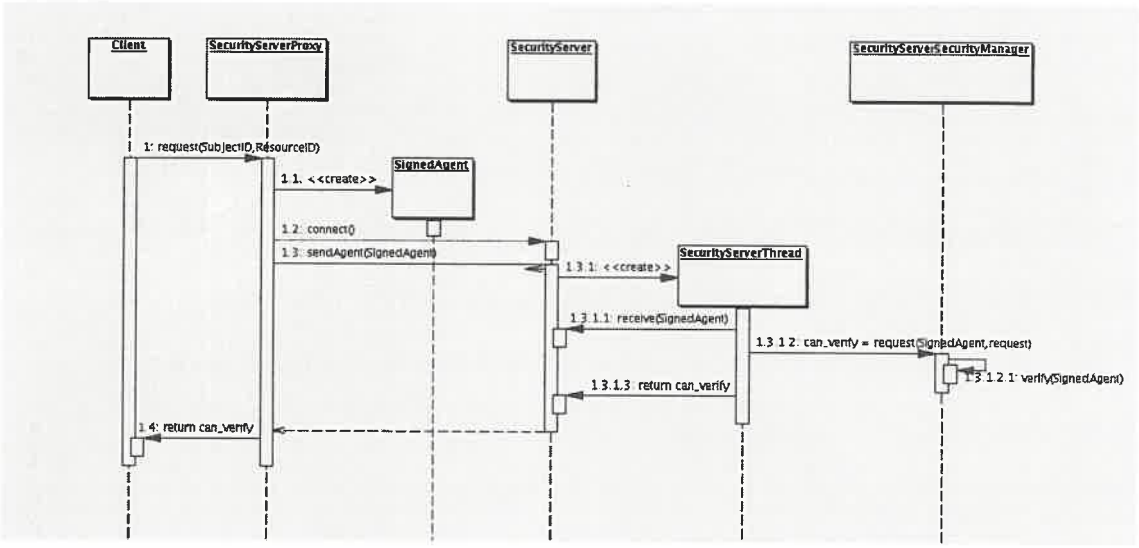


FIGURE 5.13 Security Server Module Interaction Diagram

- A Client sends request with a SubjectId and ResourceID to the *SecurityServer* via its *SecurityServerProxy*.
- The *SecurityServerProxy* creates and signs a VadorAgent object with information provided by the Client.
- The *SecurityServerProxy* then connects to the Security Server, sends the SignedAgent object via the channel, and waits for a return message.

- The *SecurityServer* launches a *SecurityServerThread* to deal with the request.
- The *SecurityServerThread* receives the *SignedAgent* object from its proxy, then applies its *SecurityManager* to verify the object's signature.
- The *SecurityServer*'s *SecurityManager* verifies the *SignedAgent*'s signature, then returns the result to the *SecurityServerThread.*
- The *SecurityServerThread* returns the result to the *SecurityServer.*
- The result finally returns to the Client by the *SecurityServer* via its proxy.

5.3.4.4 Related Patterns

- **Proxy pattern**(Gamma *et al.* (1994)) is applied to the *Security Server* module. A client's request can only be sent to the *Security Server* via the *Security Server Proxy.*
- **Single Access Point pattern** (Yoder and Barcalow (1998)): the *Security Server* module is a specialization of the Single Access Point pattern. It provides a security module to control access to the other VADOR Servers, and cannot by by passed.
- **Policy**(Guibault *et al.* (2004)) is used by the *Security Server Security Manager* to enforce security policy for the signature verification processes.

5.3.5 Security Attributes Descriptor Module

The *Security Attributes Descriptor Module* provides access to security-relevant attributes of an entity on whose behalf operations are to be performed. Types of the security attributes include *User Attributes*, *Subject Attributes*, and *Object or Information Attributes.*

This module allows an operation to specify a subset of the attributes for which it requires access, by specifying a *SecurityAttributeType*. This way the *VadorSecuritySubjectDescriptor* will not be affected when a new attribute is added to a filtered *SecurityAttributeList*.

5.3.5.1 Structure

Figure 5.14 represents structure of the *Security Attributes Descriptor Module*.

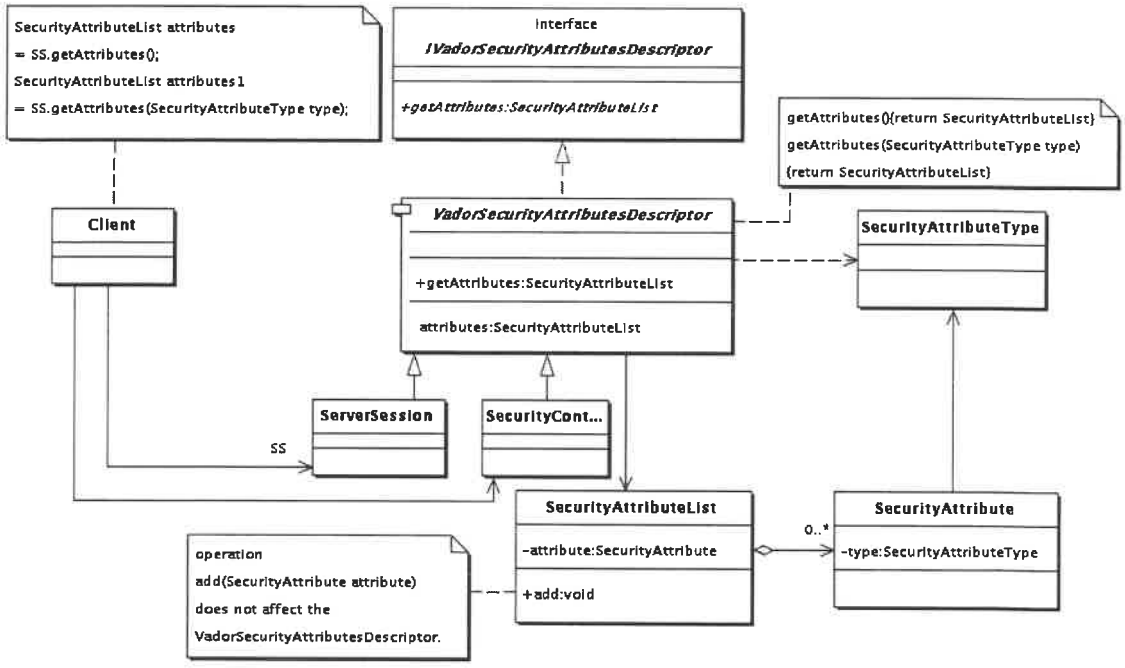


FIGURE 5.14 Security Attributes Descriptor Module Class Diagram

5.3.5.2 Participants

- **Client**
  - Any entity in the VADOR System that needs to access security-relevant attributes to perform an operation.

- Accesses to the security-relevant attributes via the *Security Attributes Descriptor Module.*

- **IVadorSecurityAttributesDescriptor**
  - Provides an interface to the *Security Attributes Descriptor Module.*

- **VadorSecurityAttributesDescriptor**
  - Implements the functions defined by the *IVadorSecurityAttributesDescriptor.*
  - Defines abstract methods that will be implemented by concrete *VadorSecurityAttributesDescriptors*.

- **SecurityAttributeList**
  - Contains sets of *SecurityAttributes*.
  - New SecurityAttributes can be added into the *SecurityAttributeList* by applying its add() function.

- **SecurityAttribute**
  - Defines SecurityAttributes and distinguishes them by their *SecurityAttributType.*

- **SecurityAttributeType**
  - Deifnes types of SecurityAttributes. For example, KEYINFO, OBJECT, and SUBJECT.

- **ServerSession**
  - A concrete class that extends the abstract *VadorSecurityAttributesDescriptor.*
  - Created by a *VadorSecurityManager* on behalf of a *ServerThread* to keep track of the security-relevant attributes associated with that ServerThread, so

that the attributes can be used by the operations on behalf of the same ServerThread.

- If the ServerThread is closed, the ServerSession is also closed.

- **SecurityContext**
  - a concrete class that extends the abstract *VadorSecurityAttributesDescriptor.*
  - created by a *VadorSecurityManager.* to describe security-relevant attributes provided by a *VadorAgent*, then it will be passed to a *SecurityVisitor* for permission checking.
  - if the permission is granted by the *PolicyApplicator*, the *SecurityContext* will be added to the previous *SeverSession* for later usage.

5.3.5.3 Related Patterns

- This module is an instance of **Subject Descriptor pattern** (OpenGroup (2002)). It provides access to security-relevant attributes on an entity, adds new attributes to the list without affecting the attributes descriptor, and allows operations to specify a subset of the attributes by specifying an attribute type.
- The ServerSession implements the **Session pattern** (Yoder and Barcalow (1998)). A ServerSession object can hold all of the security-relevant attributes that need to be shared by many objects in the same ServerThread.

5.3.6 Security Manager Module

The *Security Manager Module* performs two level access control mechanisms to the VADOR system: Active Agent *Authentication* and *Authorization.* This module is the

core of the VADOR security architecture, it in collaborates with all the security modules to control access to the VADOR system resources and protect the data and information.

5.3.6.1 Structure

Figure 5.15 represents the structure of the *Security Manager Module*. *OtherServerSecurityManager*, *OtherServerSecurityVisitor*, and *OtherServerPolicyApplicator* represent the *SecurityManager*, *SecurityVisitor*, and *PolicyApplicator* that stand on behalf of the VADOR Servers other than the *SecurityServer*.

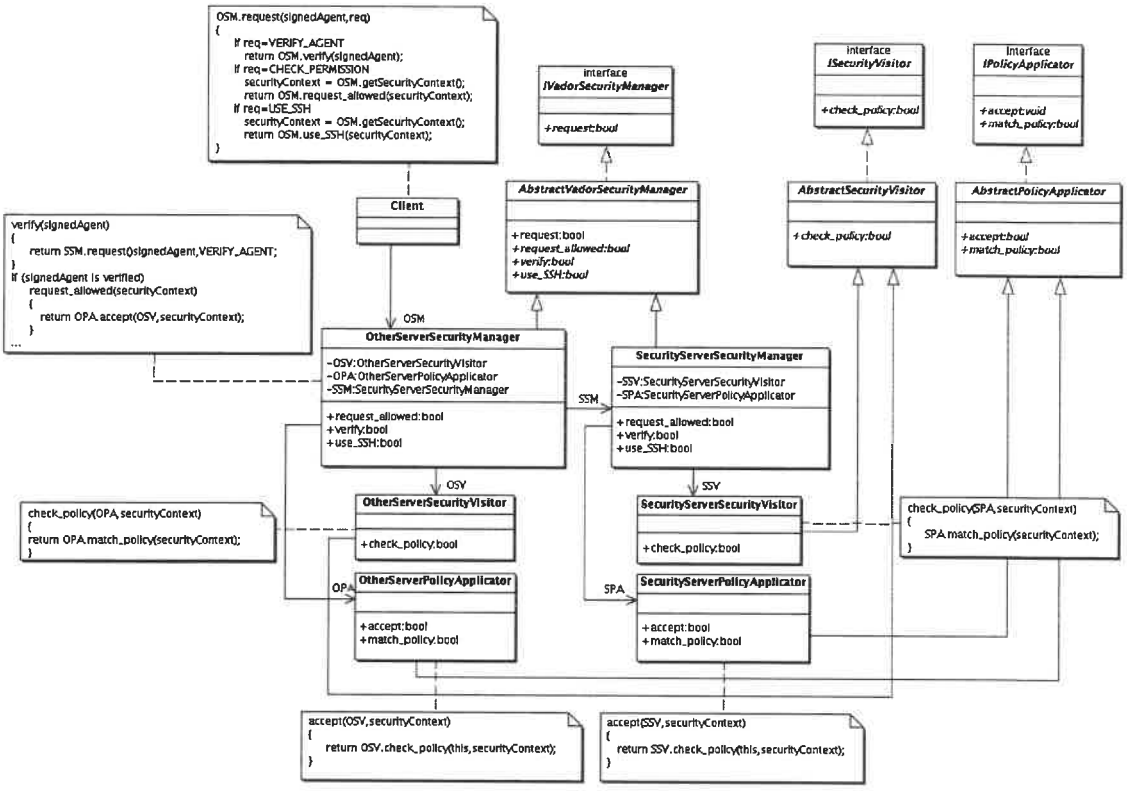


FIGURE 5.15 Security Manager Module Class Diagram

5.3.6.2 Participants

- **Client**

- Any entity in the VADOR System that needs to access to VADOR Servers to complete a task.
- In order to access VADOR Servers' resources, a client must be verified and obtain permission via the *Security Manager Module*.

- **IVadorSecurityManager**
  - Is an interface of the *Security Manager* to all the VADOR Servers.
  - Provides an abstract *request()* method that needs to be implemented.

- **AbstractVadorSecurityManager**
  - Is an abstract class that implements the interface *IVadorSecurityManager* and its *request()* method.
  - Provides three abstract methods to its subclasses: method *verify()* authenticates and verifies a signed agent object, method *request_allowed()* checks permissions on a verified agent object, and method *use_SSH()* performs SSH protection mechanisms if they are required by a Vador Server's security policy.

- **SecurityServerSecurityManager**
  - Extends the *AbstractVadorSecurityManager* class.
  - Provides a *SignedAgent* authentication and verification services to the other VADOR Servers, for example, Executive, Librarian, and Wrapper servers.
  - Implements the abstract *verify()*, *request_allowed()*, and *use_SSH()* methods defined by its supper class.

- **OtherServerSecurityManager**
  - Extends the *AbstractVadorSecurityManager* class.

- Acts on behalf of a specified Vador Server, and provides permission checking services on a verified (by the *SecurityServerSecurityManager*) ActiveAgent for the Vador Server.
- Implements the abstract *verify()*, *request_allowed()*, and *use_SSH()* methods defined by its supper class.

- **ISecurityVisitor**
  - Is an interface of the *SecurityVisitors* to all the *Security Managers* that act on behalf of the VADOR Servers.
  - Provides an abstract *check_policy()* method that needs to be implemented.

- **AbstractSecurityVisitor**
  - Is an abstract class that implements the interface *ISecurityVisitor*.
  - Provides the abstract *check_policy()* method to its subclasses.

- **SecurityServerSecurityVisitor**
  - Extends the *AbstractSecurityVisitor* class and implements the abstract *check_policy()* method.
  - Acts on behalf of the *SecurityServerSecurityManager*, and provides authentication and verification services to the Security Manager.
  - Provides *SecurityContexts* to *SecurityServerPolicyApplicator* for the actual authentication and verification algorithms.

- **OtherServerSecurityVisitor**
  - Extends the *AbstractSecurityVisitor* class and implements the abstract *check_policy()* method.
  - Acts on behalf of a specified Vador Server's Security Manager, and provides permission checking services to the Security Manager.

- Provides *SecurityContexts* to a *PolicyApplicator* that acts on behalf of the same Security Manager for the actual permissions checking algorithms.

- **IPolicyApplicator**
  - Is an interface of the *PolicyApplicators* to all the *Security Managers* that act on behalf of the VADOR Servers.
  - Provides abstract *accept()* and *match_policy()* methods that need to be implemented.

- **AbstractPolicyApplicator**
  - Is an abstract class that implements the interface *IPolicyApplicator*.
  - Provides the abstract *accept()* and *match_policy()* methods to its subclasses.

- **SecurityServerPolicyApplicator**
  - Extends the *AbstractPolicyApplicator* class and implements the abstract *accept()* and *match_policy()* methods.
  - Acts on behalf of the *SecurityServerSecurityManager*, and collaborates with the *SecurityVisitor* to perform actual authentication and verification algorithms by consulting the *External Security Policy* which is initialized by the VADOR administrator.

- **OtherServerPolicyApplicator**
  - Extends the *AbstractPolicyApplicator* class and implements the abstract *accept()* and *match_policy()* methods.
  - Acts on behalf of a specified Vador Server, and collaborates with the *SecurityVisitor* to perform actual permission checking algorithms by consulting *VADOR Security Policy* of this server which is initialized by the administrator.

5.3.6.3 Interaction

Interaction between the Security Manager Module participants is illustrated in figure 5.16.

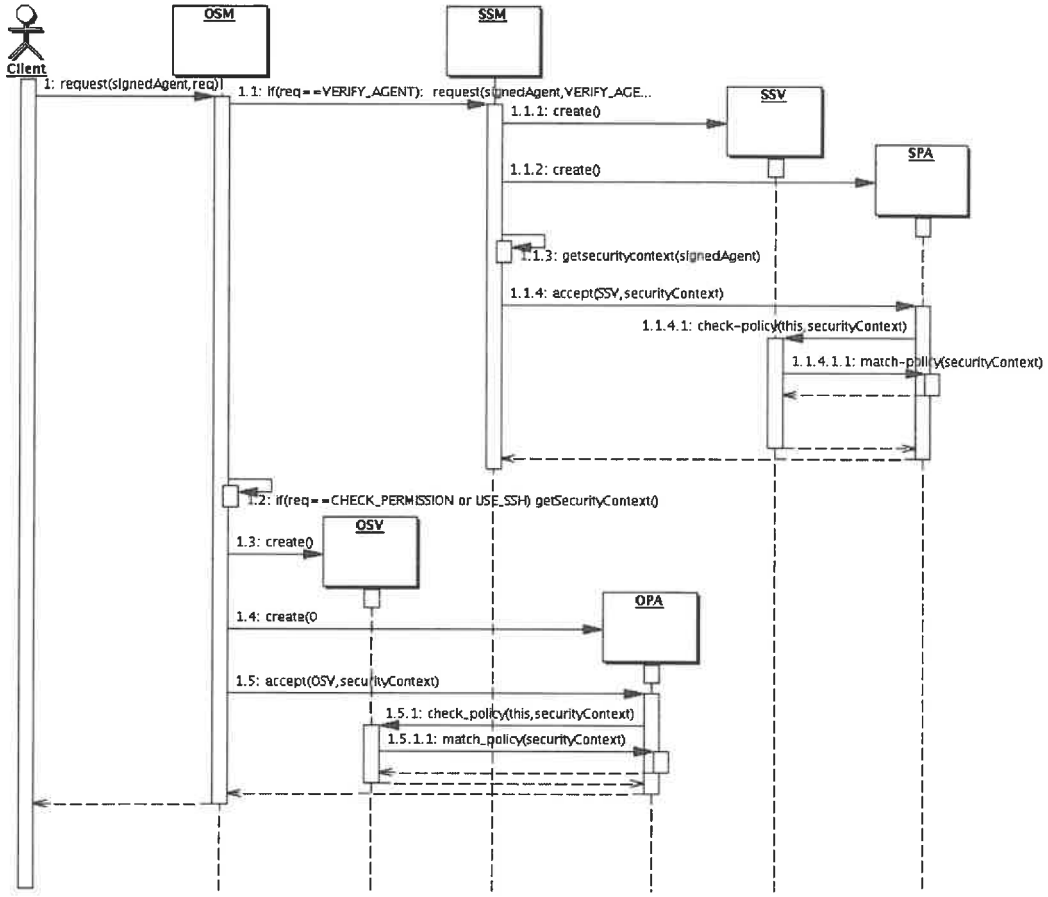


FIGURE 5.16 Security Manager Module Interaction Diagram

- A Client invokes the *request()* method of an *OtherServerSecurityManager* (OSM) that stands on behalf of a Vador Server, and passes its request with security contexts to this Security Manager. The request can be either *VERIFY AGENT*, *CHECKING PERMISSION*, or *USE SSH*, and the security contexts contain the security information related to this Client.

- If a Client passes the *VERIFY AGENT* request to an OSM, the OSM will become a Client to the *SecurityServerSecurityManager* (SSM), it then invokes the *request()* method of the SSM, forwards the *VERIFY AGENT* request and the security contexts to the SSM.

- When the SSM receives a *VERIFY AGENT* request from its client, it first creates two objects on the Security Server: a *SecurityServerSecurityVisitor* (SSV) and a *SecurityServerPolicyApplicator* (SPA). Then, it gets *Security Contexts* from the *SignedAgent* passed by the client, invokes the SPA's *accept()* method, passes the SSV object and the security contexts as parameters. The SPA's *accept()* method accepts the SSV as its visitor and allows it to visit its policy (External Security Policy) using its *match_policy()* method.

- If a Client passes a *CHECKING PERMISSION* request to an OSM, the OSM will first check if the security contexts contain a *verified* state, if it is true, it then invokes its own *request_allowed()* method, forwards the security contexts to it to continue checking processes, otherwise, it passes a *VERIFY AGENT* request to the SSM as described above.

- If a Client passes a *USE SSH* request to an OSM, the OSM will first check if the security contexts contain a *verified* state, if it is true, it then invokes its own *use_SSH()* method, forwards the security contexts to it to continue using SSH processes, otherwise, it passes a *VERIFY AGENT* request to the SSM as described above.

- In both cases of an OSM performing a *request_allowed()* or a *use_SSH()* methods, at first, the OSM needs to create two objects on the same server: an *OtherServerSecurityVisitor* object (OSV) and an *OtherServerPolicyApplicator* (OPA). Then, it gets *Security Contexts* that were obtained by the SSM and saved in a *Securtiy Session*, invokes the OPA's *accept()* method, passes the OSV object and the security

contexts as parameters. The OPA's *accept()* method accepts the OSV as its visitor and allows it to visit its policy (VADOR Security Policy) using its *match_policy()* method.

- The OSV returns the results to the OSM, the OSM then forwards the result to its server.

5.3.6.4 Related Patterns

- The *SecurityServerSecurityManager* and the *OtherServerSecurityManager* represent the *Guard*, the *SecurityVisitors* represent the *Policy*, and the *PolicyApplicators* represent the *Rule* in the **Policy Pattern** (Guibault *et al.* (2004)).
- The *SecurityVisitors* cooperates with the *PolicyApplicators* and implement the **Visitor Pattern** (Gamma *et al.* (1994)), so that the VADOR system can add a new security related operation to a *SecurityVisitor* without changing operations on the *PolicyApplicators*.
- The *PolicyApplicators* use the **Strategy Pattern** (Gamma *et al.* (1994)) to define permission checking algorithms.

5.3.7 Consequences

- Introducing a separate Security Server from the other VADOR Servers helps to avoid the Security Server monopolies the access control processes. This approach enhances management flexibility of the VADOR system. In addition, it divides responsibilities among different servers, which can improve time deduction of access control processes.
- The Security Managers act on behalf of the servers can help dynamically define

and load security policy for every server that is involved in the VADOR system, so that one Vador server's security policy change will not affect the others and prevent them from working properly.

- Although the Security manager of the Security Server performs verification and authentication processes to control accesses to all the other VADOR Servers, it is not the only access point into the VADOR system. The other VADOR Servers' Security Managers also represent access points to the system. The difference is that the first one acts on behalf of the Security Server to control accesses to the other VADOR Servers, while the others control accesses to VADOR Resources.

5.3.8 Related Patterns

- The *Protected System pattern* (Guibault *et al.* (2004)) describes the structure of the Security Manager, in which access to a set of resources by various clients can be controlled.

- The *Policy pattern* isolates policy enforcement activities into one dedicated component - SecurityVisitor to ensure that these activities are done correctly and in proper sequence.

- The *Partitioned Application pattern* constructs Security Managers on behalf of VADOR Servers. It allows Security Managers to perform simple tasks rather than having only one complex Security Manager for the whole system. Having these Security Managers facilitates the security management, especially in VADOR, because some tasks require different privilege levels and security permissions.

- The *Subject Descriptor pattern* provides a convenient abstraction for managing security relevant attributes. Since the Security Server's Security Manager is used for identifying the subject, and a Security Manager of an Execution Place is later used

to authorize (or deny) the subject access to resources, this abstraction is particularly useful in VADOR.

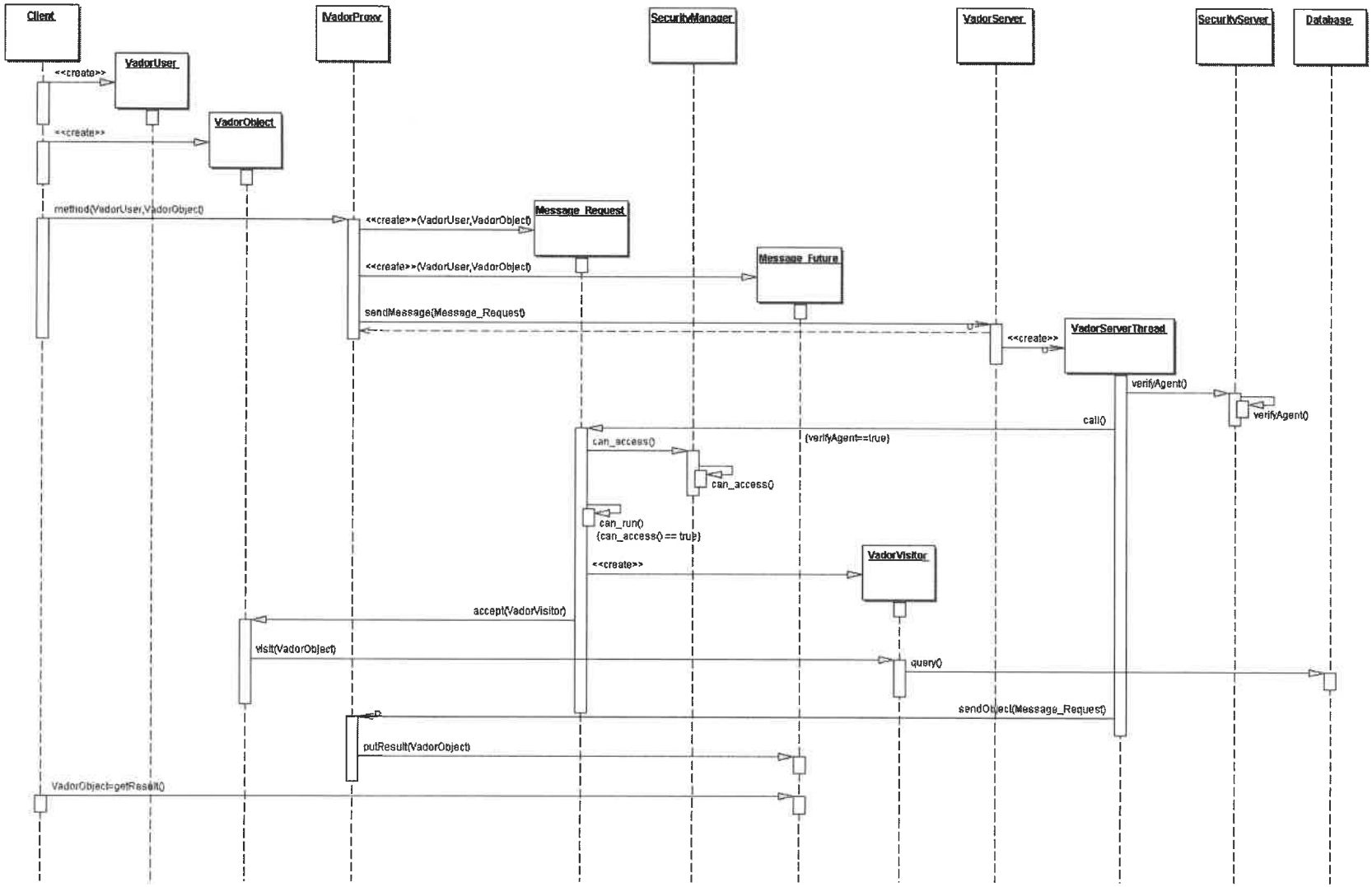
- Many other patterns are also involved in the *Security Manager pattern* , and they have been described in the modules (see section 5.3).

5.4 Summary

This chapter introduced the *Security Manager pattern*, that represents the security architecture of the VADOR system, formalizes the prevention approaches to the VADOR security defect introduced in section 3.2, and then provides solutions to the VADOR security problems (See Chapter 3). The main objective of using this pattern is to control access to VADOR data files, and it can be realized by separating authentication and verification processes on a centralized Security Server from permission checking processes on another individual Vador Server, such as Executive Server, Librarian Server, and Wrapper Server.

The *Security Manager pattern* consists of six modules (see section 5.3 for detailed description): the *Security Interface Module* provides interfaces to all the security related modules in the VADOR system, the *Agent Authentication Module* is responsible of the first level protection of the VADOR system - Agent Authentication and Verification, the *Agent Signature Module* is responsible of VadorAgent signature before an agent migrates to another execution place for task execution, the *Security Server Module* defines a Vador Server responsible of VadorAgent signature verification, the agent's authentication, and information collection, the *Security Attributes Descriptor Module* provides access to security-relevant attributes of an entity on whose behalf operations are to be performed, and the *Security Manager Module* coorperates with all the security modules control access to the VADOR system resources and protect the data and information.

This chapter has described the *Security Manager pattern* by introduction of its contexts, problems, solutions, structure, interactions, and so on, the chapter 6 will focus on validation and testing of this pattern in the VADOR system.


Client
VadorUser
VadorProxy
SecurityManager
VadorServer
SecurityServer
Database
<<create>>
VadorObject
<<create>>
method(VadorUser,VadorObject)
<<create>>(VadorUser,VadorObject)
Message_Request
<<create>>(VadorUser,VadorObject)
Message_Future
sendMessage(Message_Request)
<<create>>
VadorServerThread
verifyAgent()
verifyAgent()
call()
{verifyAgent==true}
can_access()
can_access()
can_run()
{can_access() == true}
<<create>>
VadorVisitor
accept(VadorVisitor)
visit(VadorObject)
query()
sendObject(Message_Request)
putResult(VadorObject)
VadorObject=getResult()

CHAPTER 6

VALIDATION AND TESTS

Previous chapters introduced the VADOR framework, the security problems and defects that exist in the original VADOR architectural design, and the possible solutions to the problems introduced. Then a security design pattern, named *Security Manager pattern* has been defined to prevent the security defects and resolve the security problems from design, so that it may be applied to build a security architecture for VADOR, specifically, to control accesses to its data files.

This chapter will focus on validation and tests of the *Security Manager pattern* in VADOR.

6.1 Objectives

Based on the objectives of this work as introduced in section 1.4, the objective of *Validation* and *Tests* is to validate and test the *Security Manager pattern* (See Chapter 5) that implements the *Protected System pattern* (Guibault *et al.* (2004)) and other security design patterns introduced in Chapter 2, and in which the protection mechanisms should control all operations made available by the agent components through each agent proxy. The validation and tests processes will insure that the Security Manager pattern is able to control accesses to VADOR system's resources, specifically, data files, and increase the applications usability and reliability, and reduce long-term costs.

6.1.1 Why Validate the Pattern

The *Security Manager pattern* consists of six modules (See section 5.3 for detailed description), each of them participates as a specific protection mechanism of the VADOR security architecture and is responsible for it. Many security design patterns also have been implemented in these modules to solve specific security problems. However, since the Security Manager pattern has been defined using a static technique (paper review), it may be more effective in finding, correcting and preventing problems at an earlier stage of a development process, but its functionalities and run-time behavior need to be demonstrated and analyzed using dynamic technique, such as testing.

6.1.2 Why Test the Pattern in the VADOR Framework

In order to validate the *Security Manager pattern*, some tests need to be made, specifically, these tests should focus on controlling accesses to VADOR system's resources. The reason of using the VADOR framework to test this design pattern is that VADOR is an instance of the type of distributed application towards which the *Security Manager pattern* is targeted. VADOR's system resources need to be protected from attacks of malicious users or agents.

6.2 Management of the Validation and Tests

According to the *Principles of Software Validation* introduced by *Soft Solutions International* (Soft-Solutions-International (2002)), proper validation of software includes the planning, execution, analysis, and documentation of appropriate validation activities and tasks (including testing or other verification). Security Manager pattern validation is also based on the software life cycle, but instead of an entire software, it focuses on the specific

Security issues.

The detailed validation will be described in the following subsections.

6.2.1 Plan

Based on a study of the security defects prevention requirements and related control access problems in VADOR, this section introduces the validation plan that specifies how the process will be controlled and executed.

As table 6.1 describes, the validation is planned in three steps, each step will validate one of the functionalities required to control access to VADOR system's resources using the Security Manager pattern.

TABLE 6.1 Security Manager Pattern Validation Plan

Steps	Plans	Required Functionalities
1	Validate and test *Security Policy* initialization	*External Security Policy* defines privilege of an agent to access VADOR servers and is initialized by the VADOR administrator. The *VADOR Security Policy* defines privileges of a verified agent to access VADOR resources and is initialized by the *PolicyApplicator* based on user information defined in database.
2	Validate and test verification and authentication processes	A client signs an agent before it is sent. A VADOR server verifies an agent after receiving it by consulting the *External Security Policy*
3	Validate and test authorization processes	A VADOR server authorizes an agent to access VADOR resources on its behalf by consulting the *VADOR Security Policy*

6.2.2 Procedures and Expected Results

This section will specify procedures that were established for the validation tasks and expected results from the validation procedures.

- **Procedures**

  1. Demonstrate initialization of the *External Security Policy* using the *Keytool* program provided by the Java Security package. The initialization of the *VADOR Security Policy* will be demonstrated through the validation of authorization processes.
  2. Test verification and authentication of agents, including test of the *Agent Signature* using a *Private Key* to sign an agent by its creator and sender, and test of the *Agent Authentication* using the *Certificate* corresponding to the *Public Key* by the VADOR *Security Server* if the agent receiver (agent execution place) requires it.
  3. To test agent authorization processes, the *VADOR Security Policy* must have been defined, and related VADOR user information, such as group, privilege and permissions to access resources should have been saved in the VadorPolicy database table. In addition, the agent should have been verified by the *Security Server* via the verification processes. Then the testing procedure focuses on the *Security Manager*, that acts on behalf of the VADOR server where the agent has moved on for task execution. The functionalities of the Security Manager will be tested including access control management, such as sending an agent to the *Security Server* for verification processes, and creating a *SecurityVisitor* object to visit the *VADOR Security Policy* that is initialized by *PolicyApplicator* object, which is also created by the Security Manager.

- **Expected Results**

- A VADOR administrator will be able to generate key pairs and save them into Keystores on VADOR servers for all VADOR users that will be permitted to create, sign, and send agents on these servers based on a defined *External Security Policy*. Then the key certificates will be exported from these servers and imported onto the VADOR Security Server by the administrator.
- Private Keys on VADOR Servers can be selected by VADOR users to sign agents, and Key Certificates on the Security Server can be selected to verify the signed agents. But VADOR users will not be able to select Private Keys or Certificates using incorrect Keystore, key, or certificate information, such as alias or password.
- If an agent was signed successfully, it will be sent to another VADOR server. Otherwise, it cannot be sent, and a *sendAgent_failed* message will be returned to its sender.
- If the Security Server cannot find a certificate to verify a signed agent using key information provided by the agent, or if the Security Server cannot decodify the signed agent using a fine certificate, the agent will not be authenticated. An *authentication_failed* message will then be returned to the VADOR server that required the verification. Otherwise, the agent is authenticated, and an *authentication_succeed* message will be returned.
- If a VADOR server receives an *authentication_failed* message from the Security Server, it discards the received signed agent, and forwards the *authentication_ failed* to the agent sender. If it receives *authentication_succeed* message, it will continue its agent authorization processes.
- If the VADOR server's PolicyApplicator cannot initialize VADOR Security Policy from the VadorPolicy table in the VADOR database using the VADOR user's information that comes with the agent, or it cannot match the agent's requests to its sender's privilege, it will return FALSE to the VADOR server,

so that the agent will not be authorized to execute a task on the VADOR server. The VADOR server then will discard the agent, and return an *authorization_failed* message to the agent sender.

- If the agent can be authorized, the PolicyApplicator returns TRUE to the VADOR server, so that the agent will be permitted to execute tasks on that server.

6.2.3 Test Cases and Results

This section will demonstrate test cases that correspond to the validation plan and are conducted in accordance with the established procedures. Then, results will be gathered from the test cases and evaluated by comparing them with the expected results introduced in section 6.2.2.

6.2.3.1 Test Cases

- Login as VADOR administrator, use the Java *Keytool* program to generate several security key pairs using different aliases and passwords, and save them in one or different *Keystore(s)* on VADOR servers. Test cases and results of the key pairs generation are described in table 6.2.
- Use the *Keytool* to export certificates from the VADOR servers to the VADOR *Security Server*, and then import the certificates onto the *Security Server* for the signed agent verification and authentication purpose. Table 6.3 shows the test cases and results of the certificates exporting and importing onto the Security Server.
- Login as different VADOR users, try to use both correct and incorrect key information to sign agents on the VADOR Servers. The test cases and results are showed in table 6.4.

TABLE 6.2 Keys Generation Test Cases and Results

Server	Keystore	Keystore Pass	Key Alias	Key Pass
Executive	s1	s1-pass	s1-k1 s1-k2 s1-k3	s1-k1-pass s1-k2-pass s1-k3-pass
Librarian	s2	s2-pass	s2-k1 s2-k2 s2-k3	s2-k1-pass s2-k2-pass s2-k3-pass

TABLE 6.3 Certificates Exporting, Importing Test Cases and Results

From Server	Keystore	Keystore Pass	Key Alias	Key Pass	Certificate
Executive	s1	s1-pass	s1-k1 s1-k2 s1-k3	s1-k1-pass s1-k2-pass s1-k3-pass	s1-k1-cert s1-k2-cert s1-k3-cert
Librarian	s2	s2-pass	s2-k1 s2-k2 s2-k3	s2-k1-pass s2-k2-pass s2-k3-pass	s2-k1-cert s2-k2-cert s2-k3-cert

TABLE 6.4 Agent Signature Test Cases and Results

Server	Keystore	Keystore Pass	Key Alias	Key Pass	Result
Executive	s1	s2-pass	s1-k1	s1-k1-pass	sendAgent failed
Executive	s1	s1-pass	s1-k2	s1-k2-pass	agent sent
Executive	s1	s1-pass	s1-k3	s1-k2-pass	sendAgent failed
Librarian	s2	s2-pass	s2-k2	s2-k1-pass	sendAgent failed
Librarian	s1	s1-pass	s1-k2	s1-k2-pass	sendAgent failed
Librarian	s2	s2-pass	s2-k3	s2-k3-pass	agent sent

- Login as different VADOR users, use both correct and incorrect certificates to verify a signed agent on the Security Server, and sometimes change the signed agent status so that it may become a malicious agent. Table 6.5 shows the Agent Authentication test cases and results.

TABLE 6.5 Agent Authentication Test Cases and Results

Peivate Key	Certificate	Agent Status	Result
s1-k1	s1-k1-cert	no change	authentication succeed
s1-k2	s2-k2-cert	no change	authentication failed
s1-k3	s1-k2-cert	changed	authentication failed
s2-k2	s2-k3-cert	no change	authentication failed
s1-k2	s1-k2-cert	changed	authentication failed
s2-k3	s2-k3-cert	no change	authentication succeed

- Login as different users, use both user names that do and do not exist in the Vador-Policy database table to test agent senders privileges and the agent authorization. If an user name exists, try to match operation requests from the agent to the user's privileges in the database table. Table 6.6 illustrates the VadorPolicy database table and values for the test cases. The test cases and results of the Agent Authorization are shown in table 6.7.

TABLE 6.6 VADOR Policy Database Table and Value Example

UserID	Username	group	Read	Write	Execute
1	u1	group1	true	true	true
2	u2	group1	true	false	true
3	u3	group1	true	true	true
4	u4	group2	true	false	true
5	u5	group2	true	false	false
6	u6	group2	true	false	true

6.2.3.2 Results of Validation and Tests

Section 6.2.3.1 demonstrated some test cases that followed the validation plan (See section 6.2.1 for details) and were conducted in accordance with the established procedures introduced in section 6.2.2.

TABLE 6.7 Agent Authorization Test Cases and Results

User Name	Agent Request	Return	Result
u1	Read	true	agent executes tasks
u1	Write	true	agent executes tasks
u1	Execute	true	agent executes tasks
u2	Read	true	agent executes tasks
u2	Write	false	agent is discarded
u2	Execute	true	agent executes tasks
u3	Read	true	agent executes tasks
u3	Write	true	agent executes tasks
u3	Execute	true	agent executes tasks
u4	Read	true	agent executes tasks
u4	Write	false	agent is discarded
u4	Execute	true	agent executes tasks
u5	Read	true	agent executes tasks
u5	Write	false	agent is discarded
u5	Execute	false	agent is discarded
u6	Read	true	agent executes tasks
u6	Write	false	agent is discarded
u6	Execute	true	agent executes tasks
u-unknown	Read	false	agent is discarded
u-unknown	Write	false	agent is discarded
u-unknown	Execute	false	agent is discarded

This section will compare the results of the test cases to the expected results introduced in section 6.2.2, and conclude the comparison in table 6.8 and 6.9.

6.3 Limitations on the Tests

Because the VADOR system has not setup the SSH port forwarding to manage communication between different domains, the Active Agent can only migrate between the VADOR servers reside in the same domain. Therefor, the validation and tests are also limited to the VADOR servers that are in the same domain.

Only Executive server and Librarian server have been selected for the tests purpose, because they represent the most specific VADOR servers that may send and receive Active Agent for task executions. However, the tests are only limited to the specific security issue, which is to control access to system resources, so that server and agent protection are not included in the tests.

Since the SSH protection mechanisms are managed by operating systems, it was not tested within the Security Manager pattern.

6.4 Possible Applicability to Other Systems

As mentioned in section 6.1, the main objective of the validation and tests is to insure that it is able to control access to VADOR system's resources.

By introduction of relevance and extension points, this section discusses the possibility of applying the Security Manager pattern to other systems, specifically, distributed systems.

6.4.1 Relevance of Other Systems

As introduced in Chapter 1, VADOR is built as a mobile agent environment that meets the requirements of a MDO software framework for aeronautical applications. Specifically, it is a distributed system that uses the Internet as communication media, and allows many users work in the same environment, on the same system resources at the same time. However, threats can affect the agent during its migration, they can also affect a specific user data file when there is a command that needs to be executed (See section 3.1 for detailed description).

Same as the VADOR system, threats can also affect agent and user data in other agent based distributed systems, in which the Internet is used as communication media. The

solution of control access to system resources in the VADOR systems should be able to be applied in other distributed systems.

6.4.2 Extension Points to Other Systems

The Security Manager pattern provides several extension points, they can be easily extended by other distributed systems to build security architectures for controlling access to the systems' resources. These extension points are represented in the Security Interface module, in which the following parts are include:

1. Extension points for Agent Signature
2. Extension points for Security Server
3. Extension points for Security Attributes Descriptor
4. Extension points for Security Manager

6.4.2.1 Extension points for Agent Signature

A new concrete VadorSigner class can extend the AbstractVadorSigner provided by the *Interface of Agent Signature*, so that it can apply different cryptographic algorithms to sign an agent before sending it.

6.4.2.2 Extension points for Security Server

Extend the abstract VadorProxy and ServerThread provided by the *Interface of Security Server*, other distributed systems can easily implement Security Servers for agent authentication and verification. A new concrete SecurityServerProxy can extend the VadorProxy

to act as a local representative of the Security Server. A new concrete SecurityServer-Thread that is launched by the SecurityServer is an extension of the ServerThread.

6.4.2.3 Extension points for Security Attributes Descriptor

Other distributed systems can extend the abstract VadorSecurityAttributesDescriptor defined in the *Interface of Security Attributes Descriptor*, so that they can create new concrete security attributes descriptor classes for keeping track of security attributes.

6.4.2.4 Extension points for Security Manager

Three extension points are defined by the *Interface of Security Manager*: the AbstractVadorSecurityManager, the AbstractSecurityVisitor, and the AbstractPolicyApplicator. Other distributed systems can extend them to create concrete SecurityManager, SecurityVisitor, and PolicyApplicator classes for all servers that act on behalf of the systems, so that they can perform agent authorization process for the systems.

6.5 Summary

Chapter 5 statically defined and validated the *Security Manager pattern*, that is defined to build VADOR security architecture, so that it can solve security problems and prevent security defects related to access control mechanisms in VADOR framework.

As the *Objectives* that is introduced in the beginning, this chapter dynamically validates the *Security Manager* pattern by demonstrating and testing it in the VADOR system. The management of these validation and tests including a plan, procedures and expected results, test cases and results, and finally, a comparison of the expected results and the test

results show that the *Security Manager pattern* can be implemented to formalize security mechanisms, and can help to build security architecture in VADOR, so that security defects and problems related to control accesses to system resources can be prevented and solved.

TABLE 6.8 Comparison of Expected and Test Cases Results - 1

Expected Result	**Test Cases Result**
The VADOR administrator is able to generate Key pairs and save them into specific Keystores using the Java Keytool program	Sets of Key pairs have been generated and saved in specific Keystores by the administrator using Keytool program
The VADOR administrator is able to export and import the Key Certificates	Key Certificates have been exported from the VADOR servers and imported into the VADOR Security Server by the administrator using the Keytool program
An agent sender is able to sign the agent using an existing Private Key by providing correct Key and Keystore information, and then the signed agent can be sent. Otherwise, if the sender provides incorrect information of a Key or Keystore, the agent cannot be signed and sent. A *sendAgent_failed* message will be returned to this sender	When a sender provides correct Key and Keystore information of an existing Private Key, the agent could be signed and then sent by this sender. When the Key or Keystore information is incorrect, the agent sender could not sign and send the agent, but received a *sendAgent_failed* message
The Security Server is able to use correct Key Certificates to verify signed agents for the other VADOR servers. If provided certificate information is correct and it can de-codify a signed agent using the certificate, it will return a *authentication_succeed* message to the VADOR server, otherwise, it will return a *authentication_failed* message to the VADOR server	When provide a correct Key Certificate to the Security Server, and the signed agent status (for example, sender information or requests) has not been changed, the Security Server returns *authentication_succeed* to the VADOR server. When a incorrect certificate information provided, or the certificate is correct, but the signed agent status has been changed, so that the Security Server cannot de-codify the signed agent, then a *authentication_failed* message is returned to the VADOR server that required the verification and authentication

TABLE 6.9 Comparison of Expected and Test Cases Results - 2

Expected Result	Test Cases Result
If a VADOR server receives the *authentication_succeed* message from the Security Server, it will continue the agent authorization processes. Otherwise, if it receives message *authentication_failed*, it will discard the agent and forward *authorization_failed* message to the agent sender	When a VADOR server receives a respond message from the Security Server, if the message is *authentication_succeed*, it continues on the agent authorization processes, if the message is *authentication_failed*, it discards the signed agent, and forwards the message to the agent's sender
If information of an agent sender cannot be found in the VadorPolicy table, or the agent's requests do not match its sender's privileges, the agent cannot be authorized to execute tasks on the VADOR server. Otherwise, if the sender's information can be found in the VadroPolicy table in database, and the sender's privileges match the agent's requests, the agent will be authorized to execute tasks on the VADOR server	When an agent sender's information cannot be found in the VadorPolicy database table, the agent cannot be authorized. When the agent sender's information is found in the table, if the agent's requests do not match its sender's privileges, the agent cannot be authorized. When the agent sender's information is found in the VadorPolicy table, and the agent's requests match its sender's privileges defined in the VadorPolicy table, the agent can be authorized
If an agent has been verified by the Security Server and authorized by a VADOR server, it will be allowed to operate and execute tasks on that VADOR server. Otherwise, if an agent has been verified by the Security Server, but not authorized by a VADOR server, it will not be allowed to operate and execute tasks on that VADOR server, but will be discarded by the server, and then its sender will receive a *authorization_failed* message from the server	When an agent has been verified by the Security Server, and also has been authorized by a VADOR server, it is allowed by the VADOR server to execute tasks on it. When an agent has been verified by the Security Server, but cannot be authorized by a VADOR server, it cannot execute tasks on that VADOR server, and is discarded by that server, then the server sends a *authorization_failed* message to the agent sender

CONCLUSION

Multi-user, multi-threaded distributed applications allow users working at different locations and sharing the same resources, to work more efficiently and at a lesser cost. The drawback is that systems' security defects may be exploited by malicious users, and cause security problems to the systems, such as secret information being exposed to attackers.

As introduced in Chapter 1, VADOR is an instance of a distributed application, in which resources can be shared by many users. In this context, VADOR users expect to be able to benefit from the many advantages in functionalities provided by the system, and take advantages of the many possibilities offered by a multi-user environment. At the same time, VADOR has to face the challenge of dealing with security defects and problems related to control accesses to its resources by different users.

Based on the characteristics of VADOR, this thesis focuses on VADOR security architectural design, so that the security defects that might cause security problems related to access control can be prevented.

In order to prevent security defects and solve security problems at an early stage of the development, Chapter 2 reviewed the available literature on security design patterns and defect classifications, specified the defect categories defined by the Secure design Patterns (SecurP) Project (Guibault *et al.* (2004)). This project introduced the defects that may exist in distributed applications and could be prevented using secure and reliable design patterns. This chapter then summarized the security design patterns into groups, so that they could be used to prevent the defects introduced and solve security problems.

As introduced and analyzed in Chapter 3, threats can affect an agent during its migration, and they can also affect user data files during command execution. In addition, VADOR has a fundamental security problem, that is the VADORADM owns the files

inside VADOR users' directories, so that a user's data files could be remotely manipulated by other users. Although this chapter has reviewed several security defects in the VADOR framework, the *Improper Multiple Access Control Points* defect may be exploited and lead to risks to the agent or user data files, and is considered to constitute the most fundamental security problem.

To prevent the *Improper Multiple Access Control Points* defect and protect system resources, the VADOR security model has been introduced in Chapter 4. Based on the Java 2 SDK security model, the VADOR security model consists of the following three levels protection to the VADOR system:

1. 1st level protection: Security Server that is based on JVM, and applies External Security Policy defined by the administrator using Keytool provided by the Java.
2. 2nd level protection: Security Managers that act on behalf of the VADOR Servers, and apply VADOR Security Policy defined by the administrator and stored in the VADOR database.
3. 3rd level protection: Operating system that enforces the protection mechanisms using SSH.

Based on the studied security defect and mechanisms to protect resources in the VADOR system, which is an instance of a distributed applications, the *Security Manager pattern* is defined to help security architects in developing security access control mechanisms in VADOR framework, so that the security defects and problems related to control accesses to the system resources can be prevented and solved.

As introduced in Chapter 5, the Security Manager pattern consists of the following six modules. Each of the modules is responsible of specific functionalities related to the prevention of security defects and aimed at solving specific aspects of the security problems. These modules are required by the Security Manager pattern:

- The Security Interface Module provides a programming interfaces to the other modules, so that the Security Manager pattern can be easily implemented by the other applications.
- The Agent Authentication Module manages the process of signing an agent before it is sent. It also deals with the verification and authentication of the signed agent after it was received by a Vador server.
- The Agent Signature Module acts on behalf of the Agent Authentication Module, and provides agent signature services to an agent sender.
- The Security Server Module acts on behalf of the Agent Authentication Module, and is responsible of signed agent verification and authentication that a Vador server requires.
- The Security Attributes Descriptor Module keeps track of Security Contexts that are needed for verification, authentication, and authorization processes.
- The Security Manager Module collaborates with all the other modules to provide services to specific VADOR servers, including the Security Server.

The following existing design patterns or security patterns have been implemented in the Security Manager pattern's modules:

- Cryptographic Meta pattern and Sender Authentication pattern are implemented by the Agent Authentication Module.
- Cryptographic Meta pattern, Sender Authentication pattern and Template Method pattern are implemented by the Agent Signature Module.
- Proxy pattern, Single Access Point pattern, and Policy pattern are implemented by the Security Server Module.

- Subject Descriptor pattern and Session pattern are implemented by the Security Attributes Descriptor Module.
- Policy pattern, Visitor pattern, and Strategy pattern are implemented by the Security Manager Module.
- Protected System pattern and Partitioned Application pattern are also implemented by the Security Manager pattern to describe the structure.

Three levels of protection have been defined using the Security Manager pattern and the higher two levels have been tested in the VADOR system, these levels are:

- VADOR Security Server applies External Security Policy to verify and authenticate agents.
- VADOR Servers apply VADOR Security Policy to authorize verified agents.
- SSH server apply operating system security policy to control accesses to data files by VADOR servers' authorized agents. Since the SSH protection mechanisms are managed by operating systems, and have been implemented in the VADOR prototype, they were not tested within the Security Manager pattern.

The currently defined Security Manager pattern focuses on the protection of system resources from agent attacks, these protection mechanisms cannot protect agents once they are sent out for tasks. However, several future works are needed to extend the pattern to provide full protection to both on site and transmitted information:

- Provide secure communication channels to systems, so that agents won't be interrupted, intercepted, or modified during their traveling.
- Integrate the External Security Policy initialization mechanisms into the system, so that it can reduce manual works and secure Key management.

- Integrate SSH protection mechanisms into the system, so that instead of communicating with the operating system, the systems can control accesses to data files directly, reduce risks of communications, and solve the potential problems of system integration and multi-threaded processes introduced by the use of SSH (See section 3.1.2.5 for detailed description).

REFERENCES

ALEXANDER, C., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I., AND ANGEL, S. (1977). A pattern language. Technical report, Oxford University Press New York.

BISBEY II, R. AND HOLLINGWORTH, D. (1978). Protection analysis project final report. *ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute.*

BISHOP, M. (1995). A taxonomy of unix system and network vulnerabilities. *http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-10.pdf.*

BRAGA, A. M., RUBIRA, C. M. F. AND DAHAB, R. (1998). Tropyc: A pattern language for cryptographic software. *http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P25.pdf.*

CHEN, B. (2004). *A Pattern-Based Framework Architecture for Distributed Engineering Applications.* Master thesis, École Polytechnique de Montréal.

COOPER, J. W. (1998). *The Design patterns, Java Companion.* Addison-Wesley.

CSRC (1999). Common criteria for information technology security evaluation. Technical report, National Institute of Standards and Technology.

FERNANDEZ, E. B. (1999). The authenticator pattern. *http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/Fernandez4/Authenticator3.PDF.*

FERNANDEZ, E. B., HAYS, V. AND LOUTREL, M. (2001). The object filter and access control framework. *http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Fernandez3/Fernandez3.pdf*.

FERNANDEZ, E. B. AND PAN, R. (2001). A pattern language for security models. *http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/PLoP2001*.

GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J. (1994). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GUIBAULT, F., CHEN, B., LAFLAMME, S., VALLET, M.-G. AND WANG, Y. (2004). Secure design patterns: State-of-the-art, software defects and patterns specification. Technical report, École Polytechnique de Montréal.

IBM (2002). Orthogonal defect classification for design and code. *http://www.research.ibm.com/softeng/ODC/ODC.HTM*.

KRASNER, G. AND S.T., P. (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80.

LANDWEHR, C. E., BULL, A. R., MCDERMOTT, J. P. AND C, W. S. (1994). A taxonomy of computer program security flaws, with examples. *http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf*.

LIBES, D. (1995). *Exploring Expect, A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly.

MESZAROS, G. AND DOBLE, J. (1996). A pattern language for pattern writing. *http://hillside.net/patterns/Writing/patterns.html*.

OPENGROUP, T. (2002). Guide to security patterns - draft 1. *http://www.opengroup.org/security/gsp.htm*.

OWASP (2003). The ten most critical web application security vulnerabilities. *http://www.owasp.org/documentation/topten*.

ROMANOSKY, S. (2001). Security design patterns. *http://www.cgisecurity.com/lib/securityDesignPatterns.pdf*.

SALAS, A. AND TOWNSEND, J. (1998). Framework requirements for mdo application development. *AIAA Paper 98-4740*.

SCHUMACHER, M. AND ROEDIG, U. (2001). Security engineering with patterns. *http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/PLoP2001/*.

SOBIESZCZANSK-SOBIESKI, J. AND HAFTKA, R. (1997). Multidisciplinary aerospace design and optimization. *Structural Optimization*. Survay of recent developments.

SOFT-SOLUTIONS-INTERNATIONAL (2002). Software validation ethos. *http://www.ssi-ltd.com/services/software-validation.asp*.

SSH (2004). Ssh web tutorial.

TRÉPANIER, J.-Y. (1999). Mdo - multidisciplinary design optimization. *CERCA*. An overview of the field of Multidisciplinary Design Optimization(MDO).

YODER, J. AND BARCALOW, J. (1998). Architectural patterns for enabling application security. *http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Proceedings/yoder.pdf*.

APPENDIX I

Security Defects Classes in Distributed Applications

I.1 Landwher's Classification

- ***Categories:***

  1. *By Genesis:* The genesis provides basis for understanding how a security flaw finds its way into a program, so that they can be prevented, detected, or corrected by different strategies. It includes Intentional and Inadvertent flaws. The Inadvertent flaws were categorized into the following subclasses:
     - Validation Error(Incomplete/Inconsistent)
     - Domain Error (Including Object Re-use, Residuals, and Exposed Representation Errors
     - Serialization/aliasing (Including TOCTTOU Errors)
     - Identification/Authentication Inadequate
     - Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)
     - Other Exploitable Logic Error
  2. *By Time of Introduction:* The time of introduction of a security flaw is the point of a software life cycle where the flaw was introduced, including *During Development*, *During Maintenance*, and *During Operation*. It can help developers understanding the weakness in the software development process and focus their efforts on the flaws corresponding to specified processes.
  3. *By Location:* The location is that part of the software (operating system or application) or hardware where the error lies.

- ***Limitations***

  1. This taxonomy is limited in focusing on the flaws that occur in operating system rather than in other distributed application programs.
  2. Provides approaches to evaluate problems in built systems rather than approaches that could prevent the problems during design in a development process.

I.2 Bishop's Classification

- ***Categories***

  1. *Improper protection:*
     - Improper Choice of Initial Protection Domain: The vulnerabilities in this class involve incorrectly set permissions when the system starts; these are configuration errors.
     - Improper Isolation of Implementation Detail: By their nature, these flaws arise because of multiple paths to a single object.
     - Improper Change: Flaws of this category occur when data that is meant to be consistent is not consistent; essentially, one misplaces trust in the integrity of the data.
     - Improper Naming: At the user level, handling improper naming simply means detecting objects in the user's protection domain with the same name.
     - Improper Deallocation or Deletion: When an object is improperly deallocated or deleted, the object containing the data is released but the data is not erased.

2. *Improper Validation:* It means that insufficient checks are made upon data, and the failure to do so creates a security problem.
3. *Improper Synchronization:*
   - Improper Indivisibility: This category involves operations which need to be atomic but are interruptible.
   - Improper Sequencing: It refers to the incorrect ordering of operations.
4. *Improper Choice of Operand or Operation:* This type of flaw can arise in one of two ways: an abstraction operation may be chosen incorrectly, or the implementation may be poorly (incorrectly) chosen.

- ***Limitations***

1. The taxonomy focuses on classifying security flaws in the UNIX operating systems, the security flaws in other software applications are not classified.
2. It presents rough detection and prevention mechanisms, but formal approaches for flaw prevention are not well developed enough.

I.3 The Top Ten Web Application Security Vulnerabilities

- ***Categories***

1. *Unvalidated Parameters:* Information from web requests is not validated before being used by a web application.
2. *Broken Access Control:* Restrictions on what authenticated users are allowed to do are not properly enforced.
3. *Broken Account and Session Management:* Account credentials and session tokens are not properly protected.

4. *Cross-Site Scripting (XSS) Flaws :* The web application can be used as a mechanism to transport an attack to an end user's browser.
5. *Buffer Overflows:* Web application components in some languages that do not properly validate input can be crashed and, in some cases, used to take control of a process.
6. *Command Injection Flaws:* Web applications pass parameters when they access external systems or the local operating system. If an attacker can embed malicious commands in these parameters, the external system may execute those commands on behalf of the web application.
7. *Error Handling Problems:* Error conditions that occur during normal operation are not handled properly.
8. *Insecure Use of Cryptography:* Web applications frequently use cryptographic functions to protect information and credentials. These functions and the code to integrate them have proved difficult to code properly, frequently resulting in weak protection.
9. *Remote Administration Flaws:* Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application.
10. *Web and Application Server Misconfiguration:* Having a strong server configuration standard is critical to a secure web application

- ***Limitations***

This list represents only a set of security vulnerabilities that occur in distributed web applications, it provides information on approaches to protect web applications from each flaw that is listed, but these vulnerabilities are very general.

I.4 Security Defects Related to Design

- ***Categories***

1. *Security Structural Defects:* This category represents the defects related to the system structure, and could be exploited to lead risks.
   - Untrusted Interface: An interface provides a black box picture of each system module, it constrains access to a protected resources of a system, limits the operation that can be performed, or limits the user's view to a subset of the data. An interface is untrusted if it cannot provide the above services to protect the module that it represents.
     * *Problems*: A system may fail in access control policy enforcement, illegal operations, data or information disclosure.
   - Monolithic Application: A monolithic application is a complex system in which individual system modules are not specified, so that security concerns need to be verified for the entire application.
     * *Problems*: It is difficult to restrict dangerous privilege and verify the security concerns.
2. *Security Functional Defects:* Most of the security defects were categorized into this class, because they could be exploited by threats and directly represent risks to the system's functionalities, and cause damage to the system.
   - Improper Security Auditing: Improperly recognizing, recording, storing, and analyzing information related to security relevant activities.
     * *Problems*: The resulting audit records cannot be examined to keep track of the security relevant activities correctly, so that the potential violation of the system security functions cannot promptly be detected.

- Improper Communication Protection: The identity of a party participating in a data exchange cannot be assured, including incapability to request and generate evidence of the origin/recipient of information.
  * *Problems*: The originator denies having sent the information, the recipient denies having received the information.
- Insecure Use of Cryptography: Including improper design in cryptographic key management or cryptographic operation functions.
  * *Problems*: Weak protection on user data or system security functions such as authentication/identification, non-repudiation.
- Improper User Data Protection: Including improper security policy specification and improper security policy enforcement by system security functions.
  * *Problems*: Violation on user data, including interception, interruption, modification, and deletion.
- Inadequate Authentication/Identification: Inadequately establish and verify a claimed user identity, so that users are associated with the improper security attributes (e.g. identity, groups, roles, security or integrity levels).
  * *Problems*: Security functions cannot enforce security policies properly, it may cause deny of services or data disclosure.
- Improper Security Management: Including improper management of system security function data (i.e., banners), security attributes (i.e., Access Control Lists), security functions (i.e., selection of functions), and improper definition of security roles.
  * *Problems*: System and user data disclosure, unauthorized access to confidential information because of a user may gaining access to data which he/she does not have right to access.

- Improper Protection of Privacy: Including improper protection of Anonymity, Pseudonymity, Unlinkability, and Unobservability.
  - *Problems*: A user's identity is discovered and misused by other users.
- Improper Protection of the System Security Functions: Improper integrity and management of the mechanisms that provide the system security functions and to the integrity of system security function data, which are the administrative databases that guide the enforcement of the security policy.
  - *Problems*: Violation on the system security policy, data disclosure.
- Improper Utilization of Resource: System security functions do not properly support the availability of required resources such as processing capability and/or storage capacity. Including improper support on Fault Tolerance, Priority of Service, and Resource Allocation.
  - *Problems*: Unavailability of capabilities caused by failure of the system, resources cannot be allocated to the more important or time-critical tasks and could be monopolized by lower priority tasks, the system does not provide limits on the use of available resources, therefore it cannot prevent users from monopolizing the resources.
- Improper System Access Control: Improper controlled user's session establishment. Including improper limitation on scope of selectable attributes and multiple concurrent sessions, improper session locking and establishment, and improper access banners and history.
  - *Problems*: The system access control may be broken by attackers' breaking access attempts.
- Untrusted Path/Channels: The communication path between users and the system security functions, and the communication channel between the system security functions and other IT products or systems are un-

trusted.

* *Problems*: The system cannot provide assurance that the communications between the users and the security functions, or/and between the security functions and the other systems are correct and secure, so that the Untrusted Channel may cause repudiation of participants, and the Untrusted Path may be modified or disclosed to untrusted applications.

- ***Limitations***

This defect classification represents only the security defects that occur in distributed applications and that concern the systems' security structure or functionality design. It doesn't specify defects that may exist in systems' physical assets or other development processes (ex., in hardware or in coding).

APPENDIX II

Security Design Patterns

II.1 Yoder and Barcalow

1. **Single Access Point:** Providing a security module and a way to log into the system.
   - **problems:** A security model is difficult to validate when it has multiple "front doors", "back doors", and "side doors" for entering the application.
2. **Check Point:** Organizing security checks and their repercussions.
   - **problems:** An application needs to be secure from break-in attempts, and appropriate actions should be taken when such attempts occur. Different organizations have different security policies and there needs to be a way to incorporate these policies into the system independently from the design of the application.
3. **Roles:** Organizing users with similar security privileges.
   - **problems:** Users have different security profiles, and some profiles are similar. If the user base is large enough or the security profiles are complex enough, then managing user-privilege relationships can become difficult.
4. **Session:** Localizing global information in a multi-user environment.
   - **problems:** Many objects need access to shared values, but the values are not unique throughout the system.
5. **Full View With Errors:** Provide a full view to users, showing exceptions when needed.

- **problems:** Users should not be allowed to perform illegal operations.

6. **Limited View:** Allowing users to only see what they have access to.

   - **problems:** Users should not be allowed to perform illegal operations.

7. **Secure Access Layer:** Integrating application security with low level security.

   - **problems:** Application security will be insecure if it is not properly integrated with the security of the external systems it uses.

II.2 Eduardo B. Fernandez

1. **Object Filter and Access Control Framework:** Fernandez *et al.* (2001) defined an object filter pattern and access control framework for distributed applications. This framework combines the functions of authentication, access control, and object filtering to constrain a client to access objects in specified ways defined by the client rights.

   - **problems:** In distributed systems, data or services requested by clients may need to be furnished to control the type of data provided. Also, these services may need to be restricted to be used by only some users in specific ways.

2. **The Authenticator Pattern:** Fernandez (1999) introduced the Authenticator Pattern to describe a general mechanism for providing identification and authentication to a server from a client. It has the added feature of allowing protocol negotiation to take place using the same procedures. The pattern operates by offering an authentication negotiation object which then provides the protected object only after authentication is successful.

   - **problems:** How to protect distributed objects while there are a variety of accesses to a distributed system.

Fernandez and Pan (2001) discussed three design patterns that correspond to the most common models for security in a new built system: *Authorization*, *Role-Based Access Control*, and *Multilevel Security*. These patterns can be applied at all levels of the system.

3. **Authorization:** Represents the elements of an authorization rule as classes and associations to control active entities getting access to resources in any computational environment.

    - **problems:** How to describe allowable types of accesses (authentications) by active computational entities (subjects) to passive resources (protection objects).

4. **Role-Based Access Control:** Represents the elements of a role as classes and associations to assign rights to users according to their roles in an institution.

    - **problems:** How to assign rights to users according to their roles in an institution.

5. **Multilevel Security:** Assigns classifications or clearances to users and data, so that the users can access documents based on their clearances and the sensitivity levels of data and documents.

    - **problems:** How to determine access in an environment with security classifications.

II.3 Sasha Romanosky

1. **Authoritative Source of Data:** Recognizing the correct source of data.

    - **problems:** If an application or user blindly accepts data from any source then it is at risk of processing potentially outdated or fraudulent data. Therefore, an

application needs to recognize which, of possibly many sources, is the single authority for data. Are you assured the data you are using is the cleanest and most accurate? In other words, is the data coming from a legitimate source or from an unknown party?

2. **Risk Assessment and Management:** Understanding the relative value of information and protecting it accordingly.

    - **problems:** Whenever information needs to be transferred, stored or manipulated, the privacy and integrity of that data needs to be reasonably assured. Hardware and software require protection from misconfiguration, neglect and attack. Under-protection of any of these could drive a company to bankruptcy (or legal battle) and overprotection is a waste of resources.

3. **3rd Party Communication:** Understanding the risks of third party relationships.

    - **problems:** Two companies in a business relationship may trust each other, but to what degree? Specifically, when two businesses exchange information, users and/or applications will require access to privileged resources. How can access be granted while at the same time protecting both organizations? Additionally, how can this be managed in such a way that is neither overly complex nor dangerously simplistic?

4. **The Security Provider:** Leveraging the power of a common security service across multiple applications.

    - **problems:** When disparate applications seek to provide their own security services, privacy, synchronization and management of data becomes unnecessarily complex. Moreover, applications may not provide the security features or strength required, risking the overall integrity of the data. These applications may be communicating in securely or they maybe using weak or inappropriately vulnerable methods. Without a common security infrastructure,

the management becomes unnecessarily difficult and risks the security of the entire environment.

5. **White Hats, Hack Thyself:** Testing your own security by trying to defeat it.

   - **problems:** How can you be assured of the true security of your system without real world testing?

6. **Fail Securely:** Designing systems to fail in a secure manner.

   - **problems:** In the event of a failure or misconfiguration of an application or network device, would the result be a more, or less secure environment? that is, would the consequence result in a user performing a given operation unprotected; or a device passing unauthorized information?

7. **Low Hanging Fruit:** Taking care of the "quick wins".

   - **problems:** Good security is a cycle that requires intelligent planning, careful implementation and meaningful testing. Unfortunately, administrators, developers and managers may not have the time or opportunity to properly complete this cycle. Therefore, taking advantage of the quick wins maybe the only opportunity to establish reasonable security.

II.4 Cryptographic Meta-pattern

1. **Information Secrecy:** Keep the secrecy of information.

   - **problems:** How a message can be sent from a sender to a receiver in such a way that a third person cannot possibly read its content?

2. **Message Integrity:** Avoid corruption of a message.

- **problems:** How can a message receiver determines if the message was modified or replaced after being sent or before its arrival to him?

3. **Message Authentication:** Authenticate the origin of a message.

   - **problems:** How can genuine messages be distinguished from spurious ones?

4. **Sender Authentication:** Avoid refusal of a message.

   - **problems:** How to guarantee that messages have genuine and authentic senders, in such a way that the sender cannot repudiate a message that a receiver believes was sent by him?

5. **Secrecy with Authentication:** Prove the authenticity of a secret.

   - **problems:** How can a sender authenticate an encrypted message without loss of secrecy?

6. **Secrecy with Signature:** Prove the authorship of a secret.

   - **problems:** How can a receiver prove authorship of an encrypted message without loss of secrecy in such a way that its integrity and origin authentication is also implicitly granted?

7. **Secrecy with Integrity:** Keep the integrity of a secret.

   - **problems:** How to preserve the integrity of an encrypted message without loss of secrecy?

8. **Signature with Appendix:** Separate message from signature.

   - **problems:** How to reduce the storage space required for a message and its signature while increasing the performance of the digital signature protocol?

9. **Secrecy with Signature with Appendix:** Separate secret from signature.

- **problems:** How to reduce the memory necessary to store a message and its signature, while increasing system performance, without loss of secrecy?

10. **Cryptographic Meta-pattern:** Define a generic software architecture to cryptography.

    - **problems:** How to design a flexible object-oriented micro-architecture for a cryptographic design pattern in order to increase object reuse?

II.5 Open Group

1. **Protected System:** Structure a system so that all access by clients to resources is mediated by a guard which enforces a security policy.

    - **problems:** How to protect system resources against unauthorized access.

2. **Policy Enforcement Point:** Isolate policy enforcement to a discrete component of an information system; ensure that policy enforcement activities are performed in the proper sequence.

    - **problems:** How to invoke policy enforcement functions in the correct sequence in a system that needs to enforce policy, while access is attempted to a resource which is subject to the policy.

3. **Subject Descriptor:** Provide access to security-relevant attributes of an entity on whose behalf operations are to be performed.

    - **problems:** How to access security-relevant attributes of an entity.

1. **Secure Communication:** Ensure that mutual security policy objectives are met when there is a need for two parties to communicate in the presence of threats.

- **problems:** How to secure a communication channel between two protected systems with security policy objectives applicable, so that threats such as eavesdropping, impersonation, and tampering can be prevented.

2. **Secure Association:** Establish and maintain a security relationship, between two entities that wish to communicate securely, in line with mutual security policy objectives, across a communication link that is subject to a well-known set of communication related threats.

    - **problems:** How to manage the life cycle of state containing the details of a security relationship between two entities, where the entities need to engage in some joint activity.

3. **Security Context:** Provide a container for, and mediate access to, security attributes and data relating to a particular process, operation or action.

    - **problems:** How to manage and access to contextual properties which may influence the behavior of security related functions such as access control, auditing, message protection and so on.

1. **Recoverable Component:** Structure a component so that its state can be recovered and restored in case the component fails.

    - **problems:** How to restore or recover state of a component, while the component fails.

2. **Checkpointed System:** Structure a system which can be "rolled back" to a known valid state.

    - **problems:** How to return the system to a previous state that is known to be valid, while component failures, errors in processing, data entry errors, or operator errors cause the system state to become corrupt, erroneous, or otherwise defective.

3. **Cold Standby:** Structure a system so that the service provided by one component can be resumed from a different component.

    - **problems:** How to implement a recovery mechanism that will suffice for all forms of fault or failure, up to and including the complete destruction of a component (as by fire or other environmental failure).

4. **Comparator-Checked Fault-Tolerant System:** Structure a system so that an independent failure of one component will be detected.

    - **problems:** How to detect component faults quickly, or to detect component faults at a specific point during processing, to prevent component faults from causing system failures.

5. **Journaled Component:** Record changes to a component's state so that the state can be restored using incremental updates to a previous version of the state if necessary.

    - **problems:** How to protect a system state from failures which corrupt state information.

6. **Hot Standby:** Structure a system which permits state updates to originate from multiple components, preserves the state of the overall system and of each transaction in the face of failures, and guards against loss of integrity due to incomplete application of transactions or changes.

    - **problems:** How to protect multi-component transactional systems which are often susceptible to state corruption because of failure of communication links, communication protocols, storage media, or other system elements.

7. **External Storage:** Structure a system which isolates processing from state management, so that system state is kept in a single high-integrity repository regardless of the number of processing elements or points of presence included in the system.

- **problems:** How to assure availability of transaction services in the face of failure of communication links, communication protocols, or other system elements.

8. **Replicated System:** Structure a system which allows provision of service from multiple points of presence, and recovery in case of failure of one or more components or links.

    - **problems:** How to assure availability of transaction services in the face of failure of communication links, communication protocols, or other system elements.

9. **Error Detection/Correction:** Add redundancy to data to facilitate later detection of and recovery from errors.

    - **problems:** How to deal with errors that happen to data that resides on storage media or in transit across communication links.

ÉCOLE POLYTECHNIQUE DE MONTRÉAL
3 9334 00314294 8